OMNI PILOT®
SOFTWARE

Language Guide

Lasso 8

Trademarks

Lasso, Lasso Professional, Lasso Studio, Lasso Dynamic Markup Language, LDML, Lasso Service, Lasso Connector, Lasso Web Data Engine, and OmniPilot are trademarks of OmniPilot Software, Inc. All other products mentioned may be trademarks of their respective holders. See *Appendix C: Copyright Notices* in the Lasso Professional 8 Setup Guide for additional details.

Third Party Links

This guide may contain links to third-party Web sites that are not under the control of OmniPilot. OmniPilot is not responsible for the content of any linked site. If you access a third-party Web site mentioned in this guide, then you do so at your own risk. OmniPilot provides these links only as a convenience, and the inclusion of the links does not imply that OmniPilot endorses or accepts any responsibility for the content of those third-party sites.

Copyright

Copyright © 2005 OmniPilot Software, Inc. This manual may not be copied, photocopied, reproduced, translated or converted to any electronic or machine-readable form in whole or in part without prior written approval of OmniPilot Software, Inc. See *Appendix C: Copyright Notices* for additional details.

Third Edition: March 28, 2005

OmniPilot Software, Inc.
1815 Griffin Road
Dania Beach, Florida 33004
U.S.A.

Telephone: (954) 874-3100
Email: info@omnipilot.com
Web Site: http://www.omnipilot.com

Contents

Section I
Lasso Overview 27

Chapter 1
Introduction 29
Lasso 8 Documentation 29
Lasso 8 Language Guide 30
Documentation Conventions. 31

Chapter 2
Web Application Fundamentals. 33
Web Browser Overview 33
Web Server Overview. 40
HTML Forms and URL Parameters 41
Web Application Servers 43
Web Application Server Languages 44
Error Reporting 44

Chapter 3
Format Files 47
Introduction. 48
Storage Types 48

Naming Format Files 49
Character Encoding 50
Editing Format Files 50
Functional Types 51
Action Methods 52
Table 1: Action Methods 52
Securing Format Files 56
Output Formats 57
File Management 58
Specifying Paths 60
Format File Execution Time Limit 64

Chapter 4
LDML 8 Syntax 65

Overview 66
Colon Syntax 68
Table 1: Colon Syntax Delimiters 68
Parentheses Syntax 69
Table 2: Parentheses Syntax Delimiters 69
Square Brackets 70
Table 3: Square Bracket Delimiters 70
LassoScript 73
Table 4: LassoScript Delimiters 74
HTML Form Inputs 77
URLs 77
Compound Expressions 78
Table 5: Compound Expression Delimiters 78

Chapter 5
LDML 8 Tag Language 81

Introduction 82
Tag Types 82
Table 1: LDML 8 Tag Types 82
Tag Categories and Naming 90
Table 2: LDML 8 Tag Categories 90
Table 3: LDML 8 Synonyms 93
Table 4: LDML 8 Abbreviations 93
Parameter Types 94
Table 5: Parameter Types 94
Encoding 95
Table 6: Encoding Keywords 96
Data Types 97

Table 7: Primary LDML 8 Data Types 97
Expressions and Symbols 102
Table 8: Types of LDML 8 Expressions 103
Table 9: Member Tag Symbol 104
Table 10: Retarget Symbol 104
Table 11: String Expression Symbols 106
Table 12: Math Expression Symbols 107
Table 13: Conditional Expression Symbols 109
Table 14: Logical Expression Symbols 110
Table 15: Logical Expression Symbols 110
Delimiters 112
Table 16: LDML 8 Delimiters 112
Table 17: HTML/HTTP Delimiters 113
Illegal Characters 113
Table 18: Illegal Characters 114

Chapter 6
LDML 8 Reference 115

Overview 115
Figure 1: LDML 8 Reference 116
Search 118
Figure 2: Basic Search Page 118
Figure 3: Advanced Search Page 119
Figure 4: Examples Search Page 121
Browse 122
Figure 5: Category Tags Page 122
Figure 6: Legacy Tags Page 123
Detail 123
Figure 7: Tag Detail Page 124
List 126
Figure 8: Preferred Tags Page 126

Section II
Database Interaction 129

Chapter 7
Database Interaction Fundamentals . . 131

Inline Database Actions 132
Table 1: Inline Tag 132
Table 2: Inline Database Action Parameters 134
Table 3: Response Parameters 140

Action Parameters 143
Table 4: Action Parameter Tags 144
Table 5: [Action_Params] Array Schema 146
Results 148
Table 6: Results Tags 148
Showing Database Schema 150
Table 7: -Show Parameter 150
Table 8: -Show Action Requirements 150
Table 9: Schema Tags 151
Table 10: [Field_Name] Parameters 152
Table 11: [Required_Field] Parameters 153
SQL Statements 154
Table 12: SQL Inline Parameters 154
Table 13: -SQL Helper Tags 155
SQL Transactions 158

Chapter 8
Searching and Displaying Data 161

Overview 161
Table 1: Command Tags 162
Table 2: Security Command Tags 165
Searching Records 165
Table 3: -Search Action Requirements 166
Table 4: Operator Command Tags 168
Table 5: Field Operators 169
Table 6: Results Command Tags 173
Finding All Records 176
Table 7: -FindAll Action Requirements 177
Finding Random Records 178
Table 8: -Random Action Requirements 178
Displaying Data 179
Table 9: Field Display Tags 180
Linking to Data 182
Table 10: Link Tags 183
Table 11: Link Tag Parameters 185
Table 12: Link URL Tags 186
Table 13: Link Container Tags 187
Table 14: Link Parameter Tags 188

Chapter 9
Adding and Updating Records 197

Overview 197

Table 1: Command Tags 198
Table 2: Security Command Tags 200
Adding Records 200
Table 3: -Add Action Requirements 201
Updating Records 204
Table 4: -Update Action Requirements 204
Deleting Records 209
Table 5: -Delete Action Requirements 209
Duplicating Records 211
Table 6: -Duplicate Action Requirements 211

Chapter 10
MySQL Data Sources 213

Overview 213
MySQL Tags 215
Table 1: Enhanced MySQL Tags 215
Searching Records 216
Table 2: MySQL Search Field Operators 216
Table 3: MySQL Search Command Tags 218
Adding and Updating Records 221
Value Lists 222
Table 4: MySQL Value List Tags 223
Creating Database Tables 228
Table 5: Database Creation Tags 229
Table 6: [Database_CreateTable] Parameters: 230
Table 7: [Database_CreateField] and [Database_ChangeField] Parameters: 232
Table 8: MySQL Field Types 233

Chapter 11
SQLite Data Source 237

Overview 237
SQLite Tags 239
Table 1: SQLite Tags 239
Searching Records 240
Table 2: SQLite Search Command Tags 240
Storing Bytes 242
Creating SQLite Databases 243
Table 3: SQLite Database Create Tag 243

Chapter 12
FileMaker Data Sources 245

Overview 246
Performance Tips 247
Compatibility Tips 248
FileMaker Tags 249
Table 1: FileMaker Data Source Tag 249
Primary Key Field and Record ID 250
Sorting Records 251
Displaying Data 252
Table 2: FileMaker Data Display Tags 253
Value Lists 260
Table 3: FileMaker Value List Tags 261
Container Fields 266
Table 4: Container Field Tags 266
FileMaker Scripts 268
Table 5: FileMaker Scripts Tags 268

Chapter 13
JDBC Data Sources 271

Overview 271
Using JDBC Data Sources 272
JDBC Schema Tags 273
Table 1: JDBC Schema Tags 274

Section III
Programming 275

Chapter 14
Programming Fundamentals 277

Overview 278
Figure 1: Error Page 278
Logic vs. Presentation 279
Data Output 281
Table 1: Output Tags 281
Variables 283
Includes 285
Table 2: Include Tags 288
Data Types 290
Table 3: Data Type Tags 290

Symbols 295
Member Tags 297
Forms and URLs 299

Chapter 15
Variables 301

Overview 301
Page Variables 303
Table 1: Page Variable Tags 303
Table 2: Page Variable Symbols 303
Global Variables 307
Table 3: Global Tags 308
Local Variables 311
Table 4: Local Tags 311
Table 5: Local Variable Symbols 311
References 312
Table 6: Reference Tags and Symbols 315

Chapter 16
Conditional Logic 317

If Else Conditionals 318
Table 1: If Else Tags 319
If Else Symbol 321
Table 2: If Else Symbol 321
Select Statements 322
Table 3: Select Tags 323
Conditional Tags 324
Table 4: Conditional Tags 324
Loops 325
Table 5: [Loop] Tag Parameters 325
Table 6: Loop Tags 327
Iterations 329
Table 7: Iteration Tags 329
While Loops 330
Table 8: While Tags 331
Abort Tag 331
Table 9: Abort Tag 331
Boolean Type 331
Table 10: Boolean Tag 332
Table 11: Boolean Symbols 332

Chapter 17
Encoding 335

Overview 335
Encoding Keywords 338
Table 1: Encoding Keywords 339
Encoding Controls 339
Table 2: Encoding Controls 340
Encoding Tags 340
Table 3: Encoding Tags 341

Chapter 18
Sessions 343

Overview 343
Session Tags 345
Table 1: Session Tags 345
Table 2: [Session_Start] Parameters 346
Session Example 350

Chapter 19
Error Control 353

Overview 353
Error Reporting 355
Figure 1: Built-In None Error Message 356
Figure 2: Built-In Minimal Error Message 356
Figure 3: Built-In Full Error Message 357
Table 1: Error Level Tag 357
Figure 4: Lasso Service Error Message 358
Figure 5: Authentication Dialog 358
Custom Error Page 358
Figure 6: Custom Error Page 359
Error Pages 360
Table 2: Error Response Tags 361
Error Tags 361
Table 3: Error Tags 362
Table 4: Error Type Tags 363
Error Handling 364
Table 5: Error Handling Tags 364

Section IV
Upgrading. 371

Chapter 20
Upgrading From Lasso Professional 7 373

Lasso Professional 8.0.x 374
Introduction. 381
Security Enhancements 381
SQLite. 382
Multi-Site 383
Namespaces 384
Digest Authentication 385
On-Demand LassoApps 386
Syntax Changes 386
Tag Name Changes 388
Table 1: Tag Name Changes 388

Chapter 21
Upgrading From Lasso Professional 6 389

Introduction. 390
Error Reporting 390
Unicode Support 392
Bytes Type 395
Table 1: Tags That Return the Bytes Type 395
Table 2: Byte and String Shared Member Tags 396
Table 3: Unsupported String Member Tags 397
Syntax Changes 399
Table 4: Syntax Changes. 400
Tag Name Changes 417
Table 5: Unsupported Tags 417
Table 6: Tag Name Changes 417
Table 7: Deprecated Tags. 418

Chapter 22
Upgrading From Lasso Professional 5 419

Introduction. 419
Tag Name Changes 420
Table 1: Unsupported Tags 420
Table 2: Tag Name Changes 420
Table 3: Deprecated Tags. 421

Syntax Changes 422
Table 4: Syntax Changes 422
Lasso MySQL 426
Table 5: Lasso MySQL Syntax Changes 426

Chapter 23
Upgrading From Lasso WDE 3.x 427
Introduction 428
Syntax Changes 428
Table 1: Syntax Changes 428
Table 2: Line Endings 438
Tag Name Changes 443
Table 3: Command Tag Name Changes 443
Table 4: Substitution, Process, and Container Tag Name Changes . . . 444
Unsupported Tags 445
Table 5: Unsupported Tags 445
FileMaker Pro 446

Section V
Data Types 449

Chapter 24
String Operations 451
Overview 452
Table 1: String Tag 453
String Symbols 454
Table 2: String Symbols 454
String Manipulation Tags 457
Table 3: String Manipulation Member Tags 458
Table 4: String Manipulation Tags 459
String Conversion Tags 460
Table 5: String Conversion Member Tags 461
Table 6: String Conversion Tags 461
String Validation Tags 462
Table 7: String Validation Member Tags 462
Table 8: String Validation Tags 463
String Information Tags 463
Table 9: String Information Member Tags 464
Table 10: String Information Tags 465
Table 11: Character Information Member Tags 467

Table 12: Unicode Tags 469
String Casting Tags 469
Table 13: String Casting Member Tags 469
Regular Expressions 470
Table 14: Regular Expression Tags 470
Table 15: Regular Expression Matching Symbols 471
Table 16: Regular Expression Combination Symbols 472
Table 17: Regular Expression Replacement Symbols 472
Table 18: Regular Expression Advanced Symbols 473

Chapter 25
Bytes 477

Bytes Type 477
Table 1: Byte Stream Tag 478
Table 2: Byte Stream Member Tags 478

Chapter 26
Math Operations 483

Overview 483
Table 1: Integer Tag 484
Table 2: Decimal Tag 485
Mathematical Symbols 486
Table 3: Mathematical Symbols 486
Table 4: Mathematical Assignment Symbols 487
Table 5: Mathematical Comparison Symbols 488
Decimal Member Tags 489
Table 6: Decimal Member Tag 489
Table 7: [Decimal->SetFormat] Parameters 490
Integer Member Tags 491
Table 8: Integer Member Tags 491
Table 9: [Integer->SetFormat] Parameters 492
Math Tags 494
Table 10: Math Tags 494
Table 11: [Math_Random] Parameters 496
Table 12: Trigonmetric and Advanced Math Tags 497
Locale Formatting 498
Table 13: Locale Formatting Tags 498

Chapter 27
Date and Time Operations 499

Overview 499

Date Tags 500
Table 1: Date Substitution Tags 503
Table 2: Date Format Symbols 505
Table 3: Date Format Member Tags 507
Table 4: Date Accessor Tags 508
Duration Tags 509
Table 5: Duration Tags 510
Date and Duration Math 511
Table 6: Date Math Tags 512
Table 7: Date and Duration Math Tags 513
Table 8: Date Math Symbols 515

Chapter 28
Arrays, Maps, and Compound Data Types 517

Overview 518
Arrays 522
Table 1: Array Tag 523
Table 2: Array Member Tags 524
Table 3: [Array->Merge] Parameters 531
Lists 535
Table 4: List Tag 535
Table 5: List Member Tags 536
Maps 539
Table 6: Map Tag 539
Table 7: Map Member Tags 540
Pairs 544
Table 8: Pair Tag 544
Table 9: Pair Member Tags 545
Priority Queues 546
Table 10: Priority Queue Tag 546
Table 11: Priority Queue Member Tags 547
Queues 550
Table 12: Queue Tag 550
Table 13: Queue Member Tags 551
Series 554
Table 14: Series Tag 554
Sets 554
Table 15: Set Tag 554
Table 16: Set Member Tags 555
Stacks 557
Table 17: Stack Tag 558

Table 18: Stack Member Tags 558
Tree Maps 561
Table 19: Tree Map Tag 561
Table 20: Tree Map Member Tags 562
Comparators 566
Table 21: Comparators 566
Matchers 568
Table 22: Matchers 568
Iterators 571
Table 23: Iterator Tags 573
Table 24: Iterator and Reverse Iterator Member Tags 573

Chapter 29
Files 577

File Tags 577
Table 1: File Tags 581
Table 2: Line Endings 586
File Data Type 587
Table 3: [File] Tag 587
Table 4: File Open Modes 587
Table 5: File Read Modes 588
Table 6: File Streaming Tags 588
File Uploads 591
Table 7: File Upload Tags 592
Table 8: [File_Uploads] Map Elements 592
File Serving 594
Table 9: File Serving Tags 594

Chapter 30
Images and Multimedia 597

Overview 597
Table 1: Tested and Certified Image Formats 599
Casting Images as LDML Objects 600
Table 2: [Image] Tag: 600
Table 3: [Image] Tag Parameters: 600
Getting Image Information 601
Table 4: Image Information Tags 601
Converting and Saving Images 603
Table 5: Image Conversion and File Tags 603
Manipulating Images 604
Table 6: Image Size and Orientation Tags 605
Table 7: Image Effects Tags 606

Table 8: Annotate Image Tag 609
Table 9: Composite Image Tag 610
Table 10: Composite Image Tag Operators 610
Extended ImageMagick Commands 612
Table 11: ImageMagick Execute Tag 612
Serving Image and Multimedia Files 613
Table 12: Image Serving Tag 615

Chapter 31
Networking 619

Network Communication 619
Table 1: [Net] Tags 620
Table 2: [Net] Constants 620
Table 3: [Net] Type Member Tags 621
Table 3: [Net] TCP Member Tags 622
UDP Connections 626
Table 4: [Net] UDP Member Tags 626

Chapter 32
XML 629

Overview 630
XML Glossary 631
XML Data Type 632
Table 1: XML Data Type Tag 632
Table 2: XML Member Tags 633
XPath Extraction 635
Table 3: [XML_Extract] Tag 636
Table 4: Simple XPath Expressions 637
Table 5: Conditional XPath Expressions 639
XSLT Style Sheet Transforms 640
Table 6: [XML_Transform] Tag 641
XML Stream Data Type 642
Table 7: XML Stream Data Type Tag 642
Table 8: XML Stream Node Types 643
Table 9: XML Stream Navigation Member Tags 644
Table 10: XML Stream Member Tags 645
Serving XML 647
Table 11: [XML_Serve] Serving Tags 647
Formatting XML 648
XML Templates 649
Table 12: FileMaker Pro XML Templates 650
Table 13: SQL Server XML Templates 651

Chapter 33
Portable Document Format 653

Overview 654
Creating PDF Documents 655
Table 1: [PDF_Doc] Tag and Parameters 655
Table 2: [PDF_Doc->Add] Tag and Parameters 658
Table 3: PDF Page Tags 659
Table 4: PDF Read Tags 660
Table 5: Page Insertion Tag and Parameters 661
Table 6: PDF Accessor Tags 662
Table 7: [PDF_Doc->Close] Tag 663
Creating Text Content 663
Table 8: PDF Font Tag and Parameters 664
Table 9: [PDF_Font] Member Tags 665
Table 10: [PDF_Text] Tag and Parameters 667
Table 11: [PDF_Doc->DrawText] Tag 669
Table 12: [PDF_List] Tags and Parameters. 669
Table 13: Special Characters 671
Creating and Using Forms 671
Table 14: [PDF_Doc] Form Member Tags 672
Table 16: Form Placement Parameters 674
Creating Tables 678
Table 17: [PDF_Table] Tag and Parameters 678
Table 18: [PDF_Table] Member Tags. 679
Table 19: Cell Content Tags 680
Creating Graphics 682
Table 20: [PDF_Image] Tag and Parameters 682
Table 21: [PDF_Doc] Drawing Member Tags 684
Creating Barcodes 686
Table 22: [PDF_Barcode] Tag and Parameters 686
Example PDF Files. 688
Serving PDF Files 693
Table 23: PDF Serving Tags 694

Chapter 34
JavaBeans 697

Overview 697
Installing JavaBeans: 699
JavaBeans Type. 699
Table 1: JavaBeans Type 699
Creating JavaBeans 701

Section VI
Programming 703

Chapter 35
Namespaces 705
Overview 705
Namespace Tags 708
Table 1: Namespace Tags 708

Chapter 36
Logging 711
Overview 711
Log Tags 713
Table 1: Lasso Error Log Tags 713
Log Files 714
Table 2: File Log Tags 714
Log Routing 715
Table 3: Log Preference Tag 715
Table 4: Log Message Levels 716
Table 5: Log Destination Codes 716

Chapter 37
Encryption 719
Overview 719
Encryption Tags 720
Table 1: Encryption Tags 721
Cipher Tags 724
Table 2: Cipher Tags 724
Table 3: Cipher Algorithms 725
Table 4: Digest Algorithms 725
Compression Tags 726
Table 5: Compression Tags 726

Chapter 38
Control Tags 729
Authentication Tags 729
Table 1: Authentication Tags 730
Administration Tags 732
Table 2: Administration Tags 732

Scheduling Events 735
Table 3: Scheduling Tag 736
Table 4: Scheduling Parameters 736
Process Tags 739
Table 5: Process Tags. 739
Null Data Type. 741
Table 6: Null Member Tags 741
Page Variables 743
Table 7: Page Variable Tags 743
Table 8: Page Variables 744
Configuration Tags 745
Table 9: Configuration Tags. 745
Format File Execution Time Limit 746
Table 10: Time Limit Tags 746
Page Pre- and Post-Processing 747
Table 11: Pre and Post-Process Tags 747

Chapter 39
Threads 751

Thread Tools 751
Table 1: Thread Tools 752
Table 2: [Thread_Lock] Member tags: 753
Table 3: [Thread_Semaphore] Member Tags 754
Table 4: [Thread_RWLock] Member Tags. 755
Thread Communication 756
Table 5: Thread Communication 756
Table 6: [Thread_Event] Member Tags: 757
Table 7: [Thread_Pipe] Member Tags: 758

Chapter 40
Tags and Compound Expressions 759

Tag Data Type 759
Table 1: Tag Data Type Member Tags. 761
Table 2: [Tag->Run] Parameters 761
Compound Expressions 764
LDML Parsing 765
Table 3: LDML Type Tag. 766
Table 4: LDML Type Member Tags 767
Table 5: LDML Token Types 769

Chapter 41
Miscellaneous Tags 771

Name Server Lookup 771
Table 1: Name Server Lookup Tag 771
Validation Tags 772
Table 2: Valid Tags 772
Unique ID Tags 773
Table 3: Unique ID Tag 773
Server Tags 773
Table 4: Server Tags 773

Section VII
Protocols 775

Chapter 42
Sending Email 777

Overview 777
Sending Email 779
Table 1: Email Tag 779
Table 2: [Email_Send] Parameters 779
Table 3: Additional Email_Send] Parameters 783
Table 4: SMTP Server [Email_Send] Parameters 785
Composing Email 786
Table 5: Email Composing and Queuing Tags 786
SMTP Type 789
Table 6: SMTP Tags 789

Chapter 43
POP 791

Overview 791
POP Type 792
Table 1: [Email_POP] type 792
Email Parsing 796
Table 2: [Email_Parse] type 798
Helper Tags 803
Table 3: Email Helper Tags 803

Chapter 44
HTTP/HTML Content and Controls . . .805

Include URLs 806
Table 1: Include URL Tag 806
Table 2: [Include_URL] Parameters 807
Redirect URL 809
Table 3: Redirect URL Tag 809
HTTP Tags 810
Table 4: HTTP Tags 810
FTP Tags 811
Table 5: FTP Tags 811
Cookie Tags 813
Table 6: Cookie Tags 814
Table 7: [Cookie_Set] Parameters 815
Caching Tags 818
Table 8: [Cache] Tag 819
Table 9: [Cache] Tag Parameters 819
Table 10: LDML Object Cache Tags 822
Table 11: Cache Control Tags 823
Server Push 824
Table 12: Server Push Tag 825
Header Tags 825
Table 13: Header Tags 826
Request Tags 828
Table 14: Request Tags 829
Client Tags 830
Table 15: Client Tags 830
Server Tags 831
Table 16: Server Tags 831

Chapter 45
XML-RPC 833

Overview 833
Calling a Remote Procedure 834
Table 1: [XML_RPCCall] Tag 834
Table 2: XML-RPC Built-In Methods 835
Table 3: XML-RPC and Built-In Data Types 836
Table 4: XML-RPC Data Type 836
Table 5: [XML_RPC] Call Tag 837
Creating Procedures 837
Processing an Incoming Call 838
Table 6: [XML_RPC] Processing Tags 839

Chapter 46
SOAP 841

Overview 841
Methodology 842
Calling SOAP Procedures 845
Table 1: Built-In Processors 849
Defining SOAP Procedures 850
Low-Level Details 851

Chapter 47
Wireless Devices 857

Overview 857
Formatting WML 858
WAP Tags 861
Table 1: WAP Tags 861
WML Example 862

Section VIII
LDML API 865

Chapter 48
LDML Introduction 867

Overview 867
LassoApps 868
Custom Tags 868
Custom Types 869
LDML Data Sources 869

Chapter 49
LassoApps 871

Overview 871
Table 1: LassoApp Tags 873
Default LassoApps 873
Administration 874
Serving LassoApps 876
Preparing Solutions 879
Building LassoApps 882
Table 2: [LassoApp_Create] Tag Parameters 884

Tips and Techniques 885

Chapter 50
Custom Tags 889

Overview 890
Custom Tags 893
Table 1: Tags For Creating Custom Tags 893
Table 2: [Define_Tag] Parameters 894
Container Tags 907
Web Services, Remote Procedure Calls, and SOAP 909
Asynchronous Tags 911
Overloading Tags 915
Constants 919
Table 3: [Define_Constant] Tag 919
Libraries 919

Chapter 51
Custom Types 921

Overview 922
Custom Types 923
Table 1: Tags for Creating Custom Data Types 924
Member Tags 926
Table 2: Built-In Member Tags 927
Prototypes 929
Table 3: Prototype Tag 930
Callback Tags 930
Table 4: Callback Tags 931
Symbol Overloading 935
Table 5: Overloadable Symbols 936
Table 6: Comparison Callback Tags 937
Table 7: Symbol Callback Tags 940
Table 8: Assignment Callback Tags 942
Inheritance 944
Libraries 946

Chapter 52
LDML Data Sources 947

Overview 947
Data Source Register 948
Table 1: Data Source Register 948
Data Source Type 949

Table 2: Data Source Member Tags 949
Table 3: Host Information 951
Table 4: Result Set Tags 957

Section IX
Lasso C/C++ API 959

Chapter 53
LCAPI Introduction 961
Overview 961
Requirements 962
Getting Started 962
Debugging 964
Frequently Asked Questions 965

Chapter 54
LCAPI Tags 969
Substitution Tag Operation 969
Substitution Tag Tutorial 970

Chapter 55
LCAPI Data Types 975
Data Type Operation 975
Data Type Tutorial 977

Chapter 56
LCAPI Data Sources 983
Data Source Connector Operation 983
Data Source Connector Tutorial 984

Chapter 57
Lasso Connector Protocol 993
Overview 993
Requirements 994
Lasso Web Server Connectors 994
Lasso Connector Operation 996
Table 1: LPCommandBlock Structure Members 997

Lasso Connector Protocol Reference 997
Table 2: Named Parameters 998

Section X
Lasso Java API 999

Chapter 58
LJAPI Introduction 1001
Overview 1001
What's New 1002
LJAPI vs. LCAPI 1003
Requirements 1004
Getting Started 1005
Debugging 1008

Chapter 59
LJAPI Tags 1009
Substitution Tag Operation 1009
Substitution Tag Tutorial 1010

Chapter 60
LJAPI Data Types 1017
Data Type Operation 1017
Data Type Tutorial 1017
Table 1: Type initializer and Member Tags 1018
Table 2: Accessors 1018

Chapter 61
LJAPI Data Sources 1031
Data Source Connector Operation 1031
Data Source Connector Tutorial 1032

Chapter 62
LJAPI Reference 1049
LJAPI Interface Reference 1049
LJAPI Class Reference 1050

Appendix A
LDML 8 Tag List 1089

Appendix B
Error Codes. 1091
Lasso Professional 8 Error Codes 1091
Table 1: Lasso Professional 8 Error Codes 1092
Lasso MySQL Error Codes 1096
Table 2: Lasso MySQL Error Codes 1096
FileMaker Pro Error Codes 1100
Table 3: FileMaker Pro Error Codes 1100
JDBC Error Codes 1104
Table 4: JDBC Error Codes 1104

Appendix C
Copyright Notice 1105

Appendix D
Index 1107

Section I
Lasso Overview

This section includes an introduction to the fundamental concepts and methodology for building and serving data-driven Web sites powered by Lasso 8. Every new user should read through all the chapters in this section.

- *Chapter 1: Introduction* includes information about the documentation available for Lasso 8 and about this book.
- *Chapter 2: Web Application Fundamentals* includes an introduction to essential concepts and industry terms related to serving data-driven Web sites.
- *Chapter 3: Format Files* discusses how to create and work with Lasso 8 format files.
- *Chapter 4: LDML 8 Syntax* introduces the syntax of Lasso including square brackets, LassoScript, compound expressions, colon syntax, and parentheses syntax.
- *Chapter 5: LDML 8 Tag Language* introduces Lasso Dynamic Markup Language (LDML), the language of Lasso 8.
- *Chapter 6: LDML 8 Reference* introduces the reference database which contains complete details about the syntax of every tag in LDML 8.

After completing *Section 1: Lasso Overview* you can proceed to *Section II: Database Interaction* to learn how to store and retrieve information from a database and to *Section III: Programming* to learn how to program in LDML.

Users who are upgrading from a previous version of Lasso should read the appropriate chapters in Section IV: Upgrading.

The remainder of the Language Guide includes reference material for all the many tags that Lasso supports and information about how to extend Lasso's functionality by creating custom tags, custom data sources, and custom data types in LDML, C/C++, or Java.

1

Chapter 1
Introduction

This chapter provides on overview of the Lasso 8 documentation, the section outline, and documentation conventions for this book.

- *Lasso 8 Documentation* describes the documentation included with Lasso 8 products.
- *Lasso 8 Language Guide* describes the volumes and sections in this book.
- *Documentation Conventions* includes information about typographic conventions used within the documentation.

Lasso 8 Documentation

The documentation for Lasso 8 products is divided into several different manuals and also includes several online resources. The following manuals and resources are available.

- **Lasso Professional Server 8 Setup Guide** is the main manual for Lasso Professional 8. It includes documentation of the architecture of Lasso Professional 8, installation instructions, the administration interface, and Lasso security. After the release notes, this is the first guide you should read.
- **Lasso 8 Language Guide** includes documentation of LDML (Lasso Dynamic Markup Language), the language used to access data sources, specify programming logic, and much more.
- **LDML 8 Reference** provides detailed documentation of each tag in LDML 8. This is the definitive reference to the language of Lasso 8. This reference is provided as a LassoApp and Lasso MySQL database within

Lasso Professional 8 and also as an online resource from the OmniPilot Web site.

Comments, suggestions, or corrections regarding the documentation may be sent to the following email address.

lassodocumentation@omnipilot.com

Lasso 8 Language Guide

This is the guide you are reading now. This guide contains information about programming in LDML and is organized into the following volumes and sections. The print version of this guide is separated into three distinct volumes. The PDF version of this guide includes all three volumes in the same file.

Volume 1 – Fundamentals

The first four sections of the Language Guide introduce the language of Lasso, explain fundamental database interaction and programming concepts, and describe how to upgrade existing solutions.

- *Section I: Lasso Overview* contains important information about using and programming Lasso that all developers who create custom solutions powered by Lasso will need to know.
- *Section II: Database Interaction* contains important information about how to create format files that perform database actions. Actions can be performed in the internal Lasso MySQL database or in external MySQL, FileMaker Pro, or other databases.
- *Section III: Programming* describes how to program dynamic format files using LDML. This section covers topics ranging from simple data display through advanced error handling and alternate programming syntaxes.
- *Section IV: Upgrading* includes details about what has changed in Lasso Professional 8 since Lasso Professional 7, Lasso Professional 6, Lasso Professional 5, and Lasso WDE 3.x and earlier. The appropriate chapters in this section are essential reading for any developer who is upgrading from an earlier version of Lasso.

Volume 2 – Reference

The next three sections of the Language Guide provide in-depth information about Lasso's data types, advanced programming concepts, and support for Internet protocols.

- *Section V: Data Types* describes the built-in data types in Lasso including strings, bytes, dates, compound data types, files, images, network communications, XML, PDF, and JavaBeans.
- *Section VI: Programming* describes programming concepts in LDML including namespaces, logging, encryption, control tags, threads, custom tags, and compound expressions.
- *Section VII: Protocols* describes how to use Lasso to interoperate with other Internet technologies such as email servers and remote Web servers. It describes how to use Lasso to serve images and multimedia files. It also describes how to use Lasso to serve pages to various clients including Web browsers, WAP browsers and more.

Volume 3 – Extending

The final three sections of the Language Guide describe how to extend the functionality of Lasso by programming new tags, data types, and connectors in LDML, C/C++, or Java. This volume also includes the appendices.

- *Section VIII: LDML API* contains information about creating LassoApps, custom tags, custom data types, and data source connectors in LDML.
- *Section IX: LCAPI* contains information about creating tags, data types, and data source connectors in the C/C++ programming languages. Also describes how to create new Web server connectors..
- *Section X: LJAPI* contains information about creating tags, data types, and data source connectors in the Java programming language.
- *Appendices* contain a listing of error codes as well as copyright notices and the index for all three volumes.

Documentation Conventions

The documentation uses several conventions in order to make finding information easier.

Definitions are indicated using a bold, sans-serif type face for the defined word. This makes it easy to find defined terms within a page. Terms are defined the first time they are used.

Cross References are indicated by an italicized sans-serif typeface. For instance, the current section in this chapter is *Documentation Conventions*. When necessary, arrows are used to define a path into a chapter such as *Chapter 1: Introduction > Documentation Conventions*.

Code is formatted in a narrow, sans-serif font. Code includes HTML tags, LDML tags, and any text which should be typed into a format file. Code is represented within the body text (e.g., [Field] or <body>) or is specified in its own section of text as follows:

```
[Field: 'Company_Name']
```

Code Results represent the result after code is processed. They are indicated by a black arrow and will usually be the value that is sent to the client's Web browser. The following text could be the result of the code example above.

```
➜ OmniPilot
```

Note: Notes are included to call attention to items that are of particular importance or to include comments that may be of interest to select readers. Notes may begin with ***Warning***, ***FileMaker Pro Note***, ***IIS Note***, etc. to specify the importance and audience of the note.

To perform a specific task:

The documentation assumes a task-based approach. The contents following a task heading will provide step-by-step instructions for the specific task.

2

Chapter 2
Web Application Fundamentals

This chapter presents an overview of fundamental concepts that are essential to understand before you start creating data-driven Web sites powered by Lasso Professional 8.

- *Web Browser Overview* describes how HTML pages and images are fetched and rendered.
- *Web Server Overview* describes how HTTP requests and URLs are interpreted.
- *HTML Forms and URL Parameters* describes how GET and POST arguments are sent and interpreted.
- *Web Application Servers* describes how interactive content is created and served.
- *Web Application Server Languages* describes how commands can be embedded within a format file, processed, and served.
- *Error Reporting* describes how errors are reported by Lasso and how to customize the amount of information that is provided to site visitors.

Web Browser Overview

The World Wide Web (WWW) is accessed by end-users through a Web browser application. Popular Web browsers include Microsoft Internet Explorer and Netscape Navigator. The Web browser is used to access pages served by one or more remote Web servers. Navigation is made possible via hyperlinks or HTML forms. The simple point-and-click operation of

the Web browser masks a complex series of interactions between the Web browser and Web servers.

URLs

The location of a Web site and a particular page within a site are specified using a Universal Resource Locator (URL). All URLs follow the same basic format:

```
http://www.example.com:80/folder/file.html
```

The URL is comprised of the following components:

1. The **Protocol** is specified first, http in the example above and is followed by a colon. The World Wide Web has two protocols. HTTP (HyperText Transfer Protocol) which is for standard Web pages and is the default for most Web browsers and HTTPS (HyperText Transfer Protocol Secure) which is for pages served encrypted via the Secure Socket Layer (SSL).
2. The **Host Name** is specified next, www.example.com in the example above. The host name can be anything defined by a domain name registrar. It need not necessarily begin with www, the same server may be accessible using example.com or by an IP address such as 127.0.0.1.
3. The **Port Number** follows the host name, 80 in the example above. The port number can usually be left off because a default is assumed based on the protocol. HTTP defaults to port 80 and HTTPS defaults to port 443.
4. The **File Path** follows a forward slash, /folder/file.html in the example above. The Web server uses this path to locate the desired file relative to the root of the Web serving folder configured for the specified domain name. The root of the Web serving folder is typically C:\InetPub\wwwroot\ for Windows 2000 servers and /Library/WebServer/Documents for Mac OS X servers.

HTTP Request

The URL is used by the Web browser to assemble an HTTP request which is actually sent to the Web server. The HTTP request resembles the header of an email file. It consists of several lines each of which has a label followed by a colon and a value.

Note: Most current Web browsers and Web servers support the HTTP/1.1 standard. Lasso Professional 8 also supports this standard. However, the examples in this book are written for the HTTP/1.0 standard in order to provide maximum compatibility with older Web browser clients.

The URL http://www.example.com/folder/file.html becomes the following HTTP request:

```
GET /folder/file.html HTTP/1.0
Accept: */*
Host: www.example.com
User-Agent: Web Browser/4.1
```

The HTTP request is comprised of the following components:

1. The first line defines the HTTP request. The action is GET and the path to what should be returned is specified /folder/file.html. The final piece of information is the protocol and version which should be used to return the data, HTTP/1.0 in the example above.
2. The **Accept** line specifies the types of data that can be accepted as a return value. */* means that any type of data will be accepted.
3. The **Host** line specifies the host which was requested in the URL.
4. The **User-Agent** line specifies what type of browser is requesting the information.

HTTP Response

Once an HTTP request has been submitted to a server, an HTTP response is returned. The response consists of two parts: a response header which has much the same structure as the HTTP request and the actual text or binary data of the page or image which was requested.

The URL http://www.example.com/folder/file.html might result in the following HTTP response header:

```
HTTP/1.0 200 OK
Server: Lasso Professional 8.0
MIME-Version: 1.0
Content-type: text/html; charset=iso-8859-1
Content-length: 7713
```

The HTTP response header is comprised of the following components:

1. The first line defines the type of response. The protocol and version are given followed by a response code, 200 OK in the example above.
2. The **Server** line specifies the type of Web server that returned the data. Lasso Professional 8 returns Lasso Professional 8.0 in the example above.
3. The **MIME-Version** line specifies the version of the MIME standard used to define the remaining lines in the header.

4 The **Content-type** line defines the type of data returned. text/html means that ASCII text is being returned in HTML format. This line could also read text/xml for XML data, image/gif for a GIF image or image/jpeg for a JPEG image.

   The charset=iso-8859-1 parameter specifies the character set of the page. Lasso returns pages in UTF-8 encoding by default or in the character set specified in the [Content_Type] tag.

5 **Content-length** specifies the length in bytes of the data which is returned along with this HTTP response header.

The header is followed by the text of the HTML page or binary data of the image which was requested.

Requesting a Web Page

The following are the series of steps which are performed each time a URL is requested from a Web server:

1 The Web browser determines the protocol for the URL. If the protocol is not HTTP then it might be passed off to another application. If the protocol is HTTPS then the Web browser will attempt a secure connection to the server.

2 The Web browser looks up the IP address of the server through a Domain Name Server (DNS).

3 The Web browser assembles an HTTP request including the path to the requested page.

4 The Web browser parses the HTML returned by the request and renders it for display to the visitor.

5 If the HTML contains any references to images or linked style sheets then additional HTTP requests with appropriate paths are generated and sent to the Web server.

6 The images and linked style sheets are used to modify the rendered HTML page.

7 Client-side scripting language such as JavaScript are interpreted and may further modify the rendered page.

The Web browser opens a new HTTP request for each HTML page, style sheet, or image file that is requested. All HTTP requests for a given HTML page can be sent to the same Web server or to different Web servers depending on how the HTML page is written. For example, many HTML pages reference advertisements served from a completely different Web server.

Character Sets

All Web pages must be transmitted from server to client using a character set that maps the actual bytes in the transmission to characters in the fonts used by the client's Web browser. The Content-Type header in the HTTP response specifies to the Web browser what character set the contents of the page has been encoded in.

Lasso processes all data internally using double-byte Unicode strings. Since two bytes are used to represent each character characters from single-byte ASCII are padded with an extra byte. Double-byte strings also allow for 4-byte or even larger characters using special internally encoded entities..

For transmission to the Web browser Lasso uses another Unicode standard UTF-8 which uses one byte to represent each character. UTF-8 corresponds roughly to traditional ASCII and the Latin-1 (ISO 8859-1) character set. Double-byte or 4-byte characters are represented by entities. For example, the entity 並 represents the double byte character 並.

For older browsers or other Web clients it may be necessary to send data in a specific character set. Some clients may expect data to be transmitted in the pre-Unicode standard of Latin-1 (ISO 8859-1). Lasso will honor the [Content_Type] tag in order to decide what character set to use for transmission to the Web browser. Using the following tag will result in the Latin-1 (ISO 8859-1) character set being used.

```
[Content_Type: 'text/html; charset=iso-8859-1']
```

Note: UTF-8 is an abbreviation for the 8-bit (single-byte) UCS Transformation Format. UCS is in turn an abbreviation for Universal Character Set. Since 8-bit Universal Character Set Transformation Format is such a mouthful it helps to think of UTF-8 simply as the most common Unicode character encoding.

Cookies

Cookies allow small amounts of information to be stored in the Web browser by a Web server. Each time the Web browser makes a request to a specific Web server, it sends along any cookies which the Web server has asked to be saved. This allows for the Web server to save the state of a visitor's session within the Web browser and then to retrieve that state when the visitor next visits the Web site, even if it is days later.

Cookies are set in the HTTP header for a file that is sent from the Web server. A single HTML file can set many cookies and cookies can even be set in the headers of image files. Each cookie has a name, expiration date, value, and the IP address or host name of a Web server. The following line in an HTTP header would set a cookie named session-id that expired on

January 1, 2010. The cookie will be returned in the HTTP request for any domains that end in example.com.

```
Set-Cookie: session-id=102-2659358; path=/; domain=.example.com;
expires=Wednesday, 1-January-2010 08:00:00 GMT
```

Each time a request is made to a Web server, any cookies which are labeled with the IP address or host name of the Web server are sent along with all HTTP requests for HTML files or image files. The Web server is free to read these cookies or ignore them. The HTTP request for any file on example.com or www.example.com would include the following line.

```
Cookie: session-id=102-2659358
```

Cookies are useful because small items of information can be stored on the client machine. This allows a customer ID number, shopping cart ID number, or simple site preferences to be stored and retrieved the next time the user visits the site.

Cookies are dependent upon support from the Web browser. Most Web browsers allow for cookie support to be turned off or for cookies to be rejected on a case-by-case basis. The maximum size of cookies is Web browser dependent and may be limited to 32,000 characters or fewer for each cookie or for all cookies combined.

Cookies can be set to expire after a certain number of minutes or at the end of the current user's session (until they quit their Web browser). However, this expiration behavior should not be counted on. Some Web browsers do not expire any cookies until the Web browser quits. Others do not expire cookies until the machine hosting the Web browser restarts. Some Web browsers even allow visitors to alter the expiration dates of stored cookies.

Authentication

Web browsers support authentication of the visitor. A username and password can be sent along with each HTTP request to the server. This username and password can be read or ignored by the Web server. If the Web server is expecting a username and password and does not find any or does not find a valid username and password then the server can send back a challenge which forces the browser to display an authentication dialog box.

The following lines at the start of an HTTP response header will force most Web browsers to challenge the visitor for a username and password. The response code 401 Unauthorized informs the Web browser that the user is not authorized to view the requested file.

```
HTTP/1.0 401 Unauthorized
```

A header line in the response informs the client what types of authentication are understood by the server. By default Lasso prompts for both basic and digest authentication. Clients that can perform digest authentication will use it. Older clients will use basic authentication.

```
WWW-Authenticate: Basic realm="Testing"
WWW-Authenticate: Digest realm="Testing",
        nonce="1234567890", uri="http://www.example.com/", algorithm="md5"
```

A basic authentication response includes a line like the following. The username and password are concatenated together and encoded using Base64, but are not encrypted.

```
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

A digest authentication response includes a line like the following. The realm and nonce are passed back along with the URL of the requested page. The response portion is made up of an MD5 hashed value which includes the nonce and the user's password.

```
Authorization: Digest username="test", realm="Testing",
        nonce="1234567890", uri="http://www.example.com/"
        response="9c384179f883e2e9c1eed63ca752560a"
```

The advantages of digest authentication are numerous. The user's password is never sent in plain text (or simply encoded). The realm is remembered so a user can maintain different privileges in different parts of a Web site. The nonce can also be expired in order to force a user to re-authenticate.

Using either basic or digest authentication, the same username and password will continue to be transmitted to the Web server until the user re-authenticates or quits the Web browser application.

Site visitors can also specify usernames and passwords within the URL directly. This method allows a username and password to be sent before an authorization challenge is issued.

```
http://username:password@www.example.com/folder/default.lasso
```

Note: This method is no longer supported by all Web servers due to its potential use as a Web site spoofing technique.

Lasso-based Web sites also support specifying a username and password using -Username and -Password URL parameters.

```
http://www.example.com/default.lasso?-username=username&-password=password
```

Note: See the section on *Authentication Tags* in the *Lasso Control Tags* chapter for information about LDML tags that automatically prompt for authentication information.

Web Server Overview

The World Wide Web is served to end-users by Web server applications. Popular Web servers include Apache, WebSTAR, and Microsoft Internet Information Services (IIS). The Web server handles incoming HTTP requests for URLs from Web browsers. The interaction described in the previous section from the Web browser's point of view looks a little different from the Web server's point of view.

The following are the series of steps which are performed each time a URL is requested from a Web server:

1. The HTTP request is received on one of the ports which is being listened to by the Web server. Most Web servers listen on port 80 for HTTP requests and on port 443 for secure HTTPS requests.
2. The HTTP request is parsed and split into its components: protocol, host name, file path.
3. The host name is used to decide what virtual host to serve a Web page from. Most Web servers operate from a single IP address, but serve pages for several different domain names. These may be as simple as www.example.com and example.com.
4. The path to the page request is added to the server root for the specified virtual host. The virtual hosts may all start in a different folder on the hard drive.
5. The security settings of the server are checked to see if the user needs to be authenticated to receive the page they are requesting. If an appropriate username and password are not specified in the HTTP request then a challenge is sent in the HTTP response instead of the request page.
6. Server-side plug-ins or modules are called upon to process the request page. For example, requests for HTML pages that have a file name with the suffix .lasso will be sent to Lasso Service for processing. The processed page is returned to the Web server and may even be sent through multiple server-side plug-ins or modules before being served.
7. The requested HTML page or image is returned to the user with an appropriate HTTP response header.

HTML Forms and URL Parameters

HTML forms and URLs allow for significant amounts of data to be transmitted along with the simple HTTP requests defined in the previous sections. The data to be transmitted can either be included in the URL or passed in the HTTP request itself.

URL Parameters

A URL can include a series of name/value parameters following the file path. The name/value parameters are specified following a question mark ?. The name and value are separated by an equal sign = and multiple name/value parameters are attached to a single URL with ampersands &. The following URL has two name/value parameters: name1=value1 and name2=value2.

```
http://www.example.com/folder/file.lasso?name1=value1&name2=value2
```

The URL parameters are simply added to the file path which is specified in the HTTP request. The URL above might generate the following HTTP request. Since the parameters follow the word GET they are often referred to as GET parameters.

```
GET /folder/file.lasso?name1=value1&name2=value2 HTTP/1.0
Accept: */*
Host: www.example.com
User-Agent: Web Browser/4.1
```

Since the characters : / ? & = @ # % are used to define the structure of a URL, the file path and URL parameters cannot include these characters without modifying them so that the structure of the URL is not disturbed. The characters are modified by encoding them into %nnn entities where nnn is the hexadecimal ASCII code for the character being replaced. / is encoded as %2f for example.

HTML Forms

HTML forms provide user interface elements in the Web browser so that a visitor can customize the parameters which will be transmitted to the Web server along with an HTTP request. HTML forms can be used to modify the GET parameters of a URL or can be used to send POST parameters.

Note: A full discussion of the HTML tags possible within an HTML form is beyond the scope of this section. Please see an HTML reference for a full listing of HTML form elements.

Example of an HTML form with a GET method:

The following HTML form has an action which specifies the URL that will be returned when this form is submitted. In this case the URL is http://www.example.com/folder/file.lasso. The method of the form is defined to be GET. This ensures that the parameters specified by the HTML form inputs will be added to the URL as GET parameters.

```
<form action="http://www.example.com/folder/file.Lasso" method="GET">
  <input type="text" name="value1" value="value1">
  <input type="submit" name="value2" value="value2">
</form>
```

This form generates the following HTTP request. It is exactly the same as the HTTP request created by the URL http://www.example.com/folder/file.lasso?name1=value1&name2=value2.

```
GET /folder/file.lasso?name1=value1&name2=value2 HTTP/1.0
Accept: */*
Host: www.example.com
User-Agent: Web Browser/4.1
```

Example of an HTML form with a POST method:

The following HTML form has an action which specifies the URL that will be returned when this form is submitted. In this case the URL is http://www.example.com/folder/file.lasso. The method of the form is defined to be POST. This ensures that the parameters specified by the HTML form inputs will be added to the HTTP request as POST parameters and that the URL will be left unmodified.

```
<form action="http://www.example.com/folder/file.Lasso" method="POST">
  <input type="text" name="value1" value="value1">
  <input type="submit" name="value2" value="value2">
</form>
```

This form generates the following HTTP request. The request file is simply that which was specified in the action, but the method is now POST. The HTML form parameters are specified as the content of the HTTP request. They are still URL encoded, but now appear at the end of the HTTP request, rather than as part of the URL.

```
POST /folder/file.lasso HTTP/1.0
Accept: */*
Host: www.example.com
User-Agent: Web Browser/4.1
Content-type: application/x-www-form-urlencoded
Content-length: 27
value1=value1&name2=value2
```

HTML Forms and URL Responses

The GET and POST parameters passed in HTML forms or URLs are most often used by server-side plug-ins or modules to provide interactive or data-driven Web pages. The GET and POST parameters are how values are passed to Lasso in order to specify database actions, search parameters, or for any purpose a Lasso developer wants.

Web Application Servers

A **Web Application Server** is a program that works in conjunction with a Web server and provides programmatically generated HTML pages or images to Web visitors. Web application servers include programs that adhere to the Common Gateway Interface (CGI), programs which have built-in Web servers, plug-ins or modules for Web server applications, and services or dæmons that communicate with Web server applications.

Lasso Professional 8 is a Web application server which runs as a background service and communicates with the Web server Apache via a module called Lasso Connector for Apache, the Web server WebSTAR V via a plug-in called Lasso Connector for WebSTAR, or IIS via an ISAPI filter called Lasso Connector for IIS.

Web application servers are triggered in different ways depending on the Web server being used. Many Web application servers are triggered based on file suffix. For example, all file names ending in .lasso could be processed by Lasso Service. Any file suffix can be configured to trigger processing by Lasso Service including .html so all HTML pages will be processed before being served. Web application servers can usually also be set to process all pages that are served by a Web server.

Most Web application servers function by interpreting a programming or scripting language. Commands in the appropriate language are embedded in format files and then executed when an appropriate HTML form or URL is selected by a Web site visitor. The Web application server accepts the GET and POST parameters in the HTML form or URL, interprets the commands contained within the referenced format file, and returns a rendered HTML page to the Web site visitor.

Developers can choose to develop complete Web sites using the scripting language provided by a Web application server or they can purchase solutions which are written using the scripting language of a particular Web application server.

Lasso Professional 8 is a scriptable Web application server with a powerful tag-based language called Lasso Dynamic Markup Language (LDML). Custom solutions can be created by following the instructions contained in this Lasso 8 Language guide. Links to pre-packaged, third party solutions can be found on the OmniPilot Web site.

http://www.omnipilot.com/

Web Application Server Languages

There are two main types of languages provided by Web application servers.

- **Scripting Languages** are used to specify programming logic and are generally close in function to traditional programming languages. Scripting languages can be used to assemble HTML pages and output them to the Web visitor. Server-Side JavaScript is an example of a scripting language.
- **Tag-Based Languages** are used to specify data formatting and programming logic within pre-formatted HTML or XML format files. The tags embedded in the format file are interpreted and the output is modified before the page is served to the Web visitor. Server Side Includes (SSI) is an example of a tag-based language.

Lasso Professional 8 provides one language, LDML, which functions as both a scripting language and a tag-based language. LDML tags can be used in LassoScripts as a scripting language to define programming logic. LassoScripts can be used to render individual HTML tags or to render complete HTML documents programmatically. LDML tags can also be used as a tag-based language inside square brackets within HTML or XML code.

Error Reporting

When syntax or logical errors occur while processing a format file, Lasso will display an error page. The amount of information which is provided on the error page can be customized in a number of ways.

- The error reporting level can be adjusted in Site Administration to control how much information is provided on the default error page. A reporting level of None provides only a statement that an error occurred with no details. A level of Minimal provides only the error code and a brief error message. A level of Full provides detailed troubleshooting information.

- The error reporting level can be adjusted for a single format file using the [Lasso_ErrorReporting] tag. For example, the global error reporting level could be set to Minimal. While a page is being coded it can use [Lasso_ErrorReporting] to set the level for that page only to Full.
- Using the -Local keyword, the [Lasso_ErrorReporting] tag can be used to limit the error information from sensitive custom tags or include files. With this keyword the tag adjusts the error level only for the immediate context.
- A custom error.lasso page can be created for each Web host. This custom error page can provide an appropriate level of detail to Web site visitors and can be presented in the same appearance as the rest of the Web site. In addition, the custom error page can log or even email errors to the site administrator.
- A custom site-wide error.lasso page can be created which will override the built-in error page entirely. This custom page can be created on a shared site to provide appropriate error information to all users of the site.
- A custom server-wide error.lasso page can be created which will override the built-in error page for all sites. This custom page can be created on a shared server to provide appropriate error information to all users of the server.

More information about each of these options can be found in the *Error Control* chapter. Consult that chapter for full details about how to use the [Lasso_ErrorReporting] tag and how to create custom error pages.

3

Chapter 3
Format Files

This chapter introduces the concept of format files that contain LDML tags and LassoScripts. All new users of Lasso 8 should read this chapter.

- *Introduction* includes basic information about how format files are created and used in Lasso 8.
- *Storage Types* introduces the different methods of storing and retrieving format files.
- *Naming Format Files* describes the rules for naming format files.
- *Character Encoding* describes how Lasso uses the Unicode byte order mark to determine whether to read a file using the UTF-8 or Latin-1 (also known as ISO 8859-1) character set.
- *Editing Format Files* explains the options which are available for editing format files.
- *Functional Types* describes the various ways in which format files are used and introduces functional names for different types of format files.
- *Action Methods* introduces the concept of actions and describes how format files and LDML interact to create an action.
- *Securing Format Files* explains the importance of maintaining security for your format files.
- *Output Formats* shows how to use a format file to create output of various types.
- *File Management* explains how the architecture of Lasso 8 influences where files are stored and how they can be manipulated.
- *Specifying Paths* shows how URLs, HTML forms, and paths can be used to refer to format files.
- *Format File Execution Time Limit* describes the built-in limit on the length of time that format files will be allowed to execute.

Introduction

Format files are text files that contain embedded Lasso 8 code. When a format file is processed by Lasso Service, the embedded LDML tags are interpreted, executed, and the results are substituted in place of the tags. The resulting document is then returned to the client. Web sites powered by Lasso 8 are programmed by creating format files which include user interface elements, database actions, and display logic.

This chapter describes the different methods of storing, naming, and editing format files. It also discusses how multiple format files and LDML work together to create actions. The chapter finishes with discussions of how to output different types of data with format files and how to reference format files from within LDML tags, URLs, and HTML forms.

Note: Many of the terms used in this chapter are defined in ***Appendix A: Glossary*** of the Lasso Professional 8 Setup Guide. Please consult this glossary if you are unsure how any words are being used in this language guide.

Storage Types

The term **Format File** is used to describe any text file that contains embedded Lasso 8 code. Format files are usually stored on the local disk of the machine which hosts a Lasso Web server connector, but can also be stored on a remote machine, the machine which hosts Lasso Service, or even in a database field.

Format files are always text-based, but the structure of the text is not important to Lasso. Lasso will find the embedded LDML 8 tags, process them, and replace them with the results. Lasso will not disturb the text that surrounds the LDML tags, but may modify text which is contained within LDML container tags. The most common types of format files are described below.

- **HTML Format Files** contain a mix of LDML tags and HTML tags. HTML format files can be edited in leading visual Web authoring programs with LDML tags represented as icons or displayed as plain text. The output is usually HTML suitable for viewing in a Web browser.
- **XML Format Files** contain a mix of LDML tags and XML tags. When a developer creates an XML format file it may not be strictly valid XML code. However, it is constructed in such a way that the output after being processed by Lasso is valid XML code. XML format files can be constructed so that their output conforms to any Document Type Definition (DTD) or XML Schema.

- **Text Format Files** contain a mix of LDML tags and ASCII text. Text format files can be used as the body of email messages or can be used to output data in any ASCII-compatible form.
- **LDML Format Files** contain only LDML tags. Pure LDML format files usually contain programming logic and include other content types as needed. A pure LDML format file could be a placeholder that returns the appropriate type of content to whatever client loads the page.

Lasso format files can be stored in a variety of locations depending on how they are going to be used. Four locations are listed below, along with brief descriptions of how format files stored within them are used.

- **Web Server** – Format files are typically stored as text files on the machine which hosts the Web serving software with a Lasso Web server connector. The format files are stored along with the HTML and image files that comprise the Web site. As the client browses the site, they may visit some pages which are processed by Lasso Service and others that are served without any processing.
- **Lasso Service** – Format files can be stored on the machine which hosts Lasso Service. Usually, these format files serve a special purpose such as library files in the LassoStartup folder that contain code which is executed when Lasso Service starts up.
- **Database Field** – Format files can be stored as text in a database field. When a database action is performed the contents of the field are returned to the client as if a disk-based text file had been processed and served. Permission must be granted in Lasso's administration interface in order to use a database field in this fashion. See the *Setting Up Data Sources* chapter in the Lasso Professional 8 Setup Guide for more information.
- **Remote Server** – Lasso will not process LDML code which is stored on remote servers, but it can incorporate content from remote Web servers into the results served to the client or trigger CGI actions on remote servers using the [Include_URL] tag. See the *Files and Logging* chapter for more information.

Naming Format Files

The Lasso Professional 8 Installer will automatically configure your Web server to pass files named with a .lasso suffix to Lasso Service for processing. Once it has finished processing a file, Lasso Service passes the resulting file back to the Web server, which in turn sends the file to the client's Web browser. Files with other extensions, such as .gif or .jpg image files or .html

files are served directly by the Web server without being processed by Lasso Service.

In addition, the Web server can be configured to send LDML format files with other extensions such as .xml or .wml to Lasso Service. It is even possible to configure the Web server to send all .html files to Lasso Service for processing. See the ***Setting Site Preferences*** chapter and the configuration chapters in the Lasso Professional 8 Setup Guide for more information.

In order to promote the portability of your format files between Macintosh, Windows, and UNIX platforms, it is best to name them in a multi-platform friendly fashion. Never use reserved characters such as : ? & / \ # % " ' in file names. Avoid spaces, punctuation, stray periods, and extended ASCII characters. The safest file names contain only letters, numbers, and underscores. Some file systems are case-sensitive. Make sure that all references to a file are specified using the same case as the actual file name on disk. One option is to standardize on lowercase characters for all filenames.

Character Encoding

Lasso uses the standard Unicode byte order mark to determine if a format file is encoded in UTF-8. If no byte order mark is present then the format file will be assumed to be encoded using the Macintosh (or Mac-Roman) character set on Mac OS X or the Latin-1 (or ISO 8859-1) character set on Windows or Linux. Lasso does not support UTF-16 or UTF-32 format files.

Standard text editors such as Bare Bones BBEdit can save files using UTF-8 encoding with the byte order mark included. Consult the manual for the text editor to see how to change the encoding of format files and how to include the proper byte order mark to specify the encoding.

Note: It is recommended to use the Macintosh or Latin-1 character set only for format files that do not contain extended, accented, or foreign characters.

Editing Format Files

Lasso format files can be edited in any text editor. If a format file contains markup from a specific language such as HTML, WML, or XML then it can be edited using an application which is specific to creating that type of file.

In order to make creating and editing Lasso format files which contain HTML easier, OmniPilot supplies a product called Lasso Studio. Lasso Studio provides tag-specific inspectors, wizards, and builders which allow a developer to quickly build Lasso format files within either Macromedia

Dreamweaver, or Adobe GoLive. More information about Lasso Studio is available at the following URL:

http://www.lassostudio.com/

To ease editing of Lasso format files within leading text editors such as Bare Bones BBEdit or Macromedia Home Site consult the Lasso Solutions page at the following URL for links to various third-party solutions:

http://www.lassosolutions.com/

Functional Types

Format files can be classified based on the types of LDML tags they contain or based on the commands they will perform within a Web site. The following list contains terms commonly used to refer to different types of format files. A format file can be classified as being one or more of these types.

- **Pre-Lasso** is used to refer to a format file that contains only command tags within HTML form inputs and URLs. Since Lasso does not perform any substitutions on command tags, these format files do not require any processing by Lasso before they are served to a client. Pre-Lasso format files can be named with a .html file name extension and can even be served from a Web server that does not have a Lasso Web server connector installed.
- **Post-Lasso** format files are the most common type of format files. Post-Lasso format files can contain any combination of tags in square brackets, command tags in HTML form inputs and URLs, and LassoScripts. Post-Lasso format files need to be processed by Lasso Service before they are served to the client. They are usually named with a .lasso file name extension.
- **Library** format files are used to modify Lasso's programming environment by defining new tags and data types, setting up global constants, and performing initialization code. Libraries are included in other format files to modify the environment in which a single format file is processed or loaded at startup to modify the global environment in which all format files are processed.
- **Add Page, Search Page, Update Page, Listing Page, Detail Page** and others are format file names based upon the action which the client will perform when they load the page in their Web browser. For example, a format file might implement the search page of a site. An update page would allow a user to edit a record from a database. A listing page is

usually the result of a search and contains links to a detail page which presents more information about each of the records listed.

- **Add Response, Search Response, Delete Response** and others are format files named based on the action which results in the format file being served to the client. These are typically called response pages. For example, a delete response is served in response to the client opting to delete a record from the database.
- **Error Page, Add Error, Search Error** and others are format files that provide an error message to the client based on the current action.

Action Methods

Web servers and Lasso Service are passive by nature. The software waits until an action is initiated by a client before any processing occurs. Every page load which is processed by Lasso can be thought of as an action with two components: a source and a response. A visitor selects a URL or submits an HTML form within the source format file and receives the response format file. The different types of Lasso actions are summarized in the table below and then described in more detail in the sections that follow.

Table 1: Action Methods

Action Method	Example
URL Action	http://www.example.com/default.lasso
HTML Form Action	<form action="Action.Lasso" method="post"> … </form>
Inline Action	[Inline: -Database='Contacts', …, -Search] … [/Inline]
Scheduled Action	[Event_Schedule: -URL='default.lasso', -Delay='10']
Startup Action	/LassoStartup/startup.lasso

URL Action

A URL action is initiated or called when a client selects a URL in a source file. The source file could be an HTML file from the same Web site, an HTML file from another Web site, the "favorites" of a Web browser, or could be a URL typed directly in a Web browser. The selected URL triggers a designated response file that is processed and returned to the client.

The characteristics of the URL determine the nature of the action which is performed.

- **HTML** – If the URL references a file with a .html file name extension then no processing by Lasso will occur (unless the Web server has been configured to send .html files to Lasso Service.). The referenced HTML file will be returned to the client unchanged from how it is stored on disk.

  ```
  http://www.example.com/default.html
  ```

- **Lasso** – If the URL references a file with a .lasso file name extension then Lasso Service will be called upon to process the file. The referenced format file will be returned to the client after Lasso Service has evaluated all the LDML tags contained within.

  ```
  http://www.example.com/default.lasso
  ```

- **Action.Lasso** – If the URL references Action.Lasso then any command tags contained in the URL will be evaluated and an appropriate response will be returned to the user. The response to an Action.Lasso URL will always be processed by Lasso Service whether it is a .html file, a .lasso file, or a database field.

  ```
  http://www.example.com/Action.Lasso?-Response=default.html
  ```

Note: Lasso will only process files with extensions that have been registered within Lasso Administration. See the ***Setting Site Preferences*** chapter of the Lasso Professional 8 Setup Guide for more information.

HTML Form Action

An HTML form action is initiated or called when a client submits an HTML form in a source file. The source file could be an HTML file from the same Web site or an HTML file from another Web site. The form action and inputs of the form are evaluated and trigger a designated response file that is processed and returned to the client.

The characteristics of the form action determine the nature of the action which is performed.

- **Lasso** – If the HTML form references a file with a .lasso file name extension then Lasso Service will be called upon to process the file. The referenced format file will be returned to the client after Lasso Service has evaluated all the LDML tags contained within the inputs of the form.

  ```
  <form action="default.lasso" method="post">
    …
  </form>
  ```

- **Action.Lasso** – If the HTML form references Action.Lasso then any command tags contained in the inputs in the form will be evaluated and an appropriate response will be returned to the user. The response to an HTML form with an Action.Lasso form action will always be processed by Lasso Service whether it is a .html file, a .lasso file, or a database field.

```
<form action="Action.Lasso" method="post">
  <input type="hidden" name="-Response" value="default.lasso"
  ...
</form>
```

Note: Lasso will only process files with extensions that have been registered within Lasso Administration. See the ***Setting Site Preferences*** chapter of the Lasso Professional 8 Setup Guide for more information.

Inline Action

Inline actions are initiated when the format file in which they are contained is processed by Lasso Service. The result of an inline action is the portion of the format file contained within the [Inline] ... [/Inline] tags that describe the action. As with all Lasso format files, inline actions are processed as the result of a URL being visited or an HTML form being submitted. However, inline actions are not reliant on command tags specified in the URL or HTML form.

- **Inline Tag** – The [Inline] ... [/Inline] container tags can be used to implement an inline action within a format file. The action described in the opening [Inline] tag is performed and the contents of the [Inline] ... [/Inline] tags is processed as a sub-format file specific to that action.

```
[Inline: ... Action Description ...]
  ... Response ...
[/Inline]
```

- **Multiple Inlines** – A single format file can contain many [Inline] ... [/Inline] container tags. Each set of tags is implemented in turn. A single format file can be used to perform many different database actions in different databases as the result of a single URL action or HTML form action.

```
[Inline: ... Action One Description ...]
  ... Response One ...
[/Inline]

[Inline: ... Action Two Description ...]
  ... Response Two ...
[/Inline]
```

- **Nested Inlines** – Inlines can be nested so that the results of one inline action are used to influence the processing of subsequent inline actions. Nested inline actions allow for complex processing to be performed such as copying records from one database to another or summarizing data in a database.

```
[Inline: … Action One Description …]
  [Inline: … Action Two Description …]
    … Combined Response …
  [/Inline]
[/Inline]
```

- **Named Inlines** – Inlines can be processed at the top of a format file and their results can be used later in the format file. This allows the logical processing of an action to be separated from the data formatting. The results of the inline action are retrieved by specifying the inline's name in the [Records] ... [/Records] container tag.

```
[Inline: -InlineName='Action', … Action Description …]
  … Empty …
[/Inline]
…
[Records: -InlineName='Action']
  … Response …
[/Records]
```

Scheduled Action

Scheduled actions are initiated when they are queued using the [Event_Schedule] tag in a source file. The source file could be a format file which is loaded as the result of an action by a client or could be loaded as a startup action. The response to the scheduled action is not processed until the designated date and time for the action is reached.

Any type of format file can be called as a scheduled action, but the results will not be stored. Scheduled format files can effectively be thought of as pure LDML format files. Scheduled format files can use logging or email messages to notify a client that the action has occurred. See the *Control Tags* chapter for more information.

- **Lasso** – The URL referenced when the action is scheduled will usually contain a .lasso file name extension. The referenced format file will be processed when the designated date and time is reached, but the results will not be returned to any client. For example, the following [Event_Schedule] tag schedules a call to a page that will send an email report to the administrator of the site every 24 hours (1440 minutes), even after server restarts:

```
[Event_Schedule: -URL='http://www.example.com/admin/emailreport.lasso',
  -Delay='1440', -Repeat=True, -Restart=True]
```

Startup Action

Startup actions are initiated when Lasso Service is launched by placing format files in the LassoStartup folder. Format files which are processed at startup are library files which are used to set up the global environment in which all other pages will be processed. For example, they can add tags and custom data types to the global environment, set up global constants, or queue scheduled actions.

- **Lasso** – Format files with .lasso file name extensions are used at startup to queue scheduled actions or perform routine tasks on the databases or files managed by Lasso Service. Any format files in the LassoStartup folder will be processed every time Lasso Service is launched.
- **Library** – Libraries of LDML tags and custom data types can be processed at startup in order to extend the global environment in which all other pages are processed. All LDML tags and data types in a library processed at startup will be available to all other format files processed by Lasso Service. See the *Files and Logging* chapter for more information about libraries.

Securing Format Files

The information being collected or served in a Web site is often of a sensitive nature. Credit card numbers and visitor's personal information must be kept secure. Proper format file security is the first step toward creating a Web site which only provides the information you want it to publish.

The LDML code contained in a format file should be secured so visitors cannot examine it. Format files contain information about how to access your databases. They may contain passwords, table and field names, or custom calculations.

LDML code in a format file is implicitly secured if it is stored in a format file with a .lasso file extension. The code in the file will always be processed by Lasso before it is served to visitors. Visitors can access the HTML source of the file they receive, but cannot access the LDML source of the original format file.

It is important to ensure that your format files cannot be accessed unsecurely through other Internet technologies such as FTP, Telnet, or file sharing. Make sure that the files in your Web serving folder can only be accessed by trusted developers and administrators. See the *Setting Up*

Security chapter in the Lasso Professional 8 Setup Guide for more information.

Output Formats

Although Lasso format files are always text files, they can be used to output a wide variety of different data formats. The most basic format files match the output format. For example, HTML format files are used to return HTML output to Web browsers. But, pure LDML format files can be used to return data in almost any format through the use of the [Include] tag and data from database fields.

This section describes how to output the most common data formats from Lasso format files.

Text Formats

Lasso can be used to output any text-based data format. Format files are usually based on a file of the desired type. The following are common output formats:

- **HTML** is the most common output format. Usually, HTML output is generated from HTML format files. The embedded LDML tags are processed, altering and adding to the content of the file, but the essential characteristics of the file remain unchanged.
- **XML** is rapidly becoming a standard for data exchange on the Internet. XML output is usually generated through Lasso by processing XML format files. The embedded LDML tags are processed, altering and adding content to the XML data in the file. The resulting XML data can be made to conform to any Document Type Definition (DTD) or XML Schema desired.
- **WML** is the language used to communicate with WAP-enabled wireless devices. WML is a language which is based on XML. It is an example of a DTD or XML Schema to which output data must conform. Lasso usually generates WML output by processing WML format files. Developers can create WML format files by using a WML authoring tool and then embedding LDML tags within.
- **PDF** or Portable Document Format is Adobe's machine-independent format for distribution of electronic documents. Lasso can be used in concert with PDFs in several ways. Lasso can be used to process forms embedded within PDF files and to return results to a client. Lasso can be used to generate ASCII PDFs through custom programming. Finally,

Lasso can be used to provide access control to PDFs so only authorized users are able to download certain PDFs.

Binary Formats

Lasso can be used to output a variety of binary data formats. Generally, Lasso is not used to perform any processing on the binary data being served, but is just a conduit through which pre-existing binary data is routed. See the *Images and Multimedia* chapter for more information about each of these methods. The following list describes common methods of outputting binary data.

- **URLs** can be created and manipulated using LDML. For example, a database could contain a file name in a field. LDML can be used to convert that file name into a valid URL which will then be served as part of an HTML page. The binary data will be fetched from the client directly without any further action by Lasso.
- **Database Fields** can be used to store binary data such as image files in a container or binary format. If a Lasso data source connector for the appropriate database supports fetching binary data, then Lasso can serve the binary data or image files directly from the database field using the [Field], [Image_URL] or -Image tags.
- **Binary Files** can be served through Lasso using a combination of the [Include_Raw] tag to output the binary data and the [Content_Type] tag to report to the client what type of data is being served.

File Management

Lasso 8 introduces a new distributed architecture. Lasso Service can be installed on one machine and a Lasso Web server connector can be installed into a Web server on a different machine. It is important to realize where format files are stored so they can be located on the appropriate machine.

Note: In most Lasso 8 installations Lasso Service and a Lasso Web server connector will be installed on the same machine. The discussion below still applies since the various components of Lasso 8 will operate out of different folders. An administrator can set up a machine so the same files are shared by all components of Lasso.

Lasso Web Server Connector

Most format files for a Web site will be stored on the same machine as a Lasso Web server connector in the Web serving folder which contains the HTML and image files for the Web site.

- **Client Format Files** are stored alongside the HTML and image files which comprise a Web site. To the client, these format files appear no different from plain HTML files except that they contain dynamic data.
- **Included Files** are stored in the Web serving folder. These are files which are incorporated into format files using the [Include] and [Include_Raw] tags. Included files could be other format files, plain HTML files, images files, PDF files, etc.
- **Library Files** can be stored in the Web serving folder. These files contain definitions for LDML tags and data types. Library files are referenced much like included files. The custom tags and data types defined in the library file are available only in the pages which load the library file.
- **Administrative Files** are stored in the Web serving folder in a folder named Lasso. These files comprise the Web-based administration interface for Lasso Service.

Lasso Service

Format files which are stored on the same machine as Lasso Service are used primarily when Lasso Service starts up to set up the global environment. However, other files which are manipulated by Lasso's logging and file tags are also stored on the Lasso Service machine.

- **Startup Format Files** are stored in the LassoStartup folder with Lasso Service. These files are processed when Lasso Service is launched and can perform routine tasks or modify the global environment in which all other Lasso format files will be processed. Any LDML tags, data types, or global constants defined in these libraries will be available to all pages which are processed by Lasso Service.
- **Startup LassoApps** are stored in the LassoStartup folder with Lasso Service. The default page of each LassoApp is processed at startup and the LassoApp is pre-loaded into memory for fast serving.
- **Log Files** are created using the [Log] tag. These files can be used to store information about the format files which have been processed by Lasso Service. Log files are created on the same machine as Lasso Service.
- **Uploaded Files** are stored in a temporary location in a folder with Lasso Service. Files can be uploaded by a client using a standard HTML file input. Uploaded files must be moved from their temporary location

to a permanent folder before the page on which they were uploaded finishes processing.

- **File Tags** operate on files in folders on the same machine as Lasso Service. The file tags can be used to manipulate log files or uploaded files. The file tags are also used to manipulate HTML and other format files in the Web serving folder if Lasso Service is installed on the same machine as a Lasso Web server connector or if file sharing between the two machines facilitates accessing the files as a remote volume. See the *Files and Logging* chapter for more information.

  Note: A user can only access files to which the group they belong has been granted access . See the ***Setting Up Security*** chapter in the Lasso Professional 8 Setup Guide for more information.

Database

Format files can be stored in any database which is available to Lasso Service. They can be stored in the local Lasso MySQL database or in a remote database hosted on another machine.

- **Format Files** stored in database fields can be included in a page using the [Process] tag. In the following example, the field LDML_Template is processed using the [Process] tag:

  ```
  [Process: (Field: 'LDML_Template')]
  ```

  Database fields can also be referenced through appropriate URL or HTML form parameters. See the ***Setting Up Data Sources*** chapter in the Lasso Professional 8 Setup Guide for more information about granting permission to use a field as a format file. In the following example, the field LDML_Template is used to format the response to the URL:

  ```
  http://www.example.com/Action.Lasso?-Response=Field:LDML_Template
  ```

Specifying Paths

Format files can be referenced in many different ways depending on how they are being used. They can be referenced in any of the following ways:

- A URL can be used to reference a format file with a .lasso file extension directly:

  ```
  http://www.example.com/default.lasso
  ```

- A URL can be used to reference format files with any file extensions by calling Action.Lasso and then specifying the format file in a -Response command tag:

```
http://www.example.com/Action.Lasso?-Response=default.html
```

- An HTML form can be used to reference a format file with a .lasso file extension directly in the form action:

```
<form action="default.lasso" method="post">
  …
</form>
```

- An HTML form can be used to reference format files with any file extensions by calling Action.Lasso as the form action and then specifying the format file in a -Response hidden input:

```
<form action="Action.Lasso" method="post">
  <input type="hidden" name="-Response" value="default.html">
  …
</form>
```

- A format file can be referenced from within certain LDML tags. For instance, the [Include] tag takes a single format file name as a parameter:

```
[Include: 'default.lasso']
```

Paths are specified for format files differently depending on what type of format file contains the path designation and to which type of format file is being referred.

Note: Lasso cannot be used to reference files outside of the Web server root unless specific permission has been granted within Lasso Administration. See the ***Setting Up Security*** chapter in the Lasso Professional 8 Setup Guide for more information.

Relative and Absolute Paths

Most paths in Lasso format files follow the same rules as the paths between HTML files served by the Web server. Relative and absolute paths are interpreted either by the client's Web browser or by Lasso Service. These paths are all defined within the context of the Web serving folder established by the Web server which is hosting a Lasso Web server connector. If a single Web server is used to host multiple sites, the Web serving folder could be different for each virtual host.

- **Relative Paths** between files can be specified using all the rules and features of URL file paths. For example, the following anchor tag designates a response in the same folder as the current page:

```
<a href="response.lasso">Response</a>
```

- Paths can use ../ to specify a higher level folder. The following anchor tag designates a response in the folder one level higher than that which contains the current page:

```
<a href="../response.lasso">Response</a>
```

- Relative paths designated within LDML tags follow the same basic rules except that ../ cannot be used to access the parent folder for a format file. For example, the following [Include] tag includes a file from the same folder as the current page.

```
[Include: 'include.lasso']
```

- **Absolute Paths** are referenced from the root of the Web serving folder as designated by the Web serving software. The Web server root is specified using the / character. The following anchor tag designates a response file contained at the root level of the current Web site:

```
<a href="/response.lasso">Response</a>
```

- Absolute paths designated within LDML tags work the same as absolute paths in URLs. The following [Include] tag includes a file contained at the root level of the current Web site.

```
[Include: '/include.lasso']
```

For more information about specifying relative and absolute paths, consult your favorite HTML reference or the documentation for your Web serving application.

Action.Lasso Paths

If a format file has been called using Action.Lasso in either a URL or in an HTML form action then all paths within the format file will be evaluated relative to the stated location of Action.Lasso.

- Action.Lasso could be specified as Action.Lasso so it appears to be located in the same folder as the calling format file. All paths must then be specified as if the referenced format file was located in the same folder as the calling format files. Paths relative to the referenced format file will fail, but paths relative to the calling format file will succeed.

```
<a href="Action.Lasso?-Database=Contacts& ... ">Response</a>
```

- Action.Lasso could be specified as /Action.Lasso so it appears to be located at the root of the Web serving folder. All paths must then be specified as if the referenced format file was located at the root of the Web serving folder. Paths relative to the referenced format file will fail.

```
<a href="/Action.Lasso?-Database=Contacts& ... ">Response</a>
```

- Action.Lasso can also be specified using an arbitrary path such as /Folder/Action.Lasso. In this case all paths will be relative to the specified location of Action.Lasso.

```
<a href="/Folder/Action.Lasso?-Database=Contacts& ... ">Response</a>
```

Database Field Paths

The path to database fields which are going to be used as format files is a special case since these files are not contained on the local disk of the Web serving machine.

- In a URL, a field named Example_Template can be referenced as follows. Usually, Action.Lasso is used as the target of a URL and the field is specified in a -Response command tag. The URL must contain a valid database action that returns a record from which the field will be used. The following example searches for a person from the Contacts database whose ID is 1. The value of Example_Template for that person is used as the response format file.

```
http://www.example.com/Action.Lasso?-Database=Contacts&-Table=People&
  -KeyField=ID&-KeyValue=1&-Search&-Response=Field:Example_Template
```

- In an HTML form, a field named ExampleTemplate can be referenced as follows. Usually, Action.Lasso is used as the form action and the field is specified in a hidden input using a -Response command tag. This form uses the same database action defined in the URL above.

```
<form action="Action.Lasso" method="post">
  <input type="hidden" name="-Search" value="">
  <input type="hidden" name="-Database" value="Contacts">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-KeyField" value="ID">
  <input type="hidden" name="-KeyValue" value="1">
  <input type="hidden" name="-Response" value="Field:Example_Template ">
</form>
```

Note: Permission must be granted in Lasso Administration for a field to be used as a response field. See the *Setting Up Data Sources* chapter in the Lasso Professional 8 Setup Guide for more information.

Lasso Service Paths

Paths to format files on the machine hosting Lasso Service are specified differently than those which are used in format files on the machine hosting a Lasso Web server connector. Format files on the machine hosting Lasso Service are usually only referenced by the file tags and log tag.

- Most paths should be **Fully Qualified Paths** specified from the root of the disk on which Lasso Service is installed. For example, the following path would represent a file in the same folder as Lasso Service in a typical install on a Windows 2000 machine:

```
C://Program Files/OmniPilot Software/Lasso Professional 8/default.lasso
```

- The following path would represent the same file if it were in the same folder as Lasso Service in a typical install on a Mac OS X machine:

  ///Applications/Lasso Professional 8/default.lasso

  In Mac OS X, the hard drive name is set to a slash / so the fully qualified paths must start with three slashes ///. Paths starting with a single slash / are defined to be relative to the Web server root.

For more information about specifying fully qualified paths, consult the *Files and Logging* chapter.

Note: Fully qualified paths can also be specified in a platform specific fashion. For example, the path above could be written as C:\\Program Files\OmniPilot Software\Lasso Professional 8\default.lasso on Windows or as Applications:Lasso Professional 8:default.lasso on Macintosh.

Format File Execution Time Limit

Lasso includes a limit on the length of time that a format file will be allowed to execute. This limit can help prevent errors or crashes caused by infinite loops or other common coding mistakes. If a format file runs for longer than the time limit then it is killed and a critical error is returned and logged.

The execution time limit is set to 10 minutes (600 seconds) by default and can be modified or turned off in the *Setup > Global > Settings* section of Lasso Admin. The execution time limit cannot be set below 60 seconds.

The limit can be overridden on a case by case basis by including the [Lasso_ExecutionTimeLimit] tag at the top of a format file. This tag can set the time limit higher or lower for the current page allowing it to exceed the default time limit. Using [Lasso_ExecutionTimeLimit: 0] will deactivate the time limit for the current format file altogether.

On servers where the time limit should be strictly enforced, access to the [Lasso_ExecutionTimeLimit] tag can be restricted in the *Setup > Global > Tags* and *Security > Groups > Tags* sections of Lasso Admin.

Asynchronous tags and compound expressions are not affected by the execution time limit. These processes run in a separate thread from the main format file execution.

Note: When the execution time limit is exceeded the thread that is processing the current format file will be killed. If there are any outstanding database requests or network connections open there is a potential for some memory to be leaked. The offending page should be reprogrammed to run faster or exempted from the time limit using [Lasso_ExecutionTimeLimit: 0]. Restarting Lasso Service will reclaim any lost memory.

4

Chapter 4
LDML 8 Syntax

Lasso Professional 8 supports several different syntax styles and methods of embedding LDML code within a format file.

- *Overview* provides an introduction to the different syntax styles available in Lasso Professional 8.
- *Colon Syntax* describes the traditional syntax of LDML which features a tag name followed by a colon and the parameters of the tag [Tag_Name: Parameters].
- *Parentheses Syntax* describes a new syntax style in Lasso Professional 8 which features a tag name followed by parentheses that include the parameters of the tag [Tag_Name(Parameters)].
- *Square Brackets* describes how to embed LDML tags in HTML format files surrounded by square brackets [...].
- *LassoScript* describes how to embed LDML tags in format files in a block of LassoScript <?LassoScript ... ?>.
- *HTML Form Inputs* describes how to embed LDML tags within HTML forms.
- *URLs* describes how to embed LDML tags within URLs.
- *Compound Expressions* describes how to embed LDML tags within other LDML expressions using braces { ... }.

Overview

Lasso Professional 8 offers a great deal of flexibility in how LDML code can be written and embedded within format files. This allows the developer to select the best syntax style for each programming task. The different syntax styles and embedding methods are completely interchangeable and a combination of different styles can be used throughout a single format file.

Syntax Styles

Lasso Professional 8 offers two different syntax styles. Colon syntax is the traditional syntax style of Lasso. Parentheses syntax is a new syntax style that promotes better coding style by removing ambiguities from the parser. Parentheses syntax may be an easier transition for developers who are familiar with other programming languages while colon syntax may be preferred by experienced Lasso developers.

- **Colon Syntax** – A tag name is followed by a colon : and then the parameters of the tag.

  ```
  Tag_Name: Parameters
  ```

  A tag which is used as a parameter to another tag should always be surrounded by parentheses since this ensures that Lasso associates the parameters with the proper tag.

  ```
  Tag_Name: (Sub_Tag: Parameters), More Parameters
  ```

- **Parentheses Syntax** – A tag name is followed by parentheses (...) which contain parameters of the tag.

  ```
  Tag_Name(Parameters)
  ```

  A tag which is used as a parameter to another tag can be written the same as if it were at the top-level since the parameters are automatically associated with the tag.

  ```
  Tag_Name(Sub_Tag(Parameters), More Parameters)
  ```

The two syntax methods can be mixed even within the same expression although it is generally recommended that one syntax style be used within any single block of LDML code.

```
Tag_Name((Sub_Tag: Parameters, Sub_Tag(Parameters)), More Parameters)
```

Note: The introduction of parentheses syntax does not mean that colon syntax is de-emphasized or deprecated in any way. Both syntax styles are equally supported in Lasso Professional 8.

Embedding Methods

Lasso Professional 8 offers three different embedding methods. The method to use depends on the particular needs and outputs for a given block of code. The three different methods can be used alternately throughout a single format file.

- **Square Brackets** – Each tag is surrounded by square brackets. The entire bracketed expression is replaced by the value of the tag. Square brackets are most appropriate when embedding LDML tags within HTML or other markup languages.

```
<a href="[Field('URL')]">[Field('Link_Name')]</a>
```

- **LassoScript** – An entire block of LDML code is surrounded by a single <?LassoScript ... ?> container. Each tag must end with a semi-colon. The entire LassoScript is replaced by the value of all the tags within concatenated together, but without any inter-tag whitespace. LassoScript is most appropriate when writing a large block of LDML code that does not have any output or that has highly structured output.

```
<?LassoScript
  Tag_Name(Parameters);
  Tag_Name(Parameters);
?>
```

Note: These two embedding methods are completely interchangeable. A single tag can be embedded within a LassoScript container as <?LassoScript Tag_Name(Parameters) ?> or multiple tags can be embedded within square brackets as [Tag_Name(Parameters); Tag_Name(Parameters)].

- **HTML Form Inputs** – LDML tags can be placed in HTML forms. When the form is submitted the command tags will be interpreted before the response format file is processed by Lasso.

```
<form action="Action.Lasso" method="POST">
  <input type="hidden" name="-Response" value="format.lasso" />
  <input type="submit" name="-Token.Action" value="Submit!" />
</form>
```

- **URLs** – LDML tags can be placed in URLs. When the URL is entered in a Web client the command tags will be interpreted before the response format file is processed by Lasso.

```
http://www.example.com/format.lasso?-Token.Action=Submit
```

Note: Classic Lasso syntax in which complete database operations are performed through HTML form inputs or URLs has been deprecated. It's use is no longer recommended. However, there are still some command tags that work even with Classic Lasso support deactivated.

- **Compound Expressions** – A block of LDML code can be used as a parameter to an LDML tag. This allows a whole series of tags to be executed and the result to be returned as the parameter value.

```
Tag_Name({If(Condition); Return(True); /If; Return(False);}->Eval())
```

Colon Syntax

Colon syntax style is so-named because it features a tag name followed by a colon : and then the parameters of the tag. Colon syntax is the traditional syntax of Lasso Professional. Colon syntax is fully supported within Lasso Professional 8.

Table 1: Colon Syntax Delimiters

Delimiter	Description
:	Separates a tag name from its parameters. Required for tags which have parameters.
(...)	Used to surround tags which have parameters and are used as parameters to another tag.
,	Used to separate parameters.

A simple tag with parameters in colon syntax is specified as follows:

```
Tag_Name: Parameters
```

A tag which is used as a parameter to another tag should always be surrounded by parentheses since this ensures that Lasso associates the parameters with the proper tag.

```
Tag_Name: (Sub_Tag: Parameters), More Parameters
```

A tag which does not require any parameters must be written without a colon: It is not valid to have a colon after a tag name without any parameters.

```
Simple_Tag
```

A tag which does not require any parameters can be used as a parameter to another tag with or without surrounding parentheses.

```
Tag_Name: (Simple_Tag)
```

```
Tag_Name: Simple_Tag
```

Be careful to avoid ambiguities when specifying tags and parameters. In the following example [Tag_Name] is being passed [Sub_Tag] as a parameter. It is not clear whether Parameter_3 is intended to be a parameter for [Tag_Name] or [Sub_Tag].

```
Tag_Name: Parameter_1, Sub_Tag: Parameter_2, Parameter_3
```

This code will actually be interpreted as follows by Lasso. The outermost tag is greedy and will claim all the parameters it can for itself. This leaves only Parameter_2 being passed to [Sub_Tag].

```
Tag_Name: Parameter_1, (Sub_Tag: Parameter_2), Parameter_3
```

In order to pass Parameter_2 and Parameter_3 to [Sub_Tag] the following syntax must be used.

```
Tag_Name: Parameter_1, (Sub_Tag: Parameter_2, Parameter_3)
```

Note: In early versions of Lasso the colon could be replaced by a comma. This is not allowed in Lasso Professional 8.

Parentheses Syntax

Parentheses syntax style is so-named because it features a tag name followed by parentheses which surround the parameters of the tag. Parentheses syntax is new in Lasso Professional 8. The advantages of parentheses syntax include:

- Parentheses syntax is less ambiguous than colon syntax since parameters are always clearly associated with one tag without knowledge of the parser's internal rules.
- Parentheses syntax is closer to the syntax of programming languages like JavaScript. Developers who are already familiar with other programming languages should feel right at home in parentheses syntax.

Table 2: Parentheses Syntax Delimiters

Delimiter	Description
(…)	Used immediately following a tag name to contain the parameters of the tag. Can be left empty if a tag has no parameters.
,	Used to separate parameters.

A simple tag with parameters in parentheses syntax is specified as follows:

```
Tag_Name(Parameters)
```

A tag which is used as a parameter to another tag can be specified the same as if it were at the top-level of the expression. The required parentheses in this syntax style ensure that there are no ambiguities.

```
Tag_Name(Sub_Tag(Parameters), More Parameters)
```

A tag which does not require any parameters must be written trailing parentheses. However, the trailing parentheses can also be included without any ill effects.

```
Simple_Tag
```

```
Simple_Tag()
```

A tag which does not require any parameters can be used as a parameter to another tag with or without trailing parentheses.

```
Tag_Name:(Simple_Tag)
```

```
Tag_Name(Simple_Tag())
```

It is impossible to introduce ambiguities when using this syntax. The examples below are the same as those from the *Colon Syntax* section, but since the parentheses are required there is no question how the parser will interpret these expressions.

```
Tag_Name(Parameter_1, Sub_Tag(Parameter_2), Parameter_3)
```

```
Tag_Name(Parameter_1, Sub_Tag(Parameter_2, Parameter_3))
```

Square Brackets

Square brackets allow LDML tags to be used as a tag-based markup language. Square brackets are most convenient when embedding LDML tags within HTML, XML, or another markup language. Square brackets have the following advantages:

- Brackets visually distinguish LDML code from the angle bracket delimited markup languages in which they are embedded.
- White space between tags is preserved and output to the site visitor.

LDML tags within square brackets are each executed in turn. The entire bracketed expressions is replaced by the value of the processed tag. The returned value is encoded using HTML encoding by default. This can be changed using an encoding keyword or [Encode_Set] ... [/Encode_Set] tags.

Table 3: Square Bracket Delimiters

Delimiter	Description
[	Starts a square bracket tag. Required..
]	Ends a square bracket tag. Required.

To create a square bracket tag:

Place the tag within square brackets. Either colon or parentheses syntax can be used.

```
[Field: 'Field_Name', -EncodeNone] ➜ John

[Math_Add:(, 2, 3, 4, 5)] ➜ 15
```

To use container tags with square brackets:

Container tags are specified surrounding a portion of a page. The opening tag is written normally with parameters. The closing tag has the same name as the opening tag, but preceded with a forward slash /. Indentation is often used to make the contents of the container tag clear, but is not required. In the following example a [Loop] tag is used to output the numbers 1 through 5 using the [Loop_Count] tag.

```
[Loop(5)]
  [Loop_Count]
[/Loop]

➜ 1 2 3 4 5
```

To specify a comment within square brackets:

Specify a comment using the /* ... */ delimiter. All text between these delimiters will not be processed.

```
[Field: 'First Name' /* This is a comment */]
```

To suppress output from a square bracket tag:

- Use the [Null] tag to suppress output from a single square bracket tag.

  ```
  [Null: Field('First Name')]
  ```

- Use the [Output_None] ... [/Output_None] tags around a block of square bracket tags. In the following example, the expression will return no value even though it contains several [Field] tags.

  ```
  [Output_None]
    <br />[Field('First Name')] [Field('Last Name')]
  [/Output_None]
  ```

To change the encoding for a square bracket tag:

- Use an encoding keyword on the tag. By default all tags are encoded using -EncodeHTML.

  ```
  [Field('First_Name', -EncodeNone)]
  ```

- Use an [Encode_...] tag to specify the encoding explicitly.

  ```
  [Encode_HTML: Field('First_Name')]
  ```

- Use the [Encode_Set] ... [/Encode_Set] tags around the square bracket tags. This changes the default encoding without use of the -EncodeNone keyword in each tag.

```
[Encode_Set: -EncodeNone]
  [Output: '<p>This HTML code will render<br>with breaks.']
[/Encode_Set]

➜ <p>This HTML code will render
<br>with breaks.
```

To prevent square brackets from being interpreted:

Sometimes it is desirable to have square brackets in a format file which are not interpreted by Lasso. This can be useful for including LDML samples on a page, for using array references within JavaScript, or simply for typographic design flexibility. These methods work for either square bracket or LassoScript syntax.

- Surround the code that should not be processed with [NoProcess] ... [/NoProcess] tags. Lasso will not interpret any code within this container.

```
[NoProcess]
  The [Field] tag returns the value of a database field for the current record.
[/NoProcess]

➜ The [Field] tag returns the value of a database field for the current record.
```

- Surround the code that should not be processed with HTML comments <!-- ... -->. This method is particularly useful for JavaScript code blocks.

```
<script langauge="JavaScript">
  <!--
    array[1] = array[2];
  // -->
</script>
```

LassoScript

LassoScript allows LDML tags and symbols to be used as a scripting language in a fashion which is complementary with the traditional use of LDML as a tag-based markup language. LassoScripts have the following advantages:

- Concise format allows better formatting of long code segments.
- Represented as a single object in many visual authoring environments. This makes it easy to separate the logic of a page from the presentation or to hide implementation details from Web designers who are working on the visual aspects of the page.
- Comments allow code to be self-documented for better maintainability.
- Compatible with HTML and XML standards for embedding server-side scripting commands.
- Provides scripting-like method of coding for programmers who prefer this method.

LDML tags contained within a LassoScript execute exactly as they would if they were specified within square brackets. The value returned by a LassoScript is the concatenation of all the values which are returned from the tags that make up the LassoScript. No encoding is applied to the output of a LassoScript, but normal encoding rules apply to each of the tags within a LassoScript that outputs values.

LassoScripts begin with <?LassoScript and end with ?>. LDML tags within a LassoScript are delimited by a single semi-colon ; at the end of the tag rather than by square brackets. White space within a LassoScript is ignored. Comments begin with a double forward slash // and continue to the end of the line. To continue a comment on another line, another // must be used. All text in a LassoScript must be part of a tag or part of a comment, no extraneous text is allowed.

Values returned from expressions within a LassoScript are not encoded by default. The [Encode_...] tags can be used to apply explicit encoding to values output from a LassoScript.

```
<?LassoScript
  Encode_HTML: '<br>This is the output from the LassoScript.';
?>
```

➜
This is the output from the LassoScript.

Table 4: LassoScript Delimiters

Delimiter	Description
<?LassoScript	Starts a LassoScript. Required.
?>	Ends a LassoScript. Required.
;	Ends an LDML tag. Required.
//	Comment. All text to the end of the line will be ignored.
/* … */	Block Comment. All text between the delimiters will be ignored. Allows multi-line comments.

Note: Square brackets […] are not allowed in LassoScripts. However, it is possible to replace the <?LassoScript … ?> delimiters with square brackets […].

To create a LassoScript with a single tag:

LassoScripts can be used for individual LDML tags as well as for groups of tags. When only a single tag is specified, the semi-colon at the end of the tag is optional. The container <?LassoScript … ?> can be substituted for the square bracket container […] if necessary.

```
<?LassoScript Field: 'Field_Name', -EncodeNone ?>

<?LassoScript Math_Add: 1, 2, 3, 4, 5 ?> ➜ 15
```

To use container tags within a LassoScript:

Container tags are specified just as they are in square-bracketed LDML. The opening container tag must end with a semi-colon. The closing container tag should start with a forward slash / and end with a semi-colon. Indentation is usually used to make the contents of the container tag clear, but is not required. In the following example a [Loop] tag is used to output the numbers 1 through 5 using the [Loop_Count] tag.

```
<?LassoScript
  Loop: 5;
    Loop_Count + ' ';
  /Loop;
?>

➜ 1 2 3 4 5
```

To use container tags between LassoScripts:

Container tags can be opened within one LassoScript then closed in a subsequent LassoScript. The following example shows a mixture of LassoScript and square bracket syntax which implements a loop.

```
<?LassoScript
  Loop: 5;
?>

  [Loop_Count]

<?LassoScript
  /Loop;
?>
```

➜ 1 2 3 4 5

To specify a comment within a LassoScript:

Use the // symbol to start a comment. All text until the end of the line will be part of the comment and will not be executed by the LassoScript.

```
<?LassoScript
  // This LassoScript only contains a comment.
?>

<?LassoScript
  // The following line has been commented out. It will not be processed.
  // Encode_HTML: 'Testing';
?>
```

Alternately, specify a multi-line comment using the /* ... */ delimiter. All text between these delimiters will not be processed.

```
<?LassoScript
  /*
    These lines have been commented out. The following line will not be processed.
    Encode_HTML: 'Testing';
  */
?>
```

To suppress output from a LassoScript:

Use the [Output_None] ... [/Output_None] tags around the LassoScript. In the following example, the LassoScript will return no value even though it contains several expressions that output values.

```
<?LassoScript
  Output_None;
    'This value will not be seen.';
    'Neither will this value.';
  /Output_None;
?>
```

To change the encoding for a LassoScript:

Use the [Encode_Set] ... [/Encode_Set] tags around the LassoScript. In the following example, the LassoScript will not perform any encoding so HTML values can output from the [Output] tags without use of the -EncodeNone keyword in each tag.

```
<?LassoScript
  Encode_Set: -EncodeNone;
    Output: '<p>This HTML code will render<br>with breaks.';
  /Encode_Set;
?>
```

```
➜ <p>This HTML code will render
<br>with breaks.
```

To use square brackets to surround a LassoScript:

The <?LassoScript ... ?> delimiters can be replaced by square brackets [...]. The following is a valid LassoScript.

```
[
  Encode_HTML: '<br>This is the output from the LassoScript.';
]
```

```
➜ &lt;br&gt;This is the output from the LassoScript.
```

To convert LDML square bracket code to a LassoScript:

1. Format the code so each tag is on a separate line.
2. Remove all opening square brackets [.
3. Replace all closing square brackets] with semi-colons ;.
4. Correct the indentation so tags inside container tags are indented.
5. Add <?LassoScript and ?> to the beginning and end of the code.

In the following example the same code is shown in square bracketed LDML code and then as an equivalent LassoScript.

```
[Loop: 5]
  [Loop_Count]
[/Loop]
```

```
<?LassoScript
  Loop: 5;
    Loop_Count + ' ';
  /Loop;
?>
```

HTML Form Inputs

LDML tags can be embedded within HTML form inputs in two different ways. An LDML command tag can be embedded as the name parameter of an <input>, <select>, or <textarea> tag. LDML tags in square brackets can be embedded as either the name or value parameters. For example, the following <input> tag includes an LDML command tag -ResponseAnyError as the name parameter and an LDML substitution tag [Response_FilePath] as the value parameter.

```
<input type="hidden" name="-ResponseAnyError" value="[Response_FilePath]">
```

When the format file that includes the -ResponseAnyError tag is served to a client, the -ResponseAnyError tag will not be processed until the HTML form in which this <input> is embedded is submitted by a client. However, the [Response_FilePath] substitution tag is replaced by the name of the current Web page to yield the following HTML for the <input> tag.

```
➜ <input type="hidden" name="-ResponseAnyError" value="/form.lasso">
```

Any of the various tag types can be embedded within HTML form inputs, but the details differ for each type of tag. See the section on *Tag Types* below for more details.

URLs

LDML tags can be embedded within the parameters of URLs in two different ways. An LDML command tag can be embedded as the name half of a parameter. LDML tags in square brackets can be embedded as either the name or value half of a parameter. For example, the following URL includes an LDML command tag -Token.Name as the name half of the first parameter and an LDML substitution tag [Client_Username] as the value half of the first parameter.

```
<a href="http://www.example.com/default.lasso?-Token.Name=[Client_Username]">
```

When the format file that includes this tag is served to a client the -Token.Name command tag will remain unchanged. This tag will not be processed until the URL is selected by a client. The [Client_Username] substitution tag will be replaced by the name of the current user logged in.

```
➜ <a href="http://www.example.com/default.lasso?-Token.Name=Administrator">
```

Any of the various tag types can be embedded within URLs, but the details differ for each type of tag. See the section on *Tag Types* below for more details.

Compound Expressions

Compound expressions allow for tags to be created within LDML code and executed immediately. Compound expressions can be used to process brief snippets of LDML code inline within another tag's parameters or can be used to create reusable code blocks.

Table 5: Compound Expression Delimiters

Delimiter	Description
{	Starts a compound expression. Required.
}	Ends a compound expressions. Required.
;	Ends an LDML tag. Required.

Evaluating Compound Expressions

A compound expression is defined within curly braces { }. The syntax within the curly braces should match that for LassoScripts using semi-colons between each LDML tag. For example, a simple compound expression that adds 6 to a variable myVariable would be written as follows. The expression can reference page variables.

```
[Variable: 'myExpression' = { $myVariable += 6; }]
```

The compound expression will not run until it is asked to execute using the [Tag->Eval] tag. The expression defined above can be executed as follows.

```
[Variable: 'myVariable' = 5]
[$myExpression->Eval]
[Variable: 'myVariable']
```

➜ 11

A compound expression returns values using the [Return] tag just like a custom tag. A variation of the expression above that simply returns the result of adding 6 to the variable, without modifying the original variable could be written as follows.

```
[Variable: 'myExpression' = { Return: ($myVariable + 6); }]
```

This expression can then be called using the [Tag->Eval] tag and the result of that tag will be the result of the stored calculation.

```
[Variable: 'myVariable' = 5]
[$myExpression->Eval]
```

➜ 11

Alternately, the expression can be defined and called immediately. For example, the following expression checks the value of a variable myTest and returns Yes if it is True or No if it is False. Since the expression is created and called immediately using the [Tag->Eval] tag it cannot be called again.

```
[Variable: 'myTest'= True]
[Encode_HTML: { If: $myTest; Return: 'Yes'; Else; Return: 'No'; /If; }->Eval]
```

➜ Yes

Running Compound Expressions

The same conventions for custom tags may be used within a compound expression provided it is executed using the [Tag->Run] tag. Compound expressions which are run can access the [Params] array and define local variables.

For example, the following expression accepts a single parameter and returns the value of that parameter multiplied by itself. The expression is formatted similar to a LassoScript using indentation to make the flow of logic clear.

```
[Variable: 'myExpression' = {
  Local: 'myValue' = (Params->(Get: 1));
  Return: #myValue * #myValue;
}]
```

This expression can be used as a tag by calling it with the [Tag->Run] tag with an appropriate parameter. The following example calls the stored tag with a parameter of 5.

```
[Encode_HTML: $myExpression->(Run: -Params=(Array: 5))]
```

➜ 25

5

Chapter 5
LDML 8 Tag Language

This chapter introduces the methodology behind programming data-driven Web sites powered by Lasso 8. This chapter introduces terminology which is used through the remainder of this language guide. All new users of Lasso Professional 8 should read through this chapter to familiarize themselves with the structure of Lasso Dynamic Markup Language (LDML).

- *Introduction* describes the layout of this chapter in detail.
- *Tag Types* introduces the five types of LDML 8 tags including substitution tags, process tags, container tags, member tags, and command tags.
- *Tag Categories and Naming* introduces the logic behind the names of LDML 8 tags.
- *Parameter Types* describes the different types of parameters that can be specified within a tag.
- *Encoding* contains a discussion of character encoding features for substitution tags.
- *Data Types* describes the different data types which LDML 8 offers.
- *Expressions and Symbols* introduces the concept of performing calculations directly within parameters.
- *Delimiters* includes a technical description of the characters used to delimit LDML 8 tags in any syntax.

The syntax of LDML 8 including colon syntax, parentheses syntax, square brackets, LassoScript, HTML form inputs, URLs, and compound expressions is introduced in the previous chapter *LDML 8 Syntax*.

Introduction

This chapter describes the syntax features of LDML 8. Most of the topics in this chapter are interrelated, and many of the terms used in this chapter are defined in *Appendix A: Glossary* of the Lasso Professional 8 Setup Guide. Consult this glossary if you are unsure of how any terms are used in this guide.

The first part of this chapter describes the different types and categories of LDML tags. The next part of the chapter describes the syntax of individual tags. The different components of tags are discussed, followed by an introduction to the various parameters that can be specified in LDML tags. Next, the focus shifts to the values which are used to specify parameters.

A discussion of Lasso's built-in data types sets the stage for the introduction of symbols and expressions which can be used to modify values. Finally, the chapter ends with a technical description of the delimiters used to specify all the different tag types within Lasso and a brief discussion of syntax rules and guidelines which make coding format files within Lasso easier.

Tag Types

LDML 8 tags are divided into five different types depending on how the tags are used and how their syntax is specified. Each of the five tag types is listed in the table below and then discussed in more detail in the sections that follow, including details of how each tag type can be used within a format file.

Table 1: LDML 8 Tag Types

Tag Type	Example
Substitution Tag	[Field: 'Company_Name']
Process Tag	[Event_Schedule: -URL='http://www.example.com/']
Member Tag	['String'->(Get: 3)]
Container Tag	[Loop: 5] … Looping Text … [/Loop]
Command Tag	<input type="hidden" name="-Required">

Substitution Tags

Substitution tags return a value which is substituted in place of the tag within the format file being served to a client. Most of the tags in LDML are substitution tags. Substitution tags are used to return field values from

a database query, return the results of calculations, or to display information about the state of Lasso Service and the current page request.

The basic format for substitution tags is different in colon or parentheses syntax. Colon syntax features a tag name followed by a colon and then one or more parameters separated by commas.

```
[Substitution_Tag: Tag_Parameter, -EncodingKeyword]
```

Parentheses syntax features a tag name followed by parentheses which surround the parameters of the tag.

```
[Substitution_Tag(Tag_Parameter, -EncodingKeyword)]
```

Every substitution tag also accepts an optional encoding keyword as described later.

Substitution tags have the same basic form when they are expressed in a LassoScript as when they are expressed in square brackets, except that each tag must end with a semi-colon when expressed in a LassoScript. The following example shows the format of substitution tags expressed in a LassoScript in both colon and parentheses syntax

```
<?LassoScript
  Substitution_Tag: Tag_Parameter, -EncodingKeyword;
  Substitution_Tag(Tag_Parameter, -EncodingKeyword);
?>
```

To embed a substitution tag within square brackets:

- Specify the substitution tag on its own. The tag will be replaced by its value when the page is served to a client. For example, the following [Field] tags will be replaced by the company's information from the database. The tag is shown in both colon and parentheses syntax:

  ```
  [Field: 'Company_Name'] ➜ OmniPilot
  [Field('Company_URL')] ➜ http://www.omnipilot.com
  ```

- Specify the substitution tag within HTML or XML markup tags. The LDML tag will be replaced by its value when the page is served to a client, but the markup tags will be served as written. For example, the following [Field] tags are replaced by the company's information from the database within an HTML anchor tag.

  ```
  <a href="[Field: 'Company_URL']">[Field: 'Company_Name']</a>

  ➜ <a href="http://www.omnipilot.com">OmniPilot</a>
  ```

To embed a substitution tag within a LassoScript:

- Specify the substitution tag inside the LassoScript container followed by a semi-colon. The value of the LassoScript will be the value of the lone substitution tag. For example, the [Field] tag is the value of the LassoScript in the following code in colon syntax:

```
<?LassoScript
  Field: 'Company_Name';
?>
```

➜ OmniPilot

- Specify multiple substitution tags on separate lines of the LassoScript. End each tag with a semi-colon. The value of the LassoScript will be the concatenation of the value of all the substitution tags. For example, the [String] tags and [Field] tag define the value of the LassoScript in the following code in parentheses syntax:

```
<?LassoScript
  String('<b>', -EncodeNone);
  Field('Company_Name');
  String('</b>', -EncodeNone);
?>
```

➜ OmniPilot

Note: Every substitution tag accepts an optional encoding parameter which specifies the output format for the value which is being returned by the tag. Please see the section on ***Encoding*** below for more details.

Process Tags

Process tags perform an action which does not return a value. They can be used to alter the HTTP header of an HTML file being served, to store values, to schedule tasks for later execution, to send email messages, and more.

The basic format for process tags is identical to substitution tags: a tag name followed by a colon and then one or more parameters separated by commas. Or, in parentheses syntax the tag name followed by parentheses which contain the parameters of the tag.

```
[Process_Tag: Tag_Parameter]

[Process_Tag(Tag_Parameter)]
```

Process tags have the same basic form when they are expressed in a LassoScript as when they are expressed in square brackets. Except that each tag must end with a semi-colon when expressed in a LassoScript.

The following examples shows the format of process tags expressed in a LassoScript:

```
<?LassoScript
  Process_Tag: Tag_Parameter;
  Process_Tag(Tag_Parameter);
?>
```

To embed a process tag within square brackets:

- Specify the process tag on its own. The tag will be removed from the format file when it is served. For example, the following [Email_Send] tag will send an email to a specified email address, but will return no value in the Web page being served.

```
[Email_Send: -Host='smtp.myserver.com',
             -To='Somebody@example.com',
             -From='Nobody@example.com',
             -Subject='This is the subject of the email',
             -Body='This is the message text of the email']
```

To embed a process tag within a LassoScript:

- Specify the process tag inside the LassoScript container followed by a semi-colon. Since the process tag does not return a value it will not affect the return value of the LassoScript. For example, the following [Email_Send] tag will send an email to a specified email address, but since the LassoScript contains only this tag it will return no value in the format file being served:

```
<?LassoScript
  Email_Send: -Host='smtp.myserver.com',
              -To='Somebody@example.com',
              -From='Nobody@example.com',
              -Subject='This is the subject of the email',
              -Body='This is the message text of the email';
?>
```

  A combination of substitution and process tags can be included in a LassoScript, but the output value of the LassoScript will be determined solely by the value of the substitution tags.

Member Tags

Member tags modify or return data from a value of a specific data type. Each data type in Lasso has different member tags. Member tags can either be used in the fashion of process tags to alter a value or they can be used in the fashion of substitution tags to return a value.

Member tags differ from substitution and process tags in that they must be called using the member symbol -> and a value from the appropriate data type. The following example shows the structure of member tags in both colon and parentheses syntax:

```
[Value->(Tag_Name: Parameters)]

[Value->Tag_Name(Parameters)]
```

For example the [String->Get] member tag requires a value of type string. Member tags are always written in this fashion in the documentation: the data type followed by the member symbol and the specific tag name. The following code fetches the third character of the specified string literal:

```
[Encode_HTML: 'The String'->(Get: 3)] ➜ e

[Encode_HTML('The String'->Get(3))] ➜ e
```

Member tags are defined for any of the built-in data types and third parties can create additional member tags for custom data types. The built-in data types include String, Integer, Decimal, Map, Array, and Pair. More information can be found in the section on *Data Types* below.

To embed a member tag within square brackets:

- Specify the member tag as the parameter of an [Encode_HTML] substitution tag. This makes it clear that you want to output the value returned by the member tag.

```
[Encode_HTML: 'The String'->(Get: 3)] ➜ e
[Encode_HTML: 123->(Type)] ➜ Integer
```

To embed a member tag within a LassoScript:

- Specify the member tag as the parameter of an [Encode_HTML] substitution tag. This makes it clear that you want to output the value returned by the member tag.

```
<?LassoScript
  Var:'Text'='The String';
  Encode_HTML: $Text->(Get: 3);
?>

➜ e
```

- Member tags can be specified directly if they are being used in the fashion of a process tag. In the following example, the [String->Append] member tag is used to add text to the string, but no result is returned.

```
<?LassoScript
  Var:'Text'='The String';
  $Text->(Append: ' is longer.');
?>
```

Container Tags

Container tags are a matching pair of tags which enclose a portion of a format file or LassoScript and either alter the enclosed contents or change the behavior of tags within the enclosed contents. The opening tag uses the same syntax as a substitution or process tag. The closing tag has the same name as the opening tag, but the closing tag is specified with a leading forward slash. This is similar to how HTML markup tags are paired.

In the documentation, container tags are referred to by specifying both tags with an ellipsis representing the enclosed content. The loop tag will be referred to as [Loop] ... [/Loop]. When the attributes or parameters of one half of the container tag pair is being discussed, then just the single tag will be named. The opening loop tag is [Loop] and the closing loop tag is [/Loop].

For example, the following [Loop] tag written in colon syntax has a single parameter which specifies the number of times the contents of the tag will be repeated. The [/Loop] tag defines the end of the area which will be repeated:

```
[Loop: 5] Repeated [/Loop]
```

➜ Repeated Repeated Repeated Repeated Repeated

The same loop written in parentheses syntax:

```
[Loop(5)] Repeated [/Loop]
```

➜ Repeated Repeated Repeated Repeated Repeated

To embed a container tag within square brackets:

- Specify the opening container tag followed by the contents of the container tags and the closing container tag. The contents of the container tags will be affected by the parameters passed to the opening container tag. For example, the following [If] tag will output its contents if its parameter evaluates to True. Since 1 does indeed equal 1 the output is True.

```
[If: 1 == 1] True [/If] ➜ True
```

Note: Both the opening and closing tags of a container tag must be contained within the same format file. Container tags can be nested, but all enclosed container tags must be closed before the enclosing container tag is closed. See the *Conditional Logic* chapter for more information.

To embed a container tag within a LassoScript:

- Specify the opening container tag followed by the contents of the container tag and the closing container tag. Each tag must end with a semi-colon. For readability, the contents of a container tag is often indented. For example, the following [If] tag will output the contents of the enclosed tags if its parameter evaluates to True. Since 1 does indeed equal 1 the output is True.

```
<?LassoScript
  If: 1 == 1;
    True;
  /If;
?>
```

➜ True

Command Tags

Most command tags are actually parameters of the [Inline] tag, but can be used on their own within HTML forms or URLs. Command tags are used to send additional information in a form submission or URL request that is flagged for special use by Lasso. This includes specifying field search operators, required form fields, error response pages, and passing token information.

Command tags names always start with a hyphen, e.g. -Required. Command tags can be though of as "floating parameters", as they use the same hyphenated syntax conventions as substitution, process, and container tag parameters, and can also be used directly as [Inline] tag parameters.

The basic format for a command tag is a tag name starting with a hyphen and an associated value. Since command tags can be specified within HTML form inputs, URLs, and as parameters of the [Inline] tag, the form of a command tag is different in each situation.

To embed command tags within an HTML form:

- Specify multiple command tags within the HTML form inputs. Each command tag should be specified in its own form input with the command tag as the name of the input tag.

```
<input type="hidden" name="-CommandTag" value="Command Value">
```

The following example shows a form that contains Lasso command tags. Each -Operator command tag is contained in an HTML hidden input, which augments a field inputs below it. When the form is submitted, each field passed to the searchresponse.lasso page will be passed with an Equals operator, meaning the field value submitted must match values in a database exactly before results will be returned.

```
<form action="searchresponse.lasso" method="post">
  <input type="hidden" name="-Operator" value="equals">
  <input type="text" name="Field1" value="">
  <input type="hidden" name="-Operator" value="equals">
  <input type="text" name="Field2" value="">

  <input type="submit" value="Search">

</form>
```

- Command tags occasionally accept a parameter which is specified just after the name of the tag following a period. For example, the -Token tag has a name parameter and a value parameter. The -Token tag can be specified in a form as follows:

```
<input type="text" name="-Token.Name" value="Default Value">
```

To embed command tags within a URL:

- Specify multiple command tags within the parameters of the URL. A URL consists of a page reference followed by a question mark and one or more URL parameters. Each command tag parameter should be specified as the command tag followed by an equal sign then its value. Individual command tag parameters should be separated in the URL by ampersands.

```
http://www.example.com/default.lasso?-CommandTag=Command%20Value
```

A full action would be specified as follows. The result of selecting this URL in a Web browser would be that the response page searchresponse.lasso will be returned to the visitor with the result of the search from the specified database and table.

```
http://www.example.com/searchresponse.lasso?-Operator=Equals&Field1=Value1&
-Operator=Equals&Field2=Value2
```

To embed command tags within an [Inline]:

- Specify multiple command tags within the opening [Inline] tag. The command tags will specify the action which the [Inline] is to perform. The contents of the [Inline] ... [/Inline] tags will be affected by the results of this action. The following example shows how the -Op tags can be used directly within an [Inline] tag.

```
[Inline:
  -Database='Contacts'
  -Table='People',
  -KeyField='ID',
  -Op='eq',
  'Field1'='Value1',
  -Op='eq',
  'Field2'='Value2',
  -Search]
  …
[/Inline]
```

Tag Categories and Naming

All of the tags in LDML 8 are grouped and named according to a few simple rules. These rules define where the tag can be found in Lasso 8 documentation and in Lasso Administration.

Tag Categories

The following chart describes the major tag categories in LDML 8. Each tag category is discussed in more detail later in the book. Look for a chapter which has the same name as the tag category or use the index to locate a particular tag.

Table 2: LDML 8 Tag Categories

Tag Category	Description
Action	Data source actions.
Administration	Administration and security tags.
Array	Array, map, pair, and other compound data types.
Bytes	Byte streams for manipulating binary data and converting data between character sets.
Client	Information about the current visiting client.
Comparators	Used to sort and match elements within compound data types.
Conditional	Conditional logic and looping tags.
Constant	Constant values that are used throughout Lasso.
Custom Tag	Create custom LDML tags, data types, and data sources.
Data Types	Tags to cast values to specific data types.
Database	Information about the current database.

Date	Date manipulation tags.
Email	Tags for sending, receiving, and processing email.
Encoding	Tags for encoding data.
Encryption	Encrypt data so it can be transmitted securely.
Error	Tags for reporting and handling errors.
File	Tags for manipulating files.
Image	Tags for manipulating images.
Include	Allows data to be included in a format file.
JavaBeans	Call JavaBeans from within Lasso code.
Link	Link to other records in the current found set.
Matchers	Match elements within compound data types.
Math	Mathematical operations and integer member tags.
Namespace	Use, load, and unload tag namespaces.
Networking	Tags for performing network operartions.
Operator	Set and retrieve logical and field-level operators.
Output	Tags for formatting or suppressing output.
PDF	Tags for creating PDF documents.
Results	Results from the current Lasso action.
Sessions	Create session variables.
String	String operations and string member tags.
Tags	Member tags of the Null and Tag t ypes.
Technical	Tags for performing low-level operations.
Threads	Thread communication and synchronization.
Utility	Tags which don't fit in any other category.
Variable	Tags for creating and manipulating variables.
XML	Tags for processing XML.

Tag Naming Conventions

Tags in LDML are named according to a set of well-defined naming conventions. Understanding these conventions will make it easier to locate the documentation for specific tags. We also recommend the following naming conventions when creating custom tags, libraries, and modules.

- Case is unimportant in both tag name and tag parameter names. All LDML tags can be written in uppercase, lowercase, or any combination of mixed case. Tags are always written in title case in the documentation. The following tag names would all be equivalent, but the first, e.g. title case, is preferred:

```
[Tag_Name]          [tag_name]
[TAG_NAME]          [TaG_NaMe]
```

- Core language tags usually have simple tag names and do not contain underscore characters. For example:

```
[Variable]          [Field]
[If] ... [Else] ... [/If]          [Inline] ... [/Inline]
```

- Most tag names include a category name (or namespace) followed by an underscore then the specific tag name. For example: [Math_Sin] is the tag in the "Math" category that performs the function "Sine." Similarly, [Link_NextRecordURL] is the tag in the "Link" category that returns the URL of the next record in the found set. Category names appear in tag names based on the following format:

```
[Category_TagName]
```

- Tag names never start with an underscore character. These tag names are reserved for internal use.
- Some tag names reference another tag or other component of Lasso 8 followed by an underscore then a specific tag name. For example [MaxRecords_Value] returns the value of the -MaxRecords command tag. There is no underscore in the words MaxRecords since it is referring to another tag. This association can be expressed as follows:

```
[TagReference_TagName]
```

- Many tag names include a word at the end that specifies what the output of the tag will be. For instance, [Link_NextRecord] ... [/Link_NextRecord] is a container tag that links to the next record, but [Link_NextRecordURL] is a substitution tag that returns the URL of the next record. Tags that end in "URL" output URLs. Tags that end in "List" and most tags that have plural names output arrays. Tags that end in "Name" return the name of a database entity. Tags that end in "Value" return the value of the named database entity.

```
[Link_NextRecordURL]          [File_ListDirectory]
[Action_Params]               [Variables]
[KeyField_Name]               [KeyField_Value]
```

- Member tag names are written in the documentation with the data type followed by the member symbol then the tag name. For example, the Get tag of the data type string would be written: [String->Get]. All of the member tags of a particular data type are considered to be part of the category which has the same name as the data type. All of the string member tags are part of the string category.

- Tags created by third parties should start with a prefix which identifies the creator of the tag. For example, tags from "Example Company" might all start with Ex_. This ensures that the third party tags do not conflict with built-in tags or other third party tags.

  [Ex_TagName] [ExCategory_TagName]

Synonyms and Abbreviations

The following charts detail some standard synonyms and abbreviations in LDML 8. Any of the synonyms or abbreviations in the right column can be used instead of the term in the left column, but the term in the left column is preferred.

Table 3: LDML 8 Synonyms

Preferred Term	Synonym	Example
Field	Column	[Field_Name] [Column_Name]
Record	Row	[Records] [Rows]
KeyValue	RecordID	[KeyField_Value] [RecordID_Value]
Table	Layout	[Table_Name] [Layout_Name]

Table 4: LDML 8 Abbreviations

Preferred Term	Abbreviation	Example
Operator	Op	-Operator -Op
Required	Req	-Required -Req
Variable	Var	[Variable] [Var]

Some tags which were synonyms in earlier version of Lasso are no longer supported. Please see the *Upgrading Your Solutions* section for more information. For a complete list of synonyms and abbreviations please consult the LDML 8 Reference.

Parameter Types

This section introduces the different types of parameters which can be specified within LDML tags. This discussion is applicable to substitution tags, process tags, the opening tag of container tags, and member tags. Command tag parameters are fully described in the previous section.

Table 5: Parameter Types

Parameter Type	Example
Value	[Field: 'Field_Name']
Keyword	[Error_CurrentError: -ErrorCode]
Keyword/Value	[Inline: -Database=(Database_Name), …]
Name/Value	[Variable: 'Variable_Name'='Variable_Value']

Some parameters are required for a tag to function properly. The [Field] and [Variable] tags require that the field or variable to be returned is specified. In contrast, the keyword in [Error_CurrentError] is optional and can be safely omitted. If no keyword is specified for an optional parameter then a default will be used. For a complete listing of required, optional, and default parameters for each tag, please consult the LDML 8 Reference.

A **Value** is the most basic parameter type, and consists of a basic data type contained within a tag after a colon character (:). Values include string literals, integer literals, decimal literals, sub-tags, and complex expressions.

```
[Field: 'Field_Name']                [Date: '09/29/2003']
[Var_Defined: 'Variable_Name']       [Encode_HTML: 123]
```

A value can also be the value of a sub-tag. Any substitution tag or member tag can be used as a sub-tag. The syntax of the sub-tag is the same as that for the substitution tag or member tag except that the tag is enclosed in parentheses rather than square brackets. The following [Encode_HTML] tags are used to output the value of several different sub-tags:

```
[Encode_HTML: (Field: 'Field_Name')]  [Encode_HTML: (Date)]
[Encode_HTML: (Loop_Count)]          [Encode_HTML: 'String'->(Get: 3)]
```

A **Keyword** is a tag-specific parameter that alters the behavior of a tag. Keyword names always start with a hyphen. This makes it easy to distinguish tag-specific keywords from user-defined parameters. The following examples of [Server_Date] show how the same tag can be used to generate different content based on the keyword that is specified:

```
[Server_Date: -Short] ➜ 3/24/2001
[Server_Date: -Long] ➜ March 24, 2001
[Server_Date: -Abbrev] ➜ Mar 24, 2001
[Server_Date: -Extended] ➜ 2001-03-24
```

Note: For backwards compatibility, some tags will accept keyword names without the leading hyphen. This support is not guaranteed to be in future versions of Lasso so it is recommended that you write all keyword names with the leading hyphen.

A **Keyword/Value** parameter is the combination of a tag specific keyword and a user-defined value which affects the output of a tag. The keyword name is specified followed by an equal sign and the value. Keyword/value parameters are sometimes referred to as named parameters. For example, the [Date] tag accepts multiple keyword/value parameters which specify the characteristics of the date which should be output:

```
[Date: -Year=2001, -Day=24, -Month=3] ➜ 3/24/2001
```

Command tags are used like keyword/value parameters in the [Inline] tag. The command tag functions like the keyword and is written with a leading hyphen. For example, the following [Inline] contains several command tags that define a database action:

```
[Inline: -FindAll
    -Database='Contacts',
    -Table='People',
    -KeyField='ID']
  ... Results ...
[/Inline]
```

A **Name/Value** parameter is the combination of a user-defined name with a user-defined value. The name and the value are separated by an equal sign. Name/value parameters are most commonly used in the [Inline] tag to refine the definition of a database action. For example, the previous [Inline] example can be modified to search for records where the field First_Name starts with the letter s by the addition of a name/value parameter 'First_Name'='s':

```
[Inline: -Search,
    'First_Name'='s',
    -Database='Contacts',
    -Table='People',
    -KeyField='ID']
  ... Results ...
[/Inline]
```

Encoding

Encoding keyword parameters specify the character format in which the data output from a substitution tag should be rendered. Encoding ensures that reserved or illegal characters are changed to entities so that they will

display properly in the desired output format. Encoding keywords allow substitution tags to be used to output data in any of the following formats:

- HTML text for display in a Web browser (default).
- HTML tags for display in a Web browser.
- XML data for data interchange.
- URL parameters to construct a hyperlink.
- ASCII text for inclusion in an email message or log file.

The following table demonstrates each of the encoding keywords available in LDML 8.

Table 6: Encoding Keywords

Keyword	Encoding Performed
-EncodeNone	No encoding is performed.
-EncodeHTML	Reserved, illegal, and extended ASCII characters are changed to their hexadecimal equivalent HTML entities.
-EncodeSmart	Illegal and extended ASCII characters are changed to their hexadecimal equivalent HTML entities. Reserved HTML characters are not changed.
-EncodeBreak	ASCII carriage return characters are changed to HTML .
-EncodeURL	Illegal and extended ASCII characters are changed to their equivalent hexadecimal HTTP URL entities.
-EncodeStrictURL	Reserved, illegal and extended ASCII characters are changed to their equivalent hexadecimal HTTP URL entities.
-EncodeXML	Reserved, illegal, and extended ASCII characters are changed to their UTF-8 equivalent XML entities.

To use an encoding keyword:

Append the desired encoding keyword at the end of a substitution tag. For example, angle brackets are reserved characters in HTML. If you want to include an angle bracket in your HTML output it needs to be changed into an HTML entity. The entity for < is < and the entity for > is >.

```
[String: '<b>HTML Text</b>', -EncodeHTML] ➜ &lt;b&gt;HTML Text&lt;/b&gt;
```

See the *Encoding* chapter for more information.

Data Types

Every value in Lasso is defined as belonging to a specific data type. The data type determines what member tags are available and how symbols affect the value. Data types generally correspond to everyday descriptions of a value with the addition of some data types for structured data. The following table lists the primary data types available in Lasso:

Table 7: Primary LDML 8 Data Types

Data Type	Example
String	'This is a string surrounded by single quotes'
Bytes	[Bytes: 'Binary representation of text or data']
Integer	1500
Decimal	3.14159
Date	9/29/2002 19:12:02
Duration	168:00:00
Array	[Array: 'red', 'green', 'blue', 'yellow']
Map	[Map: 'Company_Name'='OmniPilot', 'City'='Bellevue']

Note: This section describes the primary data types which are used most frequently in LDML. There are many other special-purpose data types in LDML, including PDF, Image, File, and Network Types. These special-purpose types are described in appropriate chapters later in this guide.

Strings

Strings are any series of alphanumeric characters. String literals are surrounded by single quotes. The results of a substitution tag will be considered a string if it contains any characters other than numbers. Please see the *String Operations* chapter for more information.

Some examples of string values include:

- 'String literal' is a string surrounded by single quotes.
- '123456' is a string literal since it is surrounded by single quotes.
- 'A string with \'quotes\' escaped' is a string that contains quote marks. The quote marks are considered part of the string since they are preceded by back slashes.
- The following [Field] tag returns a string value. Notice that the value of a substitution tag is a string value since it contains alphabetic characters:

  [Field: 'Company_Name'] ➜ OmniPilot

- The following code sets a variable to a string value, then retrieves that value:

```
[Variable: 'String' = 'abcdef']
[Variable: 'String'] ➜ abcdef
```

Bytes

Bytes are streams of binary data. This data type is used to represent incoming data from remote Web application servers, files on the local hard disk, or BLOB fields in MySQL databases.

Integers

Integers are any series of numeric characters that represent a whole number. Integer literals are never surrounded by quotes. The results of a substitution tag will be considered an integer if it contains only numeric characters which represent a whole number. Please see the *Math Operations* chapter for more information.

Some examples of integer values include:

- 123456 is an integer literal since it is not surrounded by quotes.
- (-50) is an integer literal. The minus sign (hyphen) is used to define a negative integer literal. The parentheses are required if the literal is to be used as the right-hand parameter of a symbol.
- The following [Field] tag returns an integer value. The value is recognized as an integer since it contains only numeric characters and represents a whole number:

```
[Field: 'Employee_Age'] ➜ 23
```

- The following code sets a variable to an integer value, then retrieves that value:

```
[Variable: 'Integer' = 1000]
[Variable: 'Integer'] ➜ 1000
```

Decimals

Decimals are any series of characters that represent a decimal number. Decimal literals are never surrounded by quotes. Decimal values must include a decimal point and can be expressed in exponential notation. Please see the *Math Operations* chapter for more information.

Some examples of decimal values include:

- 123.456 is a decimal literal since it contains a decimal point and is not surrounded by quotes.

- (-50.0) is a negative decimal literal. The parentheses are required if the literal is to be used as the right-hand parameter of a symbol.
- The following [Field] tag returns a decimal value. The value is recognized as a decimal since it contains numeric characters and a decimal point:

  ```
  [Field: 'Annual_Percentage_Rate'] ➜ 0.12
  ```

- The following code sets a variable to a decimal value, then retrieves that value:

  ```
  [Variable: 'Decimal' = 137.48]
  [Variable: 'Decimal'] ➜ 137.48
  ```

Dates

Dates are a special data type that represent a date and/or time string. Dates in Lasso 8 can be manipulated in a similar manner as integers, and calculations can be performed to determine date differences, durations, and more. For Lasso to recognize a string as a date data type, the string must be explicitly cast as a date data type using the [Date] tag. When casting as a date data type, the following date formats are automatically recognized as valid date strings by Lasso:

```
1/1/2001
1/1/2001 12:34
1/1/2001 12:34:56
1/1/2001 12:34:56 GMT
2001-01-01
2001-01-01 12:34:56
2001-01-01 12:34:56 GMT
```

The "/", "-", and ":" characters are the only punctuation marks recognized in valid date strings by Lasso. If using a date format not listed above, custom date formats can be defined as date data types using the [Date] tag with the -Format parameter. See the *Date and Time Operations* chapter for more information.

Some examples of dates include:

- [Date:'9/29/2002'] is a valid date data type recognized by Lasso.
- [Date:'9.29.2002'] is not recognized by Lasso as a valid date data type due to its punctuation, but can be converted to a date data type using the [Date] tag with the -Format parameter.

  ```
  [Date:'9.29.2002', -Format='%m.%d.%Y']
  ```

- Specific date and time information can be obtained from date data types using accessors.

  ```
  [(Date:'9/29/2002')->DayofYear] ➜ 272
  ```

- Date data types can be manipulated using math symbols. Date and time durations can be specified using the [Duration] tag.

  ```
  [(Date:'9/29/2002') + (Duration: -Day=2)] ➜ 10/01/2002
  ```

- A valid date data type can be displayed in an alternate format using the [Date_Format] tag.

  ```
  [Date_Format:(Date:'9/29/2002'), -Format='%Y-%m-%d'] ➜ 2002-09-29
  ```

  Note: Lasso uses internal standardized date libraries to automatically adjust for leap years and day light savings time when performing date calculations. The current time and time zone are based on that of the Web server. For information on special cases with date calculations during day light saving time, see the *Date and Time Operations* chapter.

Durations

Durations are a special data type that represent a length of time in hours, minutes, and seconds. Durations are not 24-hour clock times, and can represent any length of time. Duration data types in Lasso 8 are related to date data types, and can be manipulated in a similar manner. For Lasso to recognize a string as a duration data type, the string must be explicitly cast as a duration data type using the [Duration] tag. Any numeric string formatted as hours:minutes:seconds or just seconds may be cast as a duration data type.

```
168:00:00
60
```

Colon characters (:) are the only punctuation marks recognized in valid duration strings by Lasso. The [Duration] tag always outputs values in hours:minutes:seconds format regardless of what the input format was. See the *Date and Time Operations* chapter for more information.

Some examples of durations include:

- [Duration:'169:00:00'] is a valid duration data type recognized by Lasso, and represents a duration of 169 hours. This duration will be output as 169:00:00.
- [Duration:'300'] is a valid duration data type recognized by Lasso, and represents a duration of 300 seconds. This duration will be output as 00:05:00 (five minutes).

Arrays

Arrays are a series of values which can be stored and retrieved by numeric index. Arrays can contain values of any other data type, including other arrays. Only certain substitution tags return array values. Array values are never returned from database fields. Please see the *Arrays and Maps* chapter for more information.

Some examples of how to work with arrays include:

- Create an array using the [Array] tag. The following two examples create an array with the days of the week in it, where each day of the week is a string literal. The second example shows abbreviated syntax where the colon (:) character is used to specify the start of an array data type.

```
[Array: 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
[: 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

- Store an array in a variable using the following code which stores the array created in the code above in a variable named Week.

```
[Variable: 'Week' = (Array: 'Monday', 'Tuesday', 'Wednesday',
  'Thursday', 'Friday', 'Saturday', 'Sunday')]
```

- Fetch a specific item from the array using the [Array->Get] member tag. This code fetches the name of the third day of the week.

```
[(Variable: 'Week')->(Get:3)] ➜ Wednesday
```

- Set a specific item from the array using the [Array->Get] member tag. The following code sets the name of the third day of the week to its Spanish equivalent Miercoles.

```
[(Variable: 'Week')->(Get:3) = 'Miercoles']
```

  The new value of the third entry in the array can now be fetched.

```
[(Variable: 'Week')->(Get:3)] ➜ Miercoles
```

Maps

Maps are a series of values which can be stored and retrieved by name. Maps can contain values of any other data type, including arrays or other maps. Only certain substitution tags return map values. Map values are never returned from database fields. Please see the *Arrays and Maps* chapter for more information.

Some examples of how to work with maps include:

- Create a map using the [Map] tag. The following creates a map with some user information in it. The name of each item is a string literal, the values are either string literals or integer literals:

```
[Map:
  'First Name'='John',
  'Last Name'='Doe',
  'Age'=25]
```

- Store a map in a variable using the following code which stores the map created in the code above in a variable named Visitor:

```
[Variable: 'Visitor' = (Map:
  'First Name'='John',
  'Last Name'='Doe',
  'Age'=25)]
```

- Fetch a specific item from the map using the [Map->Get] member tag. This code fetches the visitor's first name:

```
[(Variable: 'Visitor')->(Get:'First Name')] ➜ John
```

- Set a specific item from the map using the [Map->Get] member tag. This code sets the age of the visitor to 29. Notice that the expression returns no value since the member tag is being used in the fashion of a process tag to set a value.

```
[(Variable: 'Visitor')->(Get:'Age') = 29]
```

  The new value of the age entry in the map can now be fetched:

```
[(Variable: 'Visitor')->(Get:'Age')] ➜ 29
```

Other Types

Lasso includes numerous other data types including null, booleans, sets, lists, queues, stacks, priority queues, tree maps, pairs, XML, XMLStream, POP, SMTP, thread tools, and more. See the ***Data Types*** section for complete documentation of the many data types that Lasso offers.

Expressions and Symbols

Virtually all of the values shown in this chapter so far have been simple string, integer or decimal literals. Any tag in LDML which accepts a value as a parameter can accept an expression in place of that value. This allows nested operations to be performed within the parameters of LDML tags.

This section discusses each of the different types of expressions that can be used as values within LDML. It starts with simple expressions and then moves on to more complex expressions.

Table 8: Types of LDML 8 Expressions

Expression	Example
Literal	'String Literal', 100, 150.34
Sub-Tag	(Variable: 'Variable_Name')
Member tag	(Array: 1, 2, 3, 4)->(Get: 4)
Retarget	$String->Uppercase & Get(4)
String Expression	'String One' + 'String Two'
Math Expression	100 / 4 + 25 - (-20)
Complex Expression	'' + 100 / 4 + ''
Conditional Expression	'azure' == 'blue'
Logical Expression	('blue' != 'orange') || ('red' != 'green')
If Else Expression	($conditional ? 'True Result' | 'False Result')
Tag Reference	\Tag_Name

This section also describes each of the different symbols that can be used to modify expressions specific to each type of expression.

Literals

Any string literal, integer literal, or decimal literal can be used as a value in LDML. These are the most basic types of values and the simplest examples of expressions. These literals are defined in the previous section on *Data Types*. Some examples of outputting literal values include:

```
['String Literal']          [123]
[100.14]                    [(-123)]
```

Sub-Tags

Substitution tags are LDML tags that return a value and any substitution tag can be used as a simple expression in LDML. The syntax of the sub-tag is the same as that for the substitution tag except that the tag is enclosed in parentheses rather than square brackets. The value of the expression is simply the value of the substitution tag. For example, the following expressions output the value of the specified sub-tag.

```
[(Field: 'Field_Name')]      [(Date)]
[(Loop_Count)]
```

Note: Substitution tags have a default encoding keyword of -EncodeHTML applied when they are the outermost tag. However, when substitution tags are used as sub-tags or in square brackets without

an [Encode_HTML] tag, no encoding is applied by default. See the *Encoding* chapter for more information.

Member Tags

Member tags that return values can be used as simple expressions in LDML. An appropriate member tag for any given data type can be attached to a value of that data type using the member symbol ->. For example, the following member tag returns a character from the specified string literal:

```
[Encode_HTML: 'String'->(Get: 3)]
```

The value on the left side of the member symbol can be any expression which is valid in LDML. It can be a string literal, integer literal, decimal literal, sub-tag, or any of the expressions which are defined below. For example, the following member tag would return the third character of the name which is returned from the database:

```
[Encode_HTML: (Field: 'First Name')->(Get: 3)]
```

Note: The [Encode_HTML] tag is not technically required in member tag expressions. ['String'->(Get: 3)] will evaluate to the character r. However, for ubiquitous HTML encoding, the use of the [Encode_HTML] tag is recommended.

Table 9: Member Tag Symbol

Symbol	Name	Example
->	Member	['abcdef'->(Get: 3)] ➜ c

Retarget

Many member tags modify the target of the tag but do not return any result. The retarget symbol & can be used to modify these tags so they modify the target and return the target as a result.

Table 10: Retarget Symbol

Symbol	Name	Example
&	Retarget	[$String->Uppercase & Get(3)] ➜ C

For example, adding the redirection symbol & to the example above results in the value of $Text being modified and returned.

```
<?LassoScript
  Var:'Text'='The String';
  $Text->Append(' is longer.') &;
?>
```

➜ The String is longer.

The retarget symbol allows multiple member tags to be strung together to perform a series of operations on the target. For example, this code sorts an array and then gets one elements within it. Without first sorting the array the result would be 4.

```
[(Array: 2, 4, 3, 5, 1)->Sort & (Get: 2)]
```

➜ 2

String Expressions

String expressions are the combination of string values with one or more string symbols. A string expression defines a series of operations that should be performed on the string values. The string values which are to be operated upon can be either string literals or any expressions which return a string value.

Symbols should always be separated from their parameters by spaces and string literals should always be surrounded by single quotes. Otherwise, Lasso may have a difficult time distinguishing literals and LDML tags.

The most common string symbol is + for concatenation. This symbol can be used to combine multiple string values into a single string value. For example, to add bold tags to the output of a [Field] tag we could use the following string expression:

```
['<b>' + (Field: 'CompanyName') + '</b>']
```

➜ OmniPilot

String symbols can also be used to compare strings. String symbols can check if two strings are equal using the equality == symbol or can check whether strings come before or after each other in alphabetical order using the greater than > or less than < symbols. For example, the following code reports the proper order for two strings:

```
[If: 'abc' == 'def']
  abc equals def
[Else: 'abc' < 'def']
  abc comes before def
[Else: 'abc' > 'def']
  abc comes after def
[/If]
```

➜ abc comes before def

Note: Always place spaces between a symbol and its parameters. The - symbol can be mistaken for the start of a negative number, command tag, keyword, or keyword/value parameter if it is placed adjacent to the parameter that follows.

Table 11: String Expression Symbols

Symbol	Name	Example
+	Concatenation	['abc' + 'def'] ➜ abcdef
*	Repetition	['abc' * 2] ➜ abcabc
-	Deletion	['abcdef' - 'cde'] ➜ abf
>>	Contains	['abcdef' >> 'bcd'] ➜ True
!>>	Not Contains	['abcdef' !>> 'bcd'] ➜ False
==	Equality (Value Only)	['abc' == 'def'] ➜ False
===	Equality (Value & Type)	['123' == 123] ➜ False
!=	Inequality (Value Only)	['abc' != 'def'] ➜ True
!==	Inequality (Value & Type)	['123' != 123] ➜ True
<	Less Than	['abc' < 'def'] ➜ True
>	Greater Than	['abc' > 'def'] ➜ False

Please see the *String Operations* chapter for more information.

Math Expressions

Math expressions are the combination of decimal or integer values with one or more math symbols. A math expression defines a series of operations that should be performed on the decimal or integer values. The numeric values which are to be operated upon can be either decimal or integer literals or any expressions which return a numeric value.

Symbols should always be separated from their parameters by spaces. This ensures that the + and - symbols are not mistaken for the sign of one of the parameters.

Simple math operations can be performed directly within an expression. For example, the following expressions return the value of the specified simple math calculations.

[10 + 5] ➜ 15
[10 - 5] ➜ 5
[10 * 5] ➜ 50
[10 / 5] ➜ 2

If the second parameter of the expression is negative it should be surrounded by parentheses.

[10 + (-5)] ➜ 5 [10 * (-5)] ➜ -50

Math expressions can be used on either decimal or integer values. If both parameters of a math symbol are integer values then an integer result will be returned. However, if either parameter of a math symbol is a decimal value then a decimal value will be returned. Decimal return values always have at least six significant digits.

Note: Always place spaces between a symbol and its parameters. The - symbol can be mistaken for the start of a negative number, command tag, keyword, or keyword/value parameter if it is placed adjacent to the parameter that follows.

Table 12: Math Expression Symbols

Symbol	Name	Example
+	Addition	[100 + 25] ➜ 125
-	Subtraction	[100 - 25] ➜ 75
*	Multiplication	[100 * 25] ➜ 2500
/	Division	[100 / 25] ➜ 4
%	Modulo	[100 % 25] ➜ 0
==	Equality (Value Only)	[100 == 25] ➜ False
===	Equality (Value & Type)	[100 === 100.0] ➜ False
!=	Inequality (Value Only)	[100 != 25] ➜ True
!==	Inequality (Value & Type)	[100 !== 100.0] ➜ True
>	Greater Than	[100 > 25] ➜ True
>=	Greater Than or Equal	[100 >= 25] ➜ True
<	Less Than	[100 < 25] ➜ False
<=	Less Than or Equal	[100 <= 25] ➜ False

Please see the *Math Operations* chapter for more information.

Complex Expressions

Complex expressions can be created by combining sub-expressions together using one or more string or math symbols. The results of the sub-expressions are used as the parameters of the enclosing parameters. Expressions can be enclosed in parentheses so that the order of operation is clear.

For example, the following complex math expression contains many nested math expressions. The expressions in the innermost parentheses are processed first and the result is used as a parameter for the enclosing

expression. Notice that spaces are used on either side of each of the mathematical symbols.

```
[(1 + (2 * 3) + (4.0 / 5) + (-6))] ➜ 1.8
```

The following complex string expressions contains many nested string expressions. The expressions in the innermost parentheses are processed first and the result is used as a parameter for the enclosing expression:

```
[('abc' + ('def' * 2) + ('abcdef' - 'def') + 'def')] ➜ abcdefdefabcdef
```

String and math expressions can be combined. The behavior of the symbols in the expression is determined by the parameters of the symbol. If either parameter is a string value then the symbol is treated as a string symbol. Only if both parameters are decimal or integer values will the symbol be treated as a math symbol. For example, the following code adds two numbers together using the math addition + symbol and then appends bold tags to the start and end of that value using the string concatenation + symbol:

```
['<b>' + (100 + (-35)) + '</b>', -EncodeNone] ➜ <b>65</b>
```

Conditional Expressions

Conditional expressions are the combination of values of any data type with one or more conditional symbols. A conditional expression defines a series of comparisons that should be performed on the parameter values. The values which are to be operated upon can be valid values or expressions.

Conditional symbols were introduced in the *String Expressions* and *Math Expressions* sections above in the context of comparing string or math values. They can actually be used on values of any data type including arrays, maps, and custom types defined by third parties.

Values are automatically converted to an appropriate data type for a comparison. For example, the following comparison returns True even though the first parameter is a number and the second parameter is a string. The second parameter is converted to the same type as the first parameter, then the values are compared:

```
[123 == '123'] ➜ True
```

Conditional expressions are used in the [If] ... [/If] and [While] ... [/While] container tags to specify the condition under which the contents of the tag will be output. For example, the following [If] tag contains a conditional expression that will evaluate to True only if the company name is OmniPilot:

```
[If: (Field: 'Company_Name') == 'OmniPilot']
  The company name is OmniPilot
[/If]
```

Table 13: Conditional Expression Symbols

Symbol	Name	Example
>>	Contains	['abcdef' >> 'bcd'] ➜ True
!>>	Not Contains	['abcdef' !>> 'bcd'] ➜ False
==	Equality (Value Only)	[100 == 25] ➜ False
===	Equality (Value & Type)	[100 == '100'] ➜ False
!=	Inequality (Value Only)	[100 != 25] ➜ True
!==	Inequality (Value & Type)	[100 !== '100'] ➜ True
>	Greater Than	[100 > 25] ➜ True
>=	Greater Than or Equal	[100 >= 25] ➜ True
<	Less Than	[100 < 25] ➜ False
<=	Less Than or Equal	[100 <= 25] ➜ False

Please see the *Conditional Logic* chapter for more information.

Logical Expressions

Logical expressions are made up of multiple conditional sub-expressions combined with one or more logical symbols. The values of the conditional sub-expressions are combined according to the operation defined by the logical symbol.

Logical expressions are most commonly used in the [If] ... [/If] container tag to specify the condition under which the contents of the tag will be output. A single [If] tag can check multiple conditional expressions if they are combined into a single logical expressions.

For example, the following [If] tag contains a logical expression that will evaluate to True if one or the other of the sub-expressions is True. The [If] ... [/If] container tag will display its contents only if the company name is OmniPilot or the product name is Lasso Professional:

```
[If: ((Field: 'Company_Name') == 'OmniPilot') ||
     ((Field: 'Product_Name') == 'Lasso Professional')]
  The company name is OmniPilot
[/If]
```

Table 14: Logical Expression Symbols

Symbol	Name	Example
&&	And	[True && False] ➜ False
||	Or	[True || False] ➜ True
!	Not	[! True] ➜ False

Please see the *Conditional Logic* chapter for more information.

Note: These logical symbols should not be confused with the logical search operators which can be used to assemble complex search criteria. See the *Database Interaction Fundamentals* chapter for more information about logical search operators.

If Else Expressions

The if else symbol ? | allows a conditional expression to be specified inline. The symbol is a good alternative to using [If] ... [/If] tags for simple conditional expressions.

Table 15: Logical Expression Symbols

Symbol	Name	Example
? |	If Else	[True ? 'TrueResult' | 'FalseResult'] ➜ TrueResult
		[False ? 'TrueResult' | 'FalseResult'] ➜ FalseResult

Please see the *Conditional Logic* chapter for more information.

In the following example the field First_Name is checked to see if it is empty. If it is then N/A is returned. Otherwise, the field value is returned.

```
[Encode_HTML: (Field: 'First_Name') == '' ? 'N/A' | (Field: 'First_Name')]
```

Tag References

The back slash \ can be used to reference tags by name. This allows the member tags of the tag data type to be used on both built-in and custom tags. For more information about the tag data type consult the Extending Lasso Guide.

For example, \Field returns a reference to the built-in [Field] tag. Each of the following code samples is an equivalent way of calling the [Field] tag.

```
<?LassoScript
  Field: 'First_Name';
  \Field->(Run: -Params=(Array: 'First_Name'));
```

```
  \Field->(Invoke: 'First_Name');
?>
```

Similarly, the member tags of a data type can be referenced using the -> symbol and the back slash \ symbol together. For example, Array->\Join would return a reference the [Array->Join] tag. Each of the following code samples is an equivalent way of calling the [Array->Join] tag.

```
<?LassoScript
  (Array: 'One', 'Two')->(Join: ' - ');
  (Array: 'One', 'Two')->\Join->(Run: -Params=(Array: ' - '));
  (Array: 'One', 'Two')->\Join->(Invoke: ' - ');
?>
```

Delimiters

This section describes the delimiters which are used to define LDML and HTML. It is important to understand how delimiters are used so that tags can be constructed with the proper syntax.

Table 16: LDML 8 Delimiters

Symbol	Name	Function
[	Square Bracket	Start of tag square bracket syntax.
]	Square Bracket	End of tag in square bracket syntax.
/	Forward Slash	Closing container tag name.
\	Back Slash	Escapes special characters in strings or returns a reference to a tag or member tag.
:	Colon	Separates tag name from tag parameters in colon syntax
()	Parentheses	Surround tag parameters in parentheses syntax. Also used to surround sub-tags or expressions.
,	Comma	Separates tag parameters.
=	Equal Sign	Separates name/value parameter.
-	Hyphen	Starts command tag name and keyword names.
'	Single Quote	Start and end of LDML string value.
<?LassoScript	LassoScript	Start of LassoScript.
?>	LassoScript	End of LassoScript.
{	Curly Brace	Start of compound expression syntax (LassoScript contained within square bracket syntax).
}	Curly Brace	End of compound expression syntax.
;	Semi-Colon	Separates tags within LassoScript.
//	Double Slash	Start of line comment in LassoScript.
/*	Asterisk Slash	Start of extended comment in LassoScript.
*/	Asterisk Slash	End of extended comment in LassoScript.
->	Member Symbol	Separates data value from member tag.
&	Retarget Symbol	Separates multiple member tags in expression.
? |	If Else Symbol	Specifies an inline conditional expression.
	Space	Specified between symbols and their parameters.

When possible, parentheses should be used around all expressions, sub-tag calls, and negative literals. The parentheses will ensure that Lasso accurately parses each expression. If an expression does not seem to be working correctly, try adding parentheses to make the order of operation explicit.

Unlike symbols, white space is generally not required around delimiters. White space may be used to format code in order to make it more readable.

Note: The double quote " was a valid LDML separator in earlier versions of Lasso but has been deprecated in Lasso Professional 8. It is not guaranteed to work in future versions of Lasso.

The following table shows the delimiters which are used in HTML pages and HTTP URLs.

Table 17: HTML/HTTP Delimiters

Symbol	Name	Function
<	Angle Bracket	Start of an HTML or XML tag.
>	Angle Bracket	End of an HTML or XML tag.
=	Equal Sign	Separates name/value parameter or attribute.
"	Double Quote	Start and end of HTML string value.
?	Question Mark	Separates path from parameters in URL.
#	Hash Mark	Separates path from target in URL.
&	Ampersand	Separates URL parameters.
/	Forward Slash	Folder delimiter in URL paths or designation of Web server root if used at the start of a URL path.
../	Dot Dot Slash	Up one folder level in URL paths.
	Space	Separates tag attributes.

Illegal Characters

The following chart details characters which can cause Lasso problems if they appear in a format file or within LDML code outside of a string literal. These characters are not valid in tag names, keyword names, or parameter names.

For best results use a dedicated HTML editor such as Macromedia Dreamweaver or Adobe GoLive or a text editor such as BareBones BBEdit or Microsoft NotePad to create LDML format files. The Zap Gremlins option

in BBEdit is particularly useful in eliminating problem characters such as these.

Table 18: Illegal Characters

Symbol	Name	Function
\0	Null Character	The null-character is often used as and end-of-file marker. Lasso may abort processing if it reads a null character within a format file.

Note: The non-breaking space is now recognized as white space by Lasso when it is encountered within LDML statements.

6

Chapter 6
LDML 8 Reference

This chapter documents how to use the LDML 8 Reference.

- *Overview* provides an overview of the LDML 8 Reference and how to access it.
- *Search* discusses searching the LDML 8 Reference.
- *Browse* discusses browsing the LDML 8 Reference by tag type or category.
- *Detail* discusses how to view information about LDML tags, and what information can be displayed.
- *List* discusses how all available tags can be listed.

Overview

The LDML 8 Reference is a resource provided by OmniPilot for finding descriptions, usage guidelines, and detailed examples of LDML tags. It is the official reference for all tags in LDML 8.

The LDML 8 Reference is a locally-stored LassoApp and Lasso MySQL database included with each installation of Lasso Professional 8, and is also available on the OmniPilot Web site.

To access the LDML 8 Reference:

- The LDML 8 Reference can be accessed through the *Support > LDML Reference* section in Lasso Administration.

- The LDML 8 Reference can also be accessed on the local machine at the following URL, substituting the actual IP address or host name of the Web server for www.example.com. The LDML 8 Reference requires the administrator username and password for local access.

  http://www.example.com/Lasso/LDMLReference.LassoApp

- The LDML 8 Reference can be accessed at OmniPilot at the following URL. This reference is open for anyone to use and includes a public comment interface.

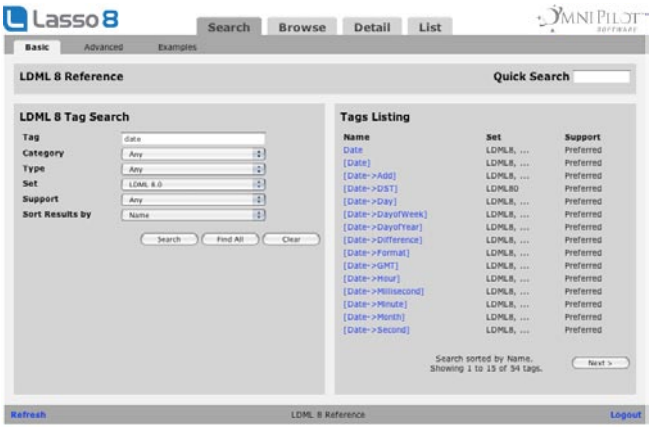
  http://ldml.omnipilot.com/

  This version contains the same information as the locally-stored LDML 8 Reference, however, it also contains documentation comments and code examples from users and developers. This is useful for finding further examples and information about particular tags.

Components

The local version of the LDML 8 Reference consists of two components. The interface is provided by the LDMLReference.LassoApp file located in the Lasso directory of the Web server root. The data for the reference is stored within Lasso MySQL in a database named LDML7_Reference. Both components are installed as part of the standard Lasso Professional 8 installation.

Figure 1: LDML 8 Reference



Sections of the Interface

The interface is divided into four sections, navigable via tabs at the top of the screen. These sections are:

- *Search* – Allows searching the LDML 8 Reference database.
- *Browse* – Allows browsing the LDML 8 Reference database by category.
- *Detail* – Shows descriptions and examples of specific LDML tags.
- *List* – Shows a listing of all available LDML tags summarized by category.

Navigation

Navigation occurs by selecting the tab for the desired section at the top of the interface. Doing so will display the default screen for that tab and additional tabs for any subsections. Many screens contain two panels. The left panel generally provides a search interface or a list of options. The right panel provides search results or details for any selected option.

Navigation within extended lists occurs via Prev and Next buttons. Listings are displayed in groups of ten or fifteen depending on the section.

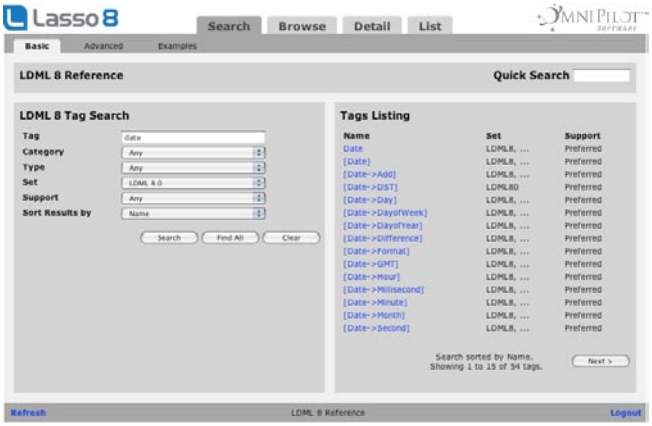
Search

This section describes searching for LDML tags in the LDML 8 Reference using the Search section of the interface.

Basic Searching

The Basic page allows one to specify a basic search for LDML tags and view the results. LDML 8 preferred tags and their synonyms, and abbreviations will be returned as well as symbols and delimiters.

Figure 2: Basic Search Page



Tags can be searched by entering or selecting values from the following fields, and then selecting the Search button:

- **Tag** – Specifies the LDML tag by name.
- **Category** – Pull-down menu listing all 30 tag categories.
- **Type** – Pull-down menu listing all possible tag types.
- **Set** – Pull-down menu listing all available tag sets. All preferred Lasso Professional 8 tags belong to the LDML 8.0 set.
- **Support** – Pull-down menu listing the types of tag support in Lasso Professional 8. A Preferred tag is part of the core syntax for LDML 8. An Abbreviation is an abbreviation of a preferred tag. A Synonym is a synonym of a preferred tag. A Deprecated tag is supported in LDML 8, but may not be supported in a future version.

  Note: Deprecated tags can only be searched using the Advanced search page.

- **Sort Results By** – Allows results to be sorted by tag name, type, set, or support.

Selecting the Find All button finds all LDML 8 tags in the LDML 8 Reference. Selecting the Clear button resets all search fields for a new search.

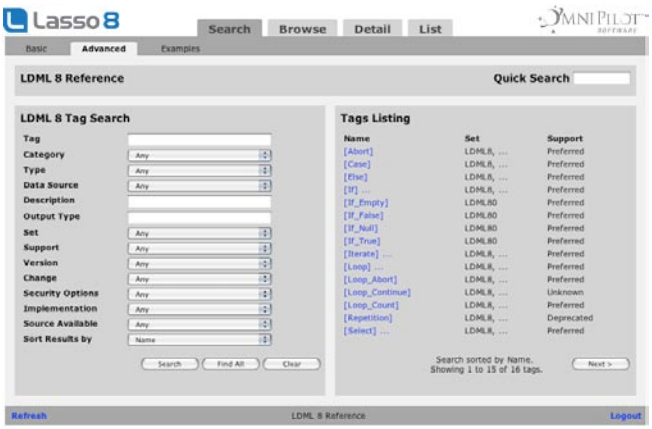
Search Results

Search results are displayed in the Tags Listing panel, which appears to the right. The Prev and Next buttons are shown if more results are returned than can be shown. Selecting the name of a tag takes one to the *Detail > Tag* page for that particular tag.

Advanced Searching

The Advanced page provides the same search fields and functionality as the Basic page. The results from the Advanced page include deprecated and unsupported tags in addition to the preferred tags returned by basic searches.

Figure 3: Advanced Search Page



Many additional search options are available including:

- **Data Source** – Specifies the data source for which the tag is used.
- **Description** – Allows searching within the tag description.
- **Output Type** – Allows searching for tags that output a value of a particular data type, e.g. Array.
- **Version** – Specifies the version of Lasso from which the tag originated (e.g. 7.0, 6.0, 5.0, 3.6.6.2, etc.).

- **Change –** Specifies whether a tag is new, updated, or unchanged between the last major release and the current release.
- **Security Options –** Specifies whether the tag is controlled by Lasso Security. The options are Classic Lasso for tags that are disabled when Classic Lasso support is disabled, Tag Permissions for tags that can be enabled or disabled by tag permissions, File Permissions for tags that can be enabled or disabled by file permissions, Database Permissions for tags that depend on database or table-level security settings, and LJAPI for tags that are disabled when LJAPI support is disabled.
- **Implementation –** Specifies the implementation of the tag. This can be one of the following:

  LDML – Implemented in LDML as part of the in Startup.LassoApp file.

  LCAPI – Implemented in C++.

  LJAPI – Implemented in Java. Will not work without a JRE installed on the system.

  Internal – Implemented in C++ as a core internal language construct. These tags have the lowest-level implementation in LDML.
- **Source Available –** Specifies whether or not the tag source code is available.

Selecting the Find All button finds all LDML 8 tags in the LDML 8 Reference. Selecting the Clear button resets all search fields for a new search.

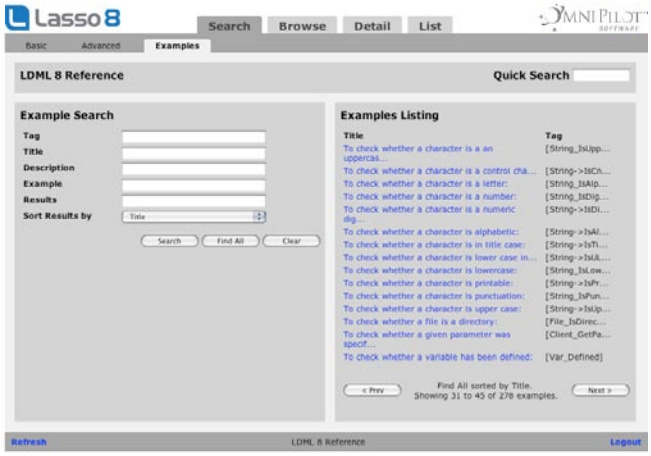
Search Results

Search results are displayed in the Tags Listing panel, which appears to the right. The Prev and Next buttons are shown if more results are returned than can be shown. Selecting the name of a tag takes one to the *Detail > Tag* page for that particular tag.

Examples Searching

The Examples page allows any of the tag examples to be searched.

Figure 4: Examples Search Page



The following search options are available.

- **Tag** – Specifies the tag for which the example is shown.
- **Title** – The title of each example can be searched.
- **Description** – The description of each example can be searched.
- **Example** – The text of each example's code can be searched.
- **Results** – The text of each example's results can be searched.

Selecting the Find All button finds all examples which have been entered in the LDML 8 Reference. Selecting the Clear button resets all search fields for a new search.

Search Results

Search results are displayed in the Examples Listing panel, which appears to the right. The Prev and Next buttons are shown if more results are returned than can be shown. Selecting the title of an example takes one to the *Detail > Tag* page for that particular example.

Quick Search

A Quick Search field appears in the upper right corner of every page. Entering text in the Quick Search field and pressing Return or Enter on the keyboard performs a basic search on the tag name field and returns results to the

Tags Listing panel in the *Search > Basic* page. The last search term entered is displayed in the Quick Search field until a new term is entered or a new search is performed.

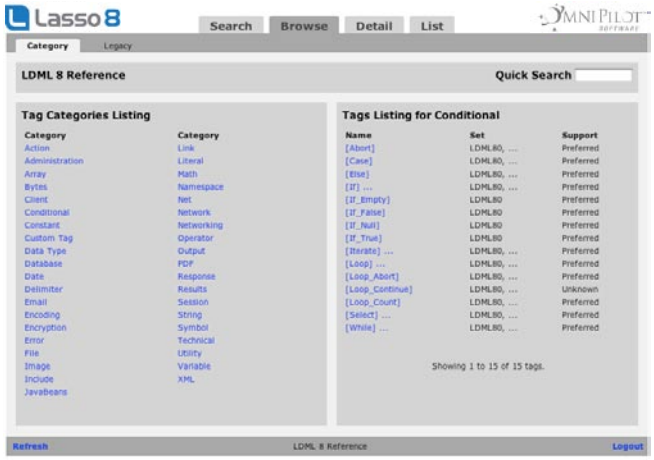
Browse

The Browse page allows one to browse the LDML 8 Reference by tag category and tag name for information about LDML tags.

Browsing by Category

The Category page allows one to browse the LDML 8 Reference by tag category and tag name for information about LDML tags.

Figure 5: Category Tags Page



Viewing Tag Categories

The Tag Categories Listing panel shows a listing of all the tag categories in LDML 8, except legacy tags, which are covered in the next section.

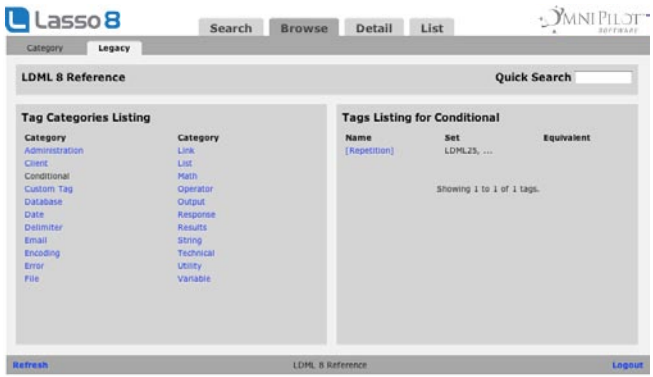
Tags Listing

When a category is selected in the Tag Categories Listing panel, it shows all tags in that category in the Tags Listing panel, which appears to the right. Prev and Next buttons appear for navigation if there are more than ten tags in a selected category. Selecting the name of a tag takes one to the Tag page with the current tag selected, which is described later in this chapter.

Browsing Legacy Tags

The Legacy page allows one to browse all legacy tags in the LDML 8 Reference.

Figure 6: Legacy Tags Page



Legacy tags include all deprecated tags from LDML 3 and earlier. As support for select legacy tags may be dropped in future releases of Lasso Professional, using these tags to build Lasso solutions is not recommended.

One is able to browse all legacy tags in the Legacy page in the same manner as in the Category page, covered in the previous section.

Detail

The Detail section shows information and examples about any selected tag.

Tag Detail

The Tag page shows all information about a selected tag. One is taken here after selecting a tag from the Search, Browse, or List sections. All information is shown in the left panel, and includes the following:

- **Description** – Defines what a tag does, and how and where it is used.
- **Syntax** – Shows the syntax for the tag.
- **Parameters** – Lists all parameters or modifiers that can be used with the tag. Required Parameters must be present in the tag syntax for the tag to work properly, while Optional Parameters do not.

- **Examples** – Provides examples of how the tag can be used to perform a specific function within a Lasso solution.
- **Change Notes** – Provides information about how a tag has changed from different versions of Lasso, and if applicable, what tag it replaces.

Figure 7: Tag Detail Page

The top panel shows the current tag selected. If a search was performed, one can navigate through the found set by selecting the Prev and Next buttons. If no search was performed, the Prev and Next buttons will navigate through the tags in each category alphabetically.

The right panel lists the following tag information:

- **Category** – Specifies the tag category (e.g. Array, Encoding, etc.). Selecting the tag category displays the *Browse > Category* page.
- **Type** – Specifies the tag type (e.g. Command, Container, etc.).
- **Set** – Specifies the versions of LDML in which the tag is supported. All native Lasso Professional 8 tags belong to the LDML 8.0 set.
- **Support** – Specifies the tag support in Lasso Professional 8. A Preferred tag is part of the core syntax for LDML 8. An Abbreviation is an abbreviation of a preferred tag. A Synonym is a synonym of a preferred tag. A Deprecated tag is supported in LDML 8, but support may be dropped in a future version of Lasso. Deprecated tags are not recommended for use in new projects. Any returns all support types.
- **Version** – Specifies the version of Lasso from which the tag originated (e.g. 7.0, 6.0, 5.0, 3.6.6.2, etc.).
- **Change –** Specifies whether a tag is new, updated, or unchanged between the last major release and the current release.

- **Data Source** – Specifies the data source with which the tag can be used.
- **Output Type** – Specifies what data type the tag will output. Many tags output multiple data types in which case each data type or Any is shown.
- **Security** – Specifies whether access to the tag can be controlled through Lasso Administration. Options include Classic for tags that are disabled with Classic Lasso, Tag for tags that are controlled by tag permissions, File for tags that are controlled by file permissions, Database for tags that are controlled by database permissions, and LJAPI for tags that are disabled if LJAPI support is disabled.

The lower right panel contains links to other tags in the database. The following types of tags are listed.

- **Synonyms** – Lists any tags that are synonyms of the current tag. Synonyms accept the same parameters and can be used interchangeably.
- **Abbreviations** – Lists any abbreviations for the current tag.
- **Related Tags** – Lists any related tags, which are tags that have similar functions or are used in a similar manner.
- **Required Tags** – Lists all tags and technologies (e.g. Java) that are required for the selected tag to work.
- **LDML 3 Equivalent** – For tags which have been updated since LDML 3, an LDML 3 tag is listed that provides similar functionality to the current tag.
- **LDML 8 Equivalent** – For tags which are not preferred LDML 8 syntax, an equivalent LDML 8 tag is listed that provides similar functionality to the current tag.

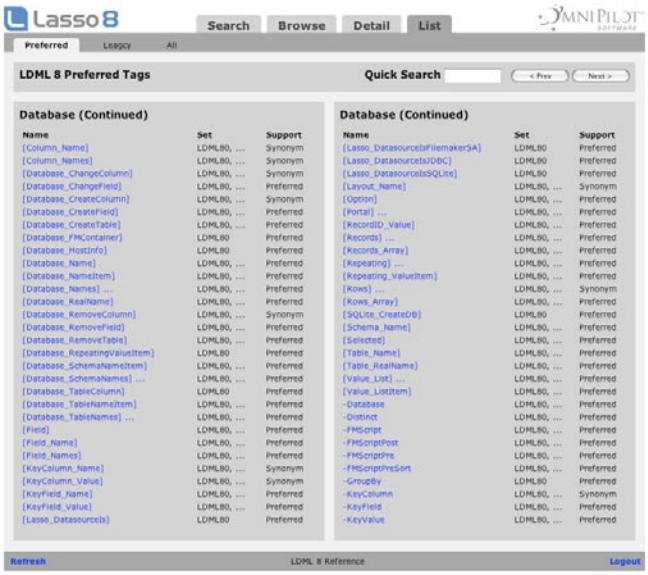
List

The List section provides a listing of all LDML tags by category.

Preferred Tags

The Preferred page provides a listing of all preferred tags, which represent the core syntax for Lasso Professional 8.

Figure 8: Preferred Tags Page



All tags are listed alphabetically beneath their category name (e.g. Array, Database, etc.) and the list spans both panels. The listing can be navigated by selecting the Prev and Next buttons at the top of the page. Selecting a tag name takes one to the Tag page, covered in the previous section.

Legacy Tags

The Legacy page provides a listing of all legacy tags, which are deprecated tags from LDML 7, LDML 6, LDML 5, and LDML 3.

All tags are listed alphabetically beneath their category name (e.g. Array, Database, etc.) and the list spans both panels. The listing can be navigated by selecting the Prev and Next buttons at the top of the page. Selecting a tag name takes one to the Tag page, covered earlier in this chapter.

All Tags

The All page provides a listing of all LDML tags available in LDML 8 including preferred tags and legacy tags. All tags are listed alphabetically, and span both panels. The listing can be navigated by selecting the Prev and Next buttons at the top of the page. Selecting a tag name takes one to the Tag page, covered earlier in this chapter.

II

Section II
Database Interaction

This section includes an introduction to interacting with databases in Lasso Professional 8 and more specific discussions of particular database actions and tags and techniques particular to Lasso MySQL and FileMaker Pro databases.

- *Chapter 7: Database Interaction Fundamentals* introduces the concepts required to work with databases in Lasso Professional 8.
- *Chapter 8: Searching and Displaying Data* discusses how to create search queries and display the results of those queries.
- *Chapter 9: Adding and Updating Records* discusses how to create queries to add, update, and delete database records.
- *Chapter 10: MySQL Data Sources* documents tags specific to the MySQL data source connector including tags to create database schema programmatically.
- *Chapter 11: SQLIte Data Sources* documents tags specific to the SQLite data source connector and MySQL data source connector.
- *Chapter 12: FileMaker Data Sources* documents tags specific to the FileMaker Pro and FileMaker Server Advanced data source connectors including tags to execute FileMaker scripts, return images from a FileMaker container field, and display information in repeating fields and portals.
- *Chapter 13: JDBC Pro Data Sources* documents tags specific to the JDBC data source connector.

7

Chapter 7
Database Interaction Fundamentals

One of the primary purposes of LDML is to perform database actions which are a combination of pre-defined and visitor-defined parameters and to format the results of those actions. This chapter introduces the fundamentals of specifying database actions in LDML.

- *Inline Database Actions* includes full details for how to use the [Inline] tag to specify database actions.
- *Action Parameters* describes how to get information about an action.
- *Results* includes information about how to return details of an LDML database action.
- *Showing Database Schema* describes the tags that can be used to examine the schema of a database.
- *SQL Statements* describes the -SQL command tag and how to issue raw SQL statements to SQL-compliant data sources.
- *SQL Transactions* describes how to perform reversible SQL transactions using Lasso.

Inline Database Actions

The [Inline] ... [/Inline] container tags are used to specify a database action and to present the results of that action within a Lasso format file. The database action is specified using parameters as keyword/value parameters within the opening [Inline] tag. Additional name/value parameters specify the user-defined parameters of the database action. A single action can be specified in an [Inline]. Additional actions can be performed in subsequent or nested [Inline] ... [/Inline] tags.

Table 1: Inline Tag

Tag/Parameter	Description
[Inline] ... [/Inline]	Performs the database action specified in the opening tag. The results of the database action are available inside the container tag.
-InlineName	Specifies a name for the inline. The same name can be used with the [Records] ... [/Records] tags to return the records from the inline later on the page. (Optional)
-Log	Specifies at what log level the statement from the inline should be logged. Values include None, Detail, Warning, and Critical. If not specified then the default log level for action statements will be used.
-StatementOnly	Specifies that the inline should generate the internal statement required to perform the action, but not actually perform the action. The statement can be fetched with [Action_Statement].

The results of the database action can be displayed within the contents of the [Inline] ... [/Inline] container tags using the [Records] ... [/Records] container tags and the [Field] substitution tag. Alternately, the [Inline] can be named and the results can be displayed later.

The entire database action can be specified directly in the opening [Inline] tag or visitor-defined aspects of the action can be retrieved from an HTML form submission. [Link_...] tags can be used to navigate a found set in concert with the use of [Inline] ... [/Inline] tags. Nested [Inline] ... [/Inline] tags can be used to create complex database actions.

Inlines can log the statement (SQL or otherwise) that they generate. The optional -Log parameter controls at what level the statement is logged. Setting -Log to None will suppress logging from the inline. If no -Log is specified then the default log-level set for the datasource in Site Administration will be used.

The -StatementOnly option instructs the datasource to generate the implementation-specific statement required to perform the desired database action, but not to actually perform it. The generated statement can be returned with [Action_Statement]. This is useful in order to see the statement Lasso will generate for an action, perform some modifications to that statement, then re-issue the statement using -SQL in another inline.

To change the log level for an inline database action:

Use the -Log parameter within the opening [Inline] tag.

- Suppress the action statement from being logged by setting -Log='None'. The action statement will not be logged no matter how the various log levels are routed.

```
[Inline: -Search, -Database='Example', -Table='Example', -Log='None', ...]
  ...
[/Inline]
```

- Log the action statement at the critical log level by setting -Log='Critical'. This can be useful when debugging a Web site since the action statement generated by this inline can be seen even if action statements are generally being suppressed by the log routing preferences..

```
[Inline: -Search, -Database='Example', -Table='Example', -Log='Critical', ...]
  ...
[/Inline]
```

To see the action statement generated by an inline database action:

Use the [Action_Statement] tag within the [Inline] ... [/Inline] tags. The tag will return the action statement that was generated by the data source connector to fulfill the specified database action. For SQL data sources like MySQL and SQLite a SQL statement will be returned. Other data sources may return a different style of action statement.

```
[Inline: -Search, -Database='Example', -Table='Example', ...]
  [Action_Statement]
  ...
[/Inline]
```

To see the action statement that would be generated by the data source without actually performing the database action the -StatementOnly parameter can be specified in the opening [Inline] tag. The [Action_Statement] tag will return the same value is would for a normal inline database action, but the database action will not actually be performed.

```
[Inline: -Search, -Database='Example', -Table='Example', -StatementOnly, ...]
  [Action_Statement]
  ...
[/Inline]
```

Database Actions

A database action is performed to retrieve data from a database or to manipulate data which is stored in a database. Database actions can be used in Lasso to query records in a database that match specific criteria, to return a particular record from a database, to add a record to a database, to delete a record from a database, to fetch information about a database, or to navigate through the found set from a database search. In addition, database actions can be used to execute SQL statements in compliant databases.

The database actions in Lasso are defined according to what action parameter is used to trigger the action. The following table lists the parameters which perform database actions that are available in LDML.

Table 2: Inline Database Action Parameters

Tag	Description
-Search	Finds records in a database that match specific criteria, returns detail for a particular record in a database, or navigates through a found set of records.
-FindAll	Returns all records in a specific database table.
-Random	Returns a single, random record from a database table.
-Add	Adds a record to a database table.
-Update	Updates a specific record from a database table.
-Duplicate	Duplicates a specific record in a database table. Only works with FileMaker Pro databases.
-Delete	Removes a specified record from a database table.
-Show	Returns information about the tables and fields within a database.
-SQL	Executes a SQL statement in a compatible data source. Only works with Lasso MySQL and other SQL databases.
-Nothing	The default action which performs no database interaction, but simply passes the parameters of the action.

Note: *Table 2: Database Action Parameters* lists all of the database actions that Lasso supports. Individual data source connectors may only support a subset of these parameters. The Lasso Connector for Lasso MySQL and the Lasso Connector for MySQL do not support the -Duplicate action. The Lasso Connector for FileMaker Pro does not support the -SQL action. See the documentation for third party data source connectors for information about what parameters they support.

Each database action parameter requires additional parameters in order to execute the proper database action. These parameters are specified using additional parameters and name/value pairs. For example, a -Database parameter specifies the database in which the action should take place and a -Table parameter specifies the specific table from that database in which the action should take place. Name/value pairs specify the query for a -Search action, the initial values for the new record created by an -Add action, or the updated values for an -Update action.

Full documentation of which [Inline] parameters are required for each action are detailed in the section specific to that action in this chapter, the *Searching and Displaying Data* chapter, or the *Adding and Updating Records* chapter.

Example of specifying a -FindAll action within an [Inline]:

The following example shows an [Inline] ... [/Inline] tag that has a -FindAll database action specified in the opening tag. The [Inline] tag includes a -FindAll parameter to specify the action, -Database and -Table parameters to specify the database and table from which records should be returned, and a -KeyField parameter which specifies the key field for the table. The entire database action is hard-coded within the [Inline] tag.

The tag [Found_Count] returns how many records are in the database. The [Records] ... [/Records] container tags repeat their contents for each record in the found set. The [Field] tags are repeated for each found record creating a listing of the names of all the people stored in the Contacts database.

```
[Inline: -FindAll,
  -Database='Contacts',
  -Table='People',
  -KeyField='ID']
  There are [Found_Count] record(s) in the People table.
  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]
```

```
➜ There are 2 record(s) in the People table.
John Doe
Jane Doe
```

Example of specifying a -Search action within an [Inline]:

The following example shows an [Inline] ... [/Inline] tag that has a -Search database action specified in the opening tag. The [Inline] tag includes a -Search parameter to specify the action, -Database and -Table parameters to specify the database and table records from which records should be returned, and a -KeyField parameter which specifies the key field for the table. The

subsequent name/value parameters, 'First_Name'='John' and 'Last_Name'='Doe', specify the query which will be performed in the database. Only records for John Doe will be returned. The entire database action is hard-coded within the [Inline] tag.

The tag [Found_Count] returns how many records for John Doe are in the database. The [Records] ... [/Records] container tags repeat their contents for each record in the found set. The [Field] tags are repeated for each found record creating a listing of all the records for John Doe stored in the Contacts database.

```
[Inline: -Search,
  -Database='Contacts',
  -Table='People',
  -KeyField='ID',
  'First_Name'='John',
  'Last_Name'='Doe']
  There were [Found_Count] record(s) found in the People table.
  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]
```

```
➜ There were 1 record(s) found in the People table.
  John Doe
```

Using HTML Forms

The previous two examples show how to specify a hard-coded database action completely within an opening [Inline] tag. This is an excellent way to embed a database action that will be the same every time a page is loaded, but does not provide any room for visitor interaction.

A more powerful technique is to use values from an HTML form or URL to allow a site visitor to modify the database action which is performed within the [Inline] tag. The following two examples demonstrate two different techniques for doing this using the singular [Action_Param] tag and the array-based [Action_Params] tag.

Example of using HTML form values within an [Inline] with [Action_Param]:

An inline-based database action can make use of visitor specified parameters by reading values from an HTML form which the visitor customizes and then submits to trigger the page containing the [Inline] ... [/Inline] tags.

The following HTML form provides two inputs into which the visitor can type information. An input is provided for First_Name and one for Last_Name. These correspond to the names of fields in the Contacts data-

base. The action of the form is set to response.lasso which will contain the [Inline] ... [/Inline] tags that perform the actual database action. The action tag specified in the form is -Nothing which instructs Lasso to perform no database action when the form is submitted.

```
<form action="/response.lasso" method="POST">
  <br>First Name: <input type="text" name="First_Name" value="">
  <br>Last Name: <input type="text" name="Last_Name" value="">
  <br><input type="submit" value="Search">
</form>
```

The [Inline] tag on response.lasso contains the name/value parameter 'First_Name'=(Action_Param: 'First_Name'). The [Action_Param] tag instructs Lasso to fetch the input named First_Name from the action which resulted in the current page being served, namely the form shown above. The [Inline] contains a similar name/value parameter for Last_Name.

```
[Inline: -Search,
  -Database='Contacts',
  -Table='People',
  -KeyField='ID',
  'First_Name'=(Action_Param: 'First_Name'),
  'Last_Name'=(Action_Param: 'Last_Name')]
  There were [Found_Count] record(s) found in the People table.
  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]
```

If the visitor entered Jane for the first name and Doe for the last name then the following results would be returned.

```
➜ There were 1 record(s) found in the People table.
  Jane Doe
```

As many parameters as are needed can be named in the HTML form and then retrieved in the response page and incorporated into the [Inline] tag.

Note: The [Action_Param] tag is equivalent to the [Form_Param] tag used in prior versions of Lasso.

Example of using an array of HTML form values within an [Inline] with [Action_Params]:

Rather than specifying each [Action_Param] individually, an entire set of HTML form parameters can be entered into an [Inline] tag using the array-based [Action_Params] tag. Inserting the [Action_Params] tag into an [Inline] functions as if all the parameters and name/value pairs in the HTML form were placed into the [Inline] at the location of the [Action_Params] parameter.

The following HTML form provides two inputs into which the visitor can type information. An input is provided for First_Name and one for Last_Name. These correspond to the names of fields in the Contacts database. The action of the form is set to response.lasso which will contain the [Inline] ... [/Inline] tags that perform the actual database action. The database action is -Nothing which instructs Lasso to perform no database action when the HTML form is submitted.

```
<form action="/response.lasso" method="POST">
  <br>First Name: <input type="text" name="First_Name" value="">
  <br>Last Name: <input type="text" name="Last_Name" value="">
  <br><input type="submit" value="Search">
</form>
```

The [Inline] tag on response.lasso contains the array parameter [Action_Params]. This instructs Lasso to take all the parameters from the HTML form or URL which results in the current page being loaded and insert them in the [Inline] as if they had been typed at the location of [Action_Params]. This will result in the name/value pairs for First_Name, Last_Name, and the -Nothing action to be inserted into the [Inline]. The latest action specified has precedence so the -Search tag specified in the actual [Inline] tag overrides the -Nothing which is passed from the HTML form.

```
[Inline: (Action_Params),
  -Search,
  -Database='Contacts',
  -Table='People',
  -KeyField='ID']
  There were [Found_Count] record(s) found in the People table.
  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]
```

If the visitor entered Jane for the first name and Doe for the last name then the following results would be returned.

```
➜ There were 1 record(s) found in the People table.
  Jane Doe
```

As many parameters as are needed can be named in the HTML form. They will all be incorporated into the [Inline] tag at the location of the [Action_Params] tag. Any parameters in the [Inline] after the [Action_Params] tag will override conflicting settings from the HTML form.

Note: [Action_Params] is a replacement for the -ReUseFormParams keyword in prior versions of Lasso. See the *Upgrading Your Solutions* section for more information.

HTML Form Response Pages

Every HTML form or URL needs to have a response page specified so Lasso knows what format file to process and return as the result of the action. The referenced format file could contain simple HTML or complex LDML calculations, but some format file must be specified.

To specify a format file within an HTML form or URL:

- The HTML form action can be set to the location of a format file. For example, the following HTML <form> tag references the file /response.lasso in the root of the Web serving folder.

```
<form action="/response.lasso" method="POST"> ... </form>
```

- The URL can reference the location of a format file before the question mark ? delimiter. For example, the following anchor tag references the file response.lasso in the same folder as the page in which this anchor is contained.

```
<a href="response.lasso?Name=Value"> Link </a>
```

- The HTML form can reference /Action.Lasso and then specify the path to the format file in a -Response tag. For example, the following HTML <form> tag references the file response.lasso in the root of the Web serving folder. The path is relative to the root because the placeholder /Action.Lasso is specified with a leading forward slash /.

```
<form action="/Action.Lasso" method="POST">
  <input type="hidden" name='-Response" value="response.lasso">
</form>
```

- The URL can reference Action.Lasso and then specify the path to the format file in a -Response tag. For example, the following anchor tag references the file response.lasso in the same folder as the page in which the link is specified. The path is relative to the local folder because the placeholder Action.Lasso is specified without a leading forward slash /.

```
<a href="Action.Lasso?-Response=response.lasso"> Link </a>
```

The -Response tag can be used on its own or action specific response tags can be used so a form is sent to different response pages if different actions are performed using the form. Response tags can also be used to send the visitor to different pages if different errors happen when the database action is attempted by Lasso. The following table details the available response tags.

Table 3: Response Parameters

Tag	Description
-Response	Default response tag. The value for this response tag is used if no others are specified.
-ResponseAnyError	Default error response tag. The value for this response tag is used if any error occurs and no more specific error response tag is set.
-ResponseReqFieldMissingError	Error to use if a -Required field is not given a value by the visitor.
-ResponseSecurityError	Error to use if a security violation occurs because the current visitor does not have permission to perform the database action.

See the *Error Control* chapter for more information about using the error response pages.

Setting HTML Form Values

If the format file containing an HTML form is the response to an HTML form or URL, then the values of the HTML form inputs can be set to values retrieved from the previous format file using [Action_Param].

For example, if a form is on default.lasso and the action of the form is default.lasso then the same page will be reloaded with new form values each time the form is submitted. The following HTML form uses [Action_Param] tags to automatically restore the values the user specified in the form previously, each time the page is reloaded.

```
<form action="default.lasso" method="POST">
  <br>First Name:
    <input type="hidden" name="First_Name" value="[Action_Param: 'First_Name']">
  <br>First Name:
    <input type="hidden" name="Last_Name" value="[Action_Param: 'Last_Name']">
  <br><input type="submit" value="Submit">
</form>
```

Tokens

Tokens can be used with HTML forms and URLs in order to pass data along with the action. Tokens are useful because they do not affect the operation of a database action, but allow data to be passed along with the action. For example, meta-data could be associated with a visitor to a Web site without using sessions or cookies.

- Tokens can be set in a form using the -Token.TokenName=TokenValue parameter. Multiple named tokens can be set in a single form.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-Token.TokenName" value="TokenValue">
</form>
```

- Tokens can be set in a URL using the -Token.TokenName=TokenValue parameter. Multiple named tokens can be set in a single URL.

```
<a href="response.lasso?-Token.TokenName=TokenValue"> Link </a>
```

- Tokens set in an HTML form or URL are available in the response page of the database action. Tokens are not available inside [Inline] ... [/Inline] tags on the responses page unless they are explicitly set within the [Inline] tag itself.
- Tokens can be set in an [Inline] using the -Token.TokenName=TokenValue parameter. Multiple named tokens can be set in a single [Inline].
- Tokens set in an [Inline] are only available immediately inside the [Inline]. They are not available to nested [Inlines] unless they are set specifically within each [Inline].
- By default, tokens are included in the [Link_...] tags and in [Action_Params]. Unless specifically set otherwise, tokens will be redefined on pages which are returned using the [Link_...] tags.

Nesting Inline Database Actions

Database actions can be combined to perform compound database actions. All the records in a database that meet certain criteria could be updated or deleted. Or, all the records from one database could be added to a different database. Or, the results of searches from several databases could be merged and used to search another database.

Database actions are combined by nesting [Inline] ... [/Inline] tags. For example, if [Inline] ... [/Inline] tags are placed inside the [Records] ... [/Records] container tag within another set of [Inline] ... [/Inline] tags then the inner [Inline] will execute once for each record found in the outer [Inline].

All database results tags function for only the innermost set of [Inline] ... [/Inline] tags. Variables can pass through into nested [Inline] ... [/Inline] tags, but tokens cannot, these need to be reset in each [Inline] tag in the hierarchy.

SQL Note: Nested inlines can also be used to perform reversible SQL transactions in transaction-compliant SQL data sources. See the ***SQL Transactions*** section at the end of this chapter for more information.

Example of nesting [Inline] ... [/Inline] tags:

This example will use nested [Inline] ... [/Inline] tags to change the last name of all people whose last name is currently Doe in a database to Person. The outer [Inline] ... [/Inline] tags perform a hard-coded search for all records with Last_Name equal to Doe. The inner [Inline] ... [/Inline] tags update each record so Last_Name is now equal to Person. The output confirms that the conversion went as expected by outputting the new values.

```
[Inline: -Search,
  -Database='Contacts',
  -Table='People',
  -KeyField='ID',
  'Last_Name'='Doe',
  -MaxRecords='All']
  [Records]
    [Inline: -Update,
      -Database='Contacts',
      -Table='People',
      -KeyField='ID',
      -KeyValue=(KeyField_Value),
      'Last_Name'='Person']
      <br>Name is now [Field: 'First_Name'] [Field: 'Last_Name']
    [/Inline]
  [/Records]
[/Inline]
```

```
➜ Name is now Jane Person
  Name is now John Person
```

Array Inline Parameters

Most LDML parameter can be used within an [Inline] tag to specify an action. In addition, parameters and name/value parameters can be stored in an array and then passed into an [Inline] as a block. Any single value in an [Inline] which is an array data type will be interpreted as a series of parameters inserted at that location in the array. This technique is useful for programmatically assembling database actions.

Many parameters can only take a single value within an [Inline] tag. For example, only a single action can be specified and only a single database can be specified. The last action parameter defines the value that will be used for the action. The last, for example, -Database parameter defines the value that will be used for the database of the action. If an array parameter comes first in an [Inline] then all subsequent parameters will override any conflicting values within the array parameter.

Example of using an array to pass values into an [Inline]:

The following LassoScript performs a -FindAll database action with the parameters first specified in an array and stored in the variable Params, then passed into the opening [Inline] tag all at once. The value for -MaxRecords in the [Inline] tag overrides the value specified within the array parameter since it is specified later. Only the number of records found in the database are returned.

```
<?LassoScript
  Variable: 'Params'=(Array:
    -FindAll='',
    -Database='Contacts',
    -Table='People',
    -MaxRecords=50
  );
  Inline: (Var: 'Params'), -MaxRecords=100;
    'There are ' + (Found_Count) + 'record(s) in the People table.';
  /Inline;
?>
```

```
➜ There are 2 record(s) in the People table.
```

Action Parameters

LDML has a set of substitution tags which allow for information about the current action to be returned. The parameters of the action itself can be returned or information about the action's results can be returned.

The following table details the substitution tags which allow information about the current action to be returned. If these tags are used within an [Inline] ... [/Inline] container tag they return information about the action specified in the opening [Inline] tag. Otherwise, these tags return information about the action which resulted in the current format file being served.

Even format files called with a simple URL such as http://www.example.com/response.lasso have an implicit -Nothing action. Many of these substitution tags return default values even for the -Nothing action.

Table 4: Action Parameter Tags

Tag	Description
[Action_Param]	Returns the value for a specified name/value parameter. Equivalent to [Form_Param].
[Action_Params]	Returns an array containing all of the parameters and name/value parameters that define the current action.
[Action_Statement]	Returns the statement that was generated by the datasource to implement the requested action. For SQL datasources this will return a SQL statement. Other datasources may return different values.
[Database_Name]	Returns the name of the current database.
[KeyField_Name]	Returns the name of the current key field.
[KeyField_Value]	Returns the name of the current key value if defined. Equivalent to [RecordID_Value].
[Lasso_CurrentAction]	Returns the name of the current Lasso action.
[MaxRecords_Value]	Returns the number of records from the found set that are currently being displayed.
[Operator_LogicalValue]	Returns the value for the logical operator.
[Response_FilePath]	Returns the path to the current format file.
[SkipRecords_Value]	Returns the current offset into a found set.
[Table_Name]	Returns the name of the current table. Equivalent to [Layout_Name].
[Token_Value]	Returns the value for a specified token.
[Search_Arguments]	Container tag repeats once for each name/value parameter of the current action.
[Search_FieldItem]	Returns the name portion of a name/value parameter of the current action.
[Search_OperatorItem]	Returns the operator associated with a name/value parameter of the current action.
[Search_ValueItem]	Returns the value portion of a name/value parameter of the current action.
[Sort_Arguments]	Container tag repeats once for each sort parameter.
[Sort_FieldItem]	Returns the field which will be sorted.
[Sort_OrderItem]	Returns the order by which the field will be sorted.

The individual substitution tags can be used to return feedback to site visitors about what database action is being performed or to return debugging information. For example, the following code inserted at the top of a response page to an HTML form or URL or in the body of an [Inline] ... [/Inline] tag will return details about the database action that was performed.

```
Action: [Lasso_CurrentAction]
Database: [Database_Name]
Table: [Table_Name]
Key Field: [KeyField_Name]
KeyValue: [KeyField_Value]
MaxRecords: [MaxRecords_Value]
SkipRecords: [SkipRecords_Value]
Logical Operator: [Operator_LogicialValue]
Statement: [Action_Statement]

➜ Action: Find All
Database: Contacts
Table: People
Key Field: ID
KeyValue: 100001
MaxRecords: 50
SkipRecords: 0
Logical Operator: AND
Statement: SELECT * FROM Contacts.People LIMIT 50
```

The [Action_Params] tag can be used to return information about the entire Lasso action in a single array. Rather than assembling information using the individual substitution tags it is often easier to extract information from the [Action_Params] array. The schema of the array returned by [Action_Params] is detailed in *Table 5: [Action_Params] Array Schema*.

The schema shows the names of the values which are returned in the array. Even if -Layout is used to specify the layout for a database action, the value of that tag is returned after -Table in the [Action_Params] array.

To output the parameters of the current database action:

The value of the [Action_Params] tag in the following example is formatted to show the elements of the returned array clearly. The [Action_Params] array contain values for -MaxRecords, -SkipRecords, and -OperatorLogical even though these aren't specified in the [Inline] tag.

```
[Inline: -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID']
  [Action_Params]
[/Inline]

➜ (Array:
    (Pair: (-Search) = ()),
    (Pair: (-Database) = (Contacts)),
    (Pair: (-Table) = (People)),
    (Pair: (-KeyField) = (ID)),
```

```
    (Pair: (-MaxRecords) = (50)),
    (Pair: (-SkipRecords) = (0)),
    (Pair: (-OperatorLogical) = (AND))
)
```

Table 5: [Action_Params] Array Schema

Name	Description
Action	The action parameter is always returned first. The name of the first item is set to the action parameter and the value is left empty.
-Database	If defined, the name of the current database.
-Table	If defined, the name of the current table.
-KeyField	If defined, the name of the field which holds the primary key for the specified table.
-KeyValue	If defined, the particular value for the primary key.
-MaxRecords	Always included. Defaults to 50.
-SkipRecords	Always included. Defaults to 0.
-OperatorLogical	Always included. Defaults to AND.
-ReturnField	If defined, can have multiple values.
-SortOrder, -SortField	If defined, can have multiple values. -SortOrder is always defined for each -SortField. Defaults to ascending.
-Token	If defined, can have multiple values each specified as -Token.TokenName with the appropriate value.
Name/Value Parameter	If defined, each name/value parameter is included.
-Required	If defined, can have multiple values. Included in order within name/value parameters.
-Operator	If defined, can have multiple values. Included in order within name/value parameters.
-OperatorBegin	If defined, can have multiple values. Included in order within name/value parameters.
-OperatorEnd	If defined, can have multiple values. Included in order within name/value parameters.

The [Action_Params] array contains all the parameters and name/value parameters required to define a database action. It does not include any -Response... parameters, the -Username and -Password parameters, -FMScript... parameters, -InlineName keyword or any legacy or unrecognized parameters.

To output the name/value parameters of the current database action:

Loop through the [Action_Params] tag and display only name/value pairs for which the name does not start with a hyphen, i.e. any name/value pairs which do not start with a keyword. The following example shows a search of the People table of the Contacts database for a person named John Doe.

```
[Inline: -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    'First_Name'='John',
    'Last_Name'='Doe']
  [Loop: (Action_Params)->Size]
    [If: !(Action_Params)->(Get: Loop_Count)->(First)->(BeginsWith: '-')]
      <br>[Encode_HTML: (Action_Params)->(Get: Loop_Count)]
    [/If]
  [/Loop]
[/Inline]
```

```
➜ <br>(Pair: (First_Name) = (John))
  <br>(Pair: (Last_Name) = (Doe))
```

To display action parameters to a site visitor:

The [Search_Arguments] ... [/Search_Arguments] container tag can be used in conjunction with the [Search_FieldItem], [Search_ValueItem] and [Search_OperatorItem] substitution tags to return a list of all name/value parameters and associated operators specified in a database action.

```
[Search_Arguments]
  <br>[Search_OperatorItem] [Search_FIeldItem] = [Search_ValueItem]
[/Search_Arguments]
```

The [Sort_Arguments] ... [/Sort_Arguments] container tag can be used in conjunction with the [Sort_FieldItem] and [Sort_OrderItem] substitution tags to return a list of all name/value parameters and associated operators specified in a database action.

```
[Sort_Arguments]
  <br>[Sort_OperatorItem] [Sort_FIeldItem] = [Sort_OrderItem]
[/Sort_Arguments]
```

Results

The following table details the substitution tags which allow information about the results of the current action to be returned. These tags provide information about the current found set rather than providing data from the database or providing information about what database action was performed.

Table 6: Results Tags

Tag	Description
[Field]	Returns the value for a specified field from the result set.
[Found_Count]	Returns the number of records found by Lasso.
[Records] … [/Records]	Loops once for each record in the found set. [Field] tags within the [Records] … [/Records] tags will return the value for the specified field in each record in turn.
[Records_Array]	Returns the complete found set in an array of arrays. The outer array contains one item for every record in the found set. The item for each record is an array containing one item for each field in the result set.
[Shown_Count]	Returns the number of records shown in the current found set. Less than or equal to [MaxRecords_Value].
[Shown_First]	Returns the number of the first record shown from the found set. Usually [SkipRecords_Value] plus one.
[Shown_Last]	Returns the number of the last record shown from the found set.
[Total_Records]	Returns the total number of records in the current table. Works with FileMaker Pro databases only.

Note: Examples of using most of these tags are provided in the following *Searching and Displaying Data* chapter.

The found set tags can be used to display information about the current found set. For example, the following code generates a status message that can be displayed under a database listing.

```
Found [Found_Count] records of [Total_Records] Total.
<br>Displaying [Shown_Count] records from [Shown_First] to [Shown_Last].
```

```
➜ Found 100 records of 1500 Total.
  Displaying 10 records from 61 to 70.
```

These tags can also be used to create links that allow a visitor to navigate through a found set.

Records Array

The [Records_Array] tag can be used to get access to all of the data from an inline operation. The tag returns an array with one element for each record in the found set. Each element is itself an array that contains one element for each field in the found set.

The tag can either be used to quickly output all of the data from the inline operation or can be used with the [Iterate] ... [/Iterate] or other tags to get access to the data programmatically.

```
[Inline: -Search, -Database='Contacts', -Table='People']
  [Records_Array]
[/Inline]

➜ (Array: (Array: (John), (Doe)), (Array: (Jane), (Doe)), ...)
```

The output can be made easier to read using the [Iterate] ... [/Iterate] tags and the [Array->Join] tag.

```
[Inline: -Search, -Database='Contacts', -Table='People']
  [Iterate: Records_Array, (Var: 'Record')]
    "[Encode_HTML: $Record->(Join: '", "')]"<br />
  [/Iterate]
[/Inline]

➜ "John", "Doe"<br />
  "Jane", "Doe"<br />
  ...
```

The output can be listed with the appropriate field names by using the [Field_Names] tag. This tag returns an array that contains each field name from the current found set. The [Field_Names] tag will always contain the same number of elements as the elements of the [Records_Array] tag.

```
[Inline: -Search, -Database='Contacts', -Table='People']
  "[Encode_HTML: Field_Names->(Join: '", "')]"<br />
  [Iterate: Records_Array, (Var: 'Record')]
    "[Encode_HTML: $Record->(Join: '", "')]"<br />
  [/Iterate]
[/Inline]

➜ "First_Name", "Last_Name"<br />
  "John", "Doe"<br />
  "Jane", "Doe"<br />
  ...
```

Together the [Field_Names] and [Records_Array] tags provide a programmatic method of accessing all the data returned by an inline action. When used appropriately these tags can yield better performance than using [Records] ... [/Records], [Field], and [Field_Name] tags.

Showing Database Schema

The schema of a database can be inspected using the [Database_...] tags or the -Show parameter which allows information about a database to be returned using the [Field_Name] tag. Value lists within FileMaker Pro databases can also be accessed using the -Show parameter. This is documented fully in the *FileMaker Pro Data Sources* chapter.

Table 7: -Show Parameter

Tag	Description
-Show	Allows information about a particular database and table to be retrieved.

The -Show parameter functions like the -Search parameter except that no name/value parameters, sort tags, results tags, or operator tags are required. -Show actions can be specified in [Inline] ... [/Inline] tags, HTML forms, or URLs.

Table 8: -Show Action Requirements

Tag	Description
-Show	The action which is to be performed. Required.
-Database	The database which should be searched. Required.
-Table	The table from the specified database which should be searched. Required.
-KeyField	The name of the field which holds the primary key for the specified table. Recommended.

The tags detailed in *Table 9: Schema Tags* allow the schema of a database to be inspected. The [Field_Name] tag must be used in concert with a -Show action or any database action that returns results including -Search, -Add, -Update, -Random, or -FindAll. The [Database_Names] ... [/Database_Names] and [Database_TableNames] ... [/Database_TableNames] tags can be used on their own.

Table 9: Schema Tags

Tag	Description
[Database_Names]	Container tag repeats for every database available to Lasso. Requires internal [Database_NameItem] tag to show results.
[Database_NameItem]	When used inside [Database_Names] … [/Database_Names] container tags returns the name of the current database.
[Database_RealName]	Returns the real name of a database given an alias.
[Database_TableNames]	Container tag repeats for every table within a database. Accepts one required parameter, the name of the database. Requires internal [Database_NameItem] tag to show results. Synonym is [Database_LayoutNames].
[Database_TableNameItem]	When used inside [Database_TableNames] … [/Database_TableNames] container tags returns the name of the current table. Synonym is [Database_LayoutNameItem].
[Field_Name]	Returns the name of a field in the current database and table. A number parameter returns the name of the field in that position within the current table. Other parameters are described below. Synonym is [Column_Name].
[Field_Names]	Returns an array containing all the field names in the current result set. This is the same data as returned by [Field_Name], but in a format more suitable for iterating or other data processing. Synonym is [Column_Names].
[Required_Field]	Returns the name of a required field. Requires one parameter which is the number of the field name to return or a -Count keyword to return the total number of required fields.
[Table_RealName]	Returns the real name of a table given an alias. Requires a -Database parameter which specifies the database in which the table or alias resides.

Note: See the previous ***Records Array*** section for an example of using [Field_Names].

To list all the databases available to Lasso:

The following example shows how to list the names of all available databases using the [Database_Names] ... [/Database_Names] and [Database_NameItem] tags.

```
[Database_Names]
  <br>[Loop_Count]: [Database_NameItem]
[/Database_Name]
```

```
➜ <br>1: Contacts
  <br>2: Examples
  <br>3: Site
```

To list all the tables within a database:

The following example shows how to list the names of all the tables within a database using the [Database_TableNames] ... [/Database_TableNames] and [Database_TableNameItem] tags. The tables within the Site database are listed.

```
[Database_TableNames: 'Site']
  <br>[Loop_Count]: [Database_TableNameItem]
[/Database_TableName]
```

```
➜ <br>1: _outgoingemail
  <br>2: _outgoingemailprefs
  <br>3: _schedule
  <br>4: _sessions
```

To list all the fields within a table:

The [Field_Name] tag accepts a number of optional parameters which allow information about the tags in the current table to be returned. These parameters are detailed in *Table 10: [Field_Name] Parameters*.

Table 10: [Field_Name] Parameters

Parameter	Description
Number	The position of the field name to be returned. Required unless -Count is specified.
-Count	Returns the number of fields in the current table.
-Type	Returns the type of the field rather than the name. Types include Text, Number, Image, Date/Time, Boolean, or Unknown. Requires that a number parameter be specified.
-Protection	Returns the protection status of the field rather than the name. Protection statuses include None or Read Only. Requires that a number parameter be specified.

To return information about the fields in a table:

The following example demonstrates how to return information about the fields in a table using the [Inline] ... [/Inline] tags to perform a -Show action. [Loop] ... [/Loop] tags loop through the number of fields in the table and the name, type, and protection status of each field is returned. The fields within the Contacts Web table are shown.

```
[Inline: -Show,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID']
  [Loop: (Field_Name: -Count)]
    <br>[Loop_Count]: [Field_Name: (Loop_Count)]
      ([Field_Name: (Loop_Count), -Type], [Field_Name: (Loop_Count), -Protection])
  [/Loop]
[/Inline]
```

```
➜ <br>1: Creation Date (Date, None)
  <br>2: ID (Number, Read Only)
  <br>3: First_Name (Text, None)
  <br>4: Last_Name (Text, None)
```

To list all the required fields within a table:

The [Required_Field] tag accepts a number of optional parameters which allow information about the tags in the current table to be returned. These parameters are detailed in *Table 11: [Required_Field] Parameters*.

Table 11: [Required_Field] Parameters

Parameter	Description
Number	The position of the field name to be returned. Required unless -Count is specified.
-Count	Returns the number of required fields in the current table.

The [Required_Field] substitution tag can be used to return a list of all required fields for the current action. A -Show action is used to retrieve the information from the database and then [Loop] ... [/Loop] tags are used to loop through all the required fields. In the example that follows the People table of the Contacts database has only one required field, the primary key field ID.

```
[Inline: -Show,
    -Database='Contacts',
    -Table='People']
  [Loop: (Required_Field: -Count)]
    <br>[Required_Field: (Loop_Count)]
  [/Loop]
[/Inline]
```

➜
ID

SQL Statements

LDML 8 provides the ability to issue SQL statements directly to SQL-compliant data sources, including the built-in Lasso MySQL data source. SQL statements are specified within the [Inline] tag using the -SQL command tag. Many third-party databases that support SQL statements also support the use of the -SQL command tag.

SQL inlines can be used as the primary method of database interaction in Lasso 8, or they can be used along side standard inline actions (e.g. -Search, -Add, -Update, -Delete) where a specific SQL function is desired that cannot be replicated using standard database commands.

SQL Language Note: Documentation of SQL itself is outside the realm of this manual. Please consult the documentation included with your data source for information on what SQL statements are supported by it.

FileMaker Note: The -SQL inline parameter is not supported for FileMaker data sources.

Table 12: SQL Inline Parameters

Tag	Description
-SQL	Issues one or more SQL command to a compatible data source. Multiple commands are delimited by a semicolon. When multiple commands are used, all will be executed, however only the last command issued will return results to the [Inline] ... [/Inline] tags..
-Database	A database in the data source in which to execute the SQL statement.
-MaxRecords	The maximum number of records to return. Optional, defaults to 50.
-SkipRecords	The offset into the found set at which to start returning records. Optional, defaults to 1.

The -Database parameter can be any database within the data source in which the SQL statement should be executed. The -Database parameter will only be used to determine the data source, all table references within the statement must include both a database name and a table name, e.g. Contacts.People. For example, to create a new database in Lasso MySQL, a CREATE DATABASE statement can be executed with -Database set to Site.

When referencing the name of a database and table in a SQL statement (e.g. Contacts.People), only the true file names of a database or table can be used as MySQL does not recognize Lasso aliases in a SQL command. LDML 8 contains two SQL helper tags that return the true file name of a SQL database or table, as shown in *Table 13: SQL Helper Tags*.

Table 13: -SQL Helper Tags

Tag	Description
[Database_RealName]	Returns the actual name of a database from an alias. Useful for determining the true name of a database for use with the -SQL tag.
[Table_RealName]	This tag returns the actual name of a table from an alias. Useful for determining the true name of a table for use with the -SQL tag.

To determine the true database and table name for a SQL statement:

Use the [Database_RealName] and [Table_RealName] tags. When using the -SQL tag to issue SQL statements to a MySQL host, only true database and tables may be used (bypassing the alias). The [Database_RealName] and [Table_RealName] tags can be used to automatically determine the true name of a database and table, allowing them to be used in a valid SQL statement.

```
[Var_Set:'Real_DB' = (Database_RealName:'Contacts_Alias')]
[Var_Set:'Real_TB' = (Table_RealName:'Contacts_Alias')]
[Inline: -Database ='Contacts_Alias', -SQL='select * from ((Var:'Real_DB') + '.' +
(Var:'Real_TB'))']
```

Results from a SQL statement are returned in a record set within the [Inline] ... [/Inline] tags. The results can be read and displayed using the [Records] ... [/Records] container tags and the [Field] substitution tag. However, many SQL statements return a synthetic record set that does not correspond to the names of the fields of the table being operated upon. This is demonstrated in the examples that follow.

To issue a SQL statement:

Specify the SQL statement within [Inline] ... [/Inline] tags in a -SQL command tag.

- The following example calculates the results of a mathematical expression 1 + 2 and returns the value as a [Field] value named Result. Note that even though this SQL statement does not reference a database, a -Database tag is still required so Lasso knows to which data source to send the statement.

```
[Inline: -Database='Example', -SQL='SELECT 1+2 AS Result']
  <br>The result is: [Field: 'Result'].
[/Inline]
```

➜
The result is 3.

- The following example calculates the results of several mathematical expressions and returns them as field values One, Two, and Three.

```
[Inline: -Database='Example',
  -SQL='SELECT 1+2 AS One, sin(.5) AS Two, 5%2 AS Three']
  <br>The results are: [Field: 'One'], [Field: 'Two'], and [Field: 'Three'].
[/Inline]
```

➜
The results are 3, 0.579426, and 1.

- The following example calculates the results of several mathematical expressions using LDML and returns them as field values One, Two, and Three. It demonstrate how the results of LDML expressions and substitution tags can be used in a SQL statement.

```
[Inline: -Database='Example',
  -SQL='SELECT ' + (1+2) + ' AS One, ' + (Math_Sin: .5) +
    ' AS Two, ' + (Math_Mod: 5, 2) + ' AS Three']
  <br>The results are: [Field: 'One'], [Field: 'Two'], and [Field: 'Three'].
[/Inline]
```

➜
The results are 3, 0.579426, and 1.

- The following example returns records from the Phone_Book table where First_Name is equal to John. This is equivalent to a -Search using LDML.

```
[Inline: -Database='Example',
    -SQL='SELECT * FROM Phone_Book WHERE First_Name = "John"']
  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]
```

➜
John Doe

John Person

To issue a SQL statement with multiple commands:

Specify the SQL statements within [Inline] ... [/Inline] tags in a -SQL command tag, with each SQL command separated by a semi-colon. The following example adds three unique records to the Contacts database. Note that all single quotes within the SQL statement have been properly escaped using the \ character, as described at the beginning of this chapter.

```
[Inline: -Database='Contacts',
  -SQL='INSERT INTO Contacts.People (First_Name, Last_Name) VALUES
          (\'John\', \'Jakob\');
        INSERT INTO Contacts.People (First_Name, Last_Name) VALUES
          (\'Tom\', \'Smith\');
        INSERT INTO Contacts.People (First_Name, Last_Name) VALUES
          (\'Sally\', \'Brown\')']
[/Inline]
```

To automatically format the results of a SQL statement:

Use the [Field_Name] tag and [Loop] ... [/Loop] tags to create an HTML table that automatically formats the results of a -SQL command. The -MaxRecords tag should be set to All so all records are returned rather than the default (50).

The following example shows a REPAIR TABLE Contacts.People SQL statement being issued to a MySQL database, and the result is automatically formatted. The statement returns a synthetic record set which shows the results of the repair.

Notice that the database Contacts is specified explicitly within the SQL statement. Even though the database is identified in the -Database command tag within the [Inline] tag it still must be explicitly specified in each table reference within the SQL statement.

```
[Inline: -Database='Contacts',
    -SQL='REPAIR TABLE Contacts.People',
    -MaxRecords='All']
  <table border="1">
    <tr>
      [Loop: (Field_Name: -Count)]
        <td><b>[Field_Name: (Loop_Count)]</b></td>
      [/Loop]
    </tr>
    [Records]
      <tr>
        [Loop: (Field_Name: -Count)]
          <td>[Field: (Field_Name: Loop_Count)]</td>
        [/Loop]
      </tr>
    [/Records]
  </table>
[/Inline]
```

The results are returned in a table with bold column headings. The following results show that the table did not require any repairs. If repairs are performed then many records will be returned.

```
➜ TableOp Msg_Type      Msg_Text
  People Check          Status      OK
```

SQL Transactions

LDML 8 supports the ability to perform reversible SQL transactions provided that the data source used (e.g. MySQL 4.x) supports this functionality. See your data source documentation to see if transactions are supported.

FileMaker Note: SQL transactions are not supported for FileMaker Pro data sources.

SQL transactions can be achieved within nested [Inline] ... [/Inline] tags. A single connection to MySQL or JDBC data sources will be held open from the opening [Inline] tag to the closing [/Inline] tag. Any nested inlines that use the same data source will make use of the same connection.

Note: When using named inlines, the connection is not available in subsequent [Records: -InlineName='Name'] ... [/Records] tags.

To open a transaction and commit or rollback in MySQL:

Use nested -SQL inlines, where the outer inline performs a transaction, and the inner inline commits or rolls back the transaction depending on the results of a conditional statement.

```
[Inline: -Database='Contacts', -SQL='START TRANSACTION;
    INSERT INTO Contacts.People (Title, Company) VALUES (\'Mr.\', \'OmniPilot\');']
 [If: (Error_CurrentError) != (Error_NoError)]
  [Inline: -Database='Contacts', -SQL='ROLLBACK;']
  [/Inline]
 [Else]
  [Inline: -Database='Contacts', -SQL='COMMIT;']
  [/Inline]
 [/If]
[/Inline]
```

To fetch the last inserted ID in MySQL:

Used nested -SQL inlines, where the outer inline performs an insert query, and the inner inline retrieves the ID of the last inserted record using the MySQL last_insert_id() function. Because the two inlines share the same connection, the inner inline will always return the value added by the outer inline.

```
[Inline: -Database='Contacts',
        -SQL='INSERT INTO People (Title, Company) VALUES (\'Mr.\', \'OmniPilot\');']
 [Inline: -SQL='SELECT last_insert_id()']
  [Field: 'last_insert_id()']
 [/Inline]
[/Inline]
```

➜ 23

8

Chapter 8
Searching and Displaying Data

This chapter documents the LDML command tags which search for records and data within Lasso compatible databases and display the results.

- *Overview* provides an introduction to the database actions described in this chapter and presents important security considerations.
- *Searching Records* includes instructions for searching records within a database.
- *Displaying Data* describes the tags that can be used to display data that result from database searches.
- *Linking to Data* includes requirements and instructions for navigating through found sets and linking to particular records within a database.

Overview

LDML provides command tags for searching records within Lasso compatible databases. These command tags are used in conjunction with additional command tags and name/value parameters in order to perform the desired database action in a specific database and table or within a specific record.

The command tags documented in this chapter are listed in *Table 1: Command Tags*. The sections that follow describe the additional command tags and name/value parameters required for each database action.

Table 1: Command Tags

Tag	Description
-Search	Searches for records within a database.
-FindAll	Finds all records within a database.
-Random	Returns a random record from a database. Only works with FileMaker Pro databases.

How Searches are Performed

This section describes the steps that take place each time a search is performed using Lasso.

1. Lasso checks the database, table, and field name specified in the search to ensure that they are all valid.
2. Lasso security is checked to ensure that the current user has permission to perform a search in the desired database, table, and field. Filters are applied to the search criteria if they are defined within Lasso Administration.
3. The search query is formatted and sent to the database application. FileMaker Pro search queries are formatted as URLs and submitted to the Web Companion. Lasso MySQL search queries are formatted as SQL statements and submitted directly to Lasso MySQL.
4. The database application performs the desired search and assembles a found set. The database application is responsible for interpreting search criteria, wild cards in search strings, field operators, and logical operators.
5. The database application sorts the found set based on sort criteria included in the search query. The database application is responsible for determining the order of records returned to Lasso.
6. A subset of the found set is sent to Lasso as the result set. Only the number of records specified by -MaxRecords starting at the offset specified by -SkipRecords are returned to Lasso. If any -ReturnField command tags are included in a search then only those fields named by the -ReturnField command tags are returned to Lasso.
7. The result set can be displayed and manipulated using LDML tags that return information about the result set and LDML tags that return fields or other values.

Character Encoding

Lasso stores and retrieves data from data sources based on the preferences established in the *Setup > Data Sources* section of Lasso Administration. The following rules apply for each standard data source.

Lasso MySQL and MySQL – By default all communication is in the Latin-1 (ISO 8859-1) character set. This is to preserve backwards compatibility with prior versions of Lasso. The character set can be changed to the Unicode standard UTF-8 character set in the *Setup > Data Sources > Tables* section of Lasso Administration.

FileMaker Pro – By default all communication is in the MacRoman character set when Lasso Professional is hosted on Mac OS X or in the Latin-1 (ISO 8859-1) character set when Lasso Professional is hosted on Windows. The preference in the *Setup > Data Sources > Databases* section of Lasso Administration can be used to change the character set for cross-platform communications.

JDBC – All communication with JDBC data sources is in the Unicode standard UTF-8 character set.

See the Lasso Professional 8 Setup Guide for more information about how to change the character set settings in Lasso Administration.

Error Reporting

After a database action has been performed, Lasso reports any errors which occurred via the [Error_CurrentError] tag. The value of this tag should be checked to ensure that the database action was successfully performed.

To display the current error code and message:

The following code can be used to display the current error message. This code should be placed in a format file which is a response to a database action or within a pair of [Inline] ... [/Inline] tags.

```
[Error_CurrentError: -ErrorCode]: [Error_CurrentError]
```

If the database action was performed successfully then the following result will be returned.

```
➜ 0: No Error
```

To check for a specific error code and message:

The following example shows how to perform code to correct or report a specific error if one occurs. The following example uses a conditional [If] ... [/If] tag to check the current error message and see if it is equal to [Error_NoRecordsFound].

```
[If: (Error_CurrentError) == (Error_NoRecordsFound)]
  No records were found!
[/If]
```

Full documentation about error tags and error codes can be found in the *Error Control* chapter. A list of all Lasso error codes and messages can be found in *Appendix B: Error Codes*.

Classic Lasso

If Classic Lasso support has been disabled within Lasso Administration then database actions will not be performed automatically if they are specified within HTML forms or URLs. Although the database action will not be performed, the -Response tag will function normally. Use the following code in the response page to the HTML forms or URL to trigger the database action.

```
[Inline: (Action_Params)]
  [Error_CurrentError: -ErrorCode]: [Error_CurrentError]
[/Inline]
```

See the *Database Interaction Fundamentals* chapter in this guide and the *Setting Site Preferences* chapter in the Lasso Professional 8 Setup Guide for more information.

Note: The use of Classic Lasso has been deprecated. All solutions should be transitioned over to the [Inline] ... [/Inline] tag based methods described in this chapter.

Security

Lasso has a robust internal security system that can be used to restrict access to database actions or to allow only specific users to perform database actions. If a database action is attempted when the current visitor has insufficient permissions then they will be prompted for a username and password. An error will be returned if the visitor does not enter a valid username and password.

An [Inline] ... [/Inline] can be specified to execute with the permissions of a specific user by specifying -Username and -Password command tags within the [Inline] tag. This allows the database action to be performed even though the current site visitor does not necessarily have permissions to perform the database action. In essence, a valid username and password are embedded into the format file.

Table 2: Security Command Tags

Tag	Description
-Username	Specifies the username from Lasso Security which should be used to execute the database action.
-Password	Specifies the password which corresponds to the username.

To specify a username and password in an [Inline]:

The following example shows a -FindAll action performed within an [Inline] tag using the permissions granted for username SiteAdmin with password Secret.

```
[Inline: -FindAll,
  -Database='Contacts',
  -Table='People',
  -Username='SiteAdmin',
  -Password='Secret']

  [Error_CurrentError: -ErrorCode]: [Error_CurrentError]

[/Inline]
```

A specified username and password is only valid for the [Inline] ... [/Inline] tags in which it is specified. It is not valid within any nested [Inline] ... [/Inline] tags. See the *Setting Up Security* chapter of the Lasso Professional 8 Setup Guide for additional important information regarding embedding usernames and passwords into [Inline] tags.

Searching Records

Searches can be performed within any Lasso compatible database using the -Search command tag. The -Search command tag is specified within [Inline] ... [/Inline] tags. The -Search command tag requires that a number of additional command tags be defined in order to perform the search. The required command tags are detailed in *Table 3: -Search Action Requirements*.

Note: If Classic Lasso syntax is enabled then the -Search command tag can also be used within HTML forms or URLs. The use of Classic Lasso syntax has been deprecated so solutions which rely on it should be updated to use the inline methods described in this chapter.

Additional command tags are described in *Table 4: Operator Command Tags* and *Table 6: Results Command Tags* in the sections that follow.

Table 3: -Search Action Requirements

Tag	Description
-Search	The action which is to be performed. Required.
-Database	The database which should be searched. Required.
-Table	The table from the specified database which should be searched. Required.
-KeyField	The name of the field which holds the primary key for the specified table. Recommended.
-KeyValue	The particular value for the primary key of the record which should be returned. Using -KeyValue overrides all the other search parameters and returns the single record specified. Optional.
Name/Value Parameters	A variable number of name/value parameters specify the query which will be performed.

Any name/value parameters included in the search action will be used to define the query that is performed in the specified table. All name/value parameters must reference a field within the database. Any fields which are not referenced will be ignored for the purposes of the search.

To search a database using [Inline] ... [/Inline] tags:

The following example shows how to search a database by specifying the required command tags within an opening [Inline] tag. -Database is set to Contacts, -Table is set to People, and -KeyField is set to ID. The search returns records which contain John with the field First_Name.

The results of the search are displayed to the visitor inside the [Inline] ... [/Inline] tags. The tags inside the [Records] ... [/Records] tags will repeat for each record in the found set. The [Field] tags will display the value for the specified field from the current record being shown.

```
[Inline: -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    'First_Name'='John']

  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]

[/Inline]
```

If the search was successful then the following results will be returned.

```
➜ <br>John Person
  <br>John Doe
```

Additional name/value parameters and command tags can be used to generate more complex searches. These techniques are documented in the following section on *Operators*.

To search a database using visitor-defined values:

The following example shows how to search a database by specifying the required command tags within an opening [Inline] tag, but allow a site visitor to specify the search criteria in an HTML form.

The visitor is presented with an HTML form in the format file default.lasso. The HTML form contains two text inputs for First_Name and Last_Name and a submit button. The action of the form is the response page response.lasso which contains the [Inline] ... [/Inline] tags that will perform the search. The contents of the default.lasso file include the following.

```
<form action="response.lasso" method="POST">
  <br>First Name: <input type="text" name="First_Name" value="">
  <br>Last Name: <input type="text" name="Last_Name" value="">
  <br><input type="submit" name="-Nothing" value="Search Database">
</form>
```

The search is performed and the results of the search are displayed to the visitor inside the [Inline] ... [/Inline] tags in response.lasso. The values entered by the visitor in the HTML form in default.lasso are inserted into the [Inline] tag using the [Action_Param] tag. The tags inside the [Records] ... [/Records] tags will repeat for each record in the found set. The [Field] tags will display the value for the specified field from the current record being shown. The contents of the response.lasso file include the following.

```
[Inline: -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    'First_Name'=(Action_Param: 'First_Name'),
    'Last_Name'=(Action_Param: 'Last_Name')]

  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]

[/Inline]
```

If the visitor entered John for First_Name and Person for Last_Name then the following result would be returned.

```
➜ <br>John Person
```

Operators

LDML includes a set of command tags that allow operators to be used to create complex database queries. These command tags are summarized in *Table 4: Operator Command Tags*.

Table 4: Operator Command Tags

Tag	Description
-OperatorLogical	Specifies the logical operator for the search. Abbreviation is -OpLogical. Defaults to and.
-Operator	When specified before a name/value parameter, establishes the search operator for that name/value parameter. Abbreviation is -Op. Defaults to bw.
-OperatorBegin	Specifies the logical operator for all search parameters until -OperatorEnd is reached. Abbreviation is -OpBegin.
-OperatorEnd	Specifies the end of a logical operator grouping started with -OperatorBegin. Abbreviation is -OpEnd.

The operator command tags are divided into two categories.

- **Field Operators** are specified using the -Operator command tag before a name/value parameter. The field operator changes the way that the named field is searched for the value. If no field operator is specified then the default begins with bw operator is used. See *Table 5: Field Operators* for a list of the possible values for this tag.
- **Logical Operators** are specified using the -OperatorLogical, -OperatorBegin, and -OperatorEnd tags. These tags specify how the results of different name/value parameters are combined to form the full results of the search.

Field Operators

The possible values for the -Operator command tag are listed in *Table 5: Field Operators*. The default operator is begins with bw. Each operator can be used in its short form cn or in its long form Contains. Case is unimportant when specifying operators.

Field operators are interpreted differently depending on which database application is being accessed. For example, FileMaker Pro interprets bw to mean that any word within a field can begin with the value specified for that field. MySQL interprets bw to mean that the first word within the field must begin with the value specified. See the chapters on each data source or the documentation that came with a third-party data source connector for more information.

Several of the field operators are only supported in Lasso MySQL or other MySQL databases. These include the ft full text operator and the rx nrx regular expression operators.

Table 5: Field Operators

Operator	Description
bw	Begins With. Default if no operator is set.
cn	Contains.
ew	Ends With.
eq	Equals.
ft	Full Text. MySQL databases only.
gt	Greater Than.
gte	Greater Than or Equals.
lt	Less Than.
lte	Less Than or Equals.
neq	Not Equals.
nrx	Not RegExp. Opposite of RegExp. MySQL databases only.
rx	RegExp. Regular expression search. MySQL databases only.

Note: In previous versions of Lasso the field operators could be specified using either a short form, e.g. bw or a long form, e.g. Begins With. In Lasso Professional 8 only the short form is preferred. Use of the long form is deprecated. It is supported in this version, but may not work in future versions of Lasso Professional.

To specify a field operator in an [Inline] tag:

Specify the field operator before the name/value parameter which it will affect. The following [Inline] ... [/Inline] tags search for records where the First_Name begins with J and the Last_Name ends with son.

```
[Inline: -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -Operator='bw', 'First_Name'='J',
    -Operator='ew', 'Last_Name'='son']
```

```
    [Records]
      <br>[Field: 'First_Name'] [Field: 'Last_Name']
    [/Records]
  [/Inline]
```

The results of the search would include the following records.

```
➜ <br>John Person
  <br>Jane Person
```

Logical Operators

The logical operator command tag -OperatorLogical can be used with a value of either AND or OR. The command tags -OperatorBegin, and -OperatorEnd can be used with values of AND, OR, or NOT. -OperatorLogical applies to all search parameters specified with an action . -OperatorBegin applies to all search parameters until the matching -OperatorEnd tag is reached. The case of the value is unimportant when specifying a logical operator.

- **AND** specifies that records which are returned should fulfill all of the search parameters listed.
- **OR** specifies that records which are returned should fulfill one or more of the search parameters listed.
- **NOT** specifies that records which match the search criteria contained between the -OperatorBegin and -OperatorEnd tags should be omitted from the found set. NOT cannot be used with the -OperatorLogical tag.

  Note: In lieu of a NOT option for -OperatorLogical, many field operators can be negated individually by substituting the opposite field operator. The following pairs of field operators are the opposites of each other: eq and neq, lt and gte, gt and lte.

FileMaker Note: The -OperatorBegin and -OperatorEnd tags do not work with Lasso Connector for FileMaker Pro.

To perform a search using an AND operator:

Use the -OperatorLogical command tag with an AND value. The following [Inline] ... [/Inline] tags return records for which the First_Name field begins with John and the Last_Name field begins with Doe. The position of the -OperatorLogical command tag within the [Inline] tag is unimportant since it applies to the entire action.

```
[Inline: -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -OperatorLogical='AND',
```

```
    'First_Name'='John',
    'Last_Name'='Doe']
  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]
```

To perform a search using an OR operator:

Use the -OperatorLogical command tag with an OR value. The following [Inline] ... [/Inline] tags return records for which the First_Name field begins with either John or Jane. The position of the -OperatorLogical command tag within the [Inline] tag is unimportant since it applies to the entire action.

```
[Inline: -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -OperatorLogical='OR',
    'First_Name'='John',
    'First_Name'='Jane']
  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]
```

To perform a search using a NOT operator:

Use the -OperatorBegin and -OperatorEnd command tags with a NOT value. The following [Inline] ... [/Inline] tags return records for which the First_Name field begins with John and the Last_Name field is not Doe. The operators tags must surround the parameters of the search which are to be negated.

```
[Inline: -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    'First_Name'='John',
    -OperatorBegin='NOT',
      'Last_Name'='Doe',
    -OperatorEnd='NOT']
```

```
  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]
```

To perform a search with a complex query:

Use the -OperatorBegin and -OperatorEnd tags to build up a complex query. As an example, a query can be constructed to find records in a database whose First_Name and Last_Name both begin with the same letter J, or M. The desired query could be written in pseudo-code as follows.

```
( (First_Name begins with J) AND (Last_Name begins with J) ) OR
( (First_Name begins with M) AND (Last_Name begins with M) )
```

The pseudo code is translated into a URL as follows. Each line of the query becomes a pair of -OpBegin=AND and -OpEnd=AND tags with a name/value parameter for First_Name and Last_Name contained inside. The two lines are then combined using a pair of -OpBegin=OR and -OpEnd=OR tags. The nesting of the command tags works like the nesting of parentheses in the pseudo code above to clarify how Lasso should combine the results of different name/value parameters.

```
<a href="/response.lasso?-Search&
    -Database=Contacts&
    -Table=People&
    -KeyField=ID&
    -OpBegin=OR&
      -OpBegin=AND&
        First_Name=J&
        Last_Name=J&
      -OpEnd=AND&
      -OpBegin=AND&
        First_Name=M&
        Last_Name=M&
      -OpEnd=AND&
    -OpEnd=OR">
  First Name and Last Name both begin with J or M
</a>
```

The following results might be returned when this link is selected.

```
➜ <br>Johnny Johnson
  <br>Jimmy James
  <br>Mark McPerson
```

Results

LDML includes a set of command tags that allow the results of a search to be customized. These command tags do not change the found set of records that are returned from the search, but they do change the data that is returned to Lasso for formatting and display to the visitor. The results command tags are summarized in *Table 6: Results Command Tags*.

Table 6: Results Command Tags

Tag	Description
-Distinct	Specifies that only records with distinct values in all returned fields should be returned. MySQL databases only.
-MaxRecords	Specifies how many records should be shown from the found set. Optional, defaults to 50.
-SkipRecords	Specifies an offset into the found set at which records should start being shown. Optional, defaults to 1.
-ReturnField	Specifies a field that should be returned in the results of the search. Multiple -ReturnField tags can be used to return multiple fields. Optional, defaults to returning all fields in the searched table.
-SortField	Specifies that the results should be sorted based on the data in the named field. Multiple -SortField tags can be used for complex sorts. Optional, defaults to returning data in the order it appears in the database.
-SortOrder	When specified after a -SortField parameter, specifies the order of the sort, either ascending, descending or custom. Optional, defaults to ascending for each -SortField.
-SortRandom	Sorts the returned results randomly. MySQL databases only.
-UseLimit	Specifies that a MySQL LIMIT should be used instead of Lasso's built-in tools for limiting the found set. MySQL databases only.

The results command tags are divided into three categories.

- **Sorting** is specified using the -SortField and -SortOrder command tags. These tags change the order of the records which are returned by the search. The sort is performed by the database application before Lasso receives the record set.

  The -SortRandom tag can be used to perform a random sort on the found set from MySQL databases. Note that the sort will be random each time

a set of records is returned so -MaxRecords and -SkipRecords cannot be used to navigate a found set that is sorted randomly.

- The portion of the **Found Set** being shown is specified using the -MaxRecords and -SkipRecords tags. -MaxRecords sets the number of records which will be shown between the [Records] ... [/Records] tags that format the results for the visitor. The -SkipRecords tag sets the offset into the found set which is shown. These two tags define the window of records which are shown and can be used to navigate through a found set.

  The -UseLimit tag instructs MySQL data sources to use a SQL LIMIT tag to restrict the found set based on the values of the -MaxRecords and -SkipRecords tags. This may increase performance when many records are being found, but -MaxRecords is set to a low value.

- The **Fields** which are available are specified using the -ReturnField tag. Normally, all fields in the table that was searched are returned. If any -ReturnField tags are specified then only those fields will be available to be returned to the visitor using the [Field] tag. Specifying -ReturnField tags can improve the performance of Lasso by not sending unnecessary data between the database and the Web server.

  Note: In order to use the [KeyField_Value] tag within an inline the keyfield must be specified as one of the -ReturnField values.

- The -Distinct tag instructs MySQL data sources to return only records which contain distinct values across all returned fields. This tag is useful when combined with a single -ReturnField tag and a -FindAll to return all distinct values from a single field in the database.

To return sorted results:

Specify -SortField and -SortOrder command tags within the search parameters. The following inline includes sort command tags. The records are first sorted by Last_Name in ascending order, then sorted by First_Name in ascending order.

```
[Inline: -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    'First_Name'='J',
    -SortField='Last_Name', -SortOrder='Ascending',
    -SortField='First_Name', -SortOrder='Ascending']
```

```
  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]
```

The following results could be returned when this inline is run. The returned records are sorted in order of Last_Name. If the Last_Name of two records are equal then those records are sorted in order of First_Name.

```
➜ <br>Jane Doe
  <br>John Doe
  <br>Jane Person
  <br>John Person
```

To return a portion of a found set:

A portion of a found set can be returned by manipulating the values for -MaxRecords and -SkipRecords. In the following example, a search is performed for records where the First_Name begins with J. This search returns four records, but only the second two records are shown. -MaxRecords is set to 2 to show only two records and -SkipRecords is set to 2 to skip the first two records.

```
[Inline: -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    'First_Name'='J',
    -MaxRecords=2,
    -SkipRecords=2]
  [Records]
    <br>[Field: 'First_Name']
  [/Records]
[/Inline]
```

The following results could be returned when this inline is run. Neither of the Doe records from the previous example are shown since they are skipped over.

```
➜ <br>Jane Person
  <br>John Person
```

To limit the fields returned in search results:

Use the -ReturnField command tag. If a single -ReturnField command tag is used then only the fields that are specified will be returned. If no -ReturnField command tags are specified then all fields within the current table will be shown. In the following example, only the First_Name field is shown since it is the only field specified within a -ReturnField command tag.

```
[Inline: -Search,
     -Database='Contacts',
     -Table='People',
     -KeyField='ID',
     'First_Name'='J',
     -ReturnField='First_Name']
   [Records]
     <br>[Field: 'First_Name']
   [/Records]
[/Inline]
```

The following results could be returned when this link is selected. The Last_Name field cannot be shown for any of these records since it was not specified in a -ReturnField command tag.

```
➜ <br>Jane
  <br>John
  <br>Jane
  <br>John
```

If [Field: 'Last_Name'] were specified inside the [Inline] ... [/Inline] tags and not specified as a -ReturnField then an error would be returned rather than the indicated results.

Finding All Records

All records can be returned from a database using the -FindAll command tag. The -FindAll command tag functions exactly like the -Search command tag except that no name/value parameters or operator tags are required. Sort tags and tags which sort and limit the found set work the same as they do for -Search actions. -FindAll actions can be specified in [Inline] ... [/Inline] tags.

Note: If Classic Lasso syntax is enabled then the -FindAll command tag can also be used within HTML forms or URLs. The use of Classic Lasso syntax has been deprecated so solutions which rely on it should be updated to use the inline methods described in this chapter.

Table 7: -FindAll Action Requirements

Tag	Description
-FindAll	The action which is to be performed. Required.
-Database	The database which should be searched. Required.
-Table	The table from the specified database which should be searched. Required.
-KeyField	The name of the field which holds the primary key for the specified table. Recommended.

To find all records within a database:

The following [Inline] ... [/Inline] tags find all records within a database Contacts and displays them. The results are shown below.

```
[Inline: -FindAll,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID']
  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]

➜ <br>Jane Doe
  <br>John Person
  <br>Jane Person
  <br>John Doe
```

To return all unique field values:

The unique values from a field in a MySQL database can be returned using the -Distinct tag. Only records which have distinct values across all fields will be returned. In the following example, a -FindAll action is used on the People table of the Contacts database. Only distinct values from the Last_Name field are returned.

```
[Inline: -FindAll,
    -Database='Contacts',
    -Table='People',
    -Distinct,
    -SortField='First_Name',
    -ReturnField='First_Name']
```

```
    [Records]
      <br>[Field: 'First_Name']
    [/Records]
  [/Inline]
```

The following results are returned. Even though there are multiple instances of John and Jane in the database, only one record for each name is returned.

```
➜ <br>Jane
  <br>John
```

Finding Random Records

A random record can be returned from a database using the -Random command tag. The -Random command tag functions exactly like the -Search command tag except that no name/value parameters or operator tags are required. -Random actions can be specified in [Inline] ... [/Inline] tags.

Note: If Classic Lasso syntax is enabled then the -Random command tag can also be used within HTML forms or URLs. The use of Classic Lasso syntax has been deprecated so solutions which rely on it should be updated to use the inline methods described in this chapter.

Table 8: -Random Action Requirements

Tag	Description
-Random	The action which is to be performed. Required.
-Database	The database which should be searched. Required.
-Table	The table from the specified database which should be searched. Required.
-KeyField	The name of the field which holds the primary key for the specified table. Recommended.

To find a single random record from a database:

The following inline finds a single random record from a FileMaker Pro database Contacts.fp3 and displays it. -MaxRecords is set to 1 to ensure that only a single record is shown. One potential result is shown below. Each time this inline is run a different record will be returned.

```
[Inline: -Random,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -MaxRecords=1]
  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]

➜ <br>Jane Person
```

To return multiple records sorted in random order:

The -SortRandom tag can be used with the -Search or -FindAll actions to return many records from a MySQL database sorted in random order. In the following example, all records from the People table of the Contacts database are returned in random order.

```
[Inline: -FindAll,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -SortRandom]
  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]

➜ <br>John Doe
  <br>Jane Doe
  <br>Jane Person
  <br>John Person
```

Displaying Data

The examples in this chapter have all relied on the [Records] ... [/Records] tags and [Field] tag to display the results of the search that have been performed. This section describes the use of these tags in more detail.

Table 9: Field Display Tags

Tag	Description
[Records] … [/Records]	Loops through each record in a found set. Optional -InlineName parameter specifies that results should be returned from a named inline. Synonym is [Rows] … [/Rows].
[Field]	Returns the value for a database field. Requires one parameter, the field name. Optional parameter -RecordIndex specifies what record in the current found set a field should be shown from. Synonym is [Column].

The [Field] tag always returns the value for a field from the current record when it is used within [Records] ... [/Records] tags. If the [Field] tag is used outside of [Records] ... [/Records] tags then it returns the value for a field from the first record in the found set. If the found set is only one record then the [Records] ... [/Records] tags are optional.

FileMaker Note: Lasso Connector for FileMaker Pro includes a collection of FileMaker Pro specific tags which return database results. See the ***FileMaker Pro Data Sources*** chapter for more information.

To display the results from a search:

Use the [Records] ... [/Records] tags and [Field] tag to display the results of a search. The following [Inline] ... [/Inline] tags perform a -FindAll action in a database Contacts. The results are returned each formatted on a line by itself. The [Loop_Count] tag is used to indicate the order within the found set.

```
[Inline: -FindAll,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID']
  [Records]
    <br>[Loop_Count]: [Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]
```

```
➜ <br>1: Jane Doe
  <br>2: John Person
  <br>3: Jane Person
  <br>4: John Doe
```

To display the results for a single record:

Use [Field] tags within the contents of the [Inline] ... [/Inline] tags. The [Records] ... [/Records] tags are unnecessary if only a single record is returned. The following [Inline] ... [/Inline] tags perform a -Search for a single record whose primary key ID equals 1. The [KeyField_Value] is shown along with the [Field] values for the record.

```
[Inline: -Search,
     -Database='Contacts',
     -Table='People',
     -KeyField='ID',
     -KeyValue=1]
  <br>[KeyField_Value]: [Field: 'First_Name'] [Field: 'Last_Name']
[/Inline]

➜ <br>1: Jane Doe
```

To display the results from a named inline:

Use the -InlineName parameter in both the opening [Inline] tag and in the opening [Records] tag. The [Records] ... [/Records] tags can be located anywhere in the page after the [Inline] ... [/Inline] tags that define the database action. The following example shows a -FindAll action at the top of a page in a LassoScript with the results formatted later.

```
<?LassoScript
  Inline: -FindAll,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -InlineName='FindAll Results';
  /Inline;
?>

... Page Contents ...

[Records: -InlineName='FindAll Results']
  <br>[Loop_Count]: [Field: 'First_Name'] [Field: 'Last_Name']
[/Records]

➜ <br>1: Jane Doe
  <br>2: John Person
  <br>3: Jane Person
  <br>4: John Doe
```

To display the results from a search out of order:

The -RecordIndex parameter of the [Field] tag can be used to show results out of order. Instead of using [Records] ... [/Records] tags to loop through a found set, the following example uses [Loop] ... [/Loop] tags to loop down through

the found set from [MaxRecords_Value] to 1. The [Field] tags all reference the [Loop_Count] in their -RecordIndex parameter.

```
[Inline: -FindAll,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID"]
  [Loop: -LoopFrom=(MaxRecords_Value), -LoopTo=1, -LoopIncrement=-1]
    <br>[Loop_Count]: [Field: 'First_Name', -RecordIndex=(Loop_Count)]
      [Field: 'Last_Name', -RecordIndex=(Loop_Count)]
  [/Loop]
[/Inline]

➜ <br>4: John Doe
  <br>3: Jane Person
  <br>2: John Person
  <br>1: Jane Doe
```

Linking to Data

This section describes how to create links which allow a visitor to manipulate the found set. The following types of links can be created.

- **Navigation** – Links can be created which allow a visitor to page through a found set. Only a portion of the found set needs to be shown, but the entire found set can be accessed.
- **Detail** – Links can be created which allow detail about a particular record to be shown in another format file.
- **Sorting** – Links can be provided to re-sort the current found set on a different field.

Note: If Classic Lasso syntax is enabled then the links tags can be used to trigger actions using command tags embedded in URLs. The use of Classic Lasso syntax has been deprecated so solutions which rely on it should be updated to use the inline methods described in this chapter.

Most of the link techniques implicitly assume that the records within the database are not going to change while the visitor is navigating through the found set. The database search is actually performed again for every page served to a visitor and if the number of results change then the records being shown to the visitor can be shifted or altered as soon as another link is selected.

Link Tags

LDML 8 includes many tags which make creating detail links and navigation links easy within Lasso solutions. The general purpose link tags are specified in *Table 10: Link Tags*. The common parameters for all link tags are specified in *Table 11: Link Tag Parameters*.

The remainder of the chapter lists and demonstrates the link URL, container, and parameter tags. Tags which generate URLs for links automatically are listed in *Table 12: Link URL Tags*. Container tags which generate entire HTML anchor tags <a> automatically are listed in *Table 13: Link Container Tags*. Tags which provide parameter arrays for each link option are listed in *Table 14: Link Parameter Tags*.

Table 10: Link Tags

Tag	Description
[Link] ... [/Link]	General purpose link tag that provides an anchor tag with the specified parameters. The -Response parameter is used as the URL for the link.
[Link_Params]	General purpose link tag that processes a set of parameters using the common rules for all link tags.
[Link_SetFormat]	Sets a standard set of options that will be used for all link tags that follow in the current format file.
[Link_URL]	General purpose link tag that provides a URL based on the specified parameters. The -Response parameter is used as the URL for the link.

Each of the general purpose link tags implement the basic behavior of all the link tags, but are not usually used on their own. The section on *Link Tag Parameters* below describes the common parameters that all link tags interpret. The following sections include the link URL, container, and parameter tags and examples of their use.

Note: The [Link_...] tags do not include values for the -SQL, -Username, -Password or the -ReturnField tags in the links they generate.

Link Tag Parameters

All of the link tags accept the same parameters which allow the link that is being formed to be customized. These parameters include all the command tags which can be passed to the opening [Inline] tag and a series of parameters detailed in *Table 11: Link Tag Parameters* which allow various command tags to be removed from the generated link tags.

The link tags interpret their parameters as follows.

- The parameters are processed in the order they are specified within the link tag. Later parameters override earlier parameters.
- Most link tags process [Action_Params] first, then any parameters specified in [Link_SetFormat], and finally the parameters specified within the link tag itself. The general purpose link tags do not include [Action_Params] automatically.
- Parameters of type array are inserted into the parameters as if each item of the array was specified in order at the location of the array.
- Many command tags will only be included once in the resulting link. These include -Database, -Table, -KeyField, -MaxRecords, and any other command tags that can only be specified once within an inline. The last value for the command tag will be included in the resulting link.
- Only one action such as -Search, -FindAll, or -Nothing will be included in the resulting link. The last action specified in the link tag will be used.
- Command tags such as -Required, -Op, -OpBegin, -OpEnd, -SortField, -SortOrder, and -Token will be included in the order they are specified within the tag.
- The resulting link will consist of the action followed by all command tags specified once in alphabetical order, and finally all name/value parameters and command tags that are specified multiple times in the same order they were specified in the parameters.
- All -No... parameters are interpreted at the location they occur in the parameters. If a -NoDatabase parameter is specified early in the parameter list and a -Database command tag is included later then the -Database command tag will be included in the resulting link.
- The -NoClassic parameter removes all command tags that are not essential to specifying the search and location in the found set to an [Inline] tag. The -Database, -Table, -KeyField, and action are all removed. All name/value parameters, -Sort... tags, -Op tags, and either -MaxRecords and -SkipRecords or -KeyValue are included.
- The value of the -Response command tag will be used as the URL for the resulting link. The link tags always link to a response file on the same server they are called. If not specified the -Response will be the same as [Response_FilePath].
- The -SQL, -Username, -Password, and -ReturnField tags are never returned by the link tags.

Note: The [Referrer] and [Referrer_URL] tags are special cases which simply return the referrer specified in the HTTP request header. They do not accept any parameters.

Table 11: Link Tag Parameters

Tag	Description
Command Tag	Inserts the specified command tag. Either appends the command tag or overrides an existing command tag with the new value.
Name/Value Pair	Inserts the specified name/value pair.
Array Parameter	An array of pairs is inserted as if each name/value pair in the array was specified in the tag parameters at the location of the array.
-NoAction	Removes the action command tag.
-NoClassic	Removes all parameters required to specify an action in Classic Lasso leaving only those parameters required to specify the query and current location in the found set.
-NoDatabase	Removes the -Database command tag.
-NoTable	Removes the -Table or -Layout command tag. -NoLayout is a synonym.
-NoKeyField	Removes the -KeyField command tag.
-NoKeyValue	Removes the -KeyValue command tag.
-NoOperatorLogical	Removes the -OperatorLogical command tag.
-NoResponse	Removes the -Response command tag.
-NoMaxRecords	Removes the -MaxRecords command tag.
-NoSkipRecords	Removes the -SkipRecords command tag.
-NoParams	Removes name/value pairs, -Operator, -OperatorBegin, -OperatorEnd, and -Required tags.
-NoSort	Removes all -Sort… command tags.
-NoToken, -NoToken.Name	Removes the -Token command tag. With a parameter as -NoToken.Name removes the specified token command tag.
-NoTokens	Removes all -Token… command tags.
-NoSchema	Removes the -Schema command tag for JDBC data sources.
-No.Name	Removes a specified name/value parameter.
-Response	Specifies the file that will be used as the URL for the link tag. The link tags always link to a file on the current server.

Link URL Tags

The tags listed in *Table 12: Link URL Tags* each return a URL based on the current database action. Each of these tags accepts the same parameters as specified in *Table 11: Link Tag Parameters* above and corresponds to matching container and parameter tags. Examples of the link tags are included in the *Link Examples* section that follows.

Table 12: Link URL Tags

Tag	Description
[Link_CurrentActionURL]	Returns a link to the current Lasso action.
[Link_FirstGroupURL]	Returns a link to the first group of records based on the current Lasso action. Sets -SkipRecords to 0.
[Link_PrevGroupURL]	Returns a link to the next group of records based on the current Lasso action. Changes -SkipRecords.
[Link_NextGroupURL]	Returns a link to the next group of records based on the current Lasso action. Changes -SkipRecords.
[Link_LastGroupURL]	Returns a link to the last group of records based on the current Lasso action. Changes -SkipRecords.
[Link_CurrentRecordURL]	Returns a link to the current record. Sets -MaxRecords to 1 and changes -SkipRecords.
[Link_FirstRecordURL]	Returns a link to the first record based on the current Lasso action. Sets -MaxRecords to 1 and -SkipRecords to 0.
[Link_PrevRecordURL]	Returns a link to the next record based on the current Lasso action. Sets -MaxRecords to 1 and changes -SkipRecords.
[Link_NextRecordURL]	Returns a link to the next record based on the current Lasso action. Sets -MaxRecords to 1 and changes -SkipRecords.
[Link_LastRecordURL]	Returns a link to the last record based on the current Lasso action. Sets -MaxRecords to 1 and changes -SkipRecords.
[Link_DetailURL]	Returns a link to the current record using the primary key and key value. Changes -KeyValue.
[Referrer_URL]	Returns a link to the previous page which the visitor was at before the current page. [Referer_URL] is a synonym.

Note: The [Referrer_URL] tag is a special case which simply returns the referrer specified in the HTTP request header. It does not accept any parameters.

Link Container Tags

The tags listed in *Table 13: Link Container Tags* each return an anchor tag based on the current database action. The anchor tags surround the contents of the container tag. If the link tag is not valid then no result is returned. Each of these tags accepts the same parameters as specified in *Table 11: Link Tag Parameters* above and corresponds to matching URL and parameter tags. Examples of the link tags are included in the *Link Examples* section that follows.

Table 13: Link Container Tags

Tag	Description
[Link_CurrentAction]	Returns a link to the current Lasso action.
[Link_FirstGroup]	Returns a link to the first group of records based on the current Lasso action. Sets -SkipRecords to 0.
[Link_PrevGroup]	Returns a link to the previous group of records based on the current Lasso action. Changes -SkipRecords.
[Link_NextGroup]	Returns a link to the next group of records based on the current Lasso action. Changes -SkipRecords.
[Link_LastGroup]	Returns a link to the last group of records based on the current Lasso action. Changes -SkipRecords.
[Link_CurrentRecord]	Returns a link to the current record. Sets -MaxRecords to 1 and changes -SkipRecords.
[Link_FirstRecord]	Returns a link to the first record based on the current Lasso action. Sets -MaxRecords to 1 and -SkipRecords to 0.
[Link_PrevRecord]	Returns a link to the previous record based on the current Lasso action. Sets -MaxRecords to 1 and changes -SkipRecords.
[Link_NextRecord]	Returns a link to the next record based on the current Lasso action. Sets -MaxRecords to 1 and changes -SkipRecords.
[Link_LastRecord]	Returns a link to the last record based on the current Lasso action. Sets -MaxRecords to 1 and changes -SkipRecords.
[Link_Detail]	Returns a link to the current record using the -KeyField and -KeyValue. Changes -KeyValue.
[Referrer]	Returns a link to the previous page which the visitor was at before the current page. [Referer] is a synonym.

Note: The [Referrer] ... [/Referrer] tag is a special case which simply returns the referrer specified in the HTTP request header. It does not accept any parameters.

Link Parameter Tags

The tags listed in *Table 14: Link Parameter Tags* each return an array of parameters based on the current database action. Each of these tags accepts the same parameters as specified in *Table 11: Link Tag Parameters* above and corresponds to matching container and URL tags. Examples of the link tags are included in the *Link Examples* section that follows.

Table 14: Link Parameter Tags

Tag	Description
[Link_CurrentActionParams]	Returns a link to the current Lasso action.
[Link_FirstGroupParams]	Returns a link to the first group of records based on the current Lasso action. Sets -SkipRecords to 0.
[Link_PrevGroupParams]	Returns a link to the previous group of records based on the current Lasso action. Changes -SkipRecords.
[Link_NextGroupParams]	Returns a link to the next group of records based on the current Lasso action. Changes -SkipRecords.
[Link_LastGroupParams]	Returns a link to the last group of records based on the current Lasso action. Changes -SkipRecords.
[Link_CurrentRecordParams]	Returns a link to the current record. Sets -MaxRecords to 1 and changes -SkipRecords.
[Link_FirstRecordParams]	Returns a link to the first record based on the current Lasso action. Sets -MaxRecords to 1 and -SkipRecords to 0.
[Link_PrevRecordParams]	Returns a link to the previous record based on the current Lasso action. Sets -MaxRecords to 1 and changes -SkipRecords.
[Link_NextRecordParams]	Returns a link to the next record based on the current Lasso action. Sets -MaxRecords to 1 and changes -SkipRecords.
[Link_LastRecordParams]	Returns a link to the last record based on the current Lasso action. Sets -MaxRecords to 1 and changes -SkipRecords.
[Link_DetailParams]	Returns a link to the current record using the primary key and key value. Changes -KeyValue.

Note: There is no link parameter tag equivalent to the referrer tags.

Link Examples

The basic technique for using the link tags is the same as that which was described to allow site visitors to enter values into HTML forms and then use those values within an [Inline] ... [/Inline] action. The [Inline] tags can have some command tags and search parameters specified explicitly, with variables, an array, [Action_Params], or one of the link tags defining the rest.

For example, an [Inline] ... [/Inline] could be specified to find all records within a database as follows. The entire action is specified within the opening [Inline] tag. Each time a page with the code on it is visited the action will be performed as written.

```
[Inline: -FindAll,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -MaxRecords=10]
  …
[/Inline]
```

The same inline can be modified so that it can accept parameters from an HTML form or URL which is used to load the page it is on, but can still act as a standalone action. This is accomplished by adding an [Action_Params] tag to the opening [Inline] tag.

```
[Inline: (Action_Params),
    -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -MaxRecords=4]
  …
[/Inline]
```

Any command tags or name/value pairs in the HTML form or URL that triggers the page with this inline will be passed into the inline through the [Action_Params] tag as if they had been typed directly into the [Inline]. However, the command tags specified directly in the [Inline] tag will override any corresponding tags from the [Action_Params].

Since the action -Search is specified after the [Action_Params] array it will override any other action from the array. The action of this inline will always be -Search. Similarly, all of the -Database, -Table, -KeyField, or -MaxRecords tags will have the values specified in the [Inline] overriding any values passed in through [Action_Params].

The various link tags can be used to generate URLs which work with the specified inline in order to change the set of records being shown, the sort order and sort field, etc. The link tags are able to override any command

tags not specified in the opening [Inline] tag, but the basic action is always performed exactly as specified.

Navigation Links

Navigation links are created by manipulating the value for -SkipRecords so that the visitor is shown a different portion of the found set each time they follow a link or by setting -KeyValue to an appropriate value to show one record in a database.

To create next and previous links:

The [Link_NextGroup] ... [/Link_NextGroup] and [Link_PrevGroup] ... [/Link_PrevGroup] tags can be used with the inline specified above to page through a set of found records.

The [Link_SetFormat] tag is used to include a -NoClassic parameter in each link tag that follows. This ensures that the -Database, -Table, and -KeyField are not included in the links generated by the link tags.

The full inline is shown below. It uses the [Records] ... [/Records] tags to show the people that have been found in the database and includes next and previous links to page through the found set.

```
[Inline: (Action_Params),
    -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -MaxRecords=4]

  <p>[Found_Count] records were found, showing [Shown_Count]
  records from [Shown_First] to [Shown_Last].

  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]

  [Link_SetFormat: -NoClassic]
  [Link_PrevGroup] <br>Previous [MaxRecords_Value] Records [/Link_PrevGroup]
  [Link_NextGroup] <br>Next [MaxRecords_Value] Records [/Link_NextGroup]
[/Inline]
```

The first time this page is loaded the first four records from the database are shown. Since this is the first group of records in the database only the Next 4 Records link is displayed.

```
➜ <p>16 records were found, showing 4 records from 1 to 4.
  <br>Jane Doe
  <br>John Person
  <br>Jane Person
  <br>John Doe
  <br>Next 4 Records
```

If the Next 4 Records link is selected then the same page is reloaded. The value for -SkipRecords is taken from the link tag and passed into the opening [Inline] tag through the [Action_Params] array. The following results are displayed. This time both the Next 4 Records and the Previous 4 Records links are displayed.

```
➜ <p>16 records were found, showing 4 records from 5 to 8.
  <br>Jane Surname
  <br>John Last_Name
  <br>Mark Last_Name
  <br>Tom Surname
  <br>Previous 4 Records
  <br>Next 4 Records
```

To create first and last links:

Links to the first and last groups of records in the found set can be added using the [Link_FirstGroup] ... [/Link_FirstGroup] and [Link_LastGroup] ... [/Link_LastGroup] tags. The following inline includes both next/previous links and first/last links.

```
[Inline: (Action_Params),
    -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -MaxRecords=4]

  <p>[Found_Count] records were found, showing [Shown_Count]
  records from [Shown_First] to [Shown_Last].

  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]

  [Link_SetFormat: -NoClassic]
  [Link_FirstGroup] <br>First [MaxRecords_Value] Records [/Link_FirstGroup]
  [Link_PrevGroup] <br>Previous [MaxRecords_Value] Records [/Link_PrevGroup]
  [Link_NextGroup] <br>Next [MaxRecords_Value] Records [/Link_NextGroup]
  [Link_LastGroup] <br>Last [MaxRecords_Value] Records [/Link_LastGroup]
[/Inline]
```

The first time this page is loaded the first four records from the database are shown. Since this is the first group of records in the database only the Next 4 Records and Last 4 Records links are displayed. The Previous 4 Records and First 4 Records links will automatically appear if either of these links are selected by the visitor.

```
➜ <p>16 records were found, showing 4 records from 1 to 4.
  <br>Jane Doe
  <br>John Person
  <br>Jane Person
  <br>John Doe
  <br>Next 4 Records
  <br>Last 4 Records
```

To create links to page through the found set:

Many Web sites include page links which allow the visitor to jump directly to any set of records within the found set. The example -FindAll returns 16 records from Contacts so four page links would be created to jump to the 1st, 5th, 9th, and 13th records.

A set of page links can be created using the [Link_CurrentActionURL] tag as a base and then customizing the -SkipRecords value as needed. The following loop creates as many page links as are needed for the current found set.

```
[Inline: (Action_Params),
    -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -MaxRecords=4]

  <p>[Found_Count] records were found, showing [Shown_Count]
  records from [Shown_First] to [Shown_Last].

  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]

  [Link_SetFormat: -NoClassic]
  [Variable: 'Count' = 0]
  [While: $Count < (Found_Count)]
    <br><a href="[Link_CurrentActionURL: -SkipRecords=$Count]">
      Page [Loop_Count]
    </a>
    [Variable: 'Count' = $Count + (MaxRecords_Value)]
  [/While]

[/Inline]
```

The results of this code for the example -Search would be the following. There are four page links. The first is equivalent to the First 4 Records link created above and the last is equivalent to the Last 4 Records link created above.

```
➜ <p>16 records were found, showing 4 records from 1 to 4.
  <br>Jane Doe
  <br>John Person
  <br>Jane Person
  <br>John Doe
  <br>Page 1
  <br>Page 2
  <br>Page 3
  <br>Page 4
```

Sorting Links

Sorting links are created by adding or manipulating -SortField and -SortOrder command tags. The same found set is shown, but the order is determined by which link is selected. Often, the column headers in a table of results from a database will represent the sort links that allow the table to be resorted by the values in that specific column.

To create links that sort the found set:

The following code performs a -Search in an inline and formats the results as a table. The column heading at the top of each table column is a link which re-sorts the results by the field values in that column. The links for sorting the found set are created by specifying -NoSort and -SortField parameters to the [Link_FirstGroup] ... [/Link_FirstGroup] tags.

```
[Inline: (Action_Params),
    -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -MaxRecords=4]

  [Link_SetFormat: -NoClassic]
  <table>
    <tr>
      <th>
        [Link_FirstGroup: -NoSort, -SortOrder='First_Name']
          First Name
        [/Link_FirstGroup]
      </th>
      <th>
        [Link_FirstGroup: -NoSort, -SortOrder='Last_Name']
```

```
        Last Name
      [/Link_FirstGroup]
    </th>
  </tr>

  [Records]
    <tr>
      <td>[Field: 'First_Name']</td>
      <td>[Field: 'Last_Name']</td>
    </tr>
  [/Records]

  </table>
[/Inline]
```

Detail Links

Detail links are created in order to show data from a particular record in the database table. Usually, a listing format file will contain only limited data from each record in the found set and a detail format file will contain significantly more information about a particular record.

A link to a particular record can be created using the [Link_Detail] ... [/Link_Detail] tags to set the -KeyField and -KeyValue fields. This method is guaranteed to return the selected record even if the database is changing while the visitor is navigating. However, it is difficult to create next and previous links on the detail page. This option is most suitable if the selected database record will need to be updated or deleted.

Alternately, a link to a particular record can be created using [Link_CurrentAction] ... [/Link_CurrentAction] and setting -MaxRecords to 1. This method allows the visitor to continue navigating by records on the detail page.

To create a link to a particular record:

There are two format files involved in most detail links. The listing format file default.lasso includes the [Inline] ... [/Inline] tags that define the search for the found set. The detail format file response.lasso includes the [Inline] ... [/Inline] tags that find and display the individual record.

1 The [Inline] tag in default.lasso simply performs a -FindAll action. Each record in the result set is displayed with a link to response.lasso created using the [Link_Detail] ... [/Link_Detail] tags.

```
[Inline:-FindAll,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
```

```
    -MaxRecords=4]
  [Link_SetFormat: -NoClassic]
  [Records]
    <br>[Link_Detail: -Response='response.lasso']
      [Field: 'First_Name'] [Field: 'Last_Name']
    [/Link_Detail]
  [/Records]
[/Inline]

➜ <br>Jane Doe
  <br>John Person
  <br>Jane Person
  <br>John Doe
```

2 The [Inline] tag on response.lasso uses [Action_Params] to pull the values from the URL generated by the link tags. The results contain more information about the particular records than is shown in the listing. In this case, the Phone_Number field is included as well as the First_Name and Last_Name.

```
[Inline:(Action_Params),
    -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID']
  <br>[Field: 'First_Name'] [Field: 'Last_Name']
  <br>[Field: 'Phone_Number']
  …
[/Inline]

➜ <br>Jane Doe
  <br>555-1212
```

To create a link to the current record in the found set:

There are two format files involved in most detail links. The listing format file default.lasso includes the [Inline] ... [/Inline] tags that define the search for the found set. The detail format file response.lasso includes the [Inline] ... [/Inline] tags that find and display the individual record. The [Link_CurrentAction] ... [/Link_CurrentAction] tags are used to create a link from default.lasso to response.lasso showing a particular record.

1 The [Inline] tag on default.lasso simply performs a -FindAll action. Each record in the result set is displayed with a link to response.lasso created using the [Link_CurrentAction] ... [/Link_CurrentAction] tag.

```
[Inline:-FindAll,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -MaxRecords=4]
```

```
  [Link_SetFormat: -NoClassic]
  [Records]
    <br>[Link_CurrentAction: -Response='response.lasso', -MaxRecords=1]
      [Field: 'First_Name'] [Field: 'Last_Name']
    [/Link_CurrentAction]
  [/Records]
[/Inline]

➜ <br>Jane Doe
  <br>John Person
  <br>Jane Person
  <br>John Doe
```

2 The [Inline] tag in response.lasso uses [Action_Params] to pull the values from the URL generated by the link tags. The results contain more information about the particular records than is shown in the listing. In this case, the Phone_Number field is included as well as the First_Name and Last_Name.

The detail page can also contain links to the previous and next records in the found set. These are created using the [Link_PrevRecord] ... [/Link_PrevRecord] and [Link_NextRecord] ... [/Link_NextRecord] tags. The visitor can continue navigating the found set record by record.

```
[Inline:(Action_Params),
    -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID']
  <br>[Field: 'First_Name'] [Field: 'Last_Name']
  <br>[Field: 'Phone_Number']
  …
  [Link_SetFormat: -NoClassic]
  <br>[Link_PrevRecord] Previous Record [/Link_PrevRecord]
  <br>[Link_NextRecord] Next Record [/Link_NextRecord]
[/Inline]

➜ <br>Jane Last_Name
  <br>555-1212
  <br>Previous Record
  <br>Next Record
```

9

Chapter 9
Adding and Updating Records

This chapter documents the LDML command tags which add, update, delete, and duplicate records within Lasso compatible databases.

- *Overview* provides an introduction to the database actions described in this chapter and presents important security considerations.
- *Adding Records* includes requirements and instructions for adding records to a database.
- *Updating Records* includes requirements and instructions for updating records within a database.
- *Deleting Records* includes requirements and instructions for deleting records within a database.
- *Duplicating Records* includes requirements and instructions for duplicating records within a database.

Overview

LDML provides command tags for adding, updating, deleting, and duplicating records within Lasso compatible databases. These command tags are used in conjunction with additional command tags and name/value parameters in order to perform the desired database action in a specific database and table or within a specific record.

The command tags documented in this chapter are listed in *Table 1: Command Tags*. The sections that follow describe the additional command tags and name/value parameters required for each database action.

Table 1: Command Tags

Tag	Description
-Add	Adds a record to a database.
-Update	Updates a record within a database.
-Delete	Removes a record from a database.
-Duplicate	Duplicates a record within a database. Works with FileMaker Pro databases.

Character Encoding

Lasso stores and retrieves data from data sources based on the preferences established in the *Setup > Data Sources* section of Lasso Administration. The following rules apply for each standard data source.

Lasso MySQL and MySQL – By default all communication is in the Latin-1 (ISO 8859-1) character set. This is to preserve backwards compatibility with prior versions of Lasso. The character set can be changed to the Unicode standard UTF-8 character set in the *Setup > Data Sources > Tables* section of Lasso Administration.

FileMaker Pro – By default all communication is in the MacRoman character set when Lasso Professional is hosted on Mac OS X or in the Latin-1 (ISO 8859-1) character set when Lasso Professional is hosted on Windows. The preference in the *Setup > Data Sources > Databases* section of Lasso Administration can be used to change the character set for cross-platform communications.

JDBC – All communication with JDBC data sources is in the Unicode standard UTF-8 character set.

See the Lasso Professional 8 Setup Guide for more information about how to change the character set settings in Lasso Administration.

Error Reporting

After a database action has been performed, Lasso reports any errors which occurred via the [Error_CurrentError] tag. The value of this tag should be checked to ensure that the database action was successfully performed.

To display the current error code and message:

The following code can be used to display the current error message. This code should be placed in a format file which is a response to a database action or within a pair of [Inline] ... [/Inline] tags.

```
[Error_CurrentError: -ErrorCode]: [Error_CurrentError]
```

If the database action was performed successfully then the following result will be returned.

```
0: No Error
```

To check for a specific error code and message:

The following example shows how to perform code to correct or report a specific error if one occurs. The following example uses a conditional [If] ... [/If] tag to check the current error message and see if it is equal to [Error_AddError].

```
[If: (Error_CurrentError) == (Error_AddError)]
  An Add Error has occurred!
[/If]
```

Full documentation about error tags and error codes can be found in the *Error Control* chapter. A list of all Lasso error codes and messages can be found in *Appendix B: Error Codes*.

Classic Lasso

If Classic Lasso support has been disabled within Lasso Administration then database actions will not be performed automatically if they are specified within HTML forms or URLs. Although the database action will not be performed, the -Response tag will function normally. Use the following code in the response page to the HTML forms or URL to trigger the database action.

```
[Inline: (Action_Params)]
  [Error_CurrentError: -ErrorCode]: [Error_CurrentError]
[/Inline]
```

See the *Database Interaction Fundamentals* chapter and the *Setting Site Preferences* chapter of the Lasso Professional 8 Setup Guide for more information.

Security

Lasso has a robust internal security system that can be used to restrict access to database actions or to allow only specific users to perform database actions. If a database action is attempted when the current visitor has insufficient permissions then they will be prompted for a username and password. An error will be returned if the visitor does not enter a valid username and password.

An [Inline] ... [/Inline] can be specified to execute with the permissions of a specific user by specifying -Username and -Password command tags within the [Inline] tag. This allows the database action to be performed even though

the current site visitor does not necessarily have permissions to perform the database action. In essence, a valid username and password are embedded into the format file.

Table 2: Security Command Tags

Tag	Description
-Username	Specifies the username from Lasso Security which should be used to execute the database action.
-Password	Specifies the password which corresponds to the username.

To specify a username and password in an [Inline]:

The following example shows a -Delete action performed within an [Inline] tag using the permissions granted for username SiteAdmin with password Secret.

```
[Inline: -Delete,
  -Database='Contacts',
  -Table='People',
  -KeyField='ID',
  -KeyValue=137,
  -Username='SiteAdmin',
  -Password='Secret']

  [Error_CurrentError: -ErrorCode]: [Error_CurrentError]

[/Inline]
```

A specified username and password is only valid for the [Inline] ... [/Inline] tags in which it is specified. It is not valid within any nested [Inline] ... [/Inline] tags. See the *Setting Up Security* chapter of the Lasso Professional 8 Setup Guide for additional important information regarding embedding usernames and passwords into [Inline] tags.

Adding Records

Records can be added to any Lasso compatible database using the -Add command tag. The -Add command tag requires that a number of additional command tags be defined in order to perform the -Add action. The required command tags are detailed in the following table.

Table 3: -Add Action Requirements

Tag	Description
-Add	The action which is to be performed. Required.
-Database	The database in which the record should be added. Required.
-Table	The table from the specified database in which the record should be added. Required.
-KeyField	The name of the field which holds the primary key for the specified table. Recommended.
Name/Value Parameters	A variable number of name/value parameters specify the initial field values for the added record. Optional.

Any name/value parameters included in the -Add action will be used to set the starting values for the record which is added to the database. All name/value parameters must reference a writable field within the database. Any fields which are not referenced will be set to their default values according to the database's configuration.

Lasso returns a reference to the record which was added to the database. The reference is different depending on what type of database to which the record was added.

- **Lasso MySQL and MySQL** – The -KeyField tag should be set to the primary key field or auto-increment field of the table. Lasso will return the added record as the result of the action by checking the specified key field for the last inserted record. The [KeyField_Value] tag can be used to inspect the value of the auto-increment field for the inserted record.

  If no -KeyField is specified, the specified -KeyField is not an auto-increment field, or -MaxRecords is set to 0 then no record will be returned as a result of the -Add action. This can be useful in situations where a large record is being added to the database and there is no need to inspect the values which were added.

- **FileMaker Pro** – The [KeyField_Value] tag is set to the value of the internal Record ID for the new record. The Record ID functions as an auto-increment field that is automatically maintained by FileMaker Pro for all records.

  FileMaker Pro automatically performs a search for the record which was added to the database. The found set resulting from an -Add action is equivalent to a search for the single record using the [KeyField_Value].

  The value for -KeyField is ignored when adding records to a FileMaker Pro database. The value for [KeyField_Value] is always the internal Record ID value.

Note: Consult the documentation for third-party data sources to see what behavior they implement when adding records to the database.

To add a record using [Inline] … [/Inline] tags:

The following example shows how to perform an -Add action by specifying the required command tags within an opening [Inline] tag. -Database is set to Contacts, -Table is set to People, and -KeyField is set to ID. Feedback that the -Add action was successful is provided to the visitor inside the [Inline] ... [/Inline] tags using the [Error_CurrentError] tag. The added record will only include default values as defined within the database itself.

```
[Inline: -Add,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID']

  <p>[Error_CurrentError: -ErrorCode]: [Error_CurrentError]

[/Inline]
```

If the -Add action is successful then the following will be returned.

```
➜ 0: No Error
```

To add a record with data using [Inline] … [/Inline] tags:

The following example shows how to perform an -Add action by specifying the required command tags within an opening [Inline] tag. In addition, the [Inline] tag includes a series of name/value parameters that define the values for various fields within the record that is to be added. The First_Name field is set to John and the Last_Name field is set to Doe. The added record will include these values as well as any default values defined in the database itself.

```
[Inline: -Add,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    'First_Name'='John',
    'Last_Name'='Doe']

  <p>[Error_CurrentError: -ErrorCode]: [Error_CurrentError]
  <br>Record [Field: 'ID'] was added for [Field: 'First_Name'] [Field: 'Last_Name'].

[/Inline]
```

The results of the -Add action contain the values for the record that was just added to the database.

```
➜ 0: No Error
  Record 2 was added for John Doe.
```

To add a record using an HTML form:

The following example shows how to perform an -Add action using an HTML form to send values into an [Inline] tag through [Action_Params]. The text inputs provide a way for the site visitor to define the initial values for various fields in the record which will be added to the database. The site visitor can set values for the fields First_Name and Last_Name.

```
<form action="response.lasso" method="POST">
  <br>First Name: <input type="text" name="First_Name" value="">
  <br>Last Name: <input type="text" name="Last_Name" value="">
  <br><input type="submit" name="-Nothing" value="Add Record">
</form>
```

The response page for the form, response.lasso, contains the following code that performs the action using an [Inline] tag and provides feedback that the record was successfully added to the database. The field values for the record that was just added to the database are automatically available within the [Inline] ... [/Inline] tags.

```
[Inline: (Action_Params),
    -Add,
    -Database='Contacts',
    -Table='People',
    -Keyfield='ID']
  <p>[Error_CurrentError: -ErrorCode]: [Error_CurrentError]
  <br>Record [Field: 'ID'] was added for [Field: 'First_Name'] [Field: 'Last_Name'].
[/Inline]
```

If the form is submitted with Mary in the First Name input and Person in the Last Name input then the following will be returned.

```
➜ 0: No Error
  Record 3 was added for Mary Person
```

To add a record using a URL:

The following example shows how to perform an -Add action using a URL to send values into an [Inline] tag through [Action_Params]. The name/value parameters in the URL define the starting values for various fields in the database: First_Name is set to John and Last_Name is set to Person.

```
<a href="response.lasso?First_Name=John&Last_Name=Person">
  Add John Person
</a>
```

The response page for the URL, response.lasso, contains the following code that performs the action using [Inline] tag and provides feedback that the record was successfully added to the database. The field values for the

record that was just added to the database are automatically available within the [Inline] ... [/Inline] tags.

```
[Inline: (Action_Params),
    -Add,
    -Database='Contacts',
    -Table='People',
    -Keyfield='ID']
  <p>[Error_CurrentError: -ErrorCode]: [Error_CurrentError]
  <br>Record [Field: 'ID'] was added for [Field: 'First_Name'] [Field: 'Last_Name'].
[/Inline]
```

If the link for Add John Person is selected then the following will be returned.

```
➜ 0: No Error
  Record 4 was added for John Person.
```

Updating Records

Records can be updated within any Lasso compatible database using the -Update command tag. The -Update command tag requires that a number of additional command tags be defined in order to perform the -Update action. The required command tags are detailed in the following table.

Table 4: -Update Action Requirements

Tag	Description
-Update	The action which is to be performed. Required.
-Database	The database in which the record should be added. Required.
-Table	The table from the specified database in which the record should be added. Required.
-KeyField	The name of the field which holds the primary key for the specified table. Required.
-KeyValue	The value of the primary key of the record which is to be updated. Required.
Name/Value Parameters	A variable number of name/value parameters specifying the field values which need to be updated. Optional.

Lasso identifies the record which is to be updated using the values for the command tags -KeyField and -KeyValue. -KeyField must be set to a field in the table which has a unique value for every record in the table. Usually, this is the primary key field for the table. -KeyValue must be set to a valid value

for the -KeyField in the table. If no record can be found with the specified -KeyValue then an error will be returned.

Any name/value parameters included in the update action will be used to set the field values for the record which is updated. All name/value parameters must reference a writable field within the database. Any fields which are not referenced will maintain the values they had before the update.

Lasso returns a reference to the record which was updated within the database. The reference is different depending on what type of database is being used.

- **Lasso MySQL and MySQL** – The [KeyField_Value] tag is set to the value of the key field which was used to identify the record to be updated. The -KeyField should always be set to the primary key or auto-increment field of the table. The results when using other fields are undefined.

  If the -KeyField is not set to the primary key field or auto-increment field of the table or if -MaxRecords is set to 0 then no record will be returned as a result of the -Update action. This is useful if a large record is being updated and the results of the update do not need to be inspected.

- **FileMaker Pro** – The [KeyField_Value] tag is set to the value of the internal Record ID for the updated record. The Record ID functions as an auto-increment field that is automatically maintained by FileMaker Pro for all records.

Lasso automatically performs a search for the record which was updated within the database. The found set resulting from an -Update action is equivalent to a search for the single record using the [KeyField_Value].

Note: Consult the documentation for third-party data sources to see what behavior they implement when updating records within a database.

To update a record with data using [Inline] … [/Inline] tags:

The following example shows how to perform an -Update action by specifying the required command tags within an opening [Inline] tag. The record with the value 2 in field ID is updated. The [Inline] tag includes a series of name/value parameters that define the new values for various fields within the record that is to be updated. The First_Name field is set to Bob and the Last_Name field is set to Surname. The updated record will include these new values, but any fields which were not included in the action will be left with the values they had before the update.

```
[Inline: -Update,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -KeyValue=2,
    'First_Name'='Bob',
    'Last_Name'='Surname']

  <p>[Error_CurrentError: -ErrorCode]: [Error_CurrentError]
  <br>Record [Field: 'ID'] was added for [Field: 'First_Name'] [Field: 'Last_Name'].

[/Inline]
```

The updated field values from the -Update action are automatically available within the [Inline].

```
➜ 0: No Error
  Record 2 was updated to Bob Surname.
```

To update a record using an HTML form:

The following example shows how to perform an -Update action using an HTML form to send values into an [Inline] tag. The text inputs provide a way for the site visitor to define the new values for various fields in the record which will be updated in the database. The site visitor can see and update the current values for the fields First_Name and Last_Name.

```
[Inline: -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -KeyValue=3]

<form action="response.lasso" method="POST">
  <input type="hidden" name="-KeyValue" value="[KeyField_Value]">

  <br>First Name: <input type="text" name="First_Name"
    value="[Field: 'First_Name']">
  <br>Last Name: <input type="text" name="Last_Name"
    value="[Field: 'Last_Name']">
  <br><input type="submit" name="-Update" value="Update Record">
</form>

[/Inline]
```

The response page for the form, response.lasso, contains the following code that performs the action using an [Inline] tag and provides feedback that the record was successfully updated in the database. The field values from the updated record are available automatically within the [Inline] ... [/Inline] tags.

```
[Inline: (Action_Params),
    -Update,
    -Database='Contacts',
    -Table='People',
    -Keyfield='ID']
  <p>[Error_CurrentError: -ErrorCode]: [Error_CurrentError]
  <br>Record [Field: 'ID'] was updated to [Field: 'First_Name'] [Field: 'Last_Name'].
[/Inline]
```

The form initially shows Mary for the First Name input and Person for the Last Name input. If the form is submitted with the Last Name changed to Peoples then the following will be returned. The First Name field is unchanged since it was left set to Mary.

```
➜ 0: No Error
  Record 3 was updated to Mary Peoples.
```

To update a record using a URL:

The following example shows how to perform an -Update action using a URL to send field values to an [Inline] tag. The name/value parameters in the URL define the new values for various fields in the database: First_Name is set to John and Last_Name is set to Person.

```
<a href="response.lasso?-KeyValue=4&
  First_Name=John&Last_Name=Person"> Update John Person </a>
```

The response page for the URL, response.lasso, contains the following code that performs the action using [Inline] ... [/Inline] tags and provides feedback that the record was successfully updated within the database.

```
[Inline: (Action_Params),
    -Update,
    -Database='Contacts',
    -Table='People',
    -Keyfield='ID']
  <p>[Error_CurrentError: -ErrorCode]: [Error_CurrentError]
  <br>Record [Field: 'ID'] was updated to [Field: 'First_Name'] [Field: 'Last_Name'].
[/Inline]
```

If the link for Update John Person is submitted then the following will be returned.

```
➜ 0: No Error
  Record 4 was updated for John Person.
```

To update several records at once:

The following example shows how to perform an -Update action on several records at once within a single database table. The goal is to update every record in the database with the last name of Person to the new last name of Peoples.

The outer [Inline] ... [/Inline] tags perform a search for all records in the database with Last_Name equal to Person. This forms the found set of records which need to be updated. The [Records] ... [/Records] tags repeat once for each record in the found set. The -MaxRecords='All' command tag ensures that all records which match the criteria are returned.

The inner [Inline] ... [/Inline] tags perform an update on each record in the found set. Substitution tags are used to retrieve the values for the required command tags -Database, -Table, -KeyField, and -KeyValue. This ensures that these values match those from the outer [Inline] ... [/Inline] tags exactly. The name/value pair 'Last_Name'='Peoples' updates the field to the new value.

```
[Inline: -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -MaxRecords='All',
    'Last_Name'='Person']
  [Records]

    [Inline: -Update,
        -Database=(Database_Name),
        -Table=(Table_Name),
        -KeyField=(KeyField_Name),
        -KeyValue=(KeyField_Value),
        'Last_Name'='Peoples']

      <p>[Error_CurrentError: -ErrorCode]: [Error_CurrentError]
      <br>Record [Field: 'ID'] was updated to
        [Field: 'First_Name'] [Field: 'Last_Name'].

    [/Inline]

  [/Records]
[/Inline]
```

This particular search only finds one record to update. If the update action is successful then the following will be returned for each updated record.

```
➜ 0: No Error
  Record 4 was updated to John Peoples.
```

Deleting Records

Records can be deleted from any Lasso compatible database using the -Delete command tag. The -Delete command tag can be specified within an [Inline] tag, an HTML form, or a URL. The -Delete command tag requires that a number of additional command tags be defined in order to perform the -Delete action. The required command tags are detailed in the following table.

Table 5: -Delete Action Requirements

Tag	Description
-Delete	The action which is to be performed. Required.
-Database	The database in which the record should be added. Required.
-Table	The table from the specified database in which the record should be added. Required.
-KeyField	The name of the field which holds the primary key for the specified table. Required.
-KeyValue	The value of the primary key of the record which is to be deleted. Required.

Lasso identifies the record which is to be deleted using the values for the command tags -KeyField and -KeyValue. -KeyField must be set to a field in the table which has a unique value for every record in the table. Usually, this is the primary key field for the table. -KeyValue must be set to a valid value for the -KeyField in the table. If no record can be found with the specified -KeyValue then an error will be returned.

Lasso returns an empty found set in response to a -Delete action. Since the record has been deleted from the database the [Field] tag can no longer be used to retrieve any values from it. The [Error_CurrentError] tag should be checked to ensure that it has a value of No Error in order to confirm that the record has been successfully deleted.

There is no confirmation or undo of a delete action. When a record is removed from a database it is removed permanently. It is important to set up Lasso security appropriately so accidental or unauthorized deletes don't occur. See the *Setting Up Security* chapter in the Lasso Professional 8 Setup Guide for more information about setting up database security.

To delete a record with data using [Inline] … [/Inline] tags:

The following example shows how to perform a delete action by specifying the required command tags within an opening [Inline] tag. The record with the value 2 in field ID is deleted.

```
[Inline: -Delete,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -KeyValue=2]

  <p>[Error_CurrentError: -ErrorCode]: [Error_CurrentError]

[/Inline]
```

If the delete action is successful then the following will be returned.

```
➜ 0: No Error
```

To delete several records at once:

The following example shows how to perform a -Delete action on several records at once within a single database table. The goal is to delete every record in the database with the last name of Peoples.

Warning: This technique can be used to remove all records from a database table. It should be used with extreme caution and tested thoroughly before being added to a public Web site.

The outer [Inline] ... [/Inline] tags perform a search for all records in the database with Last_Name equal to Peoples. This forms the found set of records which need to be updated. The [Records] ... [/Records] tags repeat once for each record in the found set. The -MaxRecords='All' command tag ensures that all records which match the criteria are returned.

The inner [Inline] ... [/Inline] tags delete each record in the found set. Substitution tags are used to retrieve the values for the required command tags -Database, -Table, -KeyField, and -KeyValue. This ensures that these values match those from the outer [Inline] ... [/Inline] tags exactly.

```
[Inline: -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -MaxRecords='All',
    'Last_Name'='Peoples']
  [Records]

    [Inline: -Delete,
        -Database=(Database_Name),
        -Table=(Table_Name),
        -KeyField=(KeyField_Name),
        -KeyValue=(KeyField_Value)]

      <p>[Error_CurrentError: -ErrorCode]: [Error_CurrentError]

    [/Inline]
```

```
    [/Records]
[/Inline]
```

This particular search only finds one record to delete. If the delete action is successful then the following will be returned for each deleted record.

```
➜ 0: No Error
```

Duplicating Records

Records can be duplicated within any Lasso compatible database using the -Duplicate command tag. The -Duplicate command tag can be specified within an [Inline] tag, an HTML form, or a URL. The -Duplicate command tag requires that a number of additional command tags be defined in order to perform the -Duplicate action. The required command tags are detailed in the following table.

Note: Lasso Connector for Lasso MySQL and Lasso Connector for MySQL do not support the -Duplicate command tag.

Table 6: -Duplicate Action Requirements

Tag	Description
-Duplicate	The action which is to be performed. Required.
-Database	The database in which the record should be added. Required.
-Table	The table from the specified database in which the record should be added. Required.
-KeyField	The name of the field which holds the primary key for the specified table. Required.
-KeyValue	The value of the primary key of the record which is to be duplicated. Required.
Name/Value Parameters	A variable number of name/value parameters specifying field values which should be modified in the duplicated record. Optional.

Lasso identifies the record which is to be duplicated using the values for the command tags -KeyField and -KeyValue. -KeyField must be set to a field in the table which has a unique value for every record in the table. Usually, this is the primary key field for the table. -KeyValue must be set to a valid value for the -KeyField in the table. If no record can be found with the specified -KeyValue then an error will be returned.

Any name/value parameters included in the duplicate action will be used to set the field values for the record which is added to the database. All name/value parameters must reference a writable field within the database. Any fields which are not referenced will maintain the values they had from the record which was duplicated.

Lasso always returns a reference to the new record which was added to the database as a result of the -Duplicate action. This is equivalent to performing a -Search action which returns a single record found set containing just the record which was added to the database.

To duplicate a record with data using [Inline] … [/Inline] tags:

The following example shows how to perform a duplicate action within a FileMaker Pro database by specifying the required command tags within an opening [Inline] tag. The record with the value 2 for the keyfield value is duplicated. The [Inline] tag includes a series of name/value parameters that define the new values for various fields within the record that is to be updated. The First_Name field is set to Joe and the Last_Name field is set to Surname. The new record will include these values, but any fields which were not specified in the action will be left with the values they had from the source record.

```
[Inline: -Duplicate,
    -Database='Contacts.fp3',
    -Table='People',
    -KeyField='ID',
    -KeyValue=2,
    'First_Name'='Joe',
    'Last_Name'='Surname']

  <p>[Error_CurrentError: -ErrorCode]: [Error_CurrentError]
  <br>Record [Field: 'ID'] was duplicated for [Field: 'First_Name'] [Field: 'Last_Name'].

[/Inline]
```

If the duplicate action is successful then the following will be returned. The values from the [Field] tags are retrieved from the record which was just added to the database as a result of the duplicate action.

```
➜ 0: No Error
  Record 6 was duplicated for Joe Surname.
```

10

Chapter 10
MySQL Data Sources

This chapter documents tags and behaviors which are specific to MySQL data sources.

- *Overview* introduces MySQL data sources.
- *MySQL Tags* describes tags specific to MySQL data sources.
- *Searching Records* describes unique search operations that can be performed using MySQL data sources.
- *Adding and Updating Records* describes unique add and update operations that can be performed using MySQL data sources.
- *Value Lists* describes how to retrieve and show lists of allowed field values for ENUM and SET fields in MySQL data sources.
- *Creating Database Tables* describes the [Database_...] tags that can be used to create, change, or remove tables and fields within MySQL data sources.

Overview

Lasso Professional 8 allows for a connection to a remote MySQL data source to be established. This chapter primarily documents tags and features unique to MySQL data sources.

Note: Although the tags and procedures defined in this chapter are primarily for use with MySQL data sources. Many of the tags and procedures will work with any SQL-based data source with minor variations, if any.

Tips for Using MySQL Data Sources

- Always specify a primary key field using the -KeyField command tag in -Search, -Add, and -Findall actions. This will ensure that the [KeyField_Value] tag will always return a value.
- Use -KeyField and -KeyValue to reference a particular record for updates, duplicates, or deletes.
- MySQL data sources are case-sensitive. For best results, reference MySQL database and table names in the same letter-case as they appear on disk in your LDML code.
- MySQL fields truncate any data beyond the length they are set up to store. Ensure that all fields in MySQL databases have sufficiently long fields for the values that need to be stored in them.
- Use -ReturnField command tags to reduce the number of fields which are returned from a -Search action. Returning only the fields that need to be used for further processing or shown to the site visitor reduces the amount of data that needs to travel between Lasso Service and MySQL.
- When an -Add or -Update action is performed on a MySQL database, the data from the added or updated record is returned inside the [Inline] ... [/Inline] tags or alternately to the Classic Lasso response page. If the -ReturnField parameter is used, then only those fields specified should be returned from an -Add or -Update action. Setting -MaxRecords=0 can be used as an indication that no record should be returned.
- See the Site *Administration Utilities* chapter in the Lasso Professional 8 Setup Guide for information about optimizing tables for optimum performance and checking tables for damage.

Security Tips

- The -SQL command tag can only be allowed or disallowed at the host level for users in Lasso Administration. Once the -SQL command tag is allowed for a user, that user may access any database within the allowed host inside of a SQL statement. For that reason, only trusted users should be allowed to issue SQL queries using the -SQL command tag. For more information, see the *Setting Up Security* chapter in the Lasso Professional 8 Setup Guide.
- SQL statements which are generated using visitor-defined data should be screened carefully for unwanted commands such as DROP or GRANT. See the *Setting Up Data Sources* chapter of the Lasso Professional 8 Setup Guide for more information.
- Always quote any inputs from site visitors that are incorporated into SQL statements. For example, the following SQL SELECT statement includes quotes around the [Action_Param] value. The quotes are escaped \' so they

will be embedded within the string rather than ending the string literal. The semi-colon at the end of the statement is optional unless multiple statements are issued.

```
[Variable: 'SQL_Statement'='SELECT * FROM Contacts.People WHERE ' +
  'First_Name LIKE \'' + (Action_Param: 'First_Name') + '\';']
```

If [Action_Param] returns John for First_Name then the SQL statement generated by this code would appear as follows.

```
SELECT * FROM Contacts.People WHERE First_Name LIKE 'John';
```

MySQL Tags

LDML 8 includes tags to identify which type of MySQL data source is being used. These tags are summarized in *Table 1: Enhanced MySQL Tags*.

Table 1: Enhanced MySQL Tags

Tag	Description
[Lasso_DatasourceIsMySQL]	Returns True if a database is hosted by MySQL. Requires one string value, which is the name of a database.

To check whether a database is hosted by MySQL:

The following example shows how to use [Lasso_DatasourceIsMySQL] to check whether the database Example is hosted by MySQL or not.

```
[If: (Lasso_DatasourceIsMySQL: 'Example')]
  Example is hosted by MySQL!
[Else]
  Example is not hosted by MySQL.
[/If]
```

```
➜ Example is hosted by MySQL!
```

To list all databases hosted by MySQL:

Use the [Database_Names] ... [/Database_Names] tags to list all databases available to Lasso. The [Lasso_DatasourceIsMySQL] tag can be used to check each database and only those that are hosted by MySQL will be returned. The result shows two databases, Site and Example, which are available through MySQL.

```
[Database_Names]
  [If: (Lasso_DatasourceIsMySQL:(Database_NameItem))]
    <br>[Database_NameItem]
  [/If]
[/Database_Names]

➜ <br>Example
  <br>Site
```

Searching Records

In LDML 8, there are unique search operations that can be performed using MySQL data sources. These search operations take advantage of special functions in MySQL such as full-text indexing, regular expressions, record limits, and distinct values to allow optimal performance and power when searching. These search operations can be used on MySQL data sources in addition to all search operations described in the *Searching and Displaying Data* chapter.

Search Field Operators

Additional field operators are available for the -Operator (or -Op) tag when searching MySQL data sources. These operators are summarized in *Table 2: MySQL Search Field Operators*. Basic use of the -Operator tag is described in the *Searching and Displaying Data* chapter.

Table 2: MySQL Search Field Operators

Operator	Description
ft	Full-Text Search. If used, a MySQL full-text search is performed on the field specified. Will only work on fields that are full-text indexed in MySQL. Records are automatically returned in order of high relevance (contains many instances of that value) to low relevance (contains few instances of the value). Only one ft operator may be used per action, and no -SortField parameter should be specified.
rx	Regular Expression. If used, then regular expressions may be used as part of the search field value. Returns records matching the regular expression value for that field.

nrx	Not Regular Expression. If used, then regular expressions may be used as part of the search field value. Returns records that do not match the regular expression value for that field.

Note: For more information on full-text searches and regular expressions supported in MySQL, see the MySQL documentation.

To perform a full-text search on a field:

If a MySQL field is indexed as full-text, then using -Op='ft' before the field in a search inline performs a MySQL full text search on that field. The example below performs a full text search on the Jobs field in the Contacts database, and returns the First_Name field for each record that contain the word Manager. Records that contain the most instances of the word Manager are returned first.

```
[Inline: -Search, -Database='Contacts', -Table='People',
-Op='ft',
'Jobs'='Manager']
  [Records]
    [Field:'First_Name']<br>
  [/Records]
[/Inline]

➜ Mike<br>
Jane<br>
```

To use regular expressions as part of a search:

Regular expressions can be used as part of a search value for a field by using -Op='rx' before the field in a search inline. The following example searches for all records where the Last_Name field contains eight characters using a regular expression.

```
[Inline: -Search, -Database='Contacts', -Table='People',
-Op='rx',
'Last_Name'='.{8}',
-MaxRecords='All']
  [Records]
    [Field:'Last_Name'], [Field:'First_Name']<br>
  [/Records]
[/Inline]

➜ Lastname, Mike<br>
Lastname, Mary Beth<br>
```

The following example searches for all records where the Last_Name field doesn't contain eight characters. This is easily accomplished using the same inline search above using -Op='nrx' instead.

```
[Inline: -Search, -Database='Contacts', -Table='People',
-Op='nrx',
'Last_Name'='.{8}',
-MaxRecords='All']
  [Records]
    [Field:'Last_Name'], [Field:'First_Name']<br>
  [/Records]
[/Inline]
```

```
➜ Doe, John<br>
Doe, Jane<br>
Surname, Bob<br>
Surname, Jane<br>
Surname, Margaret<br>
Unknown, Thomas<br>
```

Search Command Tags

Additional search command tags are available when searching MySQL data sources using the [Inline] tag. These tags allow special search functions specific to MySQL to be performed without writing SQL statements. These operators are summarized in *Table 3: MySQL Search Command Tags*.

Table 3: MySQL Search Command Tags

Tag	Description
-UseLimit	Prematurely ends a -Search or FindAll action once the specified number of records for the -MaxRecords tag have been found and returns the found records. Requires the -MaxRecords tag. This issues an internal LIMIT statement to MySQL to cause it to search more efficiently.
-SortRandom	Sorts returned records randomly. Is used in place of the -SortField and -SortOrder parameters. Does not require a value.
-Distinct	Causes a -Search action to only output records that contain unique field values (comparing only returned fields). Does not require a value. May be used with the -ReturnField parameter to limit the fields checked for distinct values.

-GropupBy	Specifies a field name which should by used as the GROUP BY statement. Allows data to be summarized based on the values of the specified field.

To have MySQL immediately return records once a limit is reached:

Use the -UseLimit tag in the search inline. Normally, Lasso will find all records that match the inline search criteria and then pair down the results based on -MaxRecords and -SkipRecords values. The -UseLimit tag instructs MySQL to terminate the specified search process once the number of records specified for -MaxRecords is found. The following example searches the Contacts database with a limit of five records.

```
[Inline: -FindAll,
-Database='Contacts', -Table='People',
-MaxRecords='5',
-UseLimit]
[Found_Count]
[/Inline]

➜ 5
```

Note: If the -UseLimit tag is used, the value of the [Found_Count] tag will always be the same as the -MaxRecords value if the limit is reached. Otherwise, the [Found_Count] tag will return the total number of records in the specified table that match the search criteria if -UseLimit is not used.

To sort results randomly:

Use the -SortRandom tag in a search inline. The following example finds all records and sorts first by last name then randomly.

```
[Inline: -FindAll, -Database='Contacts', -Table='People',
-Keyfield='ID',
-SortRandom]
  [Records]
    [Field:'ID']
  [/Records]
[/Inline]

➜ 5 2 8 1 3 6 4 7
```

Note: Due to the nature of the -SortRandom tag, the results of this example will vary upon each execution of the inline.

To return only unique records in a search:

Use the -Distinct parameter in a search inline. The following example only returns records that contain distinct values for the Last_Name field.

```
[Inline: -FindAll, -Database='Contacts', -Table='People',
-ReturnField='Last_Name',
-Distinct]
  [Records]
    [Field:'Last_Name']<br>
  [/Records]
[/Inline]
```

```
➜ Doe<br>
Surname<br>
Lastname<br>
Unknown<br>
```

The -Distinct tag is especially useful for generating lists of values that can be used in a pull-down menu. The following example is a pull-down menu of all the last names in the Contacts database.

```
[Inline: -Findall, -Database='Contacts', -Table='People',
-ReturnField='Last_Name',
-Distinct]
  <select name="Last_Name">
    [Records]
      <option value="[Field: 'Last_Name']">
        [Field: 'Last_Name']
      </Option>
    [/Records]
  </Select>
[/Inline]
```

Searching Null Values

When searching MySQL tables, NULL values may be explicitly searched for within fields using the [Null] tag. A NULL value in MySQL designates that there is no other value stored in that particular field. This is similar to searching a field for an empty string (e.g. 'fieldname=''), however NULL values and empty strings are not the same in MySQL. For more information about NULL values, see the MySQL documentation.

```
[Inline: -Search,
-Database='Contacts', -Table='People',
-Op='eq',
'Title'=(Null),
-MaxRecords='All']
```

```
    [Records]
      Record [Field:'ID'] does not have a title.<br>
    [/Records]
  [/Inline]
```

```
➜ Record 7 does not have a title.<br>
  Record 8 does not have a title.<br>
```

Adding and Updating Records

In LDML 8, there are special add and update operations that can be performed using MySQL data sources in addition to all add and update operations described in the *Adding and Updating Data* chapter.

Multiple Field Values

When adding or updating data to a field in MySQL, the same field name can be used several times in an -Add or -Update inline. The result is that all data added or updated in each instance of the field name will be concatenated in a comma-delimited form. This is particularly useful for adding data to SET field types.

To add or update multiple values to a field:

The following example adds a record with two comma delimited values in the Jobs field:

```
[Inline: -Add, -Database='Contacts', -Table='People',
-KeyField='ID',
'Jobs'='Customer Service',
'Jobs'='Sales']
[Field:'Title']
[/Inline]
```

```
➜ Customer Service, Sales
```

The following example updates the Jobs field of a record with three comma-delimited values:

```
[Inline: -Update, -Database='Contacts', -Table='People',
-KeyField='ID',
-KeyValue='5',
'Jobs'='Customer Service',
'Jobs'='Sales',
'Jobs'='Support']
[Field:'Title']
[/Inline]
```

```
➜ Customer Service, Sales, Support
```

Note: The individual values being added or updated should not contain commas.

Adding or Updating Null Values

NULL values can be explicitly added to MySQL fields using the [Null] tag. A NULL value in MySQL designates that there is no value for a particular field. This is similar to setting a field to an empty string (e.g. 'fieldname'='), however the two are different in MySQL. For more information about NULL values, see the MySQL documentation.

To add or update a null value to a field:

Use the [Null] tag as the field value. The following example adds a record with a NULL value in the Last_Name field.

```
[Inline: -Add, -Database='Contacts', -Table='People',
-KeyField='ID',
'Last_Name'=(Null)]
[/Inline]
```

The following example updates a record with a NULL value in the Last_Name field.

```
[Inline: -Update, -Database='Contacts', -Table='People',
-KeyField='ID',
-KeyValue='5',
'Last_Name'=(Null)]
[/Inline]
```

Value Lists

A value list in Lasso is a set of possible values that can be used for a field. Value lists in MySQL are lists of pre-defined and stored values for a SET or ENUM field type. A value list from a SET or ENUM field can be displayed using the tags defined in *Table 4: MySQL Value List Tags*.

Table 4: MySQL Value List Tags

Tag	Description
[Value_List] ... [/Value_List]	Container tag repeats each value allowed for ENUM or SET fields. Requires a single parameter: the name of an ENUM or SET field from the current table.
[Value_ListItem]	Returns the value for the current item in a value list. Optional -Checked or -Selected parameter returns only values currently contained in the ENUM or SET field.
[Selected]	Displays the word Selected if the current value list item is contained in the data of the ENUM or SET field.
[Checked]	Displays the word Checked if the current value list item is contained in the data of the ENUM or SET field.
[Option]	Generates a series of <option> tags for the value list. Requires a single parameter: the name of an ENUM or SET field from the current table.

Note: See the *Searching and Displaying Data* chapter for information about the -Show command tag which is used throughout this section.

To display values for an ENUM or SET field:

- Perform a -Show action to return the schema of a MySQL database and use the [Value_List] tag to display the allowed values for an ENUM or SET field. The following example shows how to display all values from the ENUM field Title in the Contacts database. SET field value lists function in the same manner as ENUM value lists, and all examples in this section may be used with either ENUM or SET field types.

```
[Inline:  -Show, -Database='Contacts', -Table='People']
  [Value_List: 'Title']
    <br>[Value_ListItem]
  [/Value_List]
[/Inline]

➜ <br>Mr.
<br>Mrs.
<br>Ms.
<br>Dr.
```

- The following example shows how to display all values from a value list using a named inline. The same name Values is referenced by -InlineName in both the [Inline] tag and [Value_List] tag.

```
[Inline: -InlineName='Values',  -Show, -Database='Contacts', -Table='People']
[/Inline]
…
[Value_List: 'Title', -InlineName='Values']
  <br>[Value_ListItem]
[/Value_List]
```

```
➜ <br>Mr.
<br>Mrs.
<br>Ms.
<br>Dr.
```

To display an HTML pop-up menu in an -Add form with all values from a value list:

- The following example shows how to format an HTML <select> ... </select> pop-up menu to show all the values from a value list. A select list can be created with the same code by including size and/or multiple parameters within the <select> tag. This code is usually used within an HTML form that performs an -Add action so the visitor can select a value from the value list for the record they create.

  The example shows a single <select> ... </select> within [Inline] ... [/Inline] tags with a -Show command. If many value lists from the same database are being formatted, they can all be contained within a single set of [Inline] ... [/Inline] tags.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-Add" value="">
  <input type="hidden" name="-Database" value="Contacts">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-KeyField" value="ID">

[Inline: -Show, -Database='Contacts', -Table='People']
  <select name="Title">
    [Value_List: 'Title']
      <option value="[Value_ListItem]">[Value_ListItem]</option>
    [/Value_List]
  </select>
[/Inline]

  <p><input type="submit" name="-Add" value="Add Record">
</form>
```

- The [Option] tag can be used to easily format a value list as an HTML <select> ... </select> pop-up menu. The [Option] tag generates all of the <option> ... </option> tags for the pop-up menu based on the value list for the specified field. The example below generates exactly the same HTML as the example above.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-Add" value="">
  <input type="hidden" name="-Database" value="Contacts">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-KeyField" value="ID">

[Inline: -Show, -Database='Contacts', -Table='People']
  <select name="Title">
    [Option: 'Title']
  </select>
[/Inline]

  <p><input type="submit" name="-Add" value="Add Record">
</form>
```

To display HTML radio buttons with all values from a value list:

The following example shows how to format a set of HTML <input> tags to show all the values from a value list as radio buttons. The visitor will be able to select one value from the value list. Check boxes can be created with the same code by changing the type from radio to checkbox.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-Add" value="">
  <input type="hidden" name="-Database" value="Contacts">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-KeyField" value="ID">

[Inline: -Show, -Database='Contacts', -Table='People']
  [Value_List: 'Title']
    <input type="radio" name="Title" value="[Value_ListItem]"> [Value_ListItem]
  [/Value_List]
[/Inline]

  <p><input type="submit" name="-Add" value="Add Record">
</form>
```

To display only selected values from a value list:

The following examples show how to display the selected values from a value list for the current record. The record for John Doe is found within the database and the selected value for the Title field, Mr. is displayed.

- The -Selected keyword in the [Value_ListItem] tag ensures that only selected value list items are shown. The following example uses a conditional to check whether [Value_ListItem: -Selected] is empty.

```
[Inline: -Search, -Database='Contacts', -Table='People',
-KeyField='ID',
-KeyValue=126]
  [Value_List: 'Title']
```

```
    [If: (Value_ListItem: -Selected) != ' ']
      <br>[Value_ListItem: -Selected]
    [/If]
  [/Value_List]
[/Inline]
```

➜
Mr.

- The [Selected] tag ensures that only selected value list items are shown. The following example uses a conditional to check whether [Selected] is empty and only shows the [Value_ListItem] if it is not.

```
[Inline: -Search, -Database='Contacts', -Table='People',
-KeyField='ID',
-KeyValue=126]
  [Value_List: 'Title']
    [If: (Selected) != ' ']
      <br>[Value_ListItem]
    [/If]
  [/Value_List]
[/Inline]
```

➜
Mr.

- The [Field] tag can also be used simply to display the current value for a field without reference to the value list.

```
<br>[Field: 'Title']
```

➜
Mr.

To display an HTML pop-up menu in an -Update form with selected value list values:

- The following example shows how to format an HTML <select> ... </select> select list to show all the values from a value list with the selected values highlighted. The [Selected] tag returns Selected if the current value list item is selected in the database or nothing otherwise. This code will usually be used in an HTML form that performs an -Update action to allow the visitor to see what values are selected in the database currently and make different choices for the updated record.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-Update" value="">
  <input type="hidden" name="-Database" value="Contacts">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-KeyField" value="ID">
  <input type="hidden" name="-KeyValue" value="127">
```

```
[Inline: -Search, -Database='Contacts', -Table='People',
-KeyField='ID',
-KeyValue=126]
  <select name="Title" multiple size="4">
    [Value_List: 'Title']
      <option value="[Value_ListItem]" [Selected]>[Value_ListItem]</option>
    [/Value_List]
  </select>
[/Inline]

  <p><input type="submit" name="-Update" value="Update Record">
</form>
```

- The [Option] tag automatically inserts Selected parameters as needed to ensure that the proper options are selected in the HTML select list. The example below generates exactly the same HTML as the example above.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-Update" value="">
  <input type="hidden" name="-Database" value="Contacts">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-KeyField" value="ID">
  <input type="hidden" name="-KeyValue" value="127">

[Inline: -Search, -Database='Contacts', -Table='People',
-KeyField='ID',
-KeyValue=126]
  <select name="Title" multiple size="4">
    [Option: 'Title']
  </select>
[/Inline]

  <p><input type="submit" name="-Update" value="Update Record">
</form>
```

To display HTML check boxes with selected value list values:

The following example shows how to format a set of HTML <input> tags to show all the values from a value list as check boxes with the selected check boxes checked. The [Checked] tag returns Checked if the current value list item is selected in the database or nothing otherwise. Radio buttons can be created with the same code by changing the type from checkbox to radio.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-Update" value="">
  <input type="hidden" name="-Database" value="Contacts">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-KeyField" value="ID">
  <input type="hidden" name="-KeyValue" value="127">
```

```
[Inline: -Search, -Database='Contacts', -Table='People',
-KeyField='ID',
-KeyValue=126]
  [Value_List: 'Title']
    <input type="checkbox" name="Title" value="[Value_ListItem]" [Checked]>
      [Value_ListItem]
  [/Value_List]
[/Inline]

  <p><input type="submit" name="-Update" value="Update Record">
</form>
```

Note: Storing multiple values is only supported using SET field types.

Creating Database Tables

LDML 8 includes a set of tags which allow tables and fields to be created, altered, or deleted within MySQL data sources.

- A solution can create its required tables automatically the first time it is accessed.
- Temporary tables can be created which store data that is eliminated the next time MySQL is restarted.
- Interactive tools can be built which allow clients to create their own tables and populate them with data.

For a visual interface that allows MySQL databases (in addition to tables and fields) to be created and altered, see the Database Builder LassoApp. This interactive tool allows databases, tables, and fields to be created, altered, or deleted. See the *Building and Browsing Databases* chapter in the Lasso Professional 8 Setup Guide for details.

Lasso stores security information about all tables and fields in an internal cache. This table must be updated whenever a new table is created, new fields within a table are added, or fields are modified. The security cache can be updated manually using the Refresh button in the *Setup > Data Sources* sections of Lasso Administration, or programmatically using the [DataSource_Reload] tag. Lasso will automatically refresh the security cache whenever an unknown table or field name is used. Perform an [Inline] ... [/Inline] database action that references any new or changed tables and fields to force Lasso to update its security cache.

Warning: These tags can be used to delete entire data tables from MySQL data sources. These tags should be used with care to ensure that essential data is not lost. If a table or field is removed there is no way to access the data that was stored in the table or field without resorting to a backup.

Table 5: Database Creation Tags

Tag	Description
[Database_CreateTable]	Creates a table. Requires a -Database parameter which specifies a MySQL database and a -Table parameter which specifies the name of the table to be created.
[Database_CreateField]	Creates a field in a table. Requires -Database and -Table parameters which specify where the field should be created and -Field and -Type parameters which give the name of the field to be created and its type. [Database_CreateColumn] is a synonym.
[Database_ChangeField]	Changes a field definition. Requires the same parameters as [Database_CreateField] in addition to an -Original parameter that specifies the field to be altered. [Database_ChangeColumn] is a synonym.
[Database_RemoveTable]	Removes a table from a database. All data in the table will be lost. Requires -Database and -Table parameters.
[Database_RemoveField]	Removes a field from a table. All data in the field will be lost. Requires -Database, -Table, and -Field parameters. [Database_RemoveColumn] is a synonym.

Tables

Tables can be created or removed from MySQL data sources. The following important points should be kept in mind when creating or deleting new tables.

- Table names are case sensitive in MySQL, but case insensitive in Lasso. For best results use a consistent naming convention and never rely on case to differentiate between two tables.
- Table names should start with a letter and contain only letters, numbers, and the underscore character _. They should not contain any spaces, periods, or other punctuation.
- New tables must be enabled within Lasso Administration before they can be accessed through Lasso.
- All tables are created with a single field automatically named ID that is set to be the primary key field and to auto-increment from 0. Usually, this field should be used as the primary key field for a table unless another structure is required.
- In terms of data storage, tables are equivalent to FileMaker Pro database files, not to FileMaker Pro layouts. The equivalent of many FileMaker Pro databases can be stored in a single MySQL database.

  However, within Lasso security and Lasso database actions, FileMaker Pro databases and MySQL databases are treated as equivalent. FileMaker

Pro layouts are treated as equivalent to MySQL tables. This makes the security model cleaner and allows for easier transition between data sources.

- When a table is removed its data is lost forever. There is no undo. See the Site *Administration Utilities* chapter in the Lasso Professional 8 Setup Guide for information about backing up tables and data recovery.

Table 6: [Database_CreateTable] Parameters:

Parameter	Description
-Database	The name of the database in which to create the table.
-Table	The name of the table to be created. Must be unique within the database. Should start with a letter and contain only letters, numbers, or underscores.
-Temporary	Creates a temporary table that will be deleted when MySQL restarts.

To create a table:

Use the [Database_CreateTable] tag to create a new table in the specified database. The [Database_CreateTable] tag will not overwrite an existing table. The name of the new table must be unique. The following example creates a new table named Phone_Book in the database Example.

```
[Database_CreateTable: -Database='Example', -Table='Phone_Book']
```

The table initially contains one field named ID that is set to be the primary key field and to auto increment. Use the tags described in the *Fields* section below to add more fields to the new table.

Note: New tables are initially disabled in Lasso Administration. Use the *Setup > Data Sources* tab in Lasso Administration to enable new tables. In addition, the Extending Lasso Guide includes the complete source code for Admin.LassoApp which demonstrates how to enable new tables automatically.

To create a temporary table:

Use the [Database_CreateTable] tag with the -Temporary keyword to create a temporary table in the specified database. The temporary table will be deleted when MySQL restarts. The following example creates a new table named Cache in the database Example. This tag could be used in a format file within LassoStartup to create a table that started empty each time the server hosting Lasso was restarted.

```
[Database_CreateTable: -Database='Example', -Table='Cache', -Temporary]
```

The table initially contains one field named ID that is set to be the primary key field and to auto increment. Use the tags described in the *Fields* section below to add more fields to the new table.

To remove a table:

Use the [Database_RemoveTable] tag to drop the specified table from its database. This will eliminate all data stored in the table. The following example will remove the table named Cache from the Example database.

```
[Database_RemoveTable: -Database='Example', -Table='Cache']
```

Fields

Each table created by the [Database_CreateTable] command starts with only a single ID field which Lasso creates automatically. Additional fields can be created, changed, or removed from any table in a MySQL database. The following important points should be kept in mind when creating, changing, or removing fields.

- Field names should start with a letter and contain only letters, numbers, and the underscore character (_). They should not contain any spaces, periods, or other punctuation. See the MySQL documentation for a list of reserved names that cannot be used as field names.
- Tables can only contain a single primary key field and a single auto-increment field. Since the ID field is automatically created with both of these attributes it must be removed if a different field needs to be created as the primary key field.
- Fields should be created with the smallest data type which can hold all possible values. See the MySQL documentation at http://www.mysql.com for information on MySQL data types.
- When a field is removed its data is lost forever. There is no undo. See the *Site Administration Utilities* chapter in the Lasso Professional 8 Setup Guide for information about backing up tables and data recovery.
- After many fields have been added, changed, or removed from a table it is good practice to optimize the table following the instructions in the *Optimizing Tables* section below.

Table 7: [Database_CreateField] and [Database_ChangeField] Parameters:

Parameter	Description
-Database	The name of the database in which to create the table.
-Table	The name of the table in which to create the field.
-Original	Used only with [Database_ChangeField]. The name of the original field which should be changed.
-Field	The name of the field to be created. Must be unique within the table. Should start with a letter and contain only letters, numbers, or underscores.
-Type	The type of the field. See Table 5: MySQL Field Types for a summary of possible types.
-Default	Optional default value for the field. The field will be set to this value when a new record is created that does not set this field explicitly.
-AutoIncrement	Sets a field to auto increment. Only one field in each table can be set to auto increment. The field will be set to 1 greater than the maximum value of the field each time a new record is created that does not set this field explicitly. Optional
-Key	Sets a field as the primary key field. Only one field in each table can be set to be the primary key field. Optional.
-Null	Specifies that a field can contain Null values. The default.
-NotNull	Specifies that a field cannot contain Null values. Should be set for primary key or auto-increment fields. Optional.
-AfterField	Specifies where in the table the field should be created. The new field will be inserted after the named field. Optional.
-BeforeField	Specifies that a field should be created before all other fields in a table. Optional.

The -Type parameter for [Database_CreateField] and [Database_ChangeField] can accept any of the values in *Table 5: MySQL Field Types*.

Table 8: MySQL Field Types

Data Type	Description
TINYINT	Integer less than about one hundred. 8-bit.
SMALLINT	Integer less than about 30 thousand. 16-bit.
MEDIUMINT	Integer less than 8 million. 24-bit.
INT	Integer less than 2 billion. 32-bit. Recommended.
BIGINT	Very large integer. Same range as Lasso integer data type. 64-bit.
FLOAT	Short decimal value. 32-bit.
DOUBLE	Long decimal value. Same range as Lasso decimal data type. 64-bit. Recommended.
DECIMAL(length, precision)	Fixed precision decimal value. Ranges vary depending on parameters.
CHAR(length)	Fixed length string of the specified length. Length can be from 0 to 255.
VARCHAR(length)	Variable length string of the specified length. Length can be from 1 to 255.
TEXT, BLOB	Text or binary data up to about 64 KB in length.
TINYTEXT, TINYBLOB	Text or binary data up to 255 bytes. Rarely used.
MEDIUMTEXT, MEDIUMBLOB	Text or binary data up to about 16 MB. Rarely used.
LONGTEXT, LONGBLOB	Text or binary data up to about 4 GB. Practical limit of about 24 MB. Rarely used.
ENUM ('Value1', 'Value2', ...)	A field that can contain one of a number of predefined string values that are indexed numerically. ENUM data can be text referring to a value, or an integer referring to the index number of a value. A maximum of 65,535 ENUM values may be predefined.
SET ('Value1', 'Value2', ...)	A field that can contain up to 64 predefined string values. SET data can be comma-delimited text referring to many values, or as an integer that is the bit representation of the values.
DATETIME	Stores a MySQL date and time in YYYY-MM-DD HH:MM:SS format. Roughly equivalent to a Lasso date string, but with a different format.
TIMESTAMP	MySQL time stamp for modification date.
DATE	Stores a MySQL date string in YYYY-MM-DD format.
TIME	Stores a MySQL time string in HH:MM:SS format.
YEAR	Efficient storage for four digit years. Rarely used.

To create a field:

Use the [Database_CreateField] tag to create a new field. The field will be inserted as the last field in the specified table.

- The following example shows two fields First_Name and Last_Name added to the People table of the Contacts database. Both fields are set to the data type VARCHAR with a maximum length of 64 characters.

```
[Database_CreateField: -Database='Contacts', -Table='People',
  -Field='First_Name', -Type='VARCHAR(64)']
[Database_CreateField: -Database='Contacts', -Table='People',
  -Field='Last_Name', -Type='VARCHAR(64)']
```

- The following example shows a field Amount_Due being added to the People table. The field will store DECIMAL values with up to 14 digits and a precision of 2. This is a good data type for dollar amounts (up to $999,999,999,999.99).

```
[Database_CreateField: -Database='Contacts', -Table='People',
  -Field='Amount_Due', -Type='DECIMAL(14,2)']
```

- The following example shows a field Notes being added to the People table. The field can store TEXT values up to 64k worth of text.

```
[Database_CreateField: -Database='Contacts', -Table='People',
  -Field='Notes', -Type='TEXT']
```

- The following example shows a field Job being added to the People table. The field can store one ENUM value selected from a list of four allowed values (Sales, Support, Management, or Engineering).

```
[Database_CreateField: -Database='Contacts', -Table='People',
  -Field='Job', -Type='ENUM('Sales', 'Support', 'Management', 'Engineering')']
```

To create a field in an existing table:

A field can be created in an existing table by using the -AfterField or -BeforeFirst parameters to the [Database_CreateField] tag. The order of fields in a database is not generally important, but it can be easier to use command line tools if the fields print out in a specific order.

- The following example shows a field Phone_Number being added to the Phone_Book table immediately after the Last_Name field. The field is defined as a fixed length CHAR data type which can store up to 16 digits.

```
[Database_CreateField: -Database='Example', -Table='Phone_Book',
  -Field='Phone_Number', -Type='CHAR(16)', -AfterField='Last_Name']
```

- The following example shows a field Title being added to the Phone_Book table before all other fields in the table. The field is defined as a fixed length CHAR data type which can store up to 8 characters.

```
[Database_CreateField: -Database='Example', -Table='Phone_Book',
  -Field='Title', -Type='CHAR(8)', -BeforeFirst]
```

Note: Perform an [Inline] ... [/Inline] database action after creating a new field in order to force Lasso Administration to refresh and update its stored database information.

To change a field:

A field can be changed using the [Database_ChangeField] tag. This tag accepts all the same parameters as [Database_CreateField] with the addition of an -Original parameter that specifies the field to be changed. All of the parameters of the new field should be specified including the required name and type, any parameters left unspecified will be returned to their default values.

When a field is changed all the data in the field is translated to the new field type. Be sure to only change fields to compatible data types, otherwise there is a potential for data loss. If a field is changed to a smaller data type then any excess data beyond the size of the new data type will be lost.

The following example shows the Notes field from the Phone_Book table being changed so that it will only store about 255 characters in a TINYTEXT data type. Any characters beyond 255 in the records of Phone_Book will be truncated to 255 characters.

```
[Database_ChangeField: -Database='Example', -Table='Phone_Book',
  -Original='Notes', -Field='Notes', -Type='TINYTEXT']
```

To remove a field:

Use the [Database_RemoveField] tag to drop the specified field from its table. This will eliminate all data stored in the field. The following example will remove a field named Title from the Phone_Book table.

```
[Database_RemoveField: -Database='Example', -Table='Phone_Book',
  -Field='Title']
```

Optimizing Tables

After adding, changing, or removing many fields within a table it is good practice to optimize the table. This will ensure that the indices are up to date and that MySQL has updated all of its internal information about the table.

Please see the *Site Administration Utilities* chapter of the Lasso Professional 8 Setup Guide for more information about optimizing tables and automating database maintenance.

To optimize a table:

Use [Inline] ... [/Inline] tags with a -SQL command that specifies the OPTIMIZE TABLE and ANALYZE TABLE SQL statements. The following example optimizes the Phone_Book table of the Example database.

```
[Inline: -Database='Example', -SQL='OPTIMIZE TABLE Example.Phone_Book'][/
Inline]
[Inline: -Database='Example', -SQL='ANALYZE TABLE Example.Phone_Book'][/Inline]
```

11

Chapter 11
SQLite Data Source

This chapter documents tags and behaviors which are specific to the SQLite data source.

- *Overview* introduces the SQLite data source.
- *SQLite Tags* describes tags specific to the SQLite data source.
- *Searching Records* describes unique search operations that can be performed using the SQLite data source.
- *Adding and Updating Records* describes unique add and update operations that can be performed using the SQLite data source.
- *Storing Bytes* describes how to store byte streams in SQLite tables.
- *Creating SQLite Databases* describes how to create SQLite databases.

Overview

Lasso Professional 8 includes a built-in SQLite data source. This data source is used to store all of the internal settings and preferences for Lasso as well as the email queue, sessions, and more. This chapter documents tags and features unique to the SQLite data source.

Note: Although the built-in SQLite data source can be used for solution databases it is recommended that an external data source such as MySQL be used instead.

Tips for Using MySQL Data Sources

- Use -KeyField and -KeyValue to reference a particular record for updates, duplicates, or deletes.

- SQLite data sources can be case-sensitive. For best results, reference SQLite database and table names in the same letter-case as they appear on disk.
- SQLite is type-less. The type of a field only controls how the field is sorted (numeric versus alphabetic). Any SQLite field type can store any data.
- SQLite records can hold a maximum of 8MB of data.
- Use -ReturnField command tags to reduce the number of fields which are returned from a -Search action. Returning only the fields that need to be used for further processing or shown to the site visitor reduces the amount of data that needs to travel between Lasso Service and SQLite.
- When an -Add or -Update action is performed on a SQLite database, the data from the added or updated record is returned inside the [Inline] ... [/Inline] tags. If the -ReturnField parameter is used, then only those fields specified should be returned from an -Add or -Update action. Setting -MaxRecords=0 can be used as an indication that no record should be returned.
- When performing many -Add or INSERT operations on a SQLite database BEGIN TRANSACTION and COMMIT TRANSACTION should be used to group all of the operations into a single transaction. This will greatly improve the performance of SQLite.

Security Tips

- The -SQL command tag can only be allowed or disallowed at the host level for users in Site Administration. Once the -SQL command tag is allowed for a user, that user may access any database within the SQLite host. For that reason, only trusted users should be allowed to issue SQL queries using the -SQL command tag. For more information, see the *Setting Up Security* chapter in the Lasso Professional 8 Setup Guide.
- Always quote any inputs from site visitors that are incorporated into SQL statements. For example, the following SQL SELECT statement includes quotes around the [Action_Param] value and uses [Encode_SQL] to ensure that the user cannot enter any invalid characters. The quotes are escaped \' so they will be embedded within the string rather than ending the string literal. The semi-colon at the end of the statement is optional unless multiple statements are issued.

```
[Variable: 'SQL_Statement'='SELECT * FROM Contacts.People WHERE ' +
  'First_Name LIKE \'' + (Encode_SQL: (Action_Param: 'First_Name')) + '\';']
```

If [Action_Param] returns John for First_Name then the SQL statement generated by this code would appear as follows.

```
SELECT * FROM Contacts.People WHERE First_Name LIKE 'John';
```

SQLite Tags

LDML 8 includes tags to identify which type of data source is being used. These tags are summarized in *Table 1: SQLite Tags*.

Table 1: SQLite Tags

Tag	Description
[Lasso_DatasourceIsMySQLite]	Returns True if the database is in the internal SQLite host. Requires one string value, which is the name of a database.

To check whether a database is SQLite:

The following example shows how to use [Lasso_DatasourceIsSQLite] to check whether the database Example is hosted by SQLite or not.

```
[If: (Lasso_DatasourceIsSQLite: 'Example')]
  Example is hosted by SQLite!
[Else]
  Example is not hosted by SQLite.
[/If]
```

➜ Example is hosted by SQLite!

To list all databases hosted by SQLite:

Use the [Database_Names] ... [/Database_Names] tags to list all databases available to Lasso. The [Lasso_DatasourceIsSQLite] tag can be used to check each database and only those that are hosted by SQLite will be returned. The result shows two databases, Site and Example, which are available through SQLite.

```
[Database_Names]
  [If: (Lasso_DatasourceIsSQLite: (Database_NameItem))]
    <br>[Database_NameItem]
  [/If]
[/Database_Names]
```

➜
Example

Site

Searching Records

In LDML 8, there are unique search operations that can be performed using SQLite data sources. These search operations take advantage of special functions in MySQL such as record limits and distinct values to allow optimal performance and power when searching.

Search Command Tags

Additional search command tags are available when searching SQLite data sources using the [Inline] tag. These tags allow special search functions specific to SQLite to be performed without writing SQL statements. These operators are summarized in *Table 2: SQLite Search Command Tags*.

Table 2: SQLite Search Command Tags

Tag	Description
-UseLimit	Prematurely ends a -Search or FindAll action once the specified number of records for the -MaxRecords tag have been found and returns the found records. Requires the -MaxRecords tag. This issues an internal LIMIT statement to MySQL to cause it to search more efficiently.
-Distinct	Causes a -Search action to only output records that contain unique field values (comparing only returned fields). Does not require a value. May be used with the -ReturnField parameter to limit the fields checked for distinct values.
-GroupBy	Specifies a field name which should by used as the GROUP BY statement. Allows data to be summarized based on the values of the specified field.

To have SQLite immediately return records once a limit is reached:

Use the -UseLimit tag in the search inline. Normally, Lasso will find all records that match the inline search criteria and then pair down the results based on -MaxRecords and -SkipRecords values. The -UseLimit tag instructs MySQL to terminate the specified search process once the number of records specified for -MaxRecords is found. The following example searches the Contacts database with a limit of five records.

```
[Inline: -FindAll,
-Database='Contacts', -Table='People',
-MaxRecords='5',
-UseLimit]
[Found_Count]
[/Inline]

➜ 5
```

Note: If the -UseLimit tag is used, the value of the [Found_Count] tag will always be the same as the -MaxRecords value if the limit is reached. Otherwise, the [Found_Count] tag will return the total number of records in the specified table that match the search criteria if -UseLimit is not used.

To return only unique records in a search:

Use the -Distinct parameter in a search inline. The following example only returns records that contain distinct values for the Last_Name field.

```
[Inline: -FindAll, -Database='Contacts', -Table='People',
-ReturnField='Last_Name',
-Distinct]
  [Records]
    [Field:'Last_Name']<br>
  [/Records]
[/Inline]

➜ Doe<br>
Surname<br>
Lastname<br>
Unknown<br>
```

The -Distinct tag is especially useful for generating lists of values that can be used in a pull-down menu. The following example is a pull-down menu of all the last names in the Contacts database.

```
[Inline: -Findall, -Database='Contacts', -Table='People',
-ReturnField='Last_Name',
-Distinct]
  <select name="Last_Name">
    [Records]
      <option value="[Field: 'Last_Name']">
        [Field: 'Last_Name']
      </Option>
    [/Records]
  </Select>
[/Inline]
```

Storing Bytes

The internal SQLite data source allows binary data to be stored in any field using the following syntax. This syntax can only be specified within a SQL statement. The data is expected to be encoded in hexadecimal using the [Encode_Hex] tag.

```
INSERT INTO table (field) VALUES (x" ... HEX DATA ...");
```

When Lasso retrieves data from the field it will be automatically decoded into a byte stream. It is not necessary to use [Decode_Hex] on the return value from the [Field] tag.

For example, the following [Inline] would insert a byte stream into a SQLite table.

```
[Var: 'bytes' = (Bytes: ' ... Byte Stream ... ')]
[Inline: -Database='Example', -Table='Example',
    -SQL='INSERT INTO example (field) VALUES (x'" + (Encode_Hex: $bytes) + "');']
[/Inline]
```

Then the following code can be used to retrieve the value from the database. The result in the variable $bytes will be a byte stream that exactly matches the value that was stored.

```
[Inline: -Database='Example', -Table='Example', -FindAll]
  [Var: 'bytes' = (Field: 'field')]
[/Inline]

[Var: 'bytes'] ➜ ... Byte Stream ...
```

Creating SQLite Databases

SQL statements can be used to create SQLite tables within an existing database. Consult the documentation for SQLite directly to see what syntax to use. Lasso Professional 8 also provides the Build section of DatabaseBrowser.LassoApp which allows SQLite databases and tables to be created.

However, SQLite databases cannot be created using SQL statements. Lasso provides a tag [SQLite_CreateDB] that creates a new SQLite database within the built-in SQLite host for the current site.

Table 3: SQLite Database Create Tag

Tag	Description
[SQLite_CreateDB]	The specified database will be created within the site SQLiteDBs folder. Requires one parameter, the name of the database to be created.

12

Chapter 12
FileMaker Data Sources

This chapter documents tags and behaviors which are specific to FileMaker Pro and FileMaker 7 Server Advanced data sources accessed using Lasso Connector for FileMaker Pro and Lasso Connector for FileMaker SA.

- *Overview* introduces FileMaker data sources.
- *Performance Tips* includes recommendations which will help ensure that FileMaker is used to its full potential.
- *Compatibility Tips* includes recommendations which help ensure that FileMaker databases can be transferred to a different back-end data source.
- *FileMaker Tags* describes tags specific to FileMaker data sources.
- *Primary Key Field and Record ID* describes how the built-in record IDs in FileMaker can be used as primary key fields.
- *Sorting Records* describes how custom sorts can be performed in FileMaker databases.
- *Displaying Data* describes methods of returning field values from FileMaker databases including repeating field values and values from portals.
- *Value Lists* describes how to retrieve and format value list data from FileMaker databases.
- *Container Fields* describes how to retrieve images and other data stored in container fields.
- *FileMaker Scripts* describes how to activate FileMaker scripts in concert with a Lasso database action.

Overview

Lasso Professional 8 allows access to FileMaker Pro data sources through Lasso Connector for FileMaker Pro. Connections can be made to any version of FileMaker Pro that includes Web Companion including FileMaker Pro 4.x and FileMaker Pro 5.x and 6.x Unlimited. FileMaker Pro 3 is not supported nor are solutions which use the FileMaker runtime engine.

Lasso Professional 8 also allows access to FileMaker 7 Server Advanced through Lasso Connector for FileMaker SA. Lasso cannot connect to FileMaker Pro 7.

Please see the *Setting Up Data Sources* chapter in the Lasso Professional 8 Setup Guide for information about how to configure FileMaker for access through Lasso Professional 8.

LDML is a predominantly data source-independent language. It does include many FileMaker specific tags which are documented in this chapter. However, all of the common procedures outlined in the *Data Source Fundamentals*, *Searching and Displaying Data*, and *Adding and Updating Records* chapters can be used with FileMaker data sources.

Note: The tags and procedures defined in this chapter can only be used with FileMaker data sources. Any solution which relies on these tags cannot be easily retargeted to work with a different back-end database.

Terminology

Since Lasso works with many different data sources this documentation uses data source agnostic terms to refer to databases, tables, and fields. The following terms which are used in the FileMaker documentation are equivalent to their Lasso counterparts.

- **Database** – Database is used to refer to a single FileMaker database file. FileMaker databases differ from other databases in Lasso in that they contain layouts rather than individual data tables. Even in FileMaker 7 Lasso see individual layouts rather than data tables. From a data storage point of view, a FileMaker database is equivalent to a single Lasso MySQL table.
- **Layout** – Within Lasso a FileMaker layout is treated as equivalent to a **Table**. The two terms can be used interchangeably. This equivalence simplifies Lasso security and makes transitioning between back-end data sources easier. All FileMaker layouts can be thought of as views of a single data table. Lasso can only access fields which are contained in the layout named within the current database action.

- **Record** – FileMaker records are referenced using a single -KeyValue rather than a -KeyField and -KeyValue pair. The -KeyField in FileMaker is always the record ID which is set internally.
- **Fields** – The value for any field in the current layout in FileMaker can be returned including the values for related fields, repeating fields, and fields in portals.

Although the equivalence of FileMaker databases to Lasso MySQL databases and FileMaker layouts to Lasso MySQL tables is imperfect, it is an essential compromise in order to map both database models onto Lasso Professional's two-tier (e.g. database and table) security model.

Performance Tips

This section contains a number of tips which will help get the best performance from a FileMaker database. Since queries must be performed sequentially within FileMaker, even small optimizations can yield significant increases in the speed of Web serving under heavy load.

- **Dedicated FileMaker Machine** – For best performance, place the FileMaker Pro or FileMaker 7 Server Advanced application on a different machine from Lasso Service and the Web server application.
- **FileMaker Server** – If a FileMaker database must be accessed by a mix of FileMaker clients and Web visitors through Lasso, it should be hosted on FileMaker Server. Lasso can access the database directly through FileMaker 7 Server Advanced or through a single FileMaker Pro client which is connected as a guest to FileMaker Server.
- **Web Companion** – When using FileMaker Pro, always ensure that the latest version of FileMaker Web Companion for the appropriate version of FileMaker is installed.
- **Index Fields** – Any fields which will be searched through Lasso should have indexing turned on. Avoid searching on unstored calculation fields, related fields, and summary fields.
- **Custom Layouts** – Layouts should be created with the minimal number of fields required for Lasso. All the data for the fields in the layout will be sent to Lasso with the query results. Limiting the number of fields can dramatically cut down the amount of data which needs to be sent from FileMaker to Lasso.
- **Return Fields** – Use the -ReturnField tag to limit the number of fields which are returned to Lasso. If no -ReturnField tag is specified then all of the data for the fields in the current layout will be sent to Lasso with the query results.

Note: -ReturnField does not work with FileMaker 7 Server Advanced.

- **Sorting** – Sorting can have a serious impact on performance if large numbers of records must be sorted. Avoid sorting large record sets and avoid sorting on calculation fields, related fields, unindexed fields, or summary fields.
- **Contains Searching** – FileMaker is optimized for the default Begins With searches (and for numerical searches). Use of the contains operator cn can dramatically slow down performance since FileMaker will not be able to use its indices to optimize searches.
- **Max Records** – Using -MaxRecords to limit the number of records returned in the result set from FileMaker can speed up performance. Use -MaxRecords and -SkipRecords or the [Link_...] tags to navigate a visitor through the found set.
- **Calculation Fields** – Calculation fields should be avoided if possible. Searching or sorting on unindexed, unstored calculation fields can have a negative effect on FileMaker performance.
- **FileMaker Scripts** – The use of FileMaker scripts should be avoided if possible. When FileMaker executes a script, no other database actions can be performed at the same time. FileMaker scripts can usually be rewritten as LDML to achieve the same effect as the script, often with greater performance.

In addition to these tips, MySQL can be used to shift some of the burden off of FileMaker. MySQL can usually perform database searches much faster than FileMaker. Lasso Professional 8 also includes sessions and compound data types that can be used to perform some of the tasks of a database, but with higher performance for small amounts of data.

Compatibility Tips

Following these tips will help to ensure that it easy to transfer data from a FileMaker database to another data source, such as the built-in Lasso MySQL database, at a future date.

- **Database Names** – Database, layout, and field names should contain only a mix of letters, numbers, and the underscore character. They should not contain any punctuation other than spaces..
- **Calculation Fields** – Avoid the use of calculation fields. Instead, perform calculations within Lasso and store the results back into regular fields if they will be needed later.
- **Summary Fields** – Avoid the use of summary fields. Instead, summarize data using [Inline] searches within Lasso.

- **Scripts** – Avoid the use of FileMaker scripts. Most actions which can be performed with scripts can be performed using the database actions available within Lasso.
- **Record ID** – Create a calculation field named ID and assign it to the following calculation. Always use the -KeyField='ID' within [Inline] database actions, HTML forms, and URLs. This ensures that when moving to a database that relies on storing the key field value explicitly, a unique key field value is available.

```
Status(CurrentRecordID)
```

FileMaker Tags

LDML 8 includes tags that allow the type of a database to be inspected.

Table 1: FileMaker Data Source Tag

Tag	Description
[Lasso_DataSourceIsFileMaker]	Returns True if the specified database is hosted by FileMaker Pro.
[Lasso_DataSourceIsFileMakerSA]	Returns True if the specified database is hosted by FileMaker 7 Server Advanced.

To check whether a database is hosted by FileMaker:

The following example shows how to use [Lasso_DataSourceIsFileMaker] and [Lasso_DataSourceIsFilemakerSA] to check whether or note the database Example is hosted by FileMaker Pro or FileMaker 7 Server Advanced.

```
[If: (Lasso_DataSourceIsFileMaker: 'Example.fp5')]
  Example is hosted by FileMaker Pro!
[Else: (Lasso_DataSourceIsFileMakerSA: 'Example.fp5')]
  Example is hosted by FileMaker 7 Server Advanced!
[Else]
  Example is not hosted by FileMaker Pro.
[/If]
```

```
➜ Example is hosted by FileMaker Pro!
```

To list all databases hosted by FileMaker:

Use the [Database_Names] ... [/Database_Names] tags to list all databases available to Lasso. The [Lasso_DataSourceIsFileMaker] tag and [Lasso_DataSourceIsFileMakerSA] tag can be used to check each database and only those that are hosted by FileMaker Pro will be returned. The result

shows two databases, Contacts.fp5 and Example.fp5, which are available through FileMaker Pro.

```
[Database_Names]
  [If: (Lasso_DataSourceIsFileMaker: (Database_NameItem))]
    <br>FMP [Database_NameItem]
  [Else: (Lasso_DataSourceIsFileMakerSA: (Database_NameItem))]
    <br>FMSA [Database_NameItem]
  [/If]
[/Dabase_Names]
```

```
➜ <br>FMP Example.fp5
  <br>FMP Contacts.fp5
```

Primary Key Field and Record ID

FileMaker databases include a built-in primary key value called the Record ID. This value is guaranteed to be unique for any record in a FileMaker database. It is predominantly sequential, but should not be relied upon to be sequential. The values of the record IDs within a database may change after an import or after a database is compressed using Save a Copy As.... Record IDs can be used within a solution to refer to a record on multiple pages, but should not be stored as permanent references to FileMaker records.

Note: The tag [RecordID_Value] can also be used to retrieve the record ID from FileMaker records. However, for best results, it is recommended that the [KeyField_Value] tag be used.

To return the current record ID:

The record ID for the current record can be returned using [KeyField_Value]. The following example shows [Inline] ... [/Inline] tags that perform a -FindAll action and return the record ID for each returned record using the [KeyFIeld_Value] tag.

```
[Inline: -Database='Contacts.fp5', -Layout='People', -FindAll]
  [Records]
    <br>[KeyField_Value]: [Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]
```

```
➜ <br>126: John Doe
  <br>127: Jane Doe
  <br>4096: Jane Person
```

To reference a record by record ID:

For -Update and -Delete command tags the record ID for the record which should be operated upon can be referenced using -KeyValue. The -KeyField does not need to be defined or should be set to an empty string if it is, -KeyField=''.

- The following example shows a record in Contacts.fp5 being updated with -KeyValue=126. The name of the person referenced by the record is changed to John Surname.

```
[Inline: -Database='Contacts.fp5',
    -Layout='People',
    -KeyValue=126,
    'First_Name'='John',
    'Last_Name'='Surname',
    -Update]
  <br>[KeyField_Value]: [Field: 'First_Name'] [Field: 'Last_Name']
[/Inline]
```

➜
126: John Surname

- The following example shows a record in Contacts.fp5 being deleted with -KeyValue=127. The -KeyField command tag is included, but its value is set to the empty string.

```
[Inline: -Database='Contacts.fp5',
    -Layout='People',
    -KeyField='',
    -KeyValue=126,
    -Delete]
[/Inline]
```

To access the record ID within FileMaker:

The record ID for the current record in FileMaker can be accessed using the calculation value Status(CurrentRecordID) within FileMaker.

Sorting Records

In addition to the Ascending and Descending values for the -SortOrder tag, FileMaker data sources can also accept a Custom value. The Custom value can be used for any field which is formatted with a value list in the current layout. The field will be sorted according to the order of values within the value list.

Note: FileMaker Server Advanced only supports the specification of a maximum of 9 sort fields in a single database search.

To return custom sorted results:

Specify -SortField and -SortOrder command tags within the search parameters. The following [Inline] ... [/Inline] tags include sort command tags specified in hidden inputs. The records are first sorted by title in custom order, then by Last_Name and First_Name in ascending order. The Title field will be sorted in the order of the elements within the value list associated with the field in the database. In this case, it will be sorted as Mr., Mrs., Ms.

```
[Inline: -FindAll,
    -Database='Contacts.fp5',
    -Table='People',
    -KeyField='ID',
    -SortField='Title', -SortOrder='Custom',
    -SortField='Last_Name', -SortOrder='Ascending',
    -SortField='First_Name', -SortOrder='Ascending']
  [Records]
    <br>[Field: 'Title'] [Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]
```

The following results could be returned when this page is loaded. Each of the records with a title of Mr. appear before each of the records with a title of Mrs. Within each title, the names are sorted in ascending alphabetical order.

```
➜ <br>Mr. John Doe
  <br>Mr. John Person
  <br>Mrs. Jane Doe
  <br>Mrs. Jane Person
```

Displaying Data

FileMaker includes a number of container tags and substitution tags that allow the different types of FileMaker fields to be displayed. These tags are summarized in *Table 2: FileMaker Data Display Tags* and then examples are included in the sections that follow.

See also the sections on *Value Lists* and *Images* for more information about returning values from FileMaker fields.

Table 2: FileMaker Data Display Tags

Tag	Description
[Field]	Can be used to reference FileMaker fields including related fields and repeating fields.
[Repeating] ... [/Repeating]	Container tag repeats for each defined repetition of a repeating field. Requires a single parameter, the name of the repeating field from the current layout.
[Repeating_ValueItem]	Returns the value for each repetition of a repeating field.
[Portal] ... [/Portal]	Container tag repeats for each record in a portal. Requires a single parameter, the name of the portal relationship from the current layout.

Note: All fields which are referenced by Lasso must be contained in the current layout in FileMaker. For portals and repeating fields only the number of repetitions shown in the current layout will be available to Lasso.

Related Fields

Related fields are named using the relationship name followed by two colons :: and the field name. For example, a related field Call_Duration from a Calls.fp5 database might be referenced as Calls.fp5::Call_Duration. Any related fields which are included in the layout specified for the current Lasso action can be referenced using this syntax. Data can be retrieved from related fields or it can be set in related fields when records are added or updated.

To return data from a related field:

Specify the name of the related field within a [Field] tag. The related field must be contained in the current layout either individually or within a portal. In a one-to-one relationship, the value from the single related record will be returned. In a one-to-many relationship, the value from the first related record as defined by the relationship options will be returned. See the section on *Portals* below for more control over one-to-many relationships.

The following example shows a -FindAll action being performed in a database Contacts.fp5. The related field Last_Call_Time from the Calls.fp5 databases is returned for each record through a relationship named Calls.fp5.

```
[Inline: -Database='Contacts.fp5', -Layout='People', -FindAll]
  [Records]
    <br>[KeyField_Value]: [Field: 'First_Name'] [Field: 'Last_Name]
      (Last call at: [Field: 'Calls::Last_Call_Time']).
  [/Records]
[/Inline]
```

```
➜ <br>126: John Doe (Last call at 12:00 pm).
<br>127: Jane Doe (Last call at 9:25 am).
<br>4096: Jane Person (Last call at 4:46 pm).
```

To set the value for a related field:

Specify the name of the related field within the action which adds or updates a record within the database. The related field must be contained in the current layout either individually or within a portal. In a one-to-one relationship, the value for the field in a single related record will be modified. In a one-to-many relationship, the value for the field in the first related record as defined by the relationship options will be modified. See the section on *Portals* below for more control over one-to-many relationships.

The following example shows an -Update action being performed in a database Contacts.fp5. The related field Last_Call_Time from the Calls.fp5 database is updated for Jane Person. The new value is returned.

```
[Inline: -Database='Contacts.fp5',
    -Layout='People',
    -KeyValue=4096,
    'Calls.fp5::Last_Call_Time'='1:45 am',
    -Update]
    <br>[KeyField_Value]: [Field: 'First_Name'] [Field: 'Last_Name]
      (Last call at: [Field: 'Calls.fp5::Last_Call_Time']).
[/Inline]
```

```
➜ <br>4096: Jane Person (Last call at 1:45 pm).
```

Portals

Portals allow one-to-many relationships to be displayed within FileMaker databases. Portals allow data from many related records to be retrieved and displayed in a single format file. A portal must be present in the current FileMaker layout in order for its values to be retrieved using Lasso.

Only the number of repetitions formatted to display within FileMaker will be displayed using Lasso. A portal must contain a scroll bar in order for all records from the portal to be displayed using Lasso.

Fields in portals are named using the same convention as related fields. The relationship name is followed by two colons :: and the field name. For example, a related field Call_Duration from a Calls.fp5 database might be referenced as Calls.fp5::Call_Duration.

Note: Everything that is possible to do with portals can also be performed using nested [Inline] ... [/Inline] tags to perform actions in the related database. Portals are unique to FileMaker databases.

To return values from a portal:

Use the [Portal] ... [/Portal] tags with the name of the portal referenced in the opening [Portal] tag. [Field] tags within the [Portal] ... [/Portal] tags should reference the fields from the current portal row using related field syntax.

The following example shows a portal Calls.fp5 that is contained in the People layout of the Contacts.fp5 database. The Time, Duration, and Number of each call is displayed.

```
[Inline: -Database='Contacts.fp5', -Layout='People', -FindAll]
  [Records]
    <p>Calls for [Field: 'First_Name'] [Field: 'Last_Name]:
    [Portal: 'Calls.fp5']
      <br>[Field: 'Calls.fp5::Number'] at [Field: 'Calls.fp5::Time']
        for [Field: 'Calls.fp5::Duration'] minutes.
    [/Portal]
  [/Records]
[/Inline]
```

```
➜ <p>Calls for John Doe:
<br>555-1212 at 12:00 pm for 15 minutes.

<p>Calls for Jane Doe:
<br>555-1212 at 9:25 am for 60 minutes.

<p>Calls for Jane Person:
<br>555-1212 at 2:23 pm for 55 minutes.
<br>555-1212 at 4:46 pm for 5 minutes.
```

To add a record to a portal:

A record can be added to a portal by adding the record directly to the related database. In the following example the Calls.fp5 database is related to the Contacts.fp5 database by virtue of a field Contact_ID that stores the ID for the contact which the calls were made to. New records added to Calls.fp5 with the appropriate Contact_ID will be shown through the portal to the next site visitor.

In the following example a new call is added to the Calls.fp5 database for John Doe. John Doe has an ID of 123 in the Contacts.fp5 database. This is the value used for the Contact_ID field in Calls.fp5.

```
[Inline: -Add,
  -Database='Calls.fp5',
  -Layout='People',
  'Contact_ID'=123,
  'Number'='555-1212',
  'Time'='12:00 am',
  'Duration'=55]
[/Inline]
```

To update a record within a portal:

In order to update records shown within a portal it is recommended that you use a field to return the record ID of each record in the portal, then use that value in nested [Inline] ... [/Inline] tags to update the related record.

Create a calculation field named RecordID within the related database (e.g. Calls.fp5) that contains the following FileMaker calculation.

```
Status(CurrentRecordID)
```

Place that field within the portal shown within the main database (e.g. Contacts.fp5). To perform an update of a portal row, use [Inline] ... [/Inline] tags which reference the related database and the RecordID from the portal.

The following example shows how to update every record contained within a portal. The field Approved is set to Yes for each call from the Calls.fp5 database for all contacts from the Contacts.fp5 database.

```
[Inline: -Database='Contacts.fp5', -Layout='People', -FindAll]
  [Records]
    [Portal: 'Calls.fp5']
      [Inline: -Database='Calls.fp5',
        -Layout='People',
        -KeyField=(Field: 'Calls.fp5::RecordID'),
        'Approved'='Yes',
        -Update']
      [/Inline]
    [/Portal]
  [/Records]
[/Inline]
```

The results of the action will be shown the next time the portal is viewed by a site visitor.

To delete a record from a portal:

The same method as described above for updating records within a portal can be used to delete records from a portal. In the following example, all records from Contacts.fp5 are returned and every record from the Calls.fp5 portal is deleted.

```
[Inline: -Database='Contacts.fp5', -Layout='People', -FindAll]
  [Records]
    [Portal: 'Calls.fp5']
      [Inline: -Database='Calls.fp5',
        -Layout='People',
        -KeyField=(Field: 'Calls.fp5::RecordID'),
        -Delete]
      [/Inline]
    [/Portal]
  [/Records]
[/Inline]
```

No records will be contained in the portal the next time the site is viewed by a site visitor. However, not all records in Calls.fp5 have necessarily been deleted. Any records which were not associated with a contact in Contacts.fp5 will still remain in the database.

Repeating Fields

Repeating fields in FileMaker allow many values to be stored in a single field. Each repeating field is defined to hold a certain number of values. These values can be retrieved using the tags defined in this section. See the documentation for FileMaker for more information about how to create and use repeating fields within FileMaker.

In order to display or set values in a repeating field, the layout referenced in the current database action must contain the repeating field formatted to show the desired number of repetitions. If a field is set to store eight repetitions, but only to show two, then it will appear to be a two-repetition field to Lasso.

Note: The use of repeating fields is not recommended. Usually a simple text field which contains multiple values separated by returns can be used for the same effect through Lasso. For more complex solutions a related database and [Portal] ... [/Portal] tags or nested [Inline] ... [/Inline] tags can often be easier to use and maintain than a solution with repeating fields.

To return values from a repeating field:

Use the [Repeating] ... [/Repeating] and [Repeating_ValueItem] tags to return each of the values from a repeating field. The opening [Repeating] tag takes a single parameter which names a field from the current FileMaker layout that repeats. The contents of the [Repeating] ... [/Repeating] tags is repeated for each repetition and the [Repeating_ValueItem] tag is used to return the value for the current repetition.

The following example shows a repeating field Customer_ID that has four repetitions. Normally, only the first repetition has a defined value, but for a contact that has multiple accounts, multiple values are defined. Since Jane Person has two customer accounts, two repetitions of Customer_ID are returned.

```
[Inline: -Database='Contacts', -Layout='People', 'Last_Name'='Person', -Search]
  [Records]
    <p>[Field: 'First_Name'] [FIeld: 'Last_Name']
    [Repeating: 'Customer_ID']
      <br>Customer ID [Loop_Count]: [Repeating_ValueItem].
    [/Repeating]
  [/Records]
[/Inline]

➜ <p>Jane Person
<br>Customer ID 1: 100123.
<br>Customer ID 2: 123654.
```

To add a record with a repeating field:

A record can be added with values in a repeating field by referencing the field multiple times within the -Add action. The following example shows a new contact being added to Contacts.fp5. The contact Jimmy Last_Name is given three customer ID numbers referenced by the field Customer_ID multiple times. The added record is returned showing all three customer IDs are stored.

```
[Inline: -Database='Contacts',
  -Layout='People',
  'First_Name'='Jimmy',
  'Last_Name'='Last_Name',
  'Customer_ID'='2001',
  'Customer_ID'='2010',
  'Customer_ID'='2061',
  -Add]
    <p>[Field: 'First_Name'] [Field: 'Last_Name']
    [Repeating: 'Customer_ID']
      <br>Customer ID [Loop_Count]: [Repeating_ValueItem].
    [/Repeating]
[/Inline]
```

```
➜ <p>Jimmy Last_Name
  <br>Customer ID 1: 2001.
  <br>Customer ID 2: 2010.
  <br>Customer ID 3: 2061.
```

To update a record with a repeating field:

A repeating field can be updated by referencing it multiple times within the -Update action. The following example shows an HTML form which displays four repetitions of the field Customer_ID and allows each of them to be modified. Notice that the four repetitions are created using the looping [Repeating] ... [/Repeating] container tags.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-Database" value="Contacts.fp5">
  <input type="hidden" name="-Layout" value="People">
  <input type="hidden" name="-KeyValue" value="[KeyField_Value]">

  <p>First Name:
    <input type="text" name="First_Name" value="[Field: 'First_Name']">
  <br>Last Name:
    <input type="text" name="Last_Name" value="[Field: 'Last_Name']">

  [Repeating: 'Customer ID'
    <br>Customer ID:
      <input type="text" name="Customer_ID" value="[Repeating_ValueItem]">
  [/Repeating]

  <p><input type="submit" name="-Update" value="Update this Record">
</form>
```

To delete values from a repeating field:

- Records which contain repeating fields can be deleted using the same technique for deleting any FileMaker records. All repetitions of the repeating field will be deleted along with the record. The following [Inline] ... [/Inline] tags will delete the record with a record ID of 127.

  ```
  [Inline: -Database='Contacts.fp5', -Table='People', -KeyValue=127, -Delete]
    <p>The record was deleted.
  [/Inline]
  ```

- A single repetition of a repeating field can be deleted by setting its value to an empty string. The other values in the repeating field will not slide down to fill in the missing repetition. The following [Inline] ... [/Inline] will set the first repetition of a repeating field Customer_ID to the empty string, but leave the second and third repetitions unchanged.

  The values for the repeating field are first placed in an array so that they can be referenced by number within the opening [Inline] tag.

```
[Variable: 'Customer_ID' = (Array: '', '', '')]
[Repeating: 'Customer_ID']
  [(Variable: 'Customer_ID')->(Get: Loop_Count) = (Repeating_ValueItem)]
[/Repeating]

[Inline: -Update,
    -Database='Contacts.fp5',
    -Table='People',
    -KeyValue=127,
    'Customer_ID'='',
    'Customer_ID'=(Variable: 'Customer_ID')->(Get: 2),
    'Customer_ID'=(Variable: 'Customer_ID')->(Get: 3),
  <p>[Field: 'First_Name'] [FIeld: 'Last_Name']
  [Repeating: 'Customer_ID']
    <br>Customer ID [Loop_Count]: [Repeating_ValueItem].
  [/Repeating]
[/Inline]
```

The results show that the value for the first repetition of the repeating field has been deleted, but the second and third repetitions remain intact.

```
➜ <p>Jimmy Last_Name
  <br>Customer ID 1: .
  <br>Customer ID 2: 2010.
  <br>Customer ID 3: 2061.
```

Value Lists

Value lists in FileMaker allow a set of possible values to be defined for a field. The items in the value list associated with a field on the current layout for a Lasso action can be retrieved using the tags defined in *Table 3: FileMaker Value List Tags*. See the documentation for FileMaker for more information about how to create and use value lists within FileMaker.

In order to display values from a value list, the layout referenced in the current database action must contain a field formatted to show the desired value list as a pop-up menu, select list, check boxes, or radio buttons. Lasso cannot reference a value list directly. Lasso can only reference a value list through a formatted field in the current layout.

Table 3: FileMaker Value List Tags

Tag	Description
[Value_List] ... [/Value_List]	Container tag repeats for each value in the named value list. Requires a single parameter, the name of a field from the current layout which has a value list assigned to it.
[Value_ListItem]	Returns the value for the current item in a value list. Optional -Checked or -Selected parameter returns only currently selected values from the value list.
[Selected]	Displays the word Selected if the current value list item is selected in the field associated with the value list.
[Checked]	Displays the word Checked if the current value list item is selected in the field associated with the value list.
[Option]	Generates a series of <option> tags for the value list. Requires a single parameter, the name of a field from the current layout which has a value list assigned to it.

Note: See the *Searching and Displaying Data* chapter for information about the -Show command tag which is used throughout this section.

To display all values from a value list:

- The following example shows how to display all values from a value list using a -Show action within [Inline] ... [/Inline] tags. The field Title in the Contacts.fp5 database contains five values Mr., Mrs., Ms., and Dr. The -Show action allows the values for value lists to be retrieved without performing a database action.

```
[Inline: -Database='Contacts.fp5', -Layout='People', -Show]
  [Value_List: 'Title']
    <br>[Value_ListItem]
  [/Value_List]
[/Inline]
```

```
➜ <br>Mr.
<br>Mrs.
<br>Ms.
<br>Dr.
```

- The following example shows how to display all values from a value list using a named inline. The same name Values is referenced by -InlineName in both the [Inline] tag and [Value_List] tag.

```
[Inline: -InlineName='Values', -Database='Contacts.fp5', -Layout='People', -Show]
[/Inline]
…
[Value_List: 'Title', -InlineName='Values']
  <br>[Value_ListItem]
[/Value_List]
```

```
➜ <br>Mr.
<br>Mrs.
<br>Ms.
<br>Dr.
```

To display an HTML pop-up menu in an -Add form with all values from a value list:

- The following example shows how to format an HTML <select> ... </select> pop-up menu to show all the values from a value list. A select list can be created with the same code by including size and/or multiple parameters within the <select> tag. This code is usually used within an HTML form that performs an -Add action so the visitor can select a value from the value list for the record they create.

  The example shows a single <select> ... </select> within [Inline] ... [/Inline] tags with a -Show command. If many value lists from the same database are being formatted, they can all be contained within a single set of [Inline] ... [/Inline] tags.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-Add" value="">
  <input type="hidden" name="-Database" value="Contacts.fp5">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-KeyField" value="ID">

[Inline: -Database='Contacts.fp5', -Layout='People', -Show]
  <select name="Title">
    [Value_List: 'Title']
      <option value="[Value_ListItem]">[Value_ListItem]</option>
    [/Value_List]
  </select>
[/Inline]

  <p><input type="submit" name="-Add" value="Add Record">
</form>
```

- The [Option] tag can be used to easily format a value list as an HTML <select> ... </select> pop-up menu. The [Option] tag generates all of the <option> ... </option> tags for the pop-up menu based on the value list for the specified field. The example below generates exactly the same HTML as the example above.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-Add" value="">
  <input type="hidden" name="-Database" value="Contacts.fp5">
  <input type="hidden" name="-Table" value="People"?
  <input type="hidden" name="-KeyField" value="ID">

[Inline: -Database='Contacts.fp5', -Layout='People', -Show]
  <select name="Title">
    [Option: 'Title']
  </select>
[/Inline]

  <p><input type="submit" name="-Add" value="Add Record">
</form>
```

To display HTML radio buttons with all values from a value list:

The following example shows how to format a set of HTML <input> tags to show all the values from a value list as radio buttons. The visitor will be able to select one value from the value list. Check boxes can be created with the same code by changing the type from radio to checkbox.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-Add" value="">
  <input type="hidden" name="-Database" value="Contacts.fp5">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-KeyField" value="ID">

[Inline: -Database='Contacts.fp5', -Layout='People', -Show]
  [Value_List: 'Title']
    <input type="radio" name="Title" value="[Value_ListItem]"> [Value_ListItem]
  [/Value_List]
[/Inline]

  <p><input type="submit" name="-Add" value="Add Record">
</form>
```

To display only selected values from a value list:

The following examples show how to display the selected values from a value list for the current record. The record for John Doe is found within the database and the selected value for the Title field, Mr. is displayed.

- The -Selected keyword in the [Value_ListItem] tag ensures that only selected value list items are shown. The following example uses a conditional to check whether [Value_ListItem: -Selected] is empty.

```
[Inline: -Database='Contacts.fp5', -Layout='People', -KeyValue=126, -Search]
  [Value_List: 'Title']
    [If: (Value_ListItem: -Selected) != '']
      <br>[Value_ListItem: -Selected]
    [/If]
  [/Value_List]
[/Inline]

➜ <br>Mr.
```

- The [Selected] tag ensures that only selected value list items are shown. The following example uses a conditional to check whether [Selected] is empty and only shows the [Value_ListItem] if it is not.

```
[Inline: -Database='Contacts.fp5', -Layout='People', -KeyValue=126, -Search]
  [Value_List: 'Title']
    [If: (Selected) != '']
      <br>[Value_ListItem]
    [/If]
  [/Value_List]
[/Inline]

➜ <br>Mr.
```

- The [Field] tag can also be used simply to display the current value for a field without reference to the value list.

```
<br>[Field: 'Title']

➜ <br>Mr.
```

To display an HTML popup menu in an -Update form with selected value list values:

- The following example shows how to format an HTML <select> ... </select> select list to show all the values from a value list with the selected values highlighted. The [Selected] tag returns Selected if the current value list item is selected in the database or nothing otherwise. This code will usually be used in an HTML form that performs an -Update action to allow the visitor to see what values are selected in the database currently and make different choices for the updated record.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-Update" value="">
  <input type="hidden" name="-Database" value="Contacts.fp5">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-KeyField" value="ID">
  <input type="hidden" name="-KeyValue" value="127">
```

```
[Inline: -Database='Contacts.fp5', -Layout='People', -KeyValue=126, -Search]
  <select name="Title" multiple size="4">
    [Value_List: 'Title']
      <option value="[Value_ListItem]" [Selected]>[Value_ListItem]</option>
    [/Value_List]
  </select>
[/Inline]

  <p><input type="submit" name="-Update" value="Update Record">
</form>
```

- The [Option] tag automatically inserts Selected parameters as needed to ensure that the proper options are selected in the HTML select list. The example below generates exactly the same HTML as the example above.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-Update" value="">
  <input type="hidden" name="-Database" value="Contacts.fp5">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-KeyField" value="ID">
  <input type="hidden" name="-KeyValue" value="127">

[Inline: -Database='Contacts.fp5', -Layout='People', -KeyValue=126, -Search]
  <select name="Title" multiple size="4">
    [Option: 'Title']
  </select>
[/Inline]

  <p><input type="submit" name="-Update" value="Update Record">
</form>
```

To display HTML check boxes with selected value list values:

The following example shows how to format a set of HTML <input> tags to show all the values from a value list as check boxes with the selected check boxes checked. The [Checked] tag returns Checked if the current value list item is selected in the database or nothing otherwise. Radio buttons can be created with the same code by changing the type from checkbox to radio.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-Update" value="">
  <input type="hidden" name="-Database" value="Contacts.fp5">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-KeyField" value="ID">
  <input type="hidden" name="-KeyValue" value="127">
```

```
[Inline: -Database='Contacts.fp5', -Layout='People', -KeyValue=126, -Search]
  [Value_List: 'Title']
    <input type="checkbox" name="Title" value="[Value_ListItem]" [Checked]>
      [Value_ListItem]
  [/Value_List]
[/Inline]

  <p><input type="submit" name="-Update" value="Update Record">
</form>
```

Container Fields

Lasso Professional 7.1 includes a new tag [Database_FMContainer] that allows the raw contents of a FileMaker container field to be returned. This tag works with either FileMaker Pro data sources of FileMaker Server Advanced data sources.

Note: The [Database_FMContainer] tag does not rely on Classic Lasso being enabled. This functionality offers a replacement for the deprecated [Image_URL] and [IMG] tags when Classic Lasso is disabled.

Table 4: Container Field Tags

Tag	Description
[Database_FMContainer]	Returns the raw data contained in a FileMaker container field. Requires one parameter which is the name of the field.

The [Database_FMContainer] tag functions differently depending on whether FileMaker Pro or FileMaker Server Advanced data sources are being accessed.

- **FileMaker Pro** – Only image data can be fetched from container fields. An optional -Type parameter can specify GIF or JPEG along with additional quality arguments the Web Companion supports.
- **FileMaker Server Advanced** – Any type of data can be fetched from a container field. The tag automatically handles any data type that can be stored in FileMaker.

The [Database_FMContainer] tag always returns a byte stream. The results of this tag will be most typically sent to the current site visitor using [File_Serve].

To retrieve data from a FileMaker container field:

Use the [Database_FMContainer] tag. In the following example the data in the Image container field is retrieved and stored in a variable ContainerData. See the following example for a demonstration of how to serve this data as an image to the site visitor.

```
[Inline: -Database='Contacts',
     -Layout='People',
     'First_Name'='John',
     'Last_Name'='Doe',
      -Search]
  [Records]
    [Variable: 'ContainerData' = (Database_FMContainer: 'Image')]
    …
  [/Records]
[/Inline]
```

To serve an image from a FileMaker container field:

Pass the value of the [Database_FMContainer] field to the [File_Serve] tag. In the following example a single image is fetched from a database based on the value of the action parameter ID. The contents of the Image field is interpreted as a JPEG and passed to [File_Serve]. To the site visitor this file will serve a file named FileMakerImage.jpg.

```
[Inline: -Database='Contacts.fp5',
     -Layout='People',
     -KeyValue=(Action_Param: 'ID')
      -Search]
  [File_Serve:
       (Database_FMContainer: 'Image'),
       -Type='image/jpeg',
       -File='FileMakerImage.jpg']
[/Inline]
```

Note: The [File_Serve] tag replaces the current output of the page with the image and performs an [Abort]. The code above represents the complete content of a Lasso page.

The code above could be saved into a Lasso page called Image.Lasso. This page would then be referenced within an HTML <img> tag as follows.

```
<img src="Image.Lasso?ID=[KeyField_Value]" />
```

For example, an image from each record in a database could be displayed as follows:

```
[Inline: -Database='Contacts',
    -Layout='People',
    'First_Name'='John',
    'Last_Name'='Doe',
     -Search]
  [Records]
    <p>[Field: 'First_Name'] [Field: 'Last_Name']
    <br /><img src="Image.Lasso?ID=[KeyField_Value]" /></p>
  [/Records]
[/Inline]
```

The result will be the first and last name of each person in the Contacts database followed by the stored picture on the next line.

FileMaker Scripts

LDML includes command tags which allow scripts in FileMaker databases to be executed. Scripts are usually executed in concert with a database action. They can be performed before the database action, after the database action but before the results are sorted, or just before the results are returned to Lasso. The command tags for executing FileMaker scripts are described in *Table 7: FileMaker Scripts Tags*.

FileMaker Tip: It is best to limit the use of FileMaker scripts. Most functionality of FileMaker scripts can be achieved in LDML with better performance especially on a busy Web server.

Table 5: FileMaker Scripts Tags

Tag	Description
-FMScript	Specifies a script to be processed after the current database action has been performed. Requires a single parameter which names a FileMaker script. Synonym is -FMScriptPost.
-FMScriptPre	Specifies a script to be processed before the current database action has been performed. Requires a single parameter which names a FileMaker script.
-FMScriptPreSort	Specifies a script to be processed after the current database action, but before the results are sorted. Requires a single parameter which names a FileMaker script.

Conditions for executing a FileMaker script:

1 The script must be defined in the database referenced by the action in which the -FMScript... tag is called.

2 The current user must have permission to execute scripts. See the Group section in the *Setting Up Security* chapter of the Lasso Professional 8 Setup Guide for more information.

3 The found set should not be empty after performing a FileMaker script. Scripts should always ensure that they return a non-empty found set after they execute.

4 All database action on the FileMaker machine must wait until the script finishes. Scripts should be as fast and efficient as possible.

To execute a FileMaker script within [Inline] ... [/Inline] tags:

The following example shows a FileMaker script named Filter_People being called after a -FindAll action is performed within a FileMaker database Contacts.fp5. The script removes certain records from the found set and returns the results.

```
[Inline: -Database='Contacts.fp5',
    -Layout='People',
    -FMScript='Filter_People',
    -FindAll]
  ...
[/Inline]
```

The results of the [Inline] ... [/Inline] tags will be the result of the script Filter_People. The record set and its order can be completely determined by the script.

To execute a FileMaker script within an HTML form:

The following example shows a FileMaker script named Clean_Up being performed before a -FindAll action is performed within Contacts.fp5. The script deletes invalid records so that the found set will only contain valid records after the -FindAll is performed. The script is performed before the database action since it is called with -FMScriptPre.

```
<form action="response.lasso" method="POST">
  <input type="hidden" name="-FindAll">
  <input type="hidden" name="-Database" value="Contacts.fp5">
  <input type="hidden" name="-Layout" value="People">
  <input type="hidden" name="-FMScriptPre" value="Clean_Up">

  <br><input type="submit" name="-FindAll" value="Find All">
</form>
```

The results of the script include all valid records that were not deleted by the Clean_Up script.

To execute a FileMaker script within a URL:

The following example shows a script named Update_Priority which is performed after the -FindAll database action, but before the results are sorted. The Update_Priority script could update a field Priority, based on the records from the current found set, which the sort depends on. The script is called using the -FMScriptPreSort tag.

```
<a href="response.lasso?-Database=Contacts.fp5&
    -Layout=People&
    -FMScriptPreSort=Update_Prioirty&
    -SortOrder=Descending&
    -SortField=Priority&
    -FindAll">
  Find All and Sort by Priority
</a>
```

The results of this URL, when it is selected, will be all records from the databases, sorted in descending order according to the value of the Priority field after it has been updated by the Update_Priority script.

```
➜ <img src="Action.Lasso?-Image&-Database=Contacts.fp5&-KeyValue=126&Photo">
```

Note: Additional parameters can be specified within the HTML <img> tag in order to specify the width and height of the returned image. The image will be scaled to the desired size. See the next section for details.

13

Chapter 13
JDBC Data Sources

This chapter documents the usage of LDML 8 with JDBC data sources.

- *Overview* introduces JDBC data source support in Lasso Professional 8.
- *Using JDBC Data Sources* describes using JDBC data sources with Lasso Professional 8.
- *JDBC Schema Tags* describes using LDML tags to return schema values from JDBC data sources that support schema ownership.

Overview

Native support for JDBC data sources is included in Lasso Professional 8 in addition to native support for FileMaker Pro and MySQL data sources. If a JDBC driver is available for a data source, it can be installed to Lasso Professional 8, allowing Lasso to instantly communicate with that data source. This feature allows Lasso Professional 8 to communicate with over 150 JDBC-compliant data sources, including Sybase, DB2, Frontbase, Openbase, Interbase, and Microsoft SQL Server 2000. For more information on JDBC connectivity and availability for a particular data source, see the data source documentation or contact the data source manufacturer.

Lasso Professional 8 functions as its own JDBC driver manager, and all JDBC drivers must be installed directly to Lasso Professional 8. Instructions on how to set up a JDBC data source for use with Lasso Professional are documented in the *Setting Up Data Sources* chapter in the Lasso Professional 8 Setup Guide.

Using JDBC Data Sources

Data source operations outlined in the *Database Interaction Fundamentals*, *Searching and Displaying Data*, and *Adding and Updating Records* chapters are supported with JDBC data sources. Because JDBC is a standardized API for connecting to tabular data sources, there are few unique tags in LDML 8 that are specific to JDBC data sources or invoke special functions specific to any JDBC data source. The only JDBC-specific LDML tags are for JDBC data sources that support schema ownership (e.g. Frontbase, Sybase), and are described in the *JDBC Schema Tags* section of this chapter.

All LDML tags documented as unique to MySQL data sources in the *MySQL Data Sources* chapter or FileMaker Pro data sources in the *FileMaker Pro Data Sources* chapter are not supported for use with JDBC data sources.

Certification Note: OmniPilot Software has tested and certified Microsoft SQL Server 2000 with Microsoft SQL Server 2000 Driver for JDBC for use with Lasso Professional 8 via JDBC. Other JDBC-compliant data sources may be used with Lasso Professional 8, but all features cannot be guaranteed to work by OmniPilot Software. See http://support.omnipilot.com for Support Central articles on connectivity with selected data sources.

Tips for Using JDBC Data Sources

The following is a list of tips to following when writing LDML for use with JDBC data sources. These tips illustrate specific concepts and behaviors to keep in mind when coding, and these tips are most similar to those for MySQL data sources (as opposed to FileMaker Pro data sources).

- Always specify a primary key field using the -KeyField command tag in -Search, -Add, and -FindAll actions. This will ensure that the [KeyField_Value] tag will always return a value.
- Use -KeyField and -KeyValue to reference a particular record for updates, duplicates, or deletes.
- Fields may truncate any data beyond the length they are set up to store. Ensure that all fields in JDBC databases have sufficiently long fields for the values that need to be stored in them.
- Use -ReturnField command tags to reduce the number of fields which are returned from a -Search action. Returning only the fields that need to be used for further processing or shown to the site visitor reduces the amount of data that needs to travel between Lasso Service and the JDBC data source.

- When an -Add or -Update action is performed on a JDBC database, the data from the added or updated record is returned inside the [Inline] ... [/Inline] tags or alternately to the Classic Lasso response page. If the -ReturnField parameter is used, then only those fields specified should be returned from an -Add or -Update action. Setting -MaxRecords=0 can be used as an indication that no record should be returned.
- The -SQL command tag can be allowed or disallowed at the host level for users in Lasso Administration. Once the -SQL command tag is allowed for a user, that user may access any database within the allowed host inside of a SQL statement. For that reason, only trusted users should be allowed to issue SQL queries using the -SQL command tag. For more information, see the *Setting Up Security* chapter in the Lasso Professional 8 Setup Guide.
- SQL statements which are generated using visitor-defined data should be screened carefully for unwanted commands such as DROP or GRANT. See the *Setting Up Data Sources* chapter of the Lasso Professional 8 Setup Guide for more information.
- Always quote any inputs from site visitors that are incorporated into SQL statements. For example, the following SQL SELECT statement includes quotes around the [Action_Param] value. The quotes are escaped \' so they will be embedded within the string rather than ending the string literal. The semi-colon at the end of the statement is optional unless multiple statements are issued.

```
[Variable: 'SQL_Statement'='SELECT * FROM Contacts.People WHERE ' +
  'First_Name LIKE \'' + (Action_Param: 'First_Name') + '\';']
```

  If [Action_Param] returns John for First_Name then the SQL statement generated by this code would appear as follows.

```
SELECT * FROM Contacts.People WHERE First_Name LIKE 'John';
```

- Lasso Professional 8 uses connection pooling when connecting to data sources via JDBC, and the JDBC connections will remain open during the time that Lasso Professional 8 is running.
- Check for OmniPilot Support Central articles at http://support.omnipilot.com for documented issues with using specific JDBC data sources.

JDBC Schema Tags

LDML 8 includes tags that return the user schemas available in a JDBC data source host for the current Lasso Service connection. These tags can only be used with data sources that use named schema ownership (e.g. Frontbase,

Sybase), and complement the other LDML schema and database tags described in the *Database Interaction Fundamentals* chapter.

Note: For information on whether or not your JDBC data source supports named schema ownership, refer to the data source documentation.

Table 1: JDBC Schema Tags

Tag	Description
-Schema	Allows a schema name to be passed as part of an [Inline] ... [/Inline] data source action. The schema name passed here overrides the default schema set for the JDBC data source host in Lasso Administration.
[Schema_Name]	Returns the name of the current schema in use in an [Inline] ... [/Inline] data source action.
[Database_SchemaNames]	Repeats for every schema name in a JDBC data source host available to Lasso. Requires the name of a database in the JDBC data source host as a parameter.
[Database_SchemaNameItem]	Returns the name of the current schema name when used inside [Database_SchemaNames] ... [/Database_SchemaNames] tags.

To reference a schema name in an inline database action:

Use the -Schema command tag to pass the name of the data source schema that should be used for the database action.

```
[Inline: -Show, -Schema='SchemaName', -Database='DBName', -Table='TBName']
  [Schema_Name]
[/Inline]
```

➜ SchemaName

To list all schema names in a JDBC data source:

Use the [Database_SchemaNames] ... [/Database_SchemaNames] tags to list all databases available in a JDBC data source host. The [Database_SchemaNameItem] tag returns the value of each schema name.

```
[Database_SchemaNames:'DBName']
  [Database_SchemaNameItem]
[/Database_SchemaNames]
```

➜ SchemaName
SchemaName2

III

Section III
Programming

This section documents the symbols, tags, expressions, and data types which allow programming logic to be specified within LDML format files. This section contains the following chapters.

- *Chapter 14: Programming Fundamentals* introduces basic concepts of LDML programming such as how to output results, how to store and retrieve variables, and how to interact with HTML forms and URLs.
- *Chapter 15: Variables* introduces concept of variables including global variables, local variables, and page variables.
- *Chapter 16: Conditional Logic* introduces the [If], [Loop], and [While] tags and demonstrates how they can be used for flow control.
- *Chapter 17: Encoding* explains how strings are encoded in Lasso for output to many different languages and the tags and keywords that can be used to control that output.
- *Chapter 18: Sessions* explains how to create server-side variables that maintain their value from page to page while a visitor traverses a Web site.
- *Chapter 19: Error Control* introduces Lasso's error reporting mechanism and explains how custom error tags can be created and what tags can be used to handle errors which occur while processing a format file.

14

Chapter 14
Programming Fundamentals

This chapter introduces the basic concepts of programming using LDML. It is important to understand these concepts before reading the chapters that follow.

- *Overview* explains how to use pages written in LDML and how to deal with errors.
- *Logic vs. Presentation* describes strategies for coding blocks of programming logic code.
- *Data Output* describes strategies for outputting calculation results in HTML or XML.
- *Variables* explains the theory behind variables and how to store and retrieve values.
- *Includes* describes how to use the [Include] and [Library] tags.
- *Data Types* explains how to recognize different data types, how to cast between data types, and casting rules.
- *Symbols* is an introduction to symbols and expressions including rules for grouping, precedence, and auto casting.
- *Member Tags* explains how to call member tags and how they differ from process and substitution tags.
- *Forms and URLs* explains how to pass data between pages using HTML forms and URLs and introduces form parameters and tokens.

Overview

LDML is a tag-based scripting language that has all the features of an advanced programming language. LDML has support for data types, object-oriented member tags, mathematical symbols, string symbols, complex nested expressions, logical flow control, threads, and custom tags which can extend Lasso's built-in functions and procedures.

Using Format Files

Format files which contain LDML must be processed by Lasso in order for the embedded tags to be interpreted. The Open... command in a Web browser should not be used to view Lasso format files. Instead, format files should be uploaded to a Web server and loaded with an appropriate URL. For example, a file named default.lasso in the root of the Web serving folder might be loaded using the following URL.

```
http://www.example.com/default.lasso
```

Simple sequences of tags and LassoScripts can be placed in a text file and then called through the Web browser in order to test LDML programming concepts without the overhead of HTML formatting tags.

Reporting Errors

If there are any LDML syntax errors in a format file which is processed by Lasso, then all processing will stop and an error message will be displayed. Depending on the current error reporting level, the error message will provide the location of the error and a description of what syntax caused the error. All errors must be corrected before the page can be fully processed.

It is recommended that the error reporting level for the server be set to Minimal or None and adjusted to High on a per-page basis using the [Lasso_ErrorReporting] tag when a site is being actively developed. See the *Error Controls* chapter for details about setting the error reporting level and customizing the built-in error page.

Figure 1: Error Page

An error occurred while processing your request.

Error Information	
Error Message:	The file include.inc was not found.
Error Code:	-9984

Note: All valid LDML code above the syntax error will be processed each time the page is loaded. If database actions are being performed, they may be performed each time a page is loaded as long as they are above the point in the page where the error occurs.

Logic vs. Presentation

LDML code can be structured in many ways in order to adapt itself to different coding styles. Some methods involve the tight integration of programming logic (LDML) with page presentation (HTML, XML, and graphics). Other methods involve abstracting the programming logic from the page presentation. LDML offers maximum flexibility for you to determine how you want to structure your pages.

It is often desirable to separate programming logic from page presentation so that different people can work on different aspects of a Web site. For example, an LDML developer can concentrate on creating LassoScripts and blocks of LDML code which define the programming logic of a site. Meanwhile, a Web designer can concentrate on the visual aspects of the Web site with only minimal knowledge of how to integrate LDML into the page presentation so that data is inserted and formatted correctly.

It is also at times desirable for all of your programming to fit tightly within the page presentation. Because LDML is an HTML-like tag language, it is easy to embed LDML within HTML, in effect enhancing static HTML to become dynamic HTML.

The following examples show how to use LDML within HTML as well as how to use LDML abstracted from HTML.

Examples of LDML embedded in HTML:

- LDML tags can be used within HTML markup to insert data from databases, the results of calculations, or LDML commands into otherwise static HTML. The following example inserts the LDML [Image_URL] tag into an HTML <img> tag in order to auto-generate a URL to an image stored in a database.

```
<img src="[Image_URL: 'ImageField']">
```

- Container tags can be used to hide or show portions of a page. The following example hides an HTML <h2> header unless the variable ShowTitle equals True.

```
[If: (Variable: 'ShowTitle') == True]
  <h2>Page Title</h2>
[/If]
```

- Container tags can be used to repeat a portion of a page to present data from many database records or to construct complex HTML tables. The following example shows the fields First_Name and Last_Name from a database search each in their own row of a constructed table. See the *Database Interaction Fundamentals* chapter for more information about [Inline] ... [/Inline] tags.

```
[Inline: -Database='Contacts', -Table='People', -KeyField='ID', -FindAll]
  <table>
    [Records]
      <tr>
        <td>[Field: 'First_Name'] [Field:'Last_Name']</td>
      </tr>
    [/Records]
  </table>
[/Inline]
```

Examples of LDML abstracted from HTML:

- LassoScripts can be used to collect programming logic into a block at the top of a format file. Code in the LassoScript can be formatted and commented separate from the HTML in a format file. Separating the programming logic from the page presentation tags allows for easier debugging and customization of format files. The following example shows an [Inline] specified in a LassoScript with an -InlineName keyword set so the results can be retrieved in the presentation portion of the format file. See the *LDML Syntax* chapter for more information.

```
<?LassoScript
  // This inline finds all records in Contacts.
  // The results are fetched using [Records: -InlineName='Results'] ... [/Records]
  Inline: -InlineName='Results', -Database='Contacts',-Table='People',-FindAll;
  /Inline;
?>
```

- The [Include] tag can be used to include format files that contain portions of the final output. In the following example, the format file shown consists of the standard HTML tags with a pair of [Include] tags that insert all of the programming logic from a file named library.lasso and the data presentation code from a file named presentation.lasso. See the *Files and Logging* chapter for more information about using [Include] tags.

```
<html>
  <head>
    <title>Lasso FormatFile</title>
    [Include: 'library.lasso']
  </head>
```

```
  <body>
    [Include: 'presentation.lasso']
  </body>
</html>
```

Data Output

The final output of most Lasso format files is an HTML page, XML page, or WML page which will be viewed by a Web site visitor in a client browser. This section describes how the results of expressions can be output and how the output of substitution tags can be controlled.

See also the *Encoding* chapter for more information about using encoding keywords.

Table 1: Output Tags

Tag	Description
[Output]	Outputs the result of a calculation or sub-tag using default encoding
[Output_None]	Hides a portion of page from being output, but processes the LDML tags within.
[HTML_Comment]	Surrounds a portion of a page with HTML comment markers, but processes the LDML tags within.

Outputting Values

Substitution tags and member tags output values to the format file which is currently being processed in place. Their values are output whether they are contained within LassoScripts or appear intermixed with HTML tags.

The [Output] tag is a substitution tag which can be used to apply the default encoding to the value of any LDML expression, member tag, or sub-tag.

Example of using the [Output] tag:

The [Output] tag allows encoding keywords to be used on the results of string expressions. The following LassoScript shows the use of the [Output] tag to return the result of a string expression with the encoding keyword -EncodeNone applied so the HTML tags are displayed properly on the page.

```
<?LassoScript
  Output: '<b>' + 'Bold Text' + '</b>', -EncodeNone;
?>

➜ <b>Bold Text</b>
```

Suppressing Output

Sometimes it is desirable to have LDML tags processed in a format file, but not to show the results in the page which is returned to the Web site visitor. The [Output_None] ... [/Output_None] tag can be used to accomplish this purpose. Any LDML tags contained within the container tag will be processed, but the results will not be returned to the Web site visitor.

The following examples use page specific variables in a block of code that will not be output to the user.

```
[Output_None]
  This text will not be returned to the site visitor.
  However, the following tags will be processed.
  [Variable: 'Page Title'='Lasso Format File']
  [Variable: 'Page Error'='None']
[/Output_None]
```

This same example could be written as a LassoScript as follows. The LassoScript will return no value to the page on which it is placed, but any tags within the LassoScript will be processed.

```
<?LassoScript
  Output_None;
    // This LassoScript will return no value.
    // However, the following tags will be processed.
    Variable: 'Page Title'='Lasso Format File';
    Variable: 'Page Error'='None';
  /Output_None;
?>
```

Another way to suppress output is to surround a portion of a page in [HTML_Comment] ... [/HTML_Comment] tags. These tags will become an HTML comment container <?-- ... --> when the page is processed. Any results of the tags inside the container tags will not be shown to the Web site visitor, but will be available if they view the source of the page. This can be useful for providing debugging information which won't affect the overall layout of a Web page. In the following example, the values of several variables are shown in an HTML comment.

```
[HTML_Comment]
  This text will be available in the source of the completed Web page.
  Page Title: [Variable: 'Page Title']
  Page Error: [Variable: 'Page Error']
[/HTML_Comment]
```

```
✜✟✍✍
  This text will be available in the source of the completed Web page.
  Page Title: Lasso Format File
  Page Error: None
?>
```

Variables

Variables are named locations where values can be stored and later retrieved. The concepts of setting and retrieving variables and performing calculations on variables are essential to understanding how to work with LDML's data types and tags.

A variable is created and set using the [Variable] tag. The following tag sets a variable named VariableName to the literal string value VariableValue.

```
[Variable: 'VariableName'='VariableValue']
```

A variable is also retrieved using the [Variable] tag. This time, the tag is simply passed the name of the variable to be retrieved. The following tag retrieves the variable named VariableName returning the literal string value VariableValue.

```
[Variable: 'VariableName'] ➜ VariableValue
```

The following LassoScript sets a variable and then retrieves the value. The result of the LassoScript is the value VariableValue.

```
<?LassoScript
  Variable: 'VariableName'='VariableValue';
  Variable: 'VariableName';
?>

➜ VariableValue
```

Creating Variables

There is only one way to create a variable, using the [Variable] tag with a name/value parameter. All variables should be created and set to a default value before they are used.

Examples of creating variables:

- An empty variable can be created by setting the variable to ".

  ```
  [Variable: 'VariableName'='']
  ```

- A variable can be created and set to the value of a string literal.

  ```
  [Variable: 'VariableName'='String Literal']
  ```

- A variable can be created and set to the value of an integer or decimal literal.

```
[Variable: 'VariableName'=123.456]
```

- A variable can be created and set to the value of any substitution tag such as a field value.

```
[Variable: 'VariableName'=(Field: 'Field_Name')]
```

Multiple variables can be created in a single [Variable] tag by listing the name/value parameters defining the variables separated by commas. The following tag defines three variables named x, y, and z.

```
[Variable: 'x'=100, 'y'=324, 'z'=1098]
```

Variable names can be any string literal and case is unimportant. For best results, variables names should start with an alphabetic character, should not contain any punctuation except for underscores and should not contain any white space except for spaces (no returns or tabs). Variable names should be descriptive of what value the variable is expected to contain.

Note: Variables cannot have their value retrieved in the same [Variable] tag they are defined. [Variable: 'x'=10, 'y'=(variable:'x')] is not valid.

Returning Variable Values

The most recent value of a variable can be returned using the [Variable] tag. For example, the following LassoScript creates a variable named VariableName, then retrieves the value of the variable using the [Variable] tag. The result is Variable Value.

```
<?LassoScript
  Variable: 'VariableName'='Variable Value';
  Variable: 'VariableName';
?>
```

➜ Variable Value

Variable values can also be retrieved using the $ symbol. The following LassoScript creates a variable named VariableName, then retrieves the value of the variable using the $ symbol. The result is Variable Value.

```
<?LassoScript
  Variable: 'VariableName'='Variable Value';
  $VariableName;
?>
```

➜ Variable Value

Setting Variables

Once a variable has been created, it can be set to different values as many times as is needed. The easiest way to set a variable is to use the [Variable] tag again just as it was used when the variable was created.

```
[Variable: 'VariableName'='New Value']
```

Variables can also be set using the expression $VariableName='NewValue'. This expression should only be used within LassoScripts so that it is not confused with a name/value parameter. This expression can be used to set a variable, but cannot be used to create a variable.

The following LassoScript creates a variable named VariableName, sets it to a value New Value using an expression, then retrieves the value of the variable. The result is New Value.

```
<?LassoScript
  Variable: 'VariableName'='';
  $VariableName='New Value';
  $VariableName;
?>
```

➜ New Value

Includes

LDML allows format files to be included within the current format file. This can be very useful for setting up site-wide navigation elements (e.g. page headers and footers), separating the graphical elements of a site from the programming elements, and for organizing a project into reusable code components. There are three types of files that can be included with the various include tags depending on how the LDML code and other data in the included file needs to be treated.

- **Format Files** can be included using the [Include] tag. The LDML code within the included format file executes at the location of the [Include] tag as if it were part of the current file. Any HTML code or text within the format file is inserted into the current format file.

  ```
  [Include: 'format.lasso']
  ```

- **Text or Binary Data** can be included using the [Include_Raw] tag. No LDML code in the included file is processed and no encoding is performed on the included data.

  ```
  [Include_Raw: 'Picture.gif']
  ```

- **LDML Code** can be included using the [Library] tag. No output is returned from the [Library] tag, but any LDML code within the file is executed.

```
[Library: 'library.lasso']
```

- **Variables** can be set to the contents of a file using the [Include] and [Include_Raw] tags. The [Include] tag inserts the results of processing any LDML code within the file into the variable. The [Include_Raw] tag inserts the raw text or binary data within the file into the variable.

```
[Variable: 'File_Data' = (Include: 'format.lasso')]
[Variable: 'File_Data' = (Include_Raw: 'Picture.gif')]
```

See the *Images and Multimedia* chapter for tips about how to use [Include_...] tags to serve images and multimedia files from Lasso.

Library Files

Library files are format files which are used to modify Lasso's programming environment by defining new tags and data types, setting up global constants, or performing initialization code. Libraries can be included within a format file using the [Library] tag or can be added to the global environment by placing the library file within the LassoStartup folder and then restarting Lasso Service.

Specifying Paths

All included files reference paths relative to the format file which contains the include tag. The path specified to the file is usually the same as the relative or absolute path which would be specified within an HTML anchor tag to reference the same file.

Files in the same folder as the current format file can be included by specifying the name of the file directly. The following tag includes a file named Format.lasso in the same folder as the file this tag is specified within.

```
[Include: 'Format.lasso']
```

Files in sub-folders within the same folder as the current format file can be included by specifying the relative path to the file which is to be included. The following tag includes a library file named Library.lasso within a folder named Includes that is in the same folder as the file this tag is specified within.

```
[Library: 'Includes/Library.lasso']
```

Files in other folders within the Web serving folder should be specified using absolute paths from the root of the Web serving folder. The ../ construct can be used to navigate up through the hierarchy of folders. The following tag includes an image file called Picture.gif from the Images folder contained in the root of the Web serving folder.

```
[Include_Raw: '/Images/Picture.gif']
```

File Suffixes

Any file which is included by Lasso including format files, library files, and response files must have an authorized file suffix within Lasso Administration. See the *Setting Site Preferences* chapter of the Lasso Professional 8 Setup Guide for more information about how to authorize file suffixes.

By default the following suffixes are authorized within Lasso Administration. Any of these files suffixes can be used for included files. The .inc file suffix is often used to make clear the role of format files which are intended to be included.

.htm	.html
.inc	.incl
.las	.Lasso
.LassoApp	.text
.txt	

Error Controls

Includes suppress many errors from propagating out to the including page. If a syntax error occurs in an included file then the [Include] tag will return the reported error to the site visitor. If a logical error occurs in an included file then the [Include] tag will return the contents of the error page with the error reported. Techniques for debugging included files are listed on the following pages.

Table 2: Include Tags

Tag	Description
[Include]	Inserts the specified format file into the current format file. Any LDML code in the included format file is executed. Accepts a single parameter, the path and name of the format file to be included.
[Include_Raw]	Inserts the specified file into the current format file. No processing or encoding is performed on the included file. Accepts a single parameter, the path and name of the file to be included.
[Library]	Executes any LDML code in the specified format file, but, inserts no result into the current format file. Accepts a single parameter, the path and name of the format file to be executed.

Note: See the *HTTP/HTML Content and Controls* chapter for documentation of the [Inlude_URL] tag. Lasso Professional 8 also supports the [Include_CGI] tag from Lasso Web Data Engine 3.x, but its use has been deprecated. The [Include_CGI] tag may not be supported in a future version of Lasso.

To include a format file:

Use the include file with the path to the format file which is to be included. The included format file will be processed and the results will be inserted into the current format file as if the code had been specified within the current file at the location of the [Include] tag. The following example shows how to include a file named format.lasso which is contained in the same folder as the current format file.

```
[Include: 'format.lasso']
```

To include a library file:

Library files which contain custom tag definitions or LDML code that does not return any output can be included using the [Library] tag. The LDML code within the library file will be executed, but no result will be returned to the current format file. The following example shows how to include a library file named library.lasso which is contained in the same folder as the current format file.

```
[Library: 'library.lasso']
```

To debug an included file:

The include tags do return errors that occur in the included file, but it can be difficult to debug problems in included files. The errors from an included file can sometimes be more easily seen by loading the

file directly within a Web browser. This will reveal any syntax errors within the included file and ensure that all the code in the included file performs properly. The following URL references a format file named format.lasso inside an Includes folder.

```
http://www.example.com/Includes/format.lasso
```

Note: Some include files rely on variables from the format file that includes them to operate properly. These include files cannot be debugged by simply loading them in a Web browser.

To debug an included library file:

Since library files do not ordinarily return any output to the current format file they can be difficult to debug.

To debug an included library file, insert debugging messages within the code of the library file. Ordinarily, these messages will never be seen since the [Library] tag does not return any output. The following example shows how to report the current error.

```
[(Error_CurrentError: -ErrorCode) + ': ' + (Error_CurrentError)]
```

If the [Library] tag which includes the code library is changed to an [Include] tag then the output of error message will be inserted into the current format file. This allows the debugging messages to be seen. Once the file is working successfully, the [Include] can be changed back to a [Library] tag to hide the debugging messages.

To prevent included files from being served directly:

Included files can be named with any file suffix which is authorized within Lasso Administration. If a file suffix is authorized within Lasso Administration, but is set to not be served by the Web server application then files with that file suffix can only be used as include files and can never be served directly. For example, to authorize the .inc file suffix the following steps must be taken.

1. Authorize .inc in Lasso Administration *Setup > Settings > File Extensions*.
2. Using the file suffix controls of your Web server applications, deny the suffix .inc so that files with that suffix cannot be served. This can usually be accomplished with specific file suffix controls or with a Web server realm. Consult the Web server documentation for more information.

Note: If LDML code is placed in an include file that is authorized for processing by Lasso (step **1** above), but is not set in the Web server preferences to always be processed by Lasso or never to be served (step **2** above),

then it may be possible for site visitors to view the unprocessed LDML code by loading the include file directly.

Advanced Methodology

Includes and library files allow LDML format files to be structured in order to create reusable components, separate programming logic from data presentation, and in general to make Web sites easier to maintain. There are many different methods of creating structured Web sites which are beyond the scope of this manual. Please consult the third party resources at the OmniPilot Web site for more information.

Data Types

Every value in Lasso is defined as belonging to a specific data type. Every value stored in a variable belongs to a specific data type. The data type determines what symbols and member tags are available for use with the value.

Table 3: Data Type Tags

Tag	Description
[Null->Type]	Returns the data type of a value.
[String]	Casts a value to data type string.
[Integer]	Casts a value to data type integer.
[Decimal]	Casts a value to data type decimal.
[Boolean]	Casts a value to data type boolean.
[Date]	Casts a value to data type date.
[Duration]	Casts a value to data type duration.
[Array]	Creates an array data type.
[Map]	Creates a map data type.
[Pair]	Creates a pair data type.
[Bytes]	Creates a bytes data type.

Note: Lasso has many more data types than those listed. See the ***Data Types*** section in this manual or the LDML Reference. for complete documentation of all the available data types.

Several data types have already been introduced:

- Strings are sequences of alphanumeric characters. String literals are delimited by single quotes as in 'String Literal'.

- Integers are whole numbers. Integer literals are specified without quotes as in 123 or -987.
- Decimals are numbers which contain a decimal point. Decimal literals are specified without quotes as in 3.1415926 or 24.99.
- Dates are alphanumeric strings that represent a date and/or time. A date must always be cast using the [Date] tag in a recognized format to be used as a date data type (e.g. [Date:'9/29/2002']).
- Durations are alphanumeric strings that represent a length time (not a 24-hour clock time). A duration must always be cast using the [Duration] tag in a recognized format to be used as a duration data type (e.g. [Duration:'168:00:00']).

Variables which are set to literal values of a specific data type are themselves said to be of that data type. Variables containing strings are string variables. Any symbols which operate on literal strings will also operate on string variables.

It is important to keep track of what type of value is stored in each variable so that the values of expressions and member tags can be safely predicted.

Returning the Type of a Variable

The [Null->Type] member tag can be used to return the type of a variable or other value. [Null->Type] is a member tag of the data type null which is a precursor to all other data types. The [Null->...] member tags can be used with values of any data type.

The following example shows the value of [Null->Type] for literals of different data types.

```
'String Value'->Type ➜ string
123->Type ➜ integer
9.999->Type ➜ decimal
```

The following example shows the value of [Null->Type] when it is used on a variable which has been set to a string literal.

```
<?LassoScript
  Variable: 'Value' = 'String Value';
  $Value->Type;
?>
```

➜ string

The [Null->Type] member tag also works on the compound data types: array, map, and pair. The following example shows the value of [Null->Type] when it is used on a variable which has been set to an array literal.

```
<?LassoScript
  Variable: 'Value' = (Array: 'One', 'Two', 'Three', 'Four');
  $Value->Type;
?>
```

➜ array

Casting a Value to a Data Type

Values can be cast from one data type to another in order to ensure that the proper member tags will be available and symbols will work as expected. Each data type defines a tag which has the same name as the data type that can be used to cast a value to that data type.

To cast a value to the string data type:

- Integer and decimal values can be cast to type string using the [String] tag. The value of the string is the same as the value of the integer or decimal value when it is output using the [Variable] tag.

  [String: 999.999] ➜ '999.999'

- Boolean values can be cast to type string using the [String] tag. The value will always either be True or False.

  [String: True] ➜ 'True'

- Arrays, maps, and pairs should not be cast to type string. The value which results is intended for debugging purposes. More information can be found in the *Arrays and Maps* chapter.

To cast a value to the integer data type:

- Decimal values can be cast to type integer using the [Integer] tag. The value of the decimal number will be truncated at the decimal point. For example, casting 999.999 to type integer results in 999 not 1000.

  [Integer: 999.999] ➜ 999

- String values can be cast to type integer using the [Integer] tag. The string must start with a numeric value. For example casting 2String1 to an integer results in 2.

  [Integer: '2001: A Space Odyssey'] ➜ 2001
  [Integer: '2String1'] ➜ 2

- Boolean values can be cast to type integer using the [Integer] tag. The value of the result will be 1 if the boolean was True or 0 if the boolean was False.

  [Integer: True] ➜ 1
  [Integer: False] ➜ 0

- Arrays, maps, and pairs should not be cast to type integer. The value which results will always be 0.

To cast a value to the decimal data type:

- Integer values can be cast to type decimal using the [Decimal] tag. The value of the integer number will simply have a decimal point added. For example, casting 123 to type integer results in 123.000000.

  [Decimal: 123] ➜ 123.000000

- String values can be cast to type decimal using the [Decimal] tag. The string must start with a numeric value. For example casting 2.5String1 to a decimal results in 2.500000. The 1 at the end of the string is ignored.

  [Decimal: '2001: A Space Odyssey'] ➜ 2001.000000
  [Decimal: '2.5String1'] ➜ 2.500000

- Boolean values can be cast to type decimal using the [Decimal] tag. The value of the result will be 1.000000 if the boolean was True or 0.000000 if the boolean was False.

  [Decimal: True] ➜ 1.000000
  [Decimal: False] ➜ 0.000000

- Arrays, maps, and pairs should not be cast to type integer. The value which results will always be 0.000000.

To cast a value to the boolean data type:

- Integer and decimal values can be cast to type boolean using the [Boolean] tag. The value of the boolean will be False if the number is zero or True if the number is non-zero.

  [Boolean: 123] ➜ True
  [Boolean: 0.0] ➜ False

- String values can be cast to type boolean using the [Boolean] tag. The value of the boolean will be False if the string contains just the word false or is empty and True otherwise.

  [Boolean: 'false'] ➜ False
  [Boolean: ''] ➜ False
  [Boolean: 'true'] ➜ True
  [Boolean: 'value'] ➜ True

- Arrays, maps, and pairs should not be cast to type boolean. The value which results will always be False.

To cast a value to the date data type:

- Specially formatted strings may be cast as date data types using the [Date] tag. For a list of date string formats that are automatically recognized as dates, see the *Date and Time Operations* chapter.

  [Date: '9/29/2002'] ➜ 9/29/2002 00:00:00
  [Date: '9/29/2002 12:30:00'] ➜ 9/29/2002 12:30:00
  [Date: '2002-09-29 12:30:00'] ➜ 2002-09-29 12:30:00

- Unrecognized date strings can be cast as date data types using the [Date] tag with the -Format parameter. All eligible date strings must contain numbers, punctuation, and/or allowed words (e.g. February, GMT) in a format that represents a valid date. For a description of how to format a date string, see the *Date and Time Operations* chapter.

  [Date: '9.29.2002', -Format='%m.%d.%Y'] ➜ 9.29.2002
  [Date: '20020929', -Format='%Y%m%d'] ➜ 20020929
  [Date: 'September 29, 2002', -Format='%B %d, %Y'] ➜ September 29, 2002

To cast a value to the duration data type:

- Specially formatted strings as either hours:minutes:seconds or just seconds may be cast as duration data types using the [Duration] tag. The [Duration] tag always returns values in hours:minutes:seconds format. For more information, see the *Date and Time Operations* chapter.

  [Duration: '169:00:00'] ➜ 169:00:00
  [Duration: '00:30:00'] ➜ 00:30:00
  [Duration: '300'] ➜ 00:05:00

To cast a value to type array, map, or pair:

Values cannot be cast to type array, map, or pair. However, an array, map, or pair can be constructed with the simple data type as its initial value. See the *Arrays and Maps* chapter for more information about how to construct these complex data types.

To cast a value to the bytes data type:

For discussion on the bytes data type, see the *Advanced Programming Topics* chapter.

Automatic Casting

Lasso will cast values to a specific data type automatically when they are used in expressions or as parameters for tags which require a particular type of value. Values will be automatically cast in the following situations:

- Values of every data type are cast to string values when they are output to the Web browser.
- Integer values are cast to decimal values when they are used as parameters in expressions with one integer parameter and one decimal parameter.
- Integer and decimal values are cast to string values when they are used as parameters in expressions with one integer or decimal parameter and one string parameter.
- Values of every data type are cast to boolean values when they are used in logical expressions.
- The [Math_...] tags will automatically cast all parameters to integer or decimal values.
- The [String_...] tags will automatically cast all parameters to string values.

Symbols

Symbols allow for powerful calculations to be performed within LDML tags. The symbols which can be used in expressions are discussed in full detail in the chapter devoted to each data type. String expressions and symbols are discussed in the *String Operations* chapter and decimal and integer expressions and symbols are discussed in the *Math Operations* chapter.

Using Symbols

Since symbols only function on values of a specific data type, values need to be cast to that data type explicitly or they will be automatically cast. For best results, explicit casting should be performed so the meaning of the symbols will be clear. Note that spaces should always be specified between a symbol and its parameters.

As explained in the *Automatic Casting* section above, values used as a parameter in an expression will be automatically cast to a string value if any parameter in the expression is a string value. Integer values will be automatically cast to decimal values. Any value used in a logical expression will be automatically cast to a boolean value.

- The following expression returns 1212 since the integer 12 is automatically cast to a string because one parameter is a string.

  ['12' + 12] ➜ 1212

- Similarly, the following expression returns 1212 since the integer 12 is automatically cast to a string because one parameter is a string.

```
[12 + '12'] ➜ 1212
```

- The following expression returns 24 since the string 12 is explicitly cast to an integer.

```
[(Integer: '12') + 12] ➜ 24
```

- The following expression returns 24.000000 since the integer 12 is automatically cast to a decimal value because one parameter is a decimal value.

```
[12 + 12.0] ➜ 24.000000
```

- The following expression returns True since the integer 12 is automatically cast to a boolean value True because it is used in a logical expression.

```
[12 && 12] ➜ True
```

When in doubt, the [String], [Integer], and [Decimal] tags should be used to explicitly cast values so that the proper symbols are used.

Note: Always place spaces between a symbol and its parameters. The - symbol can be mistaken for the start of a command tag, keyword, or keyword/value parameter if it is placed adjacent to the parameter that follows.

Assignment Symbols

Variables can be set to the result of an expression, storing that result for later use. For example, the following variable is set to the result of a simple math expression.

```
[Variable: 'MathResult'=(1 + 2)]
```

Variables can also be set using assignment symbols within LassoScripts. The equal sign = is the simplest assignment symbol. Other assignment symbols can be formed by combining a decimal, integer, or string symbol with the equal sign. For example, += is the additive assignment symbol.

The following LassoScript creates a variable named MathResult, performs a mathematical operation (adding 4) on it using the additive assignment symbol, and returns the final value.

```
<?LassoScript
  Variable: 'MathResult'=0;
  $MathResult += 4;
  $MathResult;
?>
```

➜ 4

The assignment symbol replaces the value of the variable and does not return any output. The assignment expression $MathResult += 4; is equivalent to the expression $MathResult = $MathResult + 4;. Since assignment expressions do not return a value they should only be used within LassoScripts to modify variables.

LassoScripts can use variable results to build very complex operations. For example, the following LassoScript uses several variables to perform a math expression.

```
<?LassoScript
  Variable: 'x'=100, 'y'=4;
  $x = $x / $y;
  $y = $x + $y;
  'x=' + $x + ' y=' + $y;
?>
```

➜ x=25 y=29

Note: If a negative number is used as the right-hand parameter of an assignment symbol it should be surrounded by parentheses.

Member Tags

Member tags are associated with a particular data type and can be used on any value of that data type. The data type of a member tag is represented in the documentation in the member tag name before the member tag symbol ->. For example, the tag [String->Length] can be used with values of data type string, and the tag [Decimal->SetFormat] can be used with values of data type decimal.

Member tags are available for string, decimal, integer, date, array, map, and pair data types, and are discussed in detail in the *String Operations*, *Math Operations*, *Date and Time Operations*, and *Arrays and Maps* chapters.

Using Member Tags

Since member tags only function on values of a specific data type, values need to be cast to that data type explicitly. Member tags will not automatically cast values.

For example, the member tag [String->Length] can be used to return the length of a string value. If [String->Length] is used on a number as in [123->Length] then an error will result:

```
"Length" was not a member of type "integer"
```

Instead, the integer must be cast to a string value explicitly before the member tag can be used. The following example returns the length of the string representing the integer correctly.

```
[(String: 123)->Length] ➜ 3
```

When in doubt, the [String], [Integer], [Decimal], and [Date] tags should be used to explicitly cast values so that the proper member tags are available.

Member Tag Types

Member tags can function like either substitution tags which return a value or like process tags which modify the value which the member tag is called on, but do not return a value.

For example, the member tag [String->Length] functions like a substitution tag and returns the length of the string on which it is called. The following LassoScript stores a string in a variable StringVariable then retrieves its length. The string stored in the variable is left unchanged.

```
<?LassoScript
  Variable: 'StringVariable' = 'A string value';
  $StringVariable->Length;
?>
```

➜ 14

In contrast, the member tag [Decimal->SetFormat] functions like a process tag, altering the way that a decimal variable will be output when it is cast to a string. The following LassoScript shows the normal decimal value output of a variable.

```
<?LassoScript
  Variable: 'DecimalVariable' = 123.456;
  $DecimalVariable;
?>
```

➜ 123.456000

The following LassoScript shows how the output of the decimal value changes when a [Decimal->SetFormat] tag is used on the variable DecimalVariable to truncate its output to two significant digits.

```
<?LassoScript
  Variable: 'DecimalVariable' = 123.456;
  $DecimalVariable->(SetFormat: -Precision=2);
  $DecimalVariable;
?>
```

➜ 123.45

The value stored in the variable DecimalVariable is not changed, but the value which is output is formatted according to the rules set in the [Decimal->SetFormat] tag.

Forms and URLs

This section discusses how to pass information from format file to format file through HTML forms and URLs. Data can also be passed from format file to format file using database actions or sessions. Please see the *Database Interaction Fundamentals* and *Sessions* chapters for more information.

Form Parameters

HTML forms can be used to pass values to an LDML format file. The values are retrieved in the format file using the [Action_Param] tag. Any <input>, <select>, or <textarea> values can be retrieved by name using the [Action_Param] tag except for those which contain LDML command tags.

For example, the following form has two inputs for First_Name and Last_Name and a button that submits the form.

```
<form action="response.lasso" method="POST">
  <p>First Name: <input type="test" name="First_Name" value="">
  <p>Last Name: <input type="test" name="Last_Name" value="">
  <p><input type="submit" name="Submit" value="Submit Value">
</form>
```

In the format file response.lasso—which is loaded when this form is submitted—the following LDML tags will retrieve the values submitted by the site visitor in the form.

```
First Name: [Action_Param: 'First_Name']
Last Name: [Action_Param: 'Last_Name']
```

Even the value of the submit button can be fetched. This can help distinguish between multiple buttons that each have the same name displayed in the Web browser.

```
Button Value: [Action_Param: 'Submit']
```

URL Parameters

URLs can be used to pass values to an LDML format file. The values are retrieved in the format file using the [Action_Param] tag. Any values which are passed as URL parameters can be retrieved by name using the [Action_Param] tag except for those which contain LDML command tags.

For example, the URL in the following anchor tag has two parameters for First_Name and Last_Name.

```
<a href="response.lasso?First_Name=John&Last_Name=Doe">John Doe</a>
```

In the format file response.lasso—which is loaded when this form is submitted—the following LDML tags will retrieve the values submitted by the site visitor on the form.

```
First Name: [Action_Param: 'First_Name']
Last Name: [Action_Param: 'Last_Name']
```

15

Chapter 15
Variables

This chapter introduces the basic concepts of variables in LDML including page variables, global variables, page variables, special variables, and references. It is important to understand these concepts before reading the chapters that follow.

- *Overview* explains how variables work in LDML and how the different variable scopes interact.
- Page Variables describes tags and symbols that can be used to manipulate page variables.
- Global Variables describes tags and symbols that can be used to manipulate global variables which are accessible from any page on a server.
- Local Variables describes tags and symbols that can be used to manipulate local variables within compound expressions or custom tags or data types.
- References describes how multiple variables (or compound data type members) can reference the same data.

Overview

A variable is a named location for storing a value. Variables in LDML are used extensively to store temporary values so they can be manipulated using tags, member tags, or symbols. An LDML variable can store any type of value within Lasso.

Lasso maintains a stack of environments as it processes LDML code. The first environment is created when Lasso starts up and includes global, server-wide variables. Each page has its own environment created when it is parsed which includes normal, page-wide variables. Finally, each custom

tag and data type has its own environment that includes local variables. At any point, tags can be used to examine and modify values in the environment above the current environment.

Variable Scope

Each variable in Lasso exists within a certain scope. When the scope in which a variable was created becomes invalid then all the variables within that scope are deleted. The three possible scopes are as follows:

Page Scope – The most common scope is the page scope which exists from when Lasso starts processing an LDML page until it finishes and returns the results to the client. Most variables are created within the page scope and exist for the duration of the LDML page process.

Global Scope – Lasso maintains a global scope which contains variables that can be accessed by any page that is processed by Lasso. Global variables allow values to be shared between multiple independent page loads. They are used to store some of Lasso's preferences and for cache storage.

Local Scope – Each compound expression and custom tag creates a local scope that exists for the duration of the expression or tag processing. Local variables are only available within that expression or tag call. Note that as additional custom tags are called each one creates its own independent local scope so there may be many local scopes active at one time.

Session Variables – Session variables exist within the page scope, but can be thought of as being defined within a special session scope that is reloaded each time the [Session_Start] tag is called.

Instance Variables – Custom data types can store instance variables. These variables can be thought of as being defined within an instance scope which exists for as long as the data type instance is defined. However, the data type instance itself exists within either the page, global, or local scope.

Non-Variable Values

Lasso also allows many values that are not stored in variables to be manipulated. These values include:

Literals – A literal value is one that is specified directly within an LDML page. Examples include string literals 'My String Value', integers 10, decimal values 35.6, or even arrays (Array: 1, 2, 3) or maps (Map: 'one'=1, 'two'=2).

Tag Values – Many tags return values that can be stored in a variable or otherwise manipulated. Examples include [Date], [Server_IP], [Response_FilePath], and hundreds more.

Field Values – Values from a database are returned using the [Field] or [Record_Arrays] tags.

Action Params – Values from the current HTML form action or URL are returned using the [Action_Param] tag.

Other Values – Other values from the current HTTP request can be returned using [Cookie_Value], [Token_Value], etc.

Page Variables

Page variables are the most common type of variable in LDML They only exist while the current LDML page is executing. Page variables are used to store temporary values in long calculations or to manipulate values for output. The values stored in all page variables are lost at the end of the page unless they are stored in a session.

This section includes an introduction to the tags and symbols that can be used to manipulate page variables. This is followed by sections about creating variables, retrieving variable values, setting variables, checking to see if a variable has been created, and removing a variable.

Table 1: Page Variable Tags

Tag	Description
[Variable]	Creates or sets named variables or returns their values.
[Variable_Defined]	Returns True if a variable is defined.
[Variables]	Returns a map of all page variables.
[Var]	Abbreviation of [Variable].
[Var_Defined]	Abbreviation of [Variable_Defined].
[Var_Remove]	Deletes the named variable.
[Var_Reset]	Resets the specified variable to a new value, detaching any references.
[Vars]	Abbreviation of [Variables].

Table 2: Page Variable Symbols

Symbol	Description
$	Returns the value of a variable.
=	Assigns a value to a variable: $Variable='NewValue'.
:=	Assigns a value to a variable and returns the value.

Creating Variables

Variables are created using the [Variable] tag with a name/value parameter. All variables should be created and set to a default value before they are used. Variables can also be created implicitly if they are referenced within the [Variable] tag. Implicitly created variables are set to Null.

Examples of creating variables:

- An empty variable can be created by setting the variable to ".

```
[Variable: 'VariableName'='']
```

- A variable can be created and set to the value of a string literal.

```
[Variable: 'VariableName'='String Literal']
```

- A variable can be created and set to the value of an integer or decimal literal.

```
[Variable: 'VariableName'=123.456]
```

- A variable can be created and set to the value of any substitution tag such as a field value.

```
[Variable: 'VariableName'=(Field: 'Field_Name')]
```

Multiple variables can be created in a single [Variable] tag by listing the name/value parameters defining the variables separated by commas. The following tag defines three variables named x, y, and z.

```
[Variable: 'x'=100, 'y'=324, 'z'=1098]
```

Variable names can be any string literal and case is unimportant. For best results, variables names should start with an alphabetic character, should not contain any punctuation except for underscores and should not contain any white space except for spaces (no returns or tabs). Variable names should be descriptive of what value the variable is expected to contain.

Note: Variables cannot have their value retrieved in the same [Variable] tag they are defined. [Variable: 'x'=10, 'y'=(variable:'x')] is not valid.

Returning Variable Values

The most recent value of a variable can be returned using the [Variable] tag. For example, the following LassoScript creates a variable named VariableName, then retrieves the value of the variable using the [Variable] tag. The result is Variable Value.

```
<?LassoScript
  Variable: 'VariableName'='Variable Value';
  Variable: 'VariableName';
?>
```

➜ Variable Value

Variable values can also be retrieved using the $ symbol. The following LassoScript creates a variable named VariableName, then retrieves the value of the variable using the $ symbol. The result is Variable Value.

```
<?LassoScript
  Variable: 'VariableName'='Variable Value';
  Encode_HTML: $VariableName;
?>
```

➜ Variable Value

If a variable value is retrieved using the [Variable] tag before it has been defined then the variable is implicitly created with a value of Null. The $ symbol will not implicitly create a variable. Referencing a variable name with $ that has not already been defined will result in a syntax error.

Setting Variables

Once a variable has been created, it can be set to different values as many times as is needed. The easiest way to set a variable is to use the [Variable] tag again just as it was used when the variable was created.

```
[Variable: 'VariableName'='New Value']
```

Variables can also be set using the expression $VariableName='NewValue'. This expression should only be used within LassoScripts so that it is not confused with a name/value parameter. This expression can be used to set a variable, but cannot be used to create a variable.

The following LassoScript creates a variable named VariableName, sets it to a value New Value using an expression, then retrieves the value of the variable. The result is New Value.

```
<?LassoScript
  Variable: 'VariableName'=";
  $VariableName='New Value';
  $VariableName;
?>
```

➜ New Value

The = symbol does not return a value. The variable is set to the new value, but the expression does not return a value. In order to set a variable to a new value and return the value that it was set to (for output or further processing) the := symbol can be used.

The following example is equivalent to the one above, but since the := symbol is used it is not necessary to output the variable value after it has been set.

```
<?LassoScript
  Variable: 'VariableName'=";
  $VariableName := 'New Value';
?>
```

➜ New Value

Resetting Variables

Multiple variables can point to the same underlying value in Lasso through the use of references (which are described more fully at the end of this chapter). When two variables point by reference to the same value, changing one variable changes the value of the other variable as well. A variable can be detached from any references and set to a new value using the [Var_Reset] tag.

```
<?LassoScript
  Var_Reset: 'VariableName'='New Value';
  $VariableName;
?>
```

➜ New Value

Checking to See if a Variable has been Created

The [Variable_Defined] tag can be used to check if a variable has been created and used in the current format file. The following example will return false the first time [Variable_Defined] is called, then set the variable using [Variable] and return True the second time [Variable_Defined] is called.

```
<?LassoScript
  Variable_Defined: 'VariableName';
  Variable: 'VariableName'='VariableValue';
  Variable_Defined: 'VariableName';
?>
```

➜ False True

The [Variable_Defined] tag will return True even if a variable is set to the empty string '' (two single quotes with no space) or to Null. There is no way to delete a variable once it has been created.

Deleting a Variable

Once a variable has been created it will exist until the end of the page even if its value is never retrieved. All variables are automatically deleted once the current LDML page is finished processing. However, it is also possible to delete a variable explicitly using the [Var_Remove] tag. This is not generally necessary, but can be useful in certain specific circumstances.

The following code creates a variable, checks to see if it is defined, then deletes the variable, and checks again. The result is first True then False.

```
<?LassoScript
  Var('MyVariable' = 'Testing');
  Var_Defined('MyVariable');
  Var_Remove('MyVariable');
  Var_Defined('MyVariable');
?>
```

➜ True False

Global Variables

The globals tags allow direct access to global variables from any environment. These are the preferred way of setting and retrieving global values. Globals can also be accessed implicitly from the page and local environments following the rules described in the sections below.

Note: Many global variables are used to set preferences for internal Lasso processes such as the email queue, the session handler, and the scheduler. Global variables which start with an underscore should never be modified.

Table 3: Global Tags

Tag	Description
[Global]	If called with a string parameter, retrieves the value of a global variable. If called with a name/value pair sets the value of a global variable.
[Global_Defined]	Accepts a single string parameter. Returns True if the global variable has been defined or False otherwise.
[Global_Remove]	Removes the specifies variable from the globals.
[Global_Reset]	Resets the specified variable to a new value, detaching any references.
[Globals]	Returns a map of all global variables that are currently defined.

Startup Environment

When code is executed in LassoStartup it is executed in the startup or global environment. Any variables which are set using the [Variable] tag at this level will end up as global variables when pages are executed. Similarly, any tags which are defined at this level will be made available to all pages that are executed on the server.

To set a global variable at startup:

At startup, global variables can be set either using the [Global] tag or using the [Variable] tag. All variables set at this level are implicitly global.

- Use the [Global] tag to set the value of a global variable. The global variable will be available to any page subsequently executed by Lasso. In the following example a variable Administrator_Email is created and set with the value of the administrator's email address.

```
[Global: 'Administrator_Email' = 'administrator@example.com']
```

- Use the [Variable] tag to set the values of global variables from code which is executed in the LassoStarup folder. In the following example a variable Administrator_Email is created and set with the value of the administrator's email address.

```
[Variable: 'Administrator_Email' = 'administrator@example.com']
```

Page Environment

From the page level the values of global variables can be retrieved using the [Global] tag. The $ symbol will return a global variable if no page variable of the same name has been created. Global variables should be set using the [Global] tag. The [Variable] tag cannot be used to set a global variable.

To retrieve the value of a global variable:

- Use the [Global] tag. In the following example the global variable Administrator_Email which is set above is retrieved.

```
[Global: 'Administrator_Email']
```

```
➜ administrator@example.com
```

- If the desired variable has not been overridden by a page variable of the same name then use the [Variable] tag to retrieve the value of the global variable. In the following example the global variable Administrator_Email which is set above is retrieved.

```
[Variable: 'Administrator_Email']
```

```
➜ administrator@example.com
```

To set the value of a global variable:

Either of the two following techniques can be used to set the value of a global variable from an LDML format file. The first method is preferred.

- Use the [Global] tag to set the value of a global variable. The global variable will be immediately available on any page executing by Lasso through the [Global] or [Globals] tags.

```
[Global: 'Administrator_Email' = 'new_administrator@example.com']
<br>Global: [Global: 'Administrator_Email']
```

```
➜ <br>Global: new_administrator@example.com
```

- Set the value of a global variable by reference. In the following example, the variable Administrator_Email has not been overridden on the current page. Using the $ and = symbols the global variable can be changed.

```
$Administrator_Email = 'new_administrator@example.com']
<br>Global: [Global: 'Administrator_Email']
```

```
➜ <br>Global: new_administrator@example.com
```

To override the value of a global variable:

Use the [Variable] tag to set a variable of the same name. The global variable will not be modified, but subsequent uses of the [Variable] tag will return the page variable's value. The [Global] tag can still be used to retrieve the value of the global variable.

In the following example the global variable Administrator_Email is overridden by a page variable of the same name. The values of both the page variable and the global variable are displayed.

```
[Variable: 'Administrator_Email' = 'page_administrator@example.com']
<br>Page: [Variable: 'Administrator_Email']
<br>Global: [Global: 'Administrator_Email']
➜ <br>Page: page_administrator@example.com
<br>Global: administrator@example.com
```

Local Environment

When a custom tag is executing, variables from the global scope can be accessed using the [Global] tag and variables from the page scope can be accessed using the [Variable] tag. The $ symbol will reference a global variable only if a page variable with the same name has not been defined.

Local Variables

Each custom tag and compound expression can create and manipulate its own set of local variables. These variables are separate from the page variables and are deleted when the custom tag returns. Using local variables ensures that the custom tag or compound expression does not alter any variables which other custom tags or the page developer is relying on having a certain value.

Table 4: Local Tags

Tag	Description
[Local]	If called with a string parameter, retrieves the value of a local variable. If called with a name/value pair sets the value of a local variable.
[Local_Defined]	Accepts a single string parameter. Returns True if the local variable has been defined or False otherwise.
[Local_Remove]	Removes the specifies variable from the locals.
[Local_Reset]	Resets the specified variable to a new value, detaching any references.
[Locals]	Returns a map of all local variables that are currently defined.

Table 5: Local Variable Symbols

Symbol	Description
#	Returns the value of a local variable.
=	Assigns a value to a variable: $Variable='NewValue'.
:=	Assigns a value to a variable and returns the value.

For example, many developers will use the variable Temp to store temporary values. If a page developer is using the variable Temp and then calls a custom tag which also sets the variable Temp, then the value of the variable will be different than expected.

The solution is for the custom tag author to use a local variable named Temp. The local variable does not interfere with the page variable of the same name and is automatically deleted when the custom tag returns. In the following example, a custom tag returns the sum of its parameters, storing the calculated value in Temp.

```
<?LassoScript
  Define_Tag: 'Ex_Sum';
    Local: 'Temp'=0;
    Loop: (Params)->Size;
```

```
        Local: 'Temp'=(Local: 'Temp') + (Params)->(Get: Loop_Count);
      /Loop;
      Return: #Temp;
    /Define_Tag;
?>
```

The final reference to the local variable temp is as #Temp. The # symbol works like the $ symbol for page variables, allowing the variable value to be returned using shorthand syntax.

When this tag is called, it does not interfere with the page variable named Temp.

```
[Variable: 'Temp' = 'Important value:']
[Variable: 'Sum' = (Ex_Sum: 1, 2, 3, 4, 5)]
['<br>' + $Temp + ' ' + $Sum + '.']

➜ <br>Important value: 15.
```

Local Variables and Compound Expressions

A compound expression can be used to temporarily create a local scope. This allows local variables to be used without modifying any page values. Once the compound expression completes all the local variables will be deleted. This technique can also be used to avoid creating global variables within code ran at startup.

The following example shows a compound expression with a local variable named SecretTemp. The value of SecretTemp will only be available within the compound expression.

```
<?LassoScript
  {
    Local: 'SecretTemp' = 'MyValue';
    …
  }->Run;
?>
```

References

References in Lasso Professional 8 allow multiple variables to point to the same value or object. When the shared value or object is changed, all variables that reference that value or object change. A reference can be created using the [Reference] tag or the @ reference symbol.

An example will serve to illustrate how references can be used in Lasso. The following LDML code creates two variables and sets them to default values, then outputs those values. Each variable is independent. Changing the value of the one variable will not change the value of the other variable.

```
[Variable: 'Alpha'= 1]
[Variable: 'Beta'= 2]

<br>Alpha: [Variable: 'Alpha']
<br>Beta: [Variable: 'Beta']

➜ <br>Alpha: 1
  <br>Beta: 2
```

However, if we instead define the second variable to be a reference to the first variable then the two variables will share a single value. In the following example the variable Alpha is set to 3 and the variable Beta is set to be a reference to the variable Alpha. When output, both variables return 3.

```
[Variable: 'Alpha'= 3]
[Variable: 'Beta'= (Reference: $Alpha)]

<br>Alpha: [Variable: 'Alpha']
<br>Beta: [Variable: 'Beta']

➜ <br>Alpha: 3
  <br>Beta: 3
```

Now that the two variables are linked, changing either variable will effect a change in both. For example, setting Alpha to 4 will also result in a change to Beta.

```
[Variable: 'Alpha'= 4]

<br>Alpha: [Variable: 'Alpha']
<br>Beta: [Variable: 'Beta']

➜ <br>Alpha: 4
  <br>Beta: 4
```

Similarly, setting Beta to 5 will also result in a change to Alpha.

```
[Variable: 'Beta' = 5]

<br>Alpha: [Variable: 'Alpha']
<br>Beta: [Variable: 'Beta']

➜ <br>Alpha: 5
  <br>Beta: 5
```

A variable can be set to a new value without modifying any of the other variables that might refer to the same value by reference using the [Var_Reset] tag. This tag sets the variable to a new value and detaches it from

any references. Here the value of Beta is reset to 10, detaching it from Alpha which still has a value of 5.

```
[Var_Reset: 'Beta' = 10]

<br>Alpha: [Variable: 'Alpha']
<br>Beta: [Variable: 'Beta']

➜ <br>Alpha: 5
<br>Beta: 10
```

This simple example serves to illustrate the basic principle behind Lasso's references. The remainder of this section will provide demonstrations of how references can be used to reduce the amount of memory that Lasso needs to process complex pages and to increase page processing speed.

It is impossible to have a reference to a reference. Lasso always resolves references back to the original object so if one variable is set as a reference to a second variable, then a third variable is set as reference to the first variable, all three variables end up pointing to the same object. A change to any of the three variables results in the values of all three variables being changed.

References can be detached using the [Null->DetachReference] tag. If a variable is defined as a reference to a value then calling [Null->DetachReference] will set the variable's value to Null and detach it from the referenced object. The variable can then be safely re-assigned without affecting the referenced object.

Types of References

References can be used to refer to any of the following objects within Lasso.

- **Variables** – A reference to a variable allows the same underlying data to be accessed through two different names. Changing the value of either of the linked variables will result in the values of both variables being changed. The data referenced by both variables is only stored once.

  ```
  [Variable: 'Ref_Variable' = @$First_Variable]
  ```

- **Local Variables** – A reference to a page variable can be made within a custom tag. Rather than copying the page variable into a local variable, the page variable can be referenced. This prevents duplicating data and allows any changes made to the local variable to be automatically applied to the page variable.

  ```
  [Local: 'Local_Variable' = @$First_Variable]
  ```

- **Array Elements** – A reference can be made to an array element. This allows one or more array elements to be referenced as variables separate from the array. Any changes made to the variables will be reflected in the array. The [Array->Get] tag is used to identify the array element.

  [Variable: 'Ref_Variable' = @($Array_Variable->(Get: 1)]

- **Map Elements** – A reference can be made to the value of a map element. This allows the values of one or more map elements to be referenced as variables separate from the map. Any changes made to the variables will be reflected in the map. The [Map->Find] tag is used to identify the map element.

  [Variable: 'Ref_Variable' = @($Map_Variable->(Find: 'Key')]

- **Tag Parameters** – In a custom tag a reference can be made to a tag parameter rather than copying the parameter into a local variable. This allows a referenced parameter to be modified in place.

  [Local: 'Local_Variable' = @(Params->(Get: 1))]

Table 6: Reference Tags and Symbols

Tag / Symbol	Description
@	Creates a reference to an object rather than copying the object. Usually used in a [Variable] tag to assign a variable as a link to an object.
[Reference]	Creates a reference to an object rather than copying the object. Equivalent to the @ symbol.
[Null->DetachReference]	Can be called on a variable of any data type to detach the variable from the linked object. The variable ends up with a value of Null.
[Null->RefCount]	Returns the number of references that refer to a value.

To create a custom tag that works on an array directly:

The following example creates a custom tag that works on the elements of an array in place. Using this principle can greatly speed up the execution speed of LDML code since Lasso does not have to copy each element of the array multiple times.

References are used twice in this tag. The first parameter to the tag (which is expected to be an array) is referenced by a local variable theArray. This prevents the values of the array from being copied into the local variable. Within the [Loop] ...[/Loop] tags. The variable theItem is set to a reference to each element of the tag in turn.

```
[Define_Tag: 'Ex_Square']
  [Local: 'theArray' = @(Params->(Get: 1))]
  [Loop: #theArray->Size]
    [Local: 'theItem' = @(#theArray->(Get: Loop_Count))]
    [#theItem *= #theItem]
  [/Loop]
[/Define_Tag]
```

This tag is used as follows to modify the items in an array in place. Note that the tag does not have a [Return] tag so it does not return any value.

```
[Variable: 'myArray' = (Array: 1, 2, 3)]
[Ex_Square: $myArray]
[Variable: 'myArray']
```

➜ (Array: 1, 4, 9)

Lasso automatically uses references when referencing -Required or -Optional tag parameters and when using the [Iterate]... [/Iterate] tags. It is possible to rewrite the [Ex_Square] tag using these implicit references as follows. This tag will function identically to the previous example.

```
[Define_Tag: 'Ex_Square', -Required='theArray']
  [Iterate: #theArray, (Local: 'theItem')]
    [#theItem *= #theItem]
  [/Iterate]
[/Define_Tag]
```

16

Chapter 16
Conditional Logic

Conditional tags allow programming logic to be embedded into format files. Portions of a page can be hidden or repeated multiple times. Code can be executed in every repetition of a loop or every several repetitions. Complex decision trees can be created which execute code only under very specific conditions.

- *If Else Conditionals* explains how to use the [If] ... [/If] tags and [Else] tag to conditionally determine the results of a page or to execute LDML code.
- *If Else Symbol* describes the ? | trinary symbol that allows a conditional to be embedded within an expression.
- *Select Statements* explains how to use [Select] ... [Case] ... [/Select] tags to choose what code to execute based on the value of a variable.
- *Conditional Tags* describes tags that can be used as a parameter to another tag performing a conditional within an expression.
- *Loops* explains how to use the [Loop] ... [/Loop] tags to repeat a portion of the page and documents the [Loop_Abort] and [Loop_Count] tags used in any repeating container tag.
- *Iterations* explains how to use the [Iterate] ... [/Iterate] tags to perform an action using the value of each element of a compound data type in turn.
- *While Loops* explains how to use the [While] ... [/While] tags to repeat a portion of a page while a condition is True.
- *Abort Tag* explains how to use the [Abort] tag to halt execution of a format file.
- *Boolean Data Type* describes the [Boolean] tag and boolean symbols which can be used to create complex conditional expressions.

If Else Conditionals

Code can be conditionally executed and page elements can be conditionally shown by placing them within [If] ... [/If] container tags. The code or other page elements will only be processed if the expression in the opening [If] tag evaluates to True.

```
[If: (Variable: 'Test') == True]
  This text will be shown if the variable Test equals True.
[/If]
```

The [Else] tag allows for either/or logic to be programmed. If the condition in the [If] tag is True then the code between the [If] tag and the [Else] tag is processed, otherwise the code between the [Else] tag and the closing [/If] tag is processed.

```
[If: (Variable: 'Test') == True]
  This text will be shown if the variable Test equals True.
[Else]
  This text will be shown if the variable Test does not equal True.
[/If]
```

A series of tests can be made and code associated with the first test that returns True can be shown by specifying expressions within the [Else] tags. The code between the [Else] tag with a conditional expression and the next [Else] tag will only be shown if the expression returns True. As many [Else] tags as needed can be specified within a single set of [If] ... [/If] container tags.

Note: The [Select] ... [Case] ... [/Select] tags can be used to perform a similar operation. These tags are discussed in the next section.

```
[If: (Variable: 'Test') == (-1)]
  This text will be shown if the variable Test equals -1.
[Else: (Variable: 'Test') == 2]
  This text will be shown if the variable Test equals 2.
[Else: (Variable: 'Test') == 3]
  This text will be shown if the variable Test equals 3.
[/If]
```

A final [Else] tag without a conditional expression can be included. The code between the [Else] tag and the closing [/If] tag will only be processed if the expression in the opening [If] tag returns False and the expressions in all subsequent [Else] tags return False as well.

```
[If: (Variable: 'Test') == 1]
  This text will be shown if the variable Test equals 1.
[Else: (Variable: 'Test') == 2]
  This text will be shown if the variable Test equals 2.
[Else: (Variable: 'Test') == 3]
```

```
  This text will be shown if the variable Test equals 3.
[Else]
  This text will be shown if the variable Test is not equal to 1, 2, or 3.
[/If]
```

Table 1: If Else Tags

Tag	Description
[If] … [/If]	Executes the contents of the container only if the expression in the [If] tag returns True.
[Else]	Valid only within [If] … [/If] container tags. Executes the remainder of the container tag only if the expression in the [Else] tag returns True or no expression is specified.

The rules for specifying expressions in the [If] and [Else] tags are presented in full in the following section entitled *Boolean Data Type*.

Note: The [If] and [Else] tags will simply output the result of the specified conditional expression parameter if they are called individually on a page, i.e. not as part of a valid [If] ... [Else] ... [/If] container tag.

To conditionally execute code within a LassoScript:

Use the [If] tag with an appropriate conditional expression. In the following example, the expression will only be processed if the current username returned by the [Client_Username] tag is Anonymous.

```
<?LassoScript
  If: ((Client_Username) == 'Anonymous');
    'You are an anonymous user';
  /If;
?>
```

To show a different portion of a page if an error occurs:

Errors are reported in Lasso using the [Error_CurrentError] tag. This tag can be compared with many specific error type tags to check to see if a particular error occurred. In the following example, the current error is compared to [Error_SecurityError] in order to display an appropriate message.

```
[If: (Error_CurrentError) == (Error_SecurityError)]
  You don't have permission to access that resource.
[/If]
```

Note: See the *Error Control* chapter for more information about the [Error_...] tags.

Complex Conditionals

There are two methods for creating complex conditionals. Each of these methods can be used interchangeably depending on what conditions need to be checked and the preference of the Lasso developer.

Examples of complex conditionals

- The conditional expression within the opening [If] tag can be used to check several different conditions. The conditions are appended using the and && symbol which returns True if both parameters return True or the or || symbol which returns True if either parameter returns True.

  In the following example, two fields from a database are checked to determine what title to put on a salutation. The Sex field is checked to see if the visitor is Male or Female and the Married field is checked to see if the visitor is Married or Single. Compound conditional expressions are created to check for the combination of gender and marriage status for each title.

```
[If: ((Field: 'Sex') == 'Male')]
  Dear Mr. [Field: 'First_Name'] [Field: 'Last_Name'],
[Else: ((Field: 'Sex') == 'Female') && ((Field: 'Marriage') == 'Married')]
  Dear Mrs. [Field: 'First_Name'] [Field: 'Last_Name'],
[Else: ((Field: 'Sex') == 'Female') && ((Field: 'Marriage') == 'Single')]
  Dear Ms. [Field: 'First_Name'] [Field: 'Last_Name'],
[Else]
  To whom it may concern,
[/If]
```

- Nested [If] ... [/If] tags can be used to check several conditions in turn. The conditional expression in each [If] tag is simple, but the nesting establishes that the innermost [If] ... [/If] tags are only executed if the outermost [If] ... [/If] tags evaluate their conditional expression to True.

  In the following example the [If] ... [/If] tags cause the Marriage field to be evaluated if the conditional expression in the outermost [Else] tag finds that the Sex field contains Female.

```
[If: ((Field: 'Sex') == 'Male')]
  Dear Mr. [Field: 'First_Name'] [Field: 'Last_Name'],
[Else: ((Field: 'Sex') == 'Female')]
  [If: ((Field: 'Marriage') == 'Married')]
    Dear Mrs. [Field: 'First_Name'] [Field: 'Last_Name'],
  [Else: ((Field: 'Marriage') == 'Single')]
    Dear Ms. [Field: 'First_Name'] [Field: 'Last_Name'],
  [/If]
[/If]
```

If Else Symbol

Lasso includes an expression that allows a conditional to be executed without using the [If] … [/If] tags. The ? | symbol allows a conditional to be executed within an expression. The symbol uses the following format:

```
(Conditional ? True Result | False Result)
```

If the conditional evaluates to True then the true result is evaluated otherwise the false result is evaluated. Since only one of the results is evaluated it is possible to use this tag for conditional evaluation of parts of an expression.

If the | portion of the tag and the false result are omitted and the condition returns false then Null is returned.

Table 2: If Else Symbol

Symbol	Description
? |	If the test before ? evaluates to true then the value after the ? is returned, otherwise the value after | is returned. For example (Conditional ? True Result | False Result)

Some examples will make the use of the symbol more clear.

- A variable can be set to one of two values based on a conditional using the ? | symbol. In this example the variable myValue is set to True if the $Test is True or False otherwise.

```
[Var: 'myValue' = ( $Test == True ? 'True' | 'False')]
```

- An alternate value can be returned for an empty field using the ? | symbol. In this example if the field First_Name is empty then N/A is returned.

```
[Encode_HTML: (Field: 'First_Name') == '' ? 'N/A' | (Field: 'First_Name')]
```

- The value passed into an inline can be decided using the ? | symbol. In this example if a value is equal to null then an empty string is inserted instead.

```
[Inline: -Add,
    ...
    -Op='eq', 'Field_Name'=($Field_Name === null ? '' | $field_name),
    ...
] ... [/Inline]
```

- A tag can be conditionally executed using just the ? symbol. In this example [Loop_Abort] is executed if [Loop_Count] is greater than 1000.

```
[ (Loop_Count > 1000 ? Loop_Abort ]
```

- A tag can be conditionally executed using the ? | symbol. In this example one or the other URL is included based on the conditional value, but not both.

```
[ $Conditional ?
  (Include_URL: 'http://www.omnipilot.com') |
  (Include_URL: 'http://www.apple.com')]]
```

Select Statements

Select statements can be used when a variable can take multiple values and a different block of code should be executed depending on the current value. The variable to be checked is specified in the opening [Select] tag. A series of [Case] tags follow, each specified with a possible value of the variable. If one of the [Case] tags matches the value of the variable then the code until the next [Case] tag or the closing [/Select] tag will be executed.

For example, to return different text depending on value a variable named Test current has the following [Select] ... [/Select] statement could be used.

```
[Select: (Variable: 'Test')]
  [Case (-1)]
    This text will be shown if the variable Test equals -1.
  [Case: 2]
    This text will be shown if the variable Test equals 2.
  [Case: 3]
    This text will be shown if the variable Test equals 3.
[/Select]
```

A [Case] tag without any value is used as the default value for the [Select] ... [/Select] statement in the event that no [Case] statement matches the value of the parameter of the opening [Select] tag. The first [Case] tag without any value is returned as the default value.

```
[Select: (Variable: 'Test')]
  [Case (-1)]
    This text will be shown if the variable Test equals -1.
  [Case: 2]
    This text will be shown if the variable Test equals 2.
  [Case: 3]
    This text will be shown if the variable Test equals 3.
  [Case]
    This text is shown if the variable does not equal any of the values.
[/Select]
```

Table 3: Select Tags

Tag	Description
[Select] … [/Select]	Takes a single parameter which is used to decide which enclosed [Case] tag to select. Requires one or more [Case] tags to be specified. Returns the value of the code between the selected [Case] statement and the next [Case] statement or the closing [/Select] tag.
[Case]	Accepts a single parameter which is checked against the parameter of the enclosing [Select] tag. If no parameter is specified then the tag defines the default case.

To return a different value based on the type of a variable:

Use the [Select] ... [Case] ... [/Select] tags to return a different value depending on the type of a variable. The following code outputs the value of a variable named MyVariable that could be of any type. If the variable is not of any built-in type then the default output is to cast it to string.

```
[Select: (Variable: 'MyVariable')->Type]
  [Case: 'Integer']
    <br>Integer value [Variable: 'MyVariable'].
  [Case: 'Decimal']
    <br>Decimal value [Variable: 'MyVariable'].
  [Case: 'String']
    <br>String value [Variable: 'MyVariable'].
  [Case: 'Boolean']
    <br>Boolean value [Variable: 'MyVariable'].
  [Case: 'Array']
    <br>Array value [Variable: 'MyVariable'].
  [Case: 'Map']
    <br>Map value [Variable: 'MyVariable'].
  [Case: 'Pair']
    <br>Pair value [Variable: 'MyVariable'].
  [Case]
    <br>Unknown type value [String: (Variable: 'MyVariable')].
[/Select]
```

Conditional Tags

Lasso offers a collection of tags that can be used to specify a conditional as a parameter to another tag.

Note: The if else symbol ? | documented earlier in this chapter has several advantages over these tags. Most notably, it allows conditional execution of either the true or false result of the symbol, while these tags always evaluate both results before checking the conditional.

Table 4: Conditional Tags

Tag	Description
[If_True]	The first parameter is a conditional statement. If the first parameter is True then the second parameter is returned. Otherwise the third parameter is returned.
[If_False]	The first parameter is a conditional statement. If the first parameter is False then the second parameter is returned. Otherwise the third parameter is returned.
[If_Empty]	The first parameter should be a value. If it's size is greater than 0 it is returned. Otherwise, the second parameter is returned.
[If_Null]	The first parameter should be a value. If it is not equal to Null it is returned. Otherwise, the second parameter is returned.

[If_True] and [If_False] each accept three parameters. The first parameter is a conditional expression that selects whether the second or third parameter should be returned.

In the following example the variable MyResult is set to the appropriate value depending on whether the Condition variable is true or false.

```
[Var: 'MyResult' = (If_True: $Condition, 'Condition is True', 'Condition is False')]
```

[If_Empty] and [If_Null] each accept two parameters. If the first parameter is non-empty or not equal to Null then it is returned. Otherwise, the second parameter is returned.

In the following example, a default value is used if the [Cookie_List] tag returns an empty array.

```
[Var: 'MyCookies' = (If_Empty: Cookie_List, 'No Cookies!')]
```

Loops

A portion of a page can be repeated a number of times using the [Loop] ... [/Loop] tags. The parameters to the opening [Loop] tag define how many times the portion of the page should be repeated. For example, a message in a Web page could be repeated five times using the following [Loop] tag.

```
[Loop: 5]
  <br>This is repeated five times.
[/Loop]
```

```
➜ <br>This is repeated five times.
<br>This is repeated five times.
<br>This is repeated five times.
<br>This is repeated five times.
<br>This is repeated five times.
```

The basic form of the [Loop] ... [/Loop] tags simply repeats the contents of the tags as many times as is specified by the parameter. The opening [Loop] tag can also accept a number of keyword/value parameters to create more complex repetitions.

Table 5: [Loop] Tag Parameters

Keyword	Description
-From	Specifies the starting repetition for the [Loop] tag. Can also be specified as -LoopFrom.
-To	Specifies the ending repetition for the [Loop] tag. Can also be specified as -LoopTo.
-By	Specifies how many repetitions should be skipped on each actual repetition of the contents of the [Loop] ... [/Loop] tag. Can also be specified as -LoopIncrement.

The following example shows a loop that runs backward for five repetitions by setting -From to 5, -To to 1 and -By to -1. The [Loop_Count] tag shows the number of the current repetition.

```
[Loop: -From=5, -To=1, -By=-1]
  <br>This is repetition number [Loop_Count].
[/Loop]
```

```
➜ <br>This is repetition number 5.
<br>This is repetition number 4.
<br>This is repetition number 3.
<br>This is repetition number 2.
<br>This is repetition number 1.
```

Note: The [Loop_Count] tag can be used in any looping container tag within LDML to return the number of the current repetition. This includes the [Records] ... [/Records] tags.

The [Loop_Abort] tag can be used to halt a [Loop] before it reaches the specified number of repetitions. In the following example, the [Loop] tag is stopped after the third repetition by checking to see if [Loop_Count] is equal to 3.

```
[Loop: 5]
  <br>This is repeated five times.
  [If: (Loop_Count) == 3]
    [Loop_Abort]
  [/If]
[/Loop]

➜ <br>This is repeated five times.
<br>This is repeated five times.
<br>This is repeated five times.
```

Note: The [Loop_Abort] tag can be used in any looping container tag within LDML to abort the loop. This includes the [Records] ... [/Records] tags.

The modulus symbol % can be used in an [If] ... [/If] conditional to perform a task on every other repetition (or every nth repetition). The conditional expression (Loop_Count % 2)==0 returns True for every other repetition of the loop.

```
[Loop: 5]
  [If: (Loop_Count % 2) == 0]
    <br>This is an Even loop.
  [Else]
    <br>This is an Odd loop.
  [/If]
[/Loop]

➜ <br>This is an odd loop.
<br>This is an even loop.
<br>This is an odd loop.
<br>This is an even loop.
<br>This is an odd loop.
```

The modulus symbol can be used in any looping container tag within LDML to show elements in alternate rows. This includes the [Records] ... [/Records] tags.

Note: The [Repetition] tag from earlier versions of Lasso has been deprecated. It's use is not recommended. Any code using the [Repetition] tag should be changed to the modulus operator for dramatically better speed and future compatibility.

Table 6: Loop Tags

Tag	Description
[Loop] … [/Loop]	Repeats the contents of the container tag a specified number of times.
[Loop_Count]	Returns the number of the current repetition.
[Loop_Abort]	Aborts the [Loop] … [/Loop] tag, jumping immediately to the closing tag.
[Loop_Continue]	Aborts the current repeition of the looping tag, jumping immediately to the next repetition.

To list all the field names for a table:

An [Inline] ... [/Inline] with a -Show command tag can be used to get a list of all the field names in a table. The [Field_Name] tag accepts a -Count parameter that returns how many fields are in the current table or an integer parameter that returns the name of one of the fields. The following example uses the [Loop] ... [/Loop] tags to display a list of all the field names in a table.

```
[Inline: -Database='Contacts', -Table='People', -Show]
  [Loop: (Field_Name: -Count)]
    <br>[Field_Name: (Loop_Count)]
  [/Loop]
[/Inline]
```

```
➜ ID
  First_Name
  Last_Name
```

To loop through the elements of an array:

The elements of an array can be displayed to a site visitor or otherwise manipulated by looping through the array using the [Loop] ... [/Loop] tags. The [Array->Size] tag returns the number of elements in an array and the [Array->Get] tag returns a specific element by index. The following example shows how to store the names of the days of the week in an array and then list those elements using [Loop] ... [/Loop] tags.

```
<?LassoScript
  Encode_Set: -EncodeNone;

  Variable: 'DaysOfWeek' = (Array: 'Sunday', 'Monday', 'Tuesday',
    'Wednesday', 'Thursday', 'Friday', 'Saturday');

  Loop: ($DaysOfWeek->Size);
    '<br>' + $DaysOfWeek->(Get: (Loop_Count));
  /Loop;

  /Encode_Set;
?>

➜ <br>Sunday
<br>Monday
<br>Tuesday
<br>Wednesday
<br>Thursday
<br>Friday
<br>Saturday
```

Note: See the ***Arrays and Maps*** chapter for more information about the array member tags.

To format a found set in two columns:

The modulus symbol % can be used to format a found set in two columns. In the following example, an HTML <table> is constructed with one cell for each person found by an [Inline] ... [/Inline] based -FindAll action. The modulus symbol % is used to insert the row tags every other record.

```
[Inline: -Database='Contacts', -Table='People', -FindAll]
  <table>
    <tr>
    [Records]
      <td>[Field: 'First_Name'] [Field: 'Last_Name']</td>
      [If: (Loop_Count % 2) == 0]
        </tr><tr>
      [/If]
    [/Records]
    </tr>
  </table>
[/Inline]

➜ <table>
    <tr>
      <td>Jane Person</td>
      <td>John Person</td>
```

```
  </tr><tr>
    <td>Joe Surname</td>
  </tr>
</table>
```

Iterations

The [Iterate] ... [/Iterate] tags loop through each element of a complex data type such as an array or a map. A variable is set to the value of each element of the complex data type in turn. This allows the same operation to be performed on each element.

Note: The [Iterate] ... [/Iterate] tags can be used with built-in array, map, pair, and string data types. It can also be used with any custom data type that supports the [Type->Size] and [Type->Get] member tags.

For example, to print out each element of an array stored in a variable myArray the following tags could be used. The opening [Iterate] tag contains the name of the variable storing the array and a definition for the variable that should be set to each element of the array in turn. In this case a new variable myItem will be created. The value for myItem is then output within the [Iterate] ... [/Iterate] tags.

```
[Variable: 'myArray' = (Array: 'Winter', 'Spring', 'Summer', 'Autumn')]
[Iterate: (Variable: 'myArray'), (Variable: 'myItem')]
  <br>The season is: [Variable: 'myItem'].
[/Iterate]
```

```
➜ <br>The season is: Winter.
  <br>The season is: Spring.
  <br>The season is: Summer.
  <br>The season is: Fall.
```

The [Iterate] ... [/Iterate] tags are equivalent to using [Loop] ... [/Loop] tags to cycle through each element of a complex data type, but are significantly easier to use and provide faster operation.

Table 7: Iteration Tags

Tag	Description
[Iterate] ... [/Iterate]	Cycles through each element of a compound data type in turn. The opening tag accepts two parameters. The first is the compound data type to be iterated through. The second is a reference to a variable which should be set to the value of each element of the first parameter in turn.

Note: The second parameter to the opening [Iterate] tag should either be of the form (Variable: 'NewVariableName') or should reference an existing variable using $ExistingVariable. The $ symbol cannot be used to create a new variable.

To print out each character of a string:

Use the [Iterate] ... [/Iterate] tags to cycle through each character of the string in turn. The following code prints out each character of a string on a separate line.

```
[Variable: 'myString'='blue']
[Iterate: $myString, (Variable: 'myCharacter')]
  <br>[Variable: 'myCharacter']
[/Iterate]
```

```
➜ <br>b
<br>l
<br>u
<br>e
```

While Loops

[While] ... [/While] tags allow a portion of a page to repeat while a specified conditional expression returns True. The expression specified in the opening [While] tag is checked on each pass through the loop and if the expression returns True then the contents are displayed again.

In the following example, a variable ConditionVariable is set to True. Once the [Loop_Count] is greater than 3 the variable is set to False, ending the [While] ... [/While] loop.

```
[Variable: 'ConditionVariable' = True]
[While: ($ConditionVariable == True)]
  <br>This is repetition [Loop_Count]
  [If: (Loop_Count) >= 3]
    [Variable: 'ConditionVariable' = False]
  [/If]
[/Loop]
```

```
➜ <br>This is repetition 1.
<br>This is repetition 2.
<br>This is repetition 3.
```

Table 8: While Tags

Tag	Description
[While] … [/While]	Repeats the contents of the container tag until the condition specified in the opening tag returns False.
[Loop_Count]	Returns the number of the current repetition.
[Loop_Abort]	Aborts the [While] … [/While] tag, jumping immediately to the closing tag.

Abort Tag

The [Abort] tag can be used to abort the execution of the current format file. This can be useful in a situation where an error has occurred that prevents the rest of the file from executing. An [Abort] can be used after a [Redirect_URL] so Lasso does not need to process the rest of the page before sending the redirect to the client. Finally, an [Abort] can be used in a custom error page in order to prevent the standard error message from being shown at the bottom of the page.

Table 9: Abort Tag

Tag	Description
[Abort]	Aborts the current format file, returning all of the content which has been created so far to the client.

To speed up a [Redirect_URL]:

Use the [Abort] tag immediately after the [Redirect_URL] tag. All LDML code after the [Abort] tag will be ignored so the [Redirect_URL] tag's modifications to the HTTP response will be sent to the client immediately.

```
[Redirect_URL: 'http://www.example.com/']
[Abort]
```

Boolean Type

The boolean data type simply represents True or False. All comparison symbols and boolean symbols in LDML return a value of the boolean data type.

The following values are equivalent to each of the boolean values both when automatically cast and when explicitly cast using the [Boolean] tag. However, it is recommended that you use True and False whenever possible to avoid confusion.

- True is equivalent to any positive integer or decimal such as 1, 45, or 100.15, any non-empty string such as 'String', or any non-null data type such as an array, map, or pair.
- False is equivalent to integer 0 or decimal 0.0, the empty string '', or Null.

Note: The string 'True' happens to be equivalent to True, but the string 'False' is not equivalent to False. Always type the boolean values True and False without quotation marks.

Table 10: Boolean Tag

Tag	Description
[Boolean]	Casts a value to a boolean value.

The boolean data type is most commonly associated with conditional expressions such as those specified in the opening [If] or [While] tags. Any conditional expression which uses a conditional symbol such as ==, !=, <, <=, >, >=, or >> will return a boolean value. Multiple conditional expressions can be combined using any of the boolean symbols detailed in the following table.

Table 11: Boolean Symbols

Symbol	Description
&&	And. Returns True if both parameters are True.
||	Or. Returns True if either parameter is True.
!	Not. Returns False if the parameter following is True.
==	Equality. Returns True if both parameters are equal.
!=	Inequality. Returns True if both parameters are different.

Note: Single parameter expressions must be surrounded by parentheses if they are used on the right hand side of a boolean symbol.

To check for two conditions in an [If] tag:

- In order to return True if both conditions are True use the && symbol.

```
[If: ($Condition1 == True) && ($Condition2 == True)]
  Both conditions are True.
[/If]
```

- In order to return True if either of the conditions is True use the || symbol.

```
[If: ($Condition1 == True) || ($Condition2 == True)]
  One of the conditions is True.
[/If]
```

- In order to return True if a condition is False use the ! symbol.

```
[If: !($Condition1 == True)]
  The condition is False.
[/If]
```

- In order to return True if the two conditions are equal (both True or both False) use the == symbol.

```
[If: ($Condition1 == True) == ($Condition2 == True)]
  Both conditions are True or both conditions are False.
[/If]
```

- In order to return True if the two conditions are not equal (one is True and the other is False) use the != symbol.

```
[If: ($Condition1 == True) != ($Condition2 == True)]
  One condition is True and the other is False.
[/If]
```

To use single parameter symbols in a comparison:

If expressions using the single-parameter symbols !, -, and + are going to be used as the second parameter to a comparison symbol, they should be surrounded by parentheses.

- To compare a variable to -1 use parentheses around -1 on the right-hand side of the comparison operator.

```
[If: ($Variable == (-1))]
  The variable is equal to -1.
[Else: ($Variable > (-1))]
  The variable is greater than -1.
[Else: ($Variable < (-1))]
  The variable is less than -1.
[/If]
```

- To compare a variable to the negation of an expression, use parentheses around the entire right-hand side of the comparison operator.

```
[If: ($Variable == (!True))]
  The variable is not equal to False.
[/If]
```

Note: These expressions can usually be rewritten with the opposite comparison symbol or by using the negation symbol around the entire conditional expression.

17

Chapter 17
Encoding

Lasso can be used to publish data in many different formats. Encoding ensures that only legal characters are used for the desired output format.

- *Overview* describes the different formats which LDML encoding supports.
- *Encoding Keywords* describes how to use encoding keywords to modify the output of substitution tags.
- *Encoding Controls* describes how to use the [Encode_Set] ... [/Encode_Set] tags to modify the default encoding for substitution tags.
- *Encoding Tags* describes the individual substitution tags which can be used to encode values.

Overview

Encoding controls in LDML allow the developer to specify the format in which data output from substitution tags should be rendered. Encoding controls ensure that reserved or illegal characters are changed to entities so that they will display properly in the desired output format. Encoding controls allow for data to be output in any of the ways described in the *Encoding Formats* section below.

Encoding Rules

Encoding controls apply to the data output from tags differently depending on how the tags are used. Substitution tags have default HTML encoding if they output a value to a page. The value output from a nested substitution

tag is not encoded. Substitution tags which contribute to the output of a LassoScript have default HTML encoding.

- **Substitution Tags** which output a value to the site visitor have a default encoding of -EncodeHTML. These tags are usually enclosed in square brackets and do not include nested tags which return values.

  The default encoding ensures that any reserved or illegal characters in HTML are converted to HTML entities so they display properly. The default encoding can be overridden by explicitly including an encoding keyword in the substitution tag or using the [Encode_Set] ... [/Encode_Set] tags described below.

  In the following example, some HTML code is output using the [String] substitution tag. By default the angle brackets in the code are converted to HTML entities so they will display as angle brackets within a Web browser. If the -EncodeNone keyword is specified in the [String] substitution tag then the angle brackets remain as text angle brackets and the HTML code will render as Bold Text within the Web browser.

  ```
  [String: '<b>Bold Text</b>'] ➜ &lt;b&gt;Bold Text&lt;/b&gt;

  [String: '<b>Bold Text</b>', -EncodeNone] ➜ <b>Bold Text</b>
  ```

- **Nested Substitution Tags** are not encoded by default. This ensures that string calculations can be performed without having to specify any encoding keywords. However, the encoding of a nested substitution tag can be changed by explicitly including an encoding keyword. Care should be taken so that values are not encoded multiple times.

  In the following example a string is stored in a variable using explicit HTML encoding. When the variable is output using the -EncodeNone tag, the value is output to the page with HTML encoding intact.

  ```
  [Variable: 'HTML_Text' = (Output: '<b>Bold Text</b>', -EncodeHTML)]

  [Variable: 'HTML_Text', -EncodeNone] ➜ &lt;b&gt;Bold Text&lt;/b&gt;
  ```

- Tags within **LassoScripts** are encoded using the same rules for substitution tags. Tags which add to the output of the LassoScript are HTML encoded by default unless an explicit encoding keyword is specified or the [Encode_Set] ... [/Encode_Set] tags are used. Tags which are nested are not encoded by default unless an explicit encoding keyword is specified.

  The following example shows a value output from a LassoScript first with the default HTML encoding, then with an explicit -EncodeNone keyword specified.

  ```
  <?LassoScript
    Output: '<b>Bold Text</b>';
  ?>

  ➜ &lt;b&gt;Bold Text&lt;/b&gt;
  ```

```
<?LassoScript
  Output: '<b>Bold Text</b>', -EncodeNone;
?>
```

➜ Bold Text

- **Square Bracketed Expressions** other than tags are not encoded by default. The use of the [String] tag or one of the [Encode_...] tags is recommended to ensure that encoding is properly applied to string expressions. In the following example a string expression is output directly.

```
['<b>' + 'Bold Text' + '</b>']
```

➜ Bold Text

Encoding Formats

The encoding controls in LDML can be used to output data in any of the following formats.

- **HTML Encoding** is the default output format. Reserved characters in HTML including < > " & are encoded into HTML entities. Extended-ASCII and foreign language characters are encoded into a numerical HTML entity for the character &#nnn;. Use the -EncodeHTML keyword or the [Encode_HTML] substitution tag.
- **Smart HTML Encoding** encodes only extended-ASCII and foreign language characters. The reserved characters in HTML are not encoded. This allows HTML code to be displayed with the HTML markup intact and any unsafe characters encoded using HTML entities. Use the -EncodeSmart keyword or the [Encode_Smart] substitution tag.
- **Break Encoding** encodes carriage returns and line feeds within the text to HTML <br> tags. The remainder of the text is HTML encoded. Text can be formatted using the -EncodeBreak keyword or the [Encode_Break] substitution tag.
- **XML Encoding** encodes reserved characters such as & ' " < > which are used to create the markup of XML into XML entities. This ensures that text used in XML tag names or attributes does not contain any reserved characters. Use the -EncodeXML keyword or the [Encode_XML] substitution tag.
- **Simple URL Encoding** only encodes illegal characters such as < > # % { } ' ` " | \ ^ ~ [ ] © ® into URL entities specified as %nn. Simple URL encoding can be used to encode an entire URL without disturbing the basic structure of the URL. Use the -EncodeURL keyword or the [Encode_URL] substitution tag. The following example shows a URL encoded with the [Encode_URL] tag.

```
[Encode_URL: 'http://www.example.com/Action.Lasso?The Name=A Value']
➜ http://www.example.com/Action.Lasso?The%20Name=A%20Value
```

- **Strict URL Encoding** encodes both the illegal characters shown above and the reserved characters in URLs including ; / ? : @ = &. Strict URL encoding should only be used on the names or values included as name/value parameters. Use the -EncodeStrictURL keyword or the [Encode_StrictURL] substitution tag. The following example shows only the name/value parameter of a URL encoded with the [Encode_StrictURL] tag.

```
http://www.example.com/Action.Lasso?
  [Encode_StrictURL: 'The Name']=[Encode_StrictURL: 'A Value']
➜ http://www.example.com/Action.Lasso?The%20Name=A%20Value
```

- **SQL Encoding** changes any illegal characters in SQL string values into their escaped equivalents. Quote marks and backslashes are escaped so they don't interfere with the structure of the SQL statement.

```
[Encode_SQL: 'A "String" is born.']
➜ A \"String\" is born.
```

- **Base 64 Encoding** changes any string value into a string of ASCII characters which can be safely transmitted through URLs or email. This algorithm is sometimes used to obscure data so it is difficult to read by a casual passerby without providing any actual security. Base64 is also used to transmit passwords (essentially as plain-text) to some Web servers.

Deactivate encoding for a substitution tag using the -EncodeNone keyword. By default, nested substitution tags will not have encoding applied so the -EncodeNone keyword is not required within nested substitution tags.

Encoding Keywords

Encoding keywords can be used within any substitution tag to modify the encoding of the output value of that tag. Substitution tags which output values to the page default to -EncodeHTML so this keyword does not need to be specified if HTML encoding is desired. Nested substitution tags are not encoded by default, specifying -EncodeNone in nested substitution tag is unnecessary.

Only one encoding keyword can be used in a tag. If multiple encodings are desired the [Encode_...] tags should be used.

Table 1: Encoding Keywords

Keyword	Description
-EncodeBreak	Encodes carriage returns and new line characters into HTML breaks. The remainder of the text is HTML encoded.
-EncodeHTML	Encodes HTML reserved and illegal characters into HTML entities for highest fidelity display.
-EncodeNone	Performs no encoding.
-EncodeSmart	Encodes HTML illegal characters into HTML entities. Useful for encoding strings that contain HTML markup.
-EncodeStrictURL	Encodes all URL reserved and illegal characters into URL entities for highest fidelity data transmission.
-EncodeURL	Encodes URL illegal characters into URL entities. Useful for encoding entire URLs.
-EncodeXML	Encodes XML reserved and illegal characters into XML entities for highest fidelity data transmission.

Please consult the previous section *Encoding Formats* for information about what characters each encoding keyword modifies.

Using the encoding keywords:

The following example shows how text is output from the [String] tag using first the default -EncodeHTML encoding and then an explicit -EncodeNone encoding.

```
[String: '<b>Bold Text</b>'] ➜ &lt;b&gt;Bold Text&lt;/b&gt;

[String: '<b>Bold Text</b>', -EncodeNone] ➜ <b>Bold Text</b>
```

Encoding Controls

The default encoding keyword for substitution tags which output values to the Web page being constructed can be modified using the [Encode_Set] ... [/Encode_Set] tags. All square bracketed substitution tags or tags within a LassoScript that output a value will use the encoding specified in surrounding [Encode_Set] ... [/Encode_Set] tags rather than the default HTML encoding.

The [Encode_Set] tag accepts a single parameter, an encoding keyword. Any of the valid encoding keywords from *Table 1: Encoding Keywords* can be used. All substitution tags which output values will behave as if this encoding keyword were specified within the tag.

Nested substitution tags (sub-tags) will not be affected by the [Encode_Set] ... [/Encode_Set] tags. Values from nested substitution tags are not encoded unless an encoding keyword is specified explicitly within each tag.

Table 2: Encoding Controls

Keyword	Description
[Encode_Set] ... [/Encode_Set]	Sets the default encoding for all substitution tags which output values within the container tag.

To change the default encoding for a LassoScript:

Start and end the LassoScript with [Encode_Set] ... [/Encode_Set] tags. In the following LassoScript HTML code is output using [String] tags. The default encoding for all tags is set to -EncodeNone so that the HTML is rendered properly in the output.

```
<?LassoScript
  Encode_Set: -EncodeNone;
    String: '<b>HTML Text</b>';
  /Encode_Set;
?>
```

➜ Bold Text

Encoding Tags

The encoding substitution tags can be used to explicitly encode any string value. The output of these tags is the same as the output which would be produced by using the appropriate encoding keyword on a substitution tag that returned the same value.

Note: The encoding tags do not accept encoding keywords. Use nested encoding tags to perform multiple encodings.

Table 3: Encoding Tags

Keyword	Description
[Decode_Base64]	Decodes a string which has been encoded using the base 64 algorithm. Accepts one parameter, a string to be decoded.
[Decode_Hex]	Decodes a binhex encoded string to a byte stream.
[Decode_HTML]	Decodes HTML by changing HTML entities back into extended ASCII characters.
[Decode_QHeader]	Decodes a MIME header which was encoded using the Q (quoted printable) encoding method.
[Decode_QuotedPrintable]	Decodes text using the quoted printable algorithm. Accepts two parameters: the text to be decoded and an optional character set it should be decoded from (defaults to UTF-8).
[Decode_URL]	Decodes a URL by changing URL entities back into extended ASCII characters.
[Encode_Base64]	Encodes a string using the base 64 algorithm. Accepts one parameter, a string to be encoded.
[Encode_Break]	Encodes carriage returns and new line characters into HTML breaks. The remainder of the text is HTML encoded.
[Encode_Hex]	Encodes a byte stream into a binhex encoded string.
[Encode_HTML]	Encodes HTML reserved and illegal characters into HTML entities for highest fidelity display.
[Encode_QHeader]	Encodes a MIME header which using the Q (quoted printable) encoding method. Requires a -Name and -Value parameter. Also accepts an optional -CharSet parameter (defaults to UTF-8).
[Encode_QuotedPrintable]	Encodes text using the quoted printable algorithm. Accepts two parameters: the text to be encoded and an optional character set it should be encoded as (defaults to UTF-8).
[Encode_Smart]	Encodes HTML illegal characters into HTML entities. Useful for encoding strings that contain HTML markup.
[Encode_SQL]	Encodes illegal characters in SQL string literals by escaping them with a backslash.
[Encode_StrictURL]	Encodes all URL reserved and illegal characters into URL entities for highest fidelity data transmission.
[Encode_URL]	Encodes URL illegal characters into URL entities. Useful for encoding entire URLs.
[Encode_XML]	Encodes XML reserved and illegal characters into XML entities for highest fidelity data transmission.

Using the encoding tags:

The following example shows how text is output from the [Encode_HTML] tag with all HTML reserved characters encoded. The same text is then output from an [String] tag with an encoding keyword of -EncodeNone specified.

```
[Encode_HTML: '<b>Bold Text</b>'] ➜ &lt;b&gt;Bold Text&lt;/b&gt;

[String: '<b>Bold Text</b>', -EncodeNone] ➜ <b>Bold Text</b>
```

18

Chapter 18
Sessions

This chapter documents sessions and server-side variables.

- *Overview* describes how sessions operate and how sessions can be used.
- *Session Tags* describes the tags which can be used to create, manipulate, and delete sessions.
- *Session Example* describes how to use sessions to store site preferences.

Overview

Sessions allow variables to be created which are persistent from page to page within a Web site. Rather than passing data from page to page using HTML forms or URLs, data can be stored in ordinary LDML variables which are automatically stored and retrieved by Lasso on each page a visitor loads.

Sessions are very easy to use, but the intricacies can be rather difficult to explain. The *Session Examples* section later in this chapter presents three examples for how to use sessions to perform common tasks. These examples should be consulted first to see real world examples of sessions in action before reading through the tag reference sections.

Ways in which sessions can be used:

- **Current State** – Sessions can store the current state of a Web site for a given visitor. They can determine what the last search they performed was, how the data on a results page was sorted, or in what format the data should be presented.

- **Store References to Database Records** – Key field values can be stored in a session for quick access to records associated with a site visitor. These might include records in a user database or shopping cart database.
- **Store Authentication Information** – After a visitor has authenticated themselves using a username and password, that authentication information can be stored in a session and then checked on each page to ensure that the same visitor is accessing data from page to page.
- **Store Data Without Using a Database** – Complex data types such as arrays and maps can be stored in session variables. In a Web site with multiple forms the data from each form can be stored in a session and only placed in the database once the final form is submitted. Or, a shopping cart can be stored in a session and only placed in an orders database on checkout.

How Sessions Work

A session has three characteristics: a name, a list of variables that should be stored, and an ID string that identifies a particular site visitor.

- **Name** – The session name is defined when the session is created by the [Session_Start] tag. The same session name must be used on each page in the site which wants to load the session. The name usually represents what type of data is being stored in the session, e.g. Shopping_Cart or Site_Preferences.
- **Variables** – Each session maintains a list of variables which are being stored. Variables can be added to the session using [Session_AddVariable]. The values for all variables in the session are remembered at the bottom of each page which loads the session. The last value for each variable is restored when the session is next loaded.
- **ID** – Lasso automatically creates an ID string for each site visitor when a session is created. The ID string is either stored in a cookie or passed from page to page using the -Session command tag. When a session is loaded the ID of the current visitor is combined with the name of the session to load the particular set of variables for the current visitor.

Sessions are created and loaded using the [Session_Start] tag. This tag should be used on the top of each page which needs access to the shared variables. The [Session_Start] either creates a new session or loads an existing session depending on what session name is specified and the ID for the current visitor.

Sessions can be set to expire after a specified amount of idle time. The default is 15 minutes. If the visitor has not loaded a page which starts the session within the idle time then the session will be deleted automatically.

Note that the idle timeout restarts every time a page is loaded which starts the session.

Once a variable has been added to a session using the [Session_AddVariable] tag it will be set to its stored value each time the [Session_Start] tag is called. The variable does not need to be added to the session on each page. A variable can be removed from a session using the [Session_RemoveVariable] tag. This tag does not alter the variable's value on the current page, but prevents the value of the variable from being stored in the session at the end of the current page.

Session Tags

Each of the session tags is described in *Table 1: Session Tags*. The parameters for [Session_Start] are described in more detail in *Table 2: [Session_Start] Parameters*.

Table 1: Session Tags

Tag	Description
[Session_Start]	Starts a new session or loads an existing session. Accepts four parameters: -Name is the name of the session to be started. Additional parameters are described in Table 2: [Session_Start] Parameters.
[Session_ID]	Returns the current session ID. Accepts a single parameter: -Name is the name of the session for which the session ID should be returned.
[Session_AddVariable]	Adds a variable to a specified session. Accepts two parameters: -Name is the name of the session and a second unnamed parameter is the name of the variable.
[Session_RemoveVariable]	Removes a variable from a specified session. Accepts two parameters: -Name is the name of the session and a second unnamed parameter is the name of the variable.
[Session_End]	Deletes the stored information about a named session for the current visitor. Accepts a single parameter: -Name is the name of the session to be deleted.
[Session_Abort]	Prevents the session from being stored at the end of the current page. This allows graceful recovery from an error that would otherwise corrupt data stored in the session.
[Session_Result]	When called immediately after the [Session_Start] tag, returns "new", "load", or "expire" depending on whether a new session was created, an existing session loaded, or an expired session forced a new session to be created.

Table 2: [Session_Start] Parameters

Keyword	Description
-Name	The name of the session.
-Expires	The idle expiration time for the session in minutes.
-ID	The ID for the current visitor. If no ID is specified then the cookie and link parameters will be inspected for valid visitor IDs.
-UseCookie	If specified then site visitors will be tracked by cookie. -UseCookie is the default unless -UseLink or -UseNone is specified.
-UseLink	If specified then site visitors will be tracked by modifying all the absolute and relative links in the current format file.
-UseNone	No links on the current page will be modified and a cookie will not be set. -UseNone allows custom session tracking to be used.
-UseAuto	This option automatically uses -UseCookie if cookies are available on the visitor's browser or -UseLink otherwise.

Note: -UseCookie is the default for [Session_Start] unless -UseLink is or -UseNone is specified. Use -UseLink to track a session using only links. Use both -UseLink and -UseCookie to track a session using both links and a cookie.

Starting a Session

The [Session_Start] tag is used to start a new session or to load an existing session. When the [Session_Start] tag is called with a given -Name parameter it first checks to see whether an ID is defined for the current visitor. The ID is searched for in the following three locations:

- **ID** – If the [Session_Start] tag has an -ID parameter then it is used as the ID for the current visitor.
- **Cookie** – If a session tracker cookie is found for the name of the session then the ID stored in the cookie is used.
- **-Session** – If a -Session command tag for the name of the session was specified in the link that loaded the current page then the parameter of that tag is used as the session ID.

The name of the session and the ID are used to check whether a session has already been created for the current visitor. If it has then the variables in the session are loaded replacing the values for any variables of the same name that are defined on the current page.

If no ID can be found, the specified ID is invalid, or if the session identified by the name and ID has expired then a new session is created.

After the [Session_Start] tag has been called the [Session_ID] tag can be used to retrieve the ID of the current session. It is guaranteed that either a valid session will be loaded or a new session will be created by the [Session_Start] tag.

Note: The [Session_Start] tag must be used on each page you want to access session variables.

Session Tracking

The session ID for the current visitor can be tracked using two different methods or a custom tracking system can be devised. The tracking system depends on what parameters are specified for the [Session_Start] tag.

- **Cookie** – The default session tracking method is using a cookie. If no other method is specified when creating a session then the -UseCookie method is used by default. The cookie will be inspected automatically when the visitor loads another page in the site which includes a [Session_Start] tag. No additional programming is required.

  The session tracking cookie is of the following form. The name of the cookie includes the words _Session_Tracker_ followed by the name given to the session in [Session_Start]. The value for the cookie is the session ID as returned by [Session_ID].

  ```
  _SessionTracker_SessionName=1234567890abcdefg
  ```

- **Links** – If the -UseLink parameter is specified in the [Session_Start] tag then Lasso will automatically modify links contained on the current page. The preferences for which links will be modified by Lasso can be adjusted in the *Setup > Global Settings > Sessions* section of Lasso Administration. See the Lasso Professional 8 Setup Guide for more information. No additional programming beyond specifying the -UseLink parameter is required.

  By default, links contained in the href parameter of ... and in the action parameter of <form action="..."> ... </form> tags will be modified.

  Links are only modified if they reference a file on the same machine as the current Web site. Any links which start with any of the following strings are not modified.

file://	ftp://	http://	https://
javascript:	mailto:	telnet://	#

Links are modified by adding a -Session command tag to the end of the link parameters. The value of the command tag is the session name followed by a colon and the session ID as returned by [Session_ID]. For example, an anchor tag referencing the current file would appear as follows after the -Session tag was added.

```
<a href="default.lasso?-Session=SessionName:1234567890abcdefg"> ... </a>
```

- **Auto** – If the -UseAuto parameter is specified in the [Session_Start] tag then Lasso will check for a cookie with an appropriate name for the current session. If the cookie is found then -UseCookie will be used to propagate the session. If the cookie cannot be found then -UseLink will be used to propagate the session. This allows a site to preferentially use cookies to propagate the session, but to fall back on links if cookies are disabled in the visitor's browser.
- **None** – If the -UseNone parameter is specified in the [Session_Start] tag then Lasso will not attempt to propagate the session. The techniques described later in this chapter for manually propagating the session must be used.

To start a session:

A session can be started using the [Session_Start] tag. The optional -Expires parameter specifies how long in minutes the session should be maintained after the last access by the site visitor. The default is 15 minutes. The optional -UseLink keyword specifies that absolute and relative links in the current format file should be modified to contain a reference to the session. The optional -UseCookie keyword specifies that a cookie should be set in the visitor's Web browser so that the session can be retrieved in subsequent pages.

The following example starts a session named Site_Preferences with an idle expiration of 24 hours (1440 minutes). The session will be tracked using both cookies and links.

```
[Session_Start: -Name='Site_Preferences', -Expires='1440', -UseLink, -UseCookie]
```

When the [Session_Start] tag is called it restores all stored variables. If a variable by the same name has already been created on the page then that variable value will be overwritten by the stored variable value.

To add variables to a session:

Use the [Session_AddVariable] tag to add a variable to a session. Once a variable has been added to a session its value will be remembered at the end of each format file in which the variable is used. Variables included in a session will be automatically defined when the [Session_Start] tag is called.

In the following example a variable RealName is added to a session named Site_Preferences.

```
[Session_AddVariable: -Name='Site_Preferences', 'Real_Name']
```

Variables will not be created by the [Session_AddVariable] tag. Each [Session_AddVariable] should be accompanied by a [Variable] tag that defines the starting value for the variable.

To remove variables from a session:

Use the [Session_RemoveVariable] tag to remove a variable from a session. The variable will no longer be stored with the session and its value will not be restored in subsequent pages. The value of the variable in the current page will not be affected. In the following example a variable RealName is removed from a session named Site_Preferences.

```
[Session_RemoveVariable: -Name='Site_Preferences', 'Real_Name']
```

To delete a session:

A session can be deleted using the [Session_End] tag with the name of the session. The session will be ended immediately. None of the variables in the session will be affected in the current page, but their values will not be restored in subsequent pages. Sessions can also end automatically if the timeout specified by the -Expires keyword is reached. In the following example the session Site_Preferences is ended.

```
[Session_End: -Name='Site_Preferences']
```

To pass a session in an HTML form:

Sessions can be added to URLs automatically using the -UseLink keyword in the [Session_Start] tag. In order to pass a session using a form a hidden input must be added explicitly. The hidden input should have the name -Session and the value Session_Name:Session_ID. In the following example, the ID for a session Site_Preferences is returned using [Session_ID] and passed explicitly in an HTML form.

```
<form action="repsonse.lasso" method="POST">
  <input type="hidden" name="-Session"
    value="Site_Preferences:[Session_ID: -Name='Site_Preferences']">
  …
  <input type="submit" name="-Nothing" value="Submit Form">
</form>
```

To track a session using links only if cookies are disabled:

The following example shows how to start a session using links if cookies are disabled. The -UseAuto parameter will first try setting a cookie and decorate the links on the current page. On subesequent page loads if the session cookie is found then it will be used and the links on the page will not be decorated. If the cookie cannot be found then links will be used to propagate the session..

```
[Session_Start: -Name=$Session_Name, --UseAuto]
```

Session Example

This example demonstrates how to use sessions to store site preferences which are persistent from page to page.

Web sites can be customized for individual visitors using sessions. In this example a site visitor is allowed to enter certain information about themselves in various forms throughout the Web site. When subsequent forms are encountered, the Web site should be able to pre-fill any elements that the visitor has already specified.

Sessions will be used to track the visitors RealName, EmailAddress, and FavoriteColor in three variables.

To create the session:

The following code will be specified at the top of every Web page in the Web site. The session must be started in every Web page which requires access to or which might modify the stored variables.

1 The [Session_Start] tag is used to start a session named Site_Preferences. The expiration of the session is set to 24 hours (1440 minutes). The session will be tracked by both links and cookies.

```
[Session_Start: -Name='Site_Preferences', -Expires='1440', -UseLink, -UseCookie]
```

2 The three variables RealName, EmailAddress, and FavoriteColor are added to the session using [Session_AddVariable].

```
[Session_AddVariable: -Name='Site_Preferences', 'RealName']
[Session_AddVariable: -Name='Site_Preferences', 'EmailAddress']
[Session_AddVariable: -Name='Site_Preferences', 'FavoriteColor']
```

3 Finally, default values are established for all three variables. RealName and EmailAddress are set to the empty string if they are not defined. FavoriteColor is set to blue #0000cc if it has not been defined. These default values will only be set the first time the session is started. In subsequent pages, the variables will automatically be set to the value stored in the session.

```
[If: (Variable_Defined: 'RealName' ) == False]
  [Variable: 'RealName' = '']
[/If]

[If: (Variable_Defined: 'EmailAddress') == False]
  [Variable: 'EmailAddress' = '']
[/If]

[If: (Variable_Defined: 'FavoriteColor') == False]
  [Variable: 'FavoriteColor' = '#0000cc']
[/If]
```

To use the session variables:

The session variables are used in each page as normal variables. Whatever value they are set to at the end of the Web page will be the value the variable has the next time the session is started.

- The FavoriteColor variable can be used to set the color of text by using it in an HTML <font> tag. In the following example, the visitors RealName will be shown in the specified color.

  ```
  <font color="[Variable: 'FavoriteColor']"> Welcome [Variable: 'RealName'] </font>
  ```

- The visitor's RealName and EmailAddress can be shown in a form by placing the variables in the HTML <input> tags. The following form allows the visitor to enter their name and email address and to select a favorite color from a pop-up menu.

  ```
  <form action="response.lasso" method="POST">
    <br>Your Name:
      <input type="text" name="RealName" value="[Variable: 'RealName']">
    <br>Your Email Address:
      <input type="text" name="EmailAddress" value="[Variable: 'EmailAddress']">
    <br>Your Favorite Color:
      <select name="FavoriteColor">
        <option value="#0000cc"> Blue </option>
        <option value="#cc0000"> Red </option>
        <option value="#009900"> Green </option>
      </select>
    <br>
      <input type="submit" name="-Nothing" value="Submit">
  </form>
  ```

  In the response page response.lasso, the form inputs can be retrieved using the [Action_Param] tag and stored into variables. These new values will now be stored with the session.

  ```
  [Variable: 'RealName' = (Action_Param: 'RealName')]
  [Variable: 'EmailAddress' = (Action_Param: 'EmailAddress')]
  [Variable: 'FavoriteColor' = (Action_Param: 'FavoriteColor')]
  ```

19

Chapter 19
Error Control

This chapter documents the methods Lasso uses to report errors and the tags available in LDML to capture and respond to errors.

- *Overview* provides definitions of the types of errors Lasso reports and the methods which can be used to capture and respond to them.
- *Error Reporting* documents the built-in error messages in Lasso and how to customize the amount of information provided to site visitors.
- *Custom Error Page* explains how to override the built-in error messages for the entire server or a single site with a custom error page.
- *Error Pages* documents how to create action specific error pages.
- *Error Tags* documents the [Error_...] process and substitution tags that can be used to report custom or standard errors and for basic error handling within a format file.
- *Error Handling* documents the [Protect], [Fail], and [Handle] tags for advanced error handling within a format file.

Overview

Responding to errors gracefully is the hallmark of good programming. Errors in Lasso run the gamut from expected errors such as a database search that returns no records to syntax errors that require fixing before a page will even process. Lasso provides tools to manage errors at several different levels which can act redundantly to ensure that no errors will be missed.

The following lists the types of errors that can occur in or are reported by Lasso. This chapter includes instructions for how to handle each of these types of errors.

Error Types

- **Web Server Errors** include file not found errors and access violations in realms. These will be reported with standard HTTP response codes, e.g. 404 for File Not Found.
- **Syntax Errors** include misspellings of tag names, missing delimiters, and mismatched data types. Lasso will return an error message rather than the processed format file if it encounters a syntax error.
- **Action Errors** include misspellings of database names, table names, or field names and other problems specifying database actions. The database action cannot be performed until the errors are corrected.
- **Action Results** can be reported as errors by Lasso. For example if no records were found after performing a search.
- **Database Errors** are generated by the data source application and include data type mismatches, missing required field values, and others. Lasso will report the error which was returned from the data source application without modification.
- **Logical Errors** are problems that cause a page to process unexpectedly even though the syntax of the code is correct. These include infinite loops, missing cases, and assumptions about the size or composition of a found set.
- **Security Violations** are not strictly errors, but are attempts to perform database actions or file accesses which are not allowed by the permissions set for the current user. These include permissions to perform database actions, privileges to add users to groups, permissions to use specific tags, and specific permissions to use the file tags.
- **Installation Problems** can also result in error messages if a Lasso Web server connector is improperly configured or Lasso Service is unavailable.
- **Operating System Errors** can also be reported by Lasso if they occur. Lasso will report the error without modification.

Some errors are more serious than others. Pages will not be processed at all if they contain syntax errors or if there are installation problems which prevent Lasso Service from being accessed. Other errors are commonly encountered in the normal use of a Web site. Most database errors and security violations are handled by simple means such as showing a No Records Found message or displaying a security dialog box to prompt the user for a username and password.

There are five mechanisms for handling errors which are detailed in this chapter. These mechanisms can be used singly or in concert to provide comprehensive error handling.

Error Control Types

- Automatic **Error Reporting** is performed by Lasso in response to unhandled errors. The amount of detail provided in these error messages can be customized by setting the error reporting level or by creating a custom server-wide error.lasso file.
- A **Custom Error Page** allows the automatic error page to be replaced by a custom page. Custom error pages are usually created for each site on a server.
- **Error Tags** allow action and logical errors and security violations to be handled within a format file.
- **Error Handling** tags allow advanced error handling to be built into format files. These techniques allow error handling routines to be built into a page without disrupting the normal processing of a page if no errors occur.

Error Reporting

For errors that occur while processing a page, Lasso displays error messages differently based on the current error reporting level. This allows detailed error messages to be displayed while developing a Web site and then for minimal or generic error messages to be displayed once a site has been deployed.

The default global error reporting level can be set in Lasso Administration in the Setup > Global > Settings section. The error reporting level can be set to None, Minimal, or Full. Each of these levels is described in more detail below.

The error reporting level for a particular page can be modified using the [Lasso_ErrorReporting] tag with a value of None, Minimal, or Full. This will modify the error reporting level only for the current format file and its includes without affecting the global default. See the section on the [Lasso_ErrorReporting] tag below for additional details.

No matter what level of error reporting has been specified, the standard built-in error message will be replaced by a custom error page if one is defined. See the following section *Custom Error Page* for more details.

Error Levels

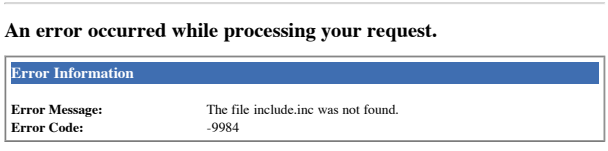
This section describes how error messages are formatted at each of the three error reporting levels:

- **None** – This level provides only a generic error message with no specific information or error code. This level can be used on a deployment server when it is desirable to provide no specific information to the site visitor. When the error page is displayed the full error code is logged as a detail error message.

Figure 1: Built-In None Error Message

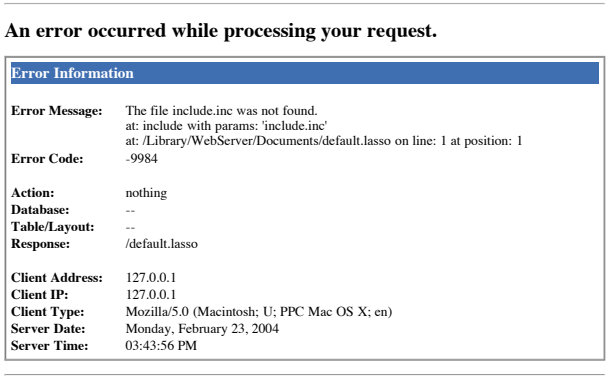
- **Minimal** – This level is the default. It provides a minimal error message and error code. No context about where the error occurred is provided. This level can be used on a deployment server in order to make troubleshooting problems easier. When the error page is displayed the full error code is logged as a detail error message.

Figure 2: Built-In Minimal Error Message



- **Full** – This level provides detailed error messages for debugging and troubleshooting. The path to the current format file is provided along with information about what files have been included and what parameters have been passed to them. If a database or action error is reported, the built-in error message provides information about what database action was performed when the error occurred.

Figure 3: Built-In Full Error Message



An error occurred while processing your request.

Error Information

Error Message:	The file include.inc was not found. at: include with params: 'include.inc' at: /Library/WebServer/Documents/default.lasso on line: 1 at position: 1
Error Code:	-9984
Action:	nothing
Database:	--
Table/Layout:	--
Response:	/default.lasso
Client Address:	127.0.0.1
Client IP:	127.0.0.1
Client Type:	Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en)
Server Date:	Monday, February 23, 2004
Server Time:	03:43:56 PM

Setting the Error Level

The error reporting level can be set for an individual format file by specifying the [Lasso_ErrorReporting] tag with the desired error level at the top of the page. If the -Local keyword is used within the tag then the error level will only be changed within the current included file, custom tag, or processed code.

Table 1: Error Level Tag

Tag	Description
[Lasso_ErrorReporting]	Sets the error reporting level for the current page to 'None', 'Minimal', or 'Full'. Defaults to the value set in Lasso Administration. An optional -Local keyword modifies the error level for only the current context.

To set the error reporting level within a format file:

Use the [Lasso_ErrorReporting] tag with the desired error reporting level. For example, the following code sets the error reporting level to Full so the current format file can be more easily debugged.

```
[Lasso_ErrorReporting: 'Full']
```

To set the error reporting level within a local context:

Use the [Lasso_ErrorReporting] tag with the -Local keyword and the desired error reporting level. For example, the following code sets the error reporting level to None so no errors are reported from the current include.

This error reporting level will only be in effect until the end of the current include, custom tag, or process tag.

```
[Lasso_ErrorReporting: 'None', -Local]
```

Other Errors

The simple error message in *Figure 2: Lasso Service Error Message* is displayed when Lasso Service cannot be contacted by a Lasso Web server connector. No processing can happen without Lasso Service. This message will be displayed if Lasso Service is quit or restarted while the Web server application is still running.

Figure 4: Lasso Service Error Message

Lasso Error

Lasso Connector could not communicate with Lasso Service.

Security violations result in an appropriate HTTP response being sent to the Web client to ask the site visitor for authentication information. An authentication dialog like that shown in *Figure 3: Authentication Dialog* is presented to the visitor. If they enter a valid username and password then processing proceeds as normal. If they enter an invalid username and password then the standard built-in error message will be shown with details about the security violation.

Figure 5: Authentication Dialog

Custom Error Page

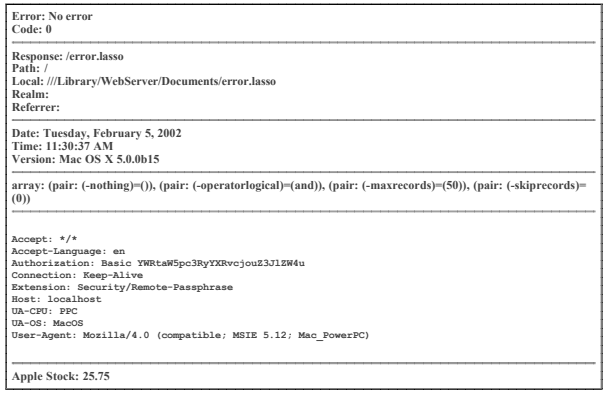
A custom error page can be defined which will be displayed to the site visitor rather than the built-in error message described in the previous section. The error message displayed on a custom error page will depend

on the error reporting level which is set in Lasso Administration and on the current page. However, the rest of the information on the custom error page is determined by the LDML used to code the page.

There are three ways to define a custom error page.:

- **Host** – The custom error page for a particular host can be modified by creating a file named error.lasso and placing it in the root of the Web serving folder. Each virtual host which has a distinct Web serving folder can have a custom error page.
- **Site** – The site-wide error page is named error.lasso and is located in the Lasso Professional 8 site folder. Modifying this page will alter the error page for all hosts that map to the site. By modifying this page it is possible to add logging or even email notification when errors occur on a hosted site.
- **Server** – The server-wide error page is named error.lasso and is located in the Admin folder in the Lasso Professional 8 application folder. Modifying this page will alter the error page for all sites that are hosted on their server. The server-wide error page will only be used for sites that do not have a site-specific error page defined.

Figure 6: Custom Error Page



To define a custom server-wide error page.

Modify the file error.lasso which is located in the Admin folder of the Lasso Professional 8 application folder. This file should be kept simple and be thoroughly debugged before it is used on a production server.

To define a custom error page:

1 Create a file named error.lasso which includes the default error message to be displayed to site visitors.

2 All image links and URLs within the custom error page should be specified as absolute paths from the root of the Web serving folder. The following <img> tag contains a reference to picture.gif contained in the images folder.

```
<img src="/Images/picture.gif">
```

3 Place the error.lasso file in the root of the Web serving folder for the Web site which is being customized. The file should be accessible by loading the following URL.

```
http://www.example.com/error.lasso
```

Note: The built-in error page will not be displayed if a custom error page is defined. The values of [Error_CurrentError] and [Error_CurrentError: -Errorcode] should be reported in some way by the custom error page.

To test a custom error page:

A properly placed error.lasso file can be tested by loading it with each of the following URLs.

- The first URL loads the page directly. This confirms that the error.lasso file is located in the right folder.

  ```
  http://www.example.com/error.lasso
  ```

- The second URL will cause an error in Lasso that should return the custom error page. A page not found error will be returned since the file fakepage.lasso is not present on the Web server.

  ```
  http://www.example.com/Action.Lasso?-response=fakepage.lasso
  ```

Error Pages

A custom error page can be specified in any HTML form or URL based Lasso action using the -ResponseAnyError command tag. The -ResponseRequiredFieldMissingError tag can be used to trap for missing values which are flagged with the -Required command tag. The -ResponseSecurityError can be used to trap for security permissions violations.

If an error occurs and no -Response... tag is specified then the default error message or a custom error page is returned as documented in the previous section *Custom Error Page*. The details of the Lasso action can be retrieved in the error page and the specific error message which triggered the error page can be returned using [Error_CurrentError].

Neither of the response command tags function within [Inline] ... [/Inline] based Lasso actions. Instead, errors should be handled directly within the [Inline] ... [/Inline] tags using the techniques outlined in the *Error Tags* and *Error Handling* sections that follow.

Table 2: Error Response Tags

Tag	Description
-ResponseAnyError	Specifies the page to return if any error occurs and no specific error page for that error is specified.
-ResponseReqFieldMissingError	Specifies the page to return if a name/value pair preceded by a -Required command tag does not have a value. Synonyms include -ResponseRequiredFieldMissingError, -ResponseReqColumnMissingError, and -ResponseRequiredColumnMissingError.
-ResponseSecurityError	Specifies the page to return if the current user does not have permission to perform the requested action.

Error Tags

The [Error_...] tags in LDML allow custom errors to be reported and provide access to the most recent error that was reported by the code executing in the current format file. This allows the developer to check for specific errors and respond if necessary with an error message or code to correct the error.

Lasso maintains a single error code and error message that is set by any tag which reports an error. The error code and error message should be checked immediately after a tag that may report an error. If any intervening tags report errors then the error code and error message will be lost.

Custom errors can be created using the [Error_SetErrorMessage] and [Error_SetErrorCode] tags. Once set, the [Error_CurrentError] tag or [Error_Code] and [Error_Msg] tags will return the custom error code and message. A developer can utilize these tags to incorporate both built-in and custom error codes into the error recovery mechanisms for a site.

Table 3: Error Tags

Tag	Description
[Error_CurrentError]	Returns the current error message. Optional -ErrorCode parameter returns the current error code.
[Error_Code]	Returns the current error code.
[Error_Msg]	Returns the current error message.
[Error_SetErrorCode]	Sets the current error code to a custom value.
[Error_SetErrorMessage]	Sets the current error message to a custom value.

To display the current error in a format file:

- Use the [Error_Msg] tag and the [Error_Code] tag. The following code will display a short error message.

```
The current error is [Error_Code]: [Error_Msg].
```

If the code on the page is executing normally and there is no current error to report then the code will return.

```
➜ The current error is 0: No Error.
```

- Use the [Error_CurrentError] tag with the optional -ErrorCode keyword. The following code will display a short error message.

```
The current error is [Error_CurrentError: -ErrorCode]: [Error_CurrentError].
```

If the code on the page is executing normally and there is no current error to report then the code will return.

```
➜ The current error is 0: No Error.
```

To set the current error in a format file:

The current error code and message can be set using the [Error_SetErrorCode] and [Error_SetErrorMessage] tags. These tags will not affect the execution of the current format file, but will simply set the current error so it will be returned by the [Error_CurrentError] tag.

In the following example, the error message is set to A custom error occurred and the error code is set to -1.

```
[Error_SetErrorMessage: 'A custom error occurred']
[Error_SetErrorCode: -1]
```

The [Error_CurrentError] tag now reports this custom error when it is called later in the page, unless any intervening code changed the error message again.

```
The current error is [Error_CurrentError: -ErrorCode]: [Error_CurrentError].
```

```
➜ The current error is -1: A custom error occurred.
```

The remainder of the [Error_...] tags provide shortcuts for reporting standard errors or checking what error is being reported by Lasso so appropriate steps can be taken. The [Error_...] tags available in LDML are described in *Table 3: Error Type Tag*. An example of how to respond to a particular error message follows.

These tags can be used with the [Error_SetErrorCode] and [Error_SetErrorMessage] tags to generate standard errors. If a page has code which deals with an "Add Error" for example, that code can be triggered by an [Inline] that reports an "Add Error" or by setting the current error to an "Add Error" explicitly using the [Error_SetErrrorCode] and [Error_SetErrorMessage] tags as shown in the following code.

```
[Error_SetErrorCode: (Error_AddError: -ErrorCode)]
[Error_SetErrorMessage: (Error_AddError)]
```

Table 4: Error Type Tags

Tag	Description
[Error_AddError]	An error occurred during an -Add action.
[Error_DatabaseConnection Unavailable]	A connection to the specified Lasso data source connector for the current database cannot be established.
[Error_DatabaseTimeout]	The connection to the Lasso data source connector timed out.
[Error_DeleteError]	An error occurred during a -Delete action such as if an invalid -KeyField or -KeyValue was specified.
[Error_FieldRestriction]	An error reported by the Lasso data source connector that a field cannot be modified. Synonym is [Error_ ColumnRestriction].
[Error_FileNotFound]	The specified file in an [Include] tag or -Response… tag cannot be found.
[Error_InvalidDatabase]	The specified database is not configured within Lasso Administration.
[Error_InvalidPassword]	The password for the specified username is invalid.
[Error_InvalidUsername]	The specified username cannot be found in the users database within Lasso security.
[Error_NoError]	The code has been executed successfully. This error code represents the lack of an error.
[Error_NoPermission]	The current user does not have permission to perform the requested database action.
[Error_OutOfMemory]	Lasso encountered an internal out of memory error that prevents the current page from processing.

[Error_RequiredFieldMissing]	A value was not specified for an HTML form or URL parameter preceded by a -Required command tag. Also [Error_RequiredColumnMissing].
[Error_UpdateError]	An error occurred during an -Update action such as if an invalid -KeyField or -KeyValue was specified.

Note: In prior versions of Lasso an [Error_NoRecordsFound] tag was defined. This tag has been deprecated in favor of checking whether the [Found_Count] is equal to zero to check if no records were found.

To check for a specific error within [Inline] … [/Inline] tags:

Use a conditional expression in [If] ... [/If] tags to compare [Error_CurrentError] with the specific error type tag you want to check. In the following example, a different message is displayed if no records were found after a -FindAll action or if the requested database was not found.

```
[Inline: -Database='Contacts', -Table='People', -FindAll]
  [If: (Error_CurrentError) == (Error_InvalidDatabase)]
    The database Contacts is not valid.
  [Else: (Error_CurrentError) == (Error_NoPermission)]
    You don't have permission to search Contacts.
  [Else: (Found_Count) == 0]
    No records were found in Contacts.
  [Else]
    … Display Found Set Here …
  [/If]
[/Inline]
```

Error Handling

LDML includes powerful error handling tags that allow areas of a page to be protected. Error-specific handlers are called if any errors occur in a protected area of a page. These tags allow comprehensive error handling to be built into a page without disturbing the code of the page with many conditionals and special cases.

Table 5: Error Handling Tags

Tag	Description
[Fail]	Halts execution of the current page or [Protect] … [/Protect] block. Takes two parameters: an integer error code and a string error message.

[Fail_If]	Conditionally halts execution of the current page or [Protect] … [/Protect] block. Takes three parameters: a conditional expression, an integer error code, and a string error message.
[Handle] ... [/Handle]	Conditionally executes after the code in the current container tag or format file is completed or a [Fail] tag is called. Takes a conditional expression as a parameter.
[Handle_Error] ... [/Handle_Error]	Functions the same as [Handle] … [/Handle] except that the contents are executed only if an error was reported in the surrounding [Protect] … [/Protect] tags.
[Protect] ... [/Protect]	Container tag that protects a portion of a page. If code inside the container throws an error or a [Fail] tag is executed inside the container then the error is not allowed to propagate outside the protected block.

Handle Tags

The [Handle] ... [/Handle] tags are used to surround a block of code that will be executed after the current code segment is completed. The opening [Handle] tag takes a single parameter which is a conditional expression. If the conditional expression returns True then the code in the [Handle] ... [/Handle] tags is executed. Every [Handle] tag is given a chance to execute in the order they were specified so multiple [Handle] ... [/Handle] tags can be executed.

[Handle] ... [/Handle] tags will not be executed if a syntax error occurs while Lasso is parsing a page. When Lasso encounters a syntax error it returns an error page instead of processing the code on a page.

[Handle] ... [/Handle] tags will be executed if a logical error occurs while Lasso is processing a page. However, the result of the page will be an error message rather than the output of the page. Code within the [Handle] ... [/Handle] tags can redirect the user to another page using [Redirect_URL] or can replace the contents of the page being served.

There are two ways to use [Handle] ... [/Handle] tags within a format file:

- When used on their own in a format file, the code inside the [Handle] ... [/Handle] tags will be conditionally executed after all the rest of the code in the format file has completed. [Handle] ... [/Handle] tags can be used to provide post-processing code for a format file.
- When used within any LDML container tag, the code inside the [Handle] ... [/Handle] tags will be conditionally executed after the closing container tag. [Handle] ... [/Handle] tags will most commonly be used within [Protect] ... [/Protect] tags to provide error handling.

To specify code to execute if a format file reports an error:

Place [Handle] ... [/Handle] tags with a check for [Error_CurrentError] anywhere in a page, but not inside any other container tags. In the following example, the opening [Handle] tag checks if [Error_CurrentError] is not equal to [Error_NoError]. The contents of the page which is being returned to the visitor is replaced by a custom error message if an error has occurred.

```
[Handle: (Error_CurrentError) != (Error_NoError)]
  [Var: '__html_reply__' = '<hr>' +
    'An error occurred while processing this page:' +
    (Error_CurrentError: -ErrorCode) + ': ' + (Error_CurrentError) + '.']
[/Handle]
```

To output debugging messages at the end of a format file:

Place [Handle] ... [/Handle] tags throughout a page that check to see if a variable named Debug equals True. The contents of the [Handle] ... [/Handle] tags will only be executed if it does. Note that the [Handle] ... [/Handle] tags can only contain static messages because they do not execute within the flow of the page.

```
[Var: 'Debug'=True]

[Handle: (Variable: 'Debug') == True]
  <p>Debugging Message
[/Handle]
```

Note: If a syntax or logical error occurs while processing the page then this handle code will execute, but the results may not be visible since the default error page will be returned in place of the processed page contents.

To specify code to post-process a format file:

Place [Handle] ... [/Handle] tags with a condition of True anywhere in the format file, but not within any other container tags. The contents of the [Handle] ... [/Handle] will execute after the rest of the format file has executed.

In the following example, the global variable which contains the text of the page which will be sent to the site visitor __html_reply__ is modified using [String_ReplaceRegExp] so that all occurrences of the words OmniPilot are wrapped with <font> tags that make them blue.

```
<?LassoScript
  // This LassoScript implements a post-processor that makes all occurrences
  // of the words OmniPilot within the current format file blue.
  Handle: True; // Unconditionally execute handler.
    Variable: '__html_reply__' = (String_ReplaceRegExp: $__html_reply__,
```

```
        -Find='([Bb]lue +[Ww]orld)',
        -Replace='<font color="blue">\1</font>');
    /Handle;
?>
```

Fail Tags

The [Fail] tag allows an error to be triggered from within LDML code. The two parameters of the tag are the integer error code and string error message of the error to be reported. Use of the [Fail] tag immediately halts execution of the current page and starts execution of any [Handle] ... [/Handle] tags contained within.

The [Fail] tag can be used in the following ways:

- To report an unrecoverable error. Just as Lasso automatically halts execution of a format file when a syntax error or internal error is encountered, LDML code can use the [Fail] tag to report an error which cannot be recovered from.

  ```
  [Fail: -1, 'An unrecoverable error occurred']
  ```

- To trigger immediate execution of the page's [Handle] ... [/Handle] tags. If an error is handled by one of the [Handle] ... [/Handle] tags specified in the format file (outside of any other container tags) then the code within the [Handle] ... [/Handle] tags will be executed.
- To trigger immediate execution of a [Protect] ... [/Protect] block's [Handle] ... [/Handle] tags. See the next section *Protect Tags* for details.

To report a standard Lasso error:

Use the appropriate [Error_...] tag to return the error code and error message for any of Lasso's standard errors. In the following example a No Records Found error is triggered.

```
[Fail: (Error_NoRecordsFound: -ErrorCode), (Error_NoRecordsFound)]
```

To conditionally execute a [Fail] tag:

[Fail_If] allows conditional execution of a [Fail] without using a full [If] ... [/If] tag. The first parameter to [Fail_If] is a conditional expression. The last two parameters are the same integer error code and string error message as in the [Fail] tag. In the following example the [Fail_If] tag is only executed if the [Found_Count] is 0.

```
[Fail_If: (Found_Count == 0),
  (Error_NoRecordsFound: -ErrorCode), (Error_NoRecordsFound)]
```

Protect Tags

The [Protect] ... [/Protect] tags are used to catch any errors that occur within the code surrounded by the container tags. They create a protected environment from which errors cannot propagate to the page itself. Even if an internal error is reported by Lasso it will be caught by the [Protect] ... [/Protect] tags allowing the rest of the page to execute successfully.

Any [Fail] or [Fail_If] tags called within [Protect] ... [/Protect] tags will halt execution only if the code is contained within the [Protect] ... [/Protect] tags. Any [Handle] ... [/Handle] tags contained within the [Protect] ... [/Protect] tags will be conditionally executed. The format file will continue executing normally after the closing [/Protect] tag.

The [Protect] ... [/Protect] tags can be used for the following purposes:

- To protect a portion of a page so that any errors that would normally result in an error message being displayed to the user are instead handled in the internal [Handle] ... [/Handle] tags.
- To provide advanced flow control in a page. Code within the [Protect] ... [/Protect] tags is executed normally until a [Fail] tag is encountered. The code then jumps immediately to the internal [Handle] ... [/Handle] tags.

To protect a portion of a page from logical errors:

Wrap the portion of the page that needs to be protected in [Protect] ... [/Protect] tags. Any internal errors that Lasso reports will be caught by the [Protect] ... [/Protect] tags and not reported to the end user. [Handle] ... [/Handle] should be included to handle the error if necessary.

In the following LassoScript an attempt is made to set the global map [Tags] to Null. This would have the effect of removing all tags from LDML so their operation is not allowed. Instead, Lasso reports a logical error. Since this code is executed within [Protect] ... [/Protect] tags no error is reported, but the [Protect] ... [/Protect] tags exit silently and the format file resumes executing after the end of the LassoScript.

```
<?LassoScript
  Protect;
    $Tags = Null;
  /Protect;
?>
```

To use the [Protect] ... [/Protect] tags with custom errors:

The following example shows [Protect] ... [/Protect] tags which surround code that contains several [Fail_If] statements with custom error codes -1 and -2. A pair of [Handle] ... [/Handle] tags inside the [Protect] ... [/Protect] tags are set to

intercept either of these custom error codes. These [Handle] ... [/Handle] tags will only execute if one of the [Fail_If] tags executes successfully.

```
[Protect]
  …
  [Fail_If: ($ConditionOne == True), -1, 'Custom error -1']
  …
  [Fail_If: ($ConditionTwo == True), -2, 'Custom error -2']
  …
  [Handle: ((Error_CurrentError: -ErrorCode) == -1)]
    … Handle custom error -1 …
  [/Handle]
  [Handle: (Error_CurrentError: -ErrorCode) == -2)]
    … Handle custom error -2 …
  [/Handle]
[/Protect]
```

IV

Section IV
Upgrading

This section contains detailed instructions for developers who are upgrading solutions developed using a previous version of Lasso to Lasso Professional 8.

- *Chapter 20: Upgrading From Lasso Professional 7 or 8* includes complete instructions for upgrading solutions that were built using Lasso Professional 7 for compatibility with Lasso Professional 8. This chapter also includes details about changes which were made between updates of Lasso Professional 8.
- *Chapter 21: Upgrading From Lasso Professional 6* includes instructions for upgrading solutions that were built using Lasso Professional 6 for compatibility with Lasso Professional 8. This chapter should be read in concert with the previous chapter for a complete list of changes between Lasso Professional 8 and Lasso Professional 6.
- *Chapter 22: Upgrading From Lasso Professional 5* includes instructions for upgrading solutions that were built using Lasso Professional 5 for compatibility with Lasso Professional 7. This chapter should be read in concert with the previous two chapters for a complete lists of changes between Lasso Professional 8 and Lasso Professional 5.
- *Chapter 23: Upgrading From Lasso WDE 3.x* includes instructions for upgrading solutions that were built using Lasso Web Data Engine 3.x for compatibility with Lasso Professional 8. This chapter should be read in concert with the three previous chapters for a complete list of changes between Lasso Professional 8 and Lasso Web Data Engine 3.x

This section should be read in concert the upgrading instructions in the Lasso Professional 8 Setup Guide.

20

Chapter 20
Upgrading From Lasso Professional 7 or 8

This chapter contains important information for users of Lasso Professional 7 or 8 who are upgrading to the latest version of Lasso Professional 8.

If upgrading from an earlier version of Lasso, this chapter should be read in conjunction with the subsequent chapters on upgrading from Lasso Professional 6, Lasso Professional 5, or Lasso Web Data Engine 3.x or earlier.

Topics in this chapter include:

- *Lasso Professional 8.0.x* includes information about important changes in the Lasso Professional 8 product updates..
- *Introduction* includes general information about what has changed in Lasso Professional 8.
- *Security Enhancements* describes updates to file extensions, Classic Lasso, and file permissions security in Lasso Professional Server 8.0.1.
- *SQLite* introduces the built-in SQLite data source.
- *Multi-Site* introduces the new multi-site features.
- *Digest Authentication* introduces the new digest authentication for Web browsers.
- *On-Demand LassoApps* explains how the files from the Lasso folder in the Web server root are now served on-demand instead.
- *Syntax Changes* contains information about what LDML syntax constructs have changed since Lasso Professional 7.

- *Tag Name Changes* details the tag names which have been changed in LDML 8 since LDML 7.

Lasso Professional 8.0.x

This section summarizes the changes that have been made to Lasso in each product update for Lasso Professional 8.

The following changes were introduced in Lasso Professional 8.0.2:

- **File Security** – The security model for the [File] tags has been modified. The paths available to each site administrator are now determined by the server administrator. The site administrator can assign permission to the site's groups for only the paths that have been assigned to the site. All users now have read permission for files in the Web server root.
- **[Bytes] Tag** – The [Bytes] tag now accepts an optional second parameter which specifies in what character set the string should be imported. See the section below for code examples.
- **Accept-Charset Header** – Lasso will now obey the Accept-Charset header which is sent by most browsers. Lasso will use this header and any included quality parameters to determine the ideal character set for the current page.
- **Content-Type Header in Forms** – Lasso will now obey the charset parameter in the Content-Type header of a form submission if specifed. Most browsers do not currently send this header, but Lasso will obey it if it is present. In the future this will help to guarantee that Lasso reads incoming form data in the proper character set.
- **-ContentType in Forms** – If a hidden input is named -ContentType in an HTML form then the subsequent parameter will be imported into Lasso encoded using the specified characters set. See the section below for code examples.
- **Storing Bytes in SQLite** – Lasso now allows byte streams to be stored in the internal SQLite data source. See the note that follows for full details about to format a -SQL statement to store bytes. The [Encode_Hex] and [Decode_Hex] tags were added to facilitate this ability.
- **Email Tags** – [Email_Parse->RawHeaders] will return the raw headers from an email message. [Email_Parse] now returns a simple version of the email message when cast to string. [Decode_QuotedPrintable] and [Decode_QHeader] tags have been added. [Email_Extract] and [Email_SafeEmail] tags have been added to extract the data or comment from email headers and to return obscured email addresses.

The following changes were introduced in Lasso Professional 8.0.1:

- **Quoted Inline Parameters** – The -ReturnField , -SortField, and -KeyField inline parameters are now added to the generated SQL statement using quotes. In addition, field names that contain certain characters such as --, #, or ` will be quoted. This makes SQL injection attacks more difficult, but also prevents the specification of SQL functions as return values without using a -SQL parameter.
- **File Extensions** – Lasso's file extensions settings have been split into two sets. One set controls which files Lasso will execute as LDML pages. The second set controls which files can be accessed through Lasso's file tags.
- **Classic Lasso** – Lasso now has an option to completely disable Classic Lasso syntax including the -Response and -Error pages. A new minimal setting allows a minimal set of non-database related Classic Lasso tags to be used.
- **File Permissions** – Lasso now has a permissions to allow files with any file extensions to be manipulated using the file extensions. This permissions can be assigned on a per-group basis within a given file root.

File Security

Several changes have been made to the file security model for Lasso Professional 8.0.2. These changes were made in order to restrict the access that site administrators and users had to files outside of their own Web server root.

- Server administration has been modified with a new *Setup > Sites > File Paths* section that allows file paths to be assigned to a site. The site administrator will only be able to modify files contained in a path assigned to the site.
- New sites will have the Web server root / and the file uploads path assigned by default. This allows site administrators to access and assign permissions for only the files within their Web server root and for them to access uploaded files. Additional paths can be assigned to the site if necessary.
- Existing Lasso Professional 8/8.0.1 sites which are upgraded to 8.0.2 will have permission to the file system root assigned to them. This will allow existing code to run on upgraded sites even if it accessed paths outside of the Web server root. If access to files outside of the Web server root is not desired then the paths should be modified in the *Setup > Sites > File Paths* section of server administration after upgrading.

- Site administrators formerly had access to any files in the file system. After upgrading to Lasso Professional 8.0.2 each site administrator will only have access to files within the file paths assigned to them in servera dministration. Site administrators will still have access to files with any (or no) file extensions.
- The Setup > Security > Files section of site administration has been modified to only allow those paths that have been assigned to the site to be modifed. Permissions can be assigned for a group to the existing file paths, but new file paths cannot be designated.
- Default permission has been granted to all users to read or inspect files within the Web server root. Since files were already accessible through the [Include_Raw] tag it did not make sense to disallow the use of the [File] tags to read the same files.

[Bytes] Tag

The [Bytes] tag now accepts an optional second parameter which specifies in what character set the string should be imported. For example, the following tag will result in a byte stream that contains the example string encoded in the iso-8859-1 character set.

```
[Bytes: 'testing emigré', 'iso-8859-1']
```

This can be useful for using different encoding styles with the [Encode_URL] tag. The following tag outputs a Unicode representation of the example string. Notice that the é character ends up as a two byte sequence. (The space is encoded using a single space since it is part of the base ASCII set common to most Western character sets).

```
[Encode_URL: 'testing emigré'] ➜ testing%20emigr%C3%A9
```

However, if the [Bytes] tag is used the URL can be encoded using iso-8859-1 single byte encoding instead. Now the é character is represented by a single byte sequence. This can be useful for communicating with servers that have not been updated to recognize Unicode encoding.

```
[Encode_URL: (Bytes: 'testing emigré', 'iso-8859-1')] ➜ testing%20emigr%E9
```

Prior to Lasso Professional 8.0.2 the following code can be used to import a string into a byte stream similarly. For example this code results in the same output as the example immediatly above.

```
[Var: 'Bytes' = (Bytes)]
[$Bytes->(ImportString: 'testing emigré', 'iso-8859-1')]
[Encode_URL: $Bytes]
```

Accept-Charset Header

Lasso will now obey the Accept-Charset header which is sent by most browsers with HTTP Web requests. An example of this header is shown below. This header specifies that UTF-8 encoding is preferred, followed by ISO-8859-1 encoding, or any encoding.

```
Accept-Encoding: utf-8;q=1.0, iso-8859-1;q=0.5, *;q=0
```

Lasso processes all of its pages in Unicode internally and decides what character set to translate a page to just before it is served. Lasso will now send the page using the highest quality requested character set which it supports.

If a [Content_Type] tag is included in a page it will override the browsers Accept-Charset header field. Otherwise, the default if no character set is preferred (which is the case in the vast majority of Web requests) is to use the character set specified in Lasso Site Administration. Lasso defaults to UTF-8 encoding if no other character set has been specified.

Note: Lasso will not encode pages using gzip or deflate encoding even if those encodings are listed as preferred in the Accept-Encoding header.

```
Accept-Encoding: gzip, deflate;q=1.0, *;q=0
```

Content-Type Headers in Forms

Lasso will now interpret incoming form data according to an included Content-Type header with a charset parameter. Lasso translates all incoming form data to Unicode for internal processing. This header will allow Lasso to use the proper character set even if it differs from Lasso's default.

In the absense of a Content-Type header Lasso will interpret all incoming form data according to the default character set which is set in Lasso Site Administration. This character set is set to UTF-8 by default so incoming form data will generally be interpreted as UTF-8 if the setting has not been changed.

See also the -ContentType parameter described in the next section that can be specified explicitly to over-ride the character set for individual form element.

Note: Most browsers do not currently set the Content-Type header so most incoming forms will be interpreted using the default character set.

-ContentType in Forms

Lasso reads data which is posted in forms according to the default character set that is set in Lasso Administration (or in the character set included in the Content-Type header). However, Web browsers usually send forms using the same encoding with which the enclosing page was sent. If these character sets are not matched (for example if the [Content_Type] tag is used to override the default encoding for a particular page) then Lasso can misinterpret the data being posted by a Web client.

Lasso Professional 8.0.2 introducts a new hidden input named -ContentType. If a hidden input is named -ContentType in an HTML form then the subsequent parameter will be imported into Lasso encoded using the specified characters set.

The value for -ContentType should be specified as charset=iso-8859-1 (or any other valid character set) as shown in the example below. The charset= part is required. It is not sufficient to just put the character set in as the value.

```
<input type="hidden" name="-ContentType" value="charset=iso-8859-1" />
<input type="hidden" name="Field Name" value="testing emigré" />
```

This will result in the Field Name input being imported into Lasso using the iso-8859-1 character set. In order to change the character set for every input in a form it is necessary to place a hidden input before each text input, text area, checkbox, select, etc. on the form.

Note: The value from [Action_Param] and [Action_Params] will be returned using the proper character set. However, the values from [Client_GetParams] and [Client_GetParams] (and the args equivalents) will use the default character set.

Storing Bytes in SQLite

The internal SQLite data source allows binary data to be stored in any field using the following syntax. This syntax can only be specified within a SQL statement. The data is expected to be encoded in hexadecimal using the [Encode_Hex] tag.

```
INSERT INTO table (field) VALUES (x" ... HEX DATA ...");
```

When Lasso retrieves data from the field it will be automatically decoded into a byte stream. It is not necessary to use [Decode_Hex] on the return value from the [Field] tag.

For example, the following [Inline] would insert a byte stream into a SQLite table.

```
[Var: 'bytes' = (Bytes: ' ... Byte Stream ... ')]
[Inline: -Database='Example', -Table='Example',
    -SQL='INSERT INTO example (field) VALUES (x'" + (Encode_Hex: $bytes) + "');']
[/Inline]
```

Then the following code can be used to retrieve the value from the database. The result in the variable $bytes will be a byte stream that exactly matches the value that was stored.

```
[Inline: -Database='Example', -Table='Example', -FindAll]
  [Var: 'bytes' = (Field: 'field')]
[/Inline]

[Var: 'bytes'] ➜ ... Byte Stream ...
```

Email Tags

A number of new tags have been added to facilitate parsing and displaying email messages.

- [Email_Parse] is not a new tag, but now returns a simple version of the email message when cast to string. This makes it easier to display downloaded email messages in a simple format. Email messages are formatted with the headers shown in the following example followed by the default body. Any headers that are empty are not included. The To, Cc, and From headers are displayed using the [Email_SafeEmail] tag.

  ```
  Date: 3/11/2005 12:34:56
  From: example
  To: example
  Cc: example
  Subject: This is an example email message
  Content-Type: multipart/alternative
  Content-Transfer-Encoding: 8bit
  Content-Disposition: Attachment
  Parts: 4
  ```

- [Email_Parse->RawHeaders] will return the raw headers from an email message. This allows the unparsed headers to be fetched without manually parsing the raw source of the email message.
- [Decode_QuotedPrintable] is a new tag that decodes data which is encoded in quoted-printable format. This tag is used internally by the [Email_Parse] type to decode the bodies of messages.
- [Decode_QHeader] is a new tag that decodes email headers which are encoded in Q (quoted-printable) format. This tag is used internally by the [Email_parse] tag to decode the headers of messages.

- [Email_Extract] is a new tag which allows the different parts of email headers to be extracted. Email headers which contain email addresses are often formatted in one of the three formats below.

```
john@example.com
"John Doe" <john@example.com>
john@example.com (John Doe)
```

  In all three of these cases the [Email_Extract] tag will return john@example.com. The angle brackets in the second example identify the email address as the important part of the header. The parentheses in the third example identify that portion of the header as a comment.

  If [Email_Extract] is called with the optional -Comment parameter then it will return john@example.com for the first example and John Doe for the two following examples.

  Note: The [Email_Parse->Header] tag accepts a -Extract parameter to return just the email address portion of a header or a -Comment parameter to return the comment portion of a header.

- [Email_SafeEmail] is a new tag which returns an obscured email address. This tag can be used to safely display email headers on the Web without attracting email address harvesters.

  If the input contains a comment then it is returned. Otherwise, the full header is returned. In either case, if the output contains an @ symbol then only the portion of the address before the symbol is returned. This would result in the following output for the example headers above.

```
➜ john
  John Doe
  John Doe
```

  Note: The [Email_Parse->Header] tag accepts a -SafeEmail parameter that automatically applies the [Email_SafeEmail] tag to the returned header.

Quoted Inline Parameters

To enhance security, column names are now automatically quoted in all non-SQL [Inline] ... [/inline] operations to MySQL datasources. In addition, column names containing --, #, or ` will end at those strings. This change affects the following inline parameters: -KeyField, -ReturnField, and -SortField, as well as their synonyms. Additionally, column names specified in the inline, for adding, updating and searching will be affected.

Note: This change was introduced in Lasso Professional 8.0.1.

File Extensions

The allowed extensions have been split into two sets. Lasso Page Extensions now controls what files Lasso will execute through URLs and the [Include] and [Library] tags. File Tags Extensions controls what files Lasso can manipulate through the file tags, image tags, PDF tags, and [Include_Raw]. Both sets of extensions can be controlled through the *Setup > Site > File* Extensions section of Site Administration.

By default the Lasso page extensions are (.lasso .lassoapp .las .htm .html .inc .incl). By default the file tags extensions are (.bmp .cmyk .gif .jpg .pdf .png .psd .rgb .text .tif .txt .uld .wsdl .xml .xsd). For best security these two sets of extensions should remain mutually exclusive. Adding .* to either set will allow all file extensions for that set.

Upgraded servers will start with the default set of Lasso page extensions. The file tag extensions will be set to the complete list of format file extensions that were already set. For best results the file tag extensions should be reset using Reset Extensions in Site Admin.

This update may require some changes to your Lasso pages if you allow third parties to upload .lasso or .html files or if you use -Response to return image, PDF, or XML files. The best solution is to change your code so that users only upload files in the file tags extensions (and aren't allowed to upload any files through Lasso which can be executed by Lasso). Any URL that uses -Response to reference an image, PDF, or XML file can be rewritten to use a straight URL referencing the appropriate file.

Classic Lasso

Classic Lasso now has three options to determine whether it is enabled or not.

- **Enabled** means that all Classic Lasso URL parameters can be used including database actions. This setting can be used for compatibility with earlier versions of Lasso, but is not recommended for new code.
- **Minimal** means that database actions are disabled, but -Response and -Error parameters can still be used in URLs. This setting is the equivalent of "disabling" Classic Lasso in earlier versions of Lasso.
- **Disabled** means that no Classic Lasso URL parameters can be used and that even -Response and the -Error tags are disabled. This is the preferred setting for new Lasso installations.

The new options can be controlled through the *Setup > Site > Settings* section of Site Administration.

File Permissions

File Permissions now has an option to allow users in a group to access files with any file extension. This permission must be turned on explicitly and will not be automatically set on upgrades. This permission does not allow access to files outside of root, but only to files that are contained within the specified File Root (which can be set to /// or e.g. C:// to allow access to all files on a server). The new permission can be controlled through the *Setup > Security > Files* section of Site Administration.

Introduction

All Lasso Professional 7 solutions which were written using preferred syntax should run without modifications in Lasso Professional 8 except for the issues mentioned in this chapter.

Significant effort has been expended to ensure that existing solutions will continue to run in Lasso Professional 8 with few if any modifications required. However, please read through this chapter to learn about changes that may require modifications to your solutions.

Lasso Studio and LDML Updater

Lasso Studio includes an LDML Updater that can be used on code from earlier versions of Lasso to bring it into compliance with the latest version of LDML. See the documentation for Lasso Studio for more information.

Security Enhancements

Lasso Professional 8.0.1 introduced a number of security enhancements that are described here.

- **Lasso Page Extensions** – The allowed extensions have been split into two sets. Lasso Page Extensions now controls what files Lasso will execute through URLs and the [Include] and [Library] tags. File Tags Extensions controls what files Lasso can manipulate through the file tags, image tags, PDF tags, and [Include_Raw]. Both sets of extensions can be controlled through the *Setup > Site > File Extensions* section of Site Administration.

  By default the Lasso page extensions are .lasso .lassoapp .las .htm .html .inc .incl. By default the file tags extensions are .bmp .cmyk .gif .jpg .pdf .png .psd .rgb .text .tif .txt .uld .wsdl .xml .xsd. For best security these two sets of exten-

sions should remain mutually exclusive. Adding .* to either set will allow all file extensions for that set.

Upgraded servers will start with the default set of Lasso page extensions. The file tag extensions will be set to the complete list of format file extensions that were already set. For best results the file tag extensions should be reset using Reset Extensions in Site Admin.

This update may require some changes to your Lasso pages if you allow third parties to upload .lasso or .html files or if you use -Response to return image, PDF, or XML files. The best solution is to change your code so that users only upload files in the file tags extensions (and aren't allowed to upload any files through Lasso which can be executed by Lasso). Any URL that uses -Response to reference an image, PDF, or XML file can be rewritten to use a straight URL referencing the appropriate file.

- **Classic Lasso** – Classic Lasso now has three options to determine whether it is enabled or not. Enabled means that all Classic Lasso URL parameters can be used including database actions. Minimal means that database actions are disabled, but -Response and -Error parameters can still be used in URLs. Disabled means that no Classic Lasso URL parameters can be used and that even -Response and the -Error tags are disabled. The new options can be controled through the *Setup > Site > Settings* section of Site Administration.
- **File Permissions** – File Permissions now has an option to allow users in a group to access files with any file extension. This permission must be turned on explicitly and will not be automatically set on upgrades. This permission does not allow access to files outside of root, but only to files that are contained within the specified File Root (which can be set to /// or e.g. C:// to allow access to all files on a server). The new permission can be controlled through the *Setup > Security > Files* section of Site Administration.

SQLite

The internal data source in Lasso Professional 8 has been changed from Lasso MySQL to SQLite. This change has several ramifications for developers who are upgrading to Lasso Professional 8.

- **Solution Databases** – Any solution databases which were hosted by Lasso MySQL should be moved to an external installation of MySQL. This is the best way to ensure that the solutions continue to run without any modifications. It is nor recommended that any solution databases be converted to SQLite.

- **Internal Databases** – All of the internal databases of Lasso have been converted to SQLite including Lasso_Internal, Lasso_Site_1 (also known as Site), Lasso_Admin, and LDML8_Reference. Any solutions which reference these databases to modify Lasso's internal settings may need to be updated for compatibility with SQLite. Non-SQL inlines will require few modifications, but -SQL inlines will need to use SQLite compatible statements.

  The names and schema of some of the internal tables have changed. The _errors table from Lasso Professional 7 is now named errors. The SMTP queue table has been completely modified for the new SMTP sending implementation.

All of the internal functionality which makes use of the internal data source including Lasso security, the email queue, scheduled events, sessions, etc. have been rewritten to use SQLite. No modification to any code that makes use of these features should be required.

Multi-Site

Lasso Professional 8 has an entirely new multi-site architecture. All solution code is run within a site that is automatically spawned by Lasso Service. Each site has its own site level folder that contains duplicates of the folders at the master level.

Lasso Professional 8 is installed with a single default site. The easiest transition from Lasso Professional 7 is to use this default site for all of the Web hosts on the server. Once the Lasso Professional 8 transition has been made additional sites can be added if needed.

The structure of the Lasso Professional 8 application folder appears below in abbreviated form. When Lasso is loading or needs a resource it checks both the site level and the master level. In general, the folders at the site level are checked first and if the resource is not found then the master level is checked.

```
Lasso Professional 8/
  LassoAdmin/
  LassoModules/
  LassoStartup/
  SQLiteDBs
  …

  LassoSites/
    default-1/
      LassoAdmin/
      LassoModules/
```

```
    LassoStartup/
    SQLiteDBs
    ...
...
```

Note: The appropriate installation chapter of the Lasso Professional Server 8 Setup Guide has a more complete listing of all installed files.

- **LassoAdmin** – Items stored by Site Administration can be found in the site level LassoAdmin folder. This includes backups, exports, built LassoApps, etc. This folder was called Admin in prior versions of Lasso.
- **LassoModules** – Modules are loaded from both the site and master levels when LassoService starts up. The same rule applies for JDBCDrivers and JavaLibraries.
- **LassoStartup** – All format files and LassoApps in both the site and master level LassoStartup folders are loaded when Lasso starts up. However, only one copy of Startup.LassoApp is loaded (from the site level if it exists or the master level otherwise).
- **SQLiteDBs** – Each site uses its own set of site level SQLite databases. Individual sites do not have access to the master level SQLite databases.
- **[Admin_LassoServicePath]** – This tag reports the location of the site level folder rather than the location of the Lasso Professional 8 folder. For example:

  ```
  ///Applications/Lasso Professional 8/LassoSites/default-1/
  ```

When upgrading a server from Lasso Professional 7, any third-party modules or JDBC drivers can be moved into the master level. They will be available to the default site and to any sites that are defined on the server.

Namespaces

Lasso Professional 8's namespace support should be transparent to most users of Lasso. However, there are a couple situations where updates may be needed.

- **Replacing Built-In Tags** – When replacing a built-in tag in Lasso Professional 8 using [Define_Tag] ... [/Define_Tag] and -Priority='Replace' the tag definition must reference the proper namespace using the -Namespace parameter. Any existing code which attempts to replace built-in tags will need to be updated with the proper -Namespace parameter.

Note: Custom tag definitions which are not redefining built-in tags do not require any modifications to work with Lasso Professional 8.

- **On-Demand Libraries** – Many built-in tags have been moved to on-demand loading from the LassoLibraries folder. This should not require any code modifications.
- **Custom Tag Names** – No changes are required to custom tags in Lasso Professional 8. Custom tag definitions will work fine without the -Namespace parameter and all the same rules for tag naming apply as for previous versions of Lasso.

  ```
  [Define_Tag: 'Ex_MyTag_TagName'] ... [/Define_Tag]
  ```

  However, if the -Namespace parameter is added to custom tag definitions then the new rules for tag naming must be followed. The tag name itself must not contain any underscores. The portion of the original tag name before the last underscore should be used as the namespace for the tag. The namespace must not contain a double underscore.

  ```
  [Define_Tag: 'TagName', -Namespace='Ex_MyTag_'] ... [/Define_Tag]
  ```

Digest Authentication

Lasso Professional 8 supports digest authentication as a Web browser authentication method. Digest authentication is more secure than the basic authentication supported by earlier versions of Lasso since passwords are only sent after they have been encrypted using a one-way hash algorithm.

Digest authentication is supported by all modern Web browsers. Lasso can send both digest and basic authentication challenges. An older browser that does not recognize digest authentication should fall back on basic authentications. Digest authentication can be turned on and off in the *Setup > Site > Syntax* section of Site Administration.

One important advantage of digest authentication is that the realm name is significant. A site visitor can be authenticated against several different realm names with different usernames and passwords for each. However, this can present a problem is a site uses many different realm names that were previously ignored. The best solution is to modify all the realm names to be the same or to use Lasso's default realm name of Lasso Security.

In order to support digest authentication Lasso must have the password for each user in the security_user table stored in plain text. In prior versions of Lasso all passwords were stored as MD5 hashes. Digest authentication will only work for users created or modified with Lasso Professional 8 so the plaintext password is stored properly. Lasso will fall back on basic authentication for users who do not have a plaintext password.

The [Client_Authorization] tag can be used to see what type of authentication is being used for the current site visitor.

On-Demand LassoApps

All of Lasso's built-in LassoApps are now provided as on-demand LassoApps which load from the LassoApps folder in the Lasso Professional 8 application folder. The /Lasso/ folder is no longer required in the Web server root. This means that the traditional URLs to load Lasso Administration will not necessarily work any more. Instead, the following URLs should be used.

Server Administration – http://www.example.com/ServerAdmin.LassoApp

Site Administration – http://www.example.com/SiteAdmin.LassoApp

Database Browser – http://www.example.com/DatabaseBrowser.LassoApp

Group Administration – http://www.example.com/GroupAdmin.LassoApp

LDML 8 Reference – http://www.example.com/LDMLReference.LassoApp

XML-RPC / SOAP – http://www.example.com/RPC.LassoApp

Lasso Studio – http://www.example.com/LassoStudio.LassoApp

If the Web server requires it (e.g. Apache on Mac OS X) a /Lasso/ folder can be created in the Web server root so that the URLs from prior versions of Lasso will work. Simply creating an empty folder will allow Lasso to load its on-demand LassoApps from the Lasso Professional 8 application folder.

Syntax Changes

None of the syntax changes in Lasso Professional 8 should require modifications to existing sites. All of these changes are fully backward compatible.

- **Parentheses Syntax** – Parentheses syntax in Lasso Professional 8 is backward compatible with Lasso Professional 7. Sites written using parentheses syntax in Lasso Professional 8 should run fine in Lasso Professional 7 (but not in earlier versions of Lasso).
- **BlowFish Encryption** – Lasso now includes a BlowFish2 encryption algorithm which should be compatible with most third-party implementations of BlowFish.

Parentheses Syntax

Lasso now supports parentheses syntax in which a tag name is followed by parentheses that include the parameters of the tag.

```
Tag_Name(-Param, -Name=Value);
```

The syntax of prior versions of Lasso is called colon syntax and is also fully supported in Lasso Professional 8. There are no plans to deprecate colon syntax.

```
Tag_Name: -Param, -Name=Value;
```

The two syntax methods can be used interchangeably, even in the same expression. No modification of existing sites should be required.

Note: Partial support for parentheses syntax was provided in Lasso Professional 7. Some sites written using parentheses syntax will run fine in Lasso Professional 7 or Lasso Professional 8. However, the use of parentheses syntax in Lasso Professional 7 is not fully supported.

BlowFish Encryption

Earlier versions of Lasso included a BlowFish implementation that was not compatible with most third-party BlowFish implementations. This could make it difficult to transmit data to other Web application servers, Java applets, etc. using a secure BlowFish channel.

A new BlowFish2 algorithm is now provided which uses an industry standard implementation. The [Encrypt_BlowFish2] and [Decrypt_BlowFish2] tags are the preferred tags to use when performing BlowFish encryption and should always be used when communicating with another software product using BlowFish.

However, the original [Encrypt_BlowFish] and [Decrypt_BlowFish] tags are still provided for backward compatibility. These tags are safe to use for existing solutions and are recommended for a solution that needs to be able to share data with older versions of Lasso or for communication with older versions of Lasso.

Tag Name Changes

The table below summarizes the names of all the tags that have changed since Lasso Professional 7. The old tag name is still supported in Lasso Professional 8, but future development should use the new tag name.

Table 1: Tag Name Changes

Old Name	New Name
[Array->FindIndex]	[Array->FindPosition]

Lasso uses positions to reference elements within compound data types. Positions run from 1 to the size of the compound data type. The [Array->FindIndex] tag has been renamed [Array->FindPosition] to reflect this terminology.

21

Chapter 21
Upgrading From Lasso Professional 6

This chapter contains important information for users of Lasso Professional 6 who are upgrading to Lasso Professional 8. Please read through this chapter before attempting to run solutions in Lasso Professional 8 that were originally developed for an earlier version of Lasso.

The upgrading chapters are cumulative so this chapter should also be read by users of Lasso Professional 5 or earlier who are upgrading to Lasso Professional 8.

Topics in this chapter include:

- *Introduction* includes general information about what has changed in Lasso Professional 8.
- *Error Reporting* describes the new error reporting customization features.
- *Unicode Support* describes how Lasso uses Unicode internally and how Lasso translated to and from other character encodings automatically.
- *Bytes Type* describes the bytes type, lists the tags that return data in the bytes type, and compares the bytes type to the string type.
- *Syntax Changes* contains information about what LDML syntax constructs have changed since Lasso Professional 6.
- *Tag Name Changes* details the tag names which have been changed since LDML 6.

Introduction

This chapter includes the upgrading instructions from Lasso Professional 6 to Lasso Professional 7. Lasso Professional implements all of the features of Lasso Professional 7. If a site is being upgraded from Lasso Professional 6 the items in this chapter should be applied first, then the items in the prior chapter about upgrading from Lasso Professional 7.

In general, most sites that are written using preferred Lasso Professional 6 syntax should run with few modifications in Lasso Professional 8.

Lasso Studio and LDML Updater

Lasso Studio includes an LDML Updater that can be used on code from earlier versions of Lasso to bring it into compliance with the latest version of LDML. See the documentation for Lasso Studio for more information.

Error Reporting

Lasso Professional 7.0.2 introduces some important enhancements to how syntax errors and logical errors are reported by Lasso. Each of these enhancements is discussed in this section and additional details are provided within the *Error Controls* chapter.

- The error reporting level can now be adjusted in Lasso Administration and overridden on individual pages. The error reporting level controls whether the built-in error page provides full troubleshooting details, minimal error messages, or no error details at all.
- The built-in error page can now be modified in order to provide a custom server-wide error page for all sites hosted on a server. This page can work in concert with the site-specific custom error pages to provide an appropriate amount of information to every site visitor.

Error Reporting Level

For errors that occur while processing a page, Lasso displays error messages differently based on the current error reporting level. This allows detailed error messages to be displayed while developing a Web site and then for

minimal or generic error messages to be displayed once a site has been deployed.

The default global error reporting level can be set in Lasso Administration in the Setup > Global > Settings section. The error reporting level can be set to None, Minimal, or Full. The default is Minimal. Each of these levels is described in more detail below.

The error reporting level for a particular page can be modified using the [Lasso_ErrorReporting] tag. This will modify the error reporting level only for the current format file and its includes without affecting the global default. See the section on the [Lasso_ErrorReporting] tag in the *Error Controls* chapter for additional details.

- **None** – This level provides only a generic error message with no specific details or error code. This level can be used on a deployment server when it is desirable to provide no specific information to site visitors.
- **Minimal** – This level is the default. It provides a minimal error message and error code. No context about where the error occurred is provided. This level can be used on a deployment server in order to make trouble-shooting problems easier.
- **Full** – This level provides detailed error messages for debugging and troubleshooting. The path to the current format file is provided along with information about what files have been included and what parameters have been passed to them. If a database or action error is reported the built-in error message provides information about what database action was performed when the error occurred.

It is recommended that the global error reporting level on a production Web server be set to the default of Minimal or to None. This will ensure that site visitors are not given detailed error messages intended for the developer of the Web site. On a page by page basis the [Lasso_ErrorReporting] tag can then be used to set the error level to Full in order to make debugging a site in development easier.

The [Lasso_ErrorReporting] tag can also be used with the -Local keyword to set the error reporting level to None within sensitive custom tags or include files in order to suppress error messages from select portions of a site.

Custom Server-Wide Error Page

The server-wide error page is now stored in the file error.lasso within the Admin folder in the Lasso Professional 8 application folder. By customizing this file the default error page for all sites hosted on the server can be modified.

This file can be customized for any of the following purposes:

- To customize the appearance of the error page. For example, a professional hosting service could provide an error page that provides information for their clients about how to handle the error.
- To provide an appropriate amount of information to site visitors. The built-in page provides different information depending on the current error reporting level. The same levels can be used to provide either more or less information depending on level.
- To provide logging or notification. Logging tags can be added to the error page in order to keep track of what errors have occurred. Email notification could be used to alert the site administrator that an error has occurred.

The customized error.lasso page should be thoroughly debugged prior to being made active, especially on a production Web server. It can be very difficult to troubleshoot problems which are occurring on a server if there is a problem with the error reporting page.

The server-wide error.lasso page will only be displayed if no site-specific error.lasso file is present or if there is an error within a site-specific error.lasso file.

Unicode Support

Lasso Professional 8 introduces Unicode support throughout Lasso Service, the database connectors, and LDML. This is a significant architectural change that alters how all string and binary data is processed by Lasso.

The Unicode standard defines a universal character set which includes characters for just about every language on the planet. The transition to a full Unicode workflow should make it easier to transfer data that contains characters which formerly required special-purpose encodings between different applications.

Unicode is rapidly achieving dominance as the standard encoding for data on the Internet, in leading operating systems, and in database products. Recently, Mac OS X and Windows have both implemented native support for Unicode. All current leading Web browsers support Unicode data. Many text editors such as BareBones BBEdit have recently introduced native support for Unicode. And, future database offerings from MySQL and other database vendors are expected to offer full support for Unicode encoding.

Every effort has been made to make the change to Unicode transparent to the Lasso developer. However, these architectural changes may require modification of some Lasso Professional 6 sites and may require some

additional planning and coding in order to work with Web browsers and databases that do not yet support Unicode.

The following list includes details about how Unicode is supported in Lasso Professional 8 and also includes details about backwards compatibility.

Note: Please also read the following section on the new ***Bytes Type*** for details about how binary and string data is handled using LDML tags.

- **Format Files** – If a format file contains a valid byte-order mark it is read using the UTF-8 character encoding. If no byte-order mark is read then the format file is assumed to be encoded in the Macintosh (or Mac-Roman) character set on Mac OS X or the Latin-1 character set (also known as ISO 8859-1) on Windows or Linux.

  Popular text editors such as BBEdit can encode text files using UTF-8 and will insert the proper byte-order mark that Lasso needs to identify the character set of the format file. Consult the documentation for the text editor for more information.

  All existing format files will be read using the Macintosh or Latin-1 (ISO 8859-1) character sets which Lasso Professional 6 used. New format files which need to take advantage of the extended character set that Unicode offers should be encoded as UTF-8 and include a proper byte-order mark.

  Note: Lasso does not support format files encoded using the UTF-16 or UTF-32 character sets.

- **Web Browser** – By default all files sent to the Web browser by Lasso Professional 8 will be encoded using UTF-8. The default page encoding option in the ***Settings > Global > Syntax*** section of Lasso Administration can be used to change the default encoding to the Lasso 6 (pre-Unicode) standard Latin-1 character set (also known as ISO 8859-1).

  If encoding different from the default is needed for a given format file the [Content_Type] tag can be used to instruct Lasso to encode the returned page using a different character set. For example the following tag instructs Lasso to use the Latin-1 (ISO 8859-1) character set.

  ```
  [Content_Type: 'text/html;charset=iso-8859-1']
  ```

  The [Content_Type] tag sets the page variable __encoding__ to the desired character set. Modifying this variable will also change the character set that will be used to return the page to the client's Web browser.

- **Forms** – Most Web browsers return data from HTML forms using the same encoding that was used to transmit the Web page to them. Lasso assumes that all incoming form data is going to use the default page encoding which is set in Lasso Administration in the *Settings > Global > Syntax* section. This means that all incoming form data must use either UTF-8 or Latin-1 (ISO 8859-1) encoding.

  It is recommended to use UTF-8 as the default page encoding since this is the emerging Internet standard. However, if forms are being submitted to Lasso from Web pages that do not use UTF-8 (or from pre-Unicode browsers) it may be necessary to change the default page encoding so data in the forms will be interpreted properly.

- **Database Connector** – Lasso communicates with each database using the character set specified in the table settings in Lasso Administration. The character set for MySQL databases can be set to either UTF-8 or Latin-1 (ISO 8859-1).

  By default, the Lasso Connectors for MySQL communicate with existing databases using the Latin-1 (ISO 8859-1) character set. If desired, the character set for each table can be changed to UTF-8 in the *Setup > Data Sources > Tables* section of Lasso Administration.

  SQL statements sent using the -SQL parameter are encoded similarly. If the -Table parameter is specified then the character set for that table will be used. If no -Table parameter is specified then the SQL statement will be encoded using the Latin-1 (ISO 8859-1) character set.

  The Lasso Connector for FileMaker Pro uses Latin-1 (ISO 8859-1) encoding by default on Windows and Linux. Macintosh (or Mac-Roman) encoding is used by default on Mac OS X. If required, the character set for each database can be changed in the *Setup > Data Sources > Databases* section of Lasso Administration.

  The Lasso Connector for JDBC always uses UTF-8 on any platform.

  Since the default character set encoding for each database connector is the same as that used in Lasso Professional 6, no changes should be required to existing solutions. However, any database containing extended characters must continue to use the same character encoding or stored data may not be interpreted properly when it is retrieved from the database.

- **LDML Tags** – All LDML tags process string data as double-byte Unicode strings. Character encoding is only performed when data is imported into Lasso or exported according to the rules specified above. The bytes type can be used to process binary data and to perform low-level character set translation if required.

See the following section for details of what LDML tags return data in the bytes type and how it compares with the string type.

Bytes Type

Since all string data is now processed using double-byte Unicode strings it is necessary to introduce a new data type that stores single-byte binary data strings. This new data type in Lasso Professional is called the bytes type and is manipulated using the [Bytes] tag and associated member tags. Data in the bytes type is often referred to as a byte stream.

The bytes type adds two important abilities to Lasso Professional 8. Binary data can be treated separately from string data and data can be converted between single-byte character sets directly within Lasso. The bytes type is fully documented in the Extending Lasso Guide. Please see that manual for additional details about the bytes type and the member tags that it supports.

The bytes type is used to return any strings in Lasso that will potentially contain binary data. Many substitution tags always return byte streams or do so under certain circumstances. These tags are listed in the following *Table 1: Tags That Return the Bytes Type*.

Table 1: Tags That Return the Bytes Type

Tag	Description
[Bytes]	Used to create a new bytes buffer or to cast a string to the bytes type. Many of the member tags of the bytes type also return byte streams.
[Decompress]	Always returns a byte stream.
[Decrypt_BlowFish]	Always returns a byte stream.
[Encode_Base64]	Always returns a byte stream.
[Encrypt_BlowFish]	Always returns a byte stream.
[Field]	Returns a byte stream only for MySQL fields of type BLOB.
[Field_Read]	Always returns a byte stream.
[File_ReadLine]	Always returns a byte stream.
[File->Read]	Always returns a byte stream.
[Include_Raw]	Always returns a byte stream.
[Include_URL]	Always returns a byte stream.
[Net->Read]	Always returns a byte stream.
[Net->ReadFrom]	Always returns a byte stream.

[String_ReplaceRegExp]	Returns a byte stream if the input is a byte stream, otherwise returns a string.
Other Tags	Many tags in LDML such as [Array->Get] or [Map->Find] return data in the same type it was stored. These tags may also return data in the bytes type.

Bytes and Strings

The bytes type and the string type support many of the same member tags. These member tags can be used on either byte streams or strings without worrying about the underlying data type. The shared member tags are listed in *Table 2: Byte and String Shared Member Tags*.

In addition to these tags both the bytes type and the string type support the standard comparison operators ==, !=, >>, !>>, ===, and !==. The bytes type also supports the + and += symbol for appending data to the end of the stream.

Table 2: Byte and String Shared Member Tags

Tag	Description
[Bytes->Append]	Appends the specified characters onto the end of the byte stream.
[Bytes->BeginsWith]	Returns true if the byte stream begins with the specified characters. Case sensitive.
[Bytes->Contains]	Returns true if the byte stream contains the specified characters. Case sensitive.
[Bytes->EndsWith]	Returns true if the byte stream ends with the specified characters. Case sensitive.
[Bytes->Find]	Returns the position of the specified characters within the byte stream. Case sensitive.
[Bytes->Get]	Returns a specified character from the byte stream.
[Bytes->Length]	Returns the length of the byte stream in bytes.
[Bytes->RemoveLeading]	Removes the specified characters from the beginning of the byte stream. Case sensitive.
[Bytes->RemoveTrailing]	Removes the specified characters from the end of the byte stream. Case sensitive.
[Bytes->Replace]	Replaces the specified characters in the byte stream with the specified replacement. Case sensitive.
[Bytes->Size]	Returns the length of the byte stream in bytes.
[Bytes->Split]	Splits the byte stream on the specified character. Case sensitive.

[Bytes->Trim]	Trims ASCII white space characters from the start and end of the byte stream. Removes spaces, tabs, return characters, and newline characters.

The table above includes all of the most commonly used string member tags. These tags help to make the string and bytes types generally interchangeable. However, there are a significant number of string member tags that are not supported by the bytes type. These are listed in *Table 3: Unsupported String Member Tags*.

In order to use any of these member tags on byte streams the data must first be converted to a string. Information on how to convert data to and from the bytes type and string type is included in the next section.

Table 3: Unsupported String Member Tags

Tag	Description
Character Tags	Tags which fetch information about the characters in a string including [String->CharDigitValue], [String->CharName], and [String->CharType].
Comparison Tags	Tags which compare strings with case sensitivity. [String->Compare] and [String->CompareCodePointOrder]
Case Tags	Tags which report the case of a string including [String->FoldCase], [String->LowerCase], [String->TitleCase], and [String->UpperCase].
Case Modification Tags	Tags which change the case of a string including [String->ToLower], [String->ToTitle], and [String->ToUpper].
Character Is Tags	Tags which report information about the characters within a string including all tags starting with [String->Is…].
Miscellaneous Tags	The following tags are also not supported by the bytes type: [String->Digit], [String->Merge], [String->PadLeading], [String->PadTrailing], [String->Remove], [String->Reverse], [String->Substring], and [String->Unescape].

Converting From Bytes to Strings

Data can be converted from byte streams to strings easily, but the method differs depending on what character set the byte stream is encoded in and how the data is going to be used.

- **Automatic Casting** – When a byte stream is passed to a substitution or process tag that is expecting a string value it is automatically cast to type string. For example, the [String_...] substitution tags automatically cast their parameters to strings.
- **Explicit Casting** – The [String] tag can be used to explicitly cast a byte stream to a string. The byte stream will be converted by assuming it is encoded using the UTF-8 character set. Explicit casting is appropriate for data read in using the [Include_URL] or [Net->Read] tags since most communication on the Internet is encoded using this character set.
- **Converting Character Sets** – The [Bytes->ExportString] tag can be used to convert a byte stream that is encoded using a character set other than UTF-8 into a string. The tag accepts a single parameter which specifies what character set the byte stream is encoded in and returns a string (encoded in the default Unicode double-byte encoding that Lasso uses internally for all strings.

  For example, a file can be read in the Mac-Roman character set and converted to a string using this code.

```
[Var: 'myfile' = (File_Read: 'myfile.txt')]
[Var: 'mystring' = $myfile->(ExportString: 'macintosh')]
```

Similar methods can be used to convert strings into byte streams. Tags that expect a byte stream as a parameter automatically cast strings to byte streams. These tags include [File_Write], [File_WriteLine], [File->Write], [Net->Write], etc. The [Bytes] tag explicitly casts strings to a byte stream. The [Bytes-ImportString] tag with a string parameter and an encoding parameter can be used to import a string into a byte stream converting it to any desired character encoding.

Bytes Member Tags

The bytes type supports a number of additional member tags which are documented in full in the Extending Lasso Guide. Please see that manual for more information.

Updating Existing Sites

In order to promote backwards compatibility the bytes type supports the core member tags of the string type and Lasso performs automatic conversions between the two types when necessary.

However, there are a couple situations which will require Lasso Professional 6 sites to be updated in order to work properly with Lasso Professional 8. These situations are detailed below.

- **Checking for String Type** – Byte streams are of type bytes so explicit checks for type string will fail. For example, the following code reads a file into a variable and then checks the type of the variable.

```
[Var: 'myfile' = (File_Read: 'myfile.txt')]
[If: $myfile->type == 'string']
  …
[/If]
```

  The conditional will fail since the variable myfile is of type bytes rather than type string. The conditional can be changed to the following to create code that works in either Lasso Professional 6 or 7.

```
[Var: 'myfile' = (File_Read: 'myfile.txt')]
[If: ($myfile->type == 'string') || ($myfile->type == 'bytes')]
  …
[/If]
```

- **String Member Tags** – If any string member tags are used on a byte stream which are not supported by the bytes type then an error will occur. For example, this code to read in a file and then convert it to uppercase will fail in Lasso Professional 8 since the tag [String->toUpper] is not implemented for the bytes type.

```
[Var: 'myfile' = (File_Read: 'myfile.txt')]
[$myfile->toUpper]
```

  There are two solutions to this issue. The easiest is to cast the output of [File_Read] to a string before storing it in a variable. This solution can be applied across a site by doing a search for each of the tags that return byte streams and adding an explicit cast using the [String] tag.

```
[Var: 'myfile' = (String: (File_Read: 'myfile.txt'))]
[$myfile->toUpper]
```

  Another possibility is to use a substitution tag rather than a member tag to perform the string conversion. The substitution tag will automatically cast the byte stream to a string and will return a string value.

```
[Var: 'myfile' = (File_Read: 'myfile.txt')]
[Var: 'myfile' = (String_UpperCase: $myfile)]
```

Syntax Changes

Lasso Professional 8 introduces changes to some of the core syntax rules of LDML. Most of these changes were made to improve the reliability and error reporting of Lasso Professional 8. Some of these changes may require you to rewrite portions of your existing Lasso-based solutions for full

compatibility with Lasso Professional 8. This section describes each change, why it was made and how to update existing format files.

Table 4: Syntax Changes

Syntax Change	Description
File Tags	The file tags have been modified to provide more consistent behavior. The ../ path is now supported to move up a directory. Updated in 7.0.2 release.
Strict Syntax	Strict syntax is now required: all parameter keywords must be preceded by a hyphen, all string literals must be surrounded by quote marks, and all tag names must be defined before being called.
Recursion Limit	A limit can be configured on the depth of nested [Include] and [LIbrary] tags allowed. By default the limit is a depth of 50.
Format File Execution Time Limit	A limit can be configured on the maximum amount of time that a format file will be allowed to execute. By default the limit is 10 minutes.
Internal Tags	Many tags are now implemented as part of the LDML parser in order to prove better performance.
Iterate Enhancement	The [Loop_Count] and [Loop_Abort] tags can now be used within [Iterate] … [/Iterate] tags.
Custom Tags Enhancement	Database results can now be retrieved from within custom tags.
Miscellaneous Shortcuts	A number of syntax shortcuts have been introduced. See the full description below for details.
Unicode Support	All strings are now processed using double-byte Unicode and output in UTF-8 format by default. See also the discussions of Unicode Support and the Bytes Type which precede this section.
Classic Lasso	Classic Lasso support is disabled by default and its use has been deprecated. Solutions relying on Classic Lasso should be transition to Inline-based methodology.
-Email… Command Tags	The -Email… commands are no longer supported in LDML 8. The [Email_Send] tag must be used instead.
Decimal Precision	Decimal numbers are output using the fewest number of significant digits possible.
Member Tags and Parentheses	Member tags which have multiple parameters must be surrounded by parentheses.
PDF -Top Parameter	The -Top parameter in various PDF tags always measures from the top margin of a document.

Global Variables	Use the [Global] tag rather than the [Variable] tag to reference global variables.
[NSLookup]	Due to changes in Mac OS X 10.3 reverse lookups may not work with all DNS servers.
[Repetition] Tag	The [Repetition] tag has been deprecated. Rewriting pages to use the modulus symbol % will result in better performance.
[TCP_…] Tags	The [TCP_…] tags have been deprecated in favor of the new [Net] type and its member tags.
[Else:If] and [Else_If]	These tags are no longer supported. Use [Else] instead.
[LoopCount] and [LoopAbort]	These tags are no longer supported. Use [Loop_Count] and [Loop_Abort] instead.
Container Tags	Container tags must be defined within LassoStartup. Container tags cannot be defined on-the-fly. New keywords allow both looping and simple container tags to be created.
Custom Tags	Parameters and return values are now passed by reference. [PreCondition] and [PostCondition] are no longer supported. Asynchronous tag update.
XML Tags	The XML tags have been re-implemented. Some modifications to existing sites may be required.
[Encode_ISOtoMac]	This tag and [Encode_MacToISO] have been deleted. Their functionality can be replicated using the new [Bytes] type.

File Tags

The behavior of the file tags when moving, copying, or renaming files has been made more consistent. The following rules will be used.

- If a file is being operated on and the destination is a file name then the file will be moved or copied to that file name. For example, the following code will rename the file example.txt to renamed.txt.

```
[File_Move: 'example.txt', 'rename.txt']
```

- If a file is being operated on and the destination is a directory then the file will be moved or copied into the directory. For example, the following code will move the file example.txt into the /directory/.

```
[File_Move: 'example.txt', '/directory/']
```

- If a directory is being operated on and the destination is a directory then the source directory will replace the destination directory. For example, the following code will replace the directory /destination/ with the directory /source/.

```
[File_Move: '/source/', '/destination/']
```

In addition, the ../ path can now be used within all file tags and include tags in order to move up one directory. The effective path will be computed and then the security settings will be checked to confirm that the current user has permission to access the specified directory.

Strict Syntax

Lasso Professional 6 introduced the option to use strict syntax checking. This option was on by default, but could be turned off for better backwards compatibility with Lasso Professional 5 and earlier.

In Lasso Professional 8, strict syntax checking is now required. It can no longer be deactivated.

With strict syntax the following rules are enforced:

- All keyword parameters to built-in and custom tags must include a hyphen. This helps to find unknown tag parameters and to catch other common syntax errors.
- All string literals must be surrounded by quotes. This helps to prevent accidental calls to tags, to identify undefined variables, and to catch other common syntax errors.
- All tag calls must be defined. Unknown tags will no longer simply return the tag value as a string.

With strict syntax any of the errors above will be reported when a page is first loaded. They must be corrected before the code on the page will be executed. When upgrading to Lasso Professional 8 it is advisable to first try existing Lasso Professional 6 sites and correct any errors that are reported.

To update existing sites for strict syntax:

If a site is relatively small then the easiest method is to load each Web page and see if any errors are reported. The following tips can be used for a more methodical search.

- Check that all string literals are surrounded by quotes. Quotes are not necessary around integers or decimal numbers, hyphenated keyword parameters, tag names, or variable names when used with the & or # symbols.
- Check that all keywords in tag calls are preceded by a hyphen. Keyword and keyword/value parameters must be preceded by a hyphen, but do not need to be quoted. Name/value parameters should include quotes around both the name and value (unless they are numbers).

- Check that all command tags used within opening [Inline] tags are preceded by a hyphen. Quotes are not necessary around command tags, even when they are specified within an array.
- Check that all client-side JavaScript is formatted properly. JavaScript should either be included in [NoProcess] ... [/NoProcess] tags or HTML comment tags <!-- ... --> which ensure that no LDML code within is processed. Or, any square brackets which are required within the JavaScript should be output from a [String] tag.

```
[String: '[array[4]]']
➜ [array[4]]
```

Recursion Limit

Lasso includes a limit on the depth of recursive include files. This limit can help prevent errors or crashes caused by some common coding mistakes. The limit sets the maximum depth of nested [Include] and [Library] tags that can be used. If the depth is exceeded then a critical error is returned and logged.

The recursion limit is set to 50 by default and can be modified or turned off in the *Setup > Global > Settings* section of Lasso Admin.

Note that the limit does not apply to the number of [Include] and [Library] tags within a single file, but to the depth reached using an [Include] tag to include a file that itself uses an [Include] tag to include another file and so on.

Format File Execution Time Limit

Lasso includes a limit on the length of time that a format file will be allowed to execute. This limit can help prevent errors or crashes caused by infinite loops or other common coding mistakes. If a format file runs for longer than the time limit then it is killed and a critical error is returned and logged.

The execution time limit is set to 10 minutes (600 seconds) by default and can be modified or turned off in the *Setup > Global > Settings* section of Lasso Admin. The execution time limit cannot be set below 60 seconds.

The limit can be overridden on a case by case basis by including the [Lasso_ExecutionTimeLimit] tag at the top of a format file. This tag can set the time limit higher or lower for the current page allowing it to exceed the default time limit. Using [Lasso_ExecutionTimeLimit: 0] will deactivate the time limit for the current format file altogether.

On servers where the time limit should be strictly enforced, access to the [Lasso_ExecutionTimeLimit] tag can be restricted in the *Setup > Global > Tags* and *Security > Groups > Tags* sections of Lasso Admin.

Asynchronous tags and compound expressions are not affected by the execution time limit. These processes run in a separate thread from the main format file execution. If a time limit is desired in an asynchronous tag the [Lasso_ExecutionTimeLimit] tag can be used to set one.

Note: When the execution time limit is exceeded the thread that is processing the current format file will be killed. If there are any outstanding database requests or network connections open there is a potential for some memory to be leaked. The offending page should be reprogrammed to run faster or exempted from the time limit using [Lasso_ExecutionTimeLimit: 0]. Restarting Lasso Service will reclaim any lost memory.

Internal Tags

Many LDML tags are now implemented directly in the LDML parser in order to provide better performance. Since the new versions of these tags implement the same functionality as the old version of these tags no changes to existing solutions are required. However, the [Lasso_TagExists] tag will report False for all of the internal tags.

The internal tags include:

[Abort], [Define_Tag] ... [/Define_Tag], [Define_Type] ... [/Define_Type], [Encode_Set] ... [/Encode_Set], [Fail], [Fail_If], [False], [Handle] ... [/Handle], [Handle_Error] ... [/Handle_Error], [If] ... [Else] ... [/If], [Iterate] ... [/Iterate], [Lasso_Abort], [Loop] ... [/Loop], [Loop_Abort], [Loop_Count], [NoProcess], [Params], [Protect] ... [/Protect], [Return], [Run_Children], [Select] ... [Case] ... [/Select], [Self], [True], and [While] ... [/While].

Iterate Enhancement

In Lasso Professional 6 the [Iterate] ... [/Iterate] tags did not support the use of the [Loop_Count] or [Loop_Abort] tags. These tags have been rewritten in Lasso Professional 8 so that all looping container tags now function identically.

In the following example the [Loop_Count] is output on each iteration and the iteration is stopped after the item Beta is seen.

```
[Iterate: (Array: 'Alpha', 'Beta', 'Gamma'), (Var: 'Temp')]
  <br>[Loop_Count]: [Var: 'Temp']
  [If: $Temp == 'Beta']
    [Loop_Abort]
  [/If]
[/Iterate]
```

```
➜ <br>1: Alpha
  <br>2: Beta
```

For more information about the [Iterate] ... [/Iterate] tags see the *Conditional Logic* chapter.

Custom Tags Enhancement

In Lasso Professional 6 it was not possible to get to the results of a database action from within a custom tag. In Lasso Professional 8 this limitation has been removed. It is now possible to write custom tags which work directly with [Field] data, the [Found_Count], [Action_Params], or any other values.

As a demonstration of this new ability the [Link_...] tags have all been rewritten as custom tags.

In the following example, a custom tag returns a string describing the results of a database action.

```
[Define_Tag: 'Ex_Results']
  [Return: 'Showing ' + (Shown_Count) + ' records of ' + (Found_Count) + ' found.']
[/Define_Tag]
```

This tag can be used as follows.

```
[Inline: -Findall, -Database='Contacts', -Table='People', -MaxRecords=4]
  [Ex_Results]
[/Inline]
```

```
➜ Showing 4 records out of 8 found.
```

For more information about the [Define_Tag] tag and custom tags see the *Custom Tags* chapter.

Miscellaneous Shortcuts

A number of shortcuts have been introduced in Lasso Professional 8 which will make coding Web sites even easier. There is no need to use any of these shortcuts. The equivalent syntax from earlier versions of Lasso will work fine.

- **Not Contains Symbol** – The negation of the contains symbol >> is now available as !>>. This makes it easy to check that a substring is not contained in a given string. The following example confirms that Green is not a part of Blue World.

  ```
  [('Blue World' !>> 'Green')]
  ```

  ```
  ➜ True
  ```

- **Equivalence Symbol** – The equals symbol == checks that two values are equal by casting them to the same data type. The new equivalence symbol === checks that two values are equal in both value and data type. The following example shows four expressions that are True using the equals symbol ==.

```
[('Alpha' == 'Alpha')] ➜ True
[('100' == 100)] ➜ True
[(3.00 == 3] ➜ True
[(True == 1)] ➜ True
```

  When the equivalency symbol === is used instead only the first expression is True. The rest of the expressions are False since the data types of the two operands are different. The second expression compares a string to an integer. The third expression compares a decimal to an integer. And, the fourth expression compares a boolean to an integer.

```
[('Alpha' === 'Alpha')] ➜ True
[('100' === 100)] ➜ False
[(3.00 === 3] ➜ False
[(True === 1)] ➜ False
```

- **String Concatenation** – Strings are now concatenated together without using the + symbol. In the following example database results are formatted without using the + symbol.

```
['Showing ' (Shown_Count) ' records of ' (Found_Count) ' found.']
```

  ➜ Showing 4 records out of 8 found.

- **Array Creation** – The : symbol can be used for array creation. Basically Array: is equivalent to simply :.

```
[: 'Alpha', 'Beta', 'Gamma']
```

  ➜ (Array: 'Alpha', 'Beta', 'Gamma')

- **Tag References** – The \ symbol can be used to reference a tag object based on its name. This allows the descriptions of tags to be fetched or for tags to be called with programmatically defined parameters. The following example shows what the output might be for the [Field] tag.

```
[Var: 'myTag' = \Field]
<br>[$myTag->Description]
<br>[$myTag->(Run: -Name='Field', -Params='First_Name')]
```

  ➜
A tag that returns a field value.

John

  See the *Advanced Programming Topics* chapter for more information.

Unicode Support

All strings in Lasso Professional 8 are represented internally by double-byte Unicode values. This makes it efficient to work with extended characters in a platform neutral fashion. All output from Lasso, whether to the client's Web browser or into a database, is formatted in UTF-8 by default.

UTF-8 is a Unicode standard that is backwards compatible with common 8-bit ASCII character sets. Any extended Unicode characters are encoded using an entity like &#4E26; where 4E26 is a hexadecimal number representing the Unicode value for the character.

Classic Lasso

Classic Lasso refers to the ability of Lasso to interpret command tags which are included in URLs or HTML forms and process the action described by those command tags before a format file is loaded.

In prior versions of Lasso this was the sole means of performing database actions. Since Lasso WDE 3.x it has been possible to perform database operation using the [Inline] ... [/Inline] tags instead. It is preferable to use this inline methodology for the following reasons.

- The database, table, and field names which are being accessed need never be revealed to the client.
- It is impossible for clients to create new URLs or HTML forms that perform unintended actions.
- The amount of data passed in URLs to and from the client can be greatly reduced. This can provide easier to read and easier to bookmark URLs.
- [Inline] ... [/Inline] tags support a number of advanced features like named inlines and accepting arrays of parameters which make it easier to separate the logic of a Web site from the presentation.
- Some actions such as issuing SQL statements to Lasso MySQL require using [Inline] ... [/Inline] functionality already.

Note: It is possible to enable Classic Lasso syntax in the ***Setup > Global > Syntax*** section of Lasso Administration, however since this functionality has been deprecated it will not be supported in a future version of Lasso. It is recommended that sites be transitioned over to inline methodology when used with Lasso Professional 8.

To update existing sites:

The [Action_Params] tag can be used to pass all parameters from the URL or HTML form that loaded the current page to an opening [Inline] ... [/Inline] tag.

In the result page, surround the part of the page that references database results with [Inline] ... [/Inline] tags. The opening [Inline] tag should have a single parameter of [Action_Params]. Often, the [Inline] ... [/Inline] tags can simply surround the entire page contents.

```
[Inline: (Action_Params)]
  ... Page Contents and Database Action Results ...
[/Inline]
```

The [Inline] ... [/Inline] tags must not be contained within any other [Inline] ... [/Inline] tags. The [Inline] ... [/Inline] tags must surround all [Records] ... [/Records], [Field], [Found_Count], [Link_...], [Error_CurrentError] and other tags that will return the database results.

In order to enhance security, command tags such as the -Database, -Table, and action can be added as parameters to the opening [Inline] tag. These parameters should be placed after the [Action_Params] parameter and will override any conflicting parameters from the URL or HTML form that loads the result page.

For example, the following [Inline] will always perform a -Search action on the People table of the Contacts database even if a -FindAll or -Delete action is specified in the URL.

```
[Inline: (Action_Params),
    -Search
    -Database='Contacts',
    -Table='People',
    -KeyField='ID']
  ... Page Contents and Database Action Results ...
[/Inline]
```

Now that the -Database, -Table, and action are specified in the opening [Inline] tag they can be removed from the URL or HTML form that loads the response page. Any command tags or name/value parameters which will be specified by the client should be left in the URL or HTML form, but static command tags can be moved as parameters into the opening [Inline] tag.

-Email... Command Tags

The -Email... commands are no longer supported in LDML 8. Enabling Classic Lasso syntax will not enable this functionality. The only way to send email through Lasso Professional 8 is to use the [Email_Send] tag.

To update existing sites:

- If emails are being sent using the [Inline] ... [/Inline] tags they can be modified to use the [Email_Send] tag as follows. The following shows the old approach based on -Email... command tags.

```
[Inline: -Email.Host='mail.example.com',
    -Email.To='me@example.com',
    -Email.From='me@exmaple.com',
    -Email.Subject='An Example Email Message',
    -Email.Format='email_format.lasso']
[/Inline]
```

The syntax for [Email_Send] is very similar. Notice that -Email.Format has been changed to -Body=(Include: ...). This is the preferred method of including another format file as the body of an email message.

```
[Email_Send: -Host='mail.example.com',
    -To='me@example.com',
    -From='me@exmaple.com',
    -Subject='An Example Email Message',
    -Body=(Include: 'email_format.lasso')]
```

- If the inline performs a database search in addition to sending an email message the two functions must be factored out as follows. The following example performs a search and sends a single email message.

```
[Inline: -FindAll,
    -Database='Contacts',
    -Table='People',
    -Email.Host='mail.example.com',
    -Email.To='me@example.com',
    -Email.From='me@exmaple.com',
    -Email.Subject='An Example Email Message',
    -Email.Format='email_format.lasso']
  [Records]
    ...
  [/Records]
[/Inline]
```

In the replacement the -Email... tags are removed from the opening [Inline] tag and the [Email_Send] tag is placed within the [Inline] ... [/Inline] tags, but not within the [Records] ... [/Records] tags. If [Email_Send] is placed in the [Records] ... [/Records] tags then one email for each found record will be sent.

```
[Inline: -FindAll,
    -Database='Contacts',
    -Table='People"]
  [Email_Send: -Host='mail.example.com',
      -To='me@example.com',
      -From='me@exmaple.com',
      -Subject='An Example Email Message',
      -Body=(Include: 'email_format.lasso')]
```

```
    [Records]
      ...
    [/Records]
  [/Inline]
```

- If emails are being sent using command tags in a URL they can be modified to use the [Email_Send] tag as follows.

  ```
  <a href="default.lasso?-Email.Host=mail.example.com&
      -Email.To=me@example.com&-Email.From='me@exmaple.com&
      -Email.Subject='An Example Email Message&
      -Email.Format=email_format.lasso"> Send Email </a>
  ```

  The URL should be simplified to contain just the name of the format file.

  ```
  <a href="default.lasso"> Send Email </a>
  ```

  The file default.lasso then must be augmented with the [Email_Send] tag. Notice that -Email.Format has been changed to -Body=(Include: ...). This is the preferred method of including another format file as the body of an email message.

  ```
  [Email_Send: -Host='mail.example.com',
      -To='me@example.com',
      -.From='me@exmaple.com',
      -.Subject='An Example Email Message',
      -Body=(Include: 'email_format.lasso')]
  ```

The same technique can be used to modify an HTML form. Simply remove the -Email... command tags from the form and place an [Email_Send] tag on the response file.

Decimal Precision

Decimal numbers are output using the fewest number of significant digits required. In prior versions of Lasso decimal numbers were always output by default using six significant digits. For example, the following math calculation outputs only two significant digits.

```
[2.02 + 2.0400] ➜ 4.06
```

In general the output from Lasso Professional 8 should be more readable than the output from Lasso Professional 6 so no changes to existing code should be required. In order to modify the number of significant digits that Lasso outputs the [Decimal->SetFormat] tag should be used.

Member Tags and Parentheses

Member tags which have multiple parameters must be surrounded by parentheses. Earlier versions of Lasso allowed some non-recommended syntax constructs to work. The new parser in Lasso Professional 8 is more strict about when parentheses are required around tag calls.

Specifically, the following syntax worked in Lasso Professional 6, but is no longer supported in Lasso Professional 8.

```
[(Date: '2003-12-01') -> Difference: (Date), -Hour)]
```

The code should be changed to the following. The parentheses around the [Date->Difference] tag clarify to which tag the -Hour parameter belongs.

```
[(Date: '2003-12-01') -> (Difference: (Date), -Hour))]
```

For best results any nested tags or member tags which require two or more parameters should be surrounded by parentheses.

PDF -Top Parameter

The -Top parameter in all PDF tags now always measures from the top margin of a document. In Lasso Professional 6 some of the PDF tags measured from the bottom of the page. See the LDML 8 Reference and the PDF chapter for additional details and a complete list of tags that have changed.

Global Variables

In Lasso Professional 8 global variables should always be manipulated using the [Global] tag rather than the [Variable] tag. The $ symbol can be used to refer to either global variables or page variables. If both a page variable and a global variable are defined with the same name then the $ symbol will return the value of the page variable.

Sites which do not use global variables do not require any modifications. The only sites that will require updates are those that used the [Variable] tag to refer to previously created global variables. These sites should be updated to use the [Global] tag instead.

[NSLookup]

Due to changes in Mac OS X 10.3 the [NSLookup] tag may not be able to perform reverse DNS lookups (from IP address to host name) on all DNS servers. Normal DNS lookups (from host name to IP address) should continue to work fine. This issue affects both Lasso Professional 6 and Lasso Professional 8 running on Mac OS X 10.3.

```
[NSLookup: '127.0.0.1']
```

[Repetition] Tag

The [Repetition] tag is deprecated in Lasso Professional 8 and will not be supported in the next version of Lasso. Converting loops that use the [Repetition] tag to use the modulus symbol % instead will result in faster code execution.

To update existing sites:

Replace the [Repetition: 2] tag with (Loop_Count % 2 == 0). The second operand of the % symbol should be whatever number was specified as a parameter to the [Repetition] tag.

For example, the following loop which makes use of [Repetition: 5] to display a message every fifth time through the loop.

```
[Loop: 100]
  [If: (Repetition: 5)]
    [Loop_Count] is divisible by 5!
  [/If]
[/Loop]
```

This loop can be rewritten using the modulus operator % as follows.

```
[Loop: 100]
  [If: (Loop_Count % 5 == 0)]
    [Loop_Count] is divisible by 5!
  [/If]
[/Loop]
```

The second loop will have exactly the same output as the first loop, but will run much faster.

[TCP_…] Tags

The [TCP_...] tags have been deprecated in favor of the new [Net] type and its member tags. Consult the *Advanced Programming Topics* chapter for details about the new tags.

[Else:If] and [Else_If]

The [Else] tag supports the functionality that was provided by the dedicated [Else:If] and [Else_If] tags in prior versions of Lasso. In order to streamline the language and provide faster code processing only the [Else] tag is supported in Lasso Professional 8.

For example, in the following code the [Else] tag is used to check several condition. Without a conditional parameter the [Else] tag is the default value for the [If] ... [/If] tags and always returns its value.

```
[If: $Condition == 'Alpha']
  ... Alpha ...
[Else: $Condition == 'Beta']
  ... Beta ...
[Else: $Condition == 'Gamma']
  ... Gamma ...
[Else]
  ... Default ...
[/If]
```

To update existing sites:

Change all [Else:If] and [Else_If] tags to [Else].

[LoopCount] and [LoopAbort]

In order to streamline the language and provide faster code processing the synonyms [LoopCount] for [Loop_Count] and [LoopAbort] for [Loop_Abort] are no longer supported in Lasso Professional 8.

To update existing sites:

Change all [LoopCount] tags to [Loop_Count] and all [LoopAbort] tags to [Loop_Abort].

Container Tags

In order to provide more efficient code execution it is now necessary for all container tags to be defined in LassoStartup. Any container tags which are defined within included files or library files will no longer function properly.

The [Define_Tag] tag now accepts two parameters for creating container tags. If the -Container keyword is used then a simple, non-looping container tag will be created. If the -Looping keyword is used then a looping container tag will be created. The only difference is that the [Loop_Count] will only be modified in looping container tags.

See the *Custom Tags* chapter for more details about defining custom container tags.

Custom Tags

All parameters and return values are now passed to custom tags by reference. Existing custom tags may need to be updated so that they do not cause any unwanted side effects or cause syntax errors.

The [PreCondition] and [PostCondition] tags are no longer supported. The -Type and -ReturnType parameters should be used in a custom tag definition in order to restrict the parameter types and return type for a custom tag.

Asynchronous custom tags do not have access to page variables from the page that called the custom tag. The documentation for Lasso Professional 6 was not clear on this point. Any variables which are required within the custom tag should be stored as globals or passed into the custom tag as parameters.

A number of other enhancements have been made to custom tags as well. See the *Custom Tags* chapter for more details about defining custom tags.

To update existing sites for parameter passed by reference:

Use different names for locals defined within a custom tag and for the parameters of the tag. For example, the following tag will cause a syntax error since it is not possible to modify the incoming literal changing its type from an integer into a string.

```
[Define_Tag: 'Ex_UpperCase', -Required='Value']
  [Local: 'Value' = (String_UpperCase: 'Value')]
  [Return: #Value]
[/Define_Tag]

[Ex_UpperCase: 1] ➜ Syntax Error
```

Instead, use a different name for the local variable within the tag. This code will work fine in Lasso Professional 8 and in Lasso Professional 6. By prefixing the local variables name with L_ there is no conflict with the incoming parameter names.

```
[Define_Tag: 'Ex_UpperCase', -Required='Value']
  [Local: 'L_Value' = (String_UpperCase: 'Value')]
  [Return: #L_Value]
[/Define_Tag]

[Ex_UpperCase: 1] ➜ 1
```

To update existing sites to remove pre- and post-conditions:

Use the -Type and -ReturnType parameters to specify the types for each parameter of a custom tag and the return type for the tag. Additional error checking can be performed with the custom tag itself.

For example, the following custom tag definition uses [PreCondition] and [PostCondition] to check that all of the tag's parameters and the tag's return value are strings.

```
Define_Tag: 'Ex_Concatenate',
    -Required='Param1',
    -Required='Param2';
  PreCondition: #Param1->Type == 'string';
  PreCondition: #Param2->Type == 'string';
  PostCondition: Return_Value == 'string';
  Return: #Param1 + #Param2;
/Define_Tag;
```

In Lasso Professional 8 this tag can be rewritten as the following. The -Type parameters specify the required type for the preceding -Required parameter. The -ReturnType parameter specifies the required type for the return value. If the parameters or return type are not of the proper type then an error will be returned.

```
Define_Tag: 'Ex_Concatenate',
    -Required='Param1', -Type='string',
    -Required='Param2', -Type='string',
    -ReturnType='string';
  Return: #Param1 + #Param2;
/Define_Tag;
```

XML Tags

The XML tags in Lasso Professional 8 have been re-implemented using native C/C++ libraries for greater speed and functionality. The behavior of some of the XML tags has changed and some modifications to existing sites may be required for full compatibility.

- [XML->Children], [XML->Attributes], and [XML->Contents] are now read-only. In Lasso Professional 6 these tags could be used to inspect or modify the XML data. In Lasso Professional 8 these tags can only be used to inspect XML data.
- [XML_Extract] – The [XML_Extract] tag will interpret some -XPath parameters differently. In particular, the new XML libraries interpret the XPath / to refer to the root of the XML data rather than the root tag in that data. /* can be used to refer to the root tag. The new [XML->Extract] tag is the preferred method of performing XPaths on XML data and uses the same XPath syntax as [XML_Extract].

- [XML->Children] – The [XML->Children] tag now includes additional text children for many XML tags. These children represent the text on either side of embedded tags. For example, the following <a> tag has three children some, the <b> tag, and text.

  ```
  <a href="..."> Some <b>Embedded</b> Text </a>
  ```

  Lasso Professional 6 would not provide access to these text children so the behavior of Lasso Professional 8 is preferred. The text children all have a name of text and may be empty if no text is specified between the various tags.

[Encode_ISOtoMac] and [Encode_MacToISO]

The [Encode_ISOtoMac] and [Encode_MacToISO] tags are not compatible with the Unicode strings that Lasso now uses to store strings. These tags must be modified in order for sites that use them to work properly with Lasso Professional 8.

To Update Existing Sites:

The output of the [Include_Raw], [File_Read], and other tags that might return data in a native character set have all been changed to the bytes type. The bytes type preserves the character set of the underlying data.

Note: See the earlier section on the ***Bytes Type*** for a full discussion of this new data type.

In Lasso Professional 6 a [File_Read] operation which read a Latin-1 (ISO 8859-1) file may have appeared like this. This code would translate the file from its native character set to Mac-Roman encoding.

```
[Variable: 'myFile' = (File_Read: 'myfile.text')]
[Variable: 'myString' = (Encode_ISOtoMac: $myFile)]
```

In Lasso Professional 8 the following code would be used. This code reads in the file as a byte stream and then uses [Bytes->ExportString] to convert the Latin-1 (ISO 8859-1) characters to the native Unicode-based double-byte strings that Lasso Professional 8 uses for character data.

```
[Variable: 'myFile' = (File_Read: 'myfile.text')]
[Variable: 'myString' = $myFile->(ExportString: 'iso8859-1')]
```

With this change the remainder of the code should not need to be changed since the end result has the same practical value, a natively encoded string.

Tag Name Changes

All tags from Lasso Professional 6 are supported in Lasso Professional 8 except for those listed in the table below. There are also a number of tag names which have changed or been deprecated in favor of new tags or methodologies in Lasso Professional 8.

The following table lists tags that are not supported in Lasso Professional 8. These tags must be replaced in order for sites to work properly in Lasso Professional 8.

Table 5: Unsupported Tags

LDML 6 Tag	LDML 8 Tag Equivalent
[Encode_MacToISO]	[Bytes->ExportString]
[Encode_ISOtoMac]	[Bytes->ExportString]

The following table lists the tag names that have been changed in Lasso Professional 8 since the release of Lasso Professional 6. The old versions of each tag will continue to work, but their use has been deprecated. Any new development in Lasso Professional 8 should use the new versions of the tag names.

Table 6: Tag Name Changes

LDML 6 Tag	LDML 8 Tag Equivalent
[Null->Up]	[Null->Parent]
[String->Length]	[String->Size]

The following table lists the tags from Lasso Professional 6 which have been deprecated in Lasso Professional 8 and what code equivalent should be used. The deprecated versions of these tags will continue to work, but any new development in Lasso Professional 8 should use the suggested code equivalent rather than the deprecated tags.

Table 7: Deprecated Tags

LDML 6 Tag	LDML 8 Tag Equivalent
[Date_GetCurrentDate]	[Date]
[Date_GetDay]	[Date->Day]
[Date_GetDayOfWeek]	[Date->DayOfWeek]
[Date_GetHour]	[Date->Hour]
[Date_GetMinute]	[Date->Minute]
[Date_GetMonth]	[Date->Month]
[Date_GetSecond]	[Date->Second]
[Date_GetYear]	[Date->Year]
[Error_NoRecordsFound]	Check for whether [Found_Count] is zero.
[PostCondition]	-ReturnType in [Define_Tag]
[PreCondition]	-Type or -Criteria in [Define_Tag]
[Repetition]	Modulus Symbol %
[TCP_Close]	[Net->Close]
[TCP_Open]	[Net->Connect]
[TCP_Send]	[Net->Read], [Net->Write]

22

Chapter 22
Upgrading From Lasso Professional 5

This chapter contains important information for users of Lasso Professional 5 who are upgrading to Lasso Professional 8. Please read through this chapter before attempting to run solutions in Lasso Professional 8 that were originally developed for an earlier version of Lasso.

The upgrading chapters are cumulative so this chapter should be read in conjunction with the preceding chapters for full information about changes to Lasso.

Topics in this chapter include:

- *Introduction* includes general information about what has changed in Lasso Professional 8.
- *Tag Name Changes* details the tag names which have been changed.
- *Syntax Changes* contains information about what LDML syntax constructs have changed.
- *Lasso MySQL* contains information about changes made to the Lasso Connector for Lasso MySQL.

Introduction

This chapter includes the upgrading instructions from Lasso Professional 5 to Lasso Professional 8. If a site is being upgraded from Lasso Professional 5 the items in this chapter should be applied first, followed by the items in the prior chapter about upgrading from Lasso Professional 6, and then the items in the chapter about upgrading from Lasso Professional 7.

Lasso Studio and LDML Updater

Lasso Studio includes an LDML Updater that can be used on code from earlier versions of Lasso to bring it into compliance with the latest version of LDML. See the documentation for Lasso Studio for more information.

Tag Name Changes

All tags from Lasso Professional 5 are supported in Lasso Professional 8 except for those listed in the table below. There are also a number of tag names which have changed or been deprecated in favor of new tags or methodologies in Lasso Professional 8.

The following table lists tags that are not supported in Lasso Professional 8. These tags must be replaced in order for sites to work properly in Lasso Professional 8.

Table 1: Unsupported Tags

LDML 5 Tag	LDML 8 Tag Equivalent
[Encode_MacToISO]	[Bytes->ExportString]
[Encode_ISOtoMac]	[Bytes->ExportString]

The following table lists the tag names that have been changed in Lasso Professional 8 since the release of Lasso Professional 5. The old versions of each tag will continue to work, but their use has been deprecated. Any new development in Lasso Professional 8 should use the new versions of the tag names.

Table 2: Tag Name Changes

LDML 5 Tag	LDML 8 Tag Equivalent
[Null->Up]	[Null->Parent]
[String->Length]	[String->Size]

The following table lists the tags from Lasso Professional 5 which have been deprecated in Lasso Professional 8 and what code equivalent should be used. The deprecated versions of these tags will continue to work, but any new development in Lasso Professional 8 should use the suggested code equivalent rather than the deprecated tags.

Table 3: Deprecated Tags

LDML 5 Tag	LDML 8 Tag Equivalent
[Date_GetCurrentDate]	[Date]
[Date_GetDay]	[Date->Day]
[Date_GetDayOfWeek]	[Date->DayOfWeek]
[Date_GetHour]	[Date->Hour]
[Date_GetMinute]	[Date->Minute]
[Date_GetMonth]	[Date->Month]
[Date_GetSecond]	[Date->Second]
[Date_GetYear]	[Date->Year]
[Error_NoRecordsFound]	Check for whether [Found_Count] is zero.
[PostCondition]	-ReturnType in [Define_Tag]
[PreCondition]	-Type or -Criteria in [Define_Tag]
[Repetition]	Modulus Symbol %
[TCP_Close]	[Net->Close]
[TCP_Open]	[Net->Connect]
[TCP_Send]	[Net->Read], [Net->Write]

Syntax Changes

Lasso Professional 7 introduces changes to some of the core syntax rules of LDML. Most of these changes were made to improve the reliability and error reporting of Lasso. Some of these changes may require you to rewrite portions of your existing Lasso-based solutions for full compatibility with Lasso Professional 8. This section describes each change, why it was made and how to update existing format files.

Table 4: Syntax Changes

Syntax Change	Description
No Process Tags	New [NoProcess] … [/NoProcess] tags allow a portion of a page to be passed to the browser without being processed.
Strict Syntax	A strict syntax option allows errors such as non-hyphenated parameters, non-quoted variables, and undefined tags to be reported as syntax errors.
Date Data Type	Date operations have been converted to a new date data type. LDML 5 date tags have some modifications.
Integer Rounding	The [Integer] tag now rounds to the nearest integer instead of truncating.
No Records Found	The [Error_NoRecordsFound] tag has been deprecated. Check whether [Found_Count] equals zero instead.

No Process Tags

Lasso Professional 8 includes a container tag [NoProcess] ... [/NoProcess] that instructs the LDML parser to ignore its contents. This allows code from other programming languages to be passed through to the browser without any processing by Lasso. These new tags do not require any changes to existing Lasso Web sites, but may make transitioning from older versions of Lasso easier.

The [NoProcess] ... [/NoProcess] tags must be embedded in a page exactly as written with no extra spaces or parameters within the square brackets. They cannot be used within LassoScript.

To instruct Lasso to ignore a portion of a page:

Use the [NoProcess] ... [/NoProcess] tags. In the following example, the entire contents of a JavaScript code block is ignored by Lasso. Any array references within the JavaScript will not be interpreted by Lasso as square bracketed tags.

```
[NoProcess]
  <script language="JavaScript">
    … JavaScript Expressions …
  </script>
[/NoProcess]
```

Strict Syntax

With strict syntax the following rules are enforced:

- All keyword parameters to built-in and custom tags must include a hyphen. This helps to find unknown tag parameters and to catch other common syntax errors.
- All string literals must be surrounded by quotes. This helps to prevent accidental calls to tags, to identify undefined variables, and to catch other common syntax errors.
- All tag calls must be defined. Unknown tags will no longer simply return the tag value as a string.

With strict syntax any of the errors above will be reported when a page is first loaded. They must be corrected before the code on the page will be executed. When upgrading to Lasso Professional 8 it is advisable to first try existing Lasso Professional 5 sites and correct any errors that are reported.

To update existing sites for strict syntax:

If a site is relatively small then the easiest method is to load each Web page and see if any errors are reported. The following tips can be used for a more methodical search.

- Check that all string literals are surrounded by quotes. Quotes are not necessary around integers or decimal numbers, hyphenated keyword parameters, tag names, or variable names when used with the & or # symbols.
- Check that all keywords in tag calls are preceded by a hyphen. Keyword and keyword/value parameters must be preceded by a hyphen, but do not need to be quoted. Name/value parameters should include quotes around both the name and value (unless they are numbers).
- Check that all command tags used within opening [Inline] tags are preceded by a hyphen. Quotes are not necessary around command tags, even when they are specified within an array.
- Check that all client-side JavaScript is formatted properly. JavaScript should either be included in [NoProcess] ... [/NoProcess] tags or HTML comment tags <!-- ... --> which ensure that no LDML code within is

processed. Or, any square brackets which are required within the JavaScript should be output from an [String] tag.

```
[String: '[array[4]]']
➜ [array[4]]
```

Date Data Type

The date tags from LDML 5 have been replaced by new date and duration data types in LDML 7. This change should not require any changes to existing code, but many LDML 5 tags have been deprecated and many other operations are significantly easier using the new tags. See the *Date and Time Operations* chapter for full documentation of the new date and duration data types.

Some highlights of the new date and duration data types include:

- The [Date] tag can be used in place of [Date_GetCurrent] date to return the current date and time.
- The [Date] tag now recognizes MySQL date formats natively as well as United States date formats.
- The [Date_Get...] tags have been replaced by member tags which perform equivalent functions. [Date->Day] returns the current day of the month and [Date->Year] returns the current 4-digit year.
- The week number can be output using [Date->Week] and the current day of the year can be output using [Date->DayOfYear].
- The duration between two dates can be output using the subtraction symbol [(Date) - (Date: 3/4/1984')] or a duration can be added to a date using the addition symbol [(Date) + (Duration: -Hour=1)].
- The output format for all date tags on a format file can be set using [Date_SetFormat]. For example, [Date_SetFormat: '%Q %T'] will set all dates to output in MySQL date format.
- Individual dates can be formatted using [Date->Format]. For example [Date->(Format: '%Q %T')] will output the current date in MySQL date format.
- Upon casting a date type, Lasso 7 automatically adjusts invalid dates to be a valid equivalent, where Lasso 5 returns a null value instead of an invalid date. For example, 9/31/2002 is an invalid date because there are not 31 days in September. The expression [Date:'9/31/2002'] returns 10/1/2002 in Lasso 7, whereas [Date:'9/31/2002'] returns no value in Lasso 5.

Integer Rounding

The [Integer] tag now rounds decimal values to the nearest integer. In Lasso Professional 5 the [Integer] tag instead truncated decimal values to the next lowest integer. The new process yields a more accurate result. In general, no changes to existing sites should be necessary.

To update existing sites:

Use the [Math_Floor] tag to return the next lowest integer rather than using the [Integer] tag.

No Records Found

The [Error_NoRecordsFound] tag has been deprecated. This tag will continue to work with the Lasso Connector for FileMaker Pro, but may not work with MySQL databases or with third party data source connectors.

To update existing sites:

Change any code which uses [Error_NoRecordsFound] to instead check whether [Found_Count] is equal to zero. For example, the following code from LDML 5:

```
[If: (Error_CurrentError) == (Error_NoRecordsFound)]
  No records were found!
[/If]
```

Can be written as follows in LDML 7:

```
[If: (Found_Count) == 0]
  No records were found!
[/If]
```

Lasso MySQL

A number of changes have been made to the Lasso Connector for Lasso MySQL in order to make its behavior match that of the Lasso Connector for FileMaker Pro. These changes will not in general require any changes to existing Lasso Professional 5 sites.

Table 5: Lasso MySQL Syntax Changes

Syntax Change	Description
-Add and -Update	The -Add and -Update actions now return the record which was just added to the database or updated within the database by default.
Full Text Searching	The ft operator allows full text indices to be searched. Lasso Administration allows full text indices to be created.
Random Sorting	The -SortRandom keyword can be used to return MySQL results in random order.
Regular Expression Searching	The rx and nrx operators allow regular expression searches to be performed and all records which match or do not match the results to be returned.
Searching for Distinct Values	The -Distinct keyword allows only distinct records from search results to be returned.
Searching for Null Values	The inline tag now recognizes Null as a value distinct from the empty string allowing Null values in databases to be found.
Using LIMIT Options	The -UseLimit keyword instructs Lasso to use LIMIT options to select the found records to show rather than using native methods. This can result in better performance on large databases with large found sets.
Value Lists	Values lists are now supported for ENUM and SET data types within MySQL databases.

See the *MySQL Data Sources* for complete documentation of these changes.

23

Chapter 23
Upgrading From Lasso WDE 3.x

This chapter contains important information for users of Lasso Web Data Engine 3.x who are upgrading to Lasso Professional 8. Please read through this chapter before attempting to run solutions in Lasso Professional 8 that were originally developed for an earlier version of Lasso.

The upgrading chapters are cumulative so this chapter should be read in conjunction with the preceding chapters for full information about changes to Lasso.

Topics in this chapter include:

- *Introduction* includes general information about what has changed in Lasso Professional 8.
- *Syntax Changes* contains information about what LDML syntax constructs have changed since Lasso WDE 3.x and how to update format files which use those syntax constructs.
- *Tag Name Changes* details the tag names which have been changed in LDML 8 since LDML 3.
- *Unsupported Tags* lists the LDML 3 tags that are no longer supported in Lasso Professional 8.
- *FileMaker Pro* contains information about how to update a solution which depended on the Apple Event based FileMaker Pro data source module to the new Lasso Connector for FileMaker Pro.

This chapter does not attempt to cover every issue that users of versions of Lasso *prior* to Lasso Web Data Engine 3.x may encounter.

Introduction

This chapter includes the upgrading instructions from Lasso Web Data Engine 3.x to Lasso Professional 5. If a site is being upgraded from Lasso Web Data Engine 3.x the items in this chapter should be applied first, then the items in the preceding upgrading chapters.

Sites that are upgraded from Lasso Web Data Engine 3.x to Lasso Professional 8 will in general require significant modifications.

Lasso Studio and LDML Updater

Lasso Studio includes an LDML Updater that can be used on code from earlier versions of Lasso to bring it into compliance with the latest version of LDML. See the documentation for Lasso Studio for more information.

Syntax Changes

Lasso Professional 8 introduces changes to some of the core syntax rules of LDML. Some of these changes may require you to rewrite portions of your existing Lasso-based solutions. This section describes each change, why it was made and how to update existing format files.

Table 1: Syntax Changes

Syntax Change	Description
Square Brackets	All expressions in square brackets are now interpreted.
Commas	Commas are no longer allowed after tag names.
Keywords	Keyword names now always begin with a hyphen.
Encoding Keywords	The default is to HTML encode outermost substitution tags and apply no encoding to nested sub-tags.
Else If	The [Else:If] tag is no longer supported. The [Else] tag has been enhanced to provide the same functionality.
Include	The [Include] tag now returns an error if the specified file does not exist.
Post Inline	The [Post_Inline] tag has been replaced by a new scheduling facility accessed through the [Event_ Schedule] tag.
SQL Inline	The [SQL_Inline] tag has been replaced by a new -SQL command tag which can be used in a normal [Inline] tag.
File Tags and Logging	The new distributed architecture means these tags work only on files accessible by Lasso Service.

Line Endings	The default line endings on Mac OS X are different from those for Mac OS 9.
JavaScript	Special care must be taken to ensure that array references in JavaScript are not interpreted by Lasso.
Macros	Macros are no longer supported. Much of their functionality can be achieved through custom tags.
Numeric Literals	Numeric literals must not be written with quotes. The conversion of strings to numeric values has changed.
Mathematical Precision	Precision is handled automatically by the new mathematical expressions and symbols and can be set explicitly using [Decimal->SetFormat] tag.
Double Quotes	Single quotes are preferred for designating string literals.
Restrictions	Restrictions on maximum values for math operations and looping tags have been eased.

Square Brackets

In earlier versions of Lasso, only tag names which were recognized by Lasso would be interpreted. In Lasso Professional 8, all square bracketed expressions are interpreted whether they contain a valid LDML tag or not. This allows expressions and member tags to be used within square brackets and allows custom tags to be used.

For example, the following expressions would all have been ignored in earlier versions of Lasso, but will be interpreted as indicated by Lasso Professional 8.

[45] ➜ 45
[1 + 2] ➜ 3
[blue] ➜ blue
['aqua' + 'marine'] ➜ aquamarine

If square brackets are used decoratively on a page, e.g. to surround link names, they will be stripped out by Lasso Professional 8.

Note: See the section on JavaScript that follows for tips on using square brackets within client-side JavaScript contained in an LDML format file.

To update existing sites:

There are several options to update existing sites depending on how the square brackets are being used on a page.

- Use the HTML entities for square brackets. These include [for [and] for]. The following example would display a link name surrounded by square brackets.

```
<a href="default.lasso"> &#91; Home &#93; </a>
```

- Use the LDML [String] tag to output an expression that includes square brackets. Lasso will not interpret the output of LDML tags so square brackets can be safely displayed on a page in this way. The following example would display a link name surrounded by square brackets.

  ```
  <a href="default.lasso"> [String: '[Home]'] </a>
  ```

  The expression can also be written without the [String] tag.

  ```
  <a href="default.lasso"> ['[Home]'] </a>
  ```

  Note: Any string literals which are output in this way should always be surrounded by single quotes, otherwise there is a danger that they might be interpreted as a tag.

- Create a custom tag that outputs text surrounded by square brackets. The following simple [Define_Tag] can be placed at the top of any page that requires it.

  ```
  [Define_Tag: 'Bracket']
    [Return: (Params->(Get: 1)]
  [/Define_Tag]
  ```

  This tag can then be called to display a link name surrounded by square brackets.

  ```
  <a href="default.lasso"> [Bracket: 'Home'] </a>
  ```

Commas

In earlier versions of Lasso, commas could optionally be used following the tag name, before any parameters of the tag. Although this syntax hasn't been recommended for some time there are still examples of it in the Lasso Web Data Engine 3.x documentation and in some Lasso-based Web sites. The following example shows the tag construct with a comma following the tag name.

```
[Tag_Name, Parameters]                    (No longer supported)
```

This syntax was particularly common with tags that took only a single keyword. For example, both of the following tags were commonly written with a comma following the tag name.

```
[Server_Date, Short]                      (No longer supported)
[Error_CurrentError, ErrorCode]           (No longer supported)
```

Using a colon after the tag name is now mandatory in Lasso Professional 8. This change was made in order to facilitate parsing of more complex expressions. The tag examples above must now be written as follows with

a colon after the tag name. The following example also demonstrates the new method of specifying keyword names with a leading hyphen.

```
[Server_Date: -Short]
[Error_CurrentError: -ErrorCode]
```

To update existing sites:

Use a regular expression to correct format files that contain the older comma syntax. Most text editors and Web authoring environments can perform a find/replace using regular expressions.

1 Search for the following regular expression pattern to find tags in square brackets which have a comma after the tag name:

```
\[([A-Za-z_]+),([^\]]*)\]
```

Use this pattern as the replacement value:

```
[\1:\2]
```

2 Search for the following regular expression pattern to find sub-tags in parentheses which have a comma after the tag name:

```
\(([A-Za-z_]+),([^\)]*)\)
```

Use this pattern as the replacement value:

```
(\1:\2)
```

What the first regular expression does is search for a square bracket followed by a tag name, a comma, then any characters up until the closing square bracket. The replacement pattern inserts an opening square bracket, the tag name, a colon, the contents after the comma, and a final closing square bracket. The second regular expression performs the same steps with parentheses instead of square brackets.

Keywords

All keywords and keyword/value parameters (formerly named parameters) start with a hyphen in LDML 8. This used to be an option for command tags used within the [Inline] tag in Lasso Web Data Engine 3.x, but is now required for all tags. Most tags which were supported in Lasso Web Data Engine 3.x will continue to accept keywords without the leading hyphen so Lasso Web Data Engine 3.x solutions do not need to be rewritten. However, all keyword names without leading hyphens have been deprecated and are not guaranteed to work in future versions of Lasso.

This change was made so that LDML keywords can be clearly differentiated from user-defined name/value parameters and from tag names. This becomes especially important as users start to create custom tags which might have the same name as the keywords of existing tags.

To update existing sites:

1 Locate all keyword names that do not begin with a hyphen. For example, the following [Server_Date] tag contains both a tag-specific keyword and an encoding keyword, neither of which has been written with a hyphen:

```
[Server_Date: Short, EncodeNone]
```

The following [Inline] tag contains several command tags or keyword/value parameter that have not been written with hyphens:

```
[Inline:
  Database='Contacts',
  Table='People',
  'State'='WA',
  Search]
```

2 Change the keywords so their names start with a hyphen. The [Server_Date] tag is changed to the following with each keyword name beginning with a hyphen:

```
[Server_Date: -Short, -EncodeNone]
```

The [Inline] tag is changed to the following with each command tag and keyword/value parameter written with a hyphen:

```
[Inline:
  -Database='Contacts',
  -Table='People',
  'State'='WA',
  -Search]
```

3 Do not change user-defined name/value parameters. In the preceding example 'State'='WA' is not changed when updating the tag for compliance with Lasso Professional 8.

Note: The name 'State' has quotes around it in the preceding examples. All string literals should be specified with single quotes. This ensures that they will not be misidentified as a sub-tag or a keyword.

Encoding Keywords

The use of encoding keywords in substitution tags has been altered in Lasso Professional 8. All substitution tags which are used as sub-tags now have a default encoding of -EncodeNone. Only the outermost substitution tag (i.e. a tag in square brackets) has a default encoding of -EncodeHTML. This change was made in order to make Lasso easier to use for new users and to reduce the length of nested tag expressions.

The following example demonstrates the benefits of the new Lasso Professional 8 syntax. In LDML 3, the following [String_Concatenate] tag contains many sub-tag parameters which all have EncodeNone specified.

```
[String_Concatenate:
  (Field: 'First_Name', EncodeNone), ' ',
  (Field: 'Middle_Name', EncodeNone), ' ',
  (Field: 'Last_Name', EncodeNone)]
```

The preceding tag can be written as follows in LDML 8. Since the default encoding of each of the sub-tags is -EncodeNone the encoding keyword can be omitted. The resulting code is considerably shorter and easier to read.

```
[String_Concatenate: (Field: 'First Name'), ' ',
  (Field: 'Middle Name'), ' ', (Field: 'Last Name')]
```

The default encoding for the outermost tag in LDML 8 is still -EncodeHTML in order to maintain the security of sites powered by Lasso Professional 8. If a field is placed on a page without encoding then any JavaScript or HTML that the code contains will be live on the Web page. Only HTML from trusted sources should be allowed on your Web site.

LDML 8 includes additional encoding enhancements. Please see the *Encoding* chapter for full details of how [Encode_Set] can be used to change the default encoding of a page and more.

Note: -EncodeHTML is now a valid encoding keyword which performs the same encoding as that which is performed if no encoding keyword is specified in an outermost substitution tag.

To update existing sites:

Encoding keywords still work as they did in Lasso Web Data Engine 3.x if they are specified in every tag. Existing code will generally work after an upgrade to Lasso Professional 8. However, the following use of encoding keywords will need to be rewritten.

1. Locate tags where the outermost tag has an EncodeNone encoding keyword and the sub-tags do not have any encoding keywords. For example, the following [String_Concatenate] tag has an EncodeNone keyword and the two [Field] tags do not have any encoding keywords.

   ```
   [String_Concatenate: EncodeNone, (Field: 'First Name'), ' ', (Field: 'Last Name')]
   ```

2. Rewrite the tag by removing the EncodeNone keyword from the outermost tag. In the resulting LDML 8 code, no encoding keywords are required.

   ```
   [String_Concatenate: (Field: 'First Name'), ' ', (Field: 'Last Name')]
   ```

Note: In the LDML 3 code, the [Field] sub-tags were automatically HTML encoded. The EncodeNone keyword in the outermost [String_Concatenate] tag ensured that double encoding was not applied. Since Lasso 7 does not encode sub-tags by default, the encoding keyword is no longer needed.

Else If

The [Else:If:] tag has been eliminated as a distinct tag, but the concept is still supported. [Else:If: Condition] is now syntactically equivalent to [Else: (If: Condition)] and the [Else] and [If] tags have been enhanced so that much of the old behavior of the [Else:If:] tag is preserved.

The following [Else:If:] tag will not work as expected in Lasso Professional 8 because the condition will be misinterpreted:

```
[Else:If: 'abc' == 'abc']
```

The condition will be interpreted as if the following tag had been written:

```
[Else: (If: 'abc') == 'abc']
```

The (If: 'abc') expression will return True and this will be compared to 'abc'. Since True is not equal to 'abc' this clause in the conditional will not be executed.

Note: If called individually, the [If] and [Else] tags will return the value of the specified conditional expressions parameter rather than returning an error about an unclosed container tag.

To update existing sites:

- Use parentheses around all conditional expressions. The following [Else:If] tag will work correctly in either Lasso Professional 8 or Lasso Web Data Engine 3.x:

  ```
  [Else:If: ('abc' == 'abc')]
  ```

- Change the [Else:If] tag to [Else]. Lasso 7's [Else] tag has been enhanced so that it now works like the old [Else:If] tag if a condition is specified, but is still the marker for the default clause of the conditional if no condition is specified. The following tag will work in Lasso Professional 8, but not in Lasso Web Data Engine 3.x:

  ```
  [Else: 'abc' == 'abc']
  ```

Include

The [Include] tag now validates whether the specified file exists and returns an error if an invalid file path is specified. This means that programmatically constructed [Include] statements need to take a precaution so errors won't be shown to the site visitor.

To update existing sites:

The [Protect] ... [/Protect] tags can be used to suppress the error that is reported by the [Include] tag. The following code will not return an error, even though the file fake.lasso does not exist.

```
[Protect]
  [Include: 'fake.lasso']
[/Protect]
```

Post Inline

The [Post_Inline] tag is no longer supported in Lasso Professional 8. This tag relied on access to files which Lasso Service might not be able to locate because they could be on a separate machine. The replacement for [Post_Inline] is called [Event_Schedule] and has the following format:

```
[Event_Schedule:
  -Start=(Date, Defaults to Today),
  -End=(Date, Defaults to Never),
  -URL=(URL to Execute, Required)
  -Repeat=(True/False, Defaults to True if -Delay is set and False otherwise),
  -Restart=(True/False, Defaults to True),
  -Delay=(Minutes, Required if -Repeat is True),
  -Username=(Username for Authentication, Optional),
  -Password=(Password for Authentication, Optional)]
```

This tag schedules the execution of the response URL at a specific start date and time. The URL is fetched just as if a client had visited it through a Web browser. After the task is performed, it is optionally repeated a specified number of minutes later until the end date and time is reached. If the restart parameter is set to True then the repeating task will be rescheduled even after server restarts. Please see the *Control Tags* chapter for complete documentation of the syntax of [Event_Schedule].

To update existing sites:

Sites that rely on [Post_Inline] tags will need to be rewritten. The following steps must be taken:

1 Determine the URL of the post-inline response page you were calling.

2 Change the initial [Post_Inline] tag to the equivalent [Event_Schedule] tag using date calculations if necessary to determine the start date and time.

3 If the [Post_Inline] tag rescheduled itself in the response page then either the rescheduling call must be changed to an equivalent [Event_Schedule] tag or the automatic repeat feature of [Event_Schedule] can be used in its place.

SQL Inline

The [SQL_Inline] tag is no longer supported in Lasso Professional 8. This tag has been replaced by a more versatile -SQL command tag that can be used as the database action within any [Inline] tag.

```
[Inline: -SQL='...SQL Statement...]
  ... Inline Results ...
[/Inline]
```

The -SQL command tag can be used to issue SQL statements to the included Lasso MySQL data source or to any MySQL data source accessed through the Lasso Connector for MySQL. The -SQL command tag may also be supported by third party data source connectors. Please see the *MySQL Data Sources* chapter for more information about using this tag.

To update existing sites:

Sites that rely on [SQL_Inline] tags will need to be rewritten. The following steps must be taken:

1 Change the opening and closing [SQL_Inline] ... [/SQL_Inline] tags to [Inline] ... [/Inline] tags. For example, following is a [SQL_Inline] that searches the People table of the Contacts database.

```
[SQL_Inline: Datasource='Contacts',
    SQLStatement='SELECT First_Name, Last_Name from People']
  ...
[/SQL_Inline]
```

The first step is to change this to the following [Inline] ... [/Inline] tags, then to perform the remainder of the steps to complete the transformation.

```
[Inline: Datasource='Contacts',
    SQLStatement='SELECT First_Name, Last_Name from People']
  ...
[/Inline]
```

2 Change the Datasource parameter to a -Database keyword/value parameter. Ensure that the database name is valid in the current Lasso Professional 8 setup.

```
[Inline: -Database='Contacts',
    SQLStatement='SELECT First_Name, Last_Name from People']
  …
[/Inline]
```

Note: The ODBC data source module is not provided with Lasso Professional 8. Data sources must be available through the included Lasso Connector for MySQL or a third-party data source connector.

3 Change the SQLStatement parameter to a -SQL command tag. Change any table references within the SQL statement so they reference both the database and table name, not just the table name.

```
[Inline: -Database='Contacts',
    -SQL='SELECT First_Name, Last_Name from Contacts.People']
  …
[/Inline]
```

4 If LDML tags are used within the SQL statement then they will need to be changed to expressions. In the following example, the name of the table is stored in a variable named MyTable and referenced using a square bracketed expression within the SQLStatement. This is no longer valid syntax.

```
[Var_Set: 'MyTable'='People']
[SQL_Inline: Datasource='Contacts',
    SQLStatement='SELECT First_Name, Last_Name from [Var: 'MyTable']']
  …
[/SQL_Inline]
```

In Lasso Professional 8, this is changed to the following string expression that concatenates the value of the variable to the SQL statement explicitly.

```
[Variable: 'MyTable'='Contacts.Table']
[Inline: -Database='Contacts',
    -SQL='SELECT First_Name, Last_Name from ' + (Variable: 'MyTable')]
  …
[/Inline]
```

Please see the *MySQL Data Sources* chapter for more examples of creating SQL statements for use with the -SQL command tag and for information about how to display the results within the [Inline] ... [/Inline] tags.

File Tags and Logging

Lasso Professional 8 features a distributed architecture where Lasso Service can run on a different machine from the Web server on which Lasso Connector for IIS or Lasso Connector for Apache is installed. The file

tags and logging tags can only manipulate files on the machine which is hosting Lasso Service. They have no access to the machine which is hosting a Lasso Web server connector.

If you are running both Lasso Service and your Web serving software on a single machine then no changes to existing file and logging tags should be necessary when you upgrade to Lasso Professional 8. Otherwise, please consult the *Files and Logging* chapter for more information about how to access files in a two machine system.

Note: In contrast to the file and logging tags, the [Include] tag works exclusively with files from the Web serving machine. No changes should be necessary to your sites which use the [Include] tag unless you are using it to access log files or files which have been manipulated by the file tags. Use the [File_Read] tag for these situations.

Line Endings

Files created in Mac OS X, Windows 2000, or versions of the Mac OS 9 and earlier each have a different standard for line endings. This can cause confusion when moving files from one platform to another or from an earlier version of the Mac OS to Mac OS X. *Table 11: Line Endings* summarizes the different standards.

Table 2: Line Endings

Tag	Description
Mac OS X	Line feed: \n. Each line is ended with a single line feed character.
Mac OS 9 and Earlier	Carriage return: \r. Each line is ended with a single carriage return character.
Windows 2000	Line feed and carriage return: \r\n. Each line is ended with both a line feed and a carriage return character.

Line ending differences are handled automatically by Web servers and Web browsers so are generally only a concern when reading and writing files using the [File_...] tags. The following tips make working with files from different platforms easier.

- The default line endings used by the [File_LineCount] and [File_ReadLine] tags match the platform default. They are \n in Mac OS X and \r\n in Windows 2000. The default for Lasso Web Date Engine 3.x's file tags on Mac OS 9 and earlier was \r.
- Specify line endings explicitly in the [File_LineCount] and [File_ReadLine] tags. For example, the following tag could be used to get the line count for a file that was originally created on Mac OS 9.

```
[File_LineCount: 'FileName.txt', -FileEndOfLine='\r']
```

Or, the following tag could be used to get the line count for a file that was originally created on Windows 2000.

```
[File_LineCount: 'FileName.txt', -FileEndOfLine='\r\n']
```

- Many FTP clients and Web browsers will automatically translate line endings when uploading or downloading files. Always check the characters which are actually used to end lines in a file. Don't assume that they will automatically be set to the standard of either the current platform or the platform from which they originated.
- A text editor such as Bare Bones BBEdit can be used to change the line endings in a file from one standard to another explicitly.

JavaScript

Since Lasso will interpret any expressions contained within square brackets special care must be taken to ensure that square brackets which are used for array accesses within client-side JavaScripts are not interpreted.

- Use the [NoProcess] ... [/NoProcess] tags to instruct Lasso not to interpret any of the code contained therein.

```
[NoProcess]
  <script language="JavaScript">
    ... JavaScript Expressions ...
  </script>
[/NoProcess]
```

- Lasso will not interpret any expressions that are contained within HTML comments. The following common method of surrounding a JavaScript with HTML comments ensures that neither Lasso nor older Web browsers will interpret the contents of the JavaScript.

```
<script language="JavaScript">
  <!--
    ... JavaScript Expressions ...
  // -->
</script>
```

The opening <!-- expression is ignored by the JavaScript interpreter. The closing --> expression is formatted as part of a JavaScript comment by including it on a line starting with the JavaScript comment characters //.

- If LDML tags need to be used within a client-side JavaScript then the HTML comment can be opened and closed in order to allow Lasso to process portions of the JavaScript, but not others.

```
<script language="JavaScript">
  <!--
    … JavaScript Expressions …
  // -->
  var VariableName='[… LDML Expression …]';
  <!--
    … JavaScript Expressions …
  // -->
</script>
```

- The LDML [String] tag can be used to output short JavaScript segments that need to make use of square brackets. This technique is useful for JavaScript that is contained within the attributes of HTML tags or for JavaScripts that contain only a few square brackets.

  In the following example, a select statement contains an [String] tag in its onChange handler that returns a JavaScript expression containing square brackets to report which option was selected.

```
<select name="Select" multiple size="4"
    onChange="[String: 'alert(this.options[this.selectedIndex])']">
  <option value="Value"> Value </option>
  …
</select>
```

Macros

Macros are not supported in Lasso Professional 8. See the Extending Lasso Guide for information about rewriting macros as custom tags using the new [Define_Tag] tag in LDML 8.

Numeric Literals

In LDML 8 there is a distinction between number values and string values. This distinction makes advanced data type specific member tags and expression symbols possible. Strings are always enclosed in single quotes. Numbers are never enclosed in quotes. If you use quotes around a numeric literal then symbols which are used to manipulate that literal may assume it is a string.

For example, the following code specifies a mathematical operation, the numerical addition of 1 and 2:

```
[1 + 2] ➜ 3
```

In contrast, the following code specifies a string operation, the string concatenation of the string '1' and the string '2', because the numbers are contained in quotes:

['1' + '2'] ➜ 12

The legacy math and string tags from Lasso Web Data Engine 3.x still perform automatic type conversions on their arguments. This ensures that existing sites will not need to be rewritten. Both of the following tags return the same result despite the fact that the parameters are specified without quotes in one and with quotes in the other:

[Math_Add: 1, 2] ➜ 3
[Math_Add: '1', '2'] ➜ 3

When a string is converted into an integer or a decimal, only a number at the beginning of the string will be converted. For example, in the following conversion only the number 800 from the phone number will be output.

[Integer: '800-555-1212'] ➜ 800

In earlier versions of Lasso all the numbers would have been extracted from the string yielding 8005551212 as the value. Existing sites may require modifications if this behavior was being counted on.

Note: Negative literals must be surrounded by parentheses when used on the right-hand side of two-operator symbols. For example, (1 + (-2)) or ($Variable == (-4)).

Mathematical Precision

Mathematical symbols in LDML 8 do not have the same rounding behavior as math tags in LDML 3. For example, the following [Math_Div] tag returns a result with the LDML 8 standard of six significant digits instead of the maximum precision of its two parameters which it would have had in LDML 3.

[Math_Div: 10, 3.000] ➜ 3.333333

In LDML 8 the mathematical symbols perform an integer operation if both parameters of the expression are integers. For example, the following division is performed and an integer result is returned:

[10 / 3] ➜ 3

In LDML 8 the mathematical symbols perform a decimal operation if either of the parameters of the expression are a decimal value. Decimal results are always returned with at least six significant digits. For example, the following expressions return six significant digits of the result since one of the parameters is specified with a decimal point:

[10.0 / 3] ➜ 3.333333
[10 / 3.0] ➜ 3.333333

Existing sites should be modified to use the [Math_Round] tag or the [Decimal->SetFormat] tag to format results from mathematical expressions if less than six significant digits is desired.

The following example shows how to use [Math_Round] to reduce a division expression to three significant digits:

```
[Math_Round: (10.0 / 3.0), 1.000] ➜ 3.333
```

The following example shows how to set a variable so it will always display three significant digits using the [Decimal->SetFormat] tag.

```
[Variable: 'Result' = (10.0 / 3.0)]
[(Variable: 'Result')->(SetFormat: -Precision=3)]
[Variable: 'Result']

➜ 3.333
```

See the *Math Operations* chapter for more information.

Double Quotes

Single quotes are preferred when specifying string literals. Double quotes are still supported, but have been deprecated. Double quotes are not guaranteed to work in the future. No changes to existing sites should be required, but all future development should use single quotes exclusively.

Restrictions

Some restrictions have been removed in LDML 8. Your site may need to be rewritten if it relied on one of these pre-defined restrictions. The following restrictions have been removed in LDML 8:

- Integer math now uses 64-bit values for greater precision. LDML 8 should support integer values up to 18,446,744,073,709,551,616. Decimal math and date calculation are also performed using 64-bit values.
- The [Loop] tag limit of 1000 iterations has been removed. It is now possible for infinite loops to occur in LDML so you may want to place your own upper limit on loop iterations as in the following code:

```
[Loop: 1000000]
  [If: (Loop_Count) > 1000][Loop_Abort][/If]
  … Loop Contents …
[/Loop]
```

Tag Name Changes

In order to promote consistency in LDML 8 many tag names from LDML 3 had to be changed. The following chart details the tag names which have changed. Please consult the appropriate chapters in this book for more information about each individual tag name.

For the most part, these tag name changes will not require modifications to existing Lasso Web Data Engine 3.x sites. The old tag name is still supported in LDML 8. However, support for these old tag names is deprecated. They are not guaranteed to be supported in a future version of Lasso. All new development should take place using the new tag names.

Table 12: Command Tag Name Changes details the command tags which have changed in LDML 8. *Table 9: Substitution, Process, and Container Tag Name Changes* details the substitution, process, and container tags which have changed in LDML 8.

Table 3: Command Tag Name Changes

LDML 3 Tag	LDML 8 Tag Equivalent
-AddError	-ResponseAddError
-AddResponse	-ResponseAdd
-AnyError	-ResponseAnyError
-AnyResponse	-ResponseAny
-ClientPassword	-Password
-ClientUsername	-Username
-DeleteResponse	-ResponseDelete
-DoScript	-FMScript
-DoScript.Post	-FMScriptPost
-DoScript.Pre	-FMScriptPre
-DoScript.PreSort	-FMScriptPreSort
-DuplicateResponse	-ResponseDuplicate
-LogicalOperator	-OperatorLogical
-NoResultsError	-ResponseNoResultsError
-RequiredFieldMissingError	-ResponseRequiredFieldMissingError
-SecurityError	-ResponseSecurityError
-UpdateError	-ResponseUpdateError
-UpdateResponse	-ResponseUpdate

Table 4: Substitution, Process, and Container Tag Name Changes

LDML 3 Tag	LDML 8 Tag Equivalent
[Choice_List]	[Value_List]
[ChoiceListItem]	[Value_ListItem]
[DB_NameItem]	[Database_NameItem]
[DB_Names]	[Database_Names]
[DB_LayoutNameItem]	[Database_TableNameItem]
[DB_LayoutNames]	[Database_TableNames]
[Encode_Breaks]	[Encode_Break]
[File_LineCount]	[File_GetLineCount]
[Lasso_Abort]	[Abort]
[Lasso_Comment]	[Output_None]
[Lasso_Process]	[Process]
[Lasso_SessionID]	[Lasso_UniqueID]
[Link_Detail]	[Link_DetailURL]
[Logical_OperatorValue]	[Operator_LogicalValue]
[LoopAbort]	[Loop_Abort]
[LoopCount]	[Loop_Count]
[RandomNumber]	[Math_Random]
[RepeatingValueItem]	[Repeating_ValueItem]
[Roman]	[Math_Roman]
[SearchFieldItem]	[Search_FieldItem]
[SearchOpItem]	[Search_OpItem]
[SearchValueItem]	[Search_ValueItem]
[Shown_NextGroup]	[Link_NextGroup]
[Shown_NextGroupURL]	[Link_NextGroupURL]
[Shown_PrevGroup]	[Link_PrevGroup]
[Shown_PrevGroupURL]	[Link_PrevGroupURL]
[SortFieldItem]	[Sort_FieldItem]
[SortOrderItem]	[Sort_OrderItem]
[String_ToDecimal]	[Decimal]
[String_ToInteger]	[Integer]
[ValueListItem]	[Value_ListItem]

Unsupported Tags

The following tags are no longer supported in LDML 8. If any of these tags are used in a Web site that was built for Lasso Web Data Engine 3.x they will need to be replaced before that Web site can be served by Lasso Professional 8. *Table 14: Unsupported Tags* is a complete list of tags that are not supported in LDML 8 including notes on how to update a Web site that relies on those tags for compatibility with Lasso Professional 8.

Table 5: Unsupported Tags

LDML 3 Tag	Notes
[4D_RefreshCache]	The 4D data source module is no longer provided.
[Apple_Event], [AE_…]	The Apple Event tags are no longer supported.
-DoScript…..Back	The -DoScript tags with a Back argument are no longer supported. Use the appropriate -FMScript… tag instead.
[Lasso_DatasourceIs4D]	The 4D data source module is no longer provided.
[Lasso_DatasourceIsODBC]	The ODBC data source module is no longer provided.
[Macro_…], -Macro	All macro tags are no longer supported. See the Extending Lasso Guide for information about custom tags.
[Post_Inline]	See the Post Inline section in this chapter for more information about how to convert [Post_Inline] calls to the [Event_Schedule] tag.
[Relation]	The [Relation] tag was equivalent to an [Inline] that performed a search in the related table.
-Scripts	This command tag only worked with the Apple Event based FileMaker Pro data source module. Use any command tag which performs a database action instead (e.g. -FindAll).
-Timeout	This command tag only worked with the Apple Event based FileMaker Pro data source module.
[Win_Exec]	This tag is no longer supported.

CDML Compatibility

Lasso Web Data Engine 3.x supported a number of CDML tags for compatibility with Web sites that were created for FileMaker Pro's Web Companion. These tags are no longer supported in Lasso Professional 8.

Early Lasso Compatibility

Lasso Web Data Engine 3.x supported a number of tags from earlier versions of Lasso for compatibility with sites that were created using the earlier versions of Lasso. These tags are no longer supported in Lasso Professional 8.

FileMaker Pro

Lasso Professional 8 includes Lasso Connector for FileMaker Pro which is the equivalent of the Lasso Web Data Engine 3.x FileMaker Pro Remote data source module. The functionality of the Apple Event based FileMaker Pro data source module is no longer supported since it was Mac specific and reliant upon the use of Apple Events.

If you were using the FileMaker Pro Remote data source module then no changes to your site should be required when you move the site over to Lasso Professional 8.

If you were not previously using the FileMaker Pro Remote data source module, some changes may be necessary. Lasso Connector for FileMaker Pro does not support the following features of the Apple Event based FileMaker Pro data source module from Lasso Web Data Engine 3.x.

- **Field-Level Search Operators** are not supported. The -OperatorBegin and -OperatorEnd tags cannot be used to create complex queries with a FileMaker Pro database.
- **Automatic Image Conversion** is not supported for PICT images stored in FileMaker Pro container field. However, GIFs and JPEGs stored in container field can be retrieved. The parameters of the [Image_URL] tag are ignored and images are served in the format stored in the database.
- Certain **Script Command Tags** are not supported including -DoScript.Back, -DoScript.Post.Back, -DoScript.PreSort.Back, -DoScript.Pre.Back. These tags all instruct FileMaker Pro to send itself to the background after the script is completed. Use the -FMScript commands without the Back argument instead.
- **FileMaker Pro 3** is no longer supported since this version does not provide the Web Companion necessary to make a remote connection to FileMaker Pro.

However, in exchange for the omissions there are some advantages to using Lasso Connector for FileMaker Pro.

- FileMaker Pro can be accessed via TCP/IP on the same machine or on a different machine.

- Multiple FileMaker Pro applications running on different machines can be accessed from a single installation of Lasso Professional 8.
- The -ReturnField tag allows you to limit the fields that are returned from a search or other database action.
- GIFs and JPEGs can be stored in FileMaker Pro container fields and served directly without any conversion.

Upgrading FileMaker Pro Based Sites

If a site was created using the FileMaker Remote data source module then no changes should be necessary when moving the site to Lasso Professional 8. Simply follow the instructions in the *Upgrading* chapter in the Lasso Professional 8 Setup Guide in order to configure Lasso Connector for FileMaker Pro to point to the appropriate FileMaker Pro Web Companion.

If a site was created using the Apple Event based FileMaker Pro data source module or relied on FileMaker Pro 3.x then the following changes will need to be made in order to ensure that the site is compatible with Lasso Professional 8.

To upgrade a FileMaker Pro based site:

1. A site that relies on FileMaker Pro 3 will need to be upgraded to FileMaker Pro 4.x or FileMaker Pro Unlimited 5.x.
2. Configure FileMaker Pro Web Companion according to the instructions inn the *Data Sources* chapter of the Lasso Professional 8 Setup Guide. The Web Companion needs to be activated and all databases that are to be shared need to have their Sharing... settings established.
3. Modify any database searches that relied on the -OperatorBegin and -OperatorEnd command tags so that they no longer reference these tags.
4. Modify any calls to -DoScript... to call one of the new -FMScript... equivalents. Any database action that relies on the -Scripts command tag needs to be rewritten with a database action such as -FindAll.
5. Ensure that the images stored in container fields are either GIFs or JPEGs. These images will be served directly by the Web Companion.

V

Section V
Data Types

This section includes an introduction to the fundamental data types of Lasso 8 including strings, byte streams, integers and decimals, dates, compound data types, files, images, networking, xml, PDF, and JavaBeans.

- *Chapter 24: String Operations* includes information about strings including symbols and tags for string manipulations.
- *Chapter 25: Bytes* includes information about byte streams.
- *Chapter 26: Math Operations* includes information about integers and decimals including symbols and tags for mathematical calculations.
- *Chapter 27: Date and Time Operations* includes information about the date and duration data types for date and time calculations.
- *Chapter 28: Arrays, Maps, and Compound Data Types* includes information about arrays, lists, maps, pairs, priority queues, queues, sets, stacks, and tree maps..
- *Chapter 29: Files* includes information about reading and writing files.
- *Chapter 30: Images and Multimedia* includes information about manipulating and serving images and multimedia files.
- *Chapter 31: Networking* includes information about communicating with remote servers using TCP or UDP protocols..
- *Chapter 32: XML* includes information about parsing and creating XML files including using XPaths and XSLT style sheets.
- *Chapter 33: PDF* includes information about creating PDF files using Lasso's built-in PDF tags.
- *Chapter 34: JavaBeans* includes information about loading, using, and creating JavaBeans..

24

Chapter 24
String Operations

Text in Lasso is stored and manipulated using the string data type or the [String] tags. This chapter details the symbols and tags that can be used to manipulate string values.

- *Overview* provides an introduction to the string data type and how to cast values to and from other data types.
- *String Symbols* details the symbols that can be used to create string expressions.
- *String Manipulation Tags* describe the member and substitution tags that can be used to modify string values.
- *String Conversion Tags* describes the member and substitution tags that can be used to convert the case of string values.
- *String Validation Tags* describes the member and substitution tags that can be used to compare strings.
- *String Information Tags* describes the member and substitution tags that can be used to get information about strings and characters.
- *String Casting Tags* describes the [String->Split] tag which can be used to cast a string to an array value.
- *Regular Expressions* describes the string tags that allow for regular expression substitutions.

Note: The string type is often used in conjunction with the bytes type to convert binary data between different character encodings (UTF-8, ISO-8859-1). See the next chapter for more information about the bytes type.

Overview

Many LDML tags are dedicated to outputting and manipulating text. LDML is used to format text-based HTML pages or XML data for output. LDML is also used to process and manipulate text-based HTML form inputs and URLs. Text processing is a central function of LDML.

As a result of this focus on text processing, the string data type is the primary data type in LDML. When necessary, all values are cast to string before subsequent tag or symbol processing occurs. All values are cast to string before they are output into the HTML page or XML data which will be served to the site visitor.

There are three types of operations that can be performed directly on strings.

- Symbols can be used to perform string calculations within LDML tags or to perform assignment operations within LassoScripts.

  ['The' + ' ' + 'String'] ➜ The String

- Member tags can be used to manipulate string values or to output portions of a string.

  ['The String'->(Substring: 4, 6)] ➜ String

- Substitution tags can be used to test the attributes of strings or to modify string values.

  [String_LowerCase: 'The String'] ➜ the string

Each of these methods is described in detail in the sections that follow. This guide contains a description of every symbol and tag and many examples of their use. The LDML Reference is the primary documentation source for LDML symbols and tags. It contains a full description of each symbol and tag including details about each parameter.

Unicode Characters

Lasso Professional 8 supports the processing of Unicode characters in all string tags. The escape sequence \u... can be used with 4, or 8 hexadecimal characters to embed a Unicode character in a string. For example \u002F reprsents a / character, \u0020 represents a space, and \u0042 represents a capital letter B. The same type of escape sequence can be used to embed any Unicode character \u4E26 represents the Traditional Chinese character 並.

Lasso also supports common escape sequences including \r for a return character, \n for a new-line character, \r\n for a Windows return/new-line, \f for a form-feed character, \t for a tab, and \v for a vertical-tab.

Casting Values to Strings

Values can be cast to the string data type automatically in many situations or they can be cast explicitly using the [String] tag.

Table 1: String Tag

Tag	Description
[String]	Casts a value to type string.

Examples of automatic string casting:

- Integer and decimal values are cast to strings automatically if they are used as a parameter to a string symbol. If either of the parameters to the symbol is a string then the other parameter is cast to a string automatically. The following example shows how the integer 123 is automatically cast to a string because the other parameter of the + symbol is the string String.

  ['String ' + 123] ➜ String 123

  The following example shows how a variable that contains the integer 123 is automatically cast to a string.

  [Variable: 'Number' = 123]
  ['String ' + (Variable: 'Number')] ➜ String 123

- Array, map, and pair values are cast to strings automatically when they are output to a Web page. The value they return is intended for the developer to be able to see the contents of the complex data type and is not intended to be displayed to site visitors.

  [(Array: 'One', 'Two', 'Three')]

  ➜ (Array: (One), (Two), (Three))

  [(Map: 'Key1'='Value1', 'Key2'='Value2')]

  ➜ (Map: (Key1)=(Value1), (Key2)=(Value2))

  [(Pair: 'Name'='Value')]

  ➜ (Pair: (Name)=(Value))

  More information can be found in the *Arrays and Maps* chapter.

- The parameters for string substitution tags are automatically cast to strings. The following example shows how to use the [String_Length] substitution tag on a numeric value from a field.

  [Field: 'Age'] ➜ 21
  [String_Length: (Field: 'Age')] ➜ 2

To explicitly cast a value to the string data type:

- Integer and decimal values can be cast to type string using the [String] tag. The value of the string is the same as the value of the integer or decimal value when it is output using the [Variable] tag.

  The following example shows a math calculation and the integer operation result 579. The next line shows the same calculation with string parameters and the string symbol result 123456.

  ```
  [123 + 456] ➜ 579
  [(String: 123) + (String: 456)] ➜ 123456
  ```

- Boolean values can be cast to type string using the [String] tag. The value will always either be True or False. The following example shows a conditional result cast to type string.

  ```
  [(String: ('dog' == 'cat'))] ➜ false
  ```

- String member tags can be used on any value by first casting that value to a string using the [String] tag. The following example shows how to use the [String->Size] member tag on a numeric value from a field by first casting the field value to type string.

  ```
  [Field: 'Age'] ➜ 21
  [(String: (Field: 'Age'))->Size] ➜ 2
  ```

String Symbols

The easiest way to manipulate values of the string data type is to use the string symbols. *Table 2: String Symbols* details all the symbols that can be used with string values.

Table 2: String Symbols

Symbol	Description
+	Concatenates two strings. This symbol should always be separated from its parameters by a space.
-	Deletes a substring. The first occurrence of the right parameter is deleted from the left parameter. This symbol should always be separated from its parameters by a space.
*	Repeats a string. The right parameter should be a number.
=	Assigns the right parameter to the variable designated by the left parameter.

+=	Concatenates the right parameter to the value of the left parameter and assigns the result to the variable designated by the left parameter.
-=	Deletes the right parameter from the value of the left parameter and assigns the result to the variable designated by the left parameter.
*=	Repeats the value of the left parameter and assigns the result to the variable designated by the left parameter.
>>	Returns True if the left parameter contains the right parameter as a substring.
!>>	Returns True if the left parameter does not contain the right parameter as a substring.
==	Returns True if the parameters are equal.
!=	Returns True if the parameters are not equal.
<	Returns True if the left parameter comes before the right parameter alphabetically.
<=	Returns True if the left parameter comes before the right parameter alphabetically or if the parameters are equal.
>	Returns True if the left parameter comes after the right parameter alphabetically.
>=	Returns True if the left parameter comes after the right parameter alphabetically or if the parameters are equal.
===	Returns True if the parameters are equal and both are of type string. No casting is performed.

Each of the string symbols takes two parameters. One of the parameters must be a string value in order for the symbol to perform the designated string operation. Many of the symbols can also be used to perform integer or decimal operations. If both parameters are integer or decimal values then the mathematical operation defined by the symbol will be performed rather than the string operation.

As long as one of the parameters of the symbol is a string the other parameter will be auto-cast to a string value before the operation defined by the symbol is performed. The two exceptions to this are the * and *= symbols which must have an integer as the right parameter.

Note: Full documentation and examples for each of the string symbols can be found in the LDML Reference.

Examples of using the string symbols:

- Two strings can be concatenated using the + symbol. Note that the symbol is separated from its parameters using spaces.

```
['Alpha ' + 'Beta'] ➜ Alpha Beta
```

- A string and an integer can be concatenated using the + symbol. The integer will be automatically cast to a string. Note that the symbol is separated from its parameters using spaces.

```
['Alpha ' + 1000] ➜ Alpha 1000
```

- A substring can be deleted from a string using the - symbol. The following example shows how to remove the substrings and from a string of HTML text. Note that the symbol is separated from its parameters using spaces.

```
['<b>Bold Text</b>' - '<b>' - '</b>'] ➜ Bold Text
```

- A string can be repeated using the * symbol. The following example shows how to repeat the word Lasso three times.

```
['Lasso ' * 3] ➜ Lasso Lasso Lasso
```

- Strings will be automatically concatenated even if the + symbol is omitted. This makes concatenating long sets of strings easier.

```
['Alpha ' 'Beta'] ➜ Alpha Beta
```

Examples of using the string assignment symbols:

- A string variable can be assigned a new value using the = symbol. The following example shows how to define a string symbol and then set it to a new value. The new value is output.

```
<?LassoScript
  Variable: 'StringVariable' = 'The String Value';
  $StringVariable = 'New String Value';
  $StringVariable;
?>
```

➜ New String Variable

- A string variable can be used as a collector by concatenating new values to it in place using the += symbol. The following example shows how to define a string symbol and then concatenate several values to it. The final value is output.

```
<?LassoScript
  Variable: 'StringVariable' = 'The ';
  $StringVariable += 'String ';
  $StringVariable += 'Variable';
  $StringVariable;
?>
```

➜ The String Variable

Examples of using the string comparison symbols:

- Two strings can be compared for equality using the == symbol and != symbol. The result is a boolean True or False.

  ['Alpha ' == 'Beta'] ➜ False
  ['Alpha ' != 'Beta'] ➜ True

- Strings can be ordered alphabetically using the <, <=, >, and <= symbols. The result is a boolean True or False.

  ['Alpha ' > 'Beta'] ➜ False
  ['Alpha ' < 'Beta'] ➜ True

- A string can be checked to see if it contains a particular substring using the >> symbol. The result is a boolean True or False.

  [''Bold Text' >> ''] ➜ True

String Manipulation Tags

The string data type includes many tags that can be used to manipulate string values. The available member tags are listed in *Table 3: String Manipulation Member Tags* and the available substitution tags are listed in *Table 4: String Manipulation Tags*.

In addition to the tags in this section, the tags in the following section on *String Conversion Tags* can be used to modify the case of a string and the tags in the section on *Regular Expression Tags* can be used for more powerful string manipulations using regular expressions.

The member tags in this section all modify the base string in place and do not return a value. For example, the [String->Append] tag works like the += symbol. In order to see the values that were appended to the string, the variable containing the string must be output.

```
[Variable: 'myString' = 'Test']
[$myString->(Append: ' string.')]
[$myString] ➜ Test string.
```

In contrast, the substitution tags return the modified string directly.

```
[String_Concatenate: 'Test', ' string.'] ➜ Test string.
```

The member tags should be used when multiple modifications need to be made to a string that is stored in a variable. The substitution tags, or string symbols, can be used when the value is required immediately for output.

Table 3: String Manipulation Member Tags

Tag	Description
[String->Append]	Casts the parameters to strings and appends them to the string. Modifies the string and returns no value. Requires one string parameter.
[String->Merge]	Inserts a merge string into the string. Requires two parameters, the location at which to insert the merge string and the string to insert. Optional third and fourth parameters specify an offset into the merge string and number of characters of the merge string to insert.
[String->PadLeading]	Pads the front of a string to a specified length with a pad character. Modifies the string and returns no value. Requires a length to pad the string. Optional second parameter is the padding character (defaults to space).
[String->PadTrailing]	Pads the end of a string to a specified length with a pad character. Modifies the string and returns no value. Requires a length to pad the string. Optional second parameter is the padding character (defaults to space).
[String->Remove]	Removes a substring from the string. The first parameter is the offset at which to start removing characters. The second parameter is the number of characters to remove. Defaults to removing to the end of the string.
[String->RemoveLeading]	Removes all instances of the parameter from the beginning of the string. Modifies the string and returns no value. Requires a single string parameter.
[String->RemoveTrailing]	Removes all instances of the parameter from the end of the string. Modifies the string and returns no value. Requires a single string parameter.
[String->Replace]	Replaces every occurence of a substring. Requires two parameters, the substring to find and the replacement string. Modifies the string and returns no value. Optional third parameter specifies the maximum number of replacements to perform.
[String->Reverse]	Reverses the string. Optional parameters specify a character offset and length for a substring to be reversed. Defaults to reversing the entire string. Modifies the string and returns no value.
[String->Trim]	Removes all white space from the start and end of the string. Modifies the string in place and returns no value.

Note: Full documentation and examples for each of the string member tags can be found in the LDML Reference.

To replace a substring:

Use the [String->Replace] tag. The following example replaces every instance of and within the string to or.

```
[Variable: 'myString' = 'Red and Yellow and Blue']
[$myString->(Replace: 'and','or')]
[$myString]
```

➜ Red or Yellow or Blue

To remove white space from the start and end of a string:

Use the [String->Trim] tag. The following example removes all the white space from the start and end of the string leaving just the relevant content.

```
[Variable: 'myString' = '       Green and Purple       ']
[$myString->(Trim)]
[$myString]
```

➜ Green and Purple

Table 4: String Manipulation Tags

Tag	Description
[String_Concatenate]	Concatenates all of its parameters into a single string.
[String_Insert]	Takes three parameters: a string, a -Text keyword/value parameter which defines the text to be inserted, and a -Position parameter which defines the offset into the string at which to insert the text. Returns a new string with the specified text inserted at the specified location.
[String_Remove]	Takes three parameters: a string, a -StartPosition keyword/value parameter, and a -EndPosition keyword/value parameter. Returns the string with the substring from -StartPosition to -EndPosition removed.
[String_RemoveLeading]	Takes two parameters: a string and a -Pattern keyword/value parameter. Returns the string with any occurrences of the pattern removed from the start.
[String_RemoveTrailing]	Takes two parameters: a string and a -Pattern keyword/value parameter. Returns the string with any occurrences of the pattern removed from the end.
[String_Replace]	Takes three parameters: a string, a -Find keyword/value parameter, and a -Replace keyword/value parameter. Returns the string with the first instance of the -Find parameter replaced by the -Replace parameter.

Note: Full documentation and examples for each of the string tags can be found in the LDML Reference.

Examples of using string manipulation tags:

- The [String_Extract] tag can be used to return a portion of a string. In the following example five characters of the string A Short String are returned

```
[String_Extract: 'A Short String', -StartPosition=3, -EndPosition=8] ➜ Short
```

- The [String_Remove] tag is similar, but rather than returning a portion of a string, it removes a portion of the string and returns the remainder. In the following example five characters of the string A Short String are removed and the remainder is returned.

```
[String_Remove: 'A Short String', -StartPosition=3, -EndPosition=8] ➜ A String
```

- The [String_RemoveLeading] and [String_RemoveTrailing] tags can be used to remove a repeating character from the start or end of a string. In the following example asterisks are removed from a string *A Short String*.

```
[String_RemoveLeading: -Pattern='*',
  (String_RemoveTrailing: -Pattern='*', '*A Short String*')]

➜ A Short String
```

- The [String_Replace] tag can be used to replace a portion of a string with new characters. In the following example the word Short is replaced by the word Long.

```
[String_Replace: 'A Short String', -Find='Short', -Replace='Long'] ➜ A Long String
```

Note: For more powerful string manipulation see the ***Regular Expressions*** section below.

String Conversion Tags

The string data type includes many tags that can be used to change the case of string values. The available member tags are listed in *Table 5: String Conversion Member Tags* and the available substitution tags are listed in *Table 6: String Conversion Tags*.

The member tags in this section all modify the base string in place and do not return a value. In order to see the converted string, the variable containing the string must be output.

```
[Variable: 'myString' = 'Test']
[$myString->(UpperCase)]
[$myString] ➜ TEST
```

In contrast, the substitution tags return the modified string directly.

```
[String_UpperCase: 'Test'] ➜ TEST
```

The member tags should be used when multiple modifications need to be made to a string that is stored in a variable. The substitution tags can be used when the value is required immediately for output.

Table 5: String Conversion Member Tags

Tag	Description
[String->Foldcase]	Converts all characters in the string for a case-insensitive comparison. Modifies the string and returns no value.
[String->Lowercase]	Converts all characters in the string to lowercase. Modifies the string in place and returns no value. Accepts an optional locale/country code for Unicode conversion.
[String->Titlecase]	Converts the string to titlecase with the first character of each word capatilized. Modifies the string in place and returns no value. Accepts an optional locale/country code for Unicode conversion.
[String->toLower]	Converts a character of the string to lowercase. Requires the position of the character to be modifed. Modifies the string in place and returns no value.
[String->toUpper]	Converts a character of the string to uppercase. Requires the position of the character to be modifed. Modifies the string in place and returns no value.
[String->toTitle]	Converts a character of the string to titlecase. Requires the position of the character to be modifed. Modifies the string in place and returns no value.
[String->Unescape]	Converts a string from the hexadecimal URL encoding.
[String->Uppercase]	Converts all characters in the string to uppercase. Modifies the string in place and returns no value. Accepts an optional locale/country code for Unicode conversion.

Note: Full documentation and examples for each of the string member tags can be found in the LDML Reference.

Table 6: String Conversion Tags

Tag	Description
[String_LowerCase]	Returns the concatenation of all of its parameters in lowercase.
[String_UpperCase]	Returns the concatenation of all of its parameters in lowercase.

Examples of using string conversion tags:

The [String_UpperCase] and [String_Lowercase] tags can be used to alter the case of a string. The following example shows the result after using these tags on the string A Short String.

```
[String_UpperCase: 'A Short String'] ➜ A SHORT STRING
[String_LowerCase: 'A Short String'] ➜ a short string
```

String Validation Tags

The string data type includes many tags that can be used to compare and validate string values. The available member tags are listed in *Table 7: String Validation Member Tags* and the available substitution tags are listed in *Table 8: String Validation Tags*.

All of these tags return a boolean value True or False depending on whether the test succeeds or not.

Table 7: String Validation Member Tags

Tag	Description
[String->BeginsWith]	Returns True if the string begins with the parameter. Comparison is case insensitive. Requires a single string parameter.
[String->Compare]	This tag has three forms. In the first, it returns 0 if the parameter is equal to the string, 1 if the characters in the string are bitwise greater than the parameter, and -1 if the characters in the string are bitwise less than the parameter. Comparison is case insensitive by default. An optional -Case parameter makes the comparison case sensitive. Requires a single string parameter. The second form requires three parameters. The first two parameters are an offset and length into the third string parameter. The comparison is only performed with this parameter substring. The third form requires two additional parameters. The fourth and fifth parametres are an offset and length into the base string. The comparison is only performed between the base and parameter substrings. [String->CompareCodePointOrder] accepts the same parameters as [String->Compare], but provides accurate comparisons for Unicode characters with code points U+10000 and above.

[String->Contains]	Returns True if the string contains the parameter as a substring. Comparison is case insensitive. Requires a single string parameter.
[String->EndsWith]	Returns True if the string ends with the parameter. Comparison is case insensitive. Requires a single string parameter.
[String->Equals]	Returns True if the parameter of the tag is equal to the string. Comparison is case insensitive. Equivalent to the == symbol.

Note: Full documentation and examples for each of the string member tags can be found in the LDML Reference.

To compare two strings:

Use the string comparison member tags. The following code checks whether a string stored in a variable is equal to or contains another string.

```
[Variable: 'testString' = 'A short string']

[$testString->(BeginsWith: 'a')] ➜ True
[$testString->(BeginsWith: 'A short')] ➜ True
[$testString->(BeginsWith: 'string')] ➜ False
[$testString->(EndsWith: 'string')] ➜ True
[$testString->(Contains: 'short')] ➜ True
[$testString->(Equals: 'a short string')] ➜ True
[$testString->(Compare: 'a short string', -Case)] ➜ False
[$testString->(Compare: 3, 5, 'short')] ➜ True
[$testString->(Compare: 3, 5, 'x short other', 3, 5)] ➜ True
```

Table 8: String Validation Tags

Tag	Description
[String_EndsWith]	Returns boolean True if the string ends with the string specified in the -Find parameter. Takes two parameters: a string value and a -Find keyword/value parameter.

Note: Full documentation and examples for each of the string tags can be found in the LDML Reference.

String Information Tags

The string data type includes many tags that can be used to get information about string and character values. The available member tags are listed in *Table 9: String Information Member Tags* and the available substitu-

tion tags are listed in *Table 10: String Information Tags*. In addition, tags which are specific to getting information about characters in a string are listed in *Table 11: Character Information Member Tags*.

These tags return different data types depending on what information is being retrieved about the string. Those tags that return a character position or require a character position as a parameter all number characters starting from 1 for the first character in the string.

Table 9: String Information Member Tags

Tag	Description
[String->Find]	Returns the position at which the first parameter is found within the string or 0 if the first parameter is not found within the string. Requires a single string parameter.
[String->Get]	Returns a specific character from the string. Requires a single integer parameter.
[String->Size]	Returns the number of characters in the string.[String->Length] is a synonym.
[String->SubString]	Returns a substring. The start of the substring is defined by the first parameter and the length of the substring is defined by the second parameter. Requires two integer parameters.

Note: Full documentation and examples for each of the string member tags can be found in the LDML Reference.

To return the length of a string:

- The length of a string can be returned using the [String->Size] tag.

```
['Alpha'->Size] ➜ 5
```

- The length of a [Variable] value, [Field] value or any value returned by an LDML tag can be returned using the [String->Size] tag.

```
[$StringVariable + ' ' + $StringVariable->Size] ➜ Alpha 5
[(Field: 'First_Name') + ' ' + (Field: 'First_Name')->Size] ➜ Joe 3
```

To return a portion of a string:

- A specific character from a string can be returned using the [String->Get] tag. In the following example, the third character of Alpha is returned.

```
['Alpha'->(Get: 3)] ➜ p
```

- A specific range of characters from a string can be returned using the [String->Substring] tag. In the following example, six characters are returned from the string, starting at the third character.

  ['A String Value'->(Substring: 3, 6)] ➜ String

- The start of a string can be returned using the [String->Substring] tag with the first parameter set to 1. The second parameter will define how many characters are returned from the start of the string. In the following example, the first eight characters of the string are returned.

  ['A String Value'->(Substring: 1, 8)] ➜ A String

- The end of a string can be returned using the [String->Substring] tag with the second parameter omitted. The following example returns the portion of the string after the tenth character.

  ['A String Value'->(Substring: 10)] ➜ Value

Table 10: String Information Tags

Tag	Description
[String_Extract]	Takes three parameters: a string, a -StartPosition keyword/value parameter, and a -EndPosition keyword/value parameter. Returns a substring from -StartPosition to -EndPosition.
[String_FindPosition]	Takes two parameters: a string value and a -Find keyword/value parameter. Returns the location of the -Find parameter in the string parameter.
[String_IsAlpha]	Returns boolean True if the string contains only alphabetic characters (a-z or A-Z).
[String_IsAlphaNumeric]	Returns boolean True if the string contains only alphabetic characters or numerals (a-z, A-Z, or 0-9).
[String_IsDigit]	Returns boolean True if the string contains only numerals (0-9).
[String_IsHexDigit]	Returns boolean True if the string contains only hexadecimal numerals (0-9 and a-f).
[String_IsLower]	Returns boolean True if the string contains only lowercase alphabetic characters (a-z).
[String_IsNumeric]	Returns boolean True if the string contains only numerals, hyphens, or periods.
[String_IsPunctuation]	Returns boolean True if the string contains only punctuation characters.
[String_IsSpace]	Returns boolean True if the string contains only white space.
[String_IsUpper]	Returns boolean True if the string contains only uppercase alphabetic characters (A-Z).

[String_Length]	Returns the number of characters in the string.

Note: Full documentation and examples for each of the string tags can be found in the LDML Reference.

Example of using [String_Length] tag:

The [String_Length] tag can be used to return the number of characters in a string. This tag returns the same results as the [String->Size] tag except the method of calling the tag is somewhat different.

The following example shows how to return the length of the string A Short String using both the [String_Length] tag and the [String->Size] tag. The result in both cases is 14.

```
[String_Length: 'A Short String'] ➜ 14
['A Short String'->Size] ➜ 14
```

Examples of using string validation tags:

The characters in a string can be checked to see if they meet certain criteria using the [String_Is...] tags. Each character in the string is checked to see if it meets the criteria of the tag. If any single character does not meet the criteria then False is returned.

- In the following example a string word is checked to see which validation strings it passes. The string is in lowercase and consists entirely of alphabetic characters. It is not in uppercase and does not consist entirely of numeric characters.

  ```
  [String_IsAlpha: 'word'] ➜ True
  [String_IsAlphaNumeric: 'word'] ➜ True
  [String_IsLower: 'word'] ➜ True
  [String_IsNumeric: 'word'] ➜ False
  [String_IsUpper: 'word'] ➜ False
  ```

- In the following example a string 2468 is checked to see which validation strings it passes. The string consists entirely of numeric characters. It does not consist entirely of alphabetic characters.

  ```
  [String_IsAlpha: '2468'] ➜ False
  [String_IsAlphaNumeric: '2468'] ➜ True
  [String_IsNumeric: '2468'] ➜ True
  ```

- Some of the validation tags are intended to be used on individual characters. The following example shows how each of these tags can be used.

  ```
  [String_IsDigit: '9'] ➜ True
  [String_IsHexDigit: 'a'] ➜ True
  [String_IsPunctuation: '.'] ➜ True
  [String_IsSpace: ' '] ➜ True
  ```

Table 11: Character Information Member Tags

Tag	Description
[String->CharDigitValue]	Returns the integer value of a character or -1 if the character is alphabetic. Requires a single parameter that specifies the location of the character to be inspected.
[String->CharName]	Returns the Unicode name of a character. Requires a single parameter that specifies the location of the character to be inspected.
[String->CharType]	Returns the Unicode type of a character. Requires a single parameter that specifies the location of the character to be inspected.
[String->Digit]	Returns the integer value of a character according to a particular radix. Requires two parameters. The first specifies the location of the character to be inspected. The second specifies the radix of the result (e.g. 16 for hexadecimal).
[String->GetNumericValue]	Returns the decimal value of a character or a negative number of the character is alphabetic. Requires a single parameter that specifies the location of the character to be inspected.
[String->IsAlnum]	Returns True if the character is alphanumeric. Requires a single parameter that specifies the location of the character to be inspected.
[String->IsAlpha]	Returns True if the character is alphabetic. Requires a single parameter that specifies the location of the character to be inspected.
[String->IsBase]	Returns True if the character is part of the base characters of Unicode. Requires a single parameter that specifies the location of the character to be inspected.
[String->IsCntrl]	Returns True if the character is a control character. Requires a single parameter that specifies the location of the character to be inspected.
[String->IsDigit]	Returns True if the character is numeric. Requires a single parameter that specifies the location of the character to be inspected.
[String->IsLower]	Returns True if the character is lowercase. Requires a single parameter that specifies the location of the character to be inspected.
[String->IsPrint]	Returns True if the character is printable (i.e. not a control character). Requires a single parameter that specifies the location of the character to be inspected.
[String->IsSpace]	Returns True if the character is a space. Requires a single parameter that specifies the location of the character to be inspected.

[String->IsTitle]	Returns True if the character is titlecase. Requires a single parameter that specifies the location of the character to be inspected.
[String->IsUpper]	Returns True if the character is uppercase. Requires a single parameter that specifies the location of the character to be inspected.
[String->IsWhitespace]	Returns True if the character is white space. Requires a single parameter that specifies the location of the character to be inspected.
[String->IsUAlphabetic]	Returns True if the character has the Unicode alphabetic attribute. Requires a single parameter that specifies the location of the character to be inspected.
[String->IsULowercase]	Returns True if the character has the Unicode lowercase attribute. Requires a single parameter that specifies the location of the character to be inspected.
[String->IsUUppercase]	Returns True if the character has the Unicode uppercase attribute. Requires a single parameter that specifies the location of the character to be inspected.
[String->IsUWhiteSpace]	Returns True if the character has the Unicode white space attribute. Requires a single parameter that specifies the location of the character to be inspected.

Note: Full documentation and examples for each of the string member tags can be found in the LDML Reference.

To inspect the Unicode properties of a string:

Use the character information member tags. The following example shows the information that is provided for a standard ASCII character b. The character name and type are provided according to the Unicode standard. The [String->Integer] tag returns the decimal ASCII value for the character. The [String->Digit] tag with a radix of 16 returns the hexadecimal value for the character.

```
['b'->(CharName: 1)] ➜ LATIN SMALL LETTER B
['b'->(CharType: 1)] ➜ LOWERCASE_LETTER
['b'->(IsLower: 1)] ➜ True
['b'->(IsUpper: 1)] ➜ False
['b'->(IsWhiteSpace: 1)] ➜ False
['b'->(Integer: 1)] ➜ 98
['b'->(Digit: 1, 16)] ➜ 11
```

The information tags can be used on any Unicode characters. The following example shows the tags being used on a Traditional Chinese character 並 that roughly translates to "and". The character is neither uppercase nor lowercase and is identified by the Unicode reference 4E26.

['並'->(CharName: 1)] ➜ CJK UNIFIED IDEOGRAPH-4E26
['並'->(CharType: 1)] ➜ OTHER_LETTER
['並'->(IsLower: 1)] ➜ False
['並'->(IsUpper: 1)] ➜ False
['並'->(IsWhiteSpace: 1)] ➜ False

Note: The character 並 can be represented in a string by \u4E26 or in HTML as the entity 並.

Table 12: Unicode Tags

Tag	Description
[String_GetUnicodeVersion]	Returns the version of the Unicode standard which Lasso supports.
[String_CharFromName]	Returns the character corresponding to the specified Unicode character name.

String Casting Tags

The string data type includes many tags which can be used to cast a value to or from the string data type. These tags are documented in the *Casting Values to Strings* section earlier in this chapter and in corresponding sections in the chapters for each data type.

In addition, the [String->Split] tag can be used to cast a string into an array. This tag is described in *Table 13: String Casting Member Tags*.

Table 13: String Casting Member Tags

Tag	Description
[String->Split]	Splits the string into an array of strings based on the delimiter specified in the first parameter. This tag does not modify the string, but returns the new array. Requires a single string parameter.

To convert a string into an array:

A string can be converted into an array using the [String->Split] tag. A single parameter defines what character should be used to split the string into the multiple elements of the array. The following example splits a string on the space character, returning an array of words from the string.

```
['A String Value'->(Split: ' ')]
```

➜ (Array: (A), (String), (Value))

Regular Expressions

The [String_FindRegExp] and [String_ReplaceRegExp] tags can be used to perform regular expressions find and replace routines on text strings. A regular expression is a powerful pattern-matching language that allows complex replacements to be specified easily.

Note: Full documentation of regular expression methodology is outside the scope of this manual. The implementation of regular expressions in LDML closely matches that in the Perl language. Consult a standard reference on regular expressions for more information about how to use this flexible technology.

Table 14: Regular Expression Tags

Tag	Description
[String_FindRegExp]	Takes two parameters: a string value and a -Find keyword/value parameter. Returns an array with each instance of the -Find regular expression in the string parameter. Optional -IgnoreCase parameter uses case insensitive patterns.
[String_ReplaceRegExp]	Takes three parameters: a string value, a -Find keyword/value parameter, and a -Replace keyword/value parameter. Returns an array with each instance of the -Find regular expression replaced by the value of the -Replace regular expression the string parameter. Optional -IgnoreCase parameter uses case insensitive parameters. Optional -ReplaceOnlyOne parameter replaces only the first pattern match.

A regular expression is assembled by creating a match string. The simplest match string is just a word containing characters or numbers. The match string bird matches the word "bird". Match strings are case sensitive unless the -IgnoreCase parameter is specified. Match strings can also contain symbols such as \w which matches any alphanumeric character. The match string b\wrd would match the word "bird" or the word "bard". Square brackets can be used to generate custom sets of characters or ranges of characters. The match string [bc]ard will match either the word "bard" or the word "card". The match string [bB]ard will match either the word "bard" or the word "Bard".

All of the symbols which can be used in match strings are detailed in *Table 15: Regular Expression Matching Symbols*.

Table 15: Regular Expression Matching Symbols

Symbol	Description
a-z A-Z 0-9	Alphanumeric characters (and any other characters not defined as symbols) match the specified character. Case sensitive.
.	Period matches any single character.
^	Circumflex matches the beginning of a line.
$	Dollar sign matches the end of a line.
\\…	Escapes the next character. This allows any symbol to be specified as a matching character.
[…]	Character class. Matches any character contained within the square brackets.
[^…]	Character exception class. Matches any character which is not contained within the square brackets.
[a-z]	Character range. Matches any character between the two character specified. Can be used with characters or numbers.
\t	Matches a tab character.
\r	Matches a return character.
\n	Matches a new-line character.
\"	Matches a double quote.
\'	Matches a single quote.
\\w	Matches an alphanumeric 'word' character (underscore included).
\\W	Matches a non-alphanumeric character.
\\s	Matches a blank, whitespace character (space, tab, carriage return, etc.).
\\S	Matches a non-blank, non-whitespace character.
\\d	Matches a digit character (0-9).
\\D	Matches a non-digit character.

Note: Other than the built-in escaped characters \n, \r, \t, \", and \' all other escaped characters in regular expressions should be preceded by two back-slashes.

Matching symbols can be used as components of more complex expressions using combination symbols. The simplest combination symbol is + which matches the preceding matching symbol one or more times. The expression [abcd]+ matches any word containing only the letters a, b, or c including "cab", "cad", "dab", "bad", "add", etc.

All of the symbols which can be used in match strings are detailed in *Table 16: Regular Expression Combination Symbols*.

Table 16: Regular Expression Combination Symbols

Symbol	Description
|	Alternation. Matches either the character before or the character after the symbol.
( )	Grouping for output. Defines a named group for output. Nine groups can be defined.
(?:)	Grouping without output. Can be used to create a logical grouping that should not be assigned to an output.
*	Asterisk. Matches 0 or more repetitions of the preceding symbol.
*?	Non-greedy variant works the same as asterisk, but matches the shortest string possible.
+	Plus Sign. Matches 1 or more repetitions of the preceding symbol.
+?	Non-greedy variant works the same as the plus sign, but matches the shortest string possible.
?	Question Mark. Makes the preceding symbol optional.
{n}	Matches n repetitions of the preceding symbol.
{n,}	Matches at least n repetitions of the preceding symbol.
{n,m}	Matches at least n, but no more than m repetitions of the preceding symbol.
{…}?	Non-greedy variant works the same as the bracketed expressions, but matches the shortest string possible.

The parentheses are a special combination symbol that defines a portion of the match string as a named sub-expression that can be referenced in the replacement string. For example a matching string of blue(world) would match the word blueworld. A replacement string of green\\1 would then result in greenworld as output. The word world is named as sub-expression 1 by virtue of the parentheses.

Table 17: Regular Expression Replacement Symbols

Symbol	Description
a-z A-Z 0-9	Alphanumeric characters (and any other characters not defined as symbols) place the specified character in the output.
\\1 … \\9	Names a group in the replace string. Up to nine groups can be specified using the numerals 1 through 9.

Note: Other than the built-in escaped characters \n, \r, \t, \", and \' all other escaped characters in regular expressions should be preceded by two back-slashes.

Table 18: Regular Expression Advanced Symbols

Symbol	Description
(#)	Regular expression comment. The contents are not interpreted as part of the regular expression.
(?i)	Sets the remainder of the regular expression to be case insensitive. Similar to specifying -IgnoreCase.
(?-i)	Sets the remainder of the regular expression to be case sensitive (the default).
(?i:)	The contents of this group will be matched case insensitive and the group will not be added to the output.
(?-i:)	The contents of this group will be matched case sensitive and the group will not be added to the output.
(?=)	Positive look ahead assertion. The contents are matched following the current position, but not added to the output pattern.
(?!)	Negative look ahead assertion. The same as above, but the content must not match following the current position.
(?<=)	Positive look behind assertion. The contents are matched preceding the current position, but not added to the output pattern.
(?<!)	Negative look behind assertion. The same as above, but the contents must not match preceding the current position.
\\b	Matches the boundary between a word and a space.
\\B	Matches a boundary not between a word and a space.

Examples of using [String_ReplaceRegExp]:

The [String_ReplaceRegExp] tag works much like [String_Replace] except that both the -Find parameter and the -Replace can be regular expressions.

- In the following example, every occurrence of the world Blue in the string is replaced by the HTML code Blue so that the word Blue appears in blue on the Web page. The -Find parameter is specified so

either a lowercase or uppercase b will be matched. The -Replace parameter references \\1 to insert the actual value matched into the output.

```
[String_ReplaceRegExp: 'Blue Lake sure is blue today.',
  -Find='([Bb]lue)',
  -Replace='<font color="blue">\\1</font>', -EncodeNone]

➜ <font color="blue">Blue</font> lake sure is <font color="blue">blue</font> today.
```

- In the following example, every email address is replaced by an HTML anchor tag that links to the same email address. The \\w symbol is used to match any alphanumeric characters or underscores. The at sign @ matches itself. The period must be escaped \\. in order to match an actual period and not just any character. This pattern matches any email address of the type name@example.com.

```
[String_ReplaceRegExp: 'Send email to lassodocumentation@omnipilot.com.',
  -Find='(\\w+@\\w+\\.\\w+)',
  -Replace='<a href="mailto:\\1">\\1</a>', -EncodeNone]

➜ Send email to
  <a href="lassodocumentation@omnipilot.com">
    lassodocumentation@omnipilot.com</a>
```

Examples of using [String_FindRegExp]:

The [String_FindRegExp] tag returns an array of items which match the specified regular expression within the string. The array contains the full matched string in the first element, followed by each of the matched subexpressions in subsequent elements.

- In the following example, every email address in a string is returned in an array.

```
[String_FindRegExp: 'Send email to lassodocumentation@omnipilot.com.',
  -Find='\\w+@\\w+\\.\\w+']

➜ (Array: (documentation@omnipilot.com))
```

- In the following example, every email address in a string is returned in an array and sub-expressions are used to divide the username and domain name portions of the email address. The result is an array with the entire match string, then each of the sub-expressions.

```
[String_FindRegExp: 'Send email to lassodocumentation@omnipilot.com.',
  -Find='(\\w+)@(\\w+\\.\\w+)']

➜ (Array: (lassodocumentation@omnipilot.com), (lassodocumentation),
    (omnipilot.com))
```

- In the following example, every word in the source is returned in an array. The first character of each word is separated as a sub-expression. The returned array contains 16 elements, one for each word in the source

string and one for the first character sub-expression of each word in the source string.

```
[String_FindRegExp: 'The quick brown fox jumped over a lazy dog.',
  -Find='(\\w)\\w*']

➜ (Array: (The), (T), (quick), (q), (brown), (b), (fox), (f), (jumped), (j),
  (over), (o), (a), (a), (lazy), (l), (dog), (d))
```

The resulting array can be divided into two arrays using the following code. This code loops through the array (stored in Result_Array) and places the odd elements in the array Word_Array and the even elements in the array Char_Array using the [Repetition] tag.

```
[Variable: 'Word_Array' = (Array), 'Char_Array'=(Array)]
[Variable: 'Result_Array' = (String_FindRegExp:
  'The quick brown fox jumped over a lazy dog.', -Find='(\\w)\\w*')]
[Loop: $Result_Array->Size]
  [If: (Repetition) == 2]
    [$Char_Array->(Insert: $Result_Array->(Get: (Loop_Count)))]
  [Else]
    [$Word_Array->(Insert: $Result_Array->(Get: (Loop_Count)))]
  [/If]
[Loop]
<br>[Output:$Word_Array]
<br>[$Char_Array]

➜ <br>(Array: (The), (quick), (brown), (fox), (jumped), (over), (a), (lazy), (dog))
  <br>(Array: (T), (q), (b), (f), (j), (o), (a), (l), (d))
```

- In the following example, every phone number in a string is returned in an array. The \\d symbol is used to match individual digits and the {3} symbol is used to specify that three repetitions must be present. The parentheses are escaped \\(and \\) so they aren't treated as grouping characters.

```
[String_FindRegExp: 'Phone (800) 555-1212 for information.'
  -Find='\\(\\d{3}\\) \\d{3}-\\d{4}']

➜ (Array: ((800) 555-1212))
```

- In the following example, only words contained within HTML bold tags ... are returned. Positive look ahead and look bind assertions are used to find the contents of the tags without the tags themselves. Note that the pattern inside the assertions uses a non-greedy modifier.

```
[String_FindRegExp: 'This is some <b>sample text</b>!'
  -Find='(?<=<b>).+?(?=</b>)']

➜ (Array: (sample text))
```

25

Chapter 25
Bytes

Binary data in Lasso is stored and manipulated using the bytes data type or the [Byte] tags. This chapter details the symbols and tags that can be used to manipulate binary data.

- *Bytes Type* describes the data type which Lasso uses for binary data.

Note: The bytes type is often used in conjunction with the string type to convert binary data between different character encodings (UTF-8, ISO-8859-1). See the previous chapter for more information about the string type.

Bytes Type

All string data in Lasso is processed as double-byte Unicode characters. The [Bytes] type is used to represent strings of single-byte binary data. The [Bytes] type is often referred to as a byte-stream or binary data.

Lasso tags return data in the [Bytes] type in the following situations.

- The [Field] tag returns a byte stream from MySQL BLOB fields.
- When the -Binary encoding type is used on any tag.
- The [Bytes] tag can be used to allocate a new byte stream.
- Other tags that return binary data. See the LDML Reference for a complete list.

Table 1: Byte Stream Tag

Tag	Description
[Bytes]	Allocates a byte stream. Can be used to cast a string data type as a bytes type, or to instantiate a new bytes instance. Accepts two optional parameters. The first is the initial size in bytes for the stream. The second is the increment to use to grow the stream when data is stored that goes beyond the current allocation.

Byte streams are similar to strings and support many of the same member tags. In addition, byte streams support a number of member tags that make it easier to deal with binary data. These tags are listed in the *Byte Stream Member Tags* table.

Table 2: Byte Stream Member Tags

Tag	Description
[Bytes->Size]	Returns the number of bytes contained in the bytes stream object.
[Bytes->Get]	Returns a single byte from the stream. Requires a parameter which specifies which byte to fetch.
[Bytes->SetSize]	Sets the byte stream to the specified number of bytes.
[Bytes->GetRange]	Gets a range of bytes from the byte stream. Requires a single parameter which is the byte position to start from. An optional second parameter specifies how many bytes to return.
[Bytes->SetRange]	Sets a range of characters within a byte stream. Requires two parameters: An integer offset into the base stream, and the binary data to be inserted. An optional third and fourth parameter specify the offset and length of the binary data to be inserted.
[Bytes->Find]	Returns the position of the beginning of the parameter sequence within the bytes instance, or 0 if the sequence is not contained within the instance. Four optional integer parameters (offset, length, parameter offset, parameter length) indicate position and length limits that can be applied to the instance and the parameter sequence.
[Bytes->Replace]	Replaces all instances of a value within a bytes stream with a new value. Requires two parameters. The first parameter is the value to find, and the second parameter is the value to replace the first parameter with.
[Bytes->Contains]	Returns true if the instance contains the parameter sequence.

[Bytes->BeginsWith]	Returns true if the instance begins with the parameter sequence.
[Bytes->EndsWith]	Returns true if the instance ends with the parameter sequence.
[Bytes->Split]	Splits the instance into an array of bytes instances using the parameter sequence as the delimiter. If the delimiter is not provided, the instance is split, byte for byte, into an array of byte instances.
[Bytes->Remove]	Removes bytes form a byte stream. Requires an offset into the byte stream. Optionally accepts a number of bytes to remove.
[Bytes->RemoveLeading]	Removes all occurrances of the parameter sequence from the beginning of the instance. Requires one parameter which is the data to be removed.
[Bytes->RemoveTrailing]	Removes all occurrances of the parameter sequence from the end of the instance. Requires one parameter which is the data to be removed.
[Bytes->Append]	Appends the specified data to the end of the bytes instance. Requires one parameter which is the data to append.
[Bytes->Trim]	Removes all whitespace ASCII characters from the beginning and the end of the instance.
[Bytes->Position]	Returns the current position at which imports will occur in the byte stream.
[Bytes->SetPosition]	Sets the current position within the byte stream. Requires a single integer parameter.
[Bytes->ExportString]	Returns a string represeting the byte stream. Accepts a single parameter which is the character encoding (e.g. ISO-8859-1, UTF-8) for the export. A parameter of 'Binary' will perform a byte for byte export of the stream.
[Bytes->Export8bits]	Returns the first byte as an integer.
[Bytes->Export16bits]	Returns the first 2 bytes as an integer.
[Bytes->Export32bits]	Returns the first 4 bytes as an integer.
[Bytes->Export64bits]	Returns the first 8 bytes as an integer.
[Bytes->ImportString]	Imports a string parameter. A second parameter specifies the encoding (e.g. ISO-8859-1, UTF-8) to use for the import. A second parameter of 'Binary' will perform a byte for byte import of the string.
[Bytes->Import8Bits]	Imports the first byte of an integer parameter.
[Bytes->Import16Bits]	Imports the first 2 bytes of an integer parameter.
[Bytes->Import32Bits]	Imports the first 4 bytes of an integer parameter.

[Bytes->Import64Bits]	Imports the first 8 bytes of an integer parameter.
[Bytes->SwapBytes]	Swaps each two bytes with each other.

To cast string data as a bytes object:

Use the [Bytes] tag. The following example converts a string to a bytes variable.

```
[Var:'Object'=(Bytes: 'This is some text')]
```

To instantiate a new bytes object:

Use the [Bytes] tag. The example below creates an empty bytes object with a size of 1024 bytes and a growth increment of 16 bytes.

```
[Var:'Object'=(Bytes: 1024, 16)]
```

To return the size of a byte stream:

Use the [Bytes->Size] tag. The example below uses a [Field] tag that has been converted to a bytes type using the -Binary parameter.

```
[Var:'Bytes'=(Field:'Name', -Binary)]
[$Bytes->Size]
```

To return a single byte from a byte stream:

Use the [Bytes->Get] tag. An integer parameter specifies the order number of the byte to return. Note that this tag returns a byte, not a fragment of the orignial data (such as a string character).

```
[Var:'Bytes'=(Field:'Name', -Binary)]
[$Bytes->(Get: 1)]
```

To find a value within a byte stream:

Use the [Bytes->Find] tag. The example below returns the starting byte number of the value OmniPilot, which is contained within the byte stream.

```
[Var:'Bytes'=(Field:'Name', -Binary)]
[$Bytes->(Find: 'OmniPilot')]
```

To determine if a value is contained within a byte stream:

Use the [Bytes->Contains] tag. The example below returns True if the value OmniPilot is contained within the byte stream.

```
[Var:'Bytes'=(Field:'Name', -Binary)]
[$Bytes->(Contains: 'OmniPilot')]
```

To add a string to a byte stream:

Use the [Bytes->Append] tag. The following example adds the string I am to the end of a bytes stream.

```
[Var:'Bytes'=(Field:'Name', -Binary)]
[$Bytes->(Append: 'I am')]
```

To find and replace values in a byte stream:

Use the [Bytes->Replace] tag. The following example finds the string Blue and replaces with the string Green within the bytes stream.

```
[Var:'Bytes'=(Bytes: 'Blue Red Yellow')]
[$Bytes->(Replace: 'Blue', 'Green')]
```

To export a string from a bytes stream:

Use the [Bytes->ExportString] tag. The following example exports a string using UTF-8 encoding.

```
[Var:'Bytes'=(Bytes: 'This is a string')]
[$Bytes->(ExportString: 'UTF-8')]
```

To import a string into a bytes stream:

Use the [Bytes->ImportString] tag. The following example imports a string using ISO-8859-1 encoding.

```
[Var:'Bytes'=(Bytes: 'This is a string')]
[$Bytes->(ImportString: 'This is some more string', 'ISO-8859-1')]
```

26

Chapter 26
Math Operations

Numbers in Lasso are stored and manipulated using the decimal and integer data types. This chapter details the symbols and tags that can be used to manipulate decimal and integer values and to perform mathematical operations.

- *Overview* provides an introduction to the decimal and integer data types and how to cast values to and from other data types.
- *Math Symbols* describes the symbols that can be used to create mathematical expressions.
- *Decimal Member Tags* describes the member tags that can be used with the decimal data type.
- *Integer Member Tags* describes the member tags that can be used with the integer data type.
- *Math Tags* describes the substitution and process tags that can be used with numeric values.

Overview

Mathematical operations and number formatting can be performed in LDML using a variety of different methods on integer and decimal values. There are three types of operations that can be performed:

- **Symbols** can be used to perform mathematical calculations within LDML tags or to perform assignment operations within LassoScripts.
- **Member Tags** can be used to format decimal or integer values or to perform bit manipulations.
- **Substitution Tags** can be used to perform advanced calculations.

Each of these methods is described in detail in the sections that follow. This guide contains a description of every symbol and tag and many examples of their use. The LDML Reference is the primary documentation source for LDML symbols and tags. It contains a full description of each symbol and tag including details about each parameter.

Integer Data Type

The integer data type represents whole number values. Basically, any positive or negative number which does not contain a decimal point is an integer value in Lasso. Examples include -123 or 456. Integer values may also contain hexadecimal values such as 0x1A or 0xff.

Spaces must be specified between the + and - symbols and the parameters, otherwise the second parameter of the symbol might be mistaken for an integer literal.

Table 1: Integer Tag

Tag	Description
[Integer]	Casts a value to type integer.

Examples of explicit integer casting:

- Strings which contain numeric data can be cast to the integer data type using the [Integer] tag. The string must start with a numeric value. In the following examples the number 123 is the result of each explicit casting. Only the first integer found in the string 123 and then 456 is recognized.

  [Integer: '123'] ➜ 123
  [Integer: '123 and then 456'] ➜ 123

- Decimals which are cast to the integer data type are rounded to the nearest integer.

  [Integer: 123.0] ➜ 123
  [Integer: 123.999] ➜ 124

Decimal Data Type

The decimal data type represents real or floating point numbers. Basically, any positive or negative number which contains a decimal point is a decimal value in Lasso. Examples include -123.0 and 456.789. Decimal values can also be written in exponential notation as in 1.23e2 which is equivalent to 1.23 times 10^2 or 123.0.

Spaces must be specified between the + and - symbols and the parameters, otherwise the second parameter of the symbol might be mistaken for a decimal literal.

Table 2: Decimal Tag

Tag	Description
[Decimal]	Casts a value to type decimal.

The precision of decimal numbers is always displayed as six decimal places even though the actual precision of the number may vary based on the size of the number and its internal representation. The output precision of decimal numbers can be controlled using the [Decimal->Format] tag described later in this chapter.

Examples of implicit decimal casting:

- Integer values are cast to decimal values automatically if they are used as a parameter to a mathematical symbol. If either of the parameters to the symbol is a decimal value then the other parameter is cast to a decimal value automatically. The following example shows how the integer 123 is automatically cast to a decimal value because the other parameter of the + symbol is the decimal value 456.0.

  [456.0 + 123] ➜ 579.0

  The following example shows how a variable with a value of 123 is automatically cast to a decimal value.

  [Variable: 'Number'=123]
  [456.0 + (Variable: 'Number')] ➜ 579.0

Examples of explicit decimal casting:

- Strings which contain numeric data can be cast to the decimal data type using the [Decimal] tag. The string must start with a numeric value. In the following examples the number 123.0 is the result of each explicit casting. Only the first decimal value found in the string 123 and then 456 is recognized.

  [Decimal: '123'] ➜ 123.0
  [Decimal: '123.0'] ➜ 123.0
  [Decimal: '123 and then 456'] ➜ 123.0

- Integers which are cast to the decimal data type simply have a decimal point appended. The value of the number does not change.

  [Decimal: 123] ➜ 123.0

Mathematical Symbols

The easiest way to manipulate integer and decimal values is to use the mathematical symbols. *Table 3: Mathematical Symbols* details all the symbols that can be used with integer and decimal values.

Table 3: Mathematical Symbols

Symbol	Description
+	Adds two numbers. This symbol should always be separated from its parameters by a space.
-	Subtracts the right parameter from the left parameter. This symbol should always be separated from its parameters by a space.
*	Multiplies two numbers.
/	Divides the left parameter by the right parameter.
%	Modulus. Calculates the left parameter modulo the right number. Both parameters must be integers.

Each of the mathematical symbols takes two parameters. If either of the parameters is a decimal value then the result will be a decimal value. Many of the symbols can also be used to perform string operations. If either of the parameters is a string value then the string operation defined by the symbol will be performed rather than the mathematical operation.

Note: Full documentation and examples for each of the mathematical symbols can be found in the LDML Reference.

Examples of using the mathematical symbols:

- Two numbers can be added using the + symbol. The output will be a decimal value if either of the parameters are a decimal value. Note that the symbol + is separated from its parameters by spaces and negative values used as the second parameter should be surrounded by parentheses.

  [100 + 50] ➜ 150
  [100 + (-12.5)] ➜ 87.5

- The difference between numbers can be calculated using the - symbol. The output will be a decimal value if either of the parameters are a decimal value.

  [100 - 50] ➜ 50
  [100 - (-12.5)] ➜ 112.5

- Two numbers can be multiplied using the * symbol. The output will be a decimal value if either of the parameters are a decimal value.

  [100 * 50] ➜ 5000
  [100 * (-12.5)] ➜ -1250.0

Table 4: Mathematical Assignment Symbols

Symbol	Description
=	Assigns the right parameter to the variable designated by the left parameter.
+=	Adds the right parameter to the value of the left parameter and assigns the result to the variable designated by the left parameter.
-=	Subtracts the right parameter from the value of the left parameter and assigns the result to the variable designated by the left parameter.
*=	Multiplies the value of the left parameter by the value of the right parameter and assigns the result to the variable designated by the left parameter.
/=	Divides the value of the left parameter by the value of the right parameter and assigns the result to the variable designated by the left parameter.
%=	Modulus. Assigns the value of the left parameter modulo the right parameter to the left parameter. Both parameters must be integers.

Each of the symbols takes two parameters. The first parameter must be a variable that holds an integer or decimal value. The second parameter can be any integer or decimal value. The result of the operation is calculated and then stored back in the variable specified as the first operator.

Note: Full documentation and examples for each of the mathematical symbols can be found in the LDML Reference.

Examples of using the mathematical assignment symbols:

- A variable can be assigned a new value using the = symbol. The following example shows how to define an integer variable and then set it to a new value. The new value is output.

```
<?LassoScript
  Variable: 'IntegerVariable'= 100;
  $IntegerVariable = 123456;
  $IntegerVariable;
?>
```

➜ 123456

- A variable can be used as a collector by adding new values using the += symbol. The following example shows how to define an integer variable and then add several values to it. The final value is output.

```
<?LassoScript
  Variable: 'IntegerVariab'e= 0;
  $IntegerVariable += 123;
  $IntegerVariable += (-456);
  $IntegerVariable;
?>
```

➜ -333

Table 5: Mathematical Comparison Symbols

Symbol	Description
==	Returns True if the parameters are equal.
!=	Returns True if the parameters are not equal.
<	Returns True if the left parameter is less than the right parameter.
<=	Returns True if the left parameter is less than or equal to the right parameter.
>	Returns True if the left parameter is greater than the right parameter.
>=	Returns True if the left parameter is greater than or equal to the right parameter.

Each of the mathematical symbols takes two parameters. If either of the parameters is a decimal value then the result will be a decimal value. Many of the symbols can also be used to perform string operations. If either of the parameters is a string value then the string operation defined by the symbol will be performed rather than the mathematical operation.

Note: Full documentation and examples for each of the mathematical symbols can be found in the LDML Reference.

Examples of using the mathematical comparison symbols:

- Two numbers can be compared for equality using the == symbol and != symbol. The result is a boolean True or False. Integers are automatically cast to decimal values when compared.

```
[100 == 123] ➜ False
[100.0 != (-123.0)] ➜ True
[100 ==100.0] ➜ True
[100.0 != (-123)] ➜ False
```

- Numbers can be ordered using the <, <=, >, and <= symbols. The result is a boolean True or False.

  [-37 > 0] ➜ False
  [100 < 1000.0] ➜ True

Decimal Member Tags

The decimal data type includes one member tag that can be used to format decimal values.

Table 6: Decimal Member Tag

Tag	Description
[Decimal->SetFormat]	Specifies the format in which the decimal value will be output when cast to string or displayed to a visitor.

Note: Full documentation and examples for this tag can be found in the LDML Reference.

Decimal Format

The [Decimal->SetFormat] tag can be used to change the output format of a variable. When the variable is next cast to data type string or output to the format file it will be formatted according to the preferences set in the last call to [Decimal->SetFormat] for the variable. If the [Decimal->SetFormat] tag is called with no parameters it resets the formatting to the default. The tag takes the following parameters.

Table 7: [Decimal->SetFormat] Parameters

Keyword	Description
-Precision	The number of decimal points of precision that should be output. Defaults to 6.
-DecimalChar	The character which should be used for the decimal point. Defaults to a period.
-GroupChar	The character which should be used for thousands grouping. Defaults to empty.
-Scientific	Set to True to force output in exponential notation. Defaults to False so decimals are only output in exponential notation if required.
-Padding	Specifies the desired length for the output. If the formatted number is less than this length then the number is padded.
-PadChar	Specifies the character that will be inserted if padding is required. Defaults to a space.
-PadRight	Set to True to pad the right side of the output. By default, padding is appended to the left side of the output.

To format a decimal number as US currency:

Create a variable that will hold the dollar amount, DollarVariable. Use [Decimal->SetFormat] to set the -Precision to 2 and the -GroupChar to comma.

```
[Variable: 'DollarVariable' = 0.0]
[$DollarVariable->(SetFormat: -Precision=2, -GroupChar=',')]
<br>$[$DollarVariable]

[Variable: 'DollarVariable' = $DollarVariable + 1000]
[$DollarVariable->(SetFormat: -Precision=2, -GroupChar=',')]
<br>$[$DollarVariable]

[Variable: 'DollarVariable' = $DollarVariable / 8]
[$DollarVariable->(SetFormat: -Precision=2, -GroupChar=',')]
<br>$[$DollarVariable]
```

```
➜ <br>$0.00
  <br>$1,000.00
  <br>$12.50
```

Integer Member Tags

The integer data type includes many member tags that can be used to format or perform bit operations on integer values. The available member tags are listed in *Table 8: Integer Member Tags*.

Table 8: Integer Member Tags

Tag	Description
[Integer->SetFormat]	Specifies the format in which the integer value will be output when cast to string or displayed to a visitor.
[Integer->BitAnd]	Performs a bitwise And operation between each bit in the base integer and the integer parameter.
[Integer->BitOr]	Performs a bitwise Or operation between each bit in the base integer and the integer parameter.
[Integer->BitXOr]	Performs a bitwise Exclusive-Or operation between each bit in the base integer and the integer parameter.
[Integer->BitNot]	Flips every bit in the base integer.
[Integer->BitShiftLeft]	Shifts the bits in the base integer left by the number specified in the integer parameter.
[Integer->BitShiftRight]	Shifts the bits in the base integer right by the number specified in the integer parameter.
[Integer->BitClear]	Clears the bit specified in the integer parameter.
[Integer->BitFlip]	Flips the bit specified in the integer parameter.
[Integer->BitSet]	Sets the bit specified in the integer parameter.
[Integer->BitTest]	Returns true if the bit specified in the integer parameter is true.

Note: Full documentation and examples for each of the integer member tags can be found in the LDML Reference.

Integer Format

The [Integer->SetFormat] tag can be used to change the output format of a variable. When the variable is next cast to data type string or output to the format file it will be formatted according to the preferences set in the last call to [Integer->SetFormat] for the variable. If the [Integer->SetFormat] tag is called with no parameters it resets the formatting to the default. The tag takes the following parameters.

Table 9: [Integer->SetFormat] Parameters

Keyword	Description
-Hexadecimal	If set to True, the integer will output in hexadecimal notation.
-Padding	Specifies the desired length for the output. If the formatted number is less than this length then the number is padded.
-PadChar	Specifies the character that will be inserted if padding is required. Defaults to a space.
-PadRight	Set to True to pad the right side of the output. By default, padding is appended to the left side of the output.

To format an integer as a hexadecimal value:

Create a variable that will hold the value, HexVariable. Use [Integer->SetFormat] to set -Hexadecimal to True..

```
[Variable: 'HexVariable' = 255]
[$HexVariable->(SetFormat: -Hexadecimal=True)]
<br>[$HexVariable]

[Variable: 'HexVariable' = $HexVariable / 5]
[$HexVariable->(SetFormat: -Hexadecimal=True)]
<br>[$HexVariable]
```

```
➜ <br>0xff
  <br>0x33
```

Bit Operations

Bit operations can be performed within Lasso's 64-bit integer values. These operations can be used to examine and manipulate binary data. They can also be used for general purpose binary set operations.

Integer literals in LDML can be specified using hexadecimal notation. This can greatly aid in constructing literals for use with the bit operation. For example, 0xff is the integer literal 255. The [Integer->SetFormat] tag with a parameter of -Hexadecimal=True can be used to output hexadecimal values.

The bit operations are divided into three categories.

- The [Integer->BitAnd], [Integer->BitOr], and [Integer->BitXOr] tags are used to combine two integer values using the specified boolean operation. In the following example the boolean Or of 0x02 and 0x04 is calculated and returned in hexadecimal notation.

```
[Var: 'BitSet'=0x02]
[$BitSet->(SetFormat: -Hexadecimal=True]
[$BitSet->(BitOr: 0x04]
[$BitSet]
```

➜ 0x06

- The [Integer->BitShiftLeft], [Integer->BitShiftRight], and [Integer->BitNot] tags are used to modify the base integer value in place. In the following example, 0x02 is shifted left by three places and output in hexadecimal notation.

```
[Var: 'BitSet'=0x02]
[$BitSet->(SetFormat: -Hexadecimal=True]
[$BitSet->(BitShift: 3]
[$BitSet]
```

➜ 0x10

- The [Integer->BitSet], [Integer->BitClear], [Integer->BitFlip], and [Integer->BitTest] tags are used to manipulate or test individual bits from an integer value. In the following example, the second bit an integer is set and then tested.

```
[Var: 'BitSet'=0]
[$BitSet->(BitSet: 2)]
[$BitSet->(BitTest 2)]
```

➜ True

Math Tags

LDML contains many substitution tags that can be used to perform mathematical functions. The functionality of many of these tags overlaps the functionality of the mathematical symbols. It is recommended that you use the equivalent symbol when one is available.

Additional tags detailed in the section on *Trigonometry and Advanced Math*.

Table 10: Math Tags

Tag	Description
[Math_Abs]	Absolute value. Requires one parameter.
[Math_Add]	Addition. Returns sum of multiple parameters.
[Math_Ceil]	Ceiling. Returns the next higher integer. Requires one parameter.
[Math_ConvertEuro]	Converts between the Euro and other European Union currencies.
[Math_Div]	Division. Divides each of multiple parameters in order from left to right.
[Math_Floor]	Floor. Returns the next lower integer. Requires one parameter.
[Math_Max]	Maximum of all parameters.
[Math_Min]	Minimum of all parameters.
[Math_Mod]	Modulo. Requires two parameters. Returns the value of the first parameter modulo the second parameter.
[Math_Mult]	Multiplication. Returns the value of multiple parameters multiplied together.
[Math_Random]	Returns a random number.
[Math_RInt]	Rounds to nearest integer. Requires one parameter
[Math_Roman]	Converts a number into roman numerals. Requires one positive integer parameter.
[Math_Round]	Rounds a number with specified precision. Requires two parameters. The first value is rounded to the same precision as the second value.
[Math_Sub]	Subtraction. Subtracts each of multiple parameters in order from left to right.

Note: Full documentation and examples for each of the math tags can be found in the LDML Reference.

If all the parameters to a mathematical substitution tag are integers then the result will be an integer. If any of the parameter to a mathematical substitution tag is a decimal then the result will be a decimal value and will be returned with six decimal points of precision.

In the following example the same calculation is performed with integer and decimal parameters to show how the results vary. The integer example returns 0 since 0.125 rounds down to zero when cast to an integer.

```
[Math_Div: 1, 8] ➜ 0
[Math_Div: 1.0, 8] ➜ 0.125000
```

Examples of using math substitution tags:

The following are all examples of using math substitution tags to calculate the results of various mathematical operations.

```
[Math_Add: 1, 2, 3, 4, 5] ➜ 15
[Math_Add: 1.0, 100.0] ➜ 101.0
[Math_Sub: 10, 5] ➜ 5
[Math_Div: 10, 9] ➜ 11
[Math_Div: 10, 8.0] ➜ 12.5
[Math_Max: 100, 200] ➜ 200
```

Rounding Numbers

Lasso provides a number of different methods for rounding numbers:

- Numbers can be rounded to integer using the [Math_RInt] tag to round to the nearest integer, the [Math_Floor] tag to round to the next lowest integer, or the [Math_Ceil] tag to found to the next highest integer.

```
[Math_RInt: 37.6] ➜ 38
[Math_Floor: 37.6] ➜ 37
[Math_Ceil: 37.6] ➜ 38
```

- Numbers can be rounded to arbitrary precision using the [Math_Round] tag with a decimal parameter. The second parameter should be of the form 0.01, 0.0001, 0.000001, etc.

```
[Math_Round: 3.1415926, 0.0001] ➜ 3.1416
[Math_Round: 3.1415926, 0.001] ➜ 3.142
[Math_Round: 3.1415926, 0.01] ➜ 3.14
[Math_Round: 3.1415926, 0.1] ➜ 3.1
```

- Numbers can be rounded to an even multiple of another number using the [Math_Round] tag with an integer or decimal parameter.

```
[Math_Round: 1463, 1000] ➜ 1000
[Math_Round: 1463, 500] ➜ 1500
[Math_Round: 1463, 20] ➜ 1460
[Math_Round: 1463, 3] ➜ 1464
```

[Math_Round: 3.1415926, 0.5] ➜ 3.0
[Math_Round: 3.1415926, 0.25] ➜ 3.25
[Math_Round: 3.1415926, 1.000] ➜ 3.000
[Math_Round: 3.1415926, 5.0] ➜ 5.0

- If a rounded result needs to be shown to the user, but the actual value stored in a variable does not need to be rounded then either the [Integer->SetFormat] or [Decimal->SetFormat] tags can be used to alter how the number is displayed. See the documentation of these tags earlier in the chapter for more information.

Random Numbers

The [Math_Random] tag can be used to return a random number in a given range. The result can optionally be returned in hexadecimal notation (for use in HTML color variables).

Note: When returning integer values [Math_Random] will return a maximum 32-bit value. The range of returned integers is approximately between +/- 2,000,000,000.

Table 11: [Math_Random] Parameters

Keyword	Description
-Min	Minimum value to be returned.
-Max	Maximum value to be returned. For integer results should be one greater than maximum desired value.
-Hex	If specified, returns the result in hexadecimal notation.

To return a random integer value:

In the following example a random number between 1 and 99 is returned. The random number will be different each time the page is loaded.

[Math_Random: -Min=1, -Max=100]

➜ 55

To return a random decimal value:

In the following example a random decimal number between 0.0 and 1.0 is returned. The random number will be different each time the page is loaded.

[Math_Random: -Min=0.0, -Max=1.0]

➜ 0.55342

To return a random color value:

In the following example a random hexadecimal color code is returned. The random number will be different each time the page is loaded. The range is from 16 to 256 to return two-digit hexadecimal values between 10 and FF.

```
<font color="#[Math_Random: -Min=16, -Max=256, -Hex][Math_Random: -Min=16,
-Max=256, -Hex][Math_Random: -Min=16, -Max=256, -Hex]">Color</font>
```

➜ Color

Trigonometry and Advanced Math

Lasso provides a number of tags for performing trigonometric functions, square roots, logarighthms, and calculating exponents.

Table 12: Trigonmetric and Advanced Math Tags

Tag	Description
[Math_ACos]	Arc Cosine. Requires one parameter. The return value is in radians between 0 and π.
[Math_ASin]	Arc Sine. Requires one parameter. The return value is in radians between -2/π and 2/π.
[Math_ATan]	Arc Tangent. Requires one parameter. The return value is in radians between -2/π and 2/π.
[Math_ATan2]	Arc Tangent of a Quotient. Requires two parameters. The return value is in radians between -π and π.
[Math_Cos]	Cosine. Requires one parameter.
[Math_Exp]	Natural Exponent. Requires one parameter. Returns e raised to the specified power.
[Math_Ln]	Natural Logarithm. Requires one parameter. Also [Math_Log].
[Math_Log10]	Base 10 Logarithm. Requires one parameter.
[Math_Pow]	Exponent. Requires two parameters: a base and an exponent. Returns the base raised to the exponent.
[Math_Sin]	Sine. Requires one parameter.
[Math_Sqrt]	Square Root. Requires one positive parameter.
[Math_Tan]	Tangent. Requires one parameter.

Examples of using advanced math substitution tags:

The following are all examples of using math substitution tags to calculate the results of various mathematical operations.

[Math_Pow: 3, 3] ➜ 27
[Math_Sqrt: 100.0] ➜ 10.0

Locale Formatting

Lasso can format currency, percentages, and scientific values according to the rules of any country or locale. The tags in *Table 13: Locale Formatting Tags* are used for this purpose. Each tag accepts an optional language code and country code which specifies the locale to use for the formatting.

The default is language en for English and country US for the United States. A list of valid language and country codes can be found in the LDML Reference.

Table 13: Locale Formatting Tags

Tag	Description
[Currency]	Formats a number as currency. Requires one parameter, the currency amount to format. The second parameter specifies the language and the third paramter specifies the country for the desired locale.
[Percent]	Formats a number as a percentage. Requires one parameter, the currency amount to format. The second parameter specifies the language and the third paramter specifies the country for the desired locale.
[Scientific]	Formats a number using scientific notation. Requires one parameter, the currency amount to format. The second parameter specifies the language and the third paramter specifies the country for the desired locale.
[Locale_Format]	Formats a number. Requires one parameter, the decimal amount to format. The second parameter specifies the language and the third paramter specifies the country for the desired locale.

27

Chapter 27
Date and Time Operations

Dates and times in Lasso can be stored and manipulated as special date and duration data types. This chapter describes the tags that can be used to manipulate dates and times.

- *Overview* provides an introduction to using the Lasso date and duration data types.
- *Date Tags* describes the substitution and member tags that can be used to cast, format, and display dates and times.
- *Duration Tags* describes the substitution and member tags that can be used to cast, format, and display durations.
- *Date and Duration Math* describes the tags that are used to perform calculations using both dates and durations.

Overview

This chapter introduces the date and the duration data types in LDML 8. Dates are a data type that represent a calendar date and/or clock time. Durations are a data type that represents a length of time in hours, minutes, and seconds. Date and duration data types can be manipulated in a similar manner as integer data types, and operations can be performed to determine date differences, time differences, and more. Date data types may also be formatted and converted to a number of predefined or custom formats, and specific information may be extrapolated from a date data type (day of week, name of month, etc.).

Since dates and durations can take many forms, values that represent a date or a duration must be explicitly cast as date or duration data types using the [Date] and [Duration] tags. For example, a value of 01/01/2002 12:30:00

will be treated as a string data type until it is cast as a date data type using the [Date] tag:

```
[Date:'01/01/2002 12:30:00']
```

Once a value is cast as a date or duration data type, special tags, accessors, conversion operations, and math operations may then be used.

Internal Date Libraries

When performing date operations, Lasso uses internal date libraries to automatically adjust for leap years and daylight saving time for the local time zone in all applicable regions of the world (not all regions recognize daylight saving time). The current time and time zone are based on that of the Web server.

Daylight Saving Time Note: Lasso extracts daylight saving time information from the operating system, and can only support daylight saving time conversions between the years 1970 and 2038. For information on special exceptions with date calculations during daylight saving time, see all the ***Date and Duration Math*** section.

Date Tags

For Lasso to recognize a string as a date data type, the string must be explicitly cast as a date data type using the [Date] tag.

```
[Date: '5/22/2002 12:30:00']
```

When casting as a date data type using the [Date] tag, the following date formats are automatically recognized as valid date strings by Lasso: These automatically recognized date formats are U.S. or MySQL dates with a four digit year followed by an optional 24-hour time with seconds. The "/", "-", and ":" characters are the only punctuation marks recognized in valid date strings by Lasso when used in the formats shown below.

```
1/25/2002
1/25/2002 12:34
1/25/2002 12:34:56
1/25/2002 12:34:56 GMT

2002-01-25
2002-01-25 12:34:56
2002-01-25 12:34:56 GMT
```

Lasso also recognizes a number of special purpose date formats which are shown below. These are useful when working with HTTP headers or email message headers.

```
20020125T12:34:56
Tue, Dec 17 2002 12:34:56 -0800
Tue Dec 17 12:34:56 PST 2002
```

The date formats which contain time zone information (e.g. -0800 or PST) will be recognized as GMT dates. The time zone will be used to automatically adjust the date/time to the equivalent GMT date/time.

If using a date format not listed above, custom date formats can be defined as date data types using the [Date] tag with the -Format parameter.

The following variations of the automatically recognized date formats are valid without using the -Format parameter.

- If the [Date] tag is used without a parameter then the current date and time are returned. Milliseconds are rounded to the nearest second.
- If the time is not specified then it is assumed to be 00:00:00, midnight on the specified date.

  mm/dd/yyyy ➜ mm/dd/yyyy 00:00:00

- If the seconds are not specified then the time is assumed to be even on the minute.

  mm/dd/yyyy hh:mm ➜ mm/dd/yyyy hh:mm:00

- An optional GMT designator can be used to specify Greenwich Mean Time rather than local time.

  mm/dd/yyyy hh:mm:ss GMT

- Two digit years are assumed to be in the 21st century if they are less than 40 or in the 20th century if they are greater than or equal to 40. Two digit years range from 1940 to 2039. For best results, always use four digit years.

  mm/dd/00 ➜ mm/dd/2000
  mm/dd/39 ➜ mm/dd/2039
  mm/dd/40 ➜ mm/dd/1940
  mm/dd/99 ➜ mm/dd/1999

- Days and months can be specified with or without leading 0s. The following are all valid Lasso date strings.

1/1/02	01/01/02
1/1/2002	01/01/2002
1/1/2002 16:35	01/01/2002 16:35
1/1/2002 16:35:45	01/01/2002 16:35:45
1/1/2002 12:35:45 GMT	01/01/2002 12:35:45 GMT

To cast a value as a date data type:

If the value is in a recognized string format described previously, simply use the [Date] tag.

```
[Date: '05/22/2002'] ➜ 05/22/2002 00:00:00
[Date: '05/22/2002 12:30:00'] ➜ 05/22/2002 12:30:00
[Date: '2002-22-05'] ➜ 2002-22-05 00:00:00
```

If the value is not in a string format described previously, use the [Date] tag with the -Format parameter. For information on how to use the -Format parameter, see the *Formatting Dates* section later in this chapter.

```
[Date: '5.22.02 12:30', -Format='%m.%d.%y %H%M'] ➜ 5.22.02 12:30
[Date: '20020522123000', -Format='%Y%m%d%H%M'] ➜ 200205221230
```

Date values which are stored in database fields or variables can be cast to the date data type using the date tag. The format of the date stored in the field or variable should be in one of the format described above or the -Format parameter must be used to explicitly specify the format.

```
[Date: (Variable: 'myDate')]
[Date: (Field: 'Modified_Date')]
[Date: (Action_Param: 'Birth_Date')]
```

Date Tags

LDML contains date substitution tags that can be used to cast date strings as date data types, format date data types, and perform date/time conversions.

Table 1: Date Substitution Tags

Tag	Description
[Date]	Used to cast values to date data types when used with a valid date string as a parameter. An optional -Format parameter with a date format string may be used to explicitly cast an unknown date format. When no parameter is used, it returns the current date and time. An optional -DateGMT keyword/value parameter returns GMT date and time. Also accepts parameters for -Second, -Minute, -Hour, -Day, -Month, -Year, and -DateGMT for constructing and outputting dates.
[Date_Format]	Changes the output format of a Lasso date. Requires a Lasso date data type or valid Lasso date string as a parameter (auto-recognizes the same formats as the [Date] tag). The -Format keyword/value parameter defines how the date should be reformatted. See the Formatting Dates section below for more information.
[Date_SetFormat]	Sets a date format for output using the [Date] tag for an entire Lasso format file. The -Format parameter uses a format string. An optional -TimeOptional parameter causes the output to not return 00:00:00 if there is no time value.
[Date_GMTToLocal]	Converts a date/time from Greenwich Mean Time to local time of the machine running Lasso Service.
[Date_LocalToGMT]	Converts a date/time from local time to Greenwich Mean Time.
[Date_GetLocalTimeZone]	Returns the current time zone of the machine running Lasso Service as a standard GMT offset string (e.g. -0700). Optional -Long parameter shows the name of the time zone (e.g. PDT).
[Date_Minimum]	Returns the minimum possible date recognized by a Date data type in Lasso.
[Date_Maximum]	Returns the maximum possible date recognized as a Date data type in Lasso.

Note: Full documentation and examples for each date tag can be found in the LDML Reference.

To display date values:

- The current date/time can be displayed with [Date]. The example below assumes a current date of 5/22/2002 14:02:05.

  [Date] ➜ 5/22/2002 14:02:05

- The [Date] tag can be used to assemble a date from individual parameters. The following tag assembles a valid Lasso date string by specifying each part of the date separately. Since the time is not specified it is assumed to be midnight on the specified day.

  [Date: -Year=2002, -Month=5, -Day=22] ➜ 5/22/2002 00:00:00

To convert date values to and from GMT:

Any date data type can instantly be converted to and from Greenwich Mean Time using the [Date_GMTToLocal] and [Date_LocalToGMT] tags. These tags will only convert the current time zone of the machine running Lasso Service. The following example uses Pacific Time (PDT) as the current time zone.

[Date_GMTToLocal:(Date:'5/22/2002 14:02:05')] ➜ 5/22/2002 09:02:05
[Date_LocalToGMT:(Date:'5/22/2002 14:02:05')] ➜ 5/22/2002 07:02:05

To show the current time zone for the server running Lasso Service:

The [Date_GetLocalTimeZone] tag displays the current time zone of the machine running Lasso Service. The following example uses Pacific Time (PDT) as the current time zone.

[Date_GetLocalTimeZone] ➜ 0700
[Date_GetLocalTimeZone: -Long] ➜ PDT

Formatting Dates

The [Date] tag and the [Date_Format] tag each have a -Format parameter which accepts a string of symbols that define the format of the date which should be parsed in the case of the [Date] tag or formatted in the case of the [Date_Format] tag. The symbols which can be used in the -Format parameter are detailed in the following table.

Table 2: Date Format Symbols

Symbol	Description
%D	U.S. date format (mm/dd/yyyy).
%Q	MySQL date format (yyyy-mm-dd).
%q	MySQL timestamp format (yyyymmddhhmmss)
%r	12-hour time format (hh:mm:ss [AM/PM]).
%T	24-hour time format (hh:mm:ss).
%Y	4-digit year.
%y	2-digit year.
%m	Month number (01=January, 12=December).
%B	Full English month name (e.g. "January").
%b	Abbreviated English month name (e.g. "Jan").
%d	Day of month (01-31).
%w	Day of week (01=Sunday, 07=Saturday).
%A	Full English weekday name (e.g. "Wednesday").
%a	Abbreviated English weekday name (e.g. "Wed").
%H	24-hour time hour (0-23).
%h	12-hour time hour (1-12).
%M	Minute (0-59).
%S	Second (0-59).
%p	AM/PM for 12-hour time.
%G	GMT time zone indicator.
%z	Time zone offset in relation to GMT (e.g. +0100, -0800).
%Z	Time zone designator (e.g. PST, GMT-1, GMT+12)

Each of the date format symbols that returns a number automatically pads that number with 0 so all values returned by the tag are the same length.

- An optional underscore _ between the percent sign % and the letter designating the symbol specifies that space should be used instead of 0 for the padding character (e.g. %_m returns the month number with space padding).

- An optional hyphen - between the percent sign % and the letter designating the symbol specifies that no padding should be performed (e.g. %-m returns the month number with no padding).
- A literal percent sign can be inserted using %%.

Note: If the %z or %Z symbols are used when parsing a date, the resulting Lasso date object will represent the equivalent GMT date/time.

To convert Lasso date data types to various formats:

The following examples show how to convert either Lasso date data types or valid Lasso date strings to alternate formats.

```
[Date_Format: '06/14/2001', -Format='%A, %B %d'] ➜ Thursday, June 14
[Date_Format: '06/14/2001', -Format='%a, %b %d'] ➜ Thu, Jun 14
[Date_Format: '2001-06-14', -Format='%Y%m%d%H%M'] ➜ 200106140000

[Date_Format: (Date:'1/4/2002'), -Format='%m.%d.%y'] ➜ 01.04.02
[Date_Format: (Date:'1/4/2002 02:30:00'), -Format='%B, %Y '] ➜ January, 2002
[Date_Format: (Date:'1/4/2002 02:30:00'), -Format='%r'] ➜ 2:30 AM
```

To import and export dates from MySQL:

A common conversion in Lasso is converting MySQL dates to and from U.S. dates. Dates are stored in MySQL in the following format yyyy-mm-dd. The following example shows how to import a date in this format to a U.S. date format using the [Date_Format] tag with an appropriate -Format parameter.

```
[Date_Format: '2001-05-22', -Format='%D'] ➜ 5/22/2001
[Date_Format: '5/22/2001', -Format='%Q'] ➜ 2001-05-22

[Date_Format: (Date:'2001-05-22'), -Format='%D'] ➜ 5/22/2001
[Date_Format: (Date:'5/22/2001'), -Format='%Q'] ➜ 2001-05-22
```

To set a custom Lasso date format for a file:

Use the [Date_SetFormat] tag. This allows all date data types on a page to be output in a custom format without the use of the [Date_Format] tag. The format specified is only valid for Lasso code contained in the same file below the [Date_SetFormat] tag.

```
[Date_SetFormat: -Format='%m%d%y']
```

The example above allows the following Lasso date to be output in a custom format without the [Date_Format] tag.

```
[Date:'01/01/2002'] ➜ 010102
```

Date Format Member Tags

In addtion to [Date_Format] and [Date_SetFormat], LDML 8 also offers the [Date->Format] member tags for performing format conversions on date data types.

Table 3: Date Format Member Tags

Symbol	Description
[Date->Format]	Changes the output format of a Lasso date data type. May only be used with Lasso date data types. Requires a date format string as a parameter.
[Date->SetFormat]	Sets a date output format for a particular Lasso date data type object. Requires a date format string as a parameter. An optional -TimeOptional parameter causes the output to not return 00:00:00 if there is no time value.
[Date->Set]	Sets one or more elements of the date to a new value. Accepts the same parameters as the [Date] tag including -Year, -Month, -Day, -Hour, -Minute, Second.

To convert Lasso date data types to various formats:

The following examples show how to convert Lasso date data types to alternate formats using the [Date->Format] tag.

```
[Var:'MyDate'=(Date:'2002-06-14 00:00:00')]
[$MyDate->Format: '%A, %B %d'] ➜ Tuesday, June 14, 2002

[Var:'MyDate'=(Date:'06/14/2002 09:00:00')]
[$MyDate->Format: '%Y%m%d%H%M'] ➜ 200206140900

[Var:'MyDate'=(Date:'01/31/2002')]
[$MyDate->Format: '%d.%m.%y'] ➜ 31.01.02

[Var:'MyDate'=(Date:'09/01/2002')]
[$MyDate->Format: '%B, %Y '] ➜ September, 2002
```

To set an output format for a specific date data type:

Use the [Date->SetFormat] tag. This causes all instances of a particular date data type object to be output in a specified format.

```
[Var:'MyDate'=(Date:'01/01/2002')]
[$MyDate->(SetFormat: '%m%d%y')]
```

The example above causes all instances of [Var:'MyDate'] in the current format file to be output in a custom format without the [Date_Format] or [Date->Format] tag.

```
[Var:'MyDate'] ➜ 010102
```

Date Accessors

A date accessor function returns a specific integer or string value from a date data type, such as the name of the current month or the seconds of the current time. All date accessor tags in LDML 8 are defined in *Table 4: Date Accessor Tags*.

Table 4: Date Accessor Tags

Tag	Description
[Date->Year]	Returns a four-digit integer representing the year for a specified date. An optional -Days parameter returns the number of days in the current year (e.g. 365).
[Date->Month]	Returns the number of the month (1=January, 12=December) for a specified date (defaults to current date). Optional -Long returns the full English month name (e.g. "January") or -Short returns an abbreviated English month name (e.g. "Jan"). An optional -Days parameter returns the number of days in the current month (e.g. 31).
[Date->Day]	Returns the integer day of the month (e.g. 15).
[Date->DayofYear]	Returns integer day of year (out of 365). Will work with leap years as well (out of 366).
[Date->DayofWeek]	Returns the number of the day of the week (1=Sunday, 7=Saturday) for a specified date. Optional -Short returns an abbreviated English day name (e.g. "Sun") and -Long returns the full English day name (e.g. "Sunday").
[Date->Week]	Returns the integer week number for the year of the specified date (out of 52). The -Sunday parameter returns the integer week of year starting from Sunday (default). A -Monday parameter returns integer week of year starting from Monday.
[Date->Hour]	Returns the hour for a specified date/time. An optional -Short parameter returns integer hour from 1 to 12 instead of 1 to 24.
[Date->Minute]	Returns integer minutes from 0 to 59 for a specified date/time.
[Date->Second]	Returns integer seconds from 0 to 59 for the specified date/time.
[Date->Millisecond]	Returns the current integer milliseconds of the current date/time only.
[Date->Time]	Returns the time of a specified date/time.
[Date->GMT]	Returns whether the specified date is in local or GMT time.

To use date accessors:

- The individual parts of the current date/time can be displayed using the [Date->...] tags.

```
[(Date:'5/22/2002 14:02:05')->Year] ➜ 2002
[(Date:'5/22/2002 14:02:05')->Month] ➜ 5
[(Date:'2/22/2002 14:02:05')->(Month: -Long)] ➜ February
[(Date:'5/22/2002 14:02:05')->Day] ➜ 22
[(Date:'5/22/2002 14:02:05')->(DayOfWeek: -Short)] ➜ Wed
[(Date:'5/22/2002 14:02:05')->Time] ➜ 14:02:05
[(Date:'5/22/2002 14:02:05')->Hour] ➜ 14
[(Date:'5/22/2002 14:02:05')->Minute] ➜ 02
[(Date:'5/22/2002 14:02:05')->Second] ➜ 05
```

- The [Date->Millisecond] tag can only return the current number of millisecond value (as related to the clock time) for the machine running Lasso Service.

```
[Date->Millisecond] ➜ 957
```

Duration Tags

A duration is a special data type that represents a length of time. A duration is not a 24-hour clock time, and may represent any number of hours, minutes, or seconds.

Similar to dates, durations must be cast as duration data types before they can be manipulated. This is done using the [Duration] tag. Durations may be cast in an hours:minutes:seconds format, or just as seconds.

```
[Duration:'1:00:00'] ➜ 1:00:00
[Duration:'3600'] ➜ 1:00:00
```

Once a value has been cast as a duration data type, duration calculations and accessors may then be used. Durations are especially useful for calculating lengths of time under 24 hours, although they can be utilized for any lengths of time. Durations are independent of calendar months and years, and durations that equal a length of time longer that one month are only estimates based on the average length of years and months (i.e. 365.2425 days per years, 30.4375 days per month). Duration tags in LDML 8 are summarized in *Table 5: Duration Tags*.

Table 5: Duration Tags

Tag	Description
[Duration]	Casts values as a duration data type. Accepts a duration string for hours:minutes:seconds, or an integer number of seconds. An optional -Week parameter automatically adds a specified number of weeks to the duration. Optional -Day, -Hour, -Minute, and -Second parameters may also be used for automatically adding day, hour, minute, and time increments to the duration.
[Duration->Year]	Returns the integer number of years in a duration (based on an average of 365.25 days per year).
[Duration->Month]	Returns the integer number of months in a duration (based on an average of 30.4375 days per month).
[Duration->Week]	Returns the integer number of weeks in the duration.
[Duration->Day]	Returns the integer number of days in the duration.
[Duration->Hour]	Returns the integer number of hours in the duration.
[Duration->Minute]	Returns the integer number of minutes in the duration.
[Duration->Second]	Returns the integer number of seconds in the duration.

To cast and display durations:

- Durations can be created using the [Duration] tag with the -Week, -Day, -Hour, -Minute, and -Second parameters. This always returns durations in hours:minutes:seconds format.

  [Duration: -Week=5, -Day=3, -Hour=12] ➜ 924:00:00
  [Duration: -Day=4, -Hour=2, -Minute=30] ➜ 98:30:00
  [Duration: -Hour=12, -Minute=45, -Second=50] ➜ 12:45:50
  [Duration: -Hour=3, -Minute=30] ➜ 03:30:00
  [Duration: -Minute=15, -Second=30] ➜ 00:15:30
  [Duration: -Second=30] ➜ 00:00:30

- The -Week, -Day, -Hour, -Minute, and -Second parameters of the [Duration] tag may also be combined with a base duration for ease of use when setting a duration value. This always returns durations in hours:minutes:seconds format.

  [Duration:'5:30:30', -Week=5, -Day=3, -Hour=12] ➜ 929:30:30
  [Duration:'1:00:00', -Day=4, -Hour=2, -Minute=30] ➜ 99:30:00
  [Duration:'3600', -Hour=12, -Minute=45, -Second=50] ➜ 13:45:50

- Specific increments of time can be returned from a duration using the [Duration->...] tags.

```
[(Duration:'8766:30:45')->Year] ➜ 1
[(Duration:'8766:30:45')->Month] ➜ 12
[(Duration:'8766:30:45')->Week] ➜ 52
[(Duration:'8766:30:45')->Day] ➜ 365
[(Duration:'8766:30:45')->Hour] ➜ 8767
[(Duration:'8766:30:45')->Minute] ➜ 525991
[(Duration:'8766:30:45')->Second] ➜ 31559445
```

Date and Duration Math

Date calculations in Lasso can be performed by using special date math tags, durations tags, and math symbols in LDML 8. Date calculations that can be performed include adding or subtracting year, month, week, day, and time increments to and from dates, and calculating time durations. Durations are a new data type that represent a length of time in seconds and are introduced in the preceding *Duration Tags* section.

Daylight Saving Time Note: Lasso does not account for changes to and from daylight saving time when performing date math and duration calculations. One should take this into consideration when performing a date or duration calculation across dates that encompass a change to or from daylight saving time (resulting date may be off by one hour).

Date Math Tags

LDML 8 provides two date math substitution tags for performing date calculations. These tags are generally used for adding increments of time to a date, and output a Lasso date in the format specified. These tags are summarized in *Table 6: Date Math Tags*.

Table 6: Date Math Tags

Tag	Description
[Date_Add]	Adds a specified amount of time to a Lasso date data type or valid Lasso date string. First parameter is a Lasso date. Keyword/value parameters define what should be added to the first parameter: -Millisecond, -Second, -Minute, -Hour, -Day, -Week, -Month, or -Year.
[Date_Subtract]	Subtracts a specified amount of time from a Lasso date data type or valid Lasso date string. First parameter is a Lasso date. Keyword/value parameters define what should be subtracted from the first parameter: -Millisecond, -Second, -Minute, -Hour, -Day, -Week, -Month, or -Year.
[Date_Difference]	Returns the time difference between two specified dates. A duration is the default return value. Optional parameters may be used to ouput a specific integer time value instead of a duration: -Millisecond, -Second, -Minute, -Hour, -Day, -Week, -Month, -Year. Lasso rounds to the nearest integer when using these optional parameters.

To add time to a date:

A specified number of hours, minutes, seconds, days, or weeks can be added to a date data type or valid date string using the [Date_Add] tag. The following examples show the result of adding different values to the current date 5/22/2002 14:02:05.

```
[Date_Add: (Date), -Second=15] ➜ 5/22/2002 14:02:20
[Date_Add: (Date), -Minute=15] ➜ 5/22/2002 14:17:05
[Date_Add: (Date), -Hour=15] ➜ 5/23/2002 05:02:05
[Date_Add: (Date), -Day=15] ➜ 6/6/2002 14:02:05
[Date_Add: (Date), -Week=15] ➜ 9/4/2002 14:02:05
[Date_Add: (Date), -Month=6] ➜ 11/22/2002 14:02:05
[Date_Add: (Date), -Year=1] ➜ 5/22/2003 14:02:05
```

To subtract time from a date:

A specified number of hours, minutes, seconds, days, or weeks can be subtracted from a date data type or valid date string using the [Date_Subtract] tag. The following examples show the result of subtracting different values from the date 5/22/2001 14:02:05.

[Date_Subtract: (Date: '5/22/2001 14:02:05'), -Second=15] ➜ 5/22/2001 14:01:50
[Date_Subtract: (Date:'5/22/2001 14:02:05'), -Minute=15] ➜ 5/22/2001 13:47:05
[Date_Subtract: (Date:'5/22/2001 14:02:05'), -Hour=15] ➜ 5/21/2001 23:02:05
[Date_Subtract: '5/22/2001 14:02:05', -Day=15] ➜ 5/7/2001 14:02:05
[Date_Subtract: '5/22/2001 14:02:05', -Week=15] ➜ 2/6/2001 14:02:05

To determine the time difference between two dates:

Use the [Date_Difference] tag. The following examples show how to calculate the time difference between two date data types or valid date strings.

[Date_Difference: (Date: '5/23/2002'), (Date:'5/22/2002')] ➜ 24:00:00
[Date_Difference: (Date:'5/23/2002'), (Date:'5/22/2002'), -Second] ➜ 86400
[Date_Difference: (Date:'5/23/2002'), '5/22/2002', -Minute] ➜ 3600
[Date_Difference: (Date: '5/23/2002'), '5/22/2002', -Hour] ➜ 24
[Date_Difference: '5/23/2002', (Date:'5/22/2002'), -Day] ➜ 1
[Date_Difference: '5/23/2002', (Date:'5/30/2002'), -Week] ➜ 1
[Date_Difference: '5/23/2002', '6/23/2002', -Month] ➜ 1
[Date_Difference: '5/23/2002', '5/23/2001', -Year] ➜ 1

Date and Duration Math Tags

LDML 8 provides three member tags that perform date math operations requiring both date and duration data types. These tags are used for adding durations to dates, subtracting a duration from a date, and determining a duration between two dates. These tags are summarized in *Table 7: Date and Duration Math Tags*.

Table 7: Date and Duration Math Tags

Tag	Description
[Date->Add]	Adds a duration to a Lasso date data type. Optional keyword/value parameters may be used in place of a duration to define what should be added to the first parameter: -Millisecond, -Second, -Minute, -Hour, -Day, -Week.
[Date->Subtract]	Subtracts a duration from a Lasso date data type. Optional keyword/value parameters may be used in place of a duration to define what should be subtracted from the first parameter: -Millisecond, -Second, -Minute, -Hour, -Day, -Week.

[Date->Difference]	Calculates the duration between two date data types. The second parameter is subtracted from the first parameter to deteremine a duration. Optional parameters may be used to ouput a specified integer time value instead of a duration: -Millisecond, -Second, -Minute, -Hour, -Day, -Week, -Month, -Year. Lasso rounds to the nearest integer when using these optional parameters.

Note: The [Date->Add] and [Date->Subtract] tags do not directly output values, but can be used to change the values of variables that conatin date or duration data types.

To add a duration to a date:

Use the [Date->Add] tag. The following examples show how to add a duration to a date and return a date.

```
[Var_Set:'MyDate'=(Date: '5/22/2002')]
[$MyDate->(Add:(Duration:'24:00:00'))]
[$MyDate] ➜ 5/23/2002 00:00:00
```

```
[Var_Set:'MyDate'=(Date: '5/22/2002')]
[$MyDate->(Add:(Duration:'3600'))]
[$MyDate] ➜ 5/22/2002 12:30:00
```

```
[Var_Set:'MyDate'=(Date: '5/22/2002')]
[$MyDate->(Add: -Week=1)]
[$MyDate] ➜ 5/29/2002 00:00:00
```

To subtract a duration from a date:

Use the [Date->Subtract] tag. The following examples show how to subtract a duration from a date and return a date.

```
[Var_Set:'MyDate'=(Date: '5/22/2002')]
[$MyDate->(Subtract:(Duration:'24:00:00'))]
[$MyDate] ➜ 5/21/2002
```

```
[Var:'MyDate'=(Date: '5/22/2002')]
[$MyDate->(Subtract:(Duration:'7200'))]
[$MyDate] ➜ 5/22/2002 9:30:00
```

```
[Var:'MyDate'=(Date: '5/22/2002')]
[$MyDate->(Subtract: -Day=3)]
[$MyDate] ➜ 5/19/2002 00:00:00
```

To determine the duration between two dates:

Use the [Date->Difference] tag. The following examples show how to calculate the time difference between two dates and return a duration.

```
[Var_Set:'MyDate'=(Date: '5/22/2002')]
[$MyDate->(Difference:(Date:'5/15/2002 01:30:00'))] ➜ 169:30:00

[Var:'MyDate'=(Date: '5/22/2002')]
[$MyDate->(Difference:(Date:'5/15/2002'), -Day)] ➜ 7
```

Using Math Symbols

In LDML 8, one has the ability to perform date and duration calculations using math symbols (similar to integer data types). If a date or duration appears to the left of a math symbol then the appropriate math operation will be performed and the result will be a date or duration as appropriate. All math symbols that can be used with dates or durations are shown in *Table 8: Date Math Symbols*.

Table 8: Date Math Symbols

Tag	Description
+	Used for adding a date and a duration, or adding two durations.
-	Used for subtracting a duration from a date, subtracting a duration from a duration, or determining the duration between two dates.
*	Used for multiplying durations by an interger value.
/	Used for dividing durations by an integer or duration value.

To add or subtract dates and durations:

The following examples show addition and subtraction operations using dates and durations.

```
[(Date: '5/22/2002') + (Duration:'24:00:00')] ➜ 5/23/2002
[(Date: '5/22/2002') - (Duration:'48:00:00')] ➜ 5/20/2002
```

To determine the duration between two dates:

The following calculates the duration between two dates using the minus symbol (-) .

```
[(Date: '5/22/2002') - (Date:'5/15/2002')] ➜ 168:00:00
```

To add one day to the current date:

The following example adds one day to the current date.

```
[(Date) + (Duration: -Day=1)]
```

To multiply or divide a durations by an integer:

The following examples show multiplication and division operations using durations and integers.

```
[(Duration: -Minute=10) * 12] ➜ 02:00:00
[(Duration: '60') * 10] ➜ 00:10:00
[(Duration: -Hour=1) / 2] ➜ 00:30:00
[(Duration: '00:30:00') / 10] ➜ 00:03:00
```

To divide a duration by a duration:

The following examples show division of durations by durations. The resulting value is a decimal data type.

```
[(Duration: -Hour=24) / (Duration: -Hour=6)] ➜ 4.0
[(Duration: '05:00:00') / (Duration: '00:30:00')] ➜ 10.0
```

To return the duration between the current date and a day in the future:

The following example returns the duration between the current date and 12/31/2004.

```
[(Date: '12/31/2004') - (Date)]
```

28

Chapter 28
Arrays, Maps, and Compound Data Types

This chapter describes the array, map, and other compound data types in LDML that allow sets of data to be stored and manipulated.

- *Overview* provides an introduction to the compound data types available in Lasso including tips for deciding which data type to use.
- *Arrays* describes the array data type and its member tags.
- *Lists* describes the list data type and its member tags.
- *Maps* describes the map data type and its member tags.
- *Pair* describes the pair data type and its member tags.
- *Priority Queues* describes the priority queue data type and its member tags.
- *Queues* describes the queue data type and its member tags.
- *Series* describes the series data type.
- *Sets* describes the set data type and its member tags.
- *Stacks* describes the stack data type and its member tags.
- *Tree Maps* describes the tree map data type and its member tags.
- **Comparators** describes tags that can be used to sort the elements within a compound data type.
- *Iterators* describes tags that can be used to cycle through all the elements in a compound data type.
- *Matchers* describe tags that can be used to find elements within a compound data type.

Overview

Lasso includes a large number of compound data types that allow many values to be stored in a single variable. The different data types share many common tags, but also have unique tags which make specific tasks easier. Each is suited to storing a different type of structured data.

- **Arrays** are the most general compound data type and are used to store a sequence of values. Arrays support random access. Values are stored and retrieved based on position. The order of values within the array is preserved and arrays can contain duplicate values. Arrays can contain elements of any data type. Operations on the last element in the array happen in constant time. Operations on other elements of the array happen in linear time.
- **Lists** are used to store a sequence of values. Lists generally allow elements at the start and end to be manipulated. Lists do not allow random access to elements. Elements can be inserted into the middle of a list using an iterator.
- **Maps** are used to store and retrieve values based on a string key. A map only stores one value per key. The order of keys within the map are not preserved. Retrieving a value by key from a map is fast, but iterating through a map is not.
- **Pairs** are used to store two values in an ordered pair. Either the first or second value can be retrieved. Pairs are most commonly used as values within an array or when retrieving parameters in custom tags.
- **Priority Queues** are used to store a sequence of values in sorted order. When a priority queue is created it is given a comparator. Every item that is inserted is automatically sorted based on this comparator. The first item in the list is always the greatest value as determined by the comparator. Only the first item in the list can be examined or removed.
- **Queues** provide first-in first-out behavior. Elements can only be added to the end of the queue. Elements can only be examined and removed from the front of the queue.
- **Series** contain a series of sequential values. Series usually contain integers, but can store a sequence of any data type that supports the ++ symbol.
- **Sets** only contain unique values. All elements in a set are stored sorted in ascending order. Sets do not support random access to elements. Sets support several logical operations including difference, union, and intersection.

- **Stacks** provide last-in first-out behavior. Elements can only be added to the front of the stack and elements can only be examined and removed from the front of the stack.
- **Tree Maps** are used to store and retrieve values based on a key of any data type. A map only stores one value per key. The order of keys within the map is determined by a comparator specified when the tree map is created. The tree map should be used when the types of keys must be preserved.

How to Select a Data Type

Selecting the proper compound data type for a job can greatly reduce the amount of time and overhead that Lasso requires to execute code. Each data type is optimized for a different task.

- The array is the most general storage type that Lasso offers. It should be used whenever there is no strong preference for one of the more specific data types. Lasso uses maps internally to store [Action_Params] and uses arrays as the return value from many tags.
- The map should be used when values need to be stored and looked up by a key. A map can be thought of as similar to a database record with named fields. Lasso uses maps internally to store page variables which are looked up by name.
- If a collection of values need to be operated on in sorted order then a priority queue should be used. As each item is added to a priority queue it is automatically sorted according to a criteria. The highest value can always be retrieved from the queue.
- If a series of values need to be operated on in order then a queue can be used. New values are stored in the end of the queue and the oldest value can be fetched from the beginning of the queue. For example, this can be useful for storing a series of actions that need to take place and then executing them in order.
- The state of a hierarchical or recursive operation can be stored in a stack. Values can be pushed onto the stack and the most recent value can be popped off the stack. For example, this can be useful for recording the current directory within a recursive file operation.
- A collection of unique values can be stored in a set. Any duplicate values will be discarded. Intersections, unions, and differences between multiple sets can be calculated. For example, a collection of categories could be stored in a set without any duplicates.

- If random access is not required in a collection of values then a list should be used. Adding and removing values from a list is more efficient than performing the same operations on an array or a set.
- If values need to be stored and looked up by non-string keys then a tree map should be used. A tree map is not as efficient as a map, but allows the type of each key to be considered when looking up values.
- A series is a shortcut to create an array that contains sequential values.

Common Tags

Many of the compound data types support a common set of tags. These tags can be used interchangeably between several different data types. Although the specific meaning of each tag varies depending on which data type it is used with.

- **Size** – The [...->Size] tags return the number of elements in each data type.
- **Get** – The [...->Get] tags return a specified element from each data type. In the array and set types the element is specified by a position greater than one and the return value can be of any data type. Negative position values can be used to return an element counting back from the last element of the type.

  In the map and tree map types the the return value will always be a pair with the key and value for one particular element of the map. The pair type supports get with a parameter of 1 or 2 corresponding to [Pair->First] and [Pair->Second].

  In the stack, queue, and priority queue types the position parameter is ignored and the first value of the type is returned and removed from the type. This means that calling [...->Get] repeatedly on a stack, queue, or priority queue will consume elements of the type, eventually leaving an empty type.
- **Insert** – The [...->Insert] tags insert a new element into a compound data type. Arrays and sets by default insert elements at the end of the type. An optional second parameter specifies at what position to insert a new element.

  The map and tree map types insert values using a key and a value. The list type allows values to be inserted either at the end or the beginnging of the type.

  The stack, queue, and priority queue types allow values only to be inserted.
- **Remove** – The [...->Remove] tags remove an element from a compound data type. The value of the removed element is not returned.

- **First, Second, and Last** – The […->First] tags allow the first element of the array, list, pair, queue, priorityqueue, set, and stack types to be inspected. The array, pair, and set types also support […->Second] tags that allow the second element of each type to be inspected. The array, list, and set types also support […->Last] tags that allow the last element of each type to be inspected.
- **Iterate** – The [Iterate] ... [/Iterate] tags insert a new element into a compound data type. Arrays and sets by default insert elements at the end of the type. An optional second parameter specifies at what position to insert a new element.

Comparators

A comparator is a tag which is able to compare two values to each other. Comparators can be passed to several different compound data type creator tags and member tags. Comparators are always passed by reference and are never called directly. These tags should not be used within conditionals.

Comparators are used to sort arrays, priority queues, sets, and tree maps. For example, a priority queue can be initialized with a comparator. Any values that are greater with respect to the comparator will be sorted to the front of the priority queue. In the following code a priority queue is created using the comparator \Compare_LessThan which has the effect of sorting the lowest value to the front of the queue.

```
[Var: 'myPriorityQueue' = (PriorityQueue: \Compare_LessThan)]
```

See the section on *Comparators* later in this chapter for full details and for instructions about how to create new comparators.

Matchers

A matcher is a data type that accepts one value when it is created. It can then be compared against other values. Matchers can be used to select a subset of a compound data type. For example, the matcher [MathRegExp] matches any element according to the specified regular expression. The following matcher would match any capatalized word.

```
[Match_RegExp: '[A-Z][a-z]+']
```

See the section on *Matchers* later in this chapter for full details and for instructions about how to create new matchers.

Iterators

Iterators allow each element within a compound data type to be explored in turn. An iterator is returned by the [...->Iterator] or [...->ReverseIterator] tags of the array, list, map, set, and tree map types. Custom types might also return iterators.

For example, the following example shows how the [While] ... [/While] tags can be used with the [Array->Iterator] to iterate through each value within an array $myArray. The [Null] tag suppresses the output from [Iterator->Forward].

```
<?LassoScript
  Var: 'myArray' = Array('Alpha','Beta','Gamma','Delta');
  Var: 'myIterator' = $myArray->Iterator;

  While: ($myIterator->atEnd == False);
    '<br />' + $myIterator->Value;
    Null: $myIterator->Forward;
  /While;
?>
```

See the section on *Iterators* later in this chapter for full details and for instructions about how to create new iterators.

Arrays

An array is a sequence of values which are stored and retrieved by numeric position. The values stored in an array can be of any data type in LDML. Arrays can store any values from strings and integers to other arrays and maps. By nesting compound data types very complex data structures can be created.

Types of Arrays

Arrays can be used in LDML for several different purposes. The same member tags can be used on each type of array, but some have specific uses when used with a particular type of array. These specific uses are described in the examples for each member tag.

- A **List Array** is a sequence of string, decimal, or integer values. New values can be appended to the end of the list or inserted between two elements of the list using [Array->Insert]. Two lists can be merged using [Array->Merge]. The order of elements in the array is important, but may be manipulated using the array member tags.

- A **Storage Array** is a sequence of "cubby holes" for values. Values are stored into a slot identified by an integer and later retrieved. The [Array->Get] tag is used to store and retrieve values, but the order of elements in the array is never altered and multiple arrays are never merged.
- A **Pair Array** is a sequence of pairs. [Action_Params] returns an array of pairs which identify the command tags and name/value pairs that comprise the current Lasso action. This array can be manipulated and then passed as a parameter to an [Inline] tag.

Creating Arrays

Arrays are created using the [Array] constructor tag. The parameters of the tag become the initial values stored in the array. The parameters can be string, decimal, or integer literals, constructor tags for other complex data types, or name/value pairs which are interpreted as pairs to be added to the array.

Table 1: Array Tag

Tag	Description
[Array]	Creates an array that contains each of the parameters of the tag. If no parameters are specified, an empty array is created.

To create an array:

- The following example creates an empty array and stores it in a variable.

```
[Variable: 'EmptyArray' = (Array)]
```

- The following example shows an array of string literals.

```
[Array: 'String One', 'String Two', 'String Three']
```

- The following example shows an array with a combination of string, decimal, and integer literals.

```
[Array: 'String One', 2, 3.333333]
```

- The following example shows how to use values from database fields, form parameters, variables, or tokens as the initial values for an array.

```
[Array: (Field: 'Field_Name'), (Action_Param: 'Parameter_Name'),
  (Variable: 'Variable_Name'), (Token_Value: 'Token_Name')]
```

- The following example shows an array of pairs. Each name/value pair becomes a single pair within the array returned by the tag.

```
[Array: 'Name_One'='Value_One', 'Name_Two'='Value_Two']
```

- The following example shows an array of arrays. The array returned by the following code will only contain two array elements. Each array element will in turn contain two integer elements. Nested arrays can be used to store mathematical multi-dimensional arrays.

```
[Array: (Array: 1, -1), (Array: -1, 0)]
```

- The following example shows how to create an array from a string. The [String->Split] tag can be used to split a string into an array which contains one element for each substring delimited by the parameter to the tag. The following string is split on the comma , character into an array of four elements.

```
['One,Two,Three,Four,Five'->Split(',')]
```

Values are always copied into an array. They are never stored by reference to the original value. This applies both to simple data types and compound data types. There is no way in LDML to store a reference to a compound data type, except for the name of the variable containing the data type.

Array Member Tags

The array data type has a number of member tags that can be used to store, retrieve or delete array elements or to otherwise manipulate array values.

Table 2: Array Member Tags

Tag	Description
[Array->Contains]	Returns True if the specified element is contained in the array. Requires a single parameter which is a value to compare to each element of the array.
[Array->Difference]	Compares the array against another returning a new array that contains only elements of the current array which are not contained in the other array. Requires one parameter which is an array to be compared. Note: Both arrays must be sorted using the same order or comparator.
[Array->Find]	Returns an array of elements that match the parameter. Accepts a single parameter of any data type.
[Array->FindPosition]	Returns an array of the indicies for elements that match the parameter. Accepts a single parameter of any data type. (Note: This tag was previously named [Array->FindIndex].)
[Array->First]	Returns the first element of the array.

[Array->ForEach]	Applies a tag to each element of the array in turn. Requires a single parameter which is a reference to a tag or compound data type. Modifies the array in place and returns no value.
[Array->Get]	Returns an item from the array. Accepts a single integer parameter identifying the position of the item to be returned. This tag can be used as the left parameter of an assignment operator to set an element of the array.
[Array->Insert]	Inserts a value into the array. Accepts a single parameter which is the value to be inserted and an optional integer parameter identifying the position of the location where the value should be inserted. Defaults to the end of the array. Returns no value.
[Array->InsertFirst]	Inserts an element at the front of the array. Accepts a single parameter which is the value to be inserted.
[Array->InsertFrom]	Inserts elements from an iterator. All of the inserted elements are inserted at the end of the array in order. Requires a single parameter which is an iterator from another compound data type.
[Array->InsertLast]	Inserts an element at the end of the array. Accepts a single parameter which is the value to be inserted
[Array->Intersection]	Compares the array against another returning a new array that contains only elements contained in both arrays. Requires one parameter which is an array to be compared. Note: Both arrays must be sorted using the same order or comparator.
[Array->Iterator]	Returns an iterator to step through every element of the array. An optional parameter specifies a comparator which selects which elements of the array to return.
[Array->Join]	Joins the items of the array into a string. Accepts a single string parameter which is placed inbetween each item from the array. The opposite of [String->Split].
[Array->Last]	Returns the last item in the array.
[Array->Merge]	Merges an array parameter into the array. Accepts an array parameter and three integer parameters that identify which items from the array parameter should be inserted into the array. Defaults to inserting the entire array parameter at the end of the array. Returns no value.
[Array->Remove]	Removes an item from the array. Accepts a single integer parameter identifying the position of the item to be removed. Defaults to the last item in the array. Returns no value.

[Array->RemoveAll]	Removes any elements that match the parameter from the array. Accepts a single parameter of any data type. Returns no value.
[Array->RemoveFirst]	Removes the first element of the array. Returns no value.
[Array->RemoveLast]	Removes the last element of the array. Returns no value.
[Array->Reverse]	Revers the order of all elements of the array. Modifies the array in place and returns no value.
[Array->ReverseIterator]	The same as iterator, but returns the elements in the reverse order. See iterator for more details.
[Array->Reserve]	Reserves storage for the specified number of elements. The size of the array is not changed by [Array->Reserve], but Lasso will internally make enough room for the specified number of elements.
[Array->Second]	Returns the second element of the array.
[Array->Size]	Returns the number of elements in the array.
[Array->Sort]	Reorders the elements of the array in alphabetical or numerical order. Accepts a single boolean parameter. Sorts in ascending order by default or if the parameter is True and in descending order if the parameter is False.
[Array->SortWith]	Reorders the elements of the array in the order defined by a comparator. Requires one parameter which is a comparator used to determine the new order of elements in the array. Modifies the array in place and returns no value.
[Array->Union]	Returns a new array that contains all of the elements from two arrays without duplicates. Requires one parameter which is an array to be added to the current array. Note: Both arrays must be sorted using the same order or comparator.

The following examples show how to manipulate an array by getting, setting, inserting, and deleting values. The examples are all based on the following array which contains the seven days of the week in English.

```
[Variable: 'DaysOfWeek' = (Array: 'Sunday', 'Monday', 'Tuesday', 'Wednesday',
  'Thursday', 'Friday', 'Saturday')]
```

To get the size of an array:

Use the [Array->Size] tag. The following example shows how to output the size of the DaysOfWeek array.

```
[$DaysOfWeek->Size] ➜ 7
```

To get elements of an array:

- To get an element of the array use the [Array->Get] tag with the appropriate position. In the following example different elements of the DaysOfWeek array are returned.

```
[$DaysOfWeek->(Get: 1)] ➜ Sunday
[$DaysOfWeek->(Get: 4)] ➜ Wednesday
```

- The last element of the array can be returned by using [Array->Get] with a parameter of [Array->Size]. [Array->Size] will return 7 since the array DaysOFWeek is 7 elements long and element 7 of the array is Saturday.

```
[$DaysOFWeek->(Get: ($DaysOfWeek->Size))] ➜ Saturday
```

- All of the elements in the array can be returned using [Iterate] ... [/Iterate] tags. The following example shows how to list all of the days of the week.

```
[Iterate: $DaysOfWeek, (Variable: 'DayName')]
  <br>[Variable: 'DayName']
[/Iterate]

➜ <br>Sunday
<br>Monday
<br>Tuesday
<br>Wednesday
<br>Thursday
<br>Friday
<br>Saturday
```

- Alternately, all of the elements in the array can be returned using [Loop] ... [/Loop] tags. The following example shows how to list all of the days of the week by using [Array->Get] with a parameter of [Loop_Count].

```
[Loop: ($DaysOfWeek->Size)]
  <br>[$DaysOFWeek->(Get: (Loop_Count))]
[/Loop]

➜ <br>Sunday
<br>Monday
<br>Tuesday
<br>Wednesday
<br>Thursday
<br>Friday
<br>Saturday
```

To set elements of an array:

The [Array->Get] member tag can be used on the left side of an assignment operator to set the value stored in the specified position within the array.

- In the following example, the value of the second element of the array DaysOfWeek is set to the Spanish word for Monday, Lunes.

```
<?LassoScript
  $DaysOfWeek->(Get: 2) = 'Lunes';
?>
```

The value of the second element of the array can then be output using the [Array->Get] tag.

```
[$DaysOfWeek->(Get: 2)] ➜ Lunes
```

- Elements of the array can be modified using any of the assignment symbols. In the following example, the substring day is removed from the third element of the array using the deletion assignment symbol -= leaving Tues. This value is then output.

```
<?LassoScript
  $DaysOFWeek->(Get: 3) -= 'day';
  $DaysOfWeek->(Get: 3);
?>

➜ Tues
```

To insert elements into an array:

- The [Array->Insert] tag can be used to insert a single element in the array. In the following example Sunday is inserted at the end of the array DaysOfWeek. The whole array is then output.

```
<?LassoScript
  $DaysOfWeek->(Insert: 'Sunday');

  Loop: ($DaysOfWeek->Size);
    $DaysOfWeek->(Get: (Loop_Count)) + ' ';
  /Loop;
?>

➜ Sunday Monday Tuesday Wednesday Thursday Friday Saturday Sunday
```

- The [Array->Insert] tag can also be used to insert a single element anywhere in the array. In the following example Tuesday is inserted as the third element of the array DaysOfWeek. This pushes back all the other elements of the array. No values in the array are removed or replaced by the [Array->Insert] tag. The whole array is then output.

```
<?LassoScript
  $DaysOfWeek->(Insert: 'Tuesday', 3);

  Loop: ($DaysOfWeek->Size);
    $DaysOfWeek->(Get: (Loop_Count)) + ' ';
  /Loop;
?>
```

```
➜ Sunday Monday Tuesday Tuesday Wednesday Thursday Friday Saturday
```

To remove elements from an array:

- The [Array->Remove] tag can be used to remove a single element from the array. If no parameter is specified then the last item of the array is removed. In the following example the last item of the array Saturday is removed and then the entire array is displayed.

```
<?LassoScript
  $DaysOfWeek->(Remove);

  Loop: ($DaysOfWeek->Size);
    $DaysOfWeek->(Get: (Loop_Count)) + ' ';
  /Loop;
?>

➜ Sunday Monday Tuesday Wednesday Thursday Friday
```

- The [Array->Remove] tag can also be used to remove a single element anywhere in the array. In the following example the fourth value in the array is removed. This removes the element Wednesday. The whole array is then output.

```
<?LassoScript
  $DaysOfWeek->(Remove: 4);

  Loop: ($DaysOfWeek->Size);
    $DaysOfWeek->(Get: (Loop_Count)) + ' ';
  /Loop;
?>

➜ Sunday Monday Tuesday Thursday Friday Saturday
```

To display the elements of an array:

- Arrays can be displayed by simply outputting the variable that contains the array. All of the elements of the array are displayed surrounded by parentheses. This is useful primarily for debugging purposes so the values in an array can be inspected without writing a loop to output all of the elements of the array.

```
[Variable: 'DaysOfWeek']

➜ (Array: (Sunday), (Monday), (Tuesday), (Wednesday), (Thursday), (Friday),
(Saturday))
```

- Arrays can be displayed by joining the elements of the array into a string. In the following example the days of the week are output with commas between each element.

```
[Output $DaysOfWeek->(Join: ',')]
```

```
➜ Sunday,Monday,Tuesday,Wednesday,Thursday,Friday,Saturday
```

Arrays and Strings

Arrays can be used for string manipulation using a combination of array and string member tags. First, the string to be manipulated is transformed into an array using the [String->Split] tag, then the array is manipulated, and finally the string is rendered from the array using the [Array->Join] tag.

The following example demonstrates how to modify a URL which is stored in a variable. This same technique can be used to modify any string which can be split into array elements based on a specific delimiter.

To parse, modify, and reassemble a URL using array tags:

1 Store the URL to be modified in a string variable, here named URL_Variable.

```
[Variable: 'URL_Variable' = 'http://www.example.com/default.lasso?
  -FindAll&-Database=Contacts&-Table=People&-KeyField=ID']
```

2 Use [String->Split] to break the URL apart into several different variables. First, the string is split on ? to split the base of the URL from the parameters. These two parameters are stored in temporary variables, URL_Base and URL_Parameters.

```
[Variable: 'Temp_Array' = ($URL_Variable->(Split: '?'))]
[Variable: 'URL_Base' = ($Temp_Array->(Get: 1))]
[Variable: 'URL_Parameters' = ($Temp_Array->(Get: 2))]
```

3 Use [String->Split] to break the URL parameters apart into an array at the ampersand & character.

```
[Variable: 'URL_Array' = ($URL_Parameters->(Split: '&'))]
```

4 Now the parameters array can be manipulated. For this example we will sort it using the [Array->Sort] command. Other options include removing or inserting elements, merging two or more URL parameter arrays, checking for the existence of specific values, etc.

```
[($URL_Array->Sort)]
```

5 Reassemble the URL parameters using the [Array->Join] tag to append each item in the array to a new variable URL_Parameters. An ampersand & is placed between each element of the array.

```
[Variable: 'URL_Parameters'=$URL_Array->(Join: '&')]
```

6 Reassemble the full URL by concatenating the original URL_Base to the new URL_Parameters and store the result in URL_Variable.

```
[Variable: 'URL_Variable' = $URL_Base + '?' + $URL_Parameters]
```

7 Display the modified URL to confirm that the modifications have been made correctly. The command tags in the URL are now sorted alphabetically.

```
[Variable: 'URL_Variable']
➜ http://www.example.com/default.lasso?
  -Database=Contacts&-FindAll&-KeyField=ID&-Table=People
```

Merging Arrays

The [Array->Merge] tag can be used to merge two arrays by placing the elements of the tag's array parameter into the base array. The [Array->Merge] accepts a number of parameters as detailed in the following table.

Table 3: [Array->Merge] Parameters

Parameter	Description
First	The array which is to be merged; the source array.
Second	The position in the destination array where the elements of the source array should be inserted. Optional, defaults to the end of the destination array.
Third	The position in the source array of the first element which should be inserted into the destination array. Optional, defaults to 1.
Fourth	The number of elements from the source array to insert into the destination array. Optional, defaults to all elements from the third parameter to the end of the source array.

The four parameters to [Array->Merge] allow for a selected subset of the source array to be placed at any location in the destination array. This allows very complex array manipulations to be performed.

To append an array to the end of another array:

Use the [Array->Merge] tag with a single array parameter. All the elements of the array parameter will be inserted at the end of the base array. In the following example, two arrays are created, each containing three integers. The elements of the second array are merged into the elements of the first array and then all the elements of the new array are displayed.

```
<?LassoScript
  Variable: 'First_Array' = (Array: 1, 2, 3);
  Variable: 'Second_Array' = (Array: 4, 5, 6);

  $First_Array->(Merge: $Second_Array);

  $First_Array;
?>
```

➜ (Array: (1), (2), (3), (4), (5), (6))

To insert a single element from one array into another array:

In the following example the third element of the Second_Array is inserted as the new first element of the First_Array using the [Array->Merge] tag. The second parameter to [Array->Merge] is set to 1 so the element will be inserted as the first element of First_Array. The third parameter is set to 3 so the third element of Second_Array will be selected. The fourth parameter is set to 1 so only one element of Second_Array will be copied.

```
<?LassoScript
  Variable: 'First_Array' = (Array: 1, 2, 3);
  Variable: 'Second_Array' = (Array: 4, 5, 6);

  $First_Array->(Merge: $Second_Array, 1, 3, 1);

  $First_Array;
?>
```

➜ (Array: (6), (1), (2), (3))

Finding Elements of an Array

The [Array->Find] tag can be used to return a subset of an array which matches a specified value. This can be used to determine whether an array contains a value or, when used in concert with the [Array->RemoveAll] tag this can be used to extract a number of elements from an array.

To determine whether an array contains a value:

In the following example the array DaysOfWeek is checked to see if it contains an element Thursday using the contains symbol >>.

```
<?LassoScript
  Variable: 'DaysOfWeek'= (Array: 'Sunday', 'Monday', 'Tuesday', 'Wednesday',
    'Thursday', 'Friday', 'Saturday');

  If: ($DaysOfWeek >> 'Thursday');
    'The array contains Thursday!';
  /If;
?>
```

To find the indices where an element occurs within the array:

In the following example, an array is returned that reports the position of each occurrence of 1 within the array.

```
<?LassoScript
  Variable: 'Find_Array' = (Array: 6, 1, 4, 1, 5, 1, 2, 3, 1);

  $Find_Array->(FindPosition: 1);
?>
```

➜ (Array: (2), (4), (6), (9))

The result array can be used to modify each occurrence of 1 within the array. In the following example each occurrence of 1 within the array is changed to 0.

```
<?LassoScript
  Variable: 'Find_Array' = (Array: 6, 1, 4, 1, 5, 1, 2, 3, 1);

  Variable: 'Temp_Array' = $Find_Array(FindPosition: 1);

  Iterate: $Temp_Array, (Variable: 'Temp_Position');
    $Find_Array->(Get: $Temp_Position) = 0;
  /Iterate;

  $Find_Array;
?>
```

➜ (Array: (6), (0), (4), (0), (5), (0), (2), (3), (0))

To delete elements with a certain value from an array:

In the following example, all elements with value 1 are deleted from an array Delete_Array. The initial array contains many different integer values. The resulting array is output after all the elements with value 1 have been deleted.

```
<?LassoScript
  Variable: 'Delete_Array' = (Array: 6, 1, 4, 1, 5, 1, 2, 3, 1);

  $Delete_Array->(RemoveAll: 1);

  $Delete_Array;
?>
```

➜ (Array: (6), (4), (5), (2), (3))

Pair Arrays

Pair arrays can be used to store a sequence of name/value pairs. The order of elements within a pair array is maintained. The [Action_Params] and [Params] tags both return pair arrays which contain the parameters passed with the current Lasso action or into a custom tag respectively.

To create a pair array:

Use the [Array] tag with name/value parameters. Each name/value parameter becomes a pair in the resulting array. The following example shows an array created with three pair elements.

```
[Array: 'Name_One'='Value_One',
  'Name_Two'='Value_Two',
  'Name_Three'='Value_Three']
```

To find pairs within a pair array:

The [Array->Find] tag can be used to find pairs within a pair array. The parameter passed to the [Array->Find] tag is only compared to the [Pair->First] element of each pair. The [Array->Find] tag returns an array that contains only the pairs whose first part matches the parameter. The following example shows an array defined with three pair elements. The [Array->Find] tag is used to return both elements for the name Alpha.

```
[Variable: 'Pair_Array' = (Array: 'Alpha'='One', 'Beta'='Two', 'Alpha'=1, 'Beta'=2)]
[$Pair_Array->(Find: 'Alpha')]
```

➜ (Array: (Pair: (Alpha)=(One)), (Pair: (Alpha)=(1)))

To insert pairs into a pair array:

Use the [Array->Insert] tag with a name/value parameter. The new element will be inserted at the end of the array by default. The following example inserts a new element Gamma=Three into Pair_Array.

```
<?LassoScript
  Variable: 'Pair_Array' = (Array: 'Alpha'='One', 'Beta'='Two', 'Alpha'=1, 'Beta'=2);
  $Pair_Array->(Insert: 'Gamma'='Three');
?>
```

Sorting Arrays

Arrays can be sorted using the [Array->Sort] tag. This tag reorders the elements of the array so they will no longer be available at the position they were originally set.

Examples of sorting arrays:

- The following LassoScript shows an array with integer elements. The array is sorted and then the values of the array are output. The default sort order is ascending.

```
<?LassoScript
  Variable: 'Sort_Array' = (Array: 6, 4, 5, 2, 3, 1);

  $Sort_Array->(Sort);

  $Sort_Array;
?>
```

➜ (Array: (1), (2), (3), (4), (5), (6))

- The following LassoScript shows the DaysOFWeek array being sorted in descending alphabetical order. The [Array->Sort] tag accepts one parameter. True for ascending order or False for descending order. The default is True.

```
<?LassoScript
  Variable: 'DaysOfWeek'= (Array: 'Sunday', 'Monday', 'Tuesday', 'Wednesday',
    'Thursday', 'Friday', 'Saturday');

  $DaysOfWeek->(Sort: False);

  $DaysOfWeek;
?>
```

➜ (Array: (Wednesday), (Tuesday), (Thursday), (Sunday), (Saturday), (Monday), (Friday))

Lists

A list is a sequence of values. values are stored and retrieved only from the front or end of a list. Lists do not support random access to value stored in the center of the list. The values stored in a list can be of any data type in LDML.

Table 4: List Tag

Tag	Description
[List]	Creates a new list. Any parameters passed to the tag are used as the initiali values for the list.

To create a list:

Lists are created using the [List] tag. The parameters of the tag define the initial element of the list. Additional lements can be added using the [List->InsertFirst] or [List->InsertLast] tags. For example, the following code creates a list with the value Two and then inserts One and Three into it.

```
<?LassoScript
  Var: 'myList' = (List: 'Two');
  $myList->InsertFirst('One');
  $myList->InsertLast('Three');
?>
```

List Member Tags

The list data type has a number of member tags that can be used to store, retrieve or delete array elements or to otherwise manipulate array values.

Table 5: List Member Tags

Tag	Description
[List->Contains]	Returns True if the specified element is contained in the list. Requires a single parameter which is a value to compare to each element of the list.
[List->Difference]	Compares the list against another returning a new list that contains only elements of the current list which are not contained in the other list. Requires one parameter which is an list to be compared. Note: Both list must be sorted using the same order or comparator.
[List->Find]	Returns an array of elements that match the parameter. Accepts a single parameter of any data type.
[List->First]	Returns the first element of the list.
[List->ForEach]	Applies a tag to each element of the liar in turn. Requires a single parameter which is a reference to a tag or compound data type. Modifies the liar in place and returns no value.
[List->Insert]	Inserts a value into the list. Accepts a single parameter which is the value to be inserted and an optional iterator identifying the location where the value should be inserted. Defaults to the end of the list. Returns no value.
[List->InsertFirst]	Inserts an element at the front of the list. Accepts a single parameter which is the value to be inserted.

[List->InsertFrom]	Inserts elements from an iterator. All of the inserted elements are inserted at the end of the list in order. Requires a single parameter which is an iterator from another compound data type.
[List->InsertLast]	Inserts an element at the end of the list. Accepts a single parameter which is the value to be inserted
[List->Intersection]	Compares the list against another returning a new list that contains only elements contained in both lists. Requires one parameter which is a list to be compared. Note: Both lists must be sorted using the same order or comparator.
[List->Iterator]	Returns an iterator to step through every element of the list. An optional parameter specifies a comparator which selects which elements of the list to return.
[List->Join]	Joins the items of the lists into a string. Accepts a single string parameter which is placed inbetween each item from the lists.
[List->Last]	Returns the last item in the lists.
[List->Remove]	Removes an item from the lists. Accepts an iterator parameter identifying the item to be removed. Defaults to the last item in the list. Returns no value.
[List->RemoveAll]	Removes any elements that match the parameter from the list. Accepts a single parameter of any data type. Returns no value.
[List->RemoveFirst]	Removes the first element of the list. Returns no value.
[List->RemoveLast]	Removes the last element of the list. Returns no value.
[List->Reverse]	Reverses the order of all elements of the list. Modifies the list in place and returns no value.
[List->ReverseIterator]	The same as iterator, but returns the elements in the reverse order. See iterator for more details.
[List->Second]	Returns the second element of the list.
[List->Size]	Returns the number of elements in the list.
[List->Sort]	Reorders the elements of the list in alphabetical or numerical order. Accepts a single boolean parameter. Sorts in ascending order by default or if the parameter is True and in descending order if the parameter is False.
[List->SortWith]	Reorders the elements of the list in the order defined by a comparator. Requires one parameter which is a comparator used to determine the new order of elements in the list. Modifies the list in place and returns no value.

[List->Union]	Returns a new list that contains all of the elements from two list without duplicates. Requires one parameter which is a list to be added to the current list. Note: Both lists must be sorted using the same order or comparator.

To inspect the first and last elements in a list:

Use the [List->First] tag and the [List->Last] tag. These tags will return the values for the first and last element the list. In the following example a list is created with some values and then the first and last values are returned.

```
<?LassoScript
  Var: 'MyList' = (List: 'Uno', 'Dos', 'Tres', 'Quatro');
  $myList->First + ', ' + $myList->Last;
?>
```

➜ Uno, Quatro

To return the number of elements in the list:

Use the [List->Size] tag. This tag returns an integer representing the number of elements that are contained within the list. The following code outputs the size of the list created above.

```
[$myList->Size]
```

➜ 4

To insert new values in a list:

Use the [List->InsertFirst] or [List->InsertLast] tags. Values can only be inserted into the beginning or end of a list. The following examples inserts additional values into the list defined above and then outputs the new list.

```
[$myList->InsertFirst('Cero')]
[$myList->InsertLast('Cinco')]

[String: $myList]
```

➜ List: Cero, Uno, Dos, Tres, Quatro, Cinco

To remove values from a list:

Use the [List->RemoveFirst] or [List->RemoveLast] tags. Values can only be removed from the beginning or end of a list. The following examples removes the first and last values from the list defined above and then outputs the new list.

```
[$myList->RemoveFirst]
[$myList->RemoveLast]

[String: $myList]
```

➜ List: Uno, Dos, Tres, Quatro

To inspect all the values in a list:

When a list is cast to string all of the values within it can be inspected. This output is intended to make it easy to debug list operations. The following code outputs the list that is created above. The elements are printed out in the order they were inserted.

```
[String: $myList]
```

➜ List: Uno, Dos, Tres, Quatro

Maps

Maps store and retrieve values based on a key. This allows for specific values to be stored under a name and then retrieved later using that same name. The name or key is always a string value.

Maps can only store one value per key. When a new value with the same key is inserted into a map it replaces the previous value which was stored in the map. If you need to create a data structure that stores more than one value per key, use an array of pairs instead.

Note: The order of elements in a map is not defined. As more elements are added to a map the order may change and should never be relied upon.

Table 6: Map Tag

Tag	Description
[Map]	Creates a map that contains each of the name/value parameters of the tag. If no parameters are specified, an empty map is created.

To create a map:

- The following example creates an empty map and stores it in a variable.

```
[Variable: 'EmptyMap' = (Map)]
```

- The following example shows a map with data stored using string literals as keys. The map is similar to a database record storing information about a particular site visitor.

```
[Map: 'First_Name'='John', 'Last_Name'='Doe', 'Phone_Number'='800-555-1212']
```

- The following example shows a map with integer literals as keys. This map could be used to lookup the name of a day of the week based on its order within the week.

```
[Map: 1='Sunday',
  2='Monday',
  3='Tuesday',
  4='Wednesday',
  5='Thursday',
  6='Friday',
  7='Saturday']
```

- The following example shows a map which contains arrays that are retrieved using string literals as keys. The map contains two arrays which are named Array_One and Array_Two.

```
[Map: 'Array_One' = (Array: 1, 2, 3, 4, 5),
  'Array_Two' = (Array: 9, 8, 7, 6, 5)]
```

Map Member Tags

The map data type has a number of member tags that can be used to store, retrieve or delete map elements by key.

Table 7: Map Member Tags

Tag	Description
[Map->Find]	Returns a value from the map by key. Accepts a single parameter which is the key of the value to be returned.
[Map->Get]	Returns a pair from the map by integer position. Accepts a single parameter which is the position of the value to be returned.
[Map->Keys]	Returns an array of all the keys specified in the map.
[Map->Insert]	Inserts a value into the map by key. Accepts a single name/value pair parameter which specifies the key and value to be inserted.
[Map->Remove]	Removes a value from the map by key. Accepts a single parameter which is the key of the value to be deleted.
[Map->Size]	Returns the number of elements (keys) in the map.
[Map->Values]	Returns an array of all the values specified in the map.

The following examples show how to manipulate a map by inserting, removing, and displaying elements. The examples are all based on the following array which contains the seven days of the week in English each with an integer key corresponding to their calendar order.

```
[Variable: 'DaysOfWeek' = (Map: 1='Sunday',
  2='Monday',
  3='Tuesday',
  4='Wednesday',
  5='Thursday',
  6='Friday',
  7='Saturday')]
```

To get values from a map:

- The value for a given key within the map can be retrieved using the [Map->Find] tag. The tag accepts a single parameter which is the key of the value to be returned. The key can be any value in Lasso. In the following example the numeric keys in the DaysOfWeek variable are used to return several days of the week.

  ```
  [$DaysOfWeek->(Find: 2)] ➜ Monday
  [$DaysOfWeek->(Find: 4)] ➜ Wednesday
  [$DaysOfWeek->(Find: 6)] ➜ Friday
  ```

- All of the keys used within a map can be displayed using the [Map->Keys] tag. In the following example, the integer keys of the DaysOfWeek map are displayed.

  ```
  [Output $DaysOfWeek->Keys]

  ➜ (Array: (1), (2), (3), (4), (5), (6), (7))
  ```

- All of the values used within a map can be displayed using the [Map->Values] tag. In the following example, the string values of the DaysOfWeek map are displayed.

  ```
  [Output $DaysOfWeek->Values]

  ➜ (Array: (Sunday), (Monday), (Tuesday), (Wednesday), (Thursday), (Friday),
    (Saturday))
  ```

- All of the elements in a map can be displayed using the [Iterate] ... [/Iterate] tags. In the following example, a temporary variable TempElement is set to the value of each element of the map in turn. The [Pair->First] and [Pair->Second] parts of each element are displayed.

  ```
  [Iterate: $DaysOfWeek, (Variable: 'TempElement')]
    <br>[$TempElement->First] = [$TempElement->Second]
  [/Iterate]
  ```

```
➜ <br>1 = Sunday
  <br>2 = Monday
  <br>3 = Tuesday
  <br>4 = Wednesday
  <br>5 = Thursday
  <br>6 = Friday
  <br>7 = Saturday
```

- Alternately, all of the elements in a map can be displayed using the [Loop] ... [/Loop] tags. In the following example, the [Map->Size] tag is used to return the size of the map and the [Map->Get] tag is used to return a particular element of the map. These tags function exactly like the same tags used on a pair array. A temporary variable TempElement is used to make the code easier to read.

```
[Loop: ($DaysOfWeek->Size)]
  [Variable: 'TempElement' = ($DaysOfWeek->(Get: (Loop_Count)))]
  <br>[$TempElement->First] = [$TempElement->Second]
[/Loop]
```

```
➜ <br>1 = Sunday
  <br>2 = Monday
  <br>3 = Tuesday
  <br>4 = Wednesday
  <br>5 = Thursday
  <br>6 = Friday
  <br>7 = Saturday
```

Note: Map elements cannot be set using the [Map->Get] member tag. Instead, map elements should be inserted using the [Map->Insert] member tag with the same key value as an element in the map.

To insert values into a map:

Elements can be added to the map or the value for a given key can be changed within a map using the [Map->Insert] tag.

- Use the [Map->Insert] tag with a name/value parameter to insert a new value into a map. The following example shows how to add an Extra Saturday to the map stored in DaysOfWeek. No value is returned by the [Map->Insert] tag, but the new value for key 8 is retrieved using [Map->Find] to show that the new element has been added.

```
<?LassoScript
  $DaysOfWeek->(Insert: 8='Extra Saturday');
  $DaysOfWeek->(Find: 8);
?>
```

```
➜ Extra Saturday
```

- Use the [Map->Insert] tag with the name of a value already stored in the map to replace that value within the map. The following example shows how to change the value for key 8 to Extra Sabado, substituting the Spanish word for Saturday. No value is returned by the [Map->Insert] tag, but the new value for key 8 is retrieved using [Map->Find] to show that the element has been modified.

```
<?LassoScript
  $DaysOfWeek->(Insert: 8='Extra Sabado');
  $DaysOfWeek->(Find: 8);
?>
```

```
➜ Extra Sabado
```

To remove values from a map:

The value for a key can be removed from a map using the [Map->Remove] tag. The tag accepts a single parameter, the name of the element to be removed. In the following example, the Extra Sabado entry is removed from the map stored in DaysOfWeek.

```
<?LassoScript
  $DaysOfWeek->(Remove: 8);
?>
```

To display the elements of a map:

For debugging purposes all of the elements of a map can be output simply by displaying the value of the variable holding the map. This is a quick way to see the value stored in a map, but is not intended to be used to show to site visitors.

```
[Variable: 'DaysOfWeek']
```

```
➜ (Map: (1)=(Sunday),
    (2)=(Monday),
    (3)=(Tuesday),
    (4)=(Wednesday),
    (5)=(Thursday),
    (6)=(Friday),
    (7)=(Saturday))
```

Maps vs Pair Arrays

Maps and pair arrays can both be used to store data which is retrieved by name. Maps store a single value per name. Pair arrays can store many different values for each name. Maps do not maintain the order of elements contained within them. Pair arrays do maintain the order of elements, though they generally cannot be sorted. Maps contain only

values associated with names (although the names and values can be of any data type). Arrays can contain a combination of pairs and other data types.

Maps should be used when the set of keys by which data will be retrieved is unique. Maps can be used as an equivalent for database records. Maps provide fast lookup of a value associated with a key.

Pair arrays should be used when multiple values need to be stored with each key or when the order of elements stored in the array is important. Pair arrays are used to return name/value parameters from Lasso actions or within custom tags.

Pairs

A pair is a compound data type that stores two elements. Pairs are most commonly used when working with lists of command tags and name/value parameters in concert with the [Action_Params] tag or when parsing parameters of a custom tag using the [Params] tag.

Table 8: Pair Tag

Tag	Description
[Pair]	Creates a pair with the specified name and value as the first and second elements.

To create a pair:

- The following example shows how to create a pair using the [Pair] tag. The tag accepts a single name/value parameter. The name part of the parameter becomes the First part of the pair. The value part of the parameter becomes the Second part of the pair.

```
[Pair: 'First_Name'='John']
```

- Pairs can be created in an [Array] constructor tag by specifying a name/value parameter as one of the parameters for the new array. The following example shows how to create an array with three pair elements.

```
[Array: 'First_Name'='John', 'Last_Name'='Doe', 'Phone_Number'='800-555-1212']
```

- Pairs are also created in a [Map] constructor tag by specifying name/value parameters. The following example shows how to create a map with three pairs.

```
[Map: 'First_Name'='John', 'Last_Name'='Doe', 'Phone_Number'='800-555-1212']
```

Pair Member Tags

The pair data type has two member tags that can be used to change or retrieve the two elements of the pair data type.

Table 9: Pair Member Tags

Tag	Description
[Pair->First]	Returns the first element of the pair. Can be used as the left parameter of an assignment operator to change the first element of the pair.
[Pair->Second]	Returns the second element of the pair. Can be used as the left parameter of an assignment operator to change the second element of the pair.

Note: For compatibility with maps and arrays the [Pair->Size] tag always returns 2 and [Pair->(Get:1)] and [Pair->(Get:2)] work to extract the first and second elements from a pair.

To get the elements of a pair:

The following example shows how to create a pair using a name/value parameter and then return the First and Second elements of the pair.

```
[Variable: 'Test_Pair' = (Pair: 'First_Name'='John')]
[$Test_Pair->First]: [$Test_Pair->Second]
```

➜ First_Name: John

To set the elements of a pair:

The following example shows how to set the first and second elements of a pair to new values using the assignment operator =. The altered pair is then displayed.

```
<?LassoScript
  Variable: 'Test_Pair' = (Pair: 'First_Name'='John');
  $Test_Pair->First = 'Last_Name';
  $Test_Pair->Second = 'Doe';
  $Test_Pair->First + ': ' + $Test_Pair->Second;
```

➜ Last_Name: Doe

To display the elements of a pair:

For debugging purposes the elements of a pair can be displayed simply by outputting the variable which contains the pair. The following example shows how to output a pair stored in a variable Test_Pair.

```
[Variable: 'Test_Pair' = (Pair: 'First_Name'='John')]
[Variable: 'Test_Pair']
➜ (Pair: (First_Name)=(John))
```

Priority Queues

A priority queue is a compound data type that stores elements in a sorted queue. When a priority queue is created a comparator can be specified which will be used to sort all of the elements of the queue. By default elements will be sorted alphabetically or numerically and the greatest element will be returned first.

When an element is inserted into a priority queue it is automatically placed in the proper position based on its value in comparison to the elements already within the queue. Only the first or greatest value of the queue can be retrieved.

Priority queues are always created empty. Elements can then be added using the [PriorityQueue->Insert] tag.

Note: priority queues pull their next value off the end of the list of contained elements. Using the \Compare_LessThan comparator will result in the greatest element being returned first. Using the \Compare_GreaterThan comparator will result in the least element being returned first. Custom comparators will need to take this behavior into account in order to get the expected results.

Table 10: Priority Queue Tag

Tag	Description
[PriorityQueue]	Creates a priority queue. Accepts an optional parameter which specifies a comparator that will be used to sort elements as they are inserted into the queue.

To create a priority queue:

Priority queues are created empty using the priority queue tag. The [PriorityQueue] tag has one optional parameter which specifies a comparator that will be used to sort the elements within the queue. Once the priority queue has been created, elements can be added using the [PriorityQueue->Insert] tag.

- For example, the following code creates a priority queue using the default comparator (which sorts elements alphabetically or numerically and returns the greatest value) and then inserts One and Two into it.

```
<?LassoScript
  Var: 'myPriorityQueue' = (PriorityQueue);
  $myPriorityQueue->Insert('One');
  $myPriorityQueue->Insert('Two');
?>
```

Note that the elements are stored in the queue in alphabetical order. [PriorityQueue->First] would return Two after the code above is run since Two is alphabetically greater than One.

- A priority queue can be created which sorts elements in the reverse order, always returning the lowest value when compared alphabetically or numerically, by specifying the \Compare_GreaterThan comparator when creating the queue.

```
<?LassoScript
  Var: 'myPriorityQueue' = (PriorityQueue: \Compare_GreaterThan);
  $myPriorityQueue->Insert('One');
  $myPriorityQueue->Insert('Two');
?>
```

Now the [PriorityQueue->First] tag would return One since the values within the queue will be sorted in reverse alphabetical order.

Priority Queue Member Tags

The priority queue data type has a number of member tags that can be used to store, retrieve or delete array elements or to otherwise manipulate array values.

Table 11: Priority Queue Member Tags

Tag	Description
[PriorityQueue->First]	Returns the first element of the priority queue. This is the maximum value according to the comparator specified when the priority queue was created.
[PriorityQueue->Get]	Returns the first element of the priority queue and removes it from the queue. Repeated calls to this tag will consume the data type, eventually leaving an empty type.
[PriorityQueue->Insert]	Inserts a value into the priority queue. Accepts a single parameter which is the value to be inserted. The element is always inserted according to the comparator specified when the priority queue was created. [PriorityQueue->InsertLast] is an alias.

[PriorityQueue->Remove]	Removes the first element from the priority queue. Returns no value. [PriorityQueue->RemoveFirst] is an alias.
[PriorityQueue->Size]	Returns the number of elements in the priority queue.

To inspect the first element in the priority queue:

Use the [PriorityQueue->First] tag. This tag will return the value of the next element on the priority queue (the greatest value according to the comparator which was specified when the queue was created) without modifying the queue in any way. The values One and Two are pushed onto the queue then the value Two is inspected.

```
<?LassoScript
  Var: 'myPriorityQueue' = (PriorityQueue);
  $myPriorityQueue->Insert('One');
  $myPriorityQueue->Insert('Two');

  $myPriorityQueue->First;
?>
```

➜ Two

To return the number of elements in the priority queue:

Use the [PriorityQueue->Size] tag. This tag returns an integer representing the number of elements that are contained within the priority queue. The following code outputs the size of the queue created above.

```
[$myPriorityQueue->Size]
```

➜ 2

To inspect all the values in the priority queue:

When a priority queue is cast to string all of the values within it can be inspected. This output is intended to make it easy to debug queue operations. The following code outputs the queue that is created above.

```
[String: $myPriorityQueue]
```

➜ PriorityQueue: One, Two

Note: The elements will not necessarily be printed out in the order that they will be returned by the member tags of the priority queue type.

To remove an element from the priority queue:

There are two methods to remove an element from the queue depending on whether the value is needed for further processing or should simply be discarded.

- The [PriorityQueue->Get] tag removes the first element from the queue and returns its value. The following code gets the value of the first element of the queue and then returns the size of the queue showing the value has been removed.

```
<?LassoScript
  Var: 'myPriorityQueue' = (PriorityQueue);
  $myPriorityQueue->Insert('One');
  $myPriorityQueue->Insert('Two');

  $myPriorityQueue->Get;

  $myPriorityQueue->Size;
?>
```

```
➜ Two
  1
```

- The [PriorityQueue->Remove] tag removes the first element from the queue, but does not return its value. The following code removes the first element from the queue and then returns its size. Notice that the [PriorityQueue->Remove] tag does not return any value so only the size is output.

```
<?LassoScript
  Var: 'myPriorityQueue' = (PriorityQueue);
  $myPriorityQueue->Insert('One');
  $myPriorityQueue->Insert('Two');

  $myPriorityQueue->Remove;

  $myPriorityQueue->Size;
?>
```

```
➜ 1
```

To perform an operation on each element of a priority queue:

The [Iterate] ... [/Iterate] tags can be used to get each element of the queue in turn. The [Iterate] ... [/Iterate] tags use the [PriorityQueue->Size] and [PriorityQueue->Get] tags in order to cycle through every element in the queue. At the end of the loop the queue will be empty.

The following example creates a queue and then uses [Iterate] ... [/Iterate] tags to print out each value in the queue. Note that the values in the queue are returned in sorted order.

```
<?LassoScript
  Var: 'myPriorityQueue' = (PriorityQueue);
  $myPriorityQueue->Insert('One');
  $myPriorityQueue->Insert('Two');

  Iterate: $myPriorityQueue, (Var: 'myItem');
    '<br />' + $myItem;
  /Iterate;

?>

➜ <br />Two
  <br />One
```

Queues

A queue is a compound data type that stores elements in "first in, first out" or FIFO order. Elements are pushed onto the end of the queue using [Queue->Insert]. Elements can be popped off the front of the queue using [Queue->Get]. Only the earliest inserted element of a queue can be retrieved or inspected.

Queues can be used when a series of values need to be kept track of and processed in order. Queues are always created empty. Elements can then be added using the [Queue->Insert] tag.

Table 12: Queue Tag

Tag	Description
[Queue]	Creates an empty queue.

To create a queue:

Queues are created empty using the queue tag. Elements can be added using the [Queue->Insert] tag. For example, the following code creates a queue and then inserts One and Two into it.

```
<?LassoScript
  Var: 'myQueue' = (Queue);
  $myQueue->Insert('One');
  $myQueue->Insert('Two');
?>
```

Note that the elements that are stored in the queue will be retrieved in the same order they are inserted. [$myQueue->First] will return One after the above code is run.

Queue Member Tags

The queue data type has a number of member tags that can be used to store, retrieve or delete queue elements.

Table 13: Queue Member Tags

Tag	Description
[Queue->First]	Returns the first element of the queue. This is always the earliest value inserted into the queue. The queue is not modified by this tag.
[Queue->Get]	Returns the first element of the queue and removes it from the queue. Repeated calls to this tag will consume the data type, eventually leaving an empty type.
[Queue->Insert]	Inserts a value onto the end of the queue. Accepts a single parameter which is the value to be inserted. [Queue->InsertLast] is an alias.
[Queue->Remove]	Removes the first element from the queue. Returns no value. [Queue->RemoveFirst] is an alias.
[Queue->Size]	Returns the number of elements in the queue.

To inspect the first element on the queue:

Use the [Queue->First] tag. This tag will return the value of the next element on the queue (the earliest inserted value) without modifying the queue in any way. The values One and Two are pushed into the queue then the value One is inspected.

```
<?LassoScript
  Var: 'myQueue' = (Queue);
  $myQueue->Insert('One');
  $myQueue->Insert('Two');

  $myQueue->First;
?>
```

➜ One

To return the number of elements in the queue:

Use the [Queue->Size] tag. This tag returns an integer representing the number of elements that are contained within the queue. The following code outputs the size of the queue created above.

```
[$myQueue->Size]
```

➜ 2

To inspect all the values in the queue:

When a queue is cast to string all of the values within it can be inspected. This output is intended to make it easy to debug queue operations. The following code outputs the queue that is created above. The elements are printed out in the order they were inserted.

```
[String: $myQueue]
```

```
➜ Queue: One, Two
```

To modify the first element on the queue:

The [Queue->First] tag returns the first element of the queue by reference so the value of the element can be changed. It is not possible to change the values of any other elements on the queue. In the folowing example the first element in the queue is modified to have the value Three and then this value is returned.

```
<?LassoScript
  Var: 'myQueue' = (Queue);
  $myQueue->Insert('One');
  $myQueue->Insert('Two');

  $myQueue->First = 'Three';

  $myQueue->First;
?>
```

```
➜ Three
```

To remove an element from the queue:

There are two methods to remove an element from the queue depending on whether the value is needed for further processing or should simply be discarded.

- The [Queue->Get] tag removes the first element from the queue and returns its value. The following code gets the value of the first element of the queue and then returns the size of the queue showing the value has been removed.

```
<?LassoScript
  Var: 'myQueue' = (Queue);
  $myQueue->Insert('One');
  $myQueue->Insert('Two');

  $myQueue->Get;

  $myQueue->Size;
?>
```

```
➜ One
  1
```

- The [Queue->Remove] tag removes the first element from the queue, but does not return its value. The following code removes the first element from the queue and then returns its size. Notice that the [Queue->Remove] tag does not return any value so only the size is output.

```
<?LassoScript
  Var: 'myQueue' = (Queue);
  $myQueue->Insert('One');
  $myQueue->Insert('Two');

  $myQueue->Remove;

  $myQueue->Size;
?>
```

➜ 1

To perform an operation on each element of a queue:

The [Iterate] ... [/Iterate] tags can be used to get each element of a queue in turn. The [Iterate] ... [/Iterate] tags use the [Queue->Size] and [Queue->Get] tags in order to cycle through every element in the queue. At the end of the loop the queue will be empty.

The following example creates a queue and then uses [Iterate] ... [/Iterate] tags to print out each value in the quue. Note that the values in the queue are returned in the same order as they were inserted.

```
<?LassoScript
  Var: 'myQueue' = (Queue);
  $myQueue->Insert('One');
  $myQueue->Insert('Two');

  Iterate: $myQueue, (Var: 'myItem');
    '<br />' + $myItem;
  /Iterate;

?>
```

➜
One

Two

Series

A series is a simple data type that represents a sequence of values. When a series is created it requires a start and an end value. The start value is incremented until it equals the end value and all of the intervening values are placed in an array. Series are usually created using integer or decimal values, but can be created using any data type that supports the ++ symbol.

Table 14: Series Tag

Tag	Description
[Series]	Creates a new series. The tag accepts two parameters which are the start and end value for the range.

To create a series:

Series are created using the [Series] tag. The tag accepts two parameters which define the start and end of the series. For example, the following code creates a series with values from 1 to 10.

```
[Series(1, 10)]
```

```
➜ Series: (1), (2), (3), (4), (5), (6), (7), (8), (9), (10)
```

The series data type supports the same member tags as the array data type. See the array examples for details about how to use those member tags.

Sets

A set is a collection of unique values. The elements within a set are always sorted. When a new element is inserted the set is first checked to see if the value is already contained. Sets can be manipulated using the [Set->Difference], [Set->Intersection], and [Set->Union] tags. The values stored in a set can be of any data type in LDML.

Table 15: Set Tag

Tag	Description
[Set]	Creates a new set. The tag accepts one parameter which is a comparator that will be used to order the elements within the set. By default the comparator sorts the elements alphanumerically..

To create a set:

Sets are created using the [Set] tag. The tag accepts an optional parameter which defines the sort order for the set. Set elements can be added using the [Set->Insert] tag. For example, the following code creates a set with the default comparator and then inserts One and Three into it. The multiple inserts of Three are ignored since the set can only ontain unique values.

```
<?LassoScript
  Var: 'mySet' = (Set);
  $mySet->Insert('One');
  $mySet->Insert('Three');
  $mySet->Insert('Three');
  $mySet->Insert('Three');

  $mySet;
?>
```

➜ Set: (One, Three)

Set Member Tags

The set data type has a number of member tags that can be used to store, retrieve or delete set elements or to otherwise manipulate set values.

Table 16: Set Member Tags

Tag	Description
[Set->Contains]	Returns True if the specified element is contained in the set. Requires a single parameter which is a value to compare to each element of the set.
[Set->Difference]	Compares the set against another returning a new set that contains only elements of the current set which are not contained in the other set. Requires one parameter which is a set to be compared.
[Set->Find]	Returns a set of elements that match the parameter. Accepts a single parameter of any data type.
[Set->ForEach]	Applies a tag to each element of the set in turn. Requires a single parameter which is a reference to a tag or compound data type. Modifies the set in place and returns no value.
[Set->Get]	Returns an item from the set. Accepts a single integer parameter identifying the position of the item to be returned. This tag can be used as the left parameter of an assignment operator to set an element of the set.
[Set->Insert]	Inserts a value into the set. Accepts a single parameter which is the value to be inserted. Returns no value.

[Set->InsertFrom]	Inserts elements from an iterator. Requires a single parameter which is an iterator from another compound data type.
[Set->Intersection]	Compares the set against another returning a new set that contains only elements contained in both sets. Requires one parameter which is a set to be compared.
[Set->Iterator]	Returns an iterator to step through every element of the set. An optional parameter specifies a comparator which selects which elements of the set to return.
[Set->Join]	Joins the items of the set into a string. Accepts a single string parameter which is placed inbetween each item from the array. The opposite of [String->Split].
[Set->Remove]	Removes an item from the set. Accepts a single integer parameter identifying the position of the item to be removed. Defaults to the last item in the set. Returns no value.
[Set->RemoveAll]	Removes any elements that match the parameter from the set. Accepts a single parameter of any data type. Returns no value.
[Set->ReverseIterator]	The same as iterator, but returns the elements in the reverse order. See iterator for more details.
[Set->Size]	Returns the number of elements in the set.
[Set->Union]	Returns a new sets that contains all of the elements from two sets without duplicates. Requires one parameter which is a set to be added to the current set.

To manipulate sets using difference, intersection, and union:

The [Set->Difference], [Set->Intersection], and [Set->Union] tags can be used to manipulate sets. Since sets contain only unique values the result of each of these tags will be a set that contains only the unique values that are the result of the operation.

Each of these examples uses the following two sets.

```
[Var: 'FirstSet' = (Set)]
[$FirstSet->(Insert: 'Alpha')]
[$FirstSet->(Insert: 'Beta')]
[$FirstSet->(Insert: 'Gamma')]

[Var: 'SecondSet' = (Set)]
[$SecondSet->(Insert: 'Beta')]
[$SecondSet->(Insert: 'Gamma')]
[$SecondSet->(Insert: 'Delta')]
```

- **Difference** – The difference between two sets is a set of unique values which are contained in the base set, but are not contained in the parameter set. For example, the following code duplicates FirstSet as ResultSet and then calculates the difference from SecondSet. The result is only those elements from FirstSet that are not contained in SecondSet.

```
[Var: 'ResultSet' = $FirstSet]
[$ResultSet->(Difference: $SecondSet)]
[$ResultSet]
```

➜ Set: (Alpha)

- **Intersection** – The intersection of two sets is a set of unique values which are contained in both the base set and the parameter set. For example, the following code duplicates FirstSet as ResultSet and then calculates the intersection with SecondSet. The result is only those elements from FirstSet that are also contained in SecondSet.

```
[Var: 'ResultSet' = $FirstSet]
[$ResultSet->(Intersection: $SecondSet)]
[$ResultSet]
```

➜ Set: (Beta), (Gamma))

- **Union** – The union between two sets is a set of unique values which are contained in either the base set or the parameter set. For example, the following code duplicates FirstSet as ResultSet and then calculates the union with SecondSet. The result is every elements from FirstSet and every element from SecondSet.

```
[Var: 'ResultSet' = $FirstSet]
[$ResultSet->(Union: $SecondSet)]
[$ResultSet]
```

➜ Set: (Alpha), (Beta), (Gamma), (Delta)

Stacks

A stack is a compound data type that stores elements in "last in, first out" or LIFO order. Elements are pushed onto the stack using [Stack->Insert]. Elements can be popped off the stack using [Stack->Get]. Only the most recently inserted element of a stack can be retrieved or inspected.

Stacks are frequently used to keep track of the current state of an ongoing process. For example, when processing a directory hierarchy a stack will often be used to record the current directory that is being processed.

Stacks are always created empty. Elements can then be added using the [Stack->Insert] tag.

Table 17: Stack Tag

Tag	Description
[Stack]	Creates an empty stack.

To create a stack:

Stacks are created empty using the stack tag. Elements can be added using the [Stack->Insert] tag. For example, the following code creates a stack and then inserts One and Two into it.

```
<?LassoScript
  Var: 'myStack' = (Stack);
  $myStack->Insert('One');
  $myStack->Insert('Two');
?>
```

Note that the elements that are stored in the stack will be retrieved in reverse order. [$myStack->First] will return Two after the above code is run.

Stack Member Tags

The stack data type has a number of member tags that can be used to push, pop, or inspect the elements on the stack.

Table 18: Stack Member Tags

Tag	Description
[Stack->First]	Returns the first element of the stack. This is always the most recent value inserted into the stack. The stack is not modified by this tag.
[Stack->Get]	Returns the first element of the stack and removes it from the stack. Equivalent to a pop operation. Repeated calls to this tag will consume the data type, eventually leaving an empty type.
[Stack->Insert]	Inserts a value into the stack. Accepts a single parameter which is the value to be inserted. Equivalent to a push operation. [Stack->InsertFirst] is an alias.
[Stack->Remove]	Removes the first element from the stack. Returns no value. [Stack->RemoveFirst] is an alias.
[Stack->Size]	Returns the number of elements in the stack.

To inspect the first element on the stack:

Use the [Stack->First] tag. This tag will return the value of the next element on the stack (the most recently inserted value) without modifying the stack in any way. The values One and Two are pushed onto the stack then the value Two is inspected.

```
<?LassoScript
  Var: 'myStack' = (Stack);
  $myStack->Insert('One');
  $myStack->Insert('Two');

  $myStack->First;
?>
```

➜ Two

To return the number of elements on the stack:

Use the [Stack->Size] tag. This tag returns an integer representing the number of elements that are contained within the stack. The following code outputs the size of the stack created above.

```
[$myStack->Size]
```

➜ 2

To inspect all the values on the stack:

When a stack is cast to string all of the values within it can be inspected. This output is intended to make it easy to debug stack operations. The following code outputs the stack that is created above. The elements are printed out in the order they were inserted.

```
[String: $myStack]
```

➜ Stack: One, Two

To modify the first element on the stack:

The [Stack->First] tag returns the first element of the stack by reference so the value of the element can be changed. It is not possible to change the values of any other elements on the stack. In the folowing example the first element on the stack is modified to have the value Three and then this value is returned.

```
<?LassoScript
  Var: 'myStack' = (Stack);
  $myStack->Insert('One');
  $myStack->Insert('Two');

  $myStack->First = 'Three';

  $myStack->First;
?>
```

➜ Three

To remove an element from the stack:

There are two methods to remove an element from the stack depending on whether the value is needed for further processing or should simply be discarded.

- The [Stack->Get] tag removes the first element from the stack and returns its value. The following code gets the value of the first element of the stack and then returns the size of the stack showing the value has been removed.

```
<?LassoScript
  Var: 'myStack' = (Stack);
  $myStack->Insert('One');
  $myStack->Insert('Two');

  $myStack->Get;

  $myStack->Size;
?>
```

➜ Two
1

- The [Stack->Remove] tag removes the first element from the stack, but does not return its value. The following code removes the first element from the stack and then returns its size. Notice that the [Stack->Remove] tag does not return any value so only the size is output.

```
<?LassoScript
  Var: 'myStack' = (Stack);
  $myStack->Insert('One');
  $myStack->Insert('Two');

  $myStack->Remove;

  $myStack->Size;
?>
```

➜ 1

To perform an operation on each element of a stack:

The [Iterate] ... [/Iterate] tags can be used to pop each element of the stack off in turn. The [Iterate] ... [/Iterate] tags use the [Stack->Size] and [Stack->Get] tags in order to cycle through every element in the stack. At the end of the loop the stack will be empty.

The following example creates a stack and then uses [Iterate] ... [/Iterate] tags to print out each value in the stack. Note that the values in the stack are returned in the reverse order from how they were inserted.

```
<?LassoScript
  Var: 'myStack' = (Stack);
  $myStack->Insert('One');
  $myStack->Insert('Two');

  Iterate: $myStack, (Var: 'myItem');
    '<br />' + $myItem;
  /Iterate;

?>

➜ <br />Two
<br />One
```

Tree Maps

Tree maps store and retrieve values based on a key. This allows for specific values to be stored under a name and then retrieved later using that same name.

Tree maps differ from maps in two respects. The keys in a tree map can be any Lasso data type. In a simple map all keys are convereted to string values. Second, the keys in a tree map can be sorted using a comparator which is provided when the tree map is created.

Tree maps can only store one value per key. When a new value with the same key is inserted into a map it replaces the previous value which was stored in the map. In order to create a data structure that stores more than one value per key, use an array of pairs instead.

Table 19: Tree Map Tag

Tag	Description
[TreeMap]	Creates a tree map that contains each of the name/value parameters of the tag. If no parameters are specified, an empty map is created.

To create a tree map:

- The following example creates an empty tree map and stores it in a variable.

```
[Variable: 'EmptyMap' = (TreeMap)]
```

- The following example shows a tree map with data stored using string literals as keys. The map is similar to a database record storing information about a particular site visitor.

```
[TreeMap: 'First_Name'='John', 'Last_Name'='Doe']
```

- The following example shows a tree map which contains elements that have an array as both the key and value.

```
[Map: (Array: 1, 5) = (Array: 1, 2, 3, 4, 5),
  (Array: 9, 5) = (Array: 9, 8, 7, 6, 5)]
```

Tree Map Member Tags

The map data type has a number of member tags that can be used to store, retrieve or delete map elements by key.

Table 20: Tree Map Member Tags

Tag	Description
[TreeMap->Find]	Returns a value from the tree map by key. Accepts a single parameter which is the key of the value to be returned.
[TreeMap->Get]	Returns a pair from the tree map by integer position. Accepts a single parameter which is the position of the value to be returned.
[TreeMap->Keys]	Returns an array of all the keys specified in the tree map.
[TreeMap->Insert]	Inserts a value into the tree map by key. Accepts a single name/value pair parameter which specifies the key and value to be inserted.
[TreeMap->Iterator]	Returns an iterator that can be used to cycle through all the elements in the tree map. An optional parameter can be used to only match certain keys within the tree map.
[TreeMap->Remove]	Removes a value from the tree map by key. Accepts a single parameter which is the key of the value to be deleted.
[TreeMap->RemoveAll]	Removes all matching values from the tree map. Requires one parameter which is the value to compare to each key of the map.
[TreeMap->Size]	Returns the number of elements in the tree map.

[TreeMap->Values]	Returns an array of all the values specified in the tree map.

The following examples show how to manipulate a tree map by inserting, removing, and displaying elements. The examples are all based on the following array which contains the seven days of the week in English each with an integer key corresponding to their calendar order.

```
[Variable: 'DaysOfWeek' = (TreeMap: 1='Sunday',
  2='Monday',
  3='Tuesday',
  4='Wednesday',
  5='Thursday',
  6='Friday',
  7='Saturday')]
```

To get values from a tree map:

- The value for a given key within the map can be retrieved using the [TreeMap->Find] tag. The tag accepts a single parameter which is the key of the value to be returned. The key can be any value in Lasso. In the following example the numeric keys in the DaysOfWeek variable are used to return several days of the week.

  ```
  [$DaysOfWeek->(Find: 2)] ➜ Monday
  [$DaysOfWeek->(Find: 4)] ➜ Wednesday
  [$DaysOfWeek->(Find: 6)] ➜ Friday
  ```

- All of the keys used within a tree map can be displayed using the [TreeMap->Keys] tag. In the following example, the integer keys of the DaysOfWeek map are displayed.

  ```
  [Output $DaysOfWeek->Keys]

  ➜ (Array: (1), (2), (3), (4), (5), (6), (7))
  ```

- All of the values within a tree map can be displayed using the [TreeMap->Values] tag. In the following example, the string values of the DaysOfWeek map are displayed.

  ```
  [Output $DaysOfWeek->Values]

  ➜ (Array: (Sunday), (Monday), (Tuesday), (Wednesday), (Thursday), (Friday),
    (Saturday))
  ```

- All of the elements in a tree map can be displayed using the [Iterate] ... [/Iterate] tags. In the following example, a temporary variable TempElement is set to the value of each element of the map in turn. The [Pair->First] and [Pair->Second] parts of each element are displayed.

```
[Iterate: $DaysOfWeek, (Variable: 'TempElement')]
  <br>[$TempElement->First] = [$TempElement->Second]
[/Iterate]
```

```
➜ <br>1 = Sunday
  <br>2 = Monday
  <br>3 = Tuesday
  <br>4 = Wednesday
  <br>5 = Thursday
  <br>6 = Friday
  <br>7 = Saturday
```

- Alternately, all of the elements in a tree map can be displayed using the [Loop] ... [/Loop] tags. In the following example, the [TreeMap->Size] tag is used to return the size of the map and the [TreeMap->Get] tag is used to return a particular element of the tree map. These tags function exactly like the same tags used on a map or pair array. A temporary variable TempElement is used to make the code easier to read.

```
[Loop: ($DaysOfWeek->Size)]
  [Variable: 'TempElement' = ($DaysOfWeek->(Get: (Loop_Count)))]
  <br>[$TempElement->First] = [$TempElement->Second]
[/Loop]
```

```
➜ <br>1 = Sunday
  <br>2 = Monday
  <br>3 = Tuesday
  <br>4 = Wednesday
  <br>5 = Thursday
  <br>6 = Friday
  <br>7 = Saturday
```

Note: Tree map elements cannot be set using the [TreeMap->Get] member tag. Instead, tree map elements should be inserted using the [TreeMap->Insert] member tag with the same key as an element in the map.

To insert values into a tree map:

Elements can be added to the tree map or the value for a given key can be changed within a tree map using the [TreeMap->Insert] tag.

- Use the [TreeMap->Insert] tag with a name/value parameter to insert a new value into a tree map. The following example shows how to add an Extra Saturday to thetree map stored in DaysOfWeek. No value is returned by the [TreeMap->Insert] tag, but the new value for key 8 is retrieved using [TreeMap->Find] to show that the new element has been added.

```
<?LassoScript
  $DaysOfWeek->(Insert: 8='Extra Saturday');
  $DaysOfWeek->(Find: 8);
?>
```

➜ Extra Saturday

- Use the [TreeMap->Insert] tag with the name of a value already stored in the map to replace that value within the tree map. The following example shows how to change the value for key 8 to Extra Sabado, substituting the Spanish word for Saturday. No value is returned by the [TreeMap->Insert] tag, but the new value for key 8 is retrieved using [TreeMap->Find] to show that the element has been modified.

```
<?LassoScript
  $DaysOfWeek->(Insert: 8='Extra Sabado');
  $DaysOfWeek->(Find: 8);
?>
```

➜ Extra Sabado

To remove values from a tree map:

The value for a key can be removed from a tree map using the [Map->Remove] tag. The tag accepts a single parameter, the key of the element to be removed. In the following example, the Extra Sabado entry is removed from the tree map stored in DaysOfWeek.

```
<?LassoScript
  $DaysOfWeek->(Remove: 8);
?>
```

To display the elements of a tree map:

For debugging purposes all of the elements of a tree map can be output simply by displaying the value of the variable holding the tree map. This is a quick way to see the value stored in a tree map, but is not intended to be used to show to site visitors.

```
[Variable: 'DaysOfWeek']
```

➜ (TreeMap: (1)=(Sunday),
(2)=(Monday),
(3)=(Tuesday),
(4)=(Wednesday),
(5)=(Thursday),
(6)=(Friday),
(7)=(Saturday))

Comparators

Comparators are used with the [Match_Comparator] matcher or to sort the elements within a compound data type. A comparator can be specified when a priority queue is created. Comparators can also be used with the [Array->SortWith] and [List->SortWith] tags to explicitly order the elements within those data types.

The default comparator for priority queues is LessThan. This comparator is equivalent to the less than < symbol. It will sort strings alphabetically and integers or decimals numerically. The largest value will be returned first from the priority queue.

Lasso provides a collection of comparators that perform the most common types of sorting and comparisons. The comparators are shown in the following table.

Table 21: Comparators

Tag	Description
\Compare_LessThan	Sorts the elements in alphabetical or numerical order with lower values first. The default for priority queues (returning the greatest value first).
\Compare_GreaterThan	Sorts the elements in alphabetical or numerical order with higher values first.
\Compare_Contains	Can be used with the matcher to return elements that contain the specified value.
\Compare_NotContains	Can be used with the matcher to return elements that do not contain the specified value.
\Compare_EqualTo	Can be used with the matcher to return elements that equal the specified value (with type conversion).
\Compare_NotEqualTo	Can be used with the matcher to return elements that do not equal the specified value (with type conversion).
\Compare_StrictEqualTo	Same as \Compare_EqualTo, but performs a strict comparison including type.
\Compare_StrictNotEqualTo	Same as \Compare_NotEqualTo, but performs a strict comparison including type.

Note: Comparators do not return True or False. Comparators generally return an integer value. A valid comparison is signaled by the return value of 0. Any other result signals that the comparison was not valid.

To sort an array using a comparator:

Arrays and other compound data types can be sorted using comparators in the ->SortWith member tag.

- An array can be sorted in ascending order using the \Compare_LessThan comparator.

```
[var('array' = array('aaa', 'bbb', 'ccc', 'aa', 'a', 'b','c','bb','cc'))]
[$array->SortWith(\Compare_LessThan)]
[Encode_HTML($array)]
```

➜ Array: (a), (aa), (aaa), (b), (bb), (bbb), (c), (cc), (ccc)

- An array can be sorted in descending order using the \Compare_GreaterThan comparator.

```
[var('array' = array('aaa', 'bbb', 'ccc', 'aa', 'a', 'b','c','bb','cc'))]
[$array->SortWith(\Compare_GreaterThan)]
[Encode_HTML($array)]
```

➜ Array: (aaa), (aa), (a), (bbb), (bb), (b), (ccc), (cc), (c)

Note: See the following section for examples of using comparators with the [Match_Comparator] matcher.

Custom Comparators

Custom comparators can be created as custom tags or as custom types by overriding the onCompare callback tag. More details are included in the chapter on custom types. The example below would compare the first element of arrays.

```
<?LassoScript
  Define_Tag: 'ex_Compare_First', -Required='Left', -Required='Right';
    If: #Left->First < #Left->Second;
      Return 0;
    /If;
    Return: -1;
  /Define_Tag;
?>
```

Matchers

Matchers are used with the [...->Iterator], [...->RemoveAll], and [...->ReverseIterator] tags to determine which elements of a compound data type should be operated on.

The most simple matchers are just strings or numeric values. For example the matcher 'Alpha' will match any string Alpha contained in an array or set, any value in a map or tree map with a key of Alpha, or any pair with a first part of Alpha.

Lasso also provides a collection of matchers that perform an operation on each element of the compound data type in order to determine whether the value should match or not. These matchers are shown in the following table.

Table 22: Matchers

Tag	Description
Literal String, Decimal, Integer	Any literal value is a matcher for that value. Automatic casting is performed just as it is for the == symbol. Only the first part of pairs or the key value for maps is compared.
[Match_RegExp]	Requires a single parameter which is a regular expression. If the regular expression matches part of a string value then a match is signaled.
[Match_NotRegExp]	The same as [Match_RegExp], but signals a match if the regular expression is not matched.
[Match_Range]	Requires two parameters: a low value and a high value. Signals a match if the compared value is equal to either end-value or within the specified range.
[Match_NotRange]	The same as [Match_Range], but signals a match if the value is not in the range.
[Match_Comparator]	Requires two parameters: a comparator and either an -RHS or an -LHS value. Signals a match if the comparator matches the value. The -RHS parameter should be used by default to compare the value to each element. The -LHS parameter can be used to instead compare each element to the value.

Note: Matchers do not return True or False. Matchers generally return an integer value. A match is signaled by the return value of 0. Any other result signals that a match did not occur.

To check whether an array contains a value using a matcher:

The ->Contains member tag of each compound data type and the contains symbol >> both accept a matcher as a parameter. They return True if the matcher matches one or more elements within the compound data type or False otherwise.

- The most basic matcher is any literal value. For example, the following code checks to see if an array of numbers contains a specific number.

  ```
  [(Array: 1, 2, 3, 4, 5, 6, 7) >> 7] ➜ True
  ```

- The [Match_Range] matcher allows the array to be checked to see if it contains a number within a specific range.

  ```
  [(Array: 1, 2, 3, 4, 5, 6, 7) >> (Match_Range: 1, 4)] ➜ True

  [(Array: 1, 2, 3, 4, 5, 6, 7) >> (Match_Range: 8, 10)] ➜ False
  ```

  The [Match_NotRange] matcher allows the array to be checked to see if it does not contains a number within a specific range.

- The [Match_RegExp] matcher allows the array to be checked to see if it contains a string that matches a regular expression. The first example returns True since the words in the array contain the letter o. The second example returns False since neither word contains the letter f.

  ```
  [(Array: 'one', 'two') >> (Match_RegExp: 'o')] ➜ True

  [(Array: 'one', 'two') >> (Match_RegExp: 'f')] ➜ False
  ```

  The [Match_NotRegExp] matcher allows the array to be checked to see if it does not contains a string that matches the specified regular expression.

- The [Match_Comparator] matcher can be used with any of the available comparators (or a custom comparator).

  Using the \Compare_LessThan comparator and a -RHS parameter of 5 the expression returns True if the array contains any element that is less than 5. Switching to the -LHS parameter returns False, checking if 5 is less than any element.

  ```
  [(Array: 1, 2, 3) >> (Match_Comparator: \Compare_LessThan, -RHS=5)] ➜ True

  [(Array: 1, 2, 3) >> (Match_Comparator: \Compare_LessThan, -LHS=5)] ➜ False
  ```

  Using the \Compare_EqualTo comparator and a -RHS parameter of 3 the expression returns True if the array contains any element that is equal to 3. Using the \Compare_StrictEqualTo comparator and a -RHS parameter of '3' (a string) the expression returns False since the array does not contain any element that is strictly equal to the string '3'.

  ```
  [(Array: 1, 2, 3) >> (Match_Comparator: \Compare_EqualTo, -RHS=3)] ➜ True

  [(Array: 1, 2, 3) >> (Match_Comparator: \Compare_StrictEqualTo, -RHS='3')] ➜
  False
  ```

To remove elements from an array using a matcher:

Multiple elements can be removed from an array simultaneously using a matcher. in the ->RemoveAll tag.

- A range of numbers can be removed from an array using the [Match_Range] matcher. The following example removes all numbers between 2 and 4 from an array and returns the result.

```
[Var('array' = Array(1, 2, 3, 4, 5, 6, 7))]
[$array->RemoveAll(Match_Range(2, 4))]
[Output_HTML($array)]
```

➜ Array: (1), (5), (6), (7)

- A set of strings that match a regular expression can be removed from an array using the [Match_RegExp] matcher. The following example removes strings that start with T from an array and returns the result.

```
[Var('array' = Array('Monday','Tuesday','Wednesday','Thursday','Friday'))]
[$array->RemoveAll(Match_RegExp('\\bT'))]
[Output_HTML($array)]
```

➜ Array: (Monday), (Wednesday), (Friday))

- The [Match_Comparator] matcher can be used to remove any elements that match any of the available comparators (or a custom comparator). Using the \Compare_LessThan comparator and a -RHS parameter of 5 the expression removes all of the elements less than 5 from the array.

```
[Var('array' = Array(1, 2, 3, 4, 5, 6, 7))]
[$array->RemoveAll(Match_Comparator(\Compare_LessThan, -RHS=5))]
[Output_HTML($array)]
```

➜ Array: (5), (6), (7)

Custom Matchers

Custom matchers can be created as custom tags or as custom types by overriding the onCompare callback tag. More details are included in the chapter on custom types. The example below would match against today's date.

```
<?LassoScript
  Define_Type: 'ex_Match_Today';
    Define_Tag: 'onCompare', -Required='Value';
      If: (Date: #Value)->(Format: '%D') == Date->(Format: '%D');
        Return 0;
      /If;
      Return: -1;
    /Define_Tag;
  /Define_Type;
?>
```

Iterators

Iterators allow each element within a compound data type to be explored in turn. An iterator is created using the [Iterator] or [ReverseIterator] tags or returned by the [...->Iterator] or [...->ReverseIterator] tags of the array, list, map, set, and tree map types. Custom types might also return iterators.

An iterator has three essential characteristics:

- It has a collection of elements which it can return in order. This could include all of the elements within a compound data type or a subset of the elements within a compound data type if a matcher is used.
- It has a location within the collection of elements. For the built-in compound data types the location is a position and the iterator can be moved forward or backward through the elements. Iterators may also support moving left and right or up and down through a multi-dimensional collection of elements.
- It has a value for the current element. The map types have both a key and a value. In addition, iterators can be used to remove the current value or to insert a new value in place of the current value.

Methodology

The basic methodology of an iterator uses the [While] ... [/While] tags to move through each element in the collection of elements. In th example below, the iterator is acquired using the [Array->Iterator] tag. The [Iterator->AtEnd] tag is checked in the while condition to see whether the iterator is at the end yet. Each time through the loop the iterator value is fetched using [Iterator->Value] and the iterator is advanced using [Iterator->Forward]. The [Null] tag suppresses the output from [Iterator->Forward].

```
<?LassoScript
  Var: 'myArray' = Array('Alpha','Beta','Gamma','Delta');
  Var: 'myIterator' = $myArray->Iterator;

  While: ($myIterator->atEnd == False);
    '<br />' + $myIterator->Value;
    Null: $myIterator->Forward;
  /While;
?>

➜ <br />Alpha
<br />Beta
<br />Gamma
<br />Delta
```

Iterators And Other Iterate Methods

Lasso provides several different methods of iterating through elements within a compound data type. Each of the different methods has benefits and drawbacks. Ultimately, the method that is used in a given solution depends on the needs of that solution.

- Most compound data types support the [...->Size] and [...->Get] tags. With the [Loop] ... [/Loop] tags, these tags represent the basic method of iterating through all the elements within a compound data type. The advantages of this method are backward compatibility and support across most compound data types. The drawback is that this method requires a lot of code compared with some of the other methods.
- Most compound data types also support the [Iterate] ... [/Iterate] tags. These tags actually use the [...->Size] and [...->Get] tags in their implementation and provide an easy way to access every element of the compound data type in turn. The advantages of this method are backward compatibility and support across most compound data types. The drawback of this method is that it always returns every element in the data type and it is difficult to remove elements from the data type while iterating.
- The map types also support the [...->Keys] tag which allows either of the above methods to be used to iterator through the key values for the data type. The value for each element in the map can then be fetched, modified, or deleted using the appropriate member tags.
- The iterators defined in this section can be used to iterator through most compound data types. Iterators have the advantage of being able to move forward or backward through the data type. Elements can be easily deleted or inserted into the data type while iterating. The iterator can be used with a matcher to select only certain elements within the data type. The disadvantage is that only certain compound data types support iterators.

  Finally, the iterators mechanism is very flexible. It provides convenient access to both the key and value in map data types. And, it supports movement in up to three dimensions for more complex custom data types. The iterator method is designed to be forward thinking so it can be used by third-party developers to implement advanced functionality.

Iterator Tags

Iterators can be obtained using the ->Iterator or ->ReverseIterator tags of compound data types (or custom data types) or by using the tags in the following table.

Table 23: Iterator Tags

Tag	Description
[Iterator]	Requires a compound data type as a parameter. Returns the iterator for the data type. A second optional parameter allows a matcher to be specified. Uses the built-in iterator if the type supports the ->Iterator member tag. Otherwise, uses ->Size and ->Get to create a generic iterator.
[ReverseIterator]	The same as [Iterator], but returns a reverse iterator.

Each iterator implements a number of member tags which can be used to move through the set of elements, to reset the iterator, or to fetch the current key or value for an element. These tags are summarized in the following table.

Table 24: Iterator and Reverse Iterator Member Tags

Tag	Description
[Iterator->Forward]	Moves the iterator forward one element. Returns True if the move was successful.
[Iterator->Backward]	Moves the iterator back one elements. Returns True if the move was successful.
[Iterator->AtEnd]	Returns True if the iterator is at the end element.
[Iterator->AtBegin]	Returns True if the iterator is at the beginning element.
[Iterator->Left]	Moves the iterator left one element. Returns True if the move was successful.
[Iterator->Right]	Moves the iterator right one element. Returns True if the move was successful.
[Iterator->AtFarLeft]	Returns True if the iterator is at the far left element.
[Iterator->AtFarRight]	Returns True if the iterator is at the far right element.
[Iterator->Up]	Moves the iterator up one element. Returns True if the move was successful.
[Iterator->Down]	Moves the iterator down one element. Returns True if the move was successful.
[Iterator->AtTop]	Returns True if the iterator is at the top element.
[Iterator->AtBottom]	Returns True if the iterator is at the bottom element.
[Iterator->Reset]	Resets the iterator to its default state. Usually resets to the beginning, far left, bottom element.
[Iterator->Key]	Returns the key for the current element if defined.
[Iterator->Value]	Returns a reference to the value of the current element.

[Iterator->RemoveCurrent]	Removes the current element from the compound data type and advances to the next value using [Iterator->Forward].
[Iterator->InsertAtCurrent]	Inserts an element into the compound data type at the current location.

The iterators for the built-in types only support the forward/backward dimension. The left/right and up/down tags will return False if a move is attempted and the test tags will return True since moving in that dimension is not possible.

There is no guarantee that moving through an iterator is symmetric. In general moving in one direction and then the opposite will return the iterator to the same element (for example moving forward then backward). However, when moving in multiple dimensions the behavior depends entirely on the underlying data structure. For example, moving up, left, down, right would not return the iterator to the starting element in most tree data structures.

To get an iterator for a data type:

There are several different methods for getting an iterator for a data type. The preferred method is to use the [Iterator] tag with the data type as a parameter.

```
[Var('myIterator' = Iterator($DataType))]
```

This method will return the built-in iterator if the data type supports the ->Iterator member tag (e.g. the array, list, map, set, and tree map types) or will construct a generic iterator using the data types ->Size and ->Get tags otherwise.

```
[Var('myIterator' = $DataType->Iterator)]
```

Note: The [ReverseIterator] tag can be used to get a reverse iterator.

To use an iterator with the while tags:

- An array iterator can be cycled through using [While] ... [/While] tags. The iterator tag [Iterator->atEnd] is checked in the condition of the opening [While] tag. If the iterator as at the end then the while loop is finished. Otherwise, the current value for the iterator is output using [Iterator->Value] and the iterator is advanced using [Iterator->Forward]. The [Null] tag suppresses the output from [Iterator->Forward].

```
<?LassoScript
  Var('myArray' = Array('One', 'Two', 'Three', 'Four'));
  Var('myIterator' = Iterator($myArray));
  While($myIterator->atEnd == False);
    '<br />' + $myIterator->Value;
    Null: $myIterator->Forward;
  /While;
?>
```

```
➜ <br />One
<br />Two
<br />Three
<br />Four
```

- The same code using [ReverseIterator] rather than [Iterator] will output the elements of the array in reverse order.

```
<?LassoScript
  Var('myArray' = Array('One', 'Two', 'Three', 'Four'));
  Var('myIterator' = ReverseIterator($myArray));
  While($myIterator->atEnd == False);
    '<br />' + $myIterator->Value;
    Null: $myIterator->Forward;
  /While;
?>
```

```
➜ <br />Four
<br />Three
<br />Two
<br />One
```

- A map iterator can be cycled through using [While] ... [/While] tags. The iterator tag [Iterator->atEnd] is checked in the condition of the opening [While] tag. If the iterator as at the end then the while loop is finished. Otherwise, the current value for the iterator is output using [Iterator->Key] and [Iterator->Value]. The iterator is advanced using [Iterator->Forward].

```
<?LassoScript
  Var('myArray' = Array(Map: 1='Sunday', 2='Monday', …));
  Var('myIterator' = Iterator($myArray));
  While($myIterator->atEnd == False);
    '<br />' + $myIterator->Key + ' = ' + $myIterator->Value;
    Null: $myIterator->Forward;
  /While;
?>
```

```
➜ <br />1 = Sunday
<br />2 = Monday
…
```

To use a matcher with an iterator:

A matcher can be passed to the [Iterator] tag or to the ->Iterator member tags of a compound data type.

- The [Match_Range] matcher can be used to restrict what part of an array is iterated over using the [Iterator] tag. For example, in the code below [Match_Range('a', 'm')] is used to show only those elements from the array that start with a letter in the first half of the alphabet.

```
<?LassoScript
  Var('myArray' = Array('One', 'Two', 'Three', 'Four'));
  Var('myIterator' = Iterator($myArray, (Match_Range: 'a', 'm')));
  While($myIterator->atEnd == False);
    '<br />' + $myIterator->Value;
    Null: $myIterator->Forward;
  /While;
?>
```

➜
Four

- Any value can be used as a matcher. For example, in the code below the keyword -SortField is used as a matcher to show only those elements from the array that start with -SortField.

```
<?LassoScript
  Var('myArray' = Array(-SortField='First_Name', -SortOrder='Descending'));
  Var('myIterator' = Iterator($myArray, -SortField));
  While($myIterator->atEnd == False);
    '<br />' + $myIterator->Value;
    Null: $myIterator->Forward;
  /While;
?>
```

➜
(Pair: (-SortField), (First_Name))

See the preceding section on matchers for a full list of possible matchers.

29

Chapter 29
Files

LDML provides three sets of tags that create and manipulate files on the Web server: include tags, logging tags, and file tags.

- *File Tags* describes the [File_...] tags which allow files and directories to be created, read, written, edited, moved, and deleted.
- *File Data Type* describes the [File] and [Directory] tags and data types, and their various member tags that allow files and folders to be manipulated using an object-oriented methodology.
- *File Uploads* describes the [File_Uploads] tag which allows files that have been uploaded with an HTML form to be manipulated and the [File_ProcessUploads] tag that automatically moves uploaded files to a destination directory.
- *File Serving* describes the [File_Serve] and [File_Stream] tags that can be used to serve files.

File Tags

The [File_...] tags can be used to read and write files on the same machine as Lasso Service. Any text or binary file with an approved file suffix can be manipulated. This chapter lists the LDML substitution tags that are available to list, inspect, read, write, modify, and delete files. Examples of using the tags are included both in this section and in the *File Upload* section that follows.

Note: See also the section on the *File Data Type* for information on how to manipulate files using an object-oriented methodology.

Specifying Paths

There are three different types of paths which can be used with the file tags depending on where the files that are to be manipulated are located.

- **Relative Paths** – Relative paths are specified from the location of the current format file. Relative paths follow the same basic rules as for paths specified within the [Include] tags or within HTML anchor <a> tags.

  For example, the following tag returns the creation date of a file named library.lasso located in the same folder as the current format file.

  ```
  [File_CreationDate: 'library.lasso']
  ```

  ➜ 11/6/2001 14:30:00

  The following tag returns the creation date of a file named library.lasso located in a sub-folder named Includes located in the same folder as the current format file.

  ```
  [File_CreationDate: 'Includes/library.lasso']
  ```

  ➜ 8/5/2001 15:35:30

  Note: The use of relative paths requires that Lasso Service and the Lasso Web server connector be running on the same machine. The file tags only work with files that are located on the same machine as Lasso Service.

- **Absolute Paths** – Absolute paths are specified from the root of the current Web serving folder. Absolute paths always start with a forward slash /. The root of the current Web serving folder is defined by the preferences of the Web server and usually corresponds to the location of the default page that is served when a simple URL such as http://www.example.com/ is visited.

  Relative paths follow the same basic rules as for paths specified within the [Include] tags or within HTML anchor <a> tags.

  For example, the following tag returns the creation date of a file named index.html located in the root of the Web serving folder.

  ```
  [File_CreationDate: '/index.html']
  ```

  ➜ 11/3/2001 16:06:15

  The following tag returns the creation date of a file named header.lasso located in a sub-folder named Includes located in the root of the Web serving folder.

  ```
  [File_CreationDate: '/Includes/header.lasso']
  ```

  ➜ 6/7/2001 8:35:45

Note: The use of relative paths requires that Lasso Service and the Lasso Web server connector be running on the same machine. The file tags only work with files that are located on the same machine as Lasso Service.

- **Mac OS X Fully Qualified Paths** – Fully qualified paths are specified from the root of the file system. They can be used to specify any files on the Web server including those outside of the Web serving root.

  In Mac OS X, fully qualified paths are always preceded by three forward slashes ///. This identifier is used to distinguish fully qualified paths from absolute paths. The root folder /// corresponds to the root of the file system as defined in the Terminal application (e.g. cd /).

  For example, the following tag returns the creation date of Lasso Service in Mac OS X.

  ```
  [File_CreationDate: '///Applications/Lasso Professional 8/LassoService']
  ```

  ➜ 11/3/2001 16:06:15

  The following tag returns the creation date of Admin.LassoApp located in the default Web serving folder in Mac OS X.

  ```
  [File_CreationDate: '///Library/WebServer/Documents/Lasso/Admin.LassoApp']
  ```

  ➜ 6/7/2001 8:35:45

  Partitions and mounted servers are located in the ///Volumes/ folder. The default Web serving folder for Apache is ///Library/WebServer/Documents/ and for WebSTAR V is ///Applications/4DWebSTAR/WebServer/DefaultSite/.

- **Windows Fully Qualified Paths** – Fully qualified paths are specified from the root of the file system. They can be used to specify any files on the Web server including those outside of the Web serving root.

  In Windows, fully qualified paths are always preceded by the letter name of a partition, a colon, and two forward slashes C:// or E:///. Any mounted partition can be referenced in this fashion.

  For example, the following tag returns the creation date of Lasso Service from the C: drive in Windows.

  ```
  [File_CreationDate: 'C://OmniPilot Software/Lasso Professional 8/LassoService.
  exe']
  ```

  ➜ 11/3/2001 16:06:15

  The following tag returns the creation date of Admin.LassoApp located in the default Web serving folder from the C: drive in Windows.

  ```
  [File_CreationDate: 'C://InetPub/WWWRoot/Lasso/Admin.LassoApp']
  ```

  ➜ 6/7/2001 8:35:45

Note: The file tags only work with files that are located on the same machine as Lasso Service or are accessible through a mounted file server.

File Suffixes

Any file which is manipulated by Lasso using the file tags must have an authorized file suffix within Site Administration. See the *Setting Site Preferences* chapter of the Lasso Professional 8 Setup Guide for more information about how to authorize file suffixes.

By default the following suffixes are authorized within Site Administration. Files named with any of these file suffixes can be used with the file tags.

.text	.txt
.bmp	.cmyk
.gif	.jpg
.pdf	.png
.psd	.rgb
.tif	.uld
.wsdl	.xml
.xsd	

Note: If permission has been granted to Any File Extension for the current user then the file suffix preferences are ignored and files with any file suffix can be manipulated.

Security

The use of file tags is restricted based upon what permissions have been granted in Site Administration. Any file operation must pass the following four security checks in order to be allowed.

- **File Tags Enabled** – The desired file tag must be enabled within the *Setup > Settings > Tags* section of Site Administration. Tags which are disabled in this section are not available for use by any user other than the global administrator.
- **File Tag Permissions** – The current user must have permission to execute the desired file tag. Permission is granted in the *Setup > Security > Tags* section of Site Administration. Permission must be granted for one of the groups in which the current user belongs or for the AnyUser group.
- **File Permissions** – The current user must have permission to execute the desired file action. Permission is granted in the *Setup > Security > Files* section of Site Administration. Permission must be granted for one of the groups to which the current user belongs or for the AnyUser group.

- **Allow Path** – The Allow Path for the current user must allow the file to be accessed. The Allow Path is specified in the *Setup > Security > Files* section of Site Administration. Any files in sub-folders of the allowed path can be manipulated using the file tags.
- **File Suffixes** – Discussed above. The file to be operated upon must be named with an approved file suffix.

The Any File Permission permission specified in the *Setup > Security > Files* section of Site Administration allows a user to access file without respect to the allowed path or file suffixes. Any files on the machine hosting Lasso Service can be manipulated.

Mac OS X Note: See the Mac OS X Tips document in the Documentation folder for information about how to configure Mac OS X file permissions.

The global administrator has permission to perform any Lasso actions and is able to access any files on the machine hosting Lasso service without regard to these security settings.

Table 1: File Tags

Tag	Description
[File_Copy]	Copies a file or directory from one location to another. Accepts two parameters, the location of the file or directory to be copied and the new location. Optional -FileOverWrite keyword specifies that the destination file should be overwritten if it exists.
[File_Create]	Creates a new, empty file or a new directory. Accepts one parameter, the location of the file or directory to be created. If the file name ends in a / then a directory is created. Optional -FileOverWrite keyword specifies that the destination file should be overwritten if it exists.
[File_CreationDate]	Returns the creation date of a file. Accepts one parameter, the name of the file or directory to be inspected.
[File_CurrentError]	Reports the last error reported by a file tag. Accepts an optional keyword -ErrorCode that returns the error code rather than the error message.
[File_Delete]	Deletes a file or directory. Accepts one parameter, the name of the file or directory to be deleted.
[File_Exists]	Returns True if the file or directory exists. Accepts one parameter, the name of the file or directory to be inspected.
[File_GetSize]	Returns the size in bytes of a file. Accepts one parameter, the name of the file to be inspected.

[File_IsDirectory]	Returns True if the specified path is a directory. Accepts one parameter, the name of the file or directory to be inspected.
[File_GetLineCount]	Returns the number of lines in a file. Accepts one parameter, the name of the file to be inspected. Optional -FileEndOfLine keyword/value parameter specifies what character represents the end of a line.
[File_ListDirectory]	Returns an array of strings. Each item in the array is the name of one file in the directory. Accepts one parameter, the name of the directory to be listed.
[File_ModDate]	Returns the modification date of a file. Accepts one parameter, the name of the file or directory to be inspected.
[File_Move]	Moves a file or directory from one location to another. Accepts two parameters, the location of the file or directory to be moved and the new location. Optional -FileOverWrite keyword specifies that the destination file should be overwritten if it exists.
[File_Read]	Reads the contents of a file. Accepts one parameter, the name of the file to be read. Two optional parameters -FileStartPos and -FileEndPos define the range of characters which should be read from the file.
[File_ReadLine]	Reads a single line from a file. Accepts two parameters, the name of the file to be read and -FileLineNumber specifying which line of the file to read. An optional keyword/value parameter -FileEndOfLine specifies what character represents the end of lines within the file.
[File_Rename]	Renames a file or directory. Accepts two parameters, the location of the file or directory to be copied and the new name. Optional -FileOverWrite keyword specifies that the destination file should be overwritten if it exists.
[File_SetSize]	Sets the size of the specified file. Accepts two parameters, the name of the file to be modified and the size in bytes which the file should be set to. Any data beyond that size in bytes will be truncated.
[File_Write]	Writes data to the specified file. Accepts two parameters, the name of the file to be written and the data which should be written into the file. Optional -FileOverWrite keyword specifies that the destination file should be overwritten if it exists, otherwise the data specified is appended to the end of the file.
[File_Chmod]	Allows the Unix file permissions of a file to be modified. Requires the path to the file to be modified and an octal permission string as parameters. This tag is currently supported on Mac OS X and Linux.

See *Appendix A: Error Codes* under the table *File Codes* for a list of error codes and messages which will be returned by the [File_CurrentError] tag.

To list a directory:

- Use the [File_ListDirectory] tag with the path to the directory. An array is returned which can be output using [Loop] ... [/Loop] tags. In the following example the contents of the Web serving folder on a Windows machine is listed by storing the array of files in an array File_Listing and then looping through the array. Each machine will have a different listing depending on what files have been installed in this directory.

```
[Variable: 'File_Listing' = (File_ListDirectory: 'C://InetPub/WWWRoot/')]
[Loop: ($File_Listing->Size)]
  <br>[$File_Listing->(Get: (Loop_Count))]
[/Loop]
```

```
➜ <br>default.htm
<br>default.lasso
<br>error.lasso
<br>Images/
<br>Lasso/
```

Note: The Web serving root on either platform can be listed using [File_ListDirectory: '/'] as long as both Lasso Service and the Web server are hosted on the same machine.

- The number of files in a directory can be counted by simply outputting the size of the array which is returned from [File_ListDirectory]. In the following example, the number of files in the Web serving folder listed above is returned.

```
[Variable: 'File_Listing' = (File_ListDirectory: 'C://InetPub/WWWRoot/')]
[$File_Listing->Size]
```

```
➜ 5
```

- More information about each of the files can be returned using the other file tags. The following example shows how to return the size, creation and modification dates of each of the files as well as whether each file is actually a file or a directory. The two directories do not have sizes or date information.

```
[Variable: 'File_Root' = 'C://InetPub/WWWRoot/']
[Variable: 'File_Listing' = (File_ListDirectory: $File_Root)]
[Loop: ($File_Listing->Size)]
  [Variable: 'File_Temp' = $File_Root + $File_Listing->(Get: (Loop_Count))]
  <br>[$File_Temp] [File_GetSize: $File_Temp]
  [File_CreationDate: $File_Temp] [File_ModDate: $File_Temp]
  [File_Exists: $File_Temp] [File_IsDirectory: $File_Temp]
[/Loop]
```

```
➜ <br>default.htm 4325 11/15/2000 14:00:12 11/13/2000 17:26:18 True False
  <br>default.lasso 12130 4/11/2000 12:33:29 3/17/2001 11:09:43 True False
  <br>error.lasso 393 11/13/2000 11:46:15 11/13/2000 11:50:47 True False
  <br>Images/ True True
  <br>Lasso/ True True
```

To create a new directory:

A new directory can be created using the [File_Create] tag. The tag creates a directory if the file name specified ends in a slash / character.

- The following tag would create a new directory named files at the root of the Web serving folder.

```
[File_Create: '/files/']
```

- The following tag would create a new directory named files at the root of the Web serving folder using a fully qualified path on Windows 2000.

```
[File_Create: 'C://InetPub/wwwroot//files/']
```

- The following tag would create a new directory named files at the root of the default Apache Web serving folder using a fully qualified path on Mac OS X.

```
[File_Create: '///Library/WebServer/Documents//files/']
```

To create a new file:

- A new file can be created using the [File_Create] tag. The data for the file in the following example comes from a variable File_Contents. The entire file newfile.lasso is written in one step using [File_Write]. If a file of the same name already exists in the specified directory it will be overwritten.

```
[File_Create: '/files/newfile.lasso', -FileOverWrite]
[File_Write: '/files/newfile.lasso', $File_Contents, -FileOverWrite]
```

- The following example shows how to do a safe file write. The code first checks to see if the desired output file is going to overwrite an existing file. A new file is created and the current error is checked using [File_CurrentError]. If no error occurred then the file is written using the [File_Write] tag.

```
[Variable: 'File_Path' = '/files/newfile.lasso']
[If: (File_Exists: $File_Path) == False]
  [File_Create: $File_Path]
  [If: (File_CurrentError) == (Error_NoError)]
    [File_Write: $File_Path, $File_Contents]
  [Else]
```

```
    <br>Error - Error Creating File
[Else]
  <br>Error - File already exists
[/If]
```

To import data from a file:

Data can be imported from a file using the [File_ReadLine] tag to read in each line of the file in turn. The lines of the file can then be parsed and stored in a database or shown to a user.

In the following example, each line of the file is assumed to be tab-delimited output from a database which is split into an array and could be later stored into a database. Each line of the file is split into an array Array_Temp and then the array is stored in the array File_Array.

```
[Variable: 'File_Path' = '///Library/WebServer/Documents/import.lasso']
[Variable: 'File_Array' = (Array)]

[If: (File_Exists: $File_Path)]
  [Loop: (File_GetLineCount: $File_Path)]
    [Variable: 'File_Temp' = (File_ReadLine: $File_Path,
        -FileLineNumber=(Loop_Count))]
    [Variable: 'Array_Temp' = $File_Temp->(Split: '\t')]
    [$File_Array->(Insert: ($Array_Temp))]
  [/Loop]
[/If]
```

The end result of importing the file is an array File_Array which contains an element for each line of the file. Each element is itself an array that contains an element for each tab-delimited item of data in the specified line.

To report errors while working with files:

Errors can be reported using the [File_CurrentError] tag. This tag works in much the same way as the [Error_CurrentError] tag. The following code creates a file and writes data into it, reporting errors at each step of the process.

```
[File_Create: 'e://files/newfile.lasso', -FileOverWrite]
<br>Error was [File_CurrentError: -ErrorCode]: [File_CurrentError].
[File_Write: 'e:\files\newfile.lasso', $File_Contents, -FileOverWrite]
<br>Error was [File_CurrentError: -ErrorCode]: [File_CurrentError].

➜ <br>Error was 0: No Error.
  <br>Error was 0: No Error.
```

See *Appendix A: Error Codes* under the table *File Codes* for a list of error codes and messages which will be returned by the [File_CurrentError] tag.

To change the Unix file permissons of a file:

Use the [File_Chmod] tag. The following example changes the Unix file system permissions of a file to -rwxrwxr-x (read, write, and execute permissions for the file owner, read, write, and execute permissions for the file group, and read and execute permissions for all other system users).

```
[File_Chmod: 'file.txt', -u='rwx', -g='rwx', -o='rx']
```

Line Endings

Files on Mac OS X, Windows, and Linux each have a different standard for line endings. The following table summarizes the different standards.

Table 2: Line Endings

Tag	Description
Mac OS X	Line feed: \n. Each line is ended with a single line feed character.
Windows	Line feed and carriage return: \r\n. Each line is ended with both a line feed and a carriage return character.
Linux	Line feed: \n. Each line is ended with a single line feed character.

Line ending differences are handled automatically by Web servers and Web browsers so are generally only a concern when reading and writing files using the [File_...] tags. The following tips make working with files from different platforms easier.

- The default line endings used by the [File_LineCount] and [File_ReadLine] tags match the platform default. They are \n in Mac OS and Linux, and \r\n in Windows.
- Specify line endings explicitly in the [File_LineCount] and [File_ReadLine] tags. For example, the following tag could be used to get the line count for a file orginally created in Linux.

  ```
  [File_LineCount: 'FileName.txt', -FileEndOfLine='\r']
  ```

  Or, the following tag could be used to get the line count for a file that was originally created on Windows.

  ```
  [File_LineCount: 'FileName.txt', -FileEndOfLine='\r\n']
  ```

- Many FTP clients and Web browsers will automatically translate line endings when uploading or downloading files. Always check the characters which are actually used to end lines in a file, don't assume that they will automatically be set to the standard of either the current platform or the platform from which they originated.

- A text editor such as Bare Bones BBEdit can be used to change the line endings in a file from one standard to another explicitly.

File Data Type

The file data type allows files to be cast as LDML objects and manipulated using member tags. This methodology is more advanced than the [File_...] tags methodology, giving Lasso developers a wide array of file connection modes and types for connecting to files.

Note: All guidelines for specifying file paths, file extensions, line endings, and permissions that were described in the ***File Tags*** section also apply to the file data type and its member tags described here.

File Data Type

To use the file streaming methodology, a file must first be cast as an LDML file variable using the [File] tag. This tag is described below.

Table 3: [File] Tag

Tag	Description
[File]	Casts a file as an LDML object, and sets the open and read modes. Requires the name and path to a file as a parameter.

When a file connection is opened using the [File] tag, several different open modes can be used. These modes optimize the file connection for best performance depending on the purpose of the connection. The open modes are described below.

Table 4: File Open Modes

Mode	Description
File_OpenRead	Sets the file connection to read-only.
File_OpenWrite	Sets the file connection to write-only.
File_OpenReadWrite	Sets the file connection to read and write.
File_OpenWriteAppend	Sets the file connection to write and append data.
File_OpenWriteTruncate	Sets the file connection to write and truncate data.

When using the [File] tag, a read mode may also be specified to determine how the file will be read. The read modes are described below.

Table 5: File Read Modes

Read Mode	Description
File_ModeChar	Reads a file character by character.
File_ModeLine	Reads a character line by line.

To cast a file as an LDML object:

Use the [File] tag. The example below casts a local file named myfile.txt as an LDML object in read-only/character mode. Note that no single quotes are used around the open and read mode designators, as they are type constants and not strings.

```
[Var:'File'=(File: 'myfile.txt', File_OpenRead, File_ModeChar)]
```

Manipulating File Objects

Once a file has been cast as an LDML file data type, various [File] member tags can be used to manipulate it. These tags can handle file specification, opening, closing, deleting, reading, writing and meta-data for files.

Table 6: File Streaming Tags

Tag	Description
[File->Open]	Opens a new connection to a file. Requires the name and path to a file as a parameter. Optional parameters may include an open mode and read mode, as in the [File] tag. The open mode and read mode set in the initial [File] tag call are used by default if not respecified.
[File->SetMode]	Sets the file read mode for the connection. This can be File_ModeLine for reading a file line by line, or File_ModeChar for reading a file character by character. Defaults to File_ModeChar if not specified.
[File->Read]	Reads data from a file. Requires the integer number of bytes (characters) to read as a parameter. Outputs the file data as bytes.
[File->Write]	Writes string data to a file. Requires the text string to write as a parameter. Two optional comma-delimited integer parameters may also be specified. The first specifies the number of characters of the text string to write, and the second specifies the number of characters in the text string to skip.
[File->SetPosition]	Sets the position of the file's read/write marker. Requires an integer line or character position (depending on mode) as a parameter. All subsequent reads and writes will occur at the given position.

[File->Position]	Returns the current file position. Defaults to 0 if no previous file operations have been performed.
[File->Get]	Returns the current character or line (depending on the file's read mode) at the current file position.
[File->SetSize]	Sets the size of the file. Requires an integer parameter that specifies the size of the file in bytes.
[File->MoveTo]	Moves the file to the new path. Requires a path on the local server as a parameter.
[File->Delete]	Deletes the file and reinitializes the type instance.
[File->Size]	Returns size of the file in bytes.
[File->Name]	Returns the file's name.
[File->Path]	Returns the full internal path to file.
[File->Close]	Closes a connection to a file. This tag should be called whenever a file streaming operation is finished.
[File->IsOpen]	Returns a value of True if the file connection has not been closed.

To read characters from a file:

Use the [File->Read] tag. The file object should be cast with an open mode that permits reading, and with the read mode set to File_ModeChar. The example below reads the first 256 characters of myfile.txt.

```
[Var:'File'=(File: 'myfile.txt', File_OpenRead, File_ModeChar)]
[$File->(Read: 256)]
[$File->Close]
```

To read characters from a file starting at a specified position:

Characters can be read starting at a set position using the [File->SetPosition] tag before the [File->Read] tag. The example below reads 240 characters starting at character number 16.

```
[Var:'File'=(File: 'myfile.txt', File_OpenRead, File_ModeChar)]
[$File->(SetPosition: 16)]
[$File->(Read: 240)]
[$File->Close]
```

To read lines from a file:

Use the [File->Read] tag. The file object should be cast with an open mode that permits reading, and with the read mode set to File_ModeLine. The example below reads the first 4 lines of myfile.txt.

```
[Var:'File'=(File: 'myfile.txt', File_OpenRead, File_ModeLine)]
[$File->(Read: 4)]
[$File->Close]
```

To read lines from a file starting at a specified position:

Lines can be read starting at a set position using the [File->SetPosition] tag before the [File->Read] tag. The example below reads 6 lines starting at line number four.

```
[Var:'File'=(File: 'myfile.txt', File_OpenRead, File_ModeLine)]
[$File->(SetPosition: 4)]
[$File->(Read: 6)]
[$File->Close]
```

To reset the read mode during file operations:

Use the [File->SetMode] tag to change the read mode. The example below starts in File_ModeLine mode, reads the first line of myfile.txt, moves to line five, changes to File_ModeChar mode, and then reads the next 16 characters.

```
[Var:'File'=(File: 'myfile.txt', File_OpenRead, File_ModeLine)]
[$File->(Read: 1)]
[$File->(SetPosition: 5)]
[$File->(SetMode: File_ModeChar)]
[$File->(Read: 16)]
[$File->Close]
```

To write text to a file:

Use the [File->Write] tag. The file object should be cast with an open mode that permits writing. The example below adds the text This is some text after the fifth line of the file.

```
[Var:'File'=(File: 'myfile.txt', File_OpenWrite, File_ModeLine)]
[$File->(SetPosition: 5)]
[$File->(Write:'this is some text')]
[$File->Close]
```

To write part of a string variable to a file:

Use the [File->Write] tag with the optional size and offset integer parameters. This is useful for truncating part of an existing string variable on-the-fly before writing it to the file. The example below adds the text five to the file out of a pre-defined string variable with a value of There are five elements.

```
[Var:'Text'='There are five elements']
[Var:'File'=(File: 'myfile.txt', File_OpenWrite, File_ModeLine)]
[$File->(Write: $Text, 4, 10)]
[$File->Close]
```

To return information about a file:

The [File->Name], [File->Path], and [File->Size] tags can be used to output the name, path, and size (in kilobytes) of a file. The example below outputs the file name, path, and size delimited by HTML line breaks.

```
[Var:'File'=(File: 'myfile.txt', File_OpenRead, File_ModeChar)]
[$File->Path]<br>
[$File->Name]<br>
[$File->Size]<br>
[$File->Close]
```

To move a file to a different folder:

Use the [File->MoveTo] tag. The following examples moves the local myfile.txt file to a different to folder on a Mac OS X hard drive.

```
[Var:'File'=(File: 'myfile.txt', File_OpenRead, File_ModeChar)]
[$File->(MoveTo:'///Library/WebServer/Documents/myfile.txt')]
[$File->Close]
```

File Uploads

Files can be uploaded to Lasso using standard HTML form inputs. Any uploaded files are processed by Lasso and stored in a temporary location. An array [File_Uploads] is provided that returns information about each of the uploaded files. The Lasso developer must write code to move the files to a safe location in the response page to the form in which they were uploaded. The [File_Copy] tag should be used to move uploaded files to a permanent location. Any files left in the temporary location once the format file has finished executing will be deleted.

File Permissions Note: File access permission for All Files is required for a user to upload files. For more information, see the ***Setting Up Security*** chapter in the Lasso Professional 8 Setup Guide.

HTML Form for File Upload

HTML forms must specify an enctype of multipart/form-data in order for file upload to work. An <input> tag with a type of file must be specified for each file that can be uploaded using the form. The following form includes a single <input> so one file can be uploaded to Lasso.

```
<form action="response.lasso" method="post" enctype="multipart/form-data">
  Select a file: <input type="file" name="upload" value="">
  <br><input type="submit" value="Upload File">
</form>
```

Once the site visitor selects a file using the file control shown in their browser and selects the Upload File button, the format file response.lasso will be called. Within this file the tag [File_Uploads] returns an array of information about each of the files uploaded with the form. In this case the array will only contain one item.

Table 7: File Upload Tags

Tag	Description
[File_Uploads]	Returns an array of maps that contain information about any files that were uploaded with the form that triggered the current format file.
[File_ProcessUploads]	Moves uploaded files into a destination directory. Allows the files to be filtered by size or file extension. Requires one parameter -Destination which is the desired directory for the uploaded files. By default files are restored to their original name. An optional -UseTempNames parameter will instead use the .uld name for each file. An optional -FileOverwrite paramater will overwrite files with matching names in the destination. An optional -Size parameter allows a maximum size in bytes to be specified. Only files smaller than this size will be moved. An optional -Extensions parameter allows an array of file extensions to be spcified. Only files that originally had one of these extensions will be moved.

Each element of the array returned by the [File_Uploads] tag is a map with the elements defined in the following table. If no files were uploaded then [File_Uploads] returns an empty array. Note that each <input> can only be used to upload one file, but multiple <input> tags can be specified in a single form to upload multiple files.

Table 8: [File_Uploads] Map Elements

Element	Description
Path	The path to the temporary location where the uploaded file is stored. Also accessible as Upload.Name.
File	A file object for the temporary file.
Size	The size of the uploaded file in bytes. Also accessible as Upload.Size.
Type	The type of the uploaded file. Also accessible as Upload.Type.
Param	The name of the form parameter which the file was uploaded in.

OrigName	The original name of the uploaded file without any path information.
OrigPath	The original path of the uploaded file (if provided) without any path information. Also accessible as Upload.RealName.
OrigExtension	The original file extensions of the uploaded file.

To display information about the uploaded files:

- Information about the uploaded files can be displayed to the site visitor by looping through the [File_Uploads] array. The following code loops through the array and returns information about each uploaded file on a separate line. The results are shown for a single uploaded file named Picture.gif.

```
[If: (File_Uploads->Size == 0)]
  No files were uploaded.
[Else]
  [Loop: (File_Uploads->Size)]
    [Variable: 'File_Temp'= (File_Uploads->(Get: (Loop_Count)))]
    <br>[$File_Temp->(Find: 'Path')]
      [$File_Temp->(Find: 'Size')]
      [$File_Temp->(Find: 'Type')]
      [$File_Temp->(Find: 'OrigPath')]
  [/Loop]
[/If]
```

```
➜ <br>E://WinNT/Temp/Lasso-tmp4.uld 128 image/gif Picture.gif
```

To move uploaded files to a permanent location:

All of the files which were uploaded will be deleted when the current format file is finished processing. Each uploaded file must be moved to another location in order to prevent it from being deleted. It is recommended that you use the [File_ProcessUploads] tag to move uploaded files.

The following code moves each file that was uploaded to a folder located at the path /uploads/ within the Web server root. Each file is restored to its original name and files with the same name in the destination are overwritten.

```
[File_ProcessUploads: -Destination='/uploads/', -FileOverwrite]
```

The code does not return any output, but all uploaded files are moved to the destination directory.

To move select uploaded files to a permanent location:

The following code moves only .gif or .jpeg files under 32 kbytes to a folder located at the path /uploads/ within the Web server root. Each file is restored to its original name and files with the same name in the destination are overwritten.

```
[File_ProcessUploads: -Destination='/uploads/', -FileOverwrite,
    -Size=32768, -Extensions=(Array: 'gif', 'jpg')]
```

The code does not return any output, but all uploaded files that match the specified criteria are moved to the destination directory.

To move uploaded files to a permanent location manually:

The following code moves each file that was uploaded to a folder located at the path e://uploads/ named upload1.txt, upload2,txt, etc. From there the files can be further manipulated or moved as needed.

```
[Variable: 'Path' = 'e://uploads/']
[If: (File_Uploads->Size == 0)]
  No files were uploaded.
[Else]
  [Loop: (File_Uploads->Size)]
    [Variable: 'File_Temp'= (File_Uploads->(Get: (Loop_Count)))]
    [File_Copy: $File_Temp->(Find: 'Upload.Name'),
      ($Path + 'upload' + (loop_count) + '.txt')]
  [/Loop]
[/If]
```

The code does not return any output if there were no files uploaded.

File Serving

Lasso can serve files in place of the current output file using the tags described in this section. Once one of these tags is called the current output is superseded by the output of the tag. These tags are usually used in a format file that acts as a proxy for the downloaded file and does not have any output other than the [File_Serve] or [File_Stream] tag.

Table 9: File Serving Tags

Tag	Description
[File_Serve]	Serves a file in place of the output of the current format file. The first parameter is the data to be served. Optional -File parameter specifies the name of the served data. Optional -Type parameter allows the MIME type to be overridden from the default of text/html.

[File_Stream]	Serves a file in place of the output of the current file. Uses very little memory so large files can be served. -File paramter specifies the path to a disk file. -Name specifies the name of the file to send to the client. Optional -Type parameter allows the MIME type to be specified (defaults to application/ octet-stream).

The two tags provide very similar functionality, but are ideal for different purposes.

- [File_Serve] should be used when the data to be served is generated within Lasso such as from the image, PDF, or file tags or from a database action. [File_Serve] is ideal for serving smaller files since the entire file must be stored in memory before it is sent to the client.
- [File_Stream] should be used when the data to be served is in a disk file. Since the tag will read the file off disk in small chunks it can be used to serve very large files with low overhead.

Note: The current user must have permission to read the specified file in order to use the [File_Stream] tag.

To serve an image from a FileMaker container field:

Pass the value of the [Database_FMContainer] field to the [File_Serve] tag. In the following example a single image is fetched from a database based on the value of the action parameter ID. The contents of the Image field is interpreted as a JPEG and passed to [File_Serve]. To the site visitor this file will serve a file named FileMakerImage.jpg.

```
[Inline: -Database='Contacts.fp5',
    -Layout='People',
    -KeyValue=(Action_Param: 'ID')
     -Search]
  [File_Serve:
      (Database_FMContainer: 'Image'),
      -Type='image/jpeg',
      -File='FileMakerImage.jpg']
[/Inline]
```

Note: The [File_Serve] tag replaces the current output of the page with the image and performs an [Abort]. The code above represents the complete content of a Lasso page.

The code above could be saved into a Lasso page called Image.Lasso. This page would then be referenced within an HTML <img> tag as follows.

```
<img src="Image.Lasso?ID=[KeyField_Value]" />
```

To stream a file to the Web client:

Use the [File_Stream] tag. The following example serves a file named example.mpg from within the Web server root.

```
[File_Stream:
    -File='/example.mpg',
    -Name='example.mpg',
    -Type='video/mpeg']
```

30

Chapter 30
Images and Multimedia

This chapter describes the methods which can be used to manipulate and serve images and multimedia files using Lasso.

- *Overview* provides an overview of the image manipulation and multimedia features included in Lasso Professional 8.
- *Casting Images as LDML Objects* describes how to cast image files as LDML objects so they can be dynamically edited using LDML.
- *Getting Image Information* describes how to access the attributes of an image using LDML.
- *Converting and Saving Images* describes how to convert images from one format to another, and how to save images to file using LDML.
- *Manipulating Images* describes how to edit image files using LDML.
- *Extended ImageMagick Commands* describes how to invoke extended ImageMagick functionality using LDML.
- *Serving Images and Multimedia Files* describes how to serve images and multimedia files through Lasso format files, and how to reference images and multimedia files stored within the Web server root.

Overview

Lasso Professional 8 includes features that allow you to manipulate and serve images and multimedia files on the fly. New LDML [Image] tags allow you to do the following with image files in supported image formats:

- Scaling and cropping images, facilitating the creation of thumbnail images on the fly.
- Rotating images and changing image orientation.

- Apply image effects such as modulation, blurring, and sharpening effects.
- Adjusting image color depth and opacity.
- Combining images, adding logos and watermarks.
- Image format conversion.
- Retrieval of image attributes, such as image dimensions, bit depth, and format.
- Executing extended ImageMagick commands.

Implementation Note: The image tags and features in LDML 8 are implemented using ImageMagick 5.5.7 (July 2003 build), which is installed as part of Lasso Professional 8 on Mac OS X 10.3. Windows requires ImageMagick to be installed separately, which is covered in chapter 4 of the Lasso Professional 8 Setup Guide. For more information on ImageMagick, visit http://www.imagemagick.com.

Introduction to Manipulating Image Files

Image files can be manipulated via LDML by setting a variable that references an image file on the server using the [Image] tag, and then using various member tags to manipulate the variable. Once the image file is manipulated, it can either be served directly to the client browser, or it can be saved to disk on the Web server.

To dynamically manipulate an image on the server:

The following shows an example of initializing, manipulating, and serving an image file named image.jpg using the [Image] tags.

```
[Var:'MyImage' = (Image: '/images/image.tif')]
[$MyImage->(Scale: -Height=35, -Width=35, -Thumbnail)]
[$MyImage->(Save: '/images/image.jpg')]

<img src="/images/image.jpg" border="0">
```

In the example above, an image file named image.tif is cast as a Lasso image data type using the [Image] tag, then resized to 35 x 35 pixels using the [Image->Scale] tag (the optional -Thumbnail parameter optimizes the image for the Web). Then, the image is converted to JPEG format and saved to file using the [Image->Save] tag, and displayed on the current page using an HTML <img> tag.

This chapter explains in detail how these and other tags are used to manipulate image and multimedia files. This chapter also shows how to output an image file to a client browser within the context of a format file.

Supported Image Formats

Because the [Image] tags are based on ImageMagick, Lasso Professional 8 supports reading and manipulating over 88 major file formats (not including sub-formats). A comprehensive list of supported image formats can be found at the following URL.

http://www.imagemagick.com/www/formats.html

A list of commonly used image formats that are certified to work with Lasso Professional 8 out of the box without additional components installed are shown in *Table 1: Tested and Certified Image Formats*.

Table 1: Tested and Certified Image Formats

Format	Description
BMP	Microsoft Windows bitmap.
CMYK	Raw cyan, magenta, yellow, and black samples.
GIF	CompuServe Graphics Interchange Format. 8-bit RGB PseudoColor with up to 256 palette entries.
JPEG	Joint Photographic Experts Group JFIF format. Also known as JPG.
PNG	Portable Network Graphics.
PSD	Adobe Photoshop bitmap file.
RGB	Raw red, green, and blue samples.
TIFF	Tagged Image File Format. Also known as TIF.

Note: Many of the supported formats listed on the ImageMagick site such as EPS and PDF may be used with the [Image_...] tags, but require additional components such as Ghostscript to be installed before they will work. These formats may be used, but because they rely heavily on third-party components, they are not officially supported by OmniPilot.

File Permissions

This section describes the file permission requirements for manipulating files on a Web server using LDML 8. In order to successfully manipulate and save image files, the following conditions must be met.

- When saving image files using the [Image] tags, the user (e.g. AnyUser group for anonymous users) must have Create Files, Read Files, and Write Files permissions allowed in the *Setup > Security > Files* section of Lasso Administration, and the folder in which the image will be created must be available to the user within the Allow Path field.

- When creating files, Lasso Service (i.e. the Lasso user in Mac OS X or LocalSystem user in Windows) must be allowed by the operating system to write and execute files inside the folder. To check folder permissions in Windows, right-click on the folder and select *Properties > Security*. For Mac OS X, refer to the included Mac_OS_X_Tips.pdf document for instructions on changing file and folder permissions.
- Any file extensions being used by the [Image] tags must be allowed in the *Setup > Global Settings > Settings* section of Lasso Administration. This can include .gif, .jpg, .png, or any other supported image format you are using.

Casting Images as LDML Objects

For Lasso to be able to edit an image via LDML, an image file or image data must first be cast as an LDML image variable using the [Image] tag. Once a variable has been set to an image data type, various Image member tags can be used to manipulate the image. Once the image file is manipulated, it can either be served directly to the client browser, or it can be saved to disk on the Web server.

Table 2: [Image] Tag:

Tag	Description
[Image]	Casts an image as an LDML object. Requires either the name and path of an image file or a binary data string to initialize an image. Once an image is cast as an object, it may be edited and saved using [Image] member tags, which are described throughout this chapter.

Table 3: [Image] Tag Parameters:

Parameter	Description
'File Path'	Path to image file on the server. Required if -Binary or -Base64 is not specified.
-Binary	Creates an image file from binary image data. Requires a valid binary string for a supported image format. Required if a file path is not specified.
-Info	Optional parameter retrieves all the attributes of an image without reading the pixel data. Allows for better performance and less memory usage when casting an image (recommended for larger files).

To cast an image file as an LDML object:

Use the [Image] to initialize an image file so it can be manipulated by Lasso.

```
[Var:'MyImage1'=(Image: '/images/image.jpg')]
```

To cast a large image file as an LDML object:

Use the [Image] to initialize an image file using the -Info parameter for increased performance with larger files.

```
[Var:'MyImage2'=(Image: '/images/largeimage.jpg', -Info)]
```

To initialize an image from binary image data:

Lasso can create an image from a binary string for a valid image type using the [Image] tag with the -Binary parameter. The image is initialized and created in memory only until it is saved using the [Image->Save] tag described later.

```
[Var:'Binary'=(Include_Raw: 'image.jpg')]
[Var:'MyImage3'=(Image: -Binary=$Binary)]
```

Getting Image Information

Information about an image can be returned using special [Image] member tags. These tags return specific values representing the attributes of an image such as size, resolution, format, and file comments. All image information tags in LDML 8 are defined in *Table 4: Image Information Tags*.

Table 4: Image Information Tags

Tag	Description
[Image->Width]	Returns the image width in pixels. Integer value returned.
[Image->Height]	Returns the image height in pixels. Integer value returned.
[Image->ResolutionH]	Returns the horizontal resolution of the image in dpi. Integer value returned.
[Image->ResolutionV]	Returns the vertical resolution of the image in dpi. Integer value returned.
[Image->Depth]	Returns the color depth of the image in bits. Can be either 8 or 16.
[Image->Format]	Returns the image format (GIF, JPEG, etc).

[Image->Pixel]	Returns the color of the pixel located at the specified pixel coordinates (X,Y). The returned value is an array of RGB color integers (0-255) by default. An optional -Hex parameter returns a hex color string (#FFCCDD) instead of an RGB array.
[Image->Comments]	Returns any comments included in the image file header.
[Image->Describe]	Lists various image attributes, mostly for debugging purposes. An optional -Short parameter displays abbreviated information.
[Image->File]	Returns the image file path and name, or null for in-memory images.

To return the height and width of an image:

Use the [Image->Height] and [Image-Width] tags on a defined image variable. This returns an integer value representing the height and width of the image in pixels.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->Width] x [$MyImage->Height]
```

➜ 400 x 300

To return the resolution of an image:

Use the [Image->ResolutionH] and [Image->ResolutionV] tags on a defined image variable. This returns a decimal value representing the horizontal and vertical dpi (dots per inch) of the image.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->ResolutionV] x [$MyImage->ResolutionH]
```

➜ 600 x 600

To return the color depth of an image:

Use the [Image->Depth] tag on a defined image variable. This returns an integer value representing the color depth of an image in bits.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->Depth]
```

➜ 16

To return the format of an image:

Use the [Image->Format] tag on a defined image variable. This returns a string value representing the file format of the image.

```
[Var: 'MyImage' =(Image: '/images/image.gif')]
[$MyImage->Format]
```

```
➜ GIF
```

To return pixel information about an image:

Use the [Image->Pixel] tag on a defined image variable. This returns a string value representing the color of the pixel at the specified coordinates.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->(Pixel: 25, 125, -Hex)]

➜ FF00FF
```

Converting and Saving Images

This section describes how image files can be converted from one format to another and saved to file. This is all accomplished using the [Image->Save] tag, which is described in the following table.

Table 5: Image Conversion and File Tags

Tag	Description
[Image->Convert]	Converts an image variable to a new format. Requires a file extension as a string parameter which represents the new format the image is being converted to (e.g. 'jpg', 'gif'). A -Quality parameter specifies the image compression ratio (integer value of 1-100) used when saving to JPEG or GIF format.
[Image->Save]	Saves the image to a file in a format defined by the file extension. Automatically converts images when the extension of the image to save as differs from that of the original image. A -Quality parameter specifies the image compression ratio (integer value of 1-100) used when saving to JPEG or GIF format.
[Image->AddComment]	Adds a file header comment to the image before it is saved. Passing a Null parameter removes any existing comments.

To convert an image file from one format to another:

Use the [Image->Convert] and [Image->Save] tags on a defined image variable, specifying the new format as part of the [Image->Convert] tag.

```
[Var: 'MyImage' =(Image: '/images/image.gif')]
[$MyImage->(Convert: 'JPG', -Quality=100)]
[$MyImage->(Save: '/images/image.jpg', -Quality=100)]
```

To automatically convert an image file from one format to another:

Use the [Image->Save] tag on a defined image variable, changing the image file extension to the desired image format. A -Quality parameter value of 100 specifies that the resulting JPEG file will be saved at the highest-quality resolution.

```
[Var: 'MyImage' =(Image: '/images/image.gif')]
[$MyImage->(Save: '/images/image.jpg', -Quality=100)]
```

To save a defined image variable to file:

Use the [Image->Save] tag on a defined image variable, specifying the desired image name, path, and format.

```
[Var: 'MyImage' =(Image: '/folder/asdf1.jpg')]
[$MyImage->(Save: '/images/image.jpg')]
```

To rename an image:

Use the [Image->Save] tag on a defined image variable, changing the existing image file name to the desired image file name.

```
[Var: 'MyImage' =(Image: '/images/image.gif')]
[$MyImage->(Save: '/images/image.jpg', -Quality=100)]
```

To add a comment to an image file header:

Use the [Image->AddComment] tag to add a comment to a defined image variable before it is saved to file. This comment is not displayed, but stored with the image file information.

```
[Var: 'MyImage' =(Image: '/images/image.gif')]
[$MyImage->(AddComment: 'This is a comment')]
[$MyImage->(Save: '/images/image.gif')]
```

To remove all comments from an image file header:

Use the [Image->AddComment] tag with a Null parameter to remove all comments from an image variable before it is saved to file. The following code adds a comment and then removes all comments. The result is an image with no comments.

```
[Var: 'MyImage' =(Image: '/images/image.gif')]
[$MyImage->(AddComment: 'This is a comment')]
[$MyImage->(AddComment: Null)]
[$MyImage->(Save: '/images/image.gif')]
```

Manipulating Images

Images can be transformed and manipulated using special [Image] member tags. These tags change the appearance of the image as it served to the client browser. This includes tags for changing image size and orientation, applying image effects, adding text to images, and merging images, which are described in the following sub-sections.

Changing Image Size and Orientation

Lasso provides tags that allow you to scale, rotate, crop, and invert images. These tags are defined in *Table 6: Image Size and Orientation Tags*.

Table 6: Image Size and Orientation Tags

Tag	Description
[Image->Scale]	Scales an image to a specified size. Requries -Width and -Height parameters, which specifiy the new size of the image using either integer pixel values (e.g. 50) or string percentage values (e.g. '50%'). An optional -Sample parameter indicates high-quality sampling should be used. An optional -Thumbnail parameter optimizes the image for display on the Web.
[Image->Rotate]	Rotates an image counterclockwise by the specified amount in degrees (integer value of 0-360). An optional -BGColor parameter specifies the hex color to fill the blank areas of the resulting image.
[Image->Crop]	Crops the original image by cutting off extra pixels beyond the boundaries specified by the parameters. Requires -Height and -Width parameters which specify the pixel size of the resulting image, and -Left and -Right parameters specify the offset of the resulting image within the initial image.
[Image->FlipV]	Creates a vertical mirror image by reflecting the pixels around the central X-axis.
[Image->FlipH]	Creates a horizontal mirror image by reflecting the pixels around the central Y-axis.

To enlarge an image:

Use the [Image->Scale] tag on a defined image variable. The following example enlarges image.jpg to 225 X 225 pixels. The optional -Sample parameter specifies that the highest-quality sampling should be used.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->(Scale: -Height=225, -Width=225, -Sample)]
[$MyImage->(Save: '/images/image.jpg')]
```

To contract an image:

Use the [Image->Scale] tag on a defined image variable. The following example contracts image.jpg to 25 x 25 pixels. The optional -Thumbnail parameter optimizes the image for the Web.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->(Scale: -Height=25, -Width=25, -Thumbnail)]
[$MyImage->(Save: '/images/image.jpg')]
```

To rotate an image:

Use the [Image->Rotate] tag on a defined image variable. The following example rotates the image 60 degrees counterclockwise on top of a white background.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->(Rotate: 60, -BGColor='FFFFFF')]
[$MyImage->(Save: '/images/image.jpg')]
```

To crop an image:

Use the [Image->Crop] tag on a defined image variable. The example below crops 10 pixels off of each side of a 70 x 70 image.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->(Crop: -Left=10, -Right=10, -Width=50, -Height=50)]
[$MyImage->(Save: '/images/image.jpg')]
```

To mirror an image:

Use the [Image->FlipV] tag on a defined image variable. The following example mirrors the image vertically.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->FlipV]
[$MyImage->(Save: '/images/image.jpg')]
```

Applying Image Effects

Lasso provides tags that allow you to add image effects by applying special image filters. This includes color modulation, image noise enhancement, sharpness controls, blur controls, contrast controls, and composite image merging. These tags are described below in *Table 7: Image Effects Tags*.

Table 7: Image Effects Tags

Tag	Description
[Image->Modulate]	Controls the brightness, saturation, and hue of an image. Brightness, saturation, and hue are controlled by three comma-delimited integer parameters, where 100 equals the original value.
[Image->Contrast]	Enhances the intensity differences between the lighter and darker elements of the image. Specify 'False' to reduce the image contrast, otherwise the contrast is increased.
[Image->Blur]	Applies either a motion or Gaussian blur to an image. To apply a motion blur, an -Angle paramater with a decimal degree value must be specified to indicate the direction of the motion. To apply a Gaussian blur, a -Gaussian keyword parameter must be specified in addition to -Radius and -Sigma parameters that require decimal values. The -Radius parameter is the radius of the Gaussian in pixels, and -Sigma is the standard deviation of the Gaussian in pixels. For reasonable results, the radius should be larger than the sigma.
[Image->Sharpen]	Sharpens an image. Requires -Radius and -Sigma parameters that require integer values. The -Radius parameter is the radius of the Gaussian sharp effect in pixels, and -Sigma is the standard deviation of the Gaussian sharp effect in pixels. For reasonable results, the radius should be larger than the sigma. Optional -Amount and -Threshold parameters may be used to add an unsharp masking effect. -Amount specifies the decimal percentage of the difference between the original and the blur image that is added back into the original, and -Threshold specifies the threshold in decimal pixels needed to apply the diffence amount.
[Image->Enhance]	Applies a filter that improves the quality of a noisy, lower-quality image.

To adjust the brightness of an image:

Use the [Image->Modulate] tag on a defined image variable and adjust the first integer parameter, representing brightness. The following example increases the brightness of an image by a factor of two.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->(Modulate: 200, 100, 100)]
[$MyImage->(Save: '/images/image.jpg')]
```

To adjust the color saturation of an image:

Use the [Image->Modulate] tag on a defined image variable and adjust the second integer parameter, representing color saturation. The following example decreases the color saturation of an image by 25%.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->(Modulate: 100, 75, 100)]
[$MyImage->(Save: '/images/image.jpg')]
```

To adjust the hue of an image:

Use the [Image->Modulate] tag on a defined image variable and adjust the third integer parameter, representing hue. The following example tints the image green by increasing the hue value. Decreasing the hue value tints the image red.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->(Modulate: 100, 100, 175)]
[$MyImage->(Save: '/images/image.jpg')]
```

To adjust the contrast of an image:

Use the [Image->Contrast] tag on a defined image variable. The first example increases the contrast. The second example uses a False parameter, which reduces the contrast instead.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->Contrast]
[$MyImage->(Save: '/images/image.jpg')]
```

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->(Contrast: 'False')]
[$MyImage->(Save: '/images/image.jpg')]
```

To apply a motion blur to an image:

Use the [Image->Blur] tag on a defined image variable. The following example applies a motion blur at 20 degrees.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->(Blur: -Angle=20)]
[$MyImage->(Save: '/images/image.jpg')]
```

To apply a Gaussian blur to an image:

Use the [Image->Blur] tag with the -Gaussian parameter on a defined image variable. The following example applies a Gaussian blur with a radius of 15 pixels and a standard deviation of 10 pixels.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->(Blur: -Radius=15, -Sigma=10, -Gaussian)]
[$MyImage->(Save: '/images/image.jpg')]
```

To sharpen an image:

Use the [Image->Sharpen] tag on a defined image variable. The following example applies a Gaussian sharp effect with a radius of 20 pixels and a standard deviation of 10 pixels.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->(Sharpen: -Radius=20, -Sigma=10)]
[$MyImage->(Save: '/images/image.jpg')]
```

To sharpen an image with an unsharp mask effect:

Use the [Image->Sharpen] tag with the -Amount and -Threshold parameters on a defined image variable. The following example applies an unsharp mask effect with a radius of 20 pixels and a standard deviation of 10 pixels.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->(Sharpen: -Radius=20, -Sigma=10, -Amount=50, -Threshold=20)]
[$MyImage->(Save: '/images/image.jpg')]
```

To enhance a low-quality image:

Use the [Image->Enhance] tag on a defined image variable.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->Enhance]
[$MyImage->(Save: '/images/image.jpg')]
```

Adding Text to Images

Lasso allows text to be overlaid on top of images using the [Image->Annotate] tag, as described below in the following table.

Table 8: Annotate Image Tag

Tag	Description
[Image->Annotate]	Overlays text on to an image. Requires a string value as a parameter, which is the text to be overlaid. Required -Left and -Top parameters specify the place of the text in pixel integers relative to the upper left corner of the image. An optional -Font parameter specifies the name (with extension) and full path to a system font to be used for the text, and an optional -Size parameter specifies the text size in integer pixels. An optional -Color parameter specifies the text color as a hex string ('#FFCCDD'). An optional -Aliased keyword parameter turns on text anti-alising.

Fonts Note: When specifiying a font, the full hard drive path to the font must be used (e.g. -Font='/Library/Fonts/Arial.ttf'). True Type (.ttf), and Type One (.pfa, .pfb) font types are officially supported.

To add text to an image:

Use the [Image->Annotate] tag on a defined image variable. The example below adds the text (c) 2003 OmniPilot Software to the specified image.

```
[Var: 'MyImage' =(Image: '/images/image.jpg')]
[$MyImage->(Annotate: '(c) 2003 OmniPilot Software', -Left=5, -Top=300,
        -Font='/Library/Fonts/Arial.ttf', -Size=8, -Color='#000000', -Aliased)]
[$MyImage->(Save: '/images/image.jpg')]
```

Merging Images

Lasso allows images to be merged using the [Image->Composite] tag. The [Image->Composite] tag supports over 20 different composite methods, which are described in the following tables.

Table 9: Composite Image Tag

Tag	Description
[Image->Composite]	Composites a second image onto the current image. Requires a second LDML image variable to be composited. An -Op parameter specifies the composite method which affects how the second image is applied to the first image (a list of operators is shown below). Optional -Left and -Top parameter specify the horizontal and vertical offset of the second image over the first in integer pixels (defaults to the upper left corner). An optional -Opacity parameter attenuates the opacity of the composited second image, where a value of 0 is fully opaque and 1.0 is fully transparent.

The table below shows the various composite methods that can be specified in the -Op parameter of the [Image->Composite] tag. The descriptions for each method are adapted from the ImageMagick Web site.

Table 10: Composite Image Tag Operators

Composite Operator	Description
Over	The result is the union of the the two image shapes with the composite image obscuring the image in the region of overlap.
In	The result is the first image cut by the shape of the second image. None of the second image data is included in the result.
Out	The result is the second image cut by the shape of the first image. None of the first image data is included in the result.
Plus	The result is the sum of the raw image data with output image color channels cropped to 255.
Minus	The result is the subtraction of the raw image data with color channel underflow cropped to zero.
Add	The result is the sum of the raw image data with color channel overflow channel wrapping around 256.
Subtract	The result is the subtraction of the raw image data with color channel underflow wrapping around 256.
Difference	Returns the difference between two images. This is useful for comparing two very similar images.
Bumpmap	The resulting image is shaded by the second image.
CopyRed	The resulting image is the red layer in the image replaced with the red layer in the second image.

CopyGreen	The resulting image is the green layer in the image replaced with the green layer in the second image.
CopyBlue	The resulting image is the blue layer in the image replaced with the blue layer in the second image.
CopyOpacity	The resulting image is the opaque layer in the image replaced with the opaque layer in the second image.
Displace	Displaces part of the first image where the second image is overlaid.
Threshold	Only colors in the second image that are darker than the colors in the first image are overlaid.
Darken	Only dark colors in the second image are overlaid.
Lighten	Only light colors in the second image are overlaid.
Colorize	Only base spectrum colors in the second image are overlaid.
Hue	Only the hue of the second image is overlaid.
Saturate	Only the saturation of the second image is overlaid.
Luminize	Only the luminousity of the the second image is overlaid.
Modulate	Has the effect of Hue, Saturate, and Luminize functions applied at the same time.

To overlay an image on top of another image:

Use the [Image->Composite] tag to add a defined image variable to a second defined image variable. The following example adds image2.jpg offset by five pixels in the upper left corner of image1.jpg.

```
[Var: 'MyImage1' =(Image: '/images/image1.jpg')]
[Var: 'MyImage2' =(Image: '/images/image2.jpg')]
[$MyImage1->(Composite: $MyImage2, -Left=5, -Top=5)]
[$MyImage1->(Save: '/images/image1.jpg')]
```

To add a watermark to an image:

Use the [Image->Composite] tag with the -Opacity parameter to add a defined image variable to a second defined image variable. The following example adds a mostly transparent version of image2.jpg to image1.jpg.

```
[Var: 'MyImage1' =(Image: '/images/image1.jpg')]
[Var: 'MyImage2' =(Image: '/images/image2.jpg')]
[$MyImage1->(Composite: $MyImage2, -Opacity=0.75)]
[$MyImage1->(Save: '/images/image1.jpg')]
```

To shade image with a second image:

Use the [Image->Composite] tag with the Bumpmap operator to shade a defined image variable over a second defined image variable.

```
[Var: 'MyImage1' =(Image: '/images/image1.jpg')]
[Var: 'MyImage2' =(Image: '/images/image2.jpg')]
[$MyImage1->(Composite: $MyImage2, -Op='Bumpmap')]
[$MyImage1->(Save: '/images/image1.jpg')]
```

To return the pixel difference between two images:

Use the [Image->Composite] tag with the Difference operator to return the pixel difference between two defined image variables.

```
[Var: 'MyImage1' =(Image: '/images/image1.jpg')]
[Var: 'MyImage2' =(Image: '/images/image2.jpg')]
[$MyImage1->(Composite: $MyImage2, -Op='Difference')]
[$MyImage1->(Save: '/images/image1.jpg')]
```

Extended ImageMagick Commands

For users who have experience using the ImageMagick command line utility, Lasso provides the [Image->Execute] tag to allow advanced users to take advantage of additional ImageMagick commands and functionality.

Table 11: ImageMagick Execute Tag

Tag	Description
[Image->Execute]	Execute ImageMagick commands. Provides direct access to the ImageMagick command-line interface. Supports the Composite, Mogrify, and Montage commands.

For detailed descriptions of the Composite, Mogrify, and Montage commands and corresponding parameters, see the following URL.

http://www.imagemagick.com/www/utilities.html

To execute an ImageMagick command using Lasso:

Use the [Image->Execute] tag on a defined image variable, with the desired command as the parameter. The following example shows the Mogrify command for adding a stunning blue border to an image.

```
[Var: 'MyImage' =(Image: '/images/image.gif')]
[$MyImage->(Execute: 'mogrify -bordercolor blue -border=3x3')]
[$MyImage->(Save: '/images/image.gif')]
```

Serving Image and Multimedia Files

This section discusses how to serve image and multimedia files, including referencing files within HTML pages and serving files separately via HTTP.

Referencing Within HTML Files

The easiest way to serve images and multimedia files is simply by referencing files stored within the Web server root using standard HTML tags such as <img> or <embed>. The path to the image file can be calculated in the format file or stored within a database field. Since the specified file is ultimately served by the Web server application which is optimized for serving images and multimedia files, this is the most efficient way to serve images and multimedia files.

To generate the path to an image or multimedia file:

- The following example shows a variable Company_Name that contains blueworld. This variable is used to construct a path to an image file stored within the Images folder named with the company name and _logo.gif to form the full file path /Images/blueworld_logo.gif.

```
[Variable: 'Company_Name'='blueworld']
<img src="/Images/[Variable: 'Company_Name']_logo.gif">

➜ <img src="/Images/blueworld_logo.gif">
```

- The following example shows a variable Company_Name that contains blueworld. This variable is used to construct a path to an image file stored within the Images folder named with the company name and _logo.gif to form the full file path /Images/blueworld_logo.gif. The path to the image file is stored within the variable Image_Path and then reference in the HTML <img> tag.

```
[Variable: 'Company_Name'='blueworld']
[Variable: 'Image_Path'='/Images/' + $Company_Name + '_logo.gif']
<img src="[Variable: 'Image_Path']">

➜ <img src="/Images/blueworld_logo.gif">
```

- The following example shows a variable Band_Name that contains ArtOfNoise. This variable is used to construct a path to sound files stored within the Sounds folder named with the band name and .mp3 to form the full file path /Sounds/ArtOfNoise.mp3. The path to the sound file is

stored within the variable Sound_Path and then reference in the HTML <a> link tag.

```
[Variable: 'Band_Name'='ArtOfNoise']
[Variable: 'Sound_Path'='/Images/' + $Band_Name + '.mp3']
<a href="[Variable: 'Sound_Path']">Download MP3</a>
➜ <img src="/Sounds/ArtOfNoise.mp3">Art of Noise Song</a>
```

Serving Files via HTTP

Lasso can also be used to serve image and multimedia files rather than merely referencing them by path. Files are served through Lasso using the [File_Serve] tag or a combination of the [Content_Type] tag and [Include_Raw] tags. LDML 8 also includes an [Image->Data] tag that automatically converts an image variable to a binary string, allowing an edited [Image] variable to be output by [File_Serve] without it first being written to file.

In order to serve an image or multimedia file through Lasso the MIME type of the file must be determined. Often, this can be discovered by looking at the configuration of the Web server or Web browser. The MIME type for a GIF is image/gif and the MIME type for a JPEG is image/jpeg.

Note: It is not recommended that you configure your Web server application to process all .gif and .jpg files through Lasso. Lasso will attempt to interpret the binary data of the image file as LDML code. Instead, use one of the procedures below to serve an image file with a .lasso extension.

Table 12: Image Serving Tag

Tag	Description
[File_Serve]	Serves a file in place of the output of the current format file. The first parameter is the data to be served. Optional -File parameter specifies the name of the served data. Optional -Type parameter allows the MIME type to be overridden from the default of text/html.
[Image->Data]	Converts an image variable to a binary string. This is useful for serving images to a browser without writing the image to file.

To serve an image file:

- Use the [File_Serve] tag to set the MIME type of the image to be served, and use the [Image->Data] tag to get the binary data from a defined [Image] variable. The [File_Serve] tag aborts the current file so it must be the

last tag to be processed. The following example shows a GIF named Picture.gif being served from an Images folder.

```
[Var:'Image'=(Image: '/Images/Picture.gif')]
[File_Serve: $Image->Data, -Type='image/gif']
```

- Use the [Content_Type] tag to set the MIME type of the image to be served and use the [Include_Raw] tag to include data from the image file. The two tags should be the only content of the file and should not be separated by any white space. The following example shows a GIF named Picture.gif being served from an Images folder.

```
[Content_Type: 'image/gif'][Include_Raw: '/Images/Picture.gif'][Abort]
```

If either of the code examples above is stored in a file named Image.lasso at the root of the Web serving folder then the image could be accessed with the following <img> tag.

```
<img src="http://www.example.com/Image.lasso">
```

To serve a multimedia file:

Use the [Content_Type] tag to set the MIME type of the file to be served and use the [Include_Raw] tag to include data from the multimedia file. The two tags should be the only content of the file and should not be separated by any white space. The following example shows a sound file named ArtOfNoise.mp3 being served from a Sounds folder.

```
[Content_Type: 'audio/mp3'][Include_Raw: '/Sounds/ArtOfNoise.mp3'][Abort]
```

If the code above is stored in a file named ArtOfNoise.lasso at the root of the Web serving folder then the sound file could be accessed with the following <a> link tag.

```
<a href="http://www.example.com/ArtOfNoise.lasso">Art of Noise Song</a>
```

This same technique can be used to serve multimedia files of any type by designating the appropriate MIME type in the [Content_Type] tag.

To serve an image file with a proper file extension:

The following example demonstrates how to serve a GIF file with a .gif extension. The extension .gif must be allowed in Lasso Administration. Use the [Content_Type] tag to set the MIME type of the image to be served and use the [Include_Raw] tag to include data from the image file.

```
[Content_Type: 'image/gif'][Include_Raw: '/Images/Picture.gif'][Abort]
```

The file will need to be referenced using Action.Lasso and a -Response command tag within the URL. If the code above is stored in a file named Image.gif at the root of the Web serving folder then the image could be accessed with the following <img> tag.

```
<img src="http://www.example.com/Action.Lasso?-Response=Image.gif">
```

To serve a multimedia file with a proper file extension:

The following example demonstrates how to serve a sound file with a .mp3 extension. The extension .mp3 must be allowed in Lasso Administration. Use the [Content_Type] tag to set the MIME type of the multimedia file to be served and use the [Include_Raw] tag to include data from the multimedia file.

```
[Content_Type: 'audio/mp3'][Include_Raw: '/Sounds/ArtOfNoise.mp3'][Abort]
```

The file will need to be referenced using Action.Lasso and a -Response command tag within the URL. If the code above is stored in a file named ArtOfNoise.mp3 at the root of the Web serving folder then the image could be accessed with the following <a> link tag.

```
<a href="http://www.example.com/Action.Lasso?-Response=ArtOfNoise.mp3">
  Art Of Noise Song
</a>
```

This same technique can be used to serve multimedia files of any type by designating the appropriate MIME type in the [Content_Type] tag.

To limit access to a file:

Since the format file can process any LDML code before serving the image it is easy to create a file that generates an error if an unauthorized person tries to access a file. The following code checks the [Client_Username] for the name John. If the current user is not named John then a file Error.gif is served instead of the desired Picture.gif file.

```
<?LassoScript
  Content_Type: 'image/gif';
  If: (Client_Username) == 'John');
    Include_Raw: '/Images/Picture.gif';
  Else;
    Include_Raw: '/Images/Error.gif';
  /If;
?>
```

This same technique can be used to restrict access to any image or multimedia file. It could actually be used to restrict access to any format file.

31

Chapter 31
Networking

LDML provides a network type that provides access to other servers through TCP and UDP communications..

- *Network Communications* describes the [Net] type.

Network Communication

Network communication in Lasso are provided by the [Net] type and its member tags. These tags allow for direct communication between Lasso and remote servers using low-level communication standards. These tags are the foundation for the implementation of specific protocols such as HTTP, RPC, or SMTP communication.

Note: Using the [Net] type requires an understanding of Internet communication standards. The examples in this chapter are purely for demonstration purposes of the [Net] tags.

The [Net] type supports the following features:

- **TCP (Transmission Control Protocol)** – Connection oriented communication with remote servers. TCP allows communication with full duplex capabilities and guaranteed delivery of data.
- **UDP (User Datagram Protocol) –** Connectionless communication with remote servers. UDP is a lightweight format that allows communication without guaranteed delivery of data.
- **SSL (Secure Socket Layer)** – Lasso allows TCP traffic to be encrypted in transit using SSL.

- **Listeners** – Lasso allows sockets to be opened to listen for either TCP or UDP traffic. Lasso can act as either the source or the recipient of TCP or UDP communication.
- **Non-Blocking** – Connections can be non-blocking so data is sent and received without synchronization with the remote host.
- **Timeouts** – Lasso has an efficient set of timeout controls that allow different timeout periods to be used when establishing connections and when participating in communication.

Note: The [TCP_...] tags from prior versions of Lasso have been deprecated. Solutions which rely on the [TCP_...] tags should be rewritten to make use of the new functionality afforded by the [Net] type.

The [Net] tag is listed in the *[Net] Tags* table. The constants that are returned from some network operations are detailed in the *[Net] Constants* table. The member tags of the [Net] type are split into three categories. The tags which are used to control connections for either TCP or UDP communication are listed in the *[Net] Type Members Tags* table. The tags specific to TCP communication are listed in the *[Net] TCP Member Tags* table and the tags specific to UDB communication are listed in the *[Net] UDP Member Tags* table.

The discussion that follows is split into three sections: *TCP Communication*, *TCP Listening*, and *UDP Communication*.

Table 1: [Net] Tags

Tag	Description
[Net]	Create a new network data type. Requires no parameter. All interaction with the network data type is performed.

Table 2: [Net] Constants

Tag	Description
[Net_ConnectOK]	Returned by [Net->Connect] if the connection was established.
[Net_Connect InProgress]	Returned by [Net->Connection] if another connection is in progress.
[Net_TypeTCP]	Passed to [Net->SetType] to establish TCP communication.
[Net_TypeSSL]	Passed to [Net->SetType] to establish SSL over TCP communication.
[Net_TypeUDP]	Passed to [Net->SetType] to establish UDP connectionless communication.

[Net_WaitRead]	Passed into and/or returned from [Net->Wait] to signal that bytes are available for reading from a connection.
[Net_WaitWrite]	Passed into and/or returned from [Net->Wait] to signal that bytes can be written into a connection.
[Net_WaitTimeout]	Returned from [Net->Wait] to signal that a timeout occurred.

Note: All of the [Net_...] constants represent values that are either passed into [Net] type member tags or returned from them. None of these tags are used on their own.

Table 3: [Net] Type Member Tags

Tag	Description
[Net->Bind]	Binds to a specific port on the local machine. Requires a single parameter which is the port on which to bind. Required for establishing a listener or reading bytes from a connectionless protocol (like UDP).
[Net->Close]	Closes an open or bound connection. Every connection which is opened should be explicilty closed when its use is completed.
[Net->LocalAddress]	Returns the address of the local host.
[Net->RemoteAddress]	Returns the address of the remote host.
[Net->SetBlocking]	Specifies whether connects, sends, and receives should block until the operation completes. Requires a single boolean parameter. The default is True to require blocking.
[Net->SetType]	Specifies whether the connection should use TCP or UDP. Requires a single parameter either [Net_TypeTCP], [Net_TypeSSL], or [Net_TypeUDP]. Defaults to TCP communication.
[Net->Wait]	For non-blocking sockets only. Waits for a specified number of seconds for the connection to enter a state. Requires one parameter which is the number of seconds to wait before timing out. A negative value will cause the tag to wait forever. An optional second parameter can be either [Net_WaitRead] or [Net_WaitWrite] specifying the state to wait for, otherwise either state will trigger a return. The tag returns the current state of the connection [Net_WaitRead] or [Net_WaitWrite] or [Net_WaitTimeout] if the timeout value was reached.

SSL Communication

SSL connections and listeners are established in exactly the same fashion as TCP connections and listeners. All of the same member tags are used except that [Net->(SetType: Net_TypeSSL)] must be called to instruct Lasso that SSL-based communication is required.

Notes are provided throughout the examples for TCP connections and listeners which provide details of how to establish SSL communication.

TCP Communication

TCP connections are some of the most common on the Internet. They are used for communication with Web servers, email servers, FTP servers, and for protocols like SSH and Telnet.

Table 3: [Net] TCP Member Tags

Tag	Description
[Net->Accept]	Accepts a single connection and returns a new [Net] instance for the connection.
[Net->Connect]	Connects to a remote host. Requires two parameters. The first is the DNS host name or IP address of the remote host. The second is the port on which to connect. Returns [Net_ConnectOK] if the connection was established or [Net_ConnectInProgress] if a connection attempt is already in progress.
[Net->Listen]	Switches the connection to an incoming, listening socket.
[Net->Read]	Reads bytes from the connection. Requires a single parameter which is the maximum number of bytes to be read. Returns the bytes read from the connection.
[Net->Write]	Writes bytes into the connection. Requires a single parameter which is the string to be written into the connection. Optional second and third parameters specify an offset and count of characters from the string to be written into the connection. Returns the number of bytes written.

To use a blocking TCP connection:

By default a TCP connection uses blocking to ensure that each communication completes before the next begins. This mode works best for command/response protocols in which commands are issued to the remote host and then the response to those commands is received back. Many standard Internet protocols like HTTP, SMTP, and FTP rely on this mechanism.

The basic outline of a TCP communication session is as follows.

1 A [Net] object is created and stored in a variable. This object will represent the communication channel with a remote server.

```
[Var: 'myConnection' = (Net)]
```

Note: If SSL communication is desired for the TCP connection then [$myConnection->(SetType: Net_TypeSSL)] should be called immediately after creating the [Net] object.

2 A connection to a remote server is established. The connection requires the DNS host name or IP address of the remote server and the port on which to connect.

```
[$myConnection->(Connect: 'localhost', 80)]
```

3 At this point the remote server might send a welcome message. HTTP servers (port 80) don't send any message. SMTP servers (port 25) send a message like the following. The parameter to [Net->Read] is the maximum number of characters to fetch. Even though 1024 is specified as the maximum number of characters to fetch, there are only about 32 characters in the connection buffer so that is all that is returned.

```
[$myConnection->(Read: 1024)]
```

```
➜ 220 localhost Mail Ready for action
```

4 A message can be sent through the channel to the remote server using the [Net->Write] tag. For example, sending GET / (followed by \r\n) to an HTTP server will get the HTML of the home page of the default site.

```
[$myConnection->(Write: 'GET /\r\n')]
```

5 The return value from the Web server can be read using [Net->Read]. Since this is a blocking connection the [Net->Read] tag will wait until the response from the remote server is complete before returning. The parameter to [Net->Read] is the maximum number of characters to fetch and should be larger than the expected result. In this case we will fetch the first 32 kilobytes of the Web page.

```
[$myConnection->(Read: 32768)]
```

```
➜ <html>\r<head>\r\t<title>Default Page</title>\r</head>\r<body>…</body>\r</html>
```

6 The connection should be closed once communication is complete.

```
[$myConnection->Close]
```

To use a non-blocking TCP connection:

When using a non-blocking TCP connection each [Net->Read] tag will return immediately with whatever data is currently available to be read. This means that if no data has been received from the remote server [Net->Read] will return with no bytes. Rather than repeatedly calling [Net->Read], the [Net->Wait] tag can be used to wait until there are bytes available to be read.

The basic outline of a non-blocking TCP session is as follows.

1. A [Net] object is created and stored in a variable. This object will represent the communication channel with a remote server.

   ```
   [Var: 'myConnection' = (Net)]
   ```

   Note: If SSL communication is desired for the TCP connection then [$myConnection->(SetType: Net_TypeSSL)] should be called immediately after creating the [Net] object.

2. A connection to a remote server is established. The connection requires the DNS host name or IP address of the remote server and the port which is to be connected to.

   ```
   [$myConnection->(Connect: 'localhost', 80)]
   ```

3. The connection is switched over to non-blocking mode using the [Net->SetBlocking] tag.

   ```
   [$myConnection->(SetBlocking: False)]
   ```

4. A message can be sent through the channel to the remote server using the [Net->Write] tag. For example, sending GET / (followed by \r\n) to an HTTP server will get the HTML of the home page of the default site.

   ```
   [$myConnection->(Write: 'GET /\r\n')]
   ```

5. A [Net->Wait] tag is used to wait until there is data which can be read through the connection. The [Net->Wait] tag takes two parameters. The first is the condition which is being waited for, in this case [Net_WaitRead], and the second is the number of seconds to wait. The tag below will wait for 60 seconds for data to be available.

   ```
   [$myConnection->(Wait: 60 Net_WaitRead)]
   ```

   This tag should be incorporated into a conditional statement so its return value can be checked. The return value from [Net->Wait] will be either [Net_WaitRead] if a read is possible or [Net_WaitTimeout] if the 60 seconds timeout was reached. The following code will perform a read from the connection only if data is available.

```
[If: ($myConnection->(Wait: 60 Net_WaitRead) == Net_WaitRead)]
  [$myConnection->(Read: 32768)]
[Else]
  Timeout!
[/If]
```

➜ <html>\r<head>\r\t<title>Default Page</title>\r</head>\r<body>…</body>\r</html>

6 The connection should be closed once communication is complete.

```
[$myConnection->Close]
```

TCP Listening

The [Net] type can be used to listen for connections coming in from remote clients. This allows Lasso to act as the server for different protocols. In theory, with this functionality Lasso itself could be used as an HTTP server or SMTP server.

The basic outline of a TCP listening session is as follows.

1 A [Net] object is created and stored in a variable. This object will represent the communication channel with a remote server.

```
[Var: 'myListener' = (Net)]
```

Note: If SSL communication is desired for the TCP listener then [$myListener->(SetType: Net_TypeSSL)] should be called immediately after creating the [Net] object.

2 The connection must be switched into listening mode and bound to a port on the local machine. This is the port that remote clients will access in order to communicate with the new service. In this example port 8000 is used.

```
[$myListener->(Bind: 8000)]
[$myListener->(Listen)]
```

3 Since this is a listener no further action is required until a remote client attempts a connection. The [Net->Accept] tag is used to wait for and accept a connection when one comes in. The result of the [Net->Accept] tag is a new [Net] object specific for the remote client that has connected. The listener is then free to call [Net->Accept] again and wait for the next connection.

```
[Var: 'myConnection' = $myListener->(Accept)]
```

4 Now, using the connection that has been established with the remote host, the particular needs of the protocol that is being implemented must be met. For this example, the connection will wait for a command from the remote client and then return a Web page in response.

```
[Var: 'myCommand' = $myConnection->(Read: 1024)]

[If: ($myCommand >> 'GET")]
  [$myConnection->(Write: '<html> … </html>')]
[Else]
  [$myConnection->(Write: 'Error: Unrecognized Command')]
[/If]

➜ <html>\r<head>\r\t<title>Default Page</title>\r</head>\r<body>…</body>\r</html>
```

6 The connection should be closed once communication is complete and if no further connections will be processed by the listener it should be closed as well.

```
[$myConnection->Close]
[$myListener->Close]
```

A listener can be blocking or non-blocking and can use the [Net->Wait] command to implement timeouts. The [Net] type can be used to create a listener that only accepts one connection at a time or to create a listener that spawns an asynchronous tag for each incoming connection so many connections can be handled simultaneously.

UDP Connections

UDP connections are generally used for simpler protocols on the Internet. UDP is considered connectionless. Rather than establishing a connection and then sending data, data will simply be sent to the remote host and a response listened for. UDP is an excellent method for one way communication, such as a status logging service, or for single command/response communication. UDP connections make use of the general [Net->Bind] and [Net->Wait] and [Net->Close] tags as well as two UDP specific tags, [Net->ReadFrom] and [Net->WriteTo].

Table 4: [Net] UDP Member Tags

Tag	Description
[Net->ReadFrom]	Reads whatever data is available from a UDP connection. Requires one parameter which is the maximum number of bytes to read. Returns a pair where the first part is the data and the second part is the name of the host that sent the data.

[Net->WriteTo]	Sends data to a specified host and port. Requires three parameters. The DNS host name or IP address of the remote host, the port to connect to, and the string data to be written. Optional additional parameters allow an offset and count into the string data to be specified. Returns the number of bytes written.

To send a message using UDP:

A message can be sent to a remote host with UDP using only the [Net->WriteTo] tag. This tag includes the connection information and message to send all in one call. This example implements a fictional TIME command that is sent to port 8000 on a remote machine. The remote machine will then send back the current time on port 8000.

1 A [Net] object is created and stored in a variable. This object will represent the communication channel with any remote UDP servers.

```
[Var: 'myConnection'= (Net)]
```

2 The [Net] object must be switched to UDP mode using the [Net->SetType] tag.

```
[$myConnection->(SetType: Net_TypeUDP)]
```

3 A message is sent to the remote server. The [Net->WriteTo] tag requires the DNS host name or IP address of the remote server and the port which is to be connected to as well as the message which is to be sent.

```
[$myConnection->(WriteTo: 'time.example.com', 8000, 'TIME')]
```

4 The connection should be closed once all UDP communication have completed. However, this same connection can be used to communicate with many different servers.

```
[$myConnection->Close]
```

Once a UDP message has been sent a listener must be established to wait for a reply. Since no connection is established there is no way to simply hold the channel open so the remote host can reply immediately.

To listen for a message using UDP:

Listening for a UDP message involves opening a port and then waiting for a message using the [Net->ReadFrom] tag. Messages can come in from any machine on the Internet. The incoming data is returned as the first part of the result from [Net->ReadFrom] and the address of the remote host is sent as the second part of the result.

1 A [Net] object is created and stored in a variable. This object will represent the communication channel with any remote UDP servers.

```
[Var: 'myListener'= (Net)]
```

2 The [Net] object must be switched to UDP mode using the [Net->SetType] tag.

```
[$myListener->(SetType: Net_TypeUDP)]
```

3 The [Net] object is bound to a local port using the [Net->Bind] tag. The local port is the port that other machines will send messages on. For this example, the listener is bound to port 8000.

```
[$myListener->(Bind: 8000)]
```

4 Now the listener must be wait for a message to come in from a remote server. The [Net->ReadFrom] tag will wait until a message comes in and then return a pair containing the data that has been received and the address of the host that sent the data. The parameter is the maximum number of bytes to read.

```
[Var: 'myMessage' = $myListener->(ReadFrom: 32768)]
[Var: 'myData' = $myMessage->First]
[Var: 'myHost' = $myMessage->Second]
```

The current time can now be output by displaying the data that was sent from the remote host.

```
The current time is: [Var: 'myData'].
```

5 The connection should be closed once all UDP communication have completed. However, this same connection can be used to communicate with many different servers.

```
[$myListener->Close]
```

32

Chapter 32
XML

This chapter describes how to parse and create Extensible Markup Language (XML) data and how to communicate using XML Remote Procedure Calls (XML-RPC).

- *Overview* introduces Lasso's XML support.
- *XML Glossary* introduces XML specific terms.
- *XML Data Type* describes how to parse and create XML data using the XML data type.
- *XPath Extraction* describes how to use XPath parameters to extract specific data from an XML file.
- *XSLT Style Sheet Transforms* describes how to transform XML data using XSLT style sheets.
- *XML Stream Data Type* describes how to parse XML documents using a stream model similar to a SAX parser.
- *Serving XML* describes how to serve XML data in place of the current format file.
- *Formatting XML* describes how to specify the MIME type and encode data for XML clients.
- *XML Templates* describes the XML templates included with Lasso and how to use them to format database action results as XML data.

Overview

Lasso provides support for a number of different XML standards which make parsing, validating, creating, transforming, and serving XML easy.

Lasso includes an XML data type that automatically parses XML from string values. The XML data type represents XML data as a tree data structure and includes member tags for manipulating the individual tags which make up the XML data. Changes can be made to the XML data type and will be automatically converted to proper XML syntax when output to the Web browser.

Lasso can validate XML data according to a Document Type Definition (DTD). If the XML data does not correspond to the structure defined by the DTD then an error will be returned to the user.

Lasso supports automatic transformations of XML data using the XSLT style sheets. An XSLT transform can be applied to XML data stored in a variable, database field, or file. In addition, a single format file can be repurposed for many different clients through the use of a stylesheet transform just prior to serving.

Lasso supports extracting individual XML elements from XML data using XPath parameters. The XPath language complements Lasso's built-in XML data type allowing sophisticated queries on XML data. The [XML_Extract] tag can be used to work with large XML documents.

XML-RPC support allows Lasso to communicate between servers. Lasso supports incoming XML-RPC requests through custom XML-RPC tags that are automatically processed or allows incoming requests to be processed by any format file. XML-RPC requests can be easily generated and sent to other servers on the Internet for processing.

Finally, Lasso can serve XML data which conforms to any Document Type Definition (DTD) or XML Schema using the same tools which allow Lasso to serve any style of HTML, WML, or other browser-based languages.

XML data needs to be formatted according to the rules defined by the World Wide Web Consortium. Documentation of this language is beyond the scope of this manual. Please consult a book on XML for more information about how to create properly formatted XML data.

Note: The XML data type should not generally be used to process XML documents larger than about 3 megabytes depending on their complexity. The [XML_Extract] tag can be used to parse much larger XML documents and extract specific elements for further processing.

XML Glossary

Here is a short glossary of essential terms which will help you understand the rest of this documentation if you are new to XML.

- **HTML** – HyperText Markup Language (HTML) is the language in which the World Wide Web is formatted and is characterized by markup tags enclosed in angle brackets. HTML is a subset of SGML.
- **XML** – Extensible Markup Language (XML) is the universal format for structured documents and data on the Web. XML is a subset of SGML.
- **SGML** – Standard Generalized Markup Language (SGML) is a system for defining markup languages. Authors mark up their documents by representing structural, presentational, and semantic information alongside content. HTML and XML are both based on SGML.
- **DTD** – A Document Type Definition (DTD) is a type of file associated with SGML and XML documents that defines how the markup tags should be interpreted by the application presenting the document.
- **Schema** – An XML-based method of specifying the structure of an XML document. Basically, a replacement for a DTD, but specified in XML syntax. This is an emerging standard which is yet to be ratified by the World Wide Web Consortium (W3C) at the time of this writing.
- **XPath** – A language which is used to define the location of one or more tags or attributes within XML data. XPaths can be used to extract tags or attributes from XML data and are used in XSLT style sheets. XPaths are used to extract specific elements from a larger XML document.
- **XSL** – Extensible Stylesheet Language (XSL) is a language for expressing stylesheets. An XSL stylesheet specifies the presentation of a class of XML documents by describing how an instance of the class is transformed into an XML document that uses the formatting vocabulary.
- **XSLT** – XSL Transformations (XSLT) is a language for transforming XML documents into other XML documents. XSLT is designed for use as part of XSL, which is a stylesheet language for XML.
- **XML-RPC** – XML Remote Procedure Call (XML-RPC) allows actions to be performed on another server on the Internet and for data to be returned.
- **WML** – Wireless Markup Language (WML) is an XML-based language in which "cards" for display on cellular phones and other wireless devices are created.

XML Data Type

The XML data type in Lasso automatically parses XML data which is stored in a variable. The member tags of the XML data type can then be used to inspect and change the XML data. The *XML Data Type Tag* table describes the tag which is used to convert string data to the XML data type.

Lasso also provides an alternate method of parsing XML data that may be more efficient for very large XML documents. This method is described in the *XML Stream Data Type* section below.

Table 1: XML Data Type Tag

Tag	Description
[XML]	Requires a single parameter which is a string containing validly formatted XML data. Optional -DTD parameter specifies a DTD against which the XML should be validated or optional -Schema parameter specifies an XML schema against which the XML should be validated. Optional -Validation parameter specifies whether the validation errors should be reported. Proper values include 'always', 'never', and 'auto'. The default is 'auto' which reports errors only if a schema or DTD was specified. Optional -Namespaces parameter specifies whether XML namespaces should be processed. Defaults to false. Optional -FullCheck parameter specifies whether full schema checking should be performed. This check can be very time consuming. Defaults to false.

XML data from any source can be parsed and manipulated using Lasso by first storing the XML data in a variable and then casting it to the XML data type using the [XML] tag. Lasso can work with XML data from a database field, XML file, remote Web application server, XML-RPC request, FTP site, etc. Or, Lasso can work with XML data that is created programmatically within a variable.

To parse XML data:

Use the [XML] tag to cast a string variable to the XML data type and parse the XML data which is contained within the variable. The following example stores a string of XML data in a variable, then casts it to XML.

```
[Variable: 'XML_String' = '<?xml version="1.0" encoding="UTF-8" ?>
  <ROOT>
    <RECORD>
      <FIELD name="First Name">John</FIELD>
      <FIELD name="Last_Name">Doe</FIELD>
    </RECORD>
  </ROOT>']
[Variable: 'XML_Data' = (XML: $XML_String)]
```

The variable XML_Data now contains a parsed representation of the data from XML_String. If the variable XML_Data is output the value of XML_String will simply be returned, but if the type of the variable is checked it will be XML.

```
[Variable: 'XML_Data']
<br>Type: [$XML_Data->Type]

➜ <?xml version="1.0" encoding="UTF-8" ?>
  <ROOT>
    <RECORD>
      <FIELD name="First_Name">John</FIELD>
      <FIELD name="Last_Name">Doe</FIELD>
    </RECORD>
  </ROOT>
<br>Type: XML
```

The parts of the parsed XML data can be accessed using the member tags of the XML data type which are detailed in the *XML Member Tags* table.

Table 2: XML Member Tags

Tag	Description
[XML->Attributes]	An array of pairs for each of the attributes of the root tag.
[XML->Name]	The name of the root tag.
[XML->Children]	An array of XML objects for each of the children tags of the root tag.
[XML->Contents]	The contents of the root tag.
[XML->Document]	Returns the root tag of the current XML document.
[XML->Extract]	Returns the value for an XPath. Requires a single parameter which is the XPath to be evaluated. Returns different values depending on the XPath.
[XML->ExtractOne]	Works the same as [XML->Extract] but only returns the first element found by the XPath.

[XML->NameSpaces]	Returns an array of namespaces for the namespaces declared for this node. The array has pairs in which the first element is the namespace prefix and the second element is the URI of the namespace.
[XML->NextSibling]	Returns the next sibling of the current node.
[XML->NodeType]	Returns the type of the current node.
[XML->Parent]	Returns the parent for the current node.
[XML->Path]	Returns the path to the current node from the root of the document.
[XML->PreviousSibling]	Returns the previous sibling for the current node.
[XML->Transform]	Performs an XSLT style sheet transformation on the current XML object. Requires a string which contains a valid XSLT style sheet. Returns a new XML object with the results of the transformation.

These member tags can be used to inspect the attributes and children of an XML tag.

To find specific children of an XML tag:

Use the [XML->Children] tag to get an array of children of an XML tag. For example, the children of the <ROOT> tag in XML_Data can be returned as follows. The result is always an array even if there is only one child of the root XML tag.

```
[$XML_Data->Children]
```

```
➜ (Array: (<RECORD> ... </RECORD>))
```

The children of the <RECORD> tag can be found by extracting the <RECORD> tag from the array of children using [Array->Get] and then using [XML->Children] to return an array of <FIELD> tags..

```
[Variable: 'XML_Record' = $XML_Data->Children->(Get: 1)]
[$XML_Record->Children]
```

```
➜ (Array: (<FIELD name="First_Name">John</FIELD>),
   (<FIELD name="Last_Name">Doe</FIELD>))
```

To display the attributes of an XML tag:

Use the [XML->Attributes] tag. The following example returns the attributes of the first element from the XML_Record variable in the previous example. The [Iterate] ... [/Iterate] tags are used to cycle through the array of attributes and the elements of each attribute pair are displayed.

```
[Variable: 'XML_Attributes' = $XML_Record->Attributes]
[Iterate: $XML_Attributes, (Variable: 'Attribute')]
  <br>[$Attribute->First] = [$Attribute->Second]
[/Iterate]
```

➜
Name = First_Name

To display the contents of an XML tag:

Use the [XML->Contents] tag. The following example returns the contents of the first element from the XML_Record variable in the example above.

```
[Encode_HTML: $XML_Record->Contents]
```

➜ John

XPath Extraction

XPath is a language that allows XML data to be searched for specific tags or attributes. An XPath expression instructs how to get to a specific tag or tags within XML data similarly to how a file system path instructs how to get to a specific file within a hard drive.

This is the preferred method of processing large XML documents. An XPath can be used to extract the relevant elements from the large XML document and then those individual elements can be converted to the XML data type for further processing.

For example, the Lasso Service application on Mac OS X is represented by the following path. The path says to go to the root of the file system /, enter the Applications folder then the Lasso Professional 8 folder, and look for the file named LassoService.

```
/Applications/Lasso Professional 8/LassoService
```

Similarly, an XPath to navigate through the following XML data can be constructed.

```
[Variable: 'XML_String' = '<?xml version="1.0" encoding="UTF-8" ?>
  <ROOT>
    <RECORD>
      <FIELD name="First Name">John</FIELD>
      <FIELD name="Last_Name">Doe</FIELD>
    </RECORD>
  </ROOT>']
```

The following XPath starts at the root of the XML data, the <ROOT> tag represented by /ROOT. It enters the <RECORD> tag and returns the <FIELD> tag which has a name attribute equal to First_Name.

```
/ROOT/RECORD/FIELD[@name="First_Name"]
```

The [XML_Extract] tag allows an XPath to be applied to XML data within Lasso and for the results to be returned.

Table 3: [XML_Extract] Tag

Tag	Description
[XML_Extract]	Accepts two parameters and returns an array of string. -XML is the XML source data or -File specifies the path to a file that contains the XML source data. -XPath is the XPath that describes what data to return.

Note: The [XML_Extract] tag will read XML data from a -File parameter if it is present. This is the preferred method of working with large XML documents since Lasso can parse the file without reading it into memory. If no -File parameter is specified then the data passed directly to the -XML parameter is used instead.

The XPath from above would be applied to the XML data in this way.

```
[XML_Extract: -XML=$XML_String,
  -XPath='/ROOT/RECORD/FIELD[@name=First_Name]']
```

The return value is an array containing a single string representing the tag which was found.

```
➜ (Array: (<FIELD name="First_Name">John</FIELD>))
```

File paths generally only allow inspecting the names of files and directories. XML tags have a name, children, attributes, contents, etc. The XPath allows any of these different aspects of XML tags to be used in specifying a path to a specific tag or set of tags.

The *Simple XPath Expressions* table includes the basic elements of an XPath. These can be be combined with the conditional functions detailed in the *Conditional XPath Expressions* table to create sophisticated queries allowing very specific sets of tags and sub tags to be extracted from XML data.

Note: A full discussion of XPath syntax is beyond the scope of this book. Please consult a book about XML for full details about XPath syntax.

Table 4: Simple XPath Expressions

Expression	Description
/	Selects the root element of the XML data. The first XML tag in the XML data is a child of the root element.
/*	Selects all children elements from the current element including XML tags and text elements. /node() is a synonym.
/tagname	Selects all XML tag children with the specified tag name from the current element.
/text()	Selects all text element children from the current element .
//*	Selects all descendents starting from the current element including XML tags and text elements. //node() is a synonym.
//tagname	Selects all XML tags descendents with the specified tag name starting from the current element.
//text()	Selects all text element descendents starting from the current element.
/@*	Selects all attributes of the current tag.
/@attribute	Selects all attributes of the current tag with the specified attribute name.

These expressions are assembled into a path by placing them in the appropriate order depending on the tags or attributes that need to be extracted. The following are some examples of XPaths and what tags they would extract from the XML data specified on the previous page.

- Select all <ROOT> tags.

  /ROOT

- Select all <RECORD> tags which are contained in the <ROOT> tag.

  /ROOT/RECORD/

- Select all <FIELD> tags which are children of a <ROOT> and <RECORD> tag.

  /ROOT/RECORD/FIELD

- Select the text contents of all <FIELD> tags which are children of a <ROOT> and <RECORD> tag.

  /ROOT/RECORD/FIELD/text()

- Select all <FIELD> tags no matter what the name of their parent tag was.

  //FIELD

- Select the name attributes from all <FIELD> tags.

  //FIELD/@name

- Select all attributes from all <FIELD> tags.

  ```
  //FIELD/@*
  ```

- Select all text elements from the XML data.

  ```
  //text()
  ```

To extract tags from XML data using simple XPath expressions:

The simple XPath expressions can be used to find a specific set of nodes within XML data. For example, using the same XML data as for the example above the following XPaths return the specified results.

```
[Variable: 'XML_String' = '<?xml version="1.0" encoding="UTF-8" ?>
  <ROOT>
    <RECORD>
      <FIELD name="First Name">John</FIELD>
      <FIELD name="Last_Name">Doe</FIELD>
    </RECORD>
  </ROOT>']
```

- The root tag of the XML data and all of its contents can be returned using /ROOT/. The <ROOT> tag and all its contents are returned.

  ```
  [XML_Extract: -XML=$XML_String, -XPath='/ROOT/']
  ```

  ➜ (Array: (<ROOT> ... </ROOT>))

- All children of the root tag can be returned using /ROOT/*. The <RECORD> tag and all its contents are returned.

  ```
  [XML_Extract: -XML=$XML_String, -XPath='/ROOT/*']
  ```

  ➜ (Array: (<RECORD> ... </RECORD>))

- All <FIELD> tags in the XML data can be returned using //FIELD. The two <FIELD> tags and all of their contents are returned.

  ```
  [XML_Extract: -XML=$XML_String, -XPath='//FIELD']
  ```

  ➜ (Array: (<FIELD name="First_Name">John</FIELD>),
  (<FIELD name="Last_Name">Doe</FIELD>))

- The name parameter from all <FIELD> tags in the XML data can be returned using //FIELD/@name. The name parameters of the <FIELD> tags are returned.

  ```
  [XML_Extract: -XML=$XML_String, -XPath='//field/@name']
  ```

  ➜ (Array: (First_Name), (Last_Name))

Many complex queries can be created using the simple XPath parameters. In addition, XPath allows for conditional expressions to be used on the simple XPath expressions. These are detailed in the *Conditional XPath Expressions* table.

Table 5: Conditional XPath Expressions

Expression	Description
[n]	A number selects one specific element from an array of returned tags or parameters.
[last()]	Returns the last element from an array of returned tags or parameters.
[tagname]	Returns only tags which have one or more children with the specified tag name.
[@attribute]	Returns only those tags which have the specified attribute.
[@attribute=value]	Returns only those tags which have the specified attribute equal to the value.
[.=value]	Returns only those tags which have their contents equal to the specified value.
[expression = value]	Returns only those tags for which the expression is equal to the specified value. Can also use < <= > >= or != for numeric comparisons.
[starts-with(expression, value)]	Returns only those tags which have an attribute or child tag that starts with the specified value.
[contains(expression, value)]	Returns only those tags which have an attribute or child tag that contains the specified value.
[count(expression)]	Returns the number of elements in an array of returned tags or parameters.

In addition to the expressions detailed in this table it is possible to use numeric functions + - * div mod, boolean operations and or or, and parentheses to create more complex expressions.

To extract tags from XML data using conditional XPath expressions:

The conditional XPath expressions can be used to find a specific set of nodes within XML data. For example, using the same XML data as the example above, the following XPaths return the specified results.

```
[Variable: 'XML_String' = '<?xml version="1.0" encoding="UTF-8" ?>
  <ROOT>
    <RECORD>
      <FIELD name="First Name">John</FIELD>
      <FIELD name="Last_Name">Doe</FIELD>
    </RECORD>
  </ROOT>']
```

- The first <FIELD> tag in the XML data can be returned using //FIELD[1]. The first <FIELD> tag and all of its contents are returned.

```
[XML_Extract: -XML=$XML_String, -XPath='//FIELD[1]']
```

```
➜ (Array: (<FIELD name="First_Name">John</FIELD>))
```

- The last <FIELD> tag descendant of the root tag can be returned using //FIELD[last()]. The second <FIELD> tag and all of its contents are returned.

```
[XML_Extract: -XML=$XML_String, -XPath='//FIELD[last()]']

➜ (Array: (<FIELD name="Last_Name">Doe</FIELD>))
```

- All <FIELD> tag descendants of the root tag which have their contents equal to John can be returned using //FIELD[.="John"]. The . in the expression represents the current tag that is being examined. The first <FIELD> tag and all of its contents are returned.

```
[XML_Extract: -XML=$XML_String, -XPath='//FIELD[.="John"]']

➜ (Array: (<FIELD name="First_Name">John</FIELD>))
```

- All <FIELD> tag descendants of the root tag which have a name parameter that contains the word Name can be returned using //FIELD[contains(@name, "Name")]. Both <FIELD> tags and all of their contents are returned.

```
[XML_Extract: -XML=$XML_String, -XPath='//field[contains(@name, "Name")]']

➜ (Array: (<FIELD name="First_Name">John</FIELD>),
  (<FIELD name="Last_Name">Doe</FIELD>))
```

XSLT Style Sheet Transforms

XML style sheets allow one set of XML data to be transformed to a different set. A single base XML document can be converted so it can be used in many different situations. For example, a single document could be converted to HTML for display in a Web browser and to WML for display in a wireless device.

Lasso allows XSLT transforming style sheets to be applied to XML data using the [XML_Transform] tag. The input of the tag is a string containing XML source data and a string containing a valid XSLT style sheet. The result is the string that is generated by applying the style sheet to the data.

Note: A full discussion of XSLT syntax is beyond the scope of this book. Please consult a book about XML for full details about XML style sheets.

Table 6: [XML_Transform] Tag

Tag	Description
[XML_Transform]	Accepts two parameters and returns a string. -XML is the XML source data. -XSL is the XSLT style sheet to be applied.

Note: It is important to include version and xmlns:xsl parameters in the opening <xsl:stylesheet> tag passed to Lasso so the XSLT processor knows what version of XSL is being used and what namespace to use when parsing the XSLT style sheet.

To transform XML data using an XSLT style sheet:

Use the [XML_Transform] tag. The following example uses an XSLT style sheet stored in XSLT_String to transform XML data stored in XML_String and output an HTML table.

The XSLT style sheet is an XML document that uses XPath expressions to select portions of the XML data and transform them into a different format. In this case, XML data is transformed into HTML for display in a Web browser.

The <xsl:template> tag specifies what XML element the style sheet will transform. The <xsl:for-each> tags accept an XPath that specifies a specific set of elements to iterate through. The contents of the tags is repeated for each iteration. The <xsl:value-of> tag returns the value of an XPath. In this example, it used both to return the name parameter from each <FIELD> tag and to return the value of the <FIELD> tag itself.

```
[Variable: 'XSLT_String' = '<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="//ROOT">
    <TABLE>
      <TR>
        <xsl:for-each select="RECORD[1]/FIELD/@name">
          <TD><B><xsl:value-of select="self()"/></B></TD>
        </xsl:for-each>
      </TR>
      <xsl:for-each select="RECORD">
        <TR>
          <xsl:for-each select="FIELD">
            <TD><xsl:value-of select="self()"/></TD>
          </xsl:for-each>
        </TR>
      </xsl:for-each>
    </TABLE>
  </xsl:template>
</xsl:stylesheet>']
```

The XML data is stored in XML_String.

```
[Variable: 'XML_String' = '<?xml version="1.0" encoding="UTF-8" ?>
  <ROOT>
    <RECORD>
      <FIELD name="First Name">John</FIELD>
      <FIELD name="Last_Name">Doe</FIELD>
    </RECORD>
  </ROOT>']
```

The transformation is performed using the [XML_Transform] tag and the results are shown.

```
[XML_Transform: -XML=$XML_String, -XSL=$XSLT_String]

➜ <TABLE>
    <TR>
      <TD><B>First_Name</B></TD>
      <TD><B>Last_Name</B></TD>
    </TR>
    <TR>
      <TD>John</TD>
      <TD>Doe</TD>
    </TR>
  </TABLE>
```

XML Stream Data Type

The XML stream data type in Lasso automatically parses XML data which is stored in a variable. The member tags of the XML stream data type can then be used to inspect and change the XML data. The *XML Stream Data Type Tag* table describes the tag which is used to convert string data to the XML stream data type.

The XML stream data type treats XML data as a stream of objects which will be consumed one by one until the end of the document is reached. This method of parsing XML documents is comparable to the SAX methodology.

The member tags of the XML data type can also be used to parse XML data. These methods are describes in the preceding *XML Data Type* section.

Table 7: XML Stream Data Type Tag

Tag	Description
[XMLStream]	Accepts a single parameter which is a string containing validly formatted XML data.

XML data from any source can be parsed and manipulated using Lasso by first storing the XML data in a variable and then casting it to the XML stream data type using the [XMLStream] tag. Lasso can work with XML data from a database field, XML file, remote Web application server, XML-RPC request, FTP site, etc. Or, Lasso can work with XML data that is created programmatically within a variable.

Navigating an XML Stream

An XML stream is made up of many objects called nodes. A node is a opening XML tag, a closing tag, an attribute of an XML tag, a string of text, CDATA, a processing instruction, a comment, or others. All of the available node types are detailed in the *XML Stream Node Types* table.

Table 8: XML Stream Node Types

Type	Description
startElement	An opening XML tag. Opening XML tags are the only nodes that have attributes.
endElement	A closing XML tag.
attributes	An attribute of an opening XML tag.
text	The text contents of an XML tag.
cdata	The CDATA contents of an XML tag.
entityref	An entity reference. Often used for extended characters like & representing the ampersand &.
entitydecl	An entity declaration.
pi	A processing instruction. LassoScript embedded in an XML document would be considered a processing instruction. <? … ?>
comment	A comment. These are formatted the same as HTML comments <!-- … -->
document	The root of the XML document.
dtd	A document type declaration.
documentfrag	A fragment of a document.
notation	A notation.

The member tags which are used to set the current node of the XML stream are detailed in the *XML Stream Navigation Member Tags* table. All of these tags return a boolean value if the desired operation could be performed. The current node for the stream is changed and the member tags in the *XML Stream Member Tags* table can then be used to inspect the current node.

Table 9: XML Stream Navigation Member Tags

Tag	Description
[XMLStream->Next]	Advances to the next node of the XML stream. Returns True if successful or False if there are no more nodes.
[XMLStream->NextSibling]	Advances to the next sibling node, bypassing any child notes. Returns True if successful or False if there are no more sibling nodes.
[XMLStream->MoveToAttribute]	Moves the position to the specified attribute. Requires a single parameter which is the attribute name to be moved to or an integer index to move to each attribute in order. Returns True if the current node could be changed.
[XMLStream->MoveToAttributeNamespace]	Moves the position to the specified attribute. Requires two parameters. The first parameter is the name of the attribute to be moved to. The second parameter is the URI of the namespace to be used. Returns True if the current node could be changed.
[XMLStream->MoveToFirstAttribute]	Moves the position to the first attribute associated with the current node. Returns True if successful or False if the current node has no attributes.
[XMLStream->MoveToNextAttribute]	Moves the position to the next attribute of the current node. Returns True if successful or False if there are no more attributes of the current node.
[XMLStream->MoveToElement]	Moves the position to the current node. Returns True if successful. This tag can be used to return to the node after moving to one or more attributes.

Navigation through an XML stream occurs only forward through the nodes. Each XML opening tag, closing tag, and other node types is visited in order using [XMLStream->Next]. The nodes are presented in the order they appear in the document without respect for the nesting of XML tags in the docuemnt.

```
<alpha name="value"> Some Text <beta> More Text </beta> </alpha>
```

For example, in the above XML document the following nodes will be visited in order: Starting at the document node. The startElement node representing the opening <alpha> tag. The text node Some Text. The startElement node representing the opening <beta> tag. The text node More Text. The endElement node representing the closing </beta> tag. And finally, the endElement node representing the closing </alpha> tag.

Node Attributes

As each node is visited its attributes can be fetched using one of the member tags detailed in the *XML Stream Member Tags* table. These tags provide tools for inspecting the attributes and contents of a tag.

Table 10: XML Stream Member Tags

Tag	Description
[XMLStream->AttributeCount]	Returns the number of attributes of the current node.
[XMLStream->BaseURI]	Returns the base URI of the current node.
[XMLStream->Depth]	Returns the depth of the current node in the tree.
[XMLStream->GetAttribute]	Returns the value of an attribute of the current node. Requires one parameter which is the name of an attribute or an integer index to retrieve the attributes in order.
[XMLStream->GetAttributeNamespace]	Returns the value of an attribute of the current node. Requires two parameters. The first is the name of a parameter. The second is the URI for a namespace.
[XMLStream->HasAttributes]	Returns True if the current node has any attributes.
[XMLStream->HasValue]	Retturns True if the current node can have a text value.
[XMLStream->isEmptyElement]	Returns True if the current node is empty.
[XMLStream->LocalName]	Returns the local name of the current node.
[XMLStream->LookupNamespace]	Returns the namespace for a prefix. Requires a single parameter which is the prefix to be looked up.
[XMLStream->NodeType]	Returns the type of the current node. The node types are identified in the following table.
[XMLStream->ReadAttributeValue]	Parses an attribute value into one or more Text and EntityReference nodes. Returns True if successful.
[XMLStream->ReadString]	Returns the text of the current node as a string.
[XMLStream->Name]	Returns the qualified name of the current node: "Prefix: LocalName".
[XMLStream->NamespaceURI]	Returns the URI defining the namespace associated with the current node.
[XMLStream->Prefix]	Returns the namespace prefix for the current node.
[XMLStream->Value]	Returns the text value of the current node if present.
[XMLStream->XMLLang]	Returns the xml:lang scope within which the current node resides.

XML Stream Example

The use of the XML stream tags depends largely on what type of XML document needs to be parsed. This example shows how a simple XML structure can be parsed and its attributes output to the browser.

This is the example XML data that will be processed for the example.

```
<xml>
  XML Data
  <tag param="value">
    A Tag
    <sub>A Sub-Tag</sub>
  </tag>
</xml>
```

To prepare to process the XMl document it must be stored in a variable and then an [XMLStream] object is initialized.

```
[var: 'xml' = '<xml> ... </xml>']
[var: 'stream' = (xmlstream: $xml)]
```

The following code advances through the XML stream using [XMLStream->Next]. It prints out various attributes of the current node and then advanced through the node's attributes (if any).

```
<?LassoScript
  while: $stream->next;
    $stream->nodetype + ': ' +
        '"' + $stream->name + '" = "' $stream->value + '"';
    '<br>';
    if: ($stream->attributecount > 0) && ($stream->movetofirstattribute);
      var: 'more' = true;
      while: $more;
        encode_html: loop_count + ' ' + $stream->nodetype + ': ' + '
            '"' + $stream->name + '" = "' $stream->value + '"';
        '<br>';
        var: 'more' = $stream->movetonextattribute;
      /while;
      $stream->(movetoelement);
    /if;

/while;
?>
```

The results of running the code on the example XML document are shown below. Each node is output with its type, name, and value. The node for the opening <tag> has an attribute which is shown with a preceding numeral.

```
➜ startElement: "xml" = ""
  text: "#text" = "XML Data"
  startElement: "tag" = ""
  1 attributes: "param" = "value"
  text: "#text" = "A Tag"
  startElement: "Sub" = ""
  text: "#text" = "A Sub-Tag"
  endElement: "sub" = ""
  endElement: "tag" = ""
  endElement: "xml" = ""
```

This same code can be run on more complex XML documents to see how the XML stream tags report information about the different nodes. By adding actions when certain node types are encountered, this code can also be adapted into a tool that will parse XML and perform actions based on the contents.

Serving XML

XML data which is created within a variable, stored in a database, or read from a file on the Web serving machine can be served in place of the current format file using the [XML_Serve] tag. When this tag is called processing of the current format file is aborted and the specified XML data is served to the site visitor.

The visitor's Web browser will determine how the XML data is formatted. Many Web browsers will show XML data in outline form where the individual tags can be collapsed or expanded to view different portions of the data.

Table 11: [XML_Serve] Serving Tags

Tag	Description
[XML_Serve]	Returns XML data in place of the current format file. The first parameter is the XML data to be served. Optional -File parameter allows the name of the XML data to be specified. Optional -Type parameter allows the MIME type to be overridden from the default of text/xml.

To serve XML data:

Use the [XML_Serve] tag. The following example serves some simple XML data in place of the current format file. No tags after the [XML_Serve] tag will be processed.

```
[Variable: 'XMLData' = '<?xml version="1.0" encoding="UTF-8" ?>
  <ROOT>
    <ROW>
      This is XML data.
    </ROW>
  </ROOT>']

[XML_Serve: $XMLData]
```

Formatting XML

XML data should be served using the MIME type of text/xml and a UTF-8 character set. The [Content_Type] tag can be used to set the MIME type and character set of a page served by Lasso. This tag simply adjusts the header of the page served by Lasso, it does not perform any conversion of the data on the page.

To specify that a format file contains XML:

Use the following tag as the very first line of any files which contain XML data. Notice that the tag accepts only a single parameter, the charset argument is appended to the MIME type argument with a semi-colon ;.

```
[Content_Type: 'text/xml; charset=UTF-8']
```

To format XML:

Most XML pages have the following format, an <?XML ... ?> declaration followed by a root tag that surrounds the entire contents of the file. This is similar to the <html> tag that typically surrounds an entire HTML page. The following example shows a <ROOT> ... </ROOT> tag with a single <ROW> ... </ROW> tag inside.

```
[Content_Type: 'text/xml; charset=UTF-8']
<?xml version="1.0" encoding="UTF-8" ?>
<ROOT>
  <ROW>
    This is XML data.
  </ROW>
</ROOT>
```

To encode data within XML:

The data within XML tags and tag parameters should be XML encoded. The [Encode_Set] ... [/Encode_Set] tags can be used to change the default encoding for all substitution tags in an entire XML page. The following example shows an XML page with an enclosing set of [Encode_Set] ... [/Encode_Set]

tags. The value of the [Variable] tag will be XML encoded, ensuring that it is recognized properly by an XML parser.

```
[Content_Type: 'text/xml; charset=UTF-8']
<?xml version="1.0" encoding="UTF-8" ?>
[Encode_Set: -EncodeXML]
  <ROOT>
    <ROW>
      [Variable: 'XML_Data']
    </ROW>
  </ROOT>
[/Encode_Set]
```

Tags which return XML tags should not have their values encoded. Tags which return XML data require an -EncodeNone encoding keyword in order to ensure that the angle brackets and other markup characters are not encoded into XML entities. The following example shows a variable that returns an entire <ROW> ... </ROW> tag. The [Variable] tag has an -EncodeNone keyword so the angle brackets within the XML data are not encoded.

```
[Content_Type: 'text/xml; charset=UTF-8']
[Variable: 'XML_Data' = '<ROW><p>This is XML data.<ROW>']
<?xml version="1.0" encoding="UTF-8" ?>
[Encode_Set: -EncodeXML]
  <wml>
    [Variable: 'XML_Data', -EncodeNone]
  </wml>
[/Encode_Set]
```

XML Templates

Lasso includes a collection of XML templates that you can incorporate into your own Web site or customize to use a different DTD or schema. In order to use the templates, you must construct a Lasso action which uses the XML template as its response.

The templates are contained in Documentation/4-LanguageGuide/Examples/XML/ folder within the Lasso Professional 8 application folder. In order to use these examples, the entire XML folder should be copied into the Web server root. The examples in this section assume the XML folder can be reached at the root of the Web server by the following URL.

```
http://www.example.com/XML/
```

- **FileMaker Pro** templates allow data to be published using the same formats as those provided with FileMaker Pro Data Source Object (DSO) and FileMaker Pro (FileMaker) templates are provided including versions

with DTDs and schemas. Each of the templates is described in the *FileMaker Pro XML Templates* table. All of the templates are included in the folder FileMaker within the XML folder.

- **SQL Server** templates allow data to be published using some of the formats provided with Microsoft SQL Server. Templates are provided for Raw SQL results and for results structured as Auto Elements (Elem). Each of the templates includes versions with DTDs and schemas. They are described in *Table 19: SQL Server XML Templates*. All of the templates are included in the folder SQLServer within the XML folder.

Each template can be used as the response to a Lasso action that returns records. The templates are written in a database-independent fashion and build their DTD or schema based on the actual field names which define the results that they are formatting.

Table 12: FileMaker Pro XML Templates

Template	Description
dso_xml.lasso	Data Source Object template uses the name of each field in the database as the name of an XML tag. Root tag is <FMPDSORESULT> which contains <ROW> tags that contain individual field tags.
dso_xml_dtd.lasso	Includes a dynamically generated DTD.
dso_xml_schema.lasso	Includes a dynamically generated Schema.
fmp_layout_xml.lasso	FileMaker Pro Layout template includes <LAYOUT>, <FIELD>, and <VALUELIST> tags which describe a FileMaker Pro layout. Root tag is <FMPXMLLAYOUT> which contains <ERRORCODE> <PRODUCT> and <LAYOUT> tags.
fmp_layout_xml_dtd.lasso	Includes a dynamically generated DTD.
fmp_layout_xml_schema.lasso	Includes a dynamically generated Schema.
fmp_xml.lasso	FileMaker Pro results template includes database structure and <RESULTS> tag with <ROW> and <COL> sub-tags. Root tag is <FMPXMLRESULT> which contains <ERRORCODE> <PRODUCT> <DATABASE>, <METADATA> and <RESULTS> tags.
fmp_xml_dtd.lasso	Includes a dynamically generated DTD.
fmp_xml_schma.lasso	Includes a dynamically generated Schema.

Table 13: SQL Server XML Templates

Template	Description
sql_xml_raw.lasso	Includes each record in a single <ROW> tag. Root tag is <ROOT> which contains <ROW> tags. Each field is specified as a parameter of the <ROW> tags.
sql_xml_raw_dtd.lasso	Includes a dynamically generated DTD.
sql_xml_raw_schema.lasso	Includes a dynamically generated Schema.
sql_xml_elem.lasso	Includes each record in a single <ROW> tag. Root tag is <ROOT> which contains <ROW> tags. Each field is specified as a tag named the same as the field name within the <ROW> tags.
sql_xml_elem_dtd.lasso	Includes a dynamically generated DTD.
sql_xml_elem_schema.lasso	Includes a dynamically generated Schema.

To use a template with an [Inline] ... [/Inline] action:

Specify a search within [Inline] ... [/Inline] tags and use an [Include] tag to insert the desired template to format the results. A [Content_Type] tag is required at the top of the file containing the [Inline] ... [/Inline] tags so the MIME type of the returned data will be properly specified. The following example finds all records in a Contacts database and formats the results using the FileMaker DSO template stored at /XML/FileMaker/dso_xml.lasso.

```
[Content_Type: 'text/xml; charset=UTF-8']
[Inline: -Database='Contacts', -Table='People', -KeyField='ID', -FindAll]
  [Include: '/XML/FileMaker/dso_xml.lasso']
[/Inline]
```

To use a template with an HTML form-based action:

Specify a search within an HTML form. The -Response to the form should be a template file. The following example finds all records in a Contacts database and formats the results using the FileMaker DSO template stored at /XML/FileMaker/dso_xml.lasso.

```
<form action="/Action.Lasso" method="POST">
  <input type="hidden" name="-Database" value="Contacts">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-KeyField" value="ID">
  <input type="hidden" name="-Response"
      value="/XML/FileMaker/dso_xml.lasso">
  <input type="submit" name="-FindAll" value="Find All Contacts">
</form>
```

To use a template with a URL-based action:

Specify a search within a URL. The -Response specified in the URL should be a reference to a template file. The following example finds all records in a Contacts database and formats the results using the FileMaker DSO template stored at /XML/FileMaker/dso_xml.lasso.

```
<a href="/Action.lasso?-Datasource=Contacts&
    -Table=People&
    -KeyField=ID&
    -Response=/XML/FileMaker/dso_xml.lasso&
    -FindAll">
  Find All Contacts
</a>
```

How to Customize

The XML templates are good examples of data source-independent design and can be used as the starting point of an automatic publishing system based on XML.

The templates are also a good starting point to create XML format files that are industry specific or to a particular database structure. Starting with a DTD or schema, an XML format file can be created that outputs data from a data source in precisely the format required.

Custom XML templates will need to be created in order to take advantage of relationships, repeating fields, stored images, or portals specific to any given data source.

33

Chapter 33
Portable Document Format

This chapter describes how to create files in Portable Document Format (PDF) using LDML 8.

- *Overview* introduces PDF support in Lasso Professional 8.
- *Creating PDF Documents* describes how to create PDF documents using LDML tags, and how to use existing PDF documents as templates.
- *Creating Text Content* describes how to add text to a PDF variable using LDML tags.
- *Creating and Using Forms* describes how to add forms to a PDF variable using LDML tags, and also discusses how PDF forms can be used to submit data to a database using Lasso.
- *Creating Tables* describes how to create and insert tables in a PDF variable using LDML tags.
- *Creating Graphics* describes how to create and insert graphics in a PDF variable using LDML tags.
- *Creating Barcodes* describes how to create and insert barcodes in a PDF variable using LDML tags.
- *Example PDF Files* provides complete examples of using LDML to create PDF files with text, forms, tables, graphics, and barcodes.
- *Serving PDF Files* describes how to display a PDF file within the context of a Lasso format file.

Overview

Lasso Professional 8 provides support for Portable Document Format (PDF) files, allowing PDF documents to be created using LDML. The PDF file format is a widely-accepted standard for electronic documentation, and facilitates superb printer-quality documents from simple graphs to complex forms such as tax forms, escrow documents, loan applications, stock reports, and user manuals. For more information on PDF technology, see the following URL.

http://www.adobe.com/products/acrobat/adobepdf.html

Implementation Note: The [PDF_...] tags in LDML 8 are implemented in LJAPI, and based on the iText Java library. For more information on the iText Java library, visit http://www.lowagie.com/iText.

Introduction to Creating PDF Files

PDF files are created in LDML by setting a variable as a [PDF_Doc] object, and using various member tags and other [PDF_...] tags to add data to the variable. The PDF is then written to file when the format file containing all code is served by the Web server.

To create a basic PDF file using LDML:

The following shows an example of creating and outputting a PDF file named MyFile.pdf using the [PDF_...] tags.

```
[Var:'MyFile'=(PDF_Doc: -File='MyFile.pdf',
                        -Size='A4',
                        -Margin=(Array: 144.0, 144.0, 72.0, 72.0))]
[Var: 'Font'=(PDF_Font: -Face='Helvetica',  -Size=36)]
[Var: 'Text'=(PDF_Text:'I am a PDF document', -Font=$Font)]
[$MyFile->(Add: $Text)]
[$MyFile->Close]
```

In the example above, a variable named MyFile is set to a [PDF_Doc] type for a file named MyFile.pdf. A single font type is defined for the document using the [PDF_Font] tag. Then, the text I am a PDF document is defined using the [PDF_Text] tag, and added using the [PDF_Doc->Add] member tag. The PDF is then written to file upon execution of the [$MyFile->Close] tag.

This chapter explains in detail how these and other tags are used to create and edit PDF files. This chapter also shows how to output a PDF file to a client browser within the context of a format file, which is described in the *Serving PDF Files* section of this chapter.

File Permissions

This section describes the file permission requirements for creating PDF files on a Web server using LDML 8. In order to successfully create PDF files, the following conditions must be met.

- When creating PDF files using the [PDF_...] tags, the current user must have Create Files, Read Files, and Write Files permissions allowed in the *Setup > Security > Files* section of Lasso Administration, and the folder in which the PDF will be created must be available to the user within the Allow Path field.
- Any file extensions being used by the [PDF_...] tags must be allowed in the *Setup > Global Settings > Settings* section of Lasso Administration. This can include .pdf, .jpg, and .gif.
- When creating files, Lasso Service must be allowed to write to the folder by the operating system (i.e. the Lasso system user in Mac OS X and Linux). For more information, see the *Files and Logging* chapter.

Creating PDF Documents

PDF documents are initialized and created using the [PDF_Doc] tag. This is the basic tag used to create PDF documents with Lasso, and is used in concert with all tags described in this chapter.

Table 1: [PDF_Doc] Tag and Parameters

Tag	Description
[PDF_Doc]	Initializes a PDF document. Uses optional parameters which set the basic specifications of the file to be created. Data is added to the variable using [PDF_Doc] member tags, which are described throughout this chapter.
-File	Defines the file name and path of the PDF document. If omitted, the PDF document is created in RAM (see the *Serving PDF Files* section of this chapter for more information). If a file name is specified without a folder path, the file is created in the same location as the format file containing the [PDF_...] tags.
-Size	Define the page size of the document. Values for this parameter are standard print sizes, and can be A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, B0, B1, B2, B3, B4, B5, ARCH_A, ARCH_B, ARCH_C, ARCH_D, ARCH_E, FLSA, FLSE, HALFLETTER, LEDGER, LEGAL, LETTER, NOTE, and TABLOID. Defaults to A4 if not used. Optional.

-Height	Defines a custom page height for the document. Accepts a decimal value which represents the size in points. This can be used with the -Width parameter instead of the -Size parameter. Optional.
-Width	Defines a custom page width for the document. Requires a decimal value which represents the size in points. This can be used with the -Height parameter instead of the -Size parameter. Optional.
-Margins	Defines the margin size for the page. Requires an array of four decimal values, which define the left, right, top, and bottom margins for the page (Left, Right, Top, Bottom). Optional.
-Color	Defines the initial text color of the PDF document. Requires a hex color string. Defaults to '#000000' if not used. Optional.
-UseDate	Adds the current date and time to the file header. Optional.
-NoCompress	Produces a PDF without compression to allow PDF code to be viewed. PDF files are compressed by default if not used. Optional.
-PageNo	Sets the starting page number for the PDF document. Requires an integer value, which is the page number of the first page. Optional.
-PageHeader	Sets text that will be displayed at the top of each page in the PDF. Requires a text string as a value. Optional.
'Header'='Content'	Adds defined file headers to the PDF document. 'Header' is replaced with the name of the file header (e.g. Title, Author), and 'Content' is replaced with the header value. Optional.

The examples below show creating basic PDF files, however these files contain little or no data. Various types of data can be added to these files using the tags described in the remainder of this chapter.

To start a basic PDF file:

Use the [PDF_Doc] tag to create a PDF file to a hard drive location on the Web server. Use the -File parameter to define the location and file name, and the -Size parameter to define a pre-defined standard size. This basic example creates a blank, one-page PDF document.

```
[Var:'MyFile'=(PDF_Doc: -File='MyFile.pdf', ,
                        -Size='A4')]
```

To start a PDF file with a custom page size:

Use the [PDF_Doc] tag with the -Height and -Width parameters to define a custom page size in points. One inch is equal to 72 points.

```
[Var:'MyFile'=(PDF_Doc: -File='MyFile.pdf',
                        -Height='648.0',
                        -Width='468.0')]
```

To start a PDF file with custom margins:

Use the [PDF_Doc] tag with the -Margin parameter to define a custom page size (in points). The following example adds a margin of 72 points (one inch) to the left and right sides of the page, but adds no margin to the top and bottom. This example also adds the date and time of creation to the file header using the -UseDate parameter.

```
[Var:'MyFile'=(PDF_Doc: -File='MyFile.pdf',
                        -Size='A4',
                        -Margin=(Array: 72.0, 72.0, 0.0, 0.0),
                        -UseDate)]
```

To start an uncompressed PDF file:

Use the [PDF_Doc] tag with the -NoCompress parameter.

```
[Var:'MyFile'=(PDF_Doc: -File='MyFile.pdf',
                        -Size='A4',
                        -NoCompress)]
```

To start a PDF file with custom file headers:

Use the [PDF_Doc] tag with appropriate 'Header'='Content' parameters.

```
[Var:'MyFile'=(PDF_Doc: -File='MyFile.pdf',
                        -Size='A4',
                        'Title'='My PDF File',
                        'Subject'='How to create PDF files',
                        'Author'='John Doe')]
```

Adding Content to PDFs

In LDML 8, there are several different types of data that can be added to a PDF document. Many of these types are first defined as objects using tags such as [PDF_Text], [PDF_List], [PDF_Image], [PDF_Table], or [PDF_BarCode], and then added to a [PDF_Doc] variable using the [PDF_Doc->Add] member tag. Each data type object is described separately in subsequent sections of this chapter.

Table 2: [PDF_Doc->Add] Tag and Parameters

Tag	Description
[PDF_Doc->Add]	Adds a PDF content object to a document. This can be used to add [PDF_Text], [PDF_List], [PDF_Image], [PDF_Table], or [PDF_BarCode] objects. If no position information is specified then the object is added to the flow of the page, otherwise it is drawn at the specified location. Requires one parameter, which is the object to be added. Optional parameters are described below.
-Align	Sets the alignment of the object in the page ('Left', 'Center', or 'Right'). Defaults to 'Left'. Works only for [PDF_Image] and [PDF_BarCode] objects. Optional.
-Wrap	Keyword parameter specifies that text should flow around the embedded object. Works only for [PDF_ Image] and [PDF_BarCode] objects. Optional.
-Left	Specifies the placement of the object relative to the left side of the document. Requires a decimal value, which is the placement offset in points. Works only for [PDF_Image] and [PDF_BarCode] objects. Optional.
-Top	Specifies the placement of the object relative to the top of the document. Requires a decimal value, which is the placement offset in points. Works only for [PDF_Image] and [PDF_BarCode] objects. Optional.
-Height	Scales the object to the specified height. Requires a decimal value which is the desired object height in points. Works only for [PDF_Image] and [PDF_BarCode] objects. Optional.
-Width	Scales the object to the specified width. Requires a decimal value which is the desired object width in points. Works only for [PDF_Image] and [PDF_BarCode] objects. Optional.

For examples of using the [PDF_Doc->Add] tag to add text, image, table, and barcode PDF objects to a [PDF_Doc] variable, see the corresponding sections in this chapter.

Adding Pages

If the content of a PDF document will span more than one page, additional pages can be added using special [PDF_Doc] member tags. These tags signal where pages start and stop within the flow of the LDML PDF creation tags.

Table 3: PDF Page Tags

Tag	Description
[PDF_Doc->AddPage]	Adds additonal blank pages to the [PDF_Doc] variable. When used, this tag ends in the current page and starts a new page.
[PDF_Doc->AddChapter]	Adds a page with a named chapter title (and bookmark) to a [PDF_Doc] variable. Requires a text string or [PDF_Text] object as a parameter, which specifies the chapter title. An additonal -Number parameter sets an integer chapter number for the chapter. An optional -HideNumber parameter specifies that no number will be shown.
[PDF_Doc->SetPageNumber]	Sets a page number for a new page. Requires an integer value.
[PDF_Doc->GetPageNumber]	Returns the current page number.

To start a new page:

Use the [PDF_Doc->AddPage] tag. The following example ends a preceding page, and starts a new page.

```
[$MyFile->(Add:'Thus, ends the discussion on page 1.')]
[$MyFile->AddPage]
[$MyFile->(Add:'On page 2, we will discuss something else.')]
```

To add a chapter title:

Use the [PDF_Doc->AddChapter] tag. The following example adds a page with the text 30. Important Chapter to the [PDF_Doc] variable with a defined chapter number of 30.

```
[$MyFile->(AddChapter:'Important Chapter', -Number=30)]
```

To set the page number for a page:

Use the [PDF_Doc->SetPageNumber] tag. The following example sets a page number of 5 for the current page.

```
[$MyFile->(SetPageNumber: 5)]
```

To return the current page number:

Use the [PDF_Doc->GetPageNumber] tag. The following example returns a page number of 1 when used within the first page of the document.

```
[$MyFile->GetPageNumber] ➜ 1
```

Adding Pages from Existing PDFs

Pages in existing PDF documents can be added to a [PDF_Doc] variable using the [PDF_Read] tag. This tag makes it possible to use existing PDF documents as templates.

Note: Lasso cannot change existing text or graphics that are contained within a PDF document read in using [PDF_Read]. Instead, Lasso is able to overlay text, graphics, and other elements on the PDF.

Table 4: PDF Read Tags

Tag	Description
[PDF_Read]	Casts an existing PDF document on the server as an LDML object. This object can be added to a [PDF_Doc] variable using the [PDF_Doc->InsertPage] tag, and then modified. Requires a -File parameter, which specifies the name and path to a PDF file on the server to read.
[PDF_Read->PageCount]	Returns the integer number of pages in the document.
[PDF_Read->PageSize]	Returns the dimensions of the first page in the PDF as an array of width and height point values (Array: Width, Height). An optional integer parameter specifies the page number in the document to return the size of.

To read in an existing PDF document:

In order to work with an existing PDF document, it must first be cast as a Lasso variable using the [PDF_Read] tag.

```
[Var:'Old_PDF'=(PDF_Read:-File='/documents/somepdf.pdf')]
```

To determine the attributes of an existing PDF document:

The number of pages and the dimensions of an existing PDF document can be returned using the [PDF_Read->PageCount] and [PDF_Read->PageSize] tags on a defined [PDF_Read] variable.

```
[Var:'Old_PDF'=(PDF_Read:-File='/documents/somepdf.pdf')]
Number of pages: [$Old_PDF->PageCount]<br>
Page size: [$Old_PDF->(PageSize: 1)]
```

Once an existing PDF document has been cast as a Lasso object using [PDF_Read], it may be added to a [PDF_Doc] variable using the [PDF-Doc->InsertPage] tag.

Table 5: Page Insertion Tag and Parameters

Tag	Description
[PDF_Doc->InsertPage]	Inserts a page from a [PDF_Read] object into a [PDF_Doc] variable. Requires the name of a [PDF_Read] variable, followed by a comma and the number of the page to insert. This tag has many optional parameters for specifying how an existing page should be insterted into a [PDF_Doc] variable. These parameters are explained below.
-NewPage	Keyword parameter specifying that the new page should be appended at the end of the document. Otherwise the page is drawn over the first page in the [PDF_Doc] variable by default.
-Top	If the page being inserted is shorter than the current pages in the [PDF_Doc] variable, this parameter may be used to specifiy the offset of the new page from the top of the current page frame in points.
-Left	If the page being inserted is not as wide the current pages in the [PDF_Doc] variable, this parameter may be used to specifiy the offset of the new page from the left of the current page frame in points.
-Width	Scales the inserted page by width. Requires either a point width value, or a percentage string (e.g. 50%).
-Height	Scales the inserted page by height. Requires either a point height value, or a percentage string (e.g. 50%).

To insert an existing page into a new PDF document:

Use the [PDF_Doc->InsertPage] tag with a defined [PDF_Read] variable. The example below makes the first page of the somepdf.pdf PDF the first page of the [PDF_Doc] variable. Content may then be overlaid on top of the new page using the tags described in the rest of this chapter.

```
[Var:'New_PDF'=(PDF_Doc: -File='MyFile.pdf',
                         -Size='A4')]
[Var:'Old_PDF'=(PDF_Read:-File='/documents/somepdf.pdf')]
[$New_PDF->(InsertPage: $Old_PDF, 1)]
```

To insert an existing page at the end of a new PDF document:

Use the [PDF_Doc->InsertPage] tag with the optional -NewPage parameter. The example below adds the first page of the somepdf.pdf PDF after all existing pages in the [PDF_Doc] variable.

```
[Var:'New_PDF'=(PDF_Doc: -File='MyFile.pdf',
                         -Size='A4')]
[Var:'Old_PDF'=(PDF_Read:-File='/documents/somepdf.pdf')]
[$New_PDF->(InsertPage: $Old_PDF, 1, -NewPage)]
```

To place an inserted page:

Use the [PDF_Doc->InsertPage] tag with the optional -Top and/or -Width parameters. The example below places the inserted page 50 points away from the top and left sides of the new document page frame.

```
[Var:'New_PDF'=(PDF_Doc: -File='MyFile.pdf',
                         -Size='A4')]
[Var:'Old_PDF'=(PDF_Read:-File='/documents/somepdf.pdf')]
[$New_PDF->(InsertPage: $Old_PDF, 1, -Width=50, -Height=50)]
```

Accessing PDF File Information

Parameter values of a [PDF_Doc] variable can be returned using special accessor tags. These tags return specific values such as the page size, margin size, or the value of any other [PDF_Doc] variable described in the previous section. All PDF accessor tags in LDML 8 are defined in *Table 6: PDF Accessor Tags*.

Table 6: PDF Accessor Tags

Tag	Description
[PDF_Doc->GetMargins]	Returns the current page margins as an array data type (Array: Left, Right, Top, Bottom).
[PDF_Doc->GetSize]	Returns the current page size as an array of width and height point values (Array: Width, Height).
[PDF_Doc->GetColor]	Returns the current color as a hex string.
[PDF_Doc->GetHeaders]	Returns all document headers as a map data type (Map: 'Header1'='Content1', 'Header2'='Content2', ...).
[PDF_Doc->SetFont]	Sets a font for all following text. The value is a [PDF_Font] object.

To return PDF page margins:

Use the [PDF_Doc->GetMargins] tag. The following example returns the current margins of a defined [PDF_Doc] variable.

```
[$MyFile->GetMargins] ➜ (Array: 72.0, 72.0, 72.0, 72.0)
```

To return a PDF page size:

Use the [PDF_Doc->GetSize] tag. The following example returns the current sizes of a defined [PDF_Doc] variable.

```
[$MyFile->GetSize] ➜ (Array: 468.0, 648.0)
```

To return a PDF base font color:

Use the [PDF_Doc->GetColor] tag. The following example returns the base font color of a defined [PDF_Doc] variable.

```
[$MyFile->GetColor] ➜ #333333
```

Saving PDF Files

Once a [PDF_Doc] variable has been filled with the desired content, the [PDF_Doc->Close] tag must be used to signal that the PDF file is finished and is ready to be written to file or served.

Table 7: [PDF_Doc->Close] Tag

Tag	Description
[PDF_Doc->Close]	Closes [PDF_Doc] variable and commits it to file after all desired data has been added to it. Additional data may not be added to the specified variable after this tag is used.

To close a PDF file:

Use the [PDF_Doc->Close] tag after all desired modifications have been performed on the [PDF_Doc] variable.

```
[Var:'MyFile'=(PDF_Doc: -File='MyFile.pdf',
                        -Size='A4',
                        -Margin=(Array: 144.0, 144.0, 72.0, 72.0))]
[Var: 'Font'=(PDF_Font: -Face='Helvetica', -Size=36)]
[Var: 'Text'=(PDF_Text:'I am a PDF document', -Font=$Font)]
[$MyFile->(Add: $Text)]
[$MyFile->Close]
```

Creating Text Content

Text content is the most basic type of data within a PDF document. PDF text is first defined as a [PDF_Text] object, and then added to a PDF variable using the [PDF_Doc->Add] tag.

[PDF_Text] objects may be positioned within the current PDF page using the -Left and -Top parameters of the [PDF_Doc->Add] tag. Otherwise, if no positioning parameters are specified, the text will be added to the top left corner of the page by default.

Using Fonts

Before adding text, it is important to first define the font and style for the text to determine how it will appear. This is done using the [PDF_Font] tag.

Table 8: PDF Font Tag and Parameters

Tag	Description
[PDF_Font]	Stores all the specifications for a font style. This include font family, size, style, and color. Parameters are used with the [PDF_Font] tag that define the font family, size, color, and specifications. The following parameters may be used with the [PDF_Font] tag.
-Face	Specifies the font by its family name. Allowed font names are Courier, Courier-Bold, Courier-Oblique, Courier-BoldOblique, Helvetica, Helvetica-Bold, Helvetica-Oblique, Helvetica-BoldOblique, Symbol, Times-Roman, Times-Bold, Times-Italic, Times-BoldItalic, and ZapfDingbats.
-File	Creates a font from a local font file. The file name and path to the font must be specified (e.g /Fonts/Courier.ttf). This parameter may be used instead of the -Face parameter. Optional.
-Size	Sets the font size in points. Requires an integer point value as a parameter (e.g 14). Optional.
-Color	Sets the font color. Require a hex color string as a parameter (e.g '#550000'). Defaults to '#000000' if not used. Optional.
-Encoding	Sets the desired font encoding. The font encoding defaults to 'CP1252' if not specified. TrueType fonts can be asked to return an array of supported encodings via the [PDF_Font->GetSupportedEncodings] member tag. Optional.
-Embed	Embeds the fonts used within the PDF document as opposed to relying on the client PDF reader for font information. Optional.

The following examples show how to set variables as [PDF_Font] types that define the font styles that are used in a PDF document.

To set a basic font style:

Set a variable as a [PDF_Font] tag. The following example sets a style to be a standard Helvetica font with a size of 14 points. The font color is green.

```
[Var:'Font1'=(PDF_Font: -Face='Helvetica',
                        -Size=14,
                        -Color='#005500')]
```

Individual parameters may be viewed and changed in a [PDF_Font] variable using [PDF_Font] member tags. These parameters are most useful for retrieving information about a [PDF_Font] object that was defined using the -File parameter, and are summarized in *Table 9: [PDF_Font] Member Tags*.

Table 9: [PDF_Font] Member Tags

Tag	Description
[PDF_Font->SetFace]	Changes the font face of the [PDF_Font] variable to one of the allowed font names.
[PDF_Font->SetColor]	Changes the font color of the [PDF_Font] variable.
[PDF_Font->SetSize]	Changes the font size of the [PDF_Font] variable.
[PDF_Font->SetEncoding]	Changes the encoding of the [PDF_Font] variable.
[PDF_Font->SetUnderline]	Sets the [PDF_Font] variable style to underlined. Requires a boolean parameter of 'True' if used.
[PDF_Font->GetFace]	Returns the current font face of a [PDF_Font] variable.
[PDF_Font->GetColor]	Returns the current font color of a [PDF_Font] variable.
[PDF_Font->GetSize]	Returns the current font size of a [PDF_Font] variable.
[PDF_Font->GetEncoding]	Returns the current encoding of a [PDF_Font] variable.
[PDF_Font->GetPSFontName]	Returns the exact Postscript font name of the current font of a [PDF_Font] variable (e.g. AdobeCorIDMinBd).
[PDF_Font->IsTrueType]	Returns True if the current font is a True Type font.
[PDF_Font->GetSupportedEncodings]	Returns an array of all supported encodings for a current True Type font face (Array:'1252 Latin 1','1253 Greek').

[PDF_Font->GetFullFontName]	Returns the full True Type name of the current font of a [PDF_Font] variable (e.g Comic Sans MS Negreta).
[PDF_Font->TextWidth]	Returns an integer value representing how wide (in pixels) the text would be using the current [PDF_Font] variable. Requires a string value, which is the text to return the width of.

To change a font face:

Use the [PDF_Font->SetFace] tag. The following example sets a defined [PDF_Font] variable to a standard Courier font.

```
[$MyFont->(SetFace:'Courier')]
```

To change a font color:

Use the [PDF_Font->SetColor] tag. The following example sets a defined [PDF_Font] variable to the color red.

```
[$MyFont->(SetColor:'#990000')]
```

To underline a font:

Use the [PDF_Font->SetUnderline] tag. The following example sets a predefined [PDF_Font] variable to use an underlined style.

```
[$MyFont->(SetUnderline: 'True')]
```

To return a font face:

Use the [PDF_Font->GetFace] tag. The following example returns the current font face of a defined [PDF_Font] variable.

```
[$MyFont->GetFace] ➜ Courier
```

To return a font encoding:

Use the [PDF_Font->GetEncoding] tag. The following example returns the encoding of the current font face of a defined [PDF_Font] variable.

```
[$MyFont->GetEncoding] ➜ CP1252
```

Adding Text

PDF text content is constructed using the [PDF_Text] tag, which is then added to a [PDF_Doc] variable using the [PDF_Doc->Add] tag. The [PDF_Text] constructor tag and parameters are described below.

Table 10: [PDF_Text] Tag and Parameters

Tag	Description
[PDF_Text]	Creates a text object to be added to a [PDF_Doc] variable. Requires the text string to be added to the PDF document as the first parameter. Optional parameters are listed below.
-Type	Specifies the text type. This can be 'Chunk', 'Phrase', or 'Paragraph'. Different parameters are available for each of these types, as described below. Defaults to the 'Paragraph' type if no -Type parameter is specified. Optional.
-Color	Sets the font color. Requires a hex color string as a parameter (e.g '#550000'). Defaults to '#000000' if not used. Optional.
-BackgroundColor	Sets the text background color. Require a hex color string as a parameter (e.g '#550000'). Optional.
-Underline	Keyword parameter underlines the text. Optional.
-TextRise	Sets the baseline shift for superscript. Requires a decimal value that specifies the text rise in points. Optional.
-Font	Sets the font for the specified text. The value is a [PDF_Font] variable, which is described in the *Using Fonts* section of this chapter. The font defaults to the current inherited font if no -Font parameter is specified.
-Anchor	Links the specified text to a URL. The value of the parameter is the URL string (e.g. 'http://www.example.com'). Optional.
-Name	Sets the name of an anchor destination within a page. The value of the parameter is the anchor name (e.g. 'Name'). Optional.
-GoTo	Links the specified text to a local anchor destination to go to. The value of the parameter is the local anchor name (e.g. 'Name'). Optional.
-File	Links the specified text to a PDF document. The value of the parameter is a PDF file name (e.g. 'Somefile.pdf'). The -Goto parameter can be used concurrently to specify an anchor name within the destination document. Optional.
-Leading	Sets the paragraph leading space in points (the space above and below the text), and requires a decimal value. For 'Phrase' and 'Paragraph' types only.
-Align	Sets the alignment of the text in the page ('Left', 'Center', or 'Right'). Optional.

-IndentLeft	Sets the left indent of the text object. Requires a decimal value which is the number of points to indent the text. Optional. Available for 'Paragraph' types only.
-IndentRight	Sets the right indent of the text object. Requires a decimal value which is the number of points to indent the text. Optional. Available for 'Paragraph' types only.

The following examples show how to add text to a defined PDF variable named MyFile that has been initialized previously using the [PDF_Doc] tag.

To add a chunk of text:

Use the [PDF_Text] tag with the -Type='Chunk' parameter. The following example adds the text OmniPilot to the [PDF_Doc] variable with a predefined font. The text is positioned in the top left corner of the page by default.

```
[Var:'Text'=(PDF_Text:'OmniPilot', -Type='Chunk', -Font=$MyFont)]
[$MyFile->(Add: $Text)]
```

To add a paragraph of text:

Use the [PDF_Text] tag with the -Type='Paragraph' parameter. The following example adds three sentences of text to the [PDF_Doc] variable with a predefined font.

```
[Var:'Text'=(PDF_Text:'The mysterious file cabinet in orbit has been successfully
lassoed. The file cabinent had been traveling at a velocity of 300 meters per second.
Top scientists suspect that the cabinet had been in orbit for some time.',
-Type='Paragraph', -Font=$MyFont, -Leading=10.0, -IndentLeft=20.0)]
```

To add a linked phrase:

Use the [PDF_Text] tag with the -Anchor parameter. The following example adds the text Click here to go somewhere to the [PDF_Doc] variable with a predefined font, and links the phrase to http://www.example.com.

```
[Var:'Text'=(PDF_Text:'Click here to go somewhere', -Type='Chunk', -Font=$MyFont,
-Anchor='http://www.example.com', -Underline=true)]
[$MyFile->(Add: $Text, -Left=100.0, -Top=100.0)]
```

Adding Floating Text

Instead of adding text to the flow of the page, text can also be positioned on a page using the [PDF_Doc->DrawText] tag. The [PDF_Doc->Drawtext] tag accepts coordiates that allow the text to be placed at an absolute position on the page.

Table 11: [PDF_Doc->DrawText] Tag

Tag	Description
[PDF_Doc->DrawText]	Adds specified text that is positioned on a page using point coordinates. A required -Leading parameter sets the text leading space in points (the space above and below the text), and requires a decimal value. A -Left parameter specifies the placement of the left side of the text from the left side of the page in points, and a -Top parameter specifies the placement of the bottom of the image from the bottom of the page in points (decimal value).

To add floating text:

Use the [PDF_Doc->DrawText] tag. The following example adds the text Some floating text to the [PDF_Doc] variable with a predefined font at the coordinates specified in the -Top and -Left parameters. The coordinates represent the distance in points from the lower and left sides of the page.

```
[$MyFile->(DrawText:'Some floating text', -Font=$MyFont,
                                          -Left=144.0, -Top=480.0)
```

Adding Lists

A list of items can be constructed using the [PDF_List] tag, which can be added to a [PDF_Doc] variable. The [PDF_List] constructor tag and parameters are described below.

Table 12: [PDF_List] Tags and Parameters

Tag	Description
[PDF_List]	Creates a list object to be added to a [PDF_Doc] variable. Text list items are added to this object using the [PDF_List->Add] tag. Optional parameters for this object are described below.
-Format	Specifies whether the list is numbered, lettered, or bulleted. Requires a value of 'Number', 'Letter', 'Bullet'. Defaults to 'Bullet' if no -Format parameter is specified. Optional.
-Bullet	Specifies a custom character to use as the bullet character. Requires a character as a parameter (e.g. 'x'). Defaults to '•' if not specified. Optional.
-Indent	Sets the space between the bullet and the list item. Requires a decimal or integer parameter which is the width of the indentation in points. Optional.

-Font	Sets the font for the specified text. The value is a [PDF_Font] variable, which is described in the *Using Fonts* section of this chapter. The font defaults to the current inherited font if no -Font parameter is specified.
-Align	Sets the alignment of the list in the page ('Left', 'Center', or 'Right'). Optional.
-Color	Sets the font color. Requires a hex color string as a parameter (e.g '#550000'). Defaults to '#000000' if not used. Optional.
-BackgroundColor	Sets the text background color. Require a hex color string as a parameter (e.g '#550000'). Optional.
-Leading	Sets the list leading space in points (the space above and below the text), and requires a decimal value. Optional.
[PDF_List->Add]	Add objects to the list. Requires a text string or a [PDF_Text] object as a parameter.

To add a numbered list:

Use the [PDF_List] tag with the -Format='Number' parameter to define the list, and the [PDF_List->Add] tag to add items to the list. The example below creates a numbered list with three items.

```
[Var:'List'=(PDF_List: -Format='Number', -Align='Center', -Font=$MyFont)]
[$List->(Add:'This is item one')]
[$List->(Add:'This is item two')]
[$List->(Add:'This is item three')]
[$MyFile->(Add: $List, -Top=400.0)]
```

To add a bulleted list:

Use the [PDF_List] tag with the -Format='Number' parameter to define the list, and the [PDF_List->Add] tag to add items to the list. The example below adds a numbered list with four items, where a hyphen (-) is used as the bullet character.

```
[Var:'List'=(PDF_List: -Format='Bullet', -Bullet='-', -Font=$MyFont)]
[$List->(Add:'This is item one')]
[$List->(Add:'This is item two')]
[$List->(Add:'This is item three')]
[$List->(Add:'This is item four')]
[$MyFile->(Add: $List, -Top=400.0)]
```

Special Characters

When adding text to a [PDF_Doc] object, special characters can be used to designate lines breaks, tabs, and more. These characters are summarized in *Table 13: Special Characters*.

Table 13: Special Characters

Character	Description
\n	Line break character (Mac OS X and Linux).
\r\n	Line break character (Windows).
\t	Tab character.
\"	Double quote character.
\'	Single quote character.
\\	Backslash character.

To use special characters in a text string:

The following example shows how to use special characters within a [PDF_Doc] text tag.

```
[$MyFile->(Add: '\\ \t \'Single Quotes\', \"Double Quotes\" \t \\', -Font=$MyFont)]
```

Creating and Using Forms

Forms can be created in PDF documents for submitting information to a Web site. PDF forms use the same attributes as HTML forms, making them useful for submitting information to a Web site in place of an HTML form. This section describes how to create form elements within a PDF file, and also how PDF forms can be used to submit data to a Lasso-enabled database.

Note: Due to the iText implementation of PDF support in Lasso Professional 8, PDF documents created may contain one form only.

Creating Forms

Form elements are created in [PDF_Doc] variables using [PDF_Doc] form member tags, which are listed in *Table 14: [PDF_Doc] Form Member Tags*.

Table 14: [PDF_Doc] Form Member Tags

Tag	Description
[PDF_Doc->AddTextField]	Adds a text field to a form. A required -Name parameter specifies the name of the text field, and a required -Value parameter specifies the default value entered. A required -Font parameter is used to specify a [PDF_ Font] variable forthe text font.
[PDF_Doc->AddPasswordField]	Adds a password field to a form. A required -Name parameter specifies the name of the password field, and a required -Value parameter specifies the default value entered. A required -Font parameter is used to specify a [PDF_Font] variable for the text font.
[PDF_Doc->AddTextArea]	Adds a text area to a form. A required -Name parameter specifies the name of the text area, and a required -Value parameter specifies the default value entered. A required -Font parameter is used to specify a [PDF_ Font] variable for the text font.
[PDF_Doc->AddCheckBox]	Adds a check box to a form. A required -Name parameter specifies the name of the checkbox, and a required -Value parameter specifies the value for the checkbox. An optional -Checked parameter specifies that the checkbox is checked by default.
[PDF_Doc->AddRadioGroup]	Adds a radio button group to a form. A required -Name parameter specifies the name of the radio button group. Radio buttons must be assigned to the group using the [PDF_Doc->AddRadioButton] tag.
[PDF_Doc->AddRadioButton]	Adds a radio button to a form. A required -Group parameter specifies the name of the radio button group, and a required -Value parameter specifies the value of the radio button.
[PDF_Doc->AddComboBox]	Adds a pull-down menu to a form. A required -Name parameter specifies the name of the pull-down menu, and a required -Values parameter specifies the array of values contained in the menu (Array: 'Value1', 'Value2'). An -Options parameter may be used instead of the -Values parameter that specifies a pair for each value. The first element in the pair is the value to be used upon form submission, and the second element is the human-readable label to be used for display only. An optional -Default parameter specifies the name of a default value selected. An optional -Editiable parameter specifies that the user may edit the values on the menu. A required -Font parameter is used to specify a [PDF_Font] variable for the text font.

[PDF_Doc->AddSelectList]	Adds a select list to a form. A required -Name parameter specifies the name of the select list, and a required -Values parameter specifies the array of values contained in the select list (Array: 'Value1', 'Value2'). An -Options parameter may be used instead of the -Values parameter that specifies a pair data type for each value. The first element in the pair is the value to be used upon form submission, and the second element is the human-readable label to be used for display only. An optional -Default parameter specifies the name of a default value selected. A required -Font parameter is used to specify a [PDF_Font] variable for the text font.
[PDF_Doc->AddHiddenField]	Adds a hidden field to a form. A required -Name parameter specifies the name of the hidden field, and a -Value parameter specifies the default value entered.
[PDF_Doc->AddSubmitButton]	Adds a submit button to a form. Also specifies the URL to which the form data will be submitted. A required -Name parameter specifies the name of the button, and a required -Value parameter specifies the name displayed on the button. A required -URL parameter specifies the URL of the response page. A -Font parameter is used to specify a [PDF_Font] variable for the button text font, and an optional -Caption parameter specifies a caption (displayed name) for the button.
[PDF_Doc->AddResetButton]	Adds a reset button to a form. A required -Name parameter specifies the name of the button, and a required -Value parameter specifies the name displayed on the button. A -Font parameter is used to specify a [PDF_Font] variable for the button text font, and an optional -Caption parameter specifies a caption (displayed name) for the button.

Field Label Note: With the exception of the [PDF_Doc->AddSubmitButton] and [PDF_Doc->AddSubmitButton] tags, no form input element tags include captions or labels with the field elements. Field captions and labels can be applied using the [PDF_Text] and [PDF_Doc->Add] tags to position text appropriately. See the *Creating Text Content* section for more information.

All [PDF_Doc] form member tags, with the exception of [PDF_Doc->AddHiddenField], require placement parameters for specifying the exact positioning of form elements within a page. These parameters are summarized in *Table 15: Form Placement Parameters*.

Table 16: Form Placement Parameters

Tag	Description
-Left	Specifies the placement of the left side of the form element from the left side of the current page in points. Requires a decimal value. Optional.
-Top	Specifies the placement of the bottom of the form element from the bottom of the current page in points. Requires a decimal value. Optional.
-Width	Specifies the width of the form element in points. Requires a decimal value. Optional.
-Height	Specifies the height of the form element in points. Requires a decimal value. Optional.

To add a text field:

Use the [PDF_Doc->AddTextField] tag. The example below adds a field named Field_Name that has Some Text entered by default. The field size is 144.0 points (two inches) wide and 36.0 points high.

```
[$MyFile->(AddTextField: -Name='Field_Name',
                         -Value='Some Text',
                         -Font=$MyFont,
                         -Left=72.0, -Top=350.0, -Width=144.0, -Height=36.0)]
```

To add a text area:

Use the [PDF_Doc->AddTextArea] tag. The example below adds a text area named Field_Name that has the text Insert default text here entered by default. The field size is 144.0 points wide and 288.0 points high.

```
[$MyFile->(AddTextArea: -Name='Field_Name',
                        -Value='Insert default text here',
                        -Font=$MyFont,
                        -Left=72.0, -Top=300.0, -Width=144.0, -Height=288.0)]
```

To add a checkbox:

Use the [PDF_Doc->AddCheckbox] tag. The example below adds a field named Field_Name with a checked value of Checked_Value that is checked by default. The checkbox is 4.0 points wide and 4.0 points high, and is positioned 272.0 points from the bottom and left sides of the page.

```
[$MyFile->(AddCheckBox: -Name='Field_Name',
                        -Value='Checked_Value',
                        -Checked,
                        -Left=272.0, -Top=272.0, -Width=4.0, -Height=4.0)]
```

To add a group of radio buttons:

Use the [PDF_Doc->AddRadioGroup] and [PDF_Doc->AddRadioButton] tags. The example below adds a radio button group named Group_Name, and adds two radio buttons with the values of Yes and No. The radio buttons are 6.0 points wide and 6.0 points high each.

Note: If the [PDF_Doc->AddRadioGroup] tag is not used, then radio buttons will not appear in the form.

```
[$MyFile->(AddRadioGroup: -Name='Group_Name')]
[$MyFile->(AddRadioButton: -Group='Group_Name',
                           -Value='Yes',
                           -Left=72.0, -Top=372.0, -Width=6.0, -Height=6.0)]
[$MyFile->(AddRadioButton: -Group='Group_Name',
                           -Value='No',
                           -Left=90.0, -Top=372.0, -Width=6.0, -Height=6.0)]
```

To add an editable pull-down menu:

Use the [PDF_Doc->AddComboBox] tag. The example below adds a pull-down menu named Menu_Name with the values One, Two, Three, and Four as menu values. The value One is selected by default, and an -Editable parameter allows the users to edit the values if desired. The pull-down menu size is 144.0 points wide and 36.0 points high.

```
[$MyFile->(AddComboBox: -Name='List_Name',
                        -Values=(Array: 'One', 'Two', 'Three', 'Four'),
                        -Default='One',
                        -Editable,
                        -Left=72.0, -Top=272.0, -Width=144.0, -Height=36.0)]
```

To add a pull-down menu with different displayed values:

Use the [PDF_Doc->AddComboBox] tag with the -Options parameter instead of the -Values parameter. The example below adds a pull-down menu named Menu_Name with the values 1, 2, 3, and 4 as submitable menu values, but displays the names One, Two, Three, and Four for each value. No value is selected by default.

```
[$MyFile->(AddComboBox: -Name='List_Name',
                        -Values=(Array: (Pair: (1)=(One)),
                                        (Pair: (2)=(Two)),
                                        (Pair: (3)=(Three)),
                                        (Pair: (4)=(Four))),
                        -Left=72.0, -Top=272.0, -Width=144.0, -Height=36.0)]
```

To add a select list:

Use the [PDF_Doc->AddSelectList] tag. The example below adds a select list named List_Name with the values One, Two, Three, and Four as list items. The select list is 144.0 points wide and 288.0 points high, and is positioned 72.0 points from the bottom and left sides of the page.

```
[$MyFile->(AddSelectList: -Name='List_Name',
                          -Values=(Array: 'One', 'Two', 'Three', 'Four'),
                          -Default='One',
                          -Left=72.0, -Top=72.0, -Width=144.0, -Height=288.0)]
```

To add a hidden field:

Use the [PDF_Doc->AddHiddenField] tag. The example below adds a hidden field named Field_Name with a value of Hidden_Value to a [PDF_Doc] variable named MyFile. No placement coordinates are needed because the field is not displayed on the page.

```
[$MyFile->(AddHiddenField: -Name='Field_Name',
                           -Value='Some_Value')]
```

To add a submit button:

Use the [PDF_Doc->AddSubmitButton] tag. The example below adds a submit button named Button_Name with a value of Submited_Value. The -URL parameter specifies that the user will be taken to http://www.example.com/responsepage.lasso when the button is selected in the form. A -Caption parameter specifies the displayed name of the button, which is Submit This Form.

```
[$MyFile->(AddSubmitButton: -Name='Button_Name',
                            -Value='Submited_Value',
                            -URL='http://www.example.com/responsepage.lasso',
                            -Caption='Submit This Form',
                            -Left=72.0, -Top=72.0, -Width=144.0, -Height=36.0)]
```

To add a reset button:

Use the [PDF_Doc->AddResetButton] tag. The example below adds a submit button named Button_Name with a value of Submited_Value. A -Caption parameter specifies the displayed name of the button, which is Reset This Form.

```
[$MyFile->(AddResetButton: -Name='Button_Name',
                           -Value='Submited_Value',
                           -Caption='Reset This Form',
                           -Left=72.0, -Top=72.0, -Width=144.0, -Height=36.0)]
```

Submitting Form Data to Lasso-Enabled Databases

In Lasso Professional 8, one has the ability to submit data from a PDF form to a Lasso-enabled database. PDF forms may be used in the same way as HTML forms to submit action parameters to an LDML response page, where database actions can occur via an [Inline] tag.

The following example shows the HTML form example in the *Database Interaction Fundamentals* chapter *Inline Method* section as it would appear in a [PDF_Doc] variable in a Form.lasso page.

To submit information to database using a PDF form:

1 In the Form.lasso page, name the PDF form fields to correspond to the names of fields in the desired database. The names of these fields will be used in the [Inline] tag in the LDML response page.

```
[Var:'MyFile'=(PDF_Doc: -File='Form.pdf', -Size='A4')]
[Var:'MyFont'=(PDF_Font: -Face='Helvetica', -Size=12)]
[$MyFile->(DrawText: 'First Name:', -Font=$MyFont, -Left=80.0, -Top=60.0)]
[$MyFile->(DrawText: 'Last Name:', -Font=$MyFont, -Left=80.0, -Top=60.0)]
[$MyFile->(AddTextField: -Name='First Name',
                         -Value='Enter First Name',
                         -Left=144.0, -Top=72.0, -Width=144.0, -Height=36.0)]
[$MyFile->(AddTextField: -Name='Last Name',
                         -Value='Enter Last Name',
                         -Left=144.0, -Top=92.0, -Width=144.0, -Height=36.0)]
```

2 Create a submit button in the Form.lasso page that contains the name and URL of the LDML response page.

```
[$MyFile->(AddSubmitButton: -Name='Search',
                            -Value='Search',
                            -Caption='Click here to Search',
                            -URL='http://www.example.com/Response.lasso',
                            -Font=$MyFont)]
[$MyFile->Close]
```

After the [PDF_Doc] variable is closed and executed on the server, a Form.pdf file will be created with a form.

3 In the Response.lasso page, create an [Inline] tag that uses the action parameters passed from the PDF form to perform a database action. This example performs a search on the Contacts database using the First_Name and Last_Name parameters passed from the PDF form.

```
[Inline: -Search,
-Database='Contacts',
-Table='People',
-KeyField='ID',
'First_Name'=(Action_Param: 'First_Name'),
```

```
'Last_Name'=(Action_Param: 'Last_Name')]
There were [Found_Count] record(s) found in the People table.
[Records]
<br>[Field: 'First_Name'] [Field: 'Last_Name']
[/Records]
[/Inline]
```

If the user of the PDF form entered Jane for the first name and Doe for the last name, then the following results would be returned.

```
➜ There were 1 record(s) found in the People table.
  Jane Doe
```

Creating Tables

Tables can be created in PDF documents for displaying data. These are created using the [PDF_Table] tag, and added to a PDF variable using [PDF_Doc] member tags, which are described in this section.

Defining Tables

Tables for organizing data can be defined for use in a PDF document using the [PDF_Table] tag. This tag is used to set a variable as a [PDF_Table] type, and the [PDF_Table] variable is then added to a [PDF_Doc] variable.

Table 17: [PDF_Table] Tag and Parameters

Tag	Description
[PDF_Table]	Creates a table to be placed in a PDF. Uses parameters which set the basic specifications of the table to be created.
-Cols	Specifies the number of columns in a table. Required.
-Rows	Specifies the number of rows in a table. Required.
-Spacing	Specifies the spacing around a table cell. Defaults to 0 (no spacing) if not specified. Optional.
-Padding	Specifies the padding within a table cell. Defaults to 0 (no padding) if not specified. Optional.
-Width	Specifies the width of the table as a percentage of the current page width. Defaults to the width of the cell text plus spacing, padding, and borders if not specified. Optional.
-BorderWidth	Specifies the border width of the table in points. Requires a decimal value. Optional.

-BorderColor	Specifies the border color of the table. Requires a hex color string (e.g. '#000000'). Optional.
-BackgroundColor	Specifies the background color of the table. Requires a hex color string (e.g. '#CCCCCC'). Optional.
-ColWidth	Sets the column width for each column in the table. Requires an array of decimals representing the width percentage of each column. Optional.

Member tags can be used to set additional specifications for a [PDF_Table] variable, as well as access parameter values from [PDF_Table] variables. These tags are summarized in *Table 18: [PDF_Table] Member Tags*.

Table 18: [PDF_Table] Member Tags

Tag	Description
[PDF_Table->GetColumnCount]	Returns the number of columns in a [PDF_Table] variable.
[PDF_Table->GetRowCount]	Returns the number of rows in a [PDF_Table] variable.
[PDF_Table->GetAbsWidth]	Returns the total [PDF_Table] variable width in pixels.

To create a basic table:

Use the [PDF_Table] tag. The example below creates a table with two columns and five rows, with table cell spacing of one point and cell padding of two points. The width of the table is set at 75 percent of the current page width.

```
[Var:'MyTable'=(PDF_Table: -Cols=2,
                           -Rows=5,
                           -Spacing=1,
                           -Padding=2,
                           -Width=75,
                           -BackgroundColor='#CCCCCC')]
```

To create a table with a border:

Use the [PDF_Table] tag with the -Border... parameters. The example below creates a basic table, and then adds a black border with a width of 3 points to the table.

```
[Var:'MyTable'=(PDF_Table: -Cols=2,
                           -Rows=5,
                           -Spacing=1,
                           -Padding=2,
                           -BorderWidth=3,
                           -BorderColor='#000000')]
```

To rotate a table:

Use the [PDF_Table] tag with the -Rotate parameter The example below creates a basic table, and then rotates it by 90 degrees clockwise.

```
[Var:'MyTable'=(PDF_Table: -Cols=2,
                           -Rows=5,
                           -Spacing=1,
                           -Padding=2,
                           -Rotate=90)]
```

To create a table with pre-specified column widths:

Use the [PDF_Table] tag with the -ColWidth parameter The example below creates a basic table with percentage widths for three columns.

```
[Var:'MyTable'=(PDF_Table: -Cols=2,
                           -Rows=5,
                           -Spacing=1,
                           -Padding=2,
                           -ColWidth=(Array: '50.0', '25.0', '25.0'))]
```

Adding Content to Table Cells

Content is added to table cells using additional [PDF_Table] member tags These tags are summarized in *Table 19: Cell Content Tags*.

Table 19: Cell Content Tags

Tag	Description
[PDF_Table->Add]	Inserts text content or a new nested table into a cell. Requires a text string or a new [PDF_Table] variable to be inserted as a parameter. Also requires parameters described in the following table.
-Col	Specifies the column number starting from 0 (numbered from left to right) of the cell to add or remove. Requires an integer value. Required.
-Row	Specifies the row number starting from 0 (numbered from top to bottom) of the cell to add or remove. Requires an integer value. Required.
-Colspan	Specifies the number of columns a cell should span. Requires an integer value. Required.
-Rowspan	Specifies the number of rows a cell should span. Requires an integer value. Required.
-VerticalAlignment	Vertical alignment for text within a cell. Accepts a value of 'Top', 'Center', or 'Bottom'. Defaults to 'Center' if not specified. Optional.

-HorizontalAlignment	Horizontal alignment for text within a cell. Accepts a value of 'Left', 'Center', or 'Right'. Defaults to 'Center' if not specified. Optional.
-BorderColor	Specifies the border color for the cell (e.g. '#440000'). Defaults to '#000000' if not specified. Optional.
-BorderWidth	Specifies the border width of the cell in points. Requires a decimal value. Defaults to 0 if not specified. Optional.
-Header	Specifies that the cell is a table header. This is typically used for cells in the first row. Optional.
-NoWrap	Specifies that the text contained in a cell should not wrap to conform to the cell size specifications. If used, the cell will expand to the right to accomodate longer text strings. Optional.

To add a cell to a table:

Use the [PDF_Table->Add] tag. The example below adds a cell to the first row and column in a table. Note that the first row and column are numbered 0.

```
[$MyTable->(Add: 'This is the first cell in my table', -Col=0, -Row=0, -Colspan=0, -Rowspan=0]
```

To add a multi-column cell to a table:

Use the [PDF_Table->Add] tag with the number of columns to span for the -Column parameter. The example below adds a cell to the first row that spans three columns. The -NoWrap parameter is used to indicate that the added text will not be wrapped into multiple lines.

```
[$MyTable->(Insert: 'This text will only stay on one line regardless of the table size', -Col=0, -Row=0, -Colspan=3, -Rowspan=1, -NoWrap)]
```

To add a header cell to a table:

Use the [PDF_Table->Add] tag with the -Header parameter. The example below adds the header My Column Title to the first column of the table.

```
[$MyTable->(Add: 'My Column Title', -Col=0, -Row=0, -Colspan=0, -Rowspan=0, -Header]
```

To add a cell with a border to a table:

Use the [PDF_Table->Add] tag with the -BorderWidth and -BorderColor parameter. The example below adds a cell with a red border to the first column of the table.

```
[$MyTable->(Add: 'This cell has a border', -Col=0, -Row=0, -Colspan=0, -Rowspan=0, -BorderWidth=45.0, -BorderColor='#440000']
```

Adding Tables

Once a [PDF_Table] object is completely defined and has cell content, it may then be added to a [PDF_Doc] objects using the [PDF_Doc->Add] tag.

To add a table to a [PDF_Doc] variable:

Use the [PDF_Doc->Add] tag. The following example adds a predefined [PDF_Table] variable named MyTable to a [PDF_Doc] variable named MyFile.

```
[$MyFile->(Add: $MyTable)]
```

Creating Graphics

This section describes how to draw custom graphic objects and insert image files within a PDF document.

Inserting Images

Image files can be placed within PDF pages via the [PDF_Doc->AddImage] tag, which is summarized in *Table 20: [PDF_Doc] Image Tag*.

Table 20: [PDF_Image] Tag and Parameters

Tag	Description
[PDF_Image]	Casts an image file as an LDML object so it can be placed in a PDF file. Requires either a -File, -URL, or -Raw parameter, as described below. Only images in JPEG, GIF, PNG, and WMF formats may be used.
-File	Specifies the local path to an image file. Required if the -URL or -Raw parameters are not used.
-URL	Specifies a URL to an image file. Required if the -File or -Raw parameters are not used.
-Raw	Inputs a raw string of bits representing the image. Required if the -URL or -File parameters are not used.
-Height	Scales the image to the specified height. Requires a decimal value which is the desired image height in points. Optional.
-Width	Scales the image to the specified width. Requires a decimal value which is the desired image width in points. Optional.
-Proportional	Keyword parameter specifying that all scaling should preserve the aspect ratio of the inserted page. Optional.
-Rotate	Rotates the image by the specified degrees clockwise. Optional.

To add an image file to a [PDF_Doc] variable:

Use the [PDF_Image] tag. The following example adds a file named Image.jpg in a /Documents/Images/ folder to a [PDF_Doc] variable named MyFile.

```
[Var:'Image'=PDF_Image: -File='/Documents/Images/Image.jpg')]
[$MyFile->(Add: $Image, -Left=144.0, -Top=300.0)]
```

To scale image file:

Use the [PDF_Image] tag with the -Height or -Width parameter. The following example proportionally reduces the size of the added image by 50%.

```
[Var:'Image'=PDF_Image: -File='/Documents/Images/Image.jpg', -Height=50.0)]
[$MyFile->(Add: $Image, -Left=144.0, -Top=300.0)]
```

To rotate an image file :

Use the [PDF_Image] tag with the -Rotate parameter. The following example rotates the added image by 90 degrees clockwise.

```
[Var:'Image'=PDF_Image: -File='/Documents/Images/Image.jpg', -Rotate=90.0)]
[$MyFile->(Add: $Image, -Left=144.0, -Top=300.0)]
```

Drawing Graphics

To draw custom graphics, Lasso uses a coordinate system to determine the placement of each graphical object. This coordinate system is a standard coordinate plane with horizontal (X) vertical (Y) axis, where a point on a page is defined by an array containing horizontal and vertical position values (X, Y). The base point of the coordinate plane (0, 0) is located in the lower left corner for the current page. Increasing an X-Value moves a point to the right in the page, and increasing the Y-Value moves the point up in the page. The maximum X and Y values are defined by the current width and height of the page in points.

Custom graphics may be drawn in PDF pages using [PDF_Doc] drawing member tags. These member tags operate by controlling a "virtual pen" which draws graphics similar to a true graphics editor. These member tags are summarized in *Table 21: [PDF_Doc] Drawing Member Tags*.

Table 21: [PDF_Doc] Drawing Member Tags

Tag	Description
[PDF_Doc->SetColor]	Sets the color and style for subsequent drawing operations. A required 'Type' parameter specifies whether the drawing action is of type Stroke, Fill, or Both. A required 'Color' parameter specifies a color type of Gray, RGB, or CMYK. If Gray is specified, a decimal specifies a color strength value. If RGB is specified,three decimal values specifiy red, green and blue values, respectively. If CMYK is specified, four decimal values specifiy cyan, magenta, yellow, and black values, respectively. Color values are specified as decimals ranging from 0 to 1.0.
[PDF_Doc->SetLineWidth]	Sets the line width for subsequent drawing actions in points. Requires a decimal point value.
[PDF_Doc->MoveTo]	Moves the virtual pen to the specified coordinates. This tag is required to move the virtual pen into the correct position before drawing a line or a curve. Requires a string of integer X-axis and Y-axis coordinates to designate the starting point (X, Y).
[PDF_Doc->Line]	Draws a line. Requires a string of integer points which specifies the starting point and ending point of the line (X1, Y1, X2, Y2).
[PDF_Doc->CurveTo]	Draws a curve. Requires a string of integer points which specifies the starting point, middle point, and ending point of the curve (X1, Y1, X2, Y2, X3, Y3).
[PDF_Doc->Rect]	Draws a rectangle. Requires a string of X and Y integer points which specifies the lower right corner of the rectangle, and the height and width of the rectangle sides from that coordinate (X, Y, Width, Height).
[PDF_Doc->Circle]	Draws a circle. Requires a string of integer points for the center coordinates and a radius length value (X, Y, R).
[PDF_Doc->Arc]	Draws an arc. Requires a string of integer points for the center coordinates and radius of the invisible circle to which the arc belongs, a starting degree which specifies the degrees of the circle at which the arc starts, and an ending degree which specifies the circle degrees at which the arc ends (X, Y, R, Start, End). Angles start with 0 to the right of the center and increase counter-clockwise.

To set the color and style for a drawing action:

Use the [PDF_Doc->SetColor] tag. The example below sets a color of red for all subsequent drawing action until another [PDF_Doc->SetColor] tag is set.

```
[$MyFile->(SetColor: 'Stroke', 'RBG', 0.9, 0.1, 0.1)]
```

To set the line width of a drawing action:

Use the [PDF_Doc->SetLineWidth] tag. The example below sets a line width of 5 points for all subsequent drawing action until another [PDF_Doc->SetLineWidth] tag is set.

```
[$MyFile->(SetLineWidth: 5.0)]
```

To draw a line:

Use the [PDF_Doc->Line] tag. The example below draws a horizontal line from points (8, 8) to points (32, 32). The [PDF_Doc->MoveTo] tag must first be used to move the virtual pen to the starting point coordinates.

```
[$MyFile->(MoveTo: 8, 8)]
[$MyFile->(Line: 8, 8, 32, 32)]
```

To draw a curve:

Use the [PDF_Doc->CurveTo] tag. The example below draws a curve starting from points (8, 8), peaking at points (32, 32), and ending at points (56, 8). The [PDF_Doc->MoveTo] tag must first be used to move the virtual pen the starting point coordinates.

```
[$MyFile->(MoveTo: 8, 8)]
[$MyFile->(CurveTo: 8, 8, 32, 32, 56, 8)]
```

To draw a rectangle:

Use the [PDF_Doc->Rect] tag. The example below draws a rectangle whose lower left corner is at coordinates (10, 60), has left and right sides that are 50 points long, and has top and bottom sides that are 20 point long.

```
[$MyFile->(Rect: 10, 60, 20, 50)]
```

To draw a circle:

Use the [PDF_Doc->Circle] tag. The example below draws a circle whose center is at coordinates (50, 50) and has a radius of 20 points.

```
[$MyFile->(Circle: 50, 50, 20)]
```

To draw an arc:

Use the [PDF_Doc->Arc] tag. The example below draws an arc whose center is at coordinates (50, 50), has a radius of 20 points, and runs from 0 degrees to 90 degrees from the center.

```
[$MyFile->(Arc: 50, 50, 20, 0, 90)]
```

Creating Barcodes

Barcodes are special device-readable images that can be created in PDF documents using the [PDF_Barcode] tag, and added to a PDF variable using [PDF_Doc] member tags, which are described in this section. Lasso Professional 8 can be used to create the following industry-standard barcodes:

- Code 39 (alphanumeric, ASCII subset)
- Code 39 Extended (alphanumeric, escaped text)
- Code 128
- Code 128 UCC/EAN
- Code 128 Raw
- EAN (8 digits)
- EAN (13 digits)
- POSTNET
- PLANET

Creating Bar Codes

Barcodes can be defined for use in a PDF file using the [PDF_Barcode] tag. This tag is used to set a variable as a [PDF_Barcode] type, and the [PDF_Barcode] variable is added to a [PDF_Doc] variable using member tags.

Table 22: [PDF_Barcode] Tag and Parameters

Tag	Description
[PDF_Barcode]	Creates a barcode image to be placed in a PDF. Uses parameters which set the basic specifications of the barcode to be created.
-Type	Specifies the type of barcode to be created. Available parameters are CODE39, CODE39_EX, CODE128, CODE128_UCC, CODE128_RAW, EAN8, EAN13, POSTNET, and PLANET. Defaults to CODE39 if not specified.
-Code	Specifies the numeric or alphanumeric barcode data. Some formats require specific data strings: EAN8 requires an 8-digit integer, EAN13 requires a 13-digit integer, POSTNET requires a zip code, and Code39 requires uppercase characters. Required.
-Color	Specifies the color of the bars in the barcode. Requires a hex string color value. Defaults to '#000000' if not specified. Optional.

-Supplemental	Adds an additional two or five digit supplemental barcode to EAN8 or EAN13 barcode types. Requires a two or five digit integer as a parameter. Optional.
-GenerateChecksum	Generates a checksum for the barcode. Optional.
-ShowCode39StartStop	Displays start and stop characters (*) in the text for Code 39 barcodes. Optional.
-ShowEANGuardBars	Show the guard bars for EAN barcodes. Optional.
-BarHeight	Sets the height of the bars in points. Requires a decimal value.
-BarWidth	Sets the width of the bars in points. Requires a decimal value.
-BaseLine	Sets the text baseline in points. Requires a decimal value.)
-ShowChecksum	Keyword parameter sets the generated checksum to be shown in the text
-Font	Sets the text font. Requires a [PDF_Font] variable.
-BarMultiplier	Sets the bar multiplier for wide bars. Requires a decimal value.
-TextSize	Sets the size of the text. Requires a decimal value.

To create a barcode:

Use the [PDF_Barcode] tag. The example below creates a basic Code 39 barcode with the data 1234567890, and uses the optional Code 39 start and stop characters (*). The barcode is then added to a [PDF_Doc] variable using [PDF_Doc->Add].

```
[Var:'Barcode'=(PDF_Barcode:
-Type='CODE39',
-Code='1234567890',
-ShowCode39StartStop)]
[$MyPDF->(Add: $Barcode, -Left=150.0, -Top=100.0)]
```

To create a barcode with a specified bar width:

Use the [PDF_Barcode] tag with the -BarWidth parameter. The following example sets a [PDF_Barcode] variable with a bar width of 0.2 points.

```
[Var:'Barcode'=(PDF_Barcode:
-Type='CODE39',
-Code='1234567890',
-BarWidth=0.2)]
[$MyPDF->(Add: $Barcode, -Left=150.0, -Top=100.0)]
```

To create a barcode with a specified bar multiplier:

Use the [PDF_Barcode] tag with the -BarMultiplier parameter. The following example sets a [PDF_Barcode] variable with a bar multiplier constant of 4.0. The barcode is then added to a [PDF_Doc] variable using [PDF_Doc->Add].

```
[Var:'Barcode'=(PDF_Barcode:
-Type='CODE39',
-Code='1234567890',
-BarMultiplier=4.0)]
[$MyPDF->(Add: $Barcode, -Left=150.0, -Top=100.0)]
```

To create a barcode with a specified text size:

Use the [PDF_Barcode] tag with the -TextSize parameter. The following example sets a [PDF_Barcode] variable with a text size of 6 points. The barcode is then added to a [PDF_Doc] variable using [PDF_Doc->Add].

```
[Var:'Barcode'=(PDF_Barcode:
-Type='CODE39',
-Code='1234567890',
-TextSize=6)]
[$MyPDF->(Add: $Barcode, -Left=150.0, -Top=100.0)]
```

To create a barcode with a specified font:

Use the [PDF_Barcode] tag with the -Font parameter. The following example sets a [PDF_Barcode] variable font specified in a [PDF_Font] variable named MyFont. The barcode is then added to a [PDF_Doc] variable using [PDF_Doc->Add].

```
[Var:'Barcode'=(PDF_Barcode:
-Type='CODE39',
-Code='1234567890',
-Font=$MyFont)]
[$MyPDF->(Add: $Barcode, -Left=150.0, -Top=100.0)]
```

Example PDF Files

This section provides complete examples of creating PDF files using the tags described in this chapter. Examples include a two-page PDF file with multiple text styles, a PDF file with a form, a PDF file with a table, a PDF file with drawn graphics, and a PDF file with a barcode.

Special Characters Note: All examples in this section use the Mac OS X line break character \n in the text sections. If creating PDF files on the Windows or Linux version of Lasso Professional 8, change all instances of \n to \r\n for Windows, or \r for Linux.

PDF Text Example

The following example creates a PDF file that contains two pages of text with multiple text styles.

```
[Var:'Text_Example'=(PDF_Doc: -File='Text_Example.pdf', -Size='A4')]
[$Text_Example->AddPage]
[$Text_Example->(SetPageNumber: 1)]
[Var:'Font1'=(PDF_Font: -Face='Helvetica', -Size='24', -Color='#990000')]
[Var:'Font2'=(PDF_Font: -Face='Helvetica', -Size='14', -Color='#000000')]
[Var:'Font3'=(PDF_Font: -Face='Helvetica', -Size='14', -Color='#0000CC')]
[Var:'Title'=(PDF_Text: 'Lasso Professional 8', -Type='Chunk', -Font=$Font1)]
[$Text_Example->(Add: $Title, -Number=1)]
[Var:'Text1'=(PDF_Text:'\n\nThe Lasso product line consists of authoring and serving
tools that allow Web designers and Web developers to quickly build and serve
powerful data-driven Web sites with maximum productivity and ease. The product
line includes Lasso Professional for building, serving, and administering data-driven
Web sites, and Lasso Studio for building and testing data-driven Web sites within a
graphical editor.\n\nLasso Professional 8 works with the following data sources:',
-Type='Paragraph', -Leading=15, -Font=$Font2)]
[$Text_Example->(Add: $Text1)]
[Var:'List'=(PDF_List: -Format='Bullet', -Bullet='-', -Font=$Font2, -Indent=30)]
[$List->(Add:'FileMaker Pro')]
[$List->(Add:'MySQL')]
[$List->(Add:'Microsoft SQL Server')]
[$List->(Add:'Frontbase')]
[$List->(Add:'Sybase')]
[$List->(Add:'PostgreSQL')]
[$List->(Add:'DB2')]
[$List->(Add:'Plus many other JDBC-compliant databases')]
[$Text_Example->(Add: $List)]
[Var:'Text2'=(PDF_Text:'\nLasso\'s innovative architecture provides an industry first
multi-platform, database-independent and open standards approach to delivering
database-driven Web sites firmly positioning Lasso technology within the rapidly
evolving server-side Web tools market. Lasso technology is used at hundreds of
thousands of Web sites worldwide.\n\n',  -Type='Paragraph', -Font=$Font2)]
[$Text_Example->(Add: $Text2)]
[Var:'Text3'=(PDF_Text:'Click here to go to the OmniPilot Web site',  -Type='Phrase',
-Font=$Font3, -Underline='true', -Anchor='http://www.omnipilot.com')]
[$Text_Example->(Add: $Text3)]
[$Text_Example->(DrawText: (String: $Text_Example->GetPageNumber),
-Font=$Font2, -Top=30, -Left=560)]
[$Text_Example->AddPage]
[$Text_Example->(SetPageNumber: 2)]
[Var:'Text4'=(PDF_Text:'Lasso Professional 8 is server-side software that adds a suite
of dynamic functionality and administration to your Web server. This functionality
empowers you to build and serve just about any dynamic Web application that can
be built with maximum productivity and ease.\n\n', -Type='Paragraph', -Leading=15,
-Font=$Font2)]
```

```
[$Text_Example->(Add: $Text4)]
[Var:'Text5'=(PDF_Text:'Lasso works by using a simple tag-based markup language
(LDML), which can be embedded in Web pages and scripts residing on your Web
server. The details of scripting and programming in LDML are covered in the LDML
8 Language Guide also included with this product. By default, Lasso Professional 8
is designed to run on the most prevalent modern Web server platforms with the most
popular Web serving applications. In addition, Lasso's extensibility allows Web Server
Connectors to be authored for any Web server for which default connectivity is not
provided.\n\n', -Type='Paragraph', -Leading=15, -Font=$Font2)]
[$Text_Example->(Add: $Text5)]
[$Text_Example->(DrawText: (String: $Text_Example->GetPageNumber),
-Font=$Font2, -Top=30, -Left=560)]
[$Text_Example->Close]
```

PDF Form Example

The following example creates a PDF file that contains both text and a form.

```
<?LassoScript
Var: 'Form_Example' = (PDF_Doc: -File='Form_Example.pdf', -Size='A4');
Var: 'myFont' = (PDF_Font: -Face='Helvetica', -Size='12');
$Form_Example->(AddText:'This PDF file contains a form. See below.\n',
  -Font=$myFont);
$Form_Example->(DrawText: 'Select List', -Font=$myFont, -Left=90, -Top=116);
$Form_Example->(AddSelectList: -Name='mySelectList', -Values=(Array: 'one',
  'two', 'three', 'four'), -Default='one', -Left=216, -Top=104, -Width=144, -Height=72,
  -Font=$myFont);
$Form_Example->(DrawText: 'Pull-Down Menu', -Font=$myFont, -Left=90, -Top=200);
$Form_Example->(AddComboBox: -Name='myComboBox', -Values=(Array: 'one',
  'two', 'three', 'four'), -Default='one', -Left=216, -Top=188, -Width=144, -Height=18,
  -Font=$myFont);
$Form_Example->(DrawText: 'Text Area', -Font=$myFont, -Left=90, -Top=238);
$Form_Example->(AddTextArea: -Name='myTextArea', -Value='Some text', -Left=216,
  -Top=230, -Width=144, -Height=72, -Font=$myFont);
$Form_Example->(DrawText: 'Password Field', -Font=$myFont, -Left=90, -Top=334);
$Form_Example->(AddPasswordField: -Name='myPassword', -Value='****', -Left=216,
  -Top=322, -Width=144, -Height=18, -Font=$myFont);
$Form_Example->(DrawText: 'Text Field', -Font=$myFont, -Left=90, -Top=368);
$Form_Example->(AddTextField: -Name='myTextField', -Value='Some More Text',
  -Left=216, -Top=360, -Width=144, -Height=18, -Font=$myFont);
$Form_Example->(AddHiddenField: -Name='myHiddenField', -Value='Shh');
$Form_Example->(AddSubmitButton: -URL='http://www.example.com/response.lasso',
  -Name='myButton', -Value='Submit', -Caption='Submit Form', -Left=216, -Top=400,
  -Width=100, -Height=26, -Font=$myFont);
$Form_Example->(AddResetButton: -Name='Reset', -Value='Reset',
```

```
  -Caption='Reset Form', -Left=365, -Top=400, -Width=100, -Height=26,
  -Font=$myFont);
$Form_Example->Close;
?>
```

PDF Table Example

The following example creates a PDF file that contains both text and a table.

```
[Var:'Table_Example'=(PDF_Doc: -File='Table_Example.pdf',
                               -Size='A4')]
[Var:'Font1'=(PDF_Font: -Face='Helvetica',
                        -Size='24')]
[Var:'Font2'=(PDF_Font: -Face='Helvetica',
                         -Size='12')]
[Var:'Text'=(PDF_Text:'This PDF file contains a table. See below.\n\n',
                                -Leading=15,
                                -Font=$Font1)]
[Var:'Cell1'=(PDF_Text:'Cell One',  -Font=$Font2)]
[Var:'Cell2'=(PDF_Text:'Cell Two',  -Font=$Font2)]
[Var:'Cell3'=(PDF_Text:'Cell Three',  -Font=$Font2)]
[Var:'Cell4'=(PDF_Text:'Cell Four',  -Font=$Font2)]
[$Table_Example->(Add: $Text)]
[Var:'MyTable'=(PDF_Table: -Cols=2,
                           -Rows=2,
                           -Spacing=4,
                           -Padding=4,
                           -Width=75,
                           -BorderWidth=7)]
[$MyTable->(Add: $Cell1, -Col=0, -Row=0,
                           -Colspan=1, -Rowspan=1,
                           -VerticalAlignment='Center',
                           -HorizontalAlignment='Center',
                           -BorderWidth=4)]
[$MyTable->(Add: $Cell2, -Col=0, -Row=1,
                           -Colspan=1, -Rowspan=1,
                           -VerticalAlignment='Center',
                           -HorizontalAlignment='Center',
                           -BorderWidth=4)]
[$MyTable->(Add: $Cell3, -Col=1, -Row=0,
                           -Colspan=1, -Rowspan=1,
                           -VerticalAlignment='Center',
                           -HorizontalAlignment='Center',
                           -BorderWidth=4)]
[$MyTable->(Add: $Cell4, -Col=1, -Row=1,
                           -Colspan=1, -Rowspan=1,
                           -VerticalAlignment='Center',
```

```
                                    -HorizontalAlignment='Center',
                                    -BorderWidth=4)]
[$Table_Example->(Add: $MyTable)]
[$Table_Example->Close]
```

PDF Graphics Example

The following example shows how to create a PDF file that contains drawn graphic objects.

```
[Var:'Graphic_Example'=(PDF_Doc: -File='Graphic_Example.pdf',
                        -Height='650',
                        -Width='550')]
[Var:'Text'=(PDF_Text:'This PDF file contains lines and circles. See below.\n')]
[$Graphic_Example->(Add: $Text)]
[$Graphic_Example->(Line: 200, 400, 400, 400)]
[$Graphic_Example->(Line: 200, 500, 400, 500)]
[$Graphic_Example->(Line: 266, 333, 266, 566)]
[$Graphic_Example->(Line: 333, 333, 333, 566)]
[$Graphic_Example->(Line: 200, 333, 400, 566)]
[$Graphic_Example->(Circle: 233, 366, 20)]
[$Graphic_Example->(Circle: 300, 452, 20)]
[$Graphic_Example->(Circle: 366, 533, 20)]
[$Graphic_Example->(Line: 220, 432, 240, 472)]
[$Graphic_Example->(Line: 220, 472, 240, 432)]
[$Graphic_Example->(Line: 360, 432, 380, 472)]
[$Graphic_Example->(Line: 360, 472, 380, 432)]
[$Graphic_Example->(Line: 220, 517, 240, 558)]
[$Graphic_Example->(Line: 220, 558, 240, 517)]
[$Graphic_Example->Close]
```

PDF Barcode Example

The following example shows how to create a PDF file that contains text accompanied by a barcode.

```
[Var:'Barcode_Example'=(PDF_Doc: -File='Barcode_Example.pdf',
                                 -Height=172,
                                 -Width=300)]
[Var:'Font1'=(PDF_Font: -Face='Courier', -Size='12')]
[Var:'MyBarcode'=(PDF_Barcode: -Type='CODE39',
                               -Code='1234567890',
                               -GenerateCheckSum,
                               -ShowCode39StartStop,
                               -TextSize: 6)]
```

```
[$Barcode_Example->(DrawText: 'The Shipping Company\n',
                                    -Font=$Font1, -Left=72, -Top=90)]
[$Barcode_Example->(Add: $MyBarcode, -Left=72, -Top=40)]
[$Barcode_Example->Close]
```

Serving PDF Files

This section describes how PDF files can be served using Lasso Professional 8. This can be done by supplying a download link to the created PDF file, or by using the [PDF_Serve] tag described in this chapter.

Syntax Note: When creating PDF files using LDML and serving data to a browser in the same page, the use of the LassoScript syntax is recommended as it does not output hard returns in the rendered HTML source code. For more information on LassoScript, see the ***LassoScript*** chapter.

Linking to PDF Files

Named PDF files may be linked to in a format file using basic HTML. Once a user clicks on a link to a file with a .pdf extension, the client browser should prompt to download the file or launch the file in PDF reader (if configured to do so).

To link a PDF file:

The example below shows how a PDF can be created and written to file, and then linked to in the format file.

```
<?LassoScript
   Var:'MyFile'=(PDF_Doc: -File='MyFile.pdf',
                                -Size='A4');
   Var:'Text'=(PDF_Text: 'Hello World');
   $MyFile->(Add: $Text);
   $MyFile->Close;
?>
<html>
<body>
<p>Click on the following link to download MyFile.pdf.</p>
<p><a href="MyFile.pdf">Click Here</a></p>
</body>
</html>
```

Serving PDF Files to Client Browsers

PDF files may also be served directly to a client browser using the [PDF_Serve] tag. This tag automatically informs the client Web browser that the data being load is a PDF file, and outputs the file with the correct file name. If the client Web browser is configured to handle PDF files via a reader, then the out PDF file will automatically be opened in the clients configured PDF reader. Otherwise, the client Web browser should prompt the user to save the file.

Table 23: PDF Serving Tags

Tag	Description
[PDF_Serve]	Serves a PDF file to a client browser with a MIME type of application/pdf. Requires a -File parameters which specifies the name of the file to be output to the browser. An optional -Type parameter may be used to specify addtional MIME types.

To serve a PDF file to a client browser:

Use the [PDF_Serve] tag to serve the created PDF file. The file parameter specifies the file name that should be output.

```
<?LassoScript
   Var:'MyFile'=(PDF_Doc: -File='MyFile.pdf',
                          -Size='A4',
                          -NoCompress);
   Var:'Text'=(PDF_Text: 'Hello World');
   $MyFile->(Add: $Text);
   $MyFile->Close;
   PDF_Serve: $MyFile, -File='MyFile.PDF'
?>
```

To serve a PDF file without writing to file:

PDF files may be served to the client browser without ever writing them to file on the local server. This is done using the [PDF_Doc] tag without the -File parameter. This allows a PDF file to be created in the system memory, but does not the save the file to a hard drive location. The resulting file can saved by the end user to a location on the end user's hard drive.

```
<?LassoScript
   Var:'MyFile'=(PDF_Doc:  -Size='A4',
                          -NoCompress);
```

```
    Var:'Text'=(PDF_Text: 'Hello World');
    $MyFile->(Add: $Text);
    $MyFile->Close;
    PDF_Serve: $MyFile, -File='MyFile.pdf';
?>
```

34

Chapter 34
JavaBeans

LDML provides access to JavaBeans through a simple [JavaBeans] data type.

- *Overview* describes what JavaBeans are and how they are used.
- *Installing JavaBeans* describes how to install new JavaBeans in Lasso.
- *JavaBeans Type* describes how to access JavaBeans through Lasso.
- *Creating JavaBeans* covers how to create JavaBeans for use with Lasso.

Overview

A JavaBean is a reusable software component written in the Java programming language. JavaBeans are operating-system independent and can be developed in one environment and run, copied, modified and extended in another. Developers can easily share JavaBeans between multiple projects.

Lasso's support for JavaBeans allows Lasso developers to use existing JavaBeans that have been created for Java or JSP programming. Lasso developers can also easily extend the functionality of Lasso without having to learn LJAPI. The Creating JavaBeans section of this chapter contains an example of how to create a JavaBean.

Lasso should be able to access any JavaBeans that are written to the JavaBeans specification. See the *Creating JavaBeans* section for a technical discussion of which JavaBeans can be used with Lasso.

Lasso supports JavaBeans through a [Java_Bean] data type that allows JavaBeans to be instantiated and their attributes to be manipulated. The overall methodology is quite similar to that implemented in JSP 1.0:

For example, in JSP a JavaBean called ColorBean could be called as follows. The JavaBean is instantiated, a property red is set to the value 125, and finally the property saturation is fetched.

```
<jsp:useBean id="bean" class=ColorBean" />
<jsp:setProperty name="bean" property="red" value="125" />
<jsp:getProperty name="bean" property="saturation" />
```

The process is very similar in LDML. The JavaBean is instantiated using the [Java_Bean] tag and stored in a variable named myBean. The [Java_Bean->SetProperty] tag is used to set red to 125, and the [Java_Bean->GetProperty] tag is used to get the property saturation.

```
[Var: 'myBean' = (Java_Bean: 'ColorBean')]
[$myBean->(SetProperty: -Keyword='red', -Value=125)]
[$myBean->(GetProperty: -Keyword='saturation')]
```

Lasso also supports the * as a special [Java_Bean->SetProperty] value to automatically set all the properties of a JavaBean from the action parameters of the current page. This JSP code sets up a JavaBean using the current action parameters of the page:

```
<jsp:useBean id="bean" class=ColorBean" />
<jsp:setProperty name="bean" property="*" />
<jsp:getProperty name="bean" property="saturation" />
```

The equivalent code in LDML is shown below:

```
[Var: 'myBean' = (Java_Bean: 'ColorBean')]
[$myBean->(SetProperty: -Keyword='*')]
[$myBean->(GetProperty: -Keyword='saturation')]
```

The LDML page could be called with this URL and would display the value of the saturation property of the JavaBean. The red, green, and blue properties would be set automatically.

```
http://www.example.com/javabean.lasso?red=25&green=0&blue=160
```

The functions that the JavaBean implements can also be called directly as member tags of the [Java_Bean] object. For example if a JavaBean implements an evaluate() function it can be called as follows.

```
[$myBean->Evaluate]
```

The [Java_Bean->BeanProperties] tag returns an array of possible JavaBeans property names.

```
[Var: 'myBean' = (Java_Bean: 'ColorBean')]
[$myBean->(BeanProperties)]
```

```
➜ (Array: (Red), (Blue), (Green), (Saturation), (Hue), (Color))
```

Installing JavaBeans:

JavaBeans can be placed in the JavaModules folder in the Lasso Professional 8 application folder. JavaBeans can be stored in Java archive files with .jar or .zip extensions.

It is also possible to load JavaBeans from .class files, but they must be placed in sub-folders of LassoModules that correspond to the class name The JavaBean com.mycompany.myBean class file is expected to be found in LassoModules/com/mycompany/myBean.class.

Note: JavaBeans may also be discovered by Lasso anywhere within the Java classpath.

JavaBeans Type

The [Java_Bean] type and its member tags provide access to any JavaBean that is installed in Lasso. The following table shows the JavaBean tags available. This is followed by several examples of how to use JavaBeans.

Table 1: JavaBeans Type

Tag	Description
[Java_Bean]	Instantiates a JavaBean. Requires the class name of a JavaBean.
[Java_Bean->BeanProperties]	Returns an array of property names available in a JavaBean.
[Java_Bean->SetProperty]	Sets a JavaBean property. Requires two parameters -Key and -Value that specify the property to set and the new value. May also be used with -Key='*' and no value to set a JavaBean's properties automatically from the current action parameters.
[Java_Bean->GetProperty]	Gets the value for a JavaBean property. Requires one -Key parameter that specifies the property to fetch.
[Java-Bean->…]	Functions that the JavaBean implements can be called directly as member tags of the JavaBean object.

Examples

To calculate the hue, saturation, and brightness for a color:

The example shows how to use the ColorBean JavaBean to calculate the hue, saturation, and brightness for the color red. The values the red, green, and blue properties are set. Then, the values for the hue, saturation, and brightness properties are displayed.

```
<?LassoScript
  Var: 'myBean' = (Java_Bean: 'ColorBean');
  $myBean->(SetProperty: -Keyword='red', -Value=255);
  $myBean->(SetProperty: -Keyword='green', -Value=0);
  $myBean->(SetProperty: -Keyword='blue', -Value=0);

  <br>Hue: $myBean->(GetProperty: -Keyword='hue');
  <br>Saturation: $myBean->(GetProperty: -Keyword='saturation');
  <br>Color: $myBean->(GetProperty: -Keyword='brightness');
?>
```

➜
```
Hue: 0
Saturation: 100
Brightness: 100
```

To compile and execute arbitrary Java code:

The StatementBean available from the following Web site allows arbitrary Java code to be compiled and executed within Lasso. The Java archive must be downloaded and installed in LassoModules before the JavaBean can be used.

```
http://www.fuka.info.waseda.ac.jp/Project/CBSE/fukabeans/statementbean.jar
```

The JavaBean is used as follows. The Java code to be executed is stored in a variable myJavaCode. The JavaBean is instantiated using the [Java_Bean] tag with the class path of the JavaBean. The property statement is set to the desired Java code. Finally, the evaluate() function of the JavaBean is called to execute the Java code and return the result.

```
<?LassoScript
  Var: 'myJavaCode' = 'new java.util.Date().toString();';

  Var: 'myBean' = (Java_Bean: 'net.washizaki.statement.StatementBean');
  $myBean->(SetProperty: -Key='statement', -Value=$myJavaCode);
  $myBean->Evaluate;
?>
```

➜
```
July 12, 2004
```

Creating JavaBeans

The JavaBeans name comes from a Java API specification for a component architecture for the Java language. The JavaBeans Specification describes the complete set of characteristics that makes an arbitrary Java class a JavaBean or not. The full text of the document is available at the following URL:

```
http://java.sun.com/products/javabeans/
```

JavaBeans are regular java classes that must follow a set of syntactical rules. In its simplest form, a JavaBean can implement a number of "properties", whose values can be accessed/manipulated through a pair of "get" or "set" methods for each property name. The following points must be considered by JavaBean developers when developing JavaBeans for use with Lasso:

- The class must be public, and provide a public constructor that accepts no arguments.
- JavaBeans have a no arguments constructor. The configuration of each bean's behavior must be accomplished separately from its instantiation. This is typically done by defining a set of properties for the bean which can be used to set up initial values for the bean and to modify the behavior of the bean.
- Each bean property will have a public "getter" and "setter" method that are used to retrieve or define the property's value, respectively. The JavaBeans specification defines a design pattern for these names, using "get" or "set" as the prefix for the property name with it's first character capitalized, e.g.:

```
public String getFirstName();
public void setFirstName(String firstName);
```

- If a bean has both a getter and a setter method for a property, the data type returned by the getter must match the data type accepted by the setter. In addition, it is contrary to the JavaBeans specification to have more than one setter with the same name, but different property types.

Note: The JavaBeans specification also describes many additional design patterns for event listeners, wiring JavaBeans together into component hierarchies, and other useful features that are beyond the scope of this document.

Example JavaBean Source

Part of the source code for the example ColorBean JavaBean that is used throughout this chapter is provided below. The JavaBean consists of a private member variable that stores a Java color. The getters and setters for

the different properties simply pass their arguments through to this color object. Only the getter and setter for the red property are shown, but the getters and setters for green, blue, hue, saturation, and brightness are similar.

```
public class ColorBean
{

  private java.awt.Color color;

  public ColorBean()
  {
    // Default Zero Argument Constructor
  }

  public void setRed(int red)
  {
    color.setRed(red);
  }

  public int getRed()
  {
    return color..getRed();
  }

  … Other Properties …

}
```

VI

Section VI
Programming

This section includes chapters about programming concepts in Lasso 8.

- *Chapter 35: Namespaces* includes information about Lasso's tag namespaces including how to load on-demand tag libraries.
- *Chapter 36: Logging* includes information about how to log messages using Lasso's built-in logging system, how to route logged messages, and how to log data to files.
- *Chapter 37: Encryption* includes information about Lasso's built-in encryption, hash, and cipher tags.
- *Chapter 38: Control Tags* includes information about tags which allow the environment in which a page is being processed to be examined and modified.
- *Chapter 39: Threads* includes information about Lasso's thread lock tags and thread communication tags including semaphores and pipes.
- *Chapter 40: Tags and Compound Expressions* includes information about creating and running tags and compound expressions
- *Chapter 41: Miscellaneous* includes documentation of a number of tags that did not fit in any other chapter.

35

Chapter 35
Namespaces

All of Lasso's tags are categorized into namespaces that correspond roughly to the category of each tag. This chapter includes an introduction to namespaces and the tags that allow manipulation of namespaces.

- *Overview* includes basic information about namespaces including the built-in namespaces and how Lasso finds tags in different scopes.
- *Name Space Tags* includes documentation of the tags that allow namespaces themselves to be manipulated and for on-demand tag libraries to be managed.

Overview

A namespace in Lasso is a collection of similarly named tags. The namespace of built-in tags can usually be determined by looking at the category name preceding the underscore in the tag name. For example, [String_ReplaceRegExp] is in the String_ namespace and [Client_IP] is in the Client_ namespace.

The underscore character _ is used as the namespace path separator. The list of namespaces immediately preceding the actual tag name designate which namespaces should be searched for the specified tag. Namespaces are always written as Example_ with a trailing underscore.

Scope

Lasso maintains three different scopes in which tags are defined. These correspond roughly to the three variable scopes: global, page, and local. Every tag that is defined within Lasso exists within one of these three scope namespaces. Each scope has its own collection of namespaces that last as

long as the scope does. Each scope also has a defualt namespace which is named with a preceding underscore as _Global_, _Page_, and _Current_. They are defined as follows:

- **_Global_** is where any built-in tags that are not in another namespace are defined. For example [Field] is not in a namespace so can be found in the _Global_ namespace and may be referred to as [_Global_Field].
- **_Page_** which is where any custom tags defined on the current page which do not reference a namespace are registered.
- **_Current_** which is an alias to the current namespace. This will reference _Global_ if called at startup, _Page_ if called within an LDML page, or the namespace of a custom tag if called within a custom tag call.

Prepending the scope namespace explicitly tells Lasso which tag to use without performing a search of the available namespaces.

Namespace Search Order

When a tag is called Lasso searches all of the namespaces in the current scope for the specified tag. If it is not found then the namespaces in the next higher scope is searched. By default the namespaces are searched in the following order:

1. If within a custom tag call, the local scope is searched first. The scope of each surrounding tag call is searched in order for the specified tag.
2. The page scope is searched next. If a tag has been defined on the current page with the desired name then that tag is used.
3. The global scope is finally searched. All of the built-in tags are defined within this scope

The [Namespace_Using] ... [/Namespace_Using] tag can be used to alter the default namespace search order. The opening container tag accepts the name of a namespace that should be searched first. The remainder of the namespaces are then searched in default order.

- A series of tags within a given namespace can be referenced more simply by referencing a specific namespace within the opening [Namespace_Using] tag. For example, a series of string operations can be performed by specifying the String_ namespace.

```
[Namespace_Using: 'String_']
  [Var: 'Test' = (LowerCase: 'Example    String');
  [Var: 'Test' = (ReplaceRegExp: $Test, -Find='\\w+', -replace=' ')]
[/NameSpace_Using]
```

Third-Party Namespaces

All built-in tags consist of a namespace followed by an underscore. For example [Namespace_TagName]. Tags provided by third parties will usually have a company name prepended to each tag name. For example a tag provided by OmniPilot might be named [OP_Namespace_TagName].

The [Namespace_Using] tag can be used to search this third party OP_ namespace first. This allows all of the third-party tags to be called using just [NameSpace_TagName] without requiring the company prefix on every tag.

```
[Namespace_Using: 'OP_']
  [Namespace_TagName]
  ...
[/Namespace_Using]
```

This technique has the added benefit of allowing tags within the OP_ namespace to implicitly override built-in tags with the same names. For example, the tag [OP_Field] could be defined. Within appropriate [Namespace_Using] ... [/Namespace_Using] tags this tag would be used instead of the built-in [Field] tag.

```
[Namespace_Using: 'OP_']
  [Field: 'First_Name']
[/Namespace_Using]
```

On-Demand Libraries

On-demand libraries allow collections of tags to be loaded only when they are first used. When a tag is called that is not defined in Lasso, the LassoLibraries folder at the site and master levels within the Lasso Professional 8 application folder are searched for an LDML page or LassoApp that defines the desired namespace.

For example, if the tag [Example_Tag] is called then Lasso will search the site level LassoLibraries folder for a file named Example.Lasso or Example.LassoApp. If no such file is found then the master level LassoLibraries folder will be searched.

If the library is found and defines the called tag then the tag will be executed normally. The tag definitions within the file are all added to a namespace in the global scope so they can be used on other LDML pages.

Many built-in tags in Lasso are installed as on-demand libraries including Cache_, Client_, Link_, Thread_, Valid_, WAP_, and more. These tags will be loaded on their first use and will be available from then on from any page processed by Lasso Service.

Namespace Tags

There are four tags defined within the Namespace_ namespace. Each of the tags is described in the following table with examples of their use after.

Table 1: Namespace Tags

Tag	Description
[Namespace_Using] ... [/Namespace_Using]	Uses the specified namespace as the start of the search order when looking up a tag. Requires a single parameter which is the namespace to use.
[Namespace_Load]	Attempts to load the specified namespace from the LassoLibraries folders. Requires a single parameter which is the namespace to load.
[Namespace_Unload]	Unloads the specified namespace. Requires a single parameter which is the namespace to unload.
[Namespace_Import]	Imports a tag or collection of tags into a namespace. The first parameter specifies either a tag name or namespace name. The second parameter -Into specifies the destination namespace.

To change the order of searched namespaces:

Use the [Namespace_Using] ... [/Namespace_Using] tags. The opening tag requires one parameter which is the name of the namespace that should be searched first. The namespace can be a built-in namespace such as _Global_ or _Page_ or a specific namespace such as String_ or OP_. Note that namespace names are always written with a trailing undesrcore.

For example, the following code uses the Date_ namespace allowing date tags to be called using short tag names.

```
[Namespace_Using: 'Date_']
  [Var: 'myDate' = Date]
  [Var: 'myDate' = (Add: $myDate, -Hour=5)]
  [Var: 'myDate' = (Subtract: $myDate, -Hour=5)]
  [Format: $myDate, -Format='%D %T']
[/Namespace_Using]
```

To load an on-demand tag library:

Use the [Namespace_Load] tag. This is not normally needed since tags will be loaded on-demand the first time they are used, but can be useful to force a library to load explicitly. For example, the following tag would force the thread library to load.

```
[Namespace_Load: 'Thread_']
```

To unload a namespace or an on-demand tag library:

Use the [Namespace_Unload] tag. This will cause any tags within the specified namespace to unload. If the tags were loaded from an on-demand library then they will be reloaded when they are next used (or the [Namespace_Load] tag can be called explicilty). For example, the following tag would force the thread library to unload.

```
[Namespace_Unload: 'Thread_']
```

Unloading a library can be useful if an update to the library is installed in the Lasso Professional 8 application folder. The new tag definitions can be used without having to restart Lasso Service or the current site.

To import tags from one namespace into another:

Use the [Namespace_Import] tag. This tag can be used to import single tags or all tags within a namespace into another namespace. The tag requires two parameters: the first is the name of a tag or namespace which should be copied and the second is the name of the namespace to import into.

The [Namespace_Import] tag is most often used within startup. This tag allows tags to be defined using a third-party company prefix as recommended. Then the tags can be imported into other namesapces as needed.

For example, the tag [OP_Valid_PhoneNumber] could be imported into the Valid_ namespace. This would have the effect of allowing the tag to be called as [Valid_PhoneNumber].

```
[Namespace_Import: OP_Valid_PhoneNumber, -into='Valid_']
```

36

Chapter 36
Logging

LDML has a built-in log routing system that allows built-in messages to be routed to several destination. LDML provides two methods of logging errors or page accesses.

- *Overview* describes the built-in log routing system for messages generated by Lasso.
- *Log Tags* describes the [Log_...] tags that allow errors, warnings, details, action statements, and deprecated warning messages from LDML code to be written to Lasso's internal error logs.
- *Log Files* describes the [Log] ... [/Log] tags which allow data to be written to a text file.
- *Log Routing* describes the tags that can be used to alter the routing of messages to Lasso's internal error logs.

Note: See the chapter on Files for general information about writing data to files.

Overview

Lasso Professional 8 has a built-in error logging system which allows warning messages to be logged at several different levels. Each log level can be routed to one or more destinations allowing for a great deal of flexibility in handling

The built-in log levels include:

- **Error** – Critical errors that affect the operation of Lasso Service. Critical errors are logged to all destination by default. Typically, the server or site administrator will need to fix whatever is causing the critical error.

- **Warning** – Warnings are informative messages about possible problems with the functioning of Lasso Service. Warnings do not always require action by the server or site administrator. Warnings are logged only to the console by default.
- **Detail** – Detailed messages about the normal functioning of Lasso Service. Includes status messages from the email queue and event scheduler, etc. Detail messages are logged only to the console by default.
- **Action Statement** – The SQL statements generated by SQL data sources are logged as action statements. Other data sources may also log their implementation-specific action statements. Action statements are logged only to the console by default.
- **Deprecated** – Flags any use of deprecated functionality in LDML code. Deprecated tags are supported in this version of Lasso, but may not be supported in a future version. Any deprecated functionality should be updated to new, preferred syntax for best compatibility with future versions of Lasso. Deprecated messages are logged only to the console by default.

The destinations which the log levels can be routed to include:

- **Database** – The _Errors table in the Lasso_Internal Lasso MySQL database, viewable via the *Utility > Errors > Lasso Errors* page in Lasso Administration.
- **Console** – The Lasso Service console window. Viewable if Lasso Service is started in console mode.
- **LassoErrors.txt** – The LassoErrors.txt file, located in the appropriate site folder in the Lasso Professional 8 application folder.

The routing of Lasso's internal log levels can be modified in the *Utility > Errors > Setup* section of Site Administration. For information on setting up Lasso's internal error routing, see the *Site Utilities* chapter in the Lasso Professional 8 Setup Guide.

See the following section on *Log Tags* for details about how to log messages from a Lasso solution using the built-in error message routing system. For details about how to change the log level routing programmatically see the subsequent *Log Routing* section.

Log Tags

The [Log_Critical], [Log_Warning], [Log_Detail], and [Log_Deprecated] tags are used to log custom data to the Lasso internal error logs with a defined Lasso error level of Crtical, Warning, Detail, or Deprecated. The following example outputs the date and time of a page request (with a literal space between) to Lasso's internal error logs with an error level of Detail.

```
[Log_Detail: (Server_Date) + ' ' + (Server_Time)]
```

Table 1: Lasso Error Log Tags

Tag	Description
[Log_Critical]	Logs to Lasso's internal error logs with an error level assignment of Critical. Requires the text to be logged as a parameter. Logging options for this error level are set in Lasso Administration.
[Log_Warning]	Logs to Lasso's internal error logs with an error level assignment of Warning. Requires the text to be logged as a parameter. Logging options for this error level are set in Lasso Administration.
[Log_Detail]	Logs to Lasso's internal error logs with an error level assignment of Detail. Requires the text to be logged as a parameter. Logging options for this error level are set in Lasso Administration.
[Log_SQL]	Logs to Lasso's internal error logs with an error level assignment of Action Statement. Requires the text to be logged as a parameter. Logging options for this error level are set in Lasso Administration.
[Log_Deprecated]	Logs to Lasso's internal error logs with an error level assignment of Deprecated. Requires the text to be logged as a parameter. Logging options for this error level are set in Lasso Administration.
[Log_Always]	Logs to Lasso's console. This error level cannot be routed, but is always sent to Lasso's console.

To log format file errors to the Lasso Service console and Lasso Administration:

Use the [Log_Critical], [Log_Warning], [Log_Detail], or [Log_Deprecated] tags. This will log any information contained in the tags in Lasso's internal error logs with a Lasso error level of Critical, Warning, Detail, or Deprecated. The following example will log a warning to Lasso's internal logs if an Out Of Memory error occurs while processing the format file.

```
[If: (Error_CurrentError) == (Error_OutOfMemory)]
  [Log_Warning: 'A memory error occured while processing this page.']
[/If]
```

➜ Warning: A memory error occured while processing this page.

If the Lasso Errors Database and Lasso Service Console options were selected for Warning in the *Utility > Errors > Setup* page in Lasso Administration, then this message will be logged and displayed in both the Lasso Service console window and the *Utility > Errors > Lasso Errors* page in Lasso Administration.

Log Files

In addition to using the built-in log level routing system, it is sometimes desirable to create a separate log file specific to a custom solution. The [Log] … [/Log] tags can be used to write text messages out to a text log file.

When executed, the contents of the [Log] ... [/Log] container tags is appended to a specified text file. The [Log] ... [/Log] tags can write to any text file that is in the Web server root and accessible from Lasso. All returns, tabs, and spaces between the [Log] ... [/Log] tags will be included in the output data.

The following [Log] ... [/Log] tags output a single line containing the date and time with a return at the end to the file specified. The tags are shown first with a Windows path, then with a Mac OS X path.

```
[Log: 'C://Logs/LassoLog.txt'][Server_Date] [Server_Time]
[/Log]

[Log: '///Logs/LassoLog.txt'][Server_Date] [Server_Time]
[/Log]
```

The path to the directory where the log will be stored should be specified according to the same rules as those for the file tags. See the *File Tags* section of this chapter for full details about relative, absolute, and fully qualified paths on both Mac OS X and Windows.

Table 2: File Log Tags

Tag	Description
[Log] ... [/Log]	Logs the contents of the container tags to a specified text file. Requires the path to the text file as a parameter: An optional -Encoding parameter specifies the character set to use to write the log file (defaults to Mac-Roman on Mac OS X and ISO-8859-1 on other platforms.

To log site visits to a file:

Use the [Server_...] and [Client_...] tags to return information about the current visitor and what page they are visiting. The following code will log the current date and time, the visitor's IP address, the name of the server and the page they were loading, and the GET and POST parameters that were specified.

```
[Log: 'E://Logs/LassoLog.txt'][Server_Date: -Extended] [Server_Time: -Extended]
[Client_IP] [Server_Name] [Response_FilePath] [Client_GETArgs] [Client_POSTArgs]
[/Log]
```

See the *HTTP/HTML Content and Controls* chapter for more information a about the [Client_...] and [Server_...] tags.

To automatically roll log files by date:

Include a date component in the name of the log file. Since the date component will change every day, a new log file will be created the first time an item is logged each day. [Server_Date: -Extended] creates a safe date format to use. The following example logs to a file named e.g. 2001-05-31.txt.

```
[Variable: 'Log_File' = ///Logs/' + (Server_Date: -Extended) + '.txt']
[Log: (Variable: 'Log_File'][Server_Date] [Server_Time]
[/Log]
```

Log Routing

The tag for setting log routing is described in the *Log Preference Tag* table. Log preferences can be viewed or changed in the *Utility > Errors > Setup* section of Lasso Administration. Use of this tag is only necessary to change the log settings programmatically.

Table 3: Log Preference Tag

Tag	Description
[Log_SetDestination]	The first parameter specifies a log message level. Subsequent parameters specify the destination to which that level of messages should be logged.

Note: The [Log_SetDestination] tag can only be used by the global administrator. Use an [Auth_Admin] tag to authorize use of this tag.

The first parameter of [Log_SetDestination] requires a log message level. The three available log message levels are detailed in the *Log Message Levels* table.

Table 4: Log Message Levels

Level	Description
Log_Level_Critical	Critical error messages that affect the proper functioning of Lasso Service or requires action by the administrator.
Log_Level_Warning	Informative messages about what actions are being performed by Lasso Service. Generally do not require action by the administrator.
Log_Level_Detail	Detailed messages about the inner workings of Lasso Service.
Log_Level_SQL	Action statements generated by inline database actions. SQL statements are logged at this level.
Log_Level_Deprecated	Warnings about the use of deprecated functionality in Lasso.

Subsequent parameters of [Log_SetDestination] require a destination code. The three available destinations available are detailed in *Table 6: Log Destination Codes*.

Table 5: Log Destination Codes

Code	Description
Log_Destination_Console	Messages are logged to the Lasso Service console. Visible on Windows 2000 when Lasso Service is launched as an application and on Mac OS X when the consoleLassoService.command script is used.
Log_Destination_File	Messages are logged to the LassoErrors.txt file which is created in the same folder as Lasso Service.
Log_Destination_Database	Messages are logged to the errors table of the site database which can be viewed in the Utility > Errors section of Lasso Administration.

To change the log preferences:

Use the [Log_SetDestination] tag to change the destination of a given log message level. In the following example, detail messages are sent to the console and to the errors table of the site database.

```
[Auth_Admin]
[Log_SetDestination: Log_Level_Detail,
  Log_Destination_Database, Log_Destination_Console]
```

To reset the log preferences:

The following four commands reset the log preferences to their default values. Critical errors are sent to all three destinations. Warnings, detail, and deprecation warning messages are sent only to the console.

```
[Auth_Admin]
[Log_SetDestination: Log_Level_Critical,
  Log_Destination_Console, Log_Destination_Database, Log_Destination_File]
[Log_SetDestination: Log_Level_Warning, Log_Destination_Console]
[Log_SetDestination: Log_Level_Detail, Log_Destination_Console]
[Log_SetDestination: Log_Level_SQL, Log_Destination_Console]
[Log_SetDestination: Log_Level_Deprecated, Log_Destination_Console]
```

37

Chapter 37
Encryption

Lasso's built-in encryption tags allow data to be stored or transmitted securely.

- *Overview* describes the different encryption types that Lasso supports.
- *Encryption Tags* describes Lasso's built-in tags for securely storing and transmitting data using industry standard algorithms.
- *Cipher Tags* describes tags that allow many different encryption and hash algorithms to be used within Lasso.
- *Compression Tags* describes tags for compressing string data for more efficient storage or transmission.

Overview

Lasso provides a set of data encryption tags which support the most commonly used encryption and hash functions used on the Internet today. These encryption tags make it possible to interoperate with other systems that require encryption and to store data in a secure fashion in data sources or files.

Lasso has built-in tags for the BlowFish encryption algorithm and for the SHA, SHA2, and MD5 hash algorithms. The new BlowFish2 algorithm is implemented in an industry standard fashion in order to allow values encrypted with Lasso to be decrypted by other BlowFish implementation or vice versa. (The original BlowFish algorithm is also provided for backward compatibility.)

Lasso's cipher tags provide access to a wide range of industry standard encryption algorithms. The [Cipher_List] tag lists what algorithms are available and the [Cipher_Encrypt], [Cipher_Decrypt], and [Cipher_Digest] tags allow values to be encrypted or decrypted or digest values to be generated.

Finally, Lasso provides a set of tags to compress or decompress data for more efficient data transmission.

Encryption Tags

LDML provides a number of tags which allow data to be encrypted for secure storage or transmission. Three different types of encryption are supplied.

- **BlowFish** is a fast, popular encryption algorithm. Lasso provides tools to encrypt and decrypt string values using a developer-defined seed. This is the best tag to use for data which needs to be stored in a database or transmitted securely.

  Note: Lasso Professional 8 includes a new implementation of BlowFish which should be interoperable with most other products that support BlowFish. The new algorithm is implemented as [Encrypt_BlowFish2]. The older algorithm is still supported as [Encrypt_BlowFish] for backward compatibility.

- **MD5** is a one-way encryption algorithm that is often used for passwords. There is no way to decrypt data which has been encrypted using MD5. See below for an example of how to use MD5 to store and check passwords securely.
- **SHA** is a one-way encryption algorithm that is often used for passwords. There is no way to decrypt data which has been encrypted using SHA.
- **SHA2** is a one-way encryption algorithm that is often used for passwords. There is no way to decrypt data which has been encrypted using SHA2.

Table 1: Encryption Tags

Tag	Description
[Encrypt_BlowFish2]	Encrypts a string using an industry standard BlowFish algorithm. Accepts two parameters, a string to be encrypted and a -Seed keyword with the key or password for the encryption.
[Decrypt_BlowFish2]	Decrypts a string encrypted using the industry standard BlowFish algorithm. Accepts two parameters, a string to be decrypted and a -Seed keyword with the key or password for the decryption.
[Encrypt_BlowFish]	Encrypts a string using the BlowFish implementation from earlier versions of Lasso. Accepts two parameters, a string to be encrypted and a -Seed keyword with the key or password for the encryption.
[Decrypt_BlowFish]	Decrypts a string encrypted by the BlowFish implementation from earlier versions of Lasso. Accepts two parameters, a string to be decrypted and a -Seed keyword with the key or password for the decryption.
[Encrypt_MD5]	Encrypts a string using the one-way MD5 hash algorithm. Accepts one parameter, a string to be encrypted. Returns a fixed size hash value in hexadecimal for the string which cannot be decrypted.

Note: The BlowFish tags are not binary safe. The output of the tag will be truncated after the first null character. It is necessary to use [Encode_Base64] or [Encode_UTF8] prior to encrypting data that might contain binary characters using these tags.

BlowFish Seeds

BlowFish requires a seed in order to encrypt or decrypt a string. The same seed which was used to encrypt data using the [Encrypt_BlowFish2] tag must be passed to the [Decrypt_BlowFish2] tag to decrypt that data. If you lose the key used to encrypt data then the data will be essentially unrecoverable.

Seeds can be any string between 4 characters and 112 characters long. Pick the longest string possible to ensure a secure encryption. Ideal seeds contain a mix of letters, digits, and punctuation.

The security considerations of storing, transmitting, and hard coding seed values is beyond the scope of this manual. In the examples that follow, we present methodologies which are easy to use, but may not provide the highest level of security possible. You should consult a security expert if security is very important for your Web site.

Note: The BlowFish algorithm will return random results if you attempt to decrypt data which was not actually encrypted using the same algorithm.

To store data securely in a database:

Use the [Encrypt_BlowFish2] and [Decrypt_BlowFish2] tags to encrypt data which will be stored in a database and then to decrypt the data when it is retrieved from the database.

1 Store the data to be encrypted into a string variable, PlainText.

```
[Variable: 'PlainText' = 'The data to be encrypted.']
```

2 Encrypt the data using the [Encrypt_BlowFish2] tag with a hard-coded -Seed value. Store the result in the variable CipherText.

```
[Variable: 'CipherText' = (Encrypt_BlowFish2: (Variable: 'PlainText'),
    -Seed='This is the blowfish seed')]
```

3 Store the data in CipherText in the database. The data will not be viewable without the seed. The following [Inline] ... [/Inline] creates a new record in an Contacts database for John Doe with the CipherText.

```
[Inline: -Add,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    'First_Name'='John',
    'Last_Name'='Doe',
    'CipherText'=(Variable: 'CipherText')]
[/Inline]
```

4 Retrieve the data from the database. The following [Inline] ... [/Inline] fetches the record from the database for John Doe and places the CipherText into a variable named CipherText.

```
[Inline: -Search,
    -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    'First_Name'='John',
    'Last_Name'='Doe']
  [Variable: 'CipherText' = (Field: 'CipherText')]
[/Inline]
```

5 Decrypt the data using the [Decrypt_BlowFish2] tag with the same hard-coded -Seed value. Store the result in the variable PlainText.

```
[Variable: 'PlainText' = (Decrypt_BlowFish2: (Variable: 'CipherText'),
    -Seed='This is the blowfish seed')]
```

6 Display the new value stored in PlainText.

```
[Variable: 'PlainText']
```

```
➜ The data to be encrypted.
```

Note: This example uses the [Encrypt_BlowFish2] and [Decrypt_BlowFish2] tags. These are the preferred BlowFish implementation to use with Lasso. The [Encrypt_BlowFish] and [Decrypt_BlowFish] tags should only be used for interoperability with older versions of Lasso.

To store and check encrypted passwords:

The [Encrypt_MD5] tag can be used to store a secure version of a password for a site visitor. On every subsequent visit, the password given by the visitor is encrypted using the same tag and compared to the stored value. If they match, then the visitor has supplied the same password they initially supplied.

1. When the visitor creates an account use [Encrypt_MD5] to create an encrypted version—a fixed size hash value—of the password they supply. In the following example, the password they supply is stored in the variable VisitorPassword and the encrypted version is stored in SecurePassword.

```
[Variable: 'SecurePassword' = (Encrypt_MD5: (Variable: 'VisitorPassword'))]
```

2. Store this MD5 hash value for the password in a database along with the visitor's username.
3. On the next visit, prompt the visitor for their username and password. Fetch the record identified by the visitor's specified username and retrieve the MD5 hash value stored in the field SecurePassword.
4. Use [Encrypt_MD5] to encrypt the password that the visitor has supplied and compare the result to the stored, encrypted MD5 hash value that was generated from the password they supplied when they created their account.

```
[If: (Encrypt_MD5: (Variable: 'VisitorPassword')) == (Field: 'SecurePassword')]
  Log in successful.
[Else]
  Password does not match.
[/If]
```

Note: For more security, most log-in solutions require both a username and a password. The password is not checked unless the username matches first. This prevents site visitors from guessing passwords unless they know a valid username. Also, many log-in solutions restrict the number of login attempts that they will accept from a client's IP address.

Cipher Tags

Lasso includes a set of tags that allow access to a wide variety of encryption algorithms. These cipher tags provide implementations of many industry standard encryption methods and can be very useful when communicating using Internet protocols or communicating with legacy systems.

The table below lists the [Cipher_...] tags in Lasso. The following tables list several of the cipher algorithms and digest algorithsm that can be used with the [Cipher_...] tags. The [Cipher_List] tag can be used to list what algorithms are supported in a particular Lasso installation.

Note: The actual list of supported algorithsm may vary from Lasso installation to Lasso installation depending on platform and system version. The algorithms listed in this manual should be available on all systems, but other more esoteric algorithms may be available on some systems and not on others.

Table 2: Cipher Tags

Tag	Description
[Cipher_Encrypt]	Encrypts a string using a specified algorithm. Requires three parameters. The data to be encrypted, a -Cipher parameter specifying what algorithm to use, and a -Key parameter specifying the key for the algorithm. An optional -Seed parameter can be used to seed algorithms with a random component.
[Cipher_Decypt]	Decrypts a string using a specified algorithm. Requires three parameters. The data to be decrypted, a -Cipher parameter specifying what algorithm to use, and a -Key parameter specifying the key for the algorithm.
[Cipher_Digest]	Encrypts a string using a specified digest algorithm. Requires two parameters. The data to be encrypted and a -Digest parameter that specifies the algorithm to be used. Optional -Hex parameter encodes the result as a hexadecimal string.
[Cipher_List]	Lists the algorithms that the cipher tags support. With a -Digest parameter returns only digest algorithms. With -SSL2 or -SSL3 returns only algorithms for that protocol.

The following two tables list some of the cipher algorithms that can be used with [Cipher_Encrypt] and some of the digest algorithms that can be used with [Cipher_Digest]. Use [Cipher_List] for a full list of supported algorithms.

Table 3: Cipher Algorithms

Algorithm	Description
AES	Advanced Encryption Standard. A symmetric key encryption algorithm which is slated to be the replacement for DES. An implementation of the Rijndael algorithm.
DES	Data Encryption Standard. A block cipher developed by IBM in 1977 and used as the government standard encryption algorithm for years.
3DES	Triple DES. This algorithm uses the DES algorithm three times in succession with different keys.
RSA	A public key algorithm named after Rivest, Shamir, and Adelmen. One of the most commonly used encyrption algorithsm. Note: Lasso does not generate public/private key pairs.

Table 4: Digest Algorithms

Algorithm	Description
DSA	Digital Signature Algorithm. Part of the Digital Signature Standard. Can be used to sign messages, but not for general encryption.
SHA1	Secure Hash Algorithm. Produces a 160-bit hash value. Used by DSA.
MD5	Message Digest. A hash function that generates a 128-bit message digest. Replaces the MD4 and MD2 algorithms (which are also supported). Also implemented in Lasso as [Encrypt_MD5].

To list all supported algorithms:

Use the [Cipher_List] tag. The following tag will return a list of all the cipher algorithms supported by Lasso

```
[Cipher_List]
```

With a -Digest parameter the tag will return a list of all the digest algorithms supported by Lasso.

```
[Cipher_List: -Digest]
```

To calculate a digest value:

Use the [Cipher_Digest] tag. The following tag will return the DSA signature for the value of a database field Message.

```
[Cipher_Digest: (Field: 'Message'), -Digest='DSA']
```

To encrypt a value using 3DES:

Use the [Cipher_Encrypt] tag. The following tag will return the 3DES encryption for the value of a database field Message.

```
[Cipher_Encrypt: (Field: 'Message'), -Cipher='3DES', -Key='My Secret Key']
```

Compression Tags

LDML provides two tags which allow data to be stored or transmitted more efficiently. The [Compress] tag can be used to compress any text string into an efficient byte stream that can be stored in a text field in a database or transmitted to another server. The [Decompress] tag can then be used to restore a compressed byte stream into the original string.

The compression algorithm should only be used on large string values. For strings of less than one hundred characters the algorithm may actually result in a larger string than the source.

These tags can be used in concert with the [Null->Serialize] tag that creates a string representation of any data type in LDML and the [Null->UnSerialize] tag that returns the original value based on a string representation. An example below shows how to compress and decompress an array variable.

Table 5: Compression Tags

Tag	Description
[Compress]	Compresses a string parameter.
[Decompress]	Decompresses a byte stream.

To compress and decompress a string:

1 Use the [Compress] tag on the variable InputVariable holding the string value you want to compress. The result is a byte stream that represents the string which is stored in CompressedVariable.

```
[Variable: 'InputVariable'='This is the string to be compressed.']
[Variable: 'CompressedVariable'=(Compress: $InputVariable)]
```

2 The CompressedVariable can now be decompressed using the [Decompress] tag. The result is stored in OutputVariable and finally displayed.

```
[Variable: 'OutputVariable'=(Decompress: $CompressedVariable)]
[Variable: 'OutputVariable']
```

➜ This is the string to be compressed.

To compress and decompress an array:

1 Store the array in a variable ArrayVariable.

```
[Variable: 'ArrayVariable'=(Array: 'one', 'two', 'three', 'four', 'five')]
```

2 Use the [Null->Serialize] tag to change the array into a string stored in InputVariable.

```
[Variable: 'InputVariable'=$ArrayVariable->Serialize]
```

3 Use the [Compress] tag on the variable InputVariable holding the string representation for the array. The result is a byte stream which is stored in CompressedVariable.

```
[Variable: 'CompressedVariable'=(Compress: $InputVariable)]
```

4 The CompressedVariable can now be decompressed using the [Decompress] tag. The result is a string stored in OutputVariable.

```
[Variable: 'OutputVariable'=(Decompress: $CompressedVariable)]
```

5 The string representation of the array can now be changed back into the array by creating a new variable ArrayVariable and then calling the [Null->UnSerialize] tag with OutputVariable as a parameter.

```
[Variable: 'ArrayVariable'=Null]
[$ArrayVariable->(UnSerialize: $OutputVariable)]
```

6 Finally, the original array can be output.

```
[Variable: 'ArrayVariable']
➜ (Array: (one), (two), (three), (four), (five))
```

38

Chapter 38
Control Tags

This chapter documents tags that allow format files to be scheduled for execution and tags that allow low-level access to Lasso's internal variables.

- *Authentication Tags* details the tags that allow security settings to be modified.
- *Administration Tags* details the tags that provide access to Lasso's environment.
- *Scheduling Events* documents how to use the [Event_Schedule] tag to schedule the loading of format files.
- *Process Tags* documents how to programmatically control what code is processed by Lasso, and how to pause Lasso execution.
- *Configuration Tags* allow the configuration of Lasso to be inspected.
- *Page Variables* documents the internal variables Lasso uses while processing a page and tags that allow access to them.
- *Null Data Type* documents the base data type and the member tags common to all data types in LDML.
- *Format File Execution Time Limit* describes the built-in time limit on the lenght of time that format files are allowed to execute.
- *Page Pre- and Post-Processing* describes how to execute code before every page load on a site or after a page load.

Authentication Tags

The authentication tags can be used to ensure that all of the code in a page is run by a registered user in Lasso Security or the global administrator. The authentication tags work by performing the following tasks when they are executed:

1 Check the current username and password stored in the client browser. If the username and password meet the requirements of the authentication tag used, then the page is served normally.

2 Otherwise, a browser authentication dialog box is shown to the visitor.

3 If the client enters a valid username and password, then the page is served normally.

4 Otherwise, the visitor is either prompted for a username and password again or is shown an error. The actual behavior is determined by the Web browser software.

Table 1: Authentication Tags

Tag	Description
[Auth]	Allows only configured Lasso users to view the page. Prompts for a username and password if the current visitor has not provided a valid username and password.
[Auth_Admin]	Allows only the global administrator to view the page. Prompts for a username and password if the current visitor is not the global administrator.
[Auth_Group]	Allows only configured Lasso users in a specified group in Lasso Security to view the page. Requires the name of a Lasso group (or an array of group names) as a parameter. Prompts for a username and password if the current visitor has not provided a valid username and password.
[Auth_User]	Allows a single specified user in Lasso Security to view the page. Requires the name of a Lasso user (or an array of user names) as a parameter. Prompts for a username and password if the current visitor has not provided a valid username and password.
[Auth_Custom]	Allows a single user with a specified username and password to view the page. Requires a custom username (or array of usernames), password (or array of corresponding passwords), and realm as parameters. The custom username does not have to be configured in Lasso Security.

To prompt a visitor for authentication:

Use the [Auth] tag at the top of a format file. Each visitor will need to enter a username and password which is configured within Lasso Administration in order to view the contents of the Web page.

```
[Auth]
```

This tag can be used to ensure that only configured users visit a format file. No Anonymous users or users who do not have a valid username and password will be allowed.

To restrict a page so only the global administrator can view it:

Use the [Auth_Admin] tag at the top of a format file. Each visitor will need to enter the global administrator's username and password in order to view the contents of the Web page.

```
[Auth_Admin]
```

This can be used to hide format files which provide status information that only the global administrator should be able to read or to protect custom format files that allow aspects of a Web site to be administered.

To restrict a page to users in a specific Lasso group:

Use the [Auth_Group] tag with the name of the desired group as a parameter. Each visitor will need to enter a username and password that belongs to the specified Lasso group in order to view the contents of the Web page.

```
[Auth_Group: 'Lasso_Group_Name']
```

To restrict a page to a specific Lasso user:

Use the [Auth_User] tag with the name of the desired user as a parameter. Each visitor will need to enter the username and password of the Lasso user specified in order to view the contents of the Web page.

```
[Auth_User: 'Lasso_User_Name']
```

To restrict a page to a custom username and password:

Use the [Auth_Custom] tag with a custom username, password, and realm as parameters (the parameters must be entered in this order). Each visitor will need to enter this username and password in order to view the contents of the Web page.

```
[Auth_Custom: 'Custom_User_Name', 'Custom_Password', 'Custom_Realm']
```

This tag is useful for authenticating a user that is not necessarily configured in Lasso Security. The custom realm will be displayed in the authentication dialog box when the user logs in, and can be used in conjunction with other realms on the Web server.

Administration Tags

Lasso Security is generally configured through the Lasso Administration interface and related LassoApps. However, Lasso also provides a number of tags that allow the security settings to be modified from within format files. These tags are summarized in *Table 2: Administration Tags*. See the *Setting Up Security* chapter of the Lasso Professional 8 Setup Guide for more information about users and groups.

Table 2: Administration Tags

Tag	Description
[Admin_CurrentUsername]	Returns the name of the current user whose permissions are being used to run the page or inline. Returns nothing for the anonymous user.
[Admin_CurrentGroups]	Returns an array of Lasso group names that the current user belongs to.
[Admin_ChangeUser]	Changes a user's password. Requires a valid Lasso username, old password, and new password as parameters. This tag can be called by any configured Lasso user.
[Admin_CreateUser]	Creates a new user with the specified password. Requires a new username and password as parameters. This tag can be called by any group administrator.
[Admin_GroupAssignUser]	Assigns the specified user to the group. Requires the name of a Lasso group and a Lasso user as parameters. This tag can only be called by a group administrator for the group.
[Admin_GroupListUsers]	Lists the users who belong to the group. This tag can only be called by a group administrator for the group.
[Admin_GroupRemoveUser]	Removes the specified user from the group. This tag can only be called by a group administrator for the group.
[Admin_ListGroups]	Lists the groups for which a group administrator has privileges.
[Admin_RefreshSecurity]	Refreshes cached security settings. This tag can be used only by the Lasso global administrator.
[Admin_ReloadDatasource]	Reloads a Lasso Data Source Connector. Requires the internal name or ID number of a Lasso Data Source Connector as a parameter (as shown in Lasso Administration). This tag can be used only by the Lasso global administrator.
[Admin_LassoServicePath]	Returns the file path to Lasso Service. This tag can be used only by the Lasso global administrator.

To return the current username within a format file:

Use the [Admin_CurrentUserName] tag. The following example displays the current Lasso user being used within an [Inline] tag.

```
[Inline: -Username='John_Doe', -Password='MyPassword']
  [Admin_CurrentUsername]
[/Inline]
```

➜ John_Doe

To change the password for a user:

Use the [Admin_ChangeUser] tag. The tag takes three parameters: the username and password of an existing user, and the new password for the user. The following example changes the password for John_Doe to MyPassword from MyOldPassword. The tag returns True if the change was successful.

```
[Admin_ChangeUser: 'John_Doe', 'MyOldPassword', 'MyPassword']
```

➜ True

To list the groups the current user belongs to:

Use the [Admin_CurrentGroups] tag. The following example lists the groups that John_Doe belongs to.

```
[Inline: -Username='John_Doe', -Password='MyPassword']
  [Admin_CurrentGroups]
[/Inline]
```

➜ (Array: (AnyUser), (Johns_Group))

To list the groups the current user can administer:

Use the [Admin_ListGroups] tag. The following example lists the groups that John_Doe can administer. The username and password for John_Doe are specified using -Username and -Password command tags in [Inline] ... [/Inline] tags surrounding the call to [Admin_ListGroups]. Since John_Doe is the administrator of one group, Johns_Group, this one name is returned in a single element array.

```
[Inline: -Username='John_Doe', -Password='MyPassword']
  [Admin_ListGroups]
[/Inline]
```

➜ (Array: (Johns_Group))

To list the users that belong to a group:

Use the [Admin_GroupListUsers] tag. The following example lists the users that belong to the group named Johns_Group. The username and password for an administrator of Johns_Group is specified in the [Inline] ... [/Inline] tags surrounding the call to [Admin_GroupListUsers]. An array of usernames is returned.

```
[Inline: -Username='John_Doe', -Password='MyPassword']
  [Admin_GroupListUsers: 'Johns_Group']
[/Inline]
```

➜ (Array: (John_Doe), (Jane_Doe), (Tex_Surname), (Bob_Peoples))

To create a new user:

Use the [Admin_CreateUser] tag. The following example creates a new user Joe_Random with the password 1234. Joe_Random will not have any permissions beyond those assigned to AnyUsers until he is assigned to a group. Since only a group administrator can create new users, the username and password for John_Doe are specified in the [Inline] ... [/Inline] tags surrounding the call to [Admin_CreateUser]. The tag returns True if the user was created or False if the username already exists.

```
[Inline: -Username='John_Doe', -Password='MyPassword']
  [Admin_CreateUser: 'Joe_Random', '1234']
[/Inline]
```

➜ True

To add a user to a group:

Use the [Admin_GroupAssignUser] tag. The following example adds the user Joe_Random to the group Johns_Group. This tag can only be called by a group administrator for Johns_Group so the username and password for John_Doe are specified in the [Inline] ... [/Inline] tags surrounding the call to [Admin_GroupAssignUser]. The tag returns True if the user is successfully added to the group.

```
[Inline: -Username='John_Doe', -Password='MyPassword']
  [Admin_GroupAssignUser: 'Johns_Group', 'Joe_Random']
[/Inline]
```

➜ True

To remove a user from a group:

Use the [Admin_GroupRemoveUser] tag. The following example removes the user Joe_Random from the group Johns_Group. This tag can only be called by a group administrator for Johns_Group so the username and password for John_Doe are specified in the [Inline] ... [/Inline] tags surrounding the call

to [Admin_GroupRemoveUser]. The tag returns True if the user is successfully removed from the group.

```
[Inline: -Username='John_Doe', -Password='MyPassword']
  [Admin_GroupRemoveUser: 'Johns_Group', 'Joe_Random']
[/Inline]
```

➜ True

To reload a data source connector:

Use the [Admin_ReloadDataSource] tag. This tag is useful if a data source has been modified and Lasso needs to be refreshed to see the new changes, and requires Lasso global administrator permission to use. The example below refreshes the Lasso MySQL data source connector, which has an internal ID number of 1 according to the *Setup > Data Sources > Connectors* section of Lasso Administration.

```
[Admin_ReloadDataSource: 1]
```

To refresh Lasso Security:

Use the [Admin_RefreshSecurity] tag. This tag is required for new settings to go into effect if Lasso Security is manually altered outside of using Lasso Administration. This tag requires Lasso global administrator permission to use.

```
[Admin_RefreshSecurity]
```

Scheduling Events

Lasso includes a built-in scheduling facility that allows URL visits to be scheduled for a specific time in the future or to be scheduled for repeated visits. The scheduling facility loads the pages as if a client Web browser had visited the specified URL at the specified time.

Since the URLs can reference any format files available to Lasso Service, this simple scheduling facility allows powerful events to be scheduled that can perform any database actions or programming commands available to Lasso.

The scheduling facility can be used to schedule any of the following events.

- A **Routine Maintenance** page that performs database cleanup routines or optimizes the internal Lasso MySQL databases.
- A **Status Email** page that emails an administrator's address with information about Lasso's current status and what database actions have occurred since the last status email.

- A **Cache Update** page that performs a database search. The results of the search are stored and used instead of the full search in order to increase performance. The cached date is updated periodically.
- Events can load pages on **Remote Servers** in order to retrieve information, trigger an action on the remote server, or check that the remote server is active.

The event scheduling facility is intended for scheduling events which will be executed within about a minute of their intended execution time. It is not intended for high-precision execution of events. Please see the Extending Lasso Guide for information about how to create custom LDML tags that can execute with greater precision.

Event Administration

Lasso Administration allows events to be scheduled, the event queue to be stopped and started, and scheduled events to be viewed, modified, and deleted. See the Site *Administration Utilities* chapter in the Lasso Professional 8 Setup Guide for more information.

Note: The event queue can be stopped in Lasso Administration, but will always be started again if Lasso Service is relaunched.

Event Tags

Events are scheduled using the [Event_Schedule] tag which is described in *Table 3: Scheduling Tags*.

Table 3: Scheduling Tag

Tag	Description
[Event_Schedule]	Schedules a URL to be loaded by Lasso at a specified time in the future or schedules repeated loads of a specified URL.

The [Event_Schedule] tag accepts many different parameters which are described in *Table 4: Scheduling Parameters*.

Table 4: Scheduling Parameters

Tag	Description
-URL	The URL which is to be loaded when the event executes. The URL can include URL parameters. Required.

-Start	The date/time to execute the event or the time to start executing a repeating event. Optional, defaults to the current time.
-End	The date/time to stop executing a repeating event. Optional, defaults to never.
-Delay	The number of minutes to wait between executions of a repeating event. Defaults to not repeating if no -Delay is specified.
-Restart	A boolean value specifying whether an event should continue execution after a restart. Defaults to True.
-Username	An optional username which will be used to authenticate the request for the event URL. Optional.
-Password	An optional password which will be used to authenticate the request for the event URL. Optional.

The parameters of the [Event_Schedule] tag interact to allow a great variety of different behaviors. The following examples make the use of the parameters clear.

To schedule an event to execute immediately:

Use the [Event_Schedule] tag with only a -URL parameter. The event will execute within about a minute of when it is scheduled. This allows a task specified in a separate format file to be executed separately from the main flow of the current format file.

```
[Event_Schedule: -URL='http://www.example.com/event.lasso']
```

To schedule an event to happen at a specific time:

Use the [Event_Schedule] tag with a -URL parameter and a -Start parameter. The event will execute within about a minute of the -Start time.

- The following event is scheduled to execute at midnight on April 5, 2005. Since the server will likely be restarted before that date, -Restart is set to True to ensure this event is not deleted when the server is next restarted.

```
[Event_Schedule:
  -URL='http://www.example.com/event.lasso',
  -Start='4/5/2005 00:00:00',
  -Restart=True]
```

- The following event is scheduled to execute at 4:00 PM on April 5, 2005. The date/time is specified in Lasso date format. -Restart is set to True to ensure this event is not deleted when the server is next restarted.

```
[Event_Schedule:
  -URL='http://www.example.com/event.lasso',
  -Start='4/5/2005 16:00:00'
  -Restart=True]
```

- The following event is scheduled to execute four hours after the page is loaded. The date/time for the -Start parameter is generated using the [Date_Add] and [Date_GetCurrentDate] tags.

```
[Event_Schedule:
  -URL='http://www.example.com/event.lasso',
  -Start=(Date_Add: (Date_GetCurrentDate), -Hour=4)]
```

To schedule an event to repeat:

Use the [Event_Schedule] tag with a -URL parameter, and a -Delay parameter which specifies that the event should repeat. -Restart is set to False. This event will not execute after the server is restarted unless we set -Restart to True.

- The following event is scheduled to repeat every fifteen minutes starting immediately.

```
[Event_Schedule:
  -URL='http://www.example.com/event.lasso',
  -Delay=15,
  -Restart=False]
```

- The following event is scheduled to repeat every hour on 12/25/2005. Note that -Restart is set to True so this event will not be deleted when Lasso Service is next restarted.

```
[Event_Schedule:
  -URL='http://www.example.com/event.lasso',
  -Start='12/25/2005 00:00:00',
  -End='12/26/2005 00:00:00',
  -Delay=60,
  -Restart=True]
```

To schedule an event with a complex schedule:

Events which need to execute on a complex schedule must be rescheduled every time they are executed. If an event is being rescheduled explicitly then the -Delay parameter should not be specified.

For example, the following code included in the http://www.example.com/event.lasso format file will reschedule the same format file as an event depending on whether or not the condition in the [If] ... [/If] tags is True. If the condition is True then the event will be rescheduled for fifteen minutes in the future, otherwise the event will be rescheduled for two hours in the future.

```
[If: (Variable: 'Reschedule') == True]
  [Event_Schedule:
      -URL='http://www.example.com/event.lasso',
      -Start=(Date_Add: (Date_GetCurrentDate), -Minute=15),
      -Restart=False]
[Else]
  [Event_Schedule:
      -URL='http://www.example.com/event.lasso',
      -Start=(Date_Add: (Date_GetCurrentDate), -Hour=2),
      -Restart=False]
[/If]
```

Process Tags

The [Process] tag can be used to process LDML code which is contained within a variable or database field. The LDML code is processed as if it were contained in the current format file at the location of the [Process] tag. The code which is processed must be complete, all container tags must be closed within the processed code.

The [NoProcess] ... [/NoProcess] tag can be used to have Lasso ignore a portion of a page. Any LDML including square bracket or LassoScript syntax which is contained within the [NoProcess] ... [/NoProcess] tags will not be processed and will be passed through to the browser unchanged. This is most useful for segments of client-side JavaScript which contains array references using square brackets or to display a sample of LDML code on a page.

The [Sleep] tag can be used to pause the Lasso processing of the current format file for a specified number of milliseconds. This may be useful if Lasso actions need to be synchronized with the actions of other applications on the Web server or on other servers.

Table 5: Process Tags

Tag	Description
[Process]	Processes its parameter as LDML code.
[NoProcess] ... [/NoProcess]	Lasso will not process any LDML code contained within this container tag.
[Sleep]	Pauses execution of the current format file for a specified number of milliseconds.

Note: The [NoProcess] ... [/NoProcess] tags cannot be used within a LassoScript or within code processed by the [Process] tag. They must be typed exactly as specified here without any parameters or spaces within the square brackets in order to work.

To process code stored in a variable:

The following example shows how to store LDML code in a variable and then process it using the [Process] tag. The result is the same as if the code has been executed within the current format file at the location of the [Process] tag.

```
[Variable: 'LDML_Code'=(Server_Name)]
['The current server is ']
[Process: (Variable: 'LDML_Code')]
['.']
```

➜ The current server is www.example.com.

To process code stored in a database field:

The following example shows how to process code which is stored in a database variable. The result is the same as if the code has been executed within the current format file at the location of the [Process] tag. All records from the MyCode table of the Example database are found and the code from the Code field is executed.

```
[Inline: -Database='Example, -Table='MyCode', -FindAll]
  [Records]
    [Process: (Field: 'Code')]
  [/Records]
[/Inline]
```

The result will be the result of the Code field for each record.

To instruct Lasso not to process a portion of a page:

Use the [NoProcess] ... [/NoProcess] tags. In the following HTML page none of the square brackets within the JavaScript will be processed by Lasso. This allows the JavaScript to be parsed properly by the browser without any additional work by the page developer.

```
<html>
  <head>
    <title>My Lasso Page!</title>
  [NoProcess]
    <script langugage="JavaScript">
      …
    </script>
  [/NoProcess]
  </head>
  <body>
    … Lasso code here will be processed …
  </body>
</html>
```

The [NoProcess] ... [/NoProcess] tags can also be used selectively around a small portion of a page that contains square brackets, but shouldn't be processed. For example, if you are using square brackets to decorate links you can use the [NoProcess] ... [/NoProcess] tags to ensure the contents of the square brackets is not processed.

```
<a href="mylink.lasso">[NoProcess][MyLink][/NoProcess]</a>
➜ <a href="mylink.lasso">[MyLink]</a>
```

To pause execution of a format file for 15 seconds:

Use the [Sleep] tag with a parameter of 15000.

```
[Sleep: 15000]
```

Null Data Type

The null data type is the base type for all other data types in LDML. All of the tags of the null data type are available for use with values of any data type in LDML. Several of the member tags of the null data type such as [Null->Type] have already been introduced.

Table 6: Null Member Tags

Tag	Description
[Null]	Returns a null literal. This tag is usually used in comparisons or as a default value for new variables.
[Null->DetachReference]	Resets any reference value to null, detaching it from the master value and allowing it to be reassigned without affecting the master value. See the Extending Lasso Guide for information on references.
[Null->Dump]	Outputs all the variables and tags for a type. This is useful as a debugging tool.
[Null->FreezeType]	Freezes the type of a value so it cannot be modified. An error is thrown if a subsequent tag attempts to change the type of the value.
[Null->FreezeValue]	Freezes the value for a data type. An error is thrown if a subsequent tag attempts to modify the value.
[Null->IsA]	Requires a type name as a parameter. Returns true if the object is of that type or inherits from that type.
[Null->Properties]	Returns a pair containing two maps. The first element is a map of all member variables in the type. The second element is a map of all member variables in the type.
[Null->RefCount]	Returns the number of variables that reference the object.

[Null->Serialize]	Converts the value to a byte stream representation. The returned string can be stored in a database.
[Null->UnSerialize]	Accepts a single parameter which is a byte stream that represents a Lasso value. The current value is replaced by the value represented by the parameter.
[Null->Type]	Returns the data type of the value.
[Null->XMLSchemaType]	Returns the type for the root data type in the standard Lasso types schema.

To return the type of any variable:

Use the [Null->Type] tag. This tag returns a string which represents the data type of the value. If the data type is not defined then 'null' is returned. The following examples show the use of [Null->Type] on literals of different data types.

```
[123->Type] ➜ Integer
[Ouput: 123.456->Type] ➜ Decimal
['String'->Type] ➜ String
[Null->Type] ➜ Null
[(Array: 1, 2, 3)->Type] ➜ Array
```

To store a complex data type:

Use the [Null->Serialize] to transform the data type into a byte stream string representation that can be stored in a database field. Then use [Null->Unserialize] to transform the byte stream string representation back into the original data type. The following example shows how to convert an array into a string and then back again.

1 Store the array in a variable ArrayVariable.

```
[Variable: 'ArrayVariable'=(Array: 'one', 'two', 'three', 'four', 'five')]
```

2 Use the [Null->Serialize] tag to change the array into a string stored in TempVariable.

```
[Variable: 'TempVariable'=$ArrayVariable->Serialize]
```

3 The string representation of the array can now be changed back into the array by creating a new variable ArrayVariable and then calling the [Null->UnSerialize] tag with TempVariable as a parameter.

```
[Variable: 'ArrayVariable'=Null]
[$ArrayVariable->(UnSerialize: $TempVariable)]
```

4 Finally, the original array is output.

```
[Variable: 'ArrayVariable']
```

➜ (Array: (one), (two), (three), (four), (five))

Extending Lasso Note: The null data type tags are used primarily to create custom tags and custom data types. To see more examples of null data type tag usage, see the *Custom Tags* and *Custom Types* chapters.

Page Variables

Lasso stores many internal values in variables which can be accessed by an LDML programmer. Lasso also provides a tag that allows access to all of the variables defined for a page as a map. These tools can be used by LDML programmers to perform low-level tasks that would not be possible otherwise.

Table 7: Page Variable Tags

Tag	Description
[Variables]	Returns a map containing every variable defined in a page. Can be abbreviated [Vars].
[Tags]	Returns a map containing every substitution, container, and process tag registered globally in Lasso. The map does not contain custom tags defined on the current page.

To list all variables defined in the current page:

Use the [Map->Keys] tag on the map returned by [Variables] to display the name of each variable defined on the current page.

```
[Variables->Keys]
```

```
➜ (Array: (__html_reply__), (__result_code__), (__result_message__),
(__http_header__), (__tag_registry__))
```

To list all substitution, container, and process tags available in LDML:

Use the [Map->Keys] tag on the map returned by [Tags] to display the name of every tag defined globally within Lasso.

```
[Tags->Keys]
```

The output of this code is a list of close to four hundred tags registered globally in Lasso. Please see the tag list in *Appendix A: LDML 8 Tag List* for a listing of the standard substitution tags.

The list of all custom tags defined on the current page can be returned using the following code. All custom tags defined on the current page are stored in the __tags__ variable.

```
[(Variable: '__tags__')->Keys]
```

The format will be the same as that for the [Tags] map above.

Table 8: Page Variables

Variable	Description
__encoding__	The character set which will be used to encode the page for delivery to the Web client. Set by [Content_Type].
__html_reply__	The current output of the format file is stored in this variable. Changing this variable will alter the output that will be sent to the site visitor.
__http_header__	The header that will be returned by Lasso as part of the http response. The header can be manipulated using the [Header] … [/Header] tags.
__tags__	Contains a map of all custom tags defined in the current page. The global version of this variable is accessible using the [Tags] tag documented above.
__prototypes__	A map of all default member tags for each LDML 8 data type is stored in this global variable (see the Extending Language Guide for more information).

Warning: Altering any variables whose names start with an underscore is not recommended except using the methods documented in this section. These variables store internal values that are used by Lasso while processing LDML.

To alter the output of the current format file:

Set the variable __html_reply__ to the desired output. In the following example, an access denied message is displayed to the user rather than the output of the current format file.

```
<?LassoScript
  // This script changes the output of the page to an access denied message.

  $__html_reply__' = '<html>\n';
  $__html_reply__ += '\t<head><title>Access Denied</title></head>\n';
  $__html_reply__ += '\t<body><h1>Access Denied</h1></body>\n';
  $__html_reply__' += '</html>\n';
?>

➜ <html>
    <head><title>Access Denied</title></head>
    <body><h1>Access Denied</h1></body>
  </html>
```

Any LDML tags or HTML code in the format file after this LassoScript will be appended to the end of the output variable. An [Abort] tag can be used to halt execution of the page and output the contents of the variable immediately.

Configuration Tags

Lasso provides a number of tags that allow the current configuration to be examined. These tags are summarized in *Table 9: Configuration Tags*.

Table 9: Configuration Tags

Tag	Description
[Lasso_DatasourceIsFileMaker]	Accepts the name of a single database. Returns True if the database is being served through Lasso Connector for FileMaker Pro.
[Lasso_DatasourceIsLassoMySQL]	Accepts the name of a single database. Returns True if the database is being served through Lasso Connector for Lasso MySQL.
[Lasso_DatasourceIsMySQL]	Accepts the name of a single database. Returns True if the database is being served through Lasso Connector for MySQL.
[Lasso_DatasourceModuleName]	Accepts the name of a single database. Returns the name of the data source connector for the database.
[Lasso_TagExists]	Checks to see if a substitution or process tag is defined. Returns True or False. Requires one parameter which is the name of the tag to be checked.
[Lasso_TagModuleName]	Returns the name of the module in which a tag is defined. Requires one parameter which is the name of the tag to be checked.
[Lasso_Version]	Returns the version of Lasso Professional.

To check whether a tag exists:

Use the [Lasso_TagExists] tag with the tag name of the substitution or process tag to be checked. The following example will return True if the [Email_Send] tag is defined.

```
[Lasso_TagExists: 'Email_Send']
```

➜ True

To check what module a tag is defined in:

Use the [Lasso_TagModule] tag with the name of the substitution or process tag to be checked. The following example will return the module that defines the [NSLookup] tag, NSLookup.class.

```
[Lasso_TagModuleName: 'NSLookup']
```

➜ NSLookup.class

Format File Execution Time Limit

Lasso includes a limit on the length of time that a format file will be allowed to execute. This limit can help prevent errors or crashes caused by infinite loops or other common coding mistakes. If a format file runs for longer than the time limit then it is killed and a critical error is returned and logged.

The execution time limit is set to 10 minutes (600 seconds) by default and can be modified or turned off in the *Setup > Global > Settings* section of Lasso Admin. The execution time limit cannot be set below 60 seconds.

Table 10: Time Limit Tags

Tag	Description
[Lasso_ExecutionTimeLimit]	Sets the time limit for an individual format file.

The limit can be overrided on a case by case basis by including the [Lasso_ExecutionTimeLimit] tag at the top of a format file. This tag can set the time limit higher or lower for the current page allowing it to exceed the default time limit. Using [Lasso_ExecutionTimeLimit: 0] will deactivate the time limit for the current format file altogether.

On servers where the time limit should be strictly enforced, access to the [Lasso_ExecutionTimeLimit] tag can be restricted in the *Setup > Global > Tags* and *Security > Groups > Tags* sections of Lasso Admin.

Asynchronous tags and compound expressions are not affected by the execution time limit. These processes run in a separate thread from the main format file execution. If a time limit is desired in an asynchronous tag the [Lasso_ExecutionTimeLimit] tag can be used to set one.

Note: When the execution time limit is exceeded the thread that is processing the current format file will be killed. If there are any outstanding database requests or network connections open there is a potential for some memory to be leaked. The offending page should be reprogrammed to run faster or exempted from the time limit using [Lasso_ExecutionTimeLimit: 0]. Restarting Lasso Service will reclaim any lost memory.

Page Pre- and Post-Processing

Lasso has the ability to perform code before each page load on a site or after a page load. This makes it possible to modify the environment in which pages run and to perform post-processing on the output which Lasso generates.

- Include a file before every page in the site runs.
- Restrict all access to a site to a range of IP addresses.
- Log or time all page loads on a site.
- Run a redirect preprocessor to parse formatted URLs.
- Optimize the HTML code that will be delivered to the browser to increase delivery speed.

Pre-process actions can only be created by the site administrator and are usually registered at startup time by placing a file in LassoStartup. A pre-process action will run for every page on the site before any code on the page is executed.

If a pre-process action calls the [Abort] tag then no other pre-process actions or any code on the current page will execute. Post-process actions will still execute even if a pre-process action aborts.

Warning: Pre-process actions must be thorougly debugged before they are activated on a live server. A syntax error in a pre-process action can cause every page on a server to fail.

Post-process actions can only be created while a page is executing. In order to have a post-process action on every page it must be registered from a pre-process action. Post-process actions are run after all the code on the page has executed, but before it is delivered to the Web client.

Table 11: Pre and Post-Process Tags

Tag	Description
[Define_AtBegin]	Defines a pre-process action. Can only be called by the site administrator. Usually called in a startup item. Requires one parameter which is an invokable object.
[Define_AtEnd]	Defines a post-process action. Can only be called on the page where the post-process action will occur (or from a pre-process action). Requires one parameter which is an invokable object.

Both [Define_AtBegin] and [Define_AtEnd] require an invokable object as a parameter. There are three common invokable objects which are used with thise tags.

- **Compound Expression** – A compound expression can include code written in LassoScript style. The code does not need a return value and can reference page variables. The compound expression can be specified directly within the [Define_AtBegin] or [Define_AtEnd] tag. The compound expression will be called without any parameters.

  ```
  [Define_AtBegin: { ... Compound Expression ... }]
  ```

- **Tag Reference** – A built-in or custom tag can be referenced using the \ operator. The tag will be called without any parameters. Custom tags must be available globally (defined at startup or in an on-demand library) to be accessible from a pre-process action. The tag will be called without any parameters. In this example a custom tag called [Example_Cleanup] is registered as a post-process action.

  ```
  [Define_AtEnd: \Example_Cleanup]
  ```

- **Pair** – A pair can be used to pass parameters to either a compound expression or a tag reference. The first element in the pair should be the compound expression or tag reference. The second element in the pair should be an array of parameters to pass to the first element.

  The two precding examples are shown here as invokable pairs with parameters.

  ```
  [Define_AtBegin: (Pair: { ... Compound Expression ... }=(Array: ... Parameters ...))]

  [Define_AtEnd: (Pair: \Example_Cleanup=(Array: (Response_FilePath)))]
  ```

To include a file before each page in a site executes:

In LassoStartup place a .lasso LDML page with the following code. [Define_AtBegin] is used to register the tag [Include] with the parameter /preprocess.lasso. The code in this file will be executed before each page on the site is executed.

```
[Define_AtBegin: (Pair: \Include = (Array: '/preprocess.lasso'))]
```

To clean up HTML code before serving Web pages:

Use [Define_AtEnd] to call a series of regular expressions that clean up the __html_reply__ variable. This variable holds the text that Lasso is going to serve to the Web client. By modifying this variable directly the text that will be served can be changed. In the following example a series of regular expressions clean up extra white space in the HTML text.

```
<?LassoScript
Define_AtEnd: {
  $__html_reply__->trim;
  var: '__html_reply__' = (string_replaceregexp: $__html_reply__,
      -find='\r\n|\n', -replace='\r');
  var: '__html_reply__' = (string_replaceregexp: $__html_reply__,
```

```
        -find='[ \t]+', -replace=' ');
    var: '__html_reply__' = (string_replaceregexp: $__html_reply__,
        -find=' ?\r ?', -replace='\r');
    var: '__html_reply__' = (string_replaceregexp: $__html_reply__,
        -find='>\r+<', -replace='>\r<');
    var: '__html_reply__' = (string_replaceregexp: $__html_reply__,
        -find='\r\r+', -replace='\r\r');
};
?>
```

To time every page load on a site:

In LassoStartup place a .lasso LDML page with the following code. Two custom tags are defined. The first [_at_end_timer] takes a single parameter which is the time the page started executing and logs a message at the detail level. The second [_at_begin_timer] registers the [_at_end_timer] tag as a post-process action with an appropriate variable. The [Define_AtBegin] tag then registered [_at_begin_timer] as a pre-process action.

```
<?LassoScript
  Log_Detail: 'TIMING All page loads will be timed on this site.';

  Define_Tag: '_at_end_timer', -Required='start';
    Local: 'end' = _date_msec;
    Local: 'time' = (#end - #start) / 1000.0;
    Log_Detail: 'TIMING ' #time ' secs ' + response_filepath ' ' client_getargs ' '
        content_type ' ' error_code ' ' error_msg;
  /Define_Tag;

  Define_Tag: '_at_begin_timer';
    Define_AtEnd: (Pair: \_at_end_timer = (Array: _date_msec));
  /Define_Tag;

  Define_AtBegin: \_at_begin_timer;
?>
```

39

Chapter 39
Threads

This chapter documents how to share access to resources with multiple threads and how to send messages between threads..

- *Thread Tools* explains the methodology for synchronizing and sharing resources between threads.
- *Thread Communication* explains how to send messages and data between threads.

Thread Tools

Lasso is a fully multi-threaded environment. Each page is parsed and executed within its own thread, asynchronous custom tags are executed in their own threads, and background processes such as the email queue or schedule watcher are executed in their own threads.

It is important in a multi-threaded environment to synchronize access to resources such as files, global variables, or database records so that two threads do not attempt to modify the same resource at the same time. Communication between threads is discussed in the section that follows.

Consider an LDML format file which maintains a global variable recording how many times the page has been visited. At the top of the page the variable is displayed to the visitor. At the bottom of the page the variable is incremented by one. Everything will work fine as long as the page is only loaded by one visitor at a time. However, if the page is loaded by two visitors who overlap a situation can develop where the following sequence of events happens.

Thread Example

1. Visitor A loads the format file by loading the URL in their Web browser.
2. Page A starts processing with the value of the global variable, e.g. 100.
3. Visitor B loads the format file by loading the URL in their Web browser.
4. Page B starts processing with the value of the global variable, e.g. 100. This is the same value as what visitor A received.
5. Page A finishes processing and the global variable is set to a new value, e.g. 101. The new value is based on the value of the variable that was fetched at the top of the page.
6. Page B finishes loading and the global variable is set to a new value, also 101. The new value is based on the value of the variable that was fetched at the top of the page and does not take into account the fact that visitor A's page load has already modified the variable.

At the end of the process the global variable has effectively lost track of one visitor. This particular example could be fixed by reading and incrementing the variable at the top of the page, but for other resources it is necessary to restrict access so only one thread or page can have access to the resource at a time.

Table 1: Thread Tools

Type	Description
[Thread_Lock]	A simple per-thread lock which allows sequential access to a shared resource.
[Thread_Semaphore]	A counter that can be incremented or decremented to provide multiple threads access to a shared resource.
[Thread_RWLock]	A lock that allows multiple readers, but only one writer for a shared resource.

Thread Lock

A [Thread_Lock] allows multiple pages or asynchronous tags to use a shared resource sequentially. A [Thread_Lock] is usually created and stored in a global variable so all pages or tags can access it. The [Thread_Lock] has two member tags.

Table 2: [Thread_Lock] Member tags:

Tag	Description
[Thread_Lock->Lock]	Accepts an optional parameter which is the number of milliseconds to wait before timing out. Returns True if the lock was successful or False if the timeout value was reached.
[Thread_Lock->Unlock]	Unlocks a previously established lock. If there is a thread waiting for a lock then it will be allowed to continue.

To control concurrent access to a shared resource:

In the following example, a global variable Counter is used by a Web page to store the number of times that the Web page has been accessed. A [Thread_Lock] is used to ensure that only one page accesses the variable at a time. A timeout of 1000 (one second) is used to ensure that no page ends up waiting too long for access to the variable.

In a page in the LassoStartup folder the following two variables are defined. Counter is the global variable that will store the number of times the page has been loaded. Counter_Lock is the [Thread_Lock] that allows for sequential access to the variable.

```
[Variable: 'Counter' = 0]
[Variable: 'Counter_Lock' = (Thread_Lock)]
```

In the format file that visitors will load the following code attempts to lock Counter_Lock. The Counter is only modified if the attempt to get the lock is successful.

```
[If: $Counter_Lock->(Lock: 1000) == True]
  [$Counter += 1]
  [$Counter_Lock->Unlock]
[/If]
```

The timeout can be used to weigh the importance of having an accurate counter against the length of delay that a site visitor should be subjected to in a busy site. With a simple example like this the timeout will likely never be reached even on a very busy site.

Use of [Thread_Lock] is entirely voluntary and can be used to handle access to any shared resource. It is up to the site designer to create the necessary [Thread_Lock] variables and then use them when accessing shared resources.

Thread Semaphore

A [Thread_Semaphore] is a thread lock which has a counter. The [Thread_Semaphore] is initialized with a maximum number of concurrent

accesses that can occur. [Thread_Semaphore] has two member tags which are used to increment or decrement the number of current accesses.

Table 3: [Thread_Semaphore] Member Tags

Tag	Description
[Thread_Semaphore->Increment]	Requires a single parameter which is the amount to increment the semaphore. Does not return until the semaphore can be incremented by that amount. A second, optional parameter specifies the number of milliseconds to wait before timing out.
[Thread_Semaphore->Decrement]	Requires a single parameter which is the amount to decrement the semaphore.

To allow a fixed number of accesses to a shared resource:

A [Thread_Semaphore] can be used with an appropriate maximum value. For example, a page which displays site-wide statistics might take a long time to load so it is desirable to only allow five users to access the page at the same time. A semaphore can be used to block any additional users from seeing the page until one or more of the other users' page loads complete.

The following code would be placed into the LassoStartup folder in order to store the semaphore in a global variable. The semaphore is set to only allow five concurrent users.

```
[Variable: 'Page_Semaphore' = (Thread_Semaphore: 5)]
```

On the page which displays the site wide statistics the semaphore is incremented at the top of the page (with a timeout of 5 seconds) and then decremented at the bottom. If more than five users are already loading the page then the increment at the top will pause until one of the users' page finishes.

```
[If: $Page_Semaphore->(Increment: 1, 5000)]
   … Contents of the Page …
  [$Page_Semaphore->(Decrement: 1)]
[Else]
  <p>Page is busy. Try again later.
[/If]
```

Thread Read/Write Lock

A [Thread_RWLock] is a thread lock which allows an unlimited number of simultaneous reads to occur on a shared resource, but only allows one thread to write to the resource at a time. Write access will not be granted until all reads and writes have completed. Read access will not be granted as long as the write access is currently in use. [Thread_RWLock] has four

member tags which are used to establish and release read access and write access.

Table 4: [Thread_RWLock] Member Tags

Tag	Description
[Thread_RWLock->ReadLock]	Establishes a read lock. If a write lock is currently in place then the tag will pause until the write lock is released. Since read locks are not exclusive it will not pause if additional read locks have already been granted.
[Thread_RWLock->ReadUnlock]	Releases a read lock.
[Thread_RWLock->WriteLock]	Establishes a write lock. If one or more read locks or a write lock is in place then the tag will pause until the locks are released.
[Thread_RWLock->WriteUnlock]	Releases a write lock.

To control write access to a resource while allowing multiple reads:

Most resources can be accessed by multiple threads which only need to read from the resource, but require that only one client write to the resource at the same time. In the following example a global variable contains a set of server-wide preferences that can be read by many pages at the same time, but must only be modified by one page at a time.

In a page in the LassoStartup folder the following two variables are defined. Preferences is the global variable that will store server-wide preferences such as the administrator's email address and a count of how many page loads there have been. Preferences_Lock is the [Thread_RWLock] that controls access to the variable.

```
[Variable: 'Preferences' = (Map: 'Email' = 'administrator@example.com' )]
[Variable: 'Preferences_Lock' = (Thread_Lock)]
```

In each format file in the site a read lock is established on the preferences. As many format files as are needed can concurrently read the preferences.

```
[$Preferences_Lock->(ReadLock)]
  … Contents of the Page …
[$Preferences_Lock->(ReadUnlock)]
```

In a page which modifies the preferences a write lock needs to be established. The following code first releases the read lock, then establishes a write lock, modifies the global Preferences variable, and finally releases the write lock and re-establishes the read lock for the remainder of the page.

```
[$Preferences_Lock->(ReadUnlock)]
  [$Preferences_Lock->(WriteLock)]
    [$Preferences->(Insert: 'Email' = (Action_Param: 'Email'))]
  [$Preferences_Lock->(WriteUnlock)]
[$Preferences_Lock->(ReadLock)]
```

The [Thread_RWLock] tags do not have a timeout value like the [Thread_Lock] page. In the example above the page which is loaded by the visitor who wants to change the preferences will simply idle until each of the pages which have established a read lock are finished loading.

Thread Communication

The previous section documented methods for sharing data between threads using global variables. Often it is desirable to not just share data, but to push data from thread to thread. This section documents techniques for sending signals and data between threads.

Table 5: Thread Communication

Type	Description
[Thread_Event]	A simple signalling method which allows threads to idle until they receive a signal to continue.
[Thread_Pipe]	Allows variables and data objects to be sent from thread to thread. The foundation of more complex messaging systems.

Thread Events

Thread events are simple signals that are either in an active or inactive state. One or more threads can wait for a signal to occur. A triggering thread can cause one or all of the threads waiting for the signal to continue processing. No data can be passed using thread events.

Table 6: [Thread_Event] Member Tags:

Tag	Description
[Thread_Event->Wait]	Accepts an optional timeout value in milliseconds. If a signal is received before the timeout then True is returned otherwise False is returned. With no timeout value the tag will pause forever.
[Thread_Event->Signal]	Allows one thread which is waiting for this signal to continue.
[Thread_Event->SignalAll]	Allows all threads which are waiting for this signal to continue.

To create an asynchronous tag that waits for a signal:

Create a [Thread_Event] signal in a page variable. Within a custom asynchronous tag, wait until the signal is triggered before continuing.

In the following example a custom tag waits for one second for the page to reach the end. It then logs whether the page completed in one second or not to the console using the [Log_Warning] tag. At the top of the page the custom tag and the signal are defined. The custom tag is called to start the one second timer. At the bottom of the page the signal is triggered.

```
[Variable: 'mySignal' = (Thread_Event)]
[Define_Tag: 'OneSecond']
  [If: $mySignal->(Wait: 1000) == True]
    [Log_Warning: 'The page took less than 1 second to load.']
  [Else]
    [Log_Warning: 'The page took more than 1 second to load.']
  [/If]
[/Define_Tag]
[OneSecond]

 ... Page Contents ...

[$mySignal->Signal]
```

Each time the page loads one of the messages will be logged to the console depending on how long the page took to process.

Thread Pipes

Thread pipes allow data to be passed from thread to thread. The [Thread_Pipe] type has two member tags.

Table 7: [Thread_Pipe] Member Tags:

Tag	Description
[Thread_Pipe->Set]	Accepts a value which will be placed into the pipe. A subsequent (or waiting) call to [Thread_Pipe->Get] will retrieve the value.
[Thread_Pipe->Get]	Accepts an optional parameter which is a timeout value in milliseconds. If an object is waiting in the pipe or is placed in the pipe before the timeout value then it is returned. Otherwise null is returned when the timeout value is reached. If the timeout value is omitted then the tag will wait forever for an object to arrive.

To create an asynchronous tag that will process messages:

Create a [Thread_Pipe] in a global variable and an asynchronous tag in the LassoStartup folder. The asynchronous tag will idle until an event is placed into the pipe. For this example, the tag will log each received event to the console, but more complex processing is possible.

In a page in the LassoStartup folder the [Thread_Pipe] and custom tag [Ex_Watcher] are defined. The custom tag has a [While: True] loop that ensures that it loops forever since the condition will always be true. The [Thread_Pipe->Get] tag has a timeout value of 10000 (10 seconds) so it can send a Still Waiting... message to the console. If a message of Abort is received then the tag aborts without doing any further processing.

```
[Variable: 'myPipe'= (Thread_Pipe)]
[Define_Tag: 'Ex_Watcher']
  [While: True]
    [Local: 'Message' = $myPipe->(Get: 10000)]
    [Select: #Message]
      [Case: 'Abort']
        [Return]
      [Case: Null]
        [Log_Warning: 'Message: Still Waiting…']
      [Case]
        [Log_Warning: 'Message: ' + #Message]
    [/Select]
  [/While]
[/Define_Tag]
[Ex_Watcher]
```

In a format file a message can be sent to the watcher by placing it into the pipe using the [Thread_Pipe->Set] tag. For example, the following code places a message saying I Got It! into the pipe. The message will appear in the console immediately, but no results will be returned to the page.

```
[$myPipe->(Set: 'I Got It!')]
```

40

Chapter 40
Tags and Compound Expressions

This chapter documents advanced programming techniques.

- *Tag Data Type* introduces the tag data type and its member tags.
- *Compound Expressions* shows how tags can be created using simple expression notation.
- *LDML Parsing* introduces the LDML type which parses LDML files allowing them to be inspected programmatically or displayed using syntax coloring within a Web browser.

Tag Data Type

Tags are represented by Lasso as objects which belong to a data type. Just like arrays or maps, tag objects have member tags which allow them to be manipulated. Tags can be stored in variables or in complex data types such as maps or arrays.

Since calling a tag, e.g. [Action_Params], returns the value that results when the tag is run rather than a reference to the tag, special steps must be taken to get a reference to the tag itself. Tag objects can be found in four ways.

- **\ Symbol** – The \ symbol can be used to find a tag object. For example, \Field will return a reference to the [Field] tag and \Action_Params will return a reference to the [Action_Params] tag. The following code stores a reference to [Action_Params] in a variable.

```
[Variable: 'myActionParamsTag' = \Action_Params]
```

- **Tags Map** – Lasso maintains a global tag map which can be retrieved using the [Tags] tag. An individual tag can be referenced using the [Map->Find] tag. For example, the following code stores a reference to [Action_Params] in a variable.

  ```
  [Variable: 'myActionParamsTag' = Tags->(Find: 'Action_Params')]
  ```

  [Tags] returns a reference to the global variable __tags__ which contains all substitution, container, and process tags defined in Lasso. Custom tags defined on the current page can be found in the page variable __tags__.

- **Data Type Properties** – Each instance of a data type maintains a list of properties for that instance which can be retrieved using the [Null->Properties] tag. These include both instance variables and member tags. For example, the following code stores a reference to the [Array->Get] tag in a variable.

  ```
  [Variable: 'myGetTag' = Array->Properties->Second->(Find: 'Get')]
  ```

  The member tags of the built-in data types can also be found in the global variable __prototypes__.

- **Compound Expressions** – These are discussed in the next section. Compound expressions allow tags to be created on the fly. For example, the following code stores a compound expression that returns the number 5 in a variable.

  ```
  [Variable: 'myTag' = { Return: 5; }]
  ```

Table 1: Tag Data Type Member Tags

Tag	Description
[Tag->Invoke]	Executes the tag as if it had been called normally. The parameters passed to this tag are passed as the parameters for the tag.
[Tag->Run]	Executes the tag as if it had been called normally. The parameters to this tag are discussed in the table that follows.
[Tag->Eval]	Evaluates a tag or compound expression in the current context. No parameters can be passed to the tag or compound expression.
[Tag->asType]	Executes the tag as a type initializer. Accepts the same parameters as [Tag->Run]
[Tag->asAsync]	Executes the tag in a new thread. Accepts the same parameters as [Tag->Run].
[Tag->Description]	Returns the description of the tag if defined.
[Tag->ParamInfo]	Returns an array of information about the parameters which the tag requires. Each element of the array has members ParamName, ParamType, and IsRequired.
[Tag->ReturnType]	Returns the type of value the tag will return.

The [Tag->Run] tag is most commonly used with built-in LDML tags and with custom tags. This tag accepts the parameters outlined in the following table.

Table 2: [Tag->Run] Parameters

Parameter	Description
-Params	An array of parameters to pass to the tag. Can be omitted if the tag does not require any parameters.
-Owner	Identifies the variable which contains the data type that should be operated on, i.e. the object that would be specified to the left of the -> symbol. Required for member tags.
-Name	Name of the tag. Many built-in LDML tags such as [Math_…], [String_…], [Server_…], etc. behave differently depending on what tag name they are called with. The -Name parameter is required for these tags to operate properly.

To run a tag:

Use the [Tag->Run] tag on a stored reference to the tag which is to be run. The following examples each retrieve a tag from the [Tags] or [Null->Properties] map and then run it using appropriate parameters.

- The [Action_Params] tag can be called as follows. First a reference to the tag is stored in a variable, then [Tag->Run] is called on the stored reference. It is always best to specify the -Name parameter explicitly since it is required by many built-in tags.

```
[Variable: 'myActionParamsTag' = Tags->(Find: 'Action_Params')]
[$myActionParamsTag->(Run: -Name='Action_Params')]
```

```
➜ (Array: (Pair: (-Nothing)=()), (Pair: (-OperatorLogical)=(and)),
(Pair: (-MaxRecords)=(50)), (Pair: (-SkipRecords)=(0)))
```

- The [Array->Get] tag can be called by retrieving the tag from the [Array->Properties] map and then calling it using [Tag->Run] with the array that is to be acted upon referenced in the -Owner parameter.

```
[Variable: 'myArray' = (Array: 'Alpha', 'Beta', 'Gamma')]
[Variable: 'myGetTag' = Array->Properties->Second->(Find: 'Get')]
[$myGetTag->(Run: -Params=2, -Owner=$myArray, -Name='Get')]
```

```
➜ Beta
```

The previous examples demonstrate how to use the tag member tags to execute tags, but each example is easy enough to write using simple LDML. The following example demonstrates how tag references can be used to create a new type of custom tag that can operate on each element of an array.

To run a tag on each element of an array:

Create a custom tag which accepts an array and a reference to a tag as parameters. The referenced tag will be used on each element of the array in turn. The custom tag [Ex_VisitArray] is defined as follows.

```
[Define_Tag: 'Ex_VisitArray', -Required='myArray', -Required='myTag']
  [Iterate: #myArray, (Local: 'myItem')]
    [#myItem= #myTag->(Run: -Params=(Array: #myItem))]
  [/Iterate]
[/Define_Tag]
```

This tag can now be used to apply a tag to each element of an array. For example, it could be used to replace each element of an array by the value of a variable of the same name. An array and three variables are created and the [Variable] tag is found in the [Tags] map.

```
[Variable: 'theArray' = (Array: 'Alpha', 'Beta', 'Gamma')]

[Variable: 'Alpha' = 100, 'Beta' = 1234, 'Gamma' = 987]
```

```
[Ex_VisitArray: $theArray, Tags->(Find: 'Variable')]

[Variable: 'theArray']

➜ (Array: 100, 1234, 987)
```

Combined with the use of compound expressions which are described in the next section this can be a very powerful technique for batch processing of data which is stored in an array.

To get information about a tag:

The [Tag->Description], [Tag->ParamInfo], and [Tag->ReturnType] tags can be used to get information about a tag. For properly defined tags this information can prove invaluable in determining how to use an unknown tag.

The following example shows the definition of a tag [myTag] and then the information that can be retrieved about it.

```
[Define_Tag: 'Ex_Repeat',
    -Required='String', -Type='string',
    -Optional='Repeat', -Type='integer',
    -ReturnType='String',
    -Description='[Ex_Repeat: String, Integer] => String']
  [Return: #String * (Integer: (Local: 'Repeat'))]
[/Define_Tag]

<br>Description: [Encode_HTML: \Ex_Repeat->Description]
<br>Returns: [Encode_HTML: \Ex_Repeat->ReturnType]
<br>Params [Iterate: \Ex_Repeat->ParamInfo, (Var: 'param')]
  <br>[Loop_Count]: [Encode_HTML: $param->ParamName]
  [If: $param->ParamType == 'null'] (Any) [Else] ([Encode_HTML: $param-
>ParamType]) [/If]
  [If: $param->IsRequired]Required[/If]
[/Iterate]

➜ Description: [Ex_Repeat: String, Integer] => String
Returns: String
Params:
1: String (string) Required
2: Repeat (integer)
```

Compound Expressions

Compound expressions allow for tags to be created within LDML code and executed immediately. Compound expressions can be used to process brief snippets of LDML code inline within another tag's parameters or can be used to create reusable code blocks.

Evaluating Compound Expressions

A compound expression is defined within curly braces { }. The syntax within the curly braces should match that for LassoScripts using semi-colons between each LDML tag. For example, a simple compound expression that adds 6 to a variable myVariable would be written as follows. The expression can reference page variables.

```
[Variable: 'myExpression' = { $myVariable += 6; }]
```

The compound expression will not run until it is asked to execute using the [Tag->Eval] tag. The expression defined above can be executed as follows.

```
[Variable: 'myVariable' = 5]
[$myExpression->Eval]
[Variable: 'myVariable']
```

➜ 11

A compound expression returns values using the [Return] tag just like a custom tag. A variation of the expression above that simply returns the result of adding 6 to the variable, without modifying the original variable could be written as follows.

```
[Variable: 'myExpression' = { Return: ($myVariable + 6); }]
```

This expression can then be called using the [Tag->Eval] tag and the result of that tag will be the result of the stored calculation.

```
[Variable: 'myVariable' = 5]
[$myExpression->Eval]
```

➜ 11

Alternately, the expression can be defined and called immediately. For example, the following expression checks the value of a variable myTest and returns Yes if it is True or No if it is False. Since the expression is created and called immediately using the [Tag->Eval] tag it cannot be called again.

```
[Variable: 'myTest'= True]
[Encode_HTML: { If: $myTest; Return: 'Yes'; Else; Return: 'No'; /If; }->Eval]
```

➜ Yes

Running Compound Expressions

The same conventions for custom tags may be used within a compound expression provided it is executed using the [Tag->Run] tag. Compound expressions which are run can access the [Params] array and define local variables.

For example, the following expression accepts a single parameter and returns the value of that parameter multiplied by itself. The expression is formatted similar to a LassoScript using indentation to make the flow of logic clear.

```
[Variable: 'myExpression' = {
  Local: 'myValue' = (Params->(Get: 1));
  Return: #myValue * #myValue;
}]
```

This expression can be used as a tag by calling it with the [Tag->Run] tag with an appropriate parameter. The following example calls the stored tag with a parameter of 5.

```
[Encode_HTML: $myExpression->(Run: -Params=(Array: 5))]
```

➜ 25

When combined with the [Ex_VisitArray] tag that was defined in the previous section, a compound expression can be used to modify every element of an array in place. In the following example, the compound expression above is used to square every element of an array.

```
[Variable: 'myArray' = (Array: 1, 2, 3)]
[Ex_VisitArray: $myArray, $myExpression]
```

➜ (Array: (1), (4), (9))

LDML Parsing

Lasso includes an LDML data type that accepts an LDML file or string containing an LDML snippet as a parameter. The LDML data type parses the LDML code and its member tags can be used to inspect the parsed form of the LDML code.

This data type makes it possible to programmatically inspect LDML files, checking what tags are used, whether tags are properly opened and closed, simple syntax checking, and more. It also makes it possible to display LDML code to site visitors in a Web browser using syntax coloring.

Table 3: LDML Type Tag

Tag	Description
[LDML]	The LDML type requires a string containg LDML code or a [File] object representing an LDML file. By default only the LDML code in a file is considered. Optional parameters allow -Delimiters and -Plaintext to be included in the parsed representation of the file as well.

The [LDML] type accepts either a string containg some LDML source code or a [File] object referencing an LDML file as a parameter. It parses all of the LDML code and generates a list of tokens representing each delimiter, keyword, tag name, parameter, symbol, etc. within the LDML source.

To parse an LDML file:

In this example the following LDML code is stored in a file named Info.Lasso.

```
<?LassoScript
  '<br>Address: ' + Client_Address;
?>
```

The file is parsed and the individual tokens within are output on individual lines using this code. The [LDML] tag is called using [Include_Raw] to read in the raw source of the Info.Lasso file. The -Delimiters and -Plaintext keywords are specified to include tag starts and ends and HTML text within the parsed output. [Iterate] ... [/Iterate] is used to cycle through all the tokens in the LDML source and output each one.

```
<?LassoScript
  Var: 'Tokens' = LDML(Include_Raw('test.lasso'), -Delimiters, -Plaintext);
  '<br>Token Count: ' + $Tokens->Size;
  '<br>Char Length: ' + $Tokens->Length;
  Iterate($Tokens, Var('Token'));
    '<br>' + Loop_Count + ': ' + Encode_HTML($Token);
  /Iterate;
?>
```

When the code above is run the output is as follows. The file contains five tokens and is 51 characters long. Each of the five tokens is output individually.

```
➜ Token Count: 5
  Char Length: 51
  1: <?LassoScript
  2: <br>Address:
  3: +
  4: Client_Address
  5: ?>
```

The member tags of the LDML data type can be used to get more information about the individual tokens within the LDML source code. The following table lists the available member tags and then examples of how to use those member tags follow.

Table 4: LDML Type Member Tags

Tag	Description
[LDML->HasMore]	Returns True if the LDML source has more tokens. Used with [LDML->Next] to advanced through the tokens.
[LDML->Next]	Advances the current token and returns the text of the now-current token.
[LDML->Position]	Returns the position of the current token as determined by [LDML->Next].
[LDML->TokenType]	Returns the type of the current token or if given an integer position the type of that token. See the following table for a list of possible types.
[LDML->Offset]	Returns the character offset of the current token or if given an integer position the character offset of that token.
[LDML->TokenLength]	Returns the character length of the current token or if given an integer position the character length of that token.
[LDML->Length]	Returns the length in characters of the LDML source.
[LDML->Size]	Returns the number of tokens in the LDML source.
[LDML->Get]	Requires a single integer position. Gets the specified token from the LDML source.

The member tags of the LDML type are split into three groups.

- The [LDML->Length] tag simply returns the length of the LDML source code in characters. You can also output the LDML source code by casting the LDML object to string as in [String: $Tokens].
- The [LDML->Get] and [LDML->Size] tags are implemented for [Iterate] compatibility. These tags can be used to quickly inspect the tokens within the LDMl source code.
- More information about each individual token can be returned using the [LDML->HasMore] tag within a [While] ... [/While] condition and the [LDML->Next] tag to advance through the tokens. The following code will display much the same output as the [Iterate] example above.

```
<?LassoScript
  Var: 'Tokens' = LDML(Include_Raw('test.lasso'), -Delimiters, -Plaintext);
  While($Tokens->HasMore);
    '<br>' + Loop_Count + ': ' + Encode_HTML($Tokens->Next);
  /While;
?>
```

```
➜ 1: <?LassoScript
  2: <br>Address:
  3: +
  4: Client_Address
  5: ?>
```

Additional member tags provide information about each token. The [LDML->Position] tag returns the position of the current token. The [LDML->TokenType] tag returns the type of the current token (see the following table for a full list of possible t ypes). The [LDML->Offset] tag returns the character offset within the LDML source of the current token and the [LDML->TokenLength] tag returns the length in characters of the current token.

A more complete example than that above will output significantly more information about each token within the LDML source. The type of each token and the character offset and length of each token are output.

```
<?LassoScript
  Var: 'Tokens' = LDML(Include_Raw('test.lasso'), -Delimiters, -Plaintext);
  While($Tokens->HasMore);
    '<br>' + Loop_Count + ': ' + Encode_HTML($Tokens->Next);
    ' - ' + $Tokens->TokenType;
    ' (' + $Tokens->Offset + ', ' + $Tokens->TokenLength + ')';
  /While;
?>
```

```
➜ 1: <?LassoScript - TAG_START (0,15)
  2: <br>Address: - LITERAL (16,13)
  3: + - OPERATOR (31,1)
  4: Client_Address - KEYWORD (33,14)
  5: ?> - TAG_END (49,2)
```

An example of using the token type to perform syntax coloring is included after the table of token types.

- The [LDML->TokenType], [LDML->Offset], and [LDML->TokenLength] tags can also be passed an integer position parameter to return informtion about tokens in arbitrary order. The output of the following LassoScript is identical to the LassoScript above.

```
<?LassoScript
  Var: 'Tokens' = LDML(Include_Raw('test.lasso'), -Delimiters, -Plaintext);
  Loop($Tokens->Size);
    '<br>' + Loop_Count + ': ' + Encode_HTML($Tokens->Get(Loop_Count));
    ' - ' + $Tokens->TokenType(Loop_Count);
    ' (' + $Tokens->Offset(Loop_Count) + ', ';
    $Tokens->TokenLength(Loop_Count) + ')';
  /Loop;
?>
```

```
➜ 1: <?LassoScript - TAG_START (0,15)
  2: <br>Address: - LITERAL (16,13)
  3: + - OPERATOR (31,1)
  4: Client_Address - KEYWORD (33,14)
  5: ?> - TAG_END (49,2)
```

Token Types

The types reported by the [LDML->TokenType] tag do not map precisely to the terminology used in the Lasso documentation. Check the types in the following table carefully to be sure that each token is being used in the expected fashion.

Table 5: LDML Token Types

Tag	Description
TAG_START	An opening square bracket [or <?LassoScript or a semi-colon within a LassoScript. Only included if the -Delimiters keywword is specified in the [LDML] tag.
TAG_END	A closing square bracket] or ?>. Only included if the -Delimiters keywword is specified in the [LDML] tag.
KEYWORD	An unquoted literal such as a tag or type name, variable name, constant, etc.
PARAMETER	A keyword parameter which starts with a hyphen and does not have a value.
PARAMS_START	A colon : which divides a tag name from a parameter list in colon syntax.
VALUE_START	An equal sign = which divides a name/keyword from a value in a single parameter.
PARAM_END	A comma , which ends a single parameter within a parameter list.
OPERATOR	A mathematical, logical, member, or comparison symbol: + - * / % += -= *= /= %= ++ -- = := == != === !== > >= < <= && || -> & # \ @.

GROUP_START	An opening parenthsis (. Used in expressions and for parameter lists in comma syntax.
GROUP_END	A closing parenthesis). Used in expressions and for parameter lists in comma syntax.
LITERAL	A literal string value.
NUMBER	A literal integer or decimal value.
PLAINTEXT	Any HTML or other text not included within a Lasso tag or LassoScript. Only included if the -PlainText keywword is specified in the [LDML] tag.
BLOCK_START	An opening brace { for a compound expression.
BLOCK_END	A closing brace } for a compound expression.

To syntax color LDML code:

The token types can be used to provide syntax coloring for LDML code. The following example shows a quick method simply by outputing each token type using a different color.

```
<?LassoScript
  Var: 'Tokens' = LDML(Include_Raw('test.lasso'), -Delimiters, -Plaintext);
  While($Tokens->HasMore && (loop_count < 10));
    Var('Token' = $Tokens->Next);
    Select($Tokens->TokenType);
      Case('TAG_START');
        '<br><span style="color: red">' + Encode_HTML($Token) + '</span>';
      Case('TAG_END');
        '<br><span style="color: red">' + Encode_HTML($Token) + '</span>';
      Case('OPERATOR');
        '<br><span style="color: red">' + Encode_HTML($Token) + '</span>';
      Case('KEYWORD');
        '<br><span style="color: blue">' + Encode_HTML($Token) + '</span>';
      Case('LITERAL');
        '<br><span style="color: green">' + Encode_HTML($Token) + '</span>';
      Case;
        '<br><span style="color: black">' + Encode_HTML($Token) + '</span>';
    /Select;
  /While;
?>
```

```
➜ <?LassoScript
  <br>Address:
  +
  Client_Address
  ?>
```

41

Chapter 41
Miscellaneous Tags

This chapter documents several tags which do not logically fit into any other chapter in the Lasso 8 Language Guide.

- *Name Server Lookup* documents the [NSLookup] tag.
- *Validation Tags* describes tags which validate credit card numbers, email addresses, and URLs.
- *Unique ID Tags* describes the [Lasso_UniqueID] tag.

Name Server Lookup

The [NSLookup] tag is implemented using LJAPI (Lasso Java API). In order to use the [NSLookup] tag, Java must be configured properly. Please see the *Setting Site Preferences* chapter and the configuration chapters of the Lasso Professional 8 Setup Guide for more information.

Table 1: Name Server Lookup Tag

Tag	Description
[NSLookup]	Requires a single parameter. Returns the IP address if the parameter is a host name or the host name if the parameter is an IP address.

To find the IP address of a specific host name:

Use the [NSLookup] tag with the host name as its parameter. The following example returns the IP address for www.example.com.

```
[NSLookup: 'www.example.com'] ➜ 127.0.0.1
```

To find the host name for a specific IP address:

Use the [NSLookup] tag with the IP address as its parameter. The following example returns the host name for the IP address 127.0.0.1.

```
[NSLookup: '127.0.0.1'] ➜ www.example.com
```

Validation Tags

LDML provides a set of tags which can be used to validate various text formats. These tags are summarized in *Table 2: Valid Tags*.

Table 2: Valid Tags

Tag	Description
[Valid_CreditCard]	Accepts a single string parameter containing a credit card number. Returns True if the credit card number is valid according to the ROT-13 algorithm.
[Valid_Date]	Accepts a single string parameter containing a date. Returns True if the date is in a format that Lasso can parse or False otherwise.
[Valid_Email]	Accepts a single string parameter containing an email address. Returns True if the email address appears to be in a valid format.
[Valid_URL]	Accepts a single string parameter containing a URL. Returns True if the URL appears to be in a valid format.

Note: See the *String Operations* chapter for information about the [String_Is...] tags that can be used to determine what type of data strings contain.

To check whether a credit card number is valid:

The [Valid_CreditCard] tag provides a quick check for the basic validity of a credit card number, but can only ensure that a card is of the right format, not that an account is active or has available credit. The following code checks the fake credit card number 8888 8888 8888 8888 and predictably returns False.

```
[Valid_CreditCard: '8888888888888888'] ➜ False
```

Unique ID Tags

The [Lasso_UniqueID] tag can be used to create a simple unique ID. The ID created by the [Lasso_UniqueID] tag has a very high probability of being unique since it is based on the current date and time, the IP address of the current visitor, and a random component.

Unique IDs are usually used to identify a particular record in a database. When a new record is added, one field is set to the value from [Lasso_UniqueID] and that same value is stored in a variable. When the record needs to be retrieved from the database, [Lasso_UniqueID] can be used again.

Table 3: Unique ID Tag

Tag	Description
[Lasso_UniqueID]	Returns a unique ID.

Server Tags

The following tags provide useful information when logging to the console. The type of output can be selected by specifying an optional parameter.

Table 4: Server Tags

Tag	Description
[Server_Date]	Returns the current date. Accepts a parameter -Short, -ShortY2K, -Abbrev, or -Long.
[Server_Day]	Returns the current weekday. Accepts a parameter -Short or -Long.
[Server_Time]	Returns the current time. Accepts a parameter -Short, -Long, -Extended.

VII

Section VII
Protocols

This section includes chapters about various Internet protocols and how they can be accessed through Lasso.

- *Chapter 42: Sending Email* discusses how to send email from Lasso (including HTML email and attachments) and how to access SMTP server directly.
- *Chapter 43: POP* discusses how to download email from POP servers and parse MIME messages.
- *Chapter 44: HTTP/HTML Content and Controls* discusses Lasso's tags for including content from remote Web servers, using FTP, setting and retriving cookie values, caching data, modifying the HTTP header, parsing the incoming request header, and more.
- *Chapter 45: XML-RPC* discusses how XML-RPC procedures can be called from within Lasso or served by Lasso.
- *Chapter 46: SOAP* discusses how SOAP procedures can be called from within Lasso or served by Lasso.
- *Chapter 47: Wireless Devices* discusses how WML files can be created and served to wireless browsers such as cell phones or PDAs.

42

Chapter 42
Sending Email

This chapter describes how to send email using Lasso.

- *Overview* introduces the SMTP email sending system.
- *Sending Email* describes the [Email_Send] tag.
- *Composing Email* describes the [Email_Compose] tag which can be used to create more complex emails with multiple attachments and parts.
- *SMTP Type* describes the SMTP type that is used to actually send messages to remote servers.

Note: The following Chapter 43: *POP* documents how to download email from a POP server using Lasso.

Overview

Lasso includes a built-in system for queuing and sending email messages to SMTP servers. Email messages can be sent to site visitors to notify them when they create a new account or to remind them of their login information. Email messages can be sent to administrators when various errors or other conditions occur. Email messages can even be sent in bulk to many email addresses to notify site visitors of updates to the Web site or other news.

Email messages are queued using the [Email_Send] tag. All outgoing messages are stored in the SMTP_Queue table of the Site database. The queue can be examined and started or stopped in the *Utility > Email* section of Site Administration.

Lasso's email system checks the queue periodically and sends any messages which are waiting. If the email system encounters an error when sending an email then it stores the error in the database and requeues the message. If too many errors are encountered then the message send will be cancelled.

By default, Lasso sends every queued message to the SMTP server that is specified in the *Utility > Email > Setup* section for Site Administration. It is recommended that an appropriate SMTP server be set up for each site. An optional SMTP AUTH username and password allows Lasso to authenticate with the SMTP server in order to send messages. Lasso will use CRAM-MD5 authentication if the SMTP server supports it.

Alternately, a specific SMTP host can be specified within the [Email_Send] tag. This can be useful if different SMTP servers need to be used for different purposes. For example, if one SMTP server needs to be used for internal company email and another for general Internet users.

Note: Lasso must either have valid SMTP AUTH credentials or be otherwise allowed to send unrestricted messages through each SMTP server that it accesses. Consult the SMTP server documentation for details about how to setup SMTP AUTH security or how to allow specific IP addresses to relay messages.

By default Lasso will send up to 100 messages to each SMTP server every connection. Lasso will open up to 5 outgoing SMTP connections at a time. Lasso selects messages to send in priority order, but once it connects to a service it delivers as many messages as possible. This means that a batch send to an SMTP server will contain high priority messages and enough medium and low priority messages to round out the 100.

Important: The maximum size of an email message including all attachments must be less than 8MB using the [Email_Send] tag. If necessary, larger messages can be sent using the [Email_Immediate] tag described in the **Email Composing** section.

The email system is administered using the *Utility > Email* section of Site Administration. The *Email Queue* can be inspected and any errors which have occurred can be reviewed. Email messages can be queued manually using the *Send Email* page. The preferences for the email system, such as how often the queue is checked for messages or how many times messages are requeued if an error is detected, can be modified using the *Setup* page.

Note: Lasso's email system is written in LDML using the network tags. All of the source code for the email system is included open source within the code for Startup.LassoApp.

Sending Email

The [Email_Send] tag is used to send most email messages from Lasso. This tag supports the most common types of email including plaintext, HTML, plaintext/HTML alternatives, and attachments.

Table 1: Email Tag

Tag	Description
[Email_Send]	Queues an email message.

The [Email_Send] tag accepts many parameters. The basic parameters are shown in the table below. Additional parameters for modifying the sent email, adding attachments, controlling character set, etc. are included in a subsequent tables.

This section assumes that a valid SMTP host has been defined in the *Utility > Email > Setup* section of Site Administration. The *alternate SMTP Servers* section that follows shows how to specify the SMTP host directly within the [Email_Send] tag.

Table 2: [Email_Send] Parameters

Keyword	Description
-To	The recipient of the message. Multiple recipients can be specified by separating their email addresses with commas. Required.
-From	The sender of the message. Required.
-Subject	The subject of the message. Required.
-Body	The body of the message. Either a -Body or -HTML part (or both) is required.
-HTML	The HTML part of the message. Either a -Body or -HTML part (or both) is required.
-CC	Carbon copy recipients of the message. At least one of -To, -CC, or -BCC is required.
-BCC	Blind carbon copy recipients of the message. At least one of -To, -CC, or -BCC is required.

Note: The -Email.... command tags from LDML 3 will not operate in LDML 8.

To send an email message:

Use the [Email_Send] tag with the desired parameters. The -From parameter must be set to a valid email address. The -Subject parameter must be set to the desired subject for the email message. One or more recipients must be specified using the -To, -CC, and -BCC parameters. The body of the message can be specified using any of the three methods described here.

- An email can be sent with a hard-coded body by specifying the message directly within the [Email_Send] tag. The following example shows an email sent to example@example.com with a hard-coded message body.

```
[Email_Send:
  -To='example@example.com',
  -From='example@example.com',
  -Subject='An Email',
  -Body='This is the body of the email.']
```

- The body of an email message can be assembled in a variable in the current format file and then sent using the [Email_Send] tag. The following example shows a variable Email_Body which has several items added to it before the message is finally sent.

```
<?LassoScript
  Variable: 'Email_Body' = 'This is the body of the email';
  $Email_Body += '\nSent on: ' + (Server_Date) + ' at ' + (Server_Time);
  $Email_Body += '\nCurrent visitor: ' + (Client_Username) + ' at ' + (Client_IP);

  Email_Send:
    -To='example@example.com',
    -From='example@example.com',
    -Subject='An Email',
    -Body=$Email_Body;
?>
```

- A format file on the Web server can be used as the message body for an email message using the [Include] tag. A format file created to be a message body should contain no extra white space. The following example shows a format file format.lasso which is contained in the same folder as the current format file being used as the message body for an email. Any LDML tags within format.lasso will be executed before the email is sent.

```
[Email_Send:
  -To='example@example.com',
  -From='example@example.com',
  -Subject='An Email',
  -Body=(Include: 'format.lasso')]
```

To send HTML email:

HTML email can be easily sent using the [Email_Send] tag using any of the following methods.

- An HTML format file can be sent as the body of the message by using [Include] as the parameter to the -HTML parameter of [Email_Send]. Any image references or URLs in the HTML format file should be specified fully qualified including the http:// prefix and server name.

```
[Email_Send:
  -To='example@example.com',
  -From='example@example.com',
  -Subject='An Email',
  -HTML=(Include: 'email_body.html')]
```

  Note that any LDML tags inside the format file will be processed by Lasso before it is served.

- A plaintext/HTML alternative email can be sent by specifying both a -Body parameter and an -HTML parameter. The text of both parts should be equivalent. Recipients with text-based email clients will see the text part while recipients with HTML-based email clients will see the HTML part.

```
[Email_Send:
  -To='example@example.com',
  -From='example@example.com',
  -Subject='An Email',
  -Body=(Include: 'format.lasso'),
  -HTML=(include: 'email_body.html')]
```

To send an email message to multiple recipients:

Email can be sent to multiple recipients by including their addresses as a comma delimited list in the -To parameter, the -CC parameter, or the -BCC parameter. Multiple -To, -CC, or -BCC parameters are not allowed.

- The following example shows an [Email_Send] tag with two recipients in the -To parameter. The recipients email addresses are specified with a comma between them example@example.com, somone@example.com. No extraneous information such as the recipients real names should be included.

```
[Email_Send:
  -To='example@example.com, someone@example.com',
  -From='example@example.com',
  -Subject='An Email',
  -Body=(Include: 'format.lasso')]
```

- The following example shows an [Email_Send] tag with one recipient in the -To parameter and two recipients in the -CC parameter. The Carbon Copy parameter is generally used to include recipients who are not the primary recipient of the email, but need to be informed of the correspondence. The addresses for the carbon copied recipients are stored in variables and concatenated together with a comma between them using a + symbol.

```
[Variable: 'President'='president@example.com']
[Variable: 'Someone'='someone@example.com']
[Email_Send:
  -To='example@example.com',
  -CC=($President + ',' + $Someone),
  -From='example@example.com',
  -Subject='An Email',
  -Body=(Include: 'format.lasso')]
```

- The following example shows an [Email_Send] tag with one recipient in the -To parameter and two recipients in the -BCC parameter. The Blind Carbon Copy parameter can be used to send email to many recipients without disclosing the full list of recipients to everyone who receives the email. Each recipient will receive an email that contains only the address in the -To parameter announce@example.com.

```
[Email_Send:
  -To='announce@example.com',
  -BCC='example@example.com, someone@example.com',
  -From='example@example.com',
  -Subject='An Email',
  -Body=(Include: 'format.lasso')]
```

Advanced Parameters

This section includes parameters for [Email_Send] that are not required for sending basic email messages, but are useful in specific situations. These parameters allow attachments to be included with email messages or for the headers of the outgoing message to be modified. In general, these parameters should only be used if required.

Table 3: Additional Email_Send] Parameters

Keyword	Description
-Priority	Specifies the priority of the message. Values include 'High' or 'Low'. Default is 'Medium'. Optional.
-Attachments	An array of file paths which should be attached to the outgoing message. Optional.
-ReplyTo	The email address that should be used for replies to this message. Optional.
-Sender	The email address that should be reported as the sender of this message. Optional.
-ContentType	The value for the Content-Type header of the message. Optional.
-TransferEncoding	The value for the Transfer-Encoding header of the message. Optional.
-CharacterSet	The character set in which the message should be encoded. Optional.
-ExtraMIMEHeaders	A pair array which defines extra MIME headers that should be added to the email message. Optional.

To change the priority of a sent message:

Specify a -Priority parameter in the [Email_Send] tag. High priority messages will tend to be send faster than lower priority messages. Low priority messages will tend to be sent after all higher priority messages.

Most messages should be sent at the default priority. Sending bulk messages like a newsletter at Low priority will ensure that the normal email from the site is sent as soon as possible rather than waiting for the entire newsletter to be sent first. High priority should be reserved for time dependent messages such as confirmation emails that a site visitor will be looking for immediately within their email client.

To send attachments with an email message:

Files can be included as attachments to email messages using the -Attachments parameter. This parameter takes an array of file paths as a value. When the email is sent, each file is read from disk and encoded using Base-64 encoding. The recipient's email client will automatically decode the attached files and make them available.

Important: The maximum size of an email message including all attachments must be less than 8MB using the [Email_Send] tag. If necessary, larger messages can be sent using the [Email_Immediate] tag described in the **Email Composing** section.

The following example shows a single attachment being sent with an email message. The attachments are named MyAttachment.txt and MyAttachment2.text. They are located in the same folder as the format file which is sending the email.

```
[Email_Send: -Host='mail.example.com',
  -To='example@example.com',
  -From='example@example.com',
  -Subject='AnEmail',
  -Body='This is the body of the Email.',
  -Attachments=(Array: 'MyAttachment.txt', 'MyAttachment2.txt')]
```

To send a message with a Reply-To and Sender header:

The -ReplyTo and -Sender parameters can be used. The -ReplyTo parameter specifies a different address from the -From address which should be used for replies. Most email clients will use this address when composing a response to a message. The -Sender parameter allows an alternate sender from the -From address to be specified. This can be useful if a message is forwarded by Lasso, but the original sender should still be recorded.

```
[Email_Send:=
  -To='example@example.com',
  -From='example@example.com',
  -ReplyTo='repsonses@example.com',
  -Sender='otheruser@example.com',
  -Subject='An Email',
  -Body=(Include: 'format.lasso')]
```

To send a message with extra headers:

The -ExtraMIMEHeaders parameter can be used to send any additional parameters that are required. The value should be an array of name/value pairs. Each of the pairs will be inserted into the email as an additional header.

```
[Email_Send:=
  -To='example@example.com',
  -From='example@example.com',
  -ReplyTo='repsonses@example.com',
  -Sender='otheruser@example.com',
  -Subject='An Email',
  -Body=(Include: 'format.lasso'),
  -ExtraMIMEHeaders=(Array: 'Header' = 'Value', 'Header' = 'Value')]
```

Alternate SMTP Servers'

It is recommended that a default SMTP server be specified in Site Administration in the *Utility > Email > Setup* section. However, if email must be sent through multiple SMTP servers then the host, port, and SMTP AUTH username and password can be specified directly within each [Email_Send] tag.

Table 4: SMTP Server [Email_Send] Parameters

Keyword	Description
-Host	Optional SMTP host through which to send messages. Default is the host defined in Lasso Administration.
-Port	Optional SMTP port. Defaults to 25.
-Username	Specifies the username for SMTP AUTH if required by the SMTP server. If specified a -Passwod is also required. Optional.
-Password	Specifies the password for SMTP AUTH if required by the SMTP server. If specified a -Username is also required. Optional.
-Timeout	Specifies the timeout for the SMTP server in seconds. Optional

To use an alternate SMTP server:

Specify the -Host in the [Email_Send] tag directly. If required the port of the SMTP server can be changed with the -Port parameter. An SMTP AUTH username and password can be provided with the -Username and -Password parameters. And, the -Timeout for the SMTP server can be changed in seconds.

```
[Email_Send: -Host='mail.example.com',
  -Username='SMTPUSER',
  -Password='MySecretPassword',
  -To='example@example.com',
  -From='example@example.com',
  -Subject='An Email',
  -Body=(Include: 'format.lasso')]
```

Composing Email

The [Email_Send] tag handles all of the most common types of email that can be sent through Lasso including plaintext messages, HTML messages, plaintext/HTML alternative messages, and messages with attachments.

For more complex messages structures the [Email_Compose] object can be used directly to create the MIME text of the message. The message can then be sent with the [Email_Queue] tag. Both of these tags are used internally by [Email_Send].

[Email_Compose] accepts the same parameters as [Email_Send] except those which specify the SMTP server and priority of the outgoing message. After creating an email object with [Email_Compose], member tags can be used to add additional text parts, html parts, attachments, or generic MIME parts. This allows very complex email structures to be created with a lot more control than [Email_Send] provides.

The [Email_Queue] tag is designed to be fed an [Email_Compose] object. It requires three parameters, the -Data, -From, and -Recipients attributes of the [Email_Compose] object. In addition, SMTP server parameters and the sending priority can be specified just like in [Email_Send]. Queued emails must be less than 8MB in size including all encoded attachments.

The [Email_Immediate] tag takes the same parameters as the [Email_Queue] tag, but sends the message immediately rather than adding it to the email queue. This tag can be used to send messages larger than 8MB if required. A status of sent or an error will still be added to the queue, but the body of the sent message will not be included. Use of the [Email_Immediate] tag is not recommended since it bypasses the priority, error handling, and connection handling features of the email sending system.

Table 5: Email Composing and Queuing Tags

Tag	Description
[Email_Compose]	Creates an email object. Accepts the same parameters as [Email_Send]: -To, -From, -Body, -HTML, -Subject etc.
[Email_Compose->AddAttachment]	Adds an attachment to an email object.The data of the attachment can be specified directly in the -Data parameter or the path to a file can be specified in the -Path parameter. The name of the attachment can be specified in the -Name parameter.
[Email_Compose->AddTextPart]	Adds a text part to an email object. The text of the part can be specified directly in the -Data parameter or the path to a file can be specified in the -Path parameter.

[Email_Compose->AddHTMLPart]	Adds an HTML part to an email object. The text of the HTML part can be specified directly in the -Data parameter or the path to a file can be specified in the -Path parameter.
[Email_Compose->AddPart]	Adds a generic part to an email object. Requires a parameter -Data which specifies the data for the part. the part must be properly formatted as a MIME part. No formatting or encoding will be performed.
[Email_Compose->Data]	Returns the MIME text of the composed email.
[Email_Compose->From]	Returns the from address of the composed email.
[Email_Compose->Recipients]	Returns a list of recipients of the composed email.
[Email_Queue]	Queues a message for sending. Requires -Data parameter including the MIME text of the email to send. -From specifying the from address for the email. -Recipients an array of recipients for the email. Can also accept -Priority and SMTP server -Host, -Port, -Timeout, -Username, and -Password parameters.
[Email_Immediate]	The same as [Email_Queue], but sends the message immediately without storing it in the database.

To compose an email message:

The [Email_Compose] object can be used to compose an email message. In this example a simple email message is created in a variable message.

```
<?LassoScript
  var: 'message' = (Email_Compose:
    -To='example@example.com',
    -From='example@example.com',
    -Subject='Example Message',
    -Body='Example Message');
?>
```

The text of the composed email message can be viewed by outputing the variable $message to the page.

```
[Var: 'message']
```

Additional text or html parts or attachments can be added using the appropriate member tags with the $message variable. For example, an attachment can be added using the [Email_Compose->AddAttachment] tag as follows:

```
<?LassoScript
  $message->(AddAttachment: -Path='ExampleFile.txt');
?>
```

To queue an email message:

An email message that was created using the [Email_Compose] object can be queued for sending using the [Email_Queue] tag. The following example shows how to send the email message created above. The three required parameters -Data, -From, and -Recipients can all be fetched from the members of the [Email_Compose] object.

```
<?LassoScript
  Email_Queue:
    -Data=$message->Data,
    -From=$emessage->From,
    -Recipients=$message->Recipients;
?>
```

The [Email_Queue] tag will send to the default SMTP server set up in the *Utility > Email > Setup* section of Site Administration. If the email needs to be sent to a different SMTP server the -Host, -Port, -Timeout, -Username, and -Password parameter can be used just as they can with the [Email_Send] tag.

SMTP Type

All communication with remote SMTP servers is handled by a data type called [Email_SMTP]. These connections are normally handled automatically by the [Email_Send], [Email_Queue], and background email sending process.

The [Email_SMTP] type can be used directly for low-level access to remote SMTP servers, but this is not generally necessary.

Table 6: SMTP Tags

Tag	Description
[Email_SMTP]	Creates a new SMTP connection object.
[Email_SMTP->Open]	Requires a -Host that specifies the SMTP host to connect to. Also accepts optional -Port, -Username, -Password, and -Timeout parameters
[Email_SMTP->Send]	Sends a single message to the SMTP server. Requires a -Message parameter with the MIME data for the message, -Recipients with an array of recpient email address, and -From with the email address of the sender.
[Email_SMTP->Command]	Sends a raw command to the SMTP server. The -Send parameter specifies the command to send. The -Expects parameter specifies the numeric result code that is expected as a result. This tag normally returns True or False depending on whether the expected result code was found. The -Read parameter can be specified to have it return the result from the SMTP server.
[Email_SMTP->Close]	Closes the connection to the remote server.

To communicate with an SMTP server:

The [Email_SMTP] object can be used to send one or more messages directly to an SMTP server. In the following example a message is created using the [Email_Compose] tag. That message is then sent to an example SMTP server smtp.example.com using an SMTP AUTH username and password. Once the message is sent the connection is closed.

This example does not perform any error checking and only sends one message. The actual source code for the built-in email sender background process presents a good example of how this code looks in a full working solution.

```
<?LassoScript

  local: 'message' = (Email_Compose:
    -To='example@example.com',
    -From='example@example.com',
    -Subject='Example Message',
    -Body='Example Message');

  local: 'smtp' = (email_smtp);
  #smtp->(open:
    -host='smtp.example.com',
    -port=25,
    -username='SMTPUSER',
    -password='mysecretpassword',
    -timeout=60);
  #smtp->(send:
     -from=#message->from,
    -recipients=#message->recipients,
    -message=#message->data + '\r\n');
  #smtp->close;

?>
```

43

Chapter 43
POP

LDML provides tags that allow email to be downloaded from POP servers and for the downloaded messages to be parsed.

- *Overview* describes basics of POP email downloading.
- *POP Type* describes the [Email_POP] type in detail.
- *Email Parsing* describes how to use the [Email_Parse] type to parse downloaded emails.
- *Helper Tags* describes a number of helper tags that are used internally by the email tags, but can be of general use as well.

Overview

Lasso allows messages to be downloaded from an account on a POP email server. This enabled developers to create solutions such as:

- A list archive for a mailing list.
- A Web mail interface allowing users to check POP accounts.
- An auto-responder which can reply to incoming messages with information.

Lasso's flexible POP implementation allows messages to be easily retrieved from a POP server with a minimal amount of coding. In addition, Lasso allows the messages available on the POP server to be inspected without downloading or deleting them. Mail can be downloaded and left on the server so it can be checked by other clients (and deleted at a later point if necessary).

All messages are downloaded as raw MIME text. The [Email_Parse] type can be used to extract the different parts of the downloaded messages, inspect the headers of the downloaded messages, or to extract attachments from the downloaded messages.

Note: Lasso does not support downloading email from IMAP servers.

POP Type

The [Email_POP] type is used to establish a connection to a POP email server, inspect the available messages, download one or more messages, and to mark messages for deletion.

Table 1: [Email_POP] type

Tag	Description
[Email_POP]	Creates a new POP object. Requires a -Host parameter. Optional -Port and -Timeout parameters. -APOP parameter selects authentication method. If -Username and -Password are specified then connection is opened to server with authentication. -Get parameter specifies what command to perform when calling [Email_POP->Get].
[Email_POP->Size]	Returns the number of messages availble for download
[Email_POP->Get]	Performs the command specified when the object was created. UniqueID by default, or can be set to Retrieve, Headers, or Delete.
[Email_POP->Delete]	Marks the current message for deletion. Optionally accepts a position to mark a specific message.
[Email_POP->Retrieve]	Retrieves the current message from the server. Optionally accepts a position to retrieve a specific message. Optional second parameter specifies the maximum number of lines to fetch for each email.
[Email_POP->UniqueID]	Gets the Uniquid ID of the current message from the server. Optionally accepts a position to get the Unique ID of a specific message.
[Email_POP->Headers]	Gets the headers of the current message from the server. Optionally accepts a position to get the headers of a specific message.
[Email_POP->Close]	Closes the POP connection, performing any specified deletes.
[Email_POP->Cancel]	Closes the POP connection, but does not perform any deletes.

[Email_POP->NOOP]	Sends a ping to the server. Allows the connection to be kept open without timing out.
[Email_POP->Authorize]	Requires a -Username and -Password parameter. Optional -APOP parameter specifies whether APOP authentication should be used or not. Opens a connection to the server if one is not already established.

Note: As of Lasso Professional 8.0.2 any of the tags that accept a position will also accept a Unique ID as a parameter.

Methodology

The [Email_POP] type is intended to be used with the [Iterate] ... [/Iterate] tags to quickly loop through all available messages on the server. The [Email_POP->Size] parameter returns the number of available messages. The [Email_POP->Get] parameter fetches the current message by default or can be set to retrieve the UniqueID of the current message, the Headers of the current message, or even to Delete the current message.

The -Host, -Username, and -Password should be passed to the [Email_POP] object when it is created. The -Get parameter specifies what command the [Email_POP->Get] tag will perform. In this case it is set to UniqueID (the default).

```
[Var: 'myPOP' = (Email_POP:
    -Host='mail.example.com',
    -Username='POPUSER',
    -Password='MySecretPassword',
    -Get='UniqueID')]
```

The [Iterate] ... [/Iterate] tags can then be used on the myPOP object. For example, this code will download and delete every message from the target server. The variable myMSG is set to the unique ID of each message in turn. The [Email_POP->Retrieve] tag fetches the current message and the [Email_POP->Delete] tag marks it for deletion..

```
[Iterate: $myPOP, (Var: 'myID')]
  [Var: 'myID']<br>
  [$myPOP->(Retrieve)]
  [$myPOP->(Delete)]
  <hr>
[/Iterate]
```

Both [Email_Pop->Retrieve] and [Email_POP->Delete] could be specified with the current [Loop_Count] as a parameter, but it is unnecessary since they pick up the loop count from the surround [Iterate] ... [/Iterate] tags.

This example only downloads the text of the messages and displays it. Most solutions will need to use the [Email_Parse] object defined below to parse the downloaded messages before they can be processed.

None of the deletes will actually be performed until the connection to the remote server is closed. The [Email_POP->Close] tag performs all deletes and closes the connection. The [Email_POP->Cancel] tag closes the connection, but cancels all of the marked deletes.

```
[$myPOP->Close]
```

Examples

This section includes examples of the most common tasks that are performed using the [Email_POP] type. See the *Email Parsing* section that follows for examples of downloading messages and parsing them for storage in a database.

To download and delete all emails from a POP server:

Open a connection to the POP server using [Email_POP] with the appropriate -Host, -Username, and -Password. The following example shows how to use [Email_POP->Retrieve] and [Email_POP->Delete] to download and delete each message from the server.

```
<?LassoScript
  Var: 'myPOP' = (Email_POP:
    -Host='mail.example.com',
    -Username='POPUSER',
    -Password='MySecretPassword');
  Iterate: $myPOP, (Var: 'myID');
    Var: 'myMSG' = $myPOP->(Retrieve);
    … Process Message …
    $myPOP->(Delete);
  /Iterate;
  $myPOP->Close;
?>
```

Each downloaded message can be processed using the techniques in the *Email Parsing* section that follows or can be stored in a database.

To leave mail on server and only download new messages:

In order to download only new messages it is necessary to store a list of all the unique IDs of messages that have already been downloaded from the server. This is usually done by storing the unique ID of each message in a database. As messages are inspected the unique ID is compared to see if the message is new or not. No delete of messages is performed in this example.

For the purposes of this example, it is assumed that unique IDs are being stored in a variable array called myUniqueIDs. For each waiting message this variable is checked to see if it contains the uniqueID of the current message. If it does not then the message is downloaded and the unique ID is inserted into myUniqueIDs.

```
<?LassoScript
  Var: 'myPOP' = (Email_POP:
    -Host='mail.example.com',
    -Username='POPUSER',
    -Password='MySecretPassword');
  Iterate: $myPOP, (Var: 'myID');
    If: ($myUniqueIDs >> $myID);
      Var: 'myMSG' = $myPOP->(Retrieve);
      $myUniqueIDs->(Insert: $myID);
      … Process Message …
    /If;
  /Iterate;
  $myPOP->Close;
?>
```

To inspect message headers:

The [Email_POP->Headers] command can be used to fetch the headers of each waiting email message. This allows the headers to be inspected prior to deciding which emails to actually download. In the following example the headers are fetched with [Email_POP->Headers] and two variable needDownload and needDelete are set to determine whether either action should take place.

```
<?LassoScript
  Var: 'myPOP' = (Email_POP:
    -Host='mail.example.com',
    -Username='POPUSER',
    -Password='MySecretPassword',
    -Get='UniqueID');
  Iterate: $myPOP, (Var: 'myID');
    Var: 'needDownload' = False;
    Var: 'needDelete' = False;
    Var: 'myHeaders' = $myPOP->(Headers);
    … Process Headers and set needDownload or needDelete to True …
    If: $needDownload;
      $myPOP->(Retrieve);
    /If;
    If: $needDelete:
      $myPOP->(Delete);
    /If;
  /Iterate;
  $myPOP->Close;
?>
```

The downloaded headers can be processed using the techniques in the *Email Parsing* section that follows.

Email Parsing

Each of the messages which is downloaded from a POP server is returned in raw MIME text form. This section describes the basic structure of email messages, then the [Email_Parse] tag that can be used to parse them into headers and parts, and finally some examples of parsing messages.

Email Structure

The basic structure of a simple email message is shown below. The message starts with a series of headers. The headers of the message are followed by a blank line then the body of the message.

The Received headers are added by each server that handles the message so there may be many of them. The Mime-Version, Content-Type, and Content-Transfer-Encoding specify what type of email message it is and how it is encoded. The Message-ID is a unique ID given to the message by the email server. The To, From, Subject, and Date fields are all specified by the sending user in their email client (or in Lasso using [Email_Send].

```
Received: From [127.0.0.1] BY example.com ([127.0.0.1]) WITH ESMTP;
  Thu, 08 Jul 2004 08:07:42 -0700
Mime-Version: 1.0
Content-Type: text/plain; charset=US-ASCII;
Message-Id: <8F6A8289-D0F0-11D8-B21D-0003936AD948@example.com>
Content-Transfer-Encoding: 7bit
From: Example Sender <example@example.com>
Subject: Test Message
Date: Thu, 8 Jul 2004 08:07:42 -0700
To: Example Recipient <example@example.com>

This is the email message!
```

The order of headers is unimportant and each header is usually specified only once (except for the Received headers which are in reverse chronological order). A header can be continued on the following line by starting the second line with a space or tab. Beyond those standard headers shown here, email messages can also contain many other headers identifying the sending software, logging SPAM and virus filtering actions, or even adding meta information like a picture of the sender.

A more complex email message is shown below. This message has a Content-Type of multipart/alternative. The body of the message is divided into two parts: one text part and one HTML part. The parts are divided using the boundary specified in the Content-Type header (---=_NEXT_fda4fcaab6).

Each of the parts is formatted similarly to an email message. They have several headers followed by a blank line and the body of the part. Each part has a Content-Type and a Content-Transfer-Encoding which specify the type part (either text/plain or text/html) and encoding.

```
Received: From [127.0.0.1] BY example.com ([127.0.0.1]) WITH ESMTP;
  Thu, 08 Jul 2004 08:07:42 -0700
Mime-Version: 1.0
Message-Id: <14501276655.1089394748105@example.com>
From: Example Sender <example@example.com>
Subject: Test Message
Date: Thu, 8 Jul 2004 08:07:42 -0700
To: Example Recipient <example@example.com>
Content-Type: multipart/alternative; boundary="---=_NEXT_fda4fcaab6";

-----=_NEXT_fda4fcaab6
Content-Type: text/plain; charset=ISO-8859-1
Content-Transfer-Encoding: 8bit

This is the text part of the email message!

-----=_NEXT_fda4fcaab6
Content-Type: text/html; charset=ISO-8859-1
Content-Transfer-Encoding: 8bit

<html>
  <body>
    <h3>This is the HTML part of the email message!</h3>
  </body>
</html>

-----=_NEXT_fda4fcaab6--
```

Attachments to an email message are included as additional parts. Typically, the file that is attached is encoded using Base 64 encoding so it appears as a block of random letters and numbers. It is possible for one part of an email to itself have a Content-Type of multipart/alternative and its own boundary. In this way, very complex recursive email structures can be created.

Lasso allows access to the headers and each part (including recursive parts) of downloaded email messages through the [Email_Parse] type.

[Email_Parse] Type

The [Email_Parse] type requires the raw MIME text of an email message as a parameter. It returns an object whose member tags can be used to inspect the headers and parts of the email message. Outputting an [Email_Parse] type to the page will result in a message formatted with the most common headers and the default body part. [Email_Parse] can be used with [Iterate] ... [/Iterate] tags to inspect each part of the message in turn.

Table 2: [Email_Parse] type

Tag	Description
[Email_Parse]	Parses the raw MIME text of an email. Requires a single string parameter. Outputs the raw data of the email if displayed on the page or cast to string.
[Email_Parse->Headers]	Returns an array of pairs containing all the headers of the message.
[Email_Parse->Header]	Returns a single specified header. Requires one parameter, the name of the header to be returned. See also the shortcuts for specific headers listed below. If -Extract is specified then any comments in the header will be stripped. If -Comment is specified then only the comments will be returned. If -SafeEmail is specified then the email address will be obscured for display on the Web.
[Email_Parse->Mode]	Returns the mode from the Content-Type for the message.
[Email_Parse->Body]	Returns the body of the message. Optional parameter specifies the prefered type of body to return (e.g. text/plain or text/html).
[Email_Parse->Size]	Returns the number of parts in the message.
[Email_Parse->Get]	Returns the specified part of the message. Requires a position parameter. The part is returned as an [Email_Parse] object that can be further inspected.
[Email_Parse->Data]	Returns the raw data of the message.
[Email_parse->RawHeaders]	Returns the raw data of the headers.

The following tags are shortcuts which return the value for the corresponding header from the email message.

[Email_Parse->To] [Email_Parse->From]
[Email_Parse->CC] [Email_Parse->Subject]
[Email_Parse->Date] [Email_Parse->Content_Type]
[Email_Parse->Boundary] [Email_Parse->Charset]
[Email_Parse->Content_Disposition]
[Email_Parse->Content_Transfer_Encoding]

Examples

This section includes examples of the most common tasks that are performed using the [Email_Parse] type. See the preceding *POP Type* section for examples of downloading messages from a POP email server.

To display a downloaded message:

Simply use the [Email_Parse] tag on the downloaded message and display it on the page. The [Email_Parse] object will output a formatted version of the email message including a plain text body if one exists.

The following example shows how to download and display all the waiting messages on an example POP mail server. The unique ID of each downloaded message is shown as well as the output of [Email_Parse] in <pre> ...</pre> tags.

```
<?LassoScript
  Var: 'myPOP' = (Email_POP:
    -Host='mail.example.com',
    -Username='POPUSER',
    -Password='MySecretPassword');
  Iterate: $myPOP, (Var: 'myID');
    Var: 'myMSG' = $myPOP->(Retrieve);
?>

    <h3>Message: [Var: 'myID']</h3>
    <pre>[Email_Parse: $myMSG]</pre>
    <hr />

<?LassoScript
  /Iterate;
  $myPOP->Close;
?>
```

To inspect the headers of a downloaded message:

There are three ways to inspect the headers of a downloaded message.

- The basic headers of a message can be inspected using the shortcut tags such as [Email_Parse->From], [Email_Parse->To], [Email_Parse->Subject], etc. The following example shows how to display the basic headers for a message. The variable $myMSG is assumed to be the output from an [Email_POP->Retrieve] tag.

```
[Var: 'myParse' = (Email_Parse: $myMSG)]
<br>To: [Encode_HTML: $myParse->To]
<br>From: [Encode_HTML: $myParse->From]
<br>Subject: [Encode_HTML: $myParse->Subject]
<br>Date: [Encode_HTML: $myParse->Date]
```

```
➜ To: Example Recipient <example@example.com>
  From: Example Sender <example@example.com>
  Subject: Test Message
  Date: Thu, 8 Jul 2004 08:07:42 -0700
```

These headers can be used in conditionals or other code as well. For example, this conditional would perform different tasks based on whether the message is to one address or another.

```
[Var: 'myParse' = (Email_Parse: $myMSG)]
[If: $myParse->To >> 'mailinglist@example.com']
  … Store the message in the mailingt list database …
[Else: $myParse->To >> 'help@example.com']
  … Forward the message to technical support …
[Else]
  … Unknown recipient …
[/If]
```

- The value for any header, including application specific headers, headers added by mail processing gateways, etc. can be inspected using the [Email_Parse->Header] tag. For example, the following code can check whether the message has Spam Assassin headers.

```
[Var: 'myParse' = (Email_Parse: $myMSG)]
[Var: 'Spam_Version' = $myParse->(Header: 'X-Spam-Checker-Version')]
[Var: 'Spam_Level' = $myParse->(Header: 'X-Spam-Level)]
[Var: 'Spam_Status' = $myParse->(Header: 'X-Spam-Status)]

<br>Spam Version: [Encode_HTML: Spam_Version]
<br>Spam Level: [Encode_HTML: $Spam_Level]
<br>Spam Status: [Encode_HTML: $Spam_Status]
```

```
➜ Spam Version: SpamAssassin 2.61
  Spam Level:
  Spam Status: No, hits=-4.6 required=5.0 tests=AWL,BAYES_00 autolearn=ham
```

The spam status can then be checked with a conditional in order to ignore any messages that have been marked as spam (note that the details will depend on what server-side spam checker and version is being used).

```
[If: $Spam_Status >> 'Yes']
  … It is spam …
[Else]
  … It is not spam …
[/If]
```

- The value for all the headers in the message can be displayed using the [Email_Parse->Headers] tag. The following example shows

```
[Var: 'myParse' = (Email_Parse: $myMSG)]
[Iterate: $myParse->Header, (Var: 'Header')]
  <br>[Encode_HTML: $Header->First]: [Encode_HTML: $Header->Second]
[/Iterate]

➜ Received: From [127.0.0.1] BY example.com ([127.0.0.1]) WITH ESMTP;
    Thu, 08 Jul 2004 08:07:42 -0700
Mime-Version: 1.0
Content-Type: text/plain; charset=US-ASCII;
Message-Id: <8F6A8289-D0F0-11D8-B21D-0003936AD948@example.com>
Content-Transfer-Encoding: 7bit
From: Example Sender <example@example.com>
Subject: Test Message
Date: Thu, 8 Jul 2004 08:07:42 -0700
To: Example Recipient <example@example.com>
```

To find the different parts of a downloaded message:

- The [Email_Parse->Body] tag can be used to find the plaintext and HTML parts of a message. The following example shows both the plaintext and HTML parts of a downloaded message.

```
[Var: 'myParse' = (Email_Parse: $myMSG)]
<pre>[Encode_HTML: $myMSG->(Body: 'text/plain']</pre>
<hr />[$myMSG->(Body: 'text/html)]<hr />
```

- The [Email_Parse->Size] and [Email_Parse->Get] tags can be used with the [Iterate] ... [/Iterate] tags to inspect every part of an email message in turn. This will show information about plaintext and HTML parts as well as information about attachments. The headers and body of each part is shown.

```
[Var: 'myParse' = (Email_Parse: $myMSG)]
[Iterate: $myParse, (Var: 'myPart')]
  [Iterate: $myPart->Header, (Var: 'Header')]
    <br>[Encode_HTML: $Header->First]: [Encode_HTML: $Header->Second]
  [/Iterate]
  <br>[Encode_HTML: $myPart->Body]
  <hr />
[/Iterate]
```

To store a downloaded message in a database:

Messages can be stored in a database in several different ways depending on how the messages are going to be used later.

- The simple headers and body of a message can be stored by placing the [Email_Parse] object directly in an inline.

```
<?LassoScript
  Var: 'myPOP' = (Email_POP:
    -Host='mail.example.com',
    -Username='POPUSER',
    -Password='MySecretPassword');
  Iterate: $myPOP, (Var: 'myID');
    Var: 'myMSG' = $myPOP->(Retrieve);
    Var: 'myParse' = (Email_Parse: $myMSG);

    Inline: -Add,
        -Database='example,'
        -Table='archive',
        'email_format' = $myParse.
    /Inline;

  /Iterate;
  $myPOP->Close;
?>
```

- Often it is desirable to store the common headers of the message in individual fields as well as the different body parts. This example shows how to do this.

```
<?LassoScript
  Var: 'myPOP' = (Email_POP:
    -Host='mail.example.com',
    -Username='POPUSER',
    -Password='MySecretPassword');
  Iterate: $myPOP, (Var: 'myID');
    Var: 'myMSG' = $myPOP->(Retrieve);
    Var: 'myParse' = (Email_Parse: $myMSG);

    Inline: -Add,
        -Database='example,'
        -Table='archive',
        'email_format' = $myParse.
        'email_to' = $myParse->To,
        'email_from' = $myParse->From,
        'email_subject' = $myParse->Subject,
        'email_date' = $myParse->Date,
        'email_cc' = $myParse->CC,
        'email_text' = $myParse->(Body: 'text/plain'),
        'email_html' = $myParse->(Body: 'text/html');
    /Inline;

  /Iterate;
  $myPOP->Close;
?>
```

- The raw text of messages can be stored using [Email_Parse->Data]. It is generally recommend that the raw text of a message be stored in addition to the more friendly format. This allows additional information to be extracted from the message later if required.

```
<?LassoScript
  Var: 'myPOP' = (Email_POP:
    -Host='mail.example.com',
    -Username='POPUSER',
    -Password='MySecretPassword');
  Iterate: $myPOP, (Var: 'myID');
    Var: 'myMSG' = $myPOP->(Retrieve);
    Var: 'myParse' = (Email_Parse: $myMSG);

    Inline: -Add,
        -Database='example,'
        -Table='archive',
        'email_text' = $myParse.
        'email_raw' = $myParse->Data;
    /Inline;

  /Iterate;
  $myPOP->Close;
?>
```

Ultimately, the choice of which parts of the email message need to be stored in the database will be solution dependent.

Helper Tags

The email tags use a number of helper tags for their implementation. The following table describes a number of these tags and how they can be used independently.

Table 3: Email Helper Tags

Tag	Description
[Email_Extract]	Strips all comments out of a MIME header. If specified with a -Comment parameter returns the comments instead.
[Email_FindEmails]	Returns an array of all email addresses found in the input.
[Email_SafeEmail]	Obscures an email address by returning the comment portion or only the username before the @ character.
[Email_TranslateBreaksToCRLF]	Translates all return characters and line feeds in the input into \r\n pairs.

Note: The *Encoding* chapter includes documentation of additional helper tags including [Encode_QuotedPrintable], [Decode_QuotedPrintable], [Encode_QHeader], and [Decode_QHeader].

[Email_Extract]

[Email_Extract] allows the different parts of email headers to be extracted. Email headers which contain email addresses are often formatted in one of the three formats below.

```
john@example.com
"John Doe" <john@example.com>
john@example.com (John Doe)
```

In all three of these cases the [Email_Extract] tag will return john@example.com. The angle brackets in the second example identify the email address as the important part of the header. The parentheses in the third example identify that portion of the header as a comment.

If [Email_Extract] is called with the optional -Comment parameter then it will return john@example.com for the first example and John Doe for the two following examples.

[Email_SafeEmail]

[Email_SafeEmail] returns an obscured email address. This tag can be used to safely display email headers on the Web without attracting email address harvesters.

If the input contains a comment then it is returned. Otherwise, the full header is returned. In either case, if the output contains an @ symbol then only the portion of the address before the symbol is returned. This would result in the following output for the example headers above.

```
➜ john
  John Doe
  John Doe
```

44

Chapter 44
HTTP/HTML Content and Controls

This chapter describes the tags which can be used to send and receive files with remote HTTP and FTP servers, include files from remote HTTP and HTTPS servers, interpret HTTP requests, alter the HTTP headers of responses, and to redirect clients to another URL.

- *Include URLs* describes how to include files from remote Web servers, including SSL-protected servers.
- *Redirect URL* describes how to forward clients to a different URL..
- *HTTP Tags* describes how to perform HTTP requests to another Web server.
- *FTP Tags* describes how to perform FTP requests to an FTP server.
- *Cookie Tags* describes how to set and retrieve cookies from a Web client
- *Caching Tags* describes how to cache format file content using Lasso.
- *Server Push* describes how to enable progressive download of HTML pages.
- *Header Tags* describes the tags which allow the current HTTP response headers to be modified.
- *Request Tags* describes the tags which return information about the current HTTP request and allow the HTTP header of the response to be manipulated.
- *Client Tags* describes the tags which return information about the current Web client.
- *Server Tags* describes the tags which return information about the current Web server.

Include URLs

The [Include_URL] tag allows data from another Web server to be included into the contents of a page which is being served to a visitor. This can include HTTP or HTTPS servers. The [Include_URL] tag is replaced by the contents of the remote Web page. Optional parameters allow GET or POST parameters, authentication information, or extra MIME headers to be sent along with the request. Other optional parameters also allow for the MIME headers of the response to be retrieved.

The [Include_URL] tag can be used for any of these purposes.

- To fetch a remote Web page to show to a site visitor. Lasso can be used as a proxy that retrieves the remote page, performs some processing, then sends the page to the visitor.
- To incorporate a portion of a remote Web page into a Lasso format file. A remote Web page can be retrieved, the desired content extracted, and placed into a Lasso format file.
- [Include_URL] can also be used in pages on the same Web server in which Lasso is running.
- To trigger an action in a remote Web application server. [Include_URL] could be used to trigger a CGI on another Web server.
- To trigger an action in a remote Web application server protected via SSL. [Include_URL] could be used to initiate a credit card transaction at a secure HTTPS processing site.
- To trigger an action on the same Web server in which Lasso is running. The [Event_Schedule] tag uses [Include_URL] to call format files at the designated time.

Implementation Note: The [Include_URL] tag is implemented in LDML 8 using libCURL 7.9.5 with OpenSSL for communication with HTTP and HTTPS servers. For more information on libCURL, visit http://curl.sourceforge.net. For more information on OpenSSL, visit http://www.openssl.org.

Table 1: Include URL Tag

Tag	Description
[Include_URL]	Includes a Web page from a remote HTTP or HTTPS server, or from the local server. Requires the target URL as a value parameter and accepts many optional parameters.

The [Include_URL] tag accepts many parameters which define how the remote page should be fetched. These are summarized in *Table 2: [Include_URL] Parameters*.

Code which is returned by [Include_URL] will be HTML encoded by default. Specify -EncodeNone so fetched HTML code will be rendered as part of your Web page. Code which is included with [Include_URL] will not undergo further processing by Lasso unless the [Process] tag is called explicitly on the results.

Table 2: [Include_URL] Parameters

Parameter	Description
URL	Specifies the URL which is to be fetched. Required.
-POSTParams	Specifies an array or map of POST parameters. Optional. The request will be sent using the POST method if this parameter is included. -POSTParams can also be used to post a string to a remote server.
-GETParams	Specifies an array or map of GET parameters. Optional.
-SendMIMEHeaders	Specifies an array of additional MIME headers that should be included with the request. Optional.
-Username	Specifies the username that should be used to authenticate the request. Optional.
-Password	Specifies the password that should be used to authenticate the request. Passwords are encoded in Base 64. Optional.
-RetrieveMIMEHeaders	Specifies the name of a variable which will be set to an array containing all of the MIME headers in the response. Optional.
-NoData	Specifies that the data from the request should not be returned. Optional.
-VerifyPeer	If specified then Lasso will perform a mor rigorous check of the remote server's SSL certificate if the HTTPS protocol is being used.

To include a URL into the current format file:

Use the [Include_URL] tag with the URL of the remote page. The following example shows how to include OmniPilot's front page.

```
[Include_URL: 'http://www.omnipilot.com/']
```

The port of a Web server may also be specified in the URL.

```
[Include_URL: 'http://www.omnipilot.com:80/']
[Include_URL: 'http://www.omnipilot.com:1024/']
[Include_URL: 'http://www.omnipilot.com:8180/']
```

To include a URL from a password-protected HTTPS server into the current format file:

Use the [Include_URL] tag with the -Username and -Password parameters (recommended). The following example shows how to include an SSL-protected page.

```
[Include_URL: 'https://store.example.com/', -Username='my_username',
 -Password='my_password']
```

This can also be achieved without the use of the -Username and -Password parameters by submitting the username and password in the URL.

```
[Include_URL: 'my_username:my_password@https://store.example.com']
```

To simulate an HTML form submission:

An HTML form submission can be simulated using the [Include_URL] tag with the -POSTParams parameter. The inputs of the form should be included as name/value parameters within an array. The following form is shown below as an equivalent [Include_URL] tag.

```
<form action="http://www.example.com/response.lasso" method="POST">
  <input type="hidden" name="-Database" value="Example">
  <input type="hidden" name="-Table" value="Contacts">
  <input type="hidden" name="-KeyField" value="ID">
  <p><input type="submit" name="-FindAll" value=Find All Records"">
</form>
```

The form inputs are assembled into an array as follows.

```
[Variable: 'POST_Params' = (Array: -Database='Example',
  -Table='Contacts', -KeyField='ID', -FindAll='Find All Records')]
```

This variable can then be included in the [Include_URL] tag. The following tag will include the contents of the format file response.lasso with the results of the -FindAll action.

```
[Include_URL: 'http://www.example.com/response.lasso',
  -POSTParams=(Variable: 'POST_Params')]
```

To process an included URL:

Often it is necessary to do some post-processing on an included URL in order to extract a portion of the page. Most included pages will have <html>, <head> and <body> tags which are redundant since they are specified in the format file which is including the remote page. The following code extracts everything between the opening <body> tag and closing </body> tag using a regular expression.

```
[Variable: 'Page_Text' = (Include_URL: 'http://www.omnipilot.com/')]
[String_ReplaceRegExp: (Variable: 'Page_Text'),
  -Find='.*<body[^>]*>(.*)</body>.*',
  -Replace='\1']
```

The regular expression will match the entire included file. .* will match all characters until the <body> tag which is written as <body[^>]*> so that any parameters of the <body> tag will be included. The parenthesized expressions (.*) matches the contents of the body tag which is ended by </body>. Finally, .* matches all other characters until the end of the file.

The replacement is simply \1 which replaces the entire expression with the contents of the first parenthesized expression, the contents of the <body> ... </body> tags.

Redirect URL

The [Redirect_URL] tag can be used to send a client to a different URL. Processing of the current page stops as soon as [Redirect_URL] is called (except for [Handle] ... [/Handle] tags which execute normally). [Redirect_URL] works by altering the HTTP response header which is returned to the client. The use of [Redirect_URL] may override specific settings made using the [Header] ... [/Header] tags. [Redirect_URL] cannot be used on a page below a [Server_Push] tag.

The parameter to [Redirect_URL] must be a full URL and should include the explicit protocol, e.g. http://. Specifying an absolute or relative path to another format file on the Web server will not work. For example, to reference the home page of the Web server www.example.com, the following full URL would be used.

```
http://www.example.com/default.lasso
```

Table 3: Redirect URL Tag

Tag	Description
[Redirect_URL]	Accepts a single parameter which is a URL to which the client should be sent.

To redirect a client to another page:

Specify the full URL of the page to which the client should be redirected within a [Redirect_URL] tag.

- The following examples show how to redirect a client to Microsoft's or Apple's Web sites.

```
[Redirect_URL: 'http://www.microsoft.com/']

[Redirect_URL: 'http://www.apple.com/']
```

- The following example shows how to redirect a client to the login page login.lasso contained in the root folder of the www.example.com Web site.

```
[Redirect_URL: 'http://www.example.com/login.lasso']
```

- The following example shows how to redirect a client to another page redirect.lasso in the same folder as the current page. The [Server_Name] and [Response_Path] tags are used to return the name of the current server and the path to the current response page.

```
[Redirect_URL: 'http://' + (Server_Name) + (Response_Path) + 'redirect.lasso']
```

HTTP Tags

LDML 8 provides HTTP protocol tags that allow developers to send and receive files via HTTP. These tags can generally be used for programmatically uploading and downloading files to and from another Web server.

Implementation Note: The [HTTP_...] tags are implemented in LDML 8 using libCURL 7.9.5. For more information on libCURL, visit http://curl.sourceforge.net.

Table 4: HTTP Tags

Tag	Description
[HTTP_GetFile]	Downloads a file from a remote HTTP server. Requires the -URL parameter, which is the URL from which the file will be downloaded, and the -File parameter, which is the name and path to the local file to be created. Optional -Username and -Password parameters may be used to specifiy a username and password needed to log in to the remote HTTP server. This tag is similar to [Include_URL], except that the file is written to disk rather than being output inside a Lasso format file.

File Permissions Note: The current Lasso user must have adequate file and folder permissions to copy a file or write to folder on the local machine. The same file permissions are required for [HTTP_...] tags as the [File_...] tags. See the *Files and Logging* chapter for more information.

To download a file from an HTTP server:

Use the [HTTP_GetFile] tag. The following example downloads a file named download.zip from the URL http://www.example.com/download.zip to the local documents folder.

```
[HTTP_GetFile: -URL='http://www.example.com/download.zip', -File='/documents/
download.zip']
```

To download a file from a password-protected HTTP server on a non-default port:

Use the [HTTP_GetFile] tag with the -Username and -Password parameters. The following example downloads a file named download.zip at http://www.example.com:1024/private/download.zip where a username and password are required to access to the private folder.

```
[HTTP_GetFile: -URL='http://www.example.com:1024/private/'download.zip, -File='/
documents/download.zip', -Username='my_username', -Password='my_password']
```

FTP Tags

LDML 8 also provides FTP protocol tags that allow developers to send and receive files via FTP. These tags can generally be used for programmatically uploading and downloading files to and from an FTP server.

Note: These tags do not make Lasso Professional 8 an FTP server, but allow Lasso Professional 8 to put and get files from other FTP servers similar to an FTP client.

Implementation Note: The [FTP_...] tags are implemented in LDML 8 using libCURL 7.9.5. For more information on libCURL, visit http://curl.sourceforge.net.

Table 5: FTP Tags

Tag	Description
[FTP_PutFile]	Uploads a local file up to a remote FTP server. Requires the -URL parameter, which is the URL folder and file name of the file to be uploaded, and the -File parameter, which is the path to the local file to be uploaded. Optional -Username and -Password parameters may be used to specifiy a username and password needed to log in to the remote FTP server.

[FTP_GetFile]	Downloads a file from a remote FTP server. Requires the -URL parameter, which is the URL from which the file will be downloaded, and the -File parameter, which is the name and path to the local file to be created. Optional -Username and -Password parameters may be used to specifiy a username and password needed to log in to the remote FTP server.
[FTP_GetListing]	Lists all files accessible to the current user in the remote FTP server URL folder. Outputs an array of maps for each file entry containing the file name, type (directory, file, or link), modification date/time, and size in bytes (for files only). Requires the -URL parameter, which is the URL of the folder to be listed. Optional -Username and -Password parameters may be used to specifiy a username and password needed to log in to the remote FTP server. The username and password values often determine which files are shown by the FTP server.

File Permissions Note: The current Lasso user must have adequate file and folder permissions to copy a file or write to a folder on the local machine. The same file permissions are required for [FTP_...] tags as the [File_...] tags. See the *Files and Logging* chapter for more information.

To upload a file to an FTP server:

Use the [FTP_PutFile] tag. The following example uploads a file named myfile.zip to the URL ftp://ftp.example.com.

```
[FTP_PutFile: -URL='ftp://ftp.example.com/myfile.zip', -File='/documents/myfile.zip']
```

To upload a file to a password-protected FTP server:

Use the [FTP_PutFile] tag with the -Username and -Password parameters. The following example uploads a file named myfile.zip to ftp://ftp.example.com/private/ which requires a username and password to access the private folder.

```
[FTP_PutFile: -URL='ftp://ftp.example.com/private/myfile.zip', -File='/documents/myfile.
zip', -Username='my_username', -Password='my_password']
```

To download a file from an FTP server:

Use the [FTP_GetFile] tag. The following example downloads a file named download.zip from the URL ftp://ftp.example.com/download.zip.

```
[FTP_GetFile: -URL='ftp://ftp.example.com/download.zip', -File='/documents/
download.zip']
```

To download a file from a password-protected FTP server:

Use the [FTP_GetFile] tag with the -Username and -Password parameters. The following example downloads a file named download.zip from ftp://ftp.example.com/private/donwload.zip where a username and password are required to access the private folder.

```
[FTP_GetFile: -URL='ftp://ftp.example.com/private/download.zip', -File='/documents/
download.zip', -Username='my_username', -Password='my_password']
```

To list all files available in a folder on a password-protected FTP server:

Use the [FTP_GetListing] tag with the -Username and -Password parameters. The following example lists all files in the ftp://ftp.example.com/private/ folder that are available to the my_username user.

```
[FTP_GetListing: -URL='ftp://ftp.example.com/private/', -Username='my_username',
-Password='my_password']

➜[Array: (Map: 'FileName'='download.zip',
               'FileSize'='101k',
               'FileType'='File',
               'FileType'='2002-09-29 15:30:00'),
         (Map: 'FileName'='More_Files',
               'FileType'='File',
               'FileType'='2002-09-12 12:14:39')]
```

Note: The modification date for each file using the [FTP_GetListing] tag will be returned using the date format that is used by the remote FTP server.

Cookie Tags

Cookies allow small amounts of information to be stored in the Web browser by Lasso. Each time another page on the same server is loaded, all stored cookies are sent back to Lasso. Multiple cookies can be stored in a client's Web browser and then retrieved on subsequent pages. Cookies can be used to store a client's authentication information, customer ID, site preferences, or even an entire shopping cart. Lasso's sessions make automatic use of cookies to store each client's session ID so server-side variables can be made persistent from page to page.

Please see the *Sessions* chapter for an introduction to sessions and the *Cookies* section of the *Web Application Fundamentals* chapter for a technical introduction to cookies.

Cookies are reliant on support from the client's Web browser for much of their functionality. Preferences for when cookies expire and what domains can retrieve a cookie can be established using the [Cookie_Set] tag, but those preferences must be enforced by the client's Web browser in order for them to have any effect. Clients can even turn off cookie support altogether in most Web browsers.

Cookies are communicated to and from the Web server in the HTTP response header and subsequent HTTP requests. Cookies are not available in the page within which they were set, they are only available in subsequent pages loaded by the same client.

Table 6: Cookie Tags

Tag	Description
[Cookie]	Returns the value for a named cookie. Accepts one required parameter, the name of the cookie whose value should be returned.
[Cookie_Set]	Sets a cookie with a given name and value. See the table below for details about this tag's parameters.
[Client_CookieList]	Returns a string which contains every cookie sent along with the current HTTP request.
[Client_Cookies]	Returns a pair array containing every cookie sent along with the current HTTP request.

Setting Cookies

Cookies are set using the [Cookie_Set] tag. The one required parameter of the tag is a user-defined name/value parameter specifying the name of the cookie and the value which is to be stored under that name. However, it is recommended that all parameters of the [Cookie_Set] tag be specified in order to ensure compatibility with the greatest range of Web browsers.

Table 7: [Cookie_Set] Parameters

Tag	Description
Name/Value	A name/value parameter defines the name of the cookie and the value which should be stored under that name. Required.
-Expires	The number of minutes until the cookie expires. Optional. If left blank, most cookies expire when the client quits their Web browser application. A negative value instructs a Web browser to expire a cookie immediately.
-Domain	The domain of the cookie. Cookies will only be sent to servers with this domain. Optional, but recommended.
-Path	The path of the cookies. Cookies will only be sent to pages which are in subfolders of this path. Optional, but recommended.
-Secure	If specified then the cookie will only be transmitted back through secure HTTPS protocol.

The total number of characters of the name/value parameter and all other parameters of the [Cookie_Set] tag must be less than 2048 characters. The name of the cookie must be less than 1024 characters. The value of the cookie must be less than 1024 characters. The -Expires parameter should be no more than 10 digits. The -Path and -Domain parameters should be no more than 256 characters each.

Note: The parameters required for the [Cookie_Set] tag vary depending on what Web clients are being used by site visitors. In general, it is safest to specify each of -Expires, -Domain, and -Path in order to ensure maximum compatibility.

To set a cookie:

Use the [Cookie_Set] tag with each of the parameters defined. The following example shows how to create a cookie named Cookie_Name with the value Cookie_Value for the domain example.com with an expiration time of 24 hours (1440 minutes). The path is set to / so all pages in the site will have access to the cookie.

```
[Cookie_Set: 'Cookie_Name'='Cookie_Value'
  -Domain='example.com',
  -Path='/',
  -Expires=1440]
```

The example above shows -Domain set to example.com. Setting -Domain to the name of the domain rather than to the name of a particular Web server ensures that any server within the domain can retrieve the cookie.

For example, mail.example.com and images.example.com could retrieve Cookie_Name set above.

To set a cookie that can be retrieved by another Web server:

The -Domain parameter can be used to define that a cookie be returned to another Web server. This can be useful for interactions where a customer needs to be tracked as they move between different Web servers.

The following example shows how a cookie can be set so that it will be served to the server www.otherserver.com. This cookie will not be sent to subsequent pages loaded on the current server by the client. It will only be served to www.otherserver.com when they visit that Web server.

```
[Cookie_Set: 'Cookie_Name'='Cookie_Value'
  -Domain='otherserver.com',
  -Path='/',
  -Expires=1440]
```

Note: Many Web browsers have a preference which prohibits cookies being set which will be read by a different server. If the -Domain parameter is not specified to the same domain as the Web server hosting Lasso Service then the cookie may not be set in all Web browsers.

To delete a cookie:

Cookies can be deleted by setting the -Expires parameter to a negative number or by resetting the value of the cookie. It is good practice to delete cookies which are no longer needed. Some Web browsers do not delete expired cookies properly so extra data may end up being sent to the Web server with every URL request. The following example shows how to delete the cookie Cookie_Name by setting it to the empty string '' and setting its expiration to -1.

```
[Cookie_Set: 'Cookie_Name'='',
  -Domain='example.com',
  -Path='/',
  -Expires=-1]
```

Retrieving Cookies

Cookies are retrieved by name. However, only cookies which were sent by the client's Web browser along with the current HTTP request can be retrieved by Lasso. The Web browser determines what cookies to send based on the domain, path, and expiration set for each cookie. The implementation differs from browser to browser so some client's may not support all types of cookies.

To retrieve a cookie:

If a cookie is available it can be retrieved using the [Cookie] tag. This tag accepts a single parameter which is the name of the cookie to be retrieved. The tag will return an empty string if the cookie is not defined. The following code returns the value Cookie_Value for the cookie Cookie_Name.

```
[Cookie: 'Cookie_Name'] ➜ Cookie_Value
```

To retrieve all cookies:

There are two ways to retrieve a list of all cookies that have been sent along with the current HTTP request.

- Use [Client_Cookies] to return an array of all cookies set for the current HTTP request. [Client_Cookies] returns a pair array where each pair contains the name and value of a cookie. The following example shows how to display all cookies that are currently set using [Loop] ... [/Loop] tags. The result is the single Cookie_Name with value Cookie_Value.

```
[Loop: (Client_Cookies->Size)]
  [Variable: 'Temp_Cookie' = (Client_Cookies->(Get: (Loop_Count)))]
  <br>[Encode_HTML: $Temp_Cookie->First + ': ' + $Temp_Cookie->Second]
[/Loop]

➜ <br>Cookie_Name: Cookie_Value
```

- Use [Client_CookieList] to return a string that contains the names and values of all cookies set for the current HTTP request. This tag can be used for debugging purposes to quickly display a list of all cookies. The cookies are returned separated by semi-colons ; with the name of the cookie separated from the value by an equal sign =. The following example shows a single cookie Cookie_Name with value Cookie_Value.

```
[Client_CookieList] ➜ Cookie_Name: Cookie_Value;
```

To check if a cookie is set:

Use the [Array->Find] tag to search through the [Client_Cookies] tag. If the [Array->Find] returns a pair value then the cookie is set. If it returns Null then the cookie is not set. Using this method is more reliable than simply calling [Cookie] with the name of a cookie since it is impossible to tell whether a returned value of the empty string '' is due to a cookie not being set or due to a cookie being set to the empty string. The following example returns True since the cookie Cookie_Name is set.

```
[If: (Client_Cookies)->(Find: 'Cookie_Name') != Null] True [/If]

➜ True
```

Checking for Cookie Support

Since cookies can be deactivated within a client's Web browser it is important to check whether cookies are supported before allowing a client to view portions of a Web site that require cookies. The following code will perform a check for cookie support by setting a cookie and then redirecting the client to another page which checks the cookie value.

To check whether cookies are supported:

The page cookie_set.lasso contains a [Cookie_Set] tag that sets a cookie Test_Cookie to the value Test_Value and a [Redirect_URL] tag which sends the client to the page cookie_check.lasso.

```
[Cookie_Set: 'Test_Cookie'='Test_Value',
  -Domain='example.com',
  -Path='/']
[Redirect_URL: 'http://www.example.com/cookie_check.lasso']
```

The page cookie_check.lasso checks to see if the cookie is set using the [Array->Find] tag on [Client_Cookies]. If it is set, it redirects the user to the default page of the Web site default.lasso. If cookies are not supported then the user is redirected to the page error.lasso which contains a warning message.

```
[If: (Client_Cookies)->(Find: 'Cookie_Name') != Null]
  [Redirect_URL: 'http://www.example.com/default.lasso']
[Else]
  [Redirect_URL: 'http://www.example.com/error.lasso']
[/If]
```

Caching Tags

New content caching tags in Lasso Professional 8 allow a portion of a page to be cached either to a global variable or to a session. Lasso is able to cache the output of dynamic LDML code and data source queries, as well as the values of named LDML variables for later use. These tags allow developers to reduce database and server load by having Lasso only recalculate various portions of a page periodically.

The first time a page with [Cache] ... [/Cache] tags is hit, the contents of the tags are remembered for a specified period of time. The Lasso cache can be set to refresh itself at scheduled time intervals, or when certain conditions are met.

Important: When using the cache tags, it is important to know that any dynamic changes that occur in cached LDML code will be ignored, and only the original cached values will be output until the cache expires.

Caching Output Values

Lasso allows the values output by dynamic LDML code to be cached so that the dynamic operations (such as a data query) are not performed again until the cache expires or is dumped. This is accomplished by surrounding the code to cache with the [Cache] ... [/Cache] container tags, which are described below.

Table 8: [Cache] Tag

Tag	Description
[Cache] ... [/Cache]	Container tag used for caching elements on a page in Lasso's internal cache. Requires a -Name parameter which specifies the name of the cache, and several optional parameters may be used as shown in the following table.

The optional parameters for the [Cache] ... [/Cache] container tags are described in *Table 9: [Cache] Tag Parameters*.

Table 9: [Cache] Tag Parameters

Tag	Description
-Name	Specifies the name of the cache. The name of the cache identifies the cached contents so it can be referenced on several pages. This is the only required parameter.
-Expires	Specifies how many seconds the cached contents should last. If the cached contents is older than the time interval specified, then new content values will be cached on the next page load.
-Condition	Allows arbitrary refresh conditions to be specified. Accepts a boolean value of True or False and refreshes the contents immediately when True. Conditional expressions may be used to output the required True or False value. For example, -Condition=((Action_Param: 'Refresh') == 'Yes') refreshes the cache if the action param is equal to Yes.
-Session	Specifies the name of a session that the cached content should be stored in. This allows the cached content to be user specific. If no -Session parameter is specified, then the content will be stored in a global variable instead.

-UseGlobal	Can be used in concert with -Session to store cached data in both a global variable and a session. All caches are stored in a global variable by default if neither -Session or -UseGlobal is specified.
-Key	Can be used to secure a cache on a server. Requires a password string value. Once a cache has been created with a -Key parameter and value, the cache will only be returned if subsequent [Cache_...] tags contain the same -Key parameter and value. Optional.

Restart Note: Caches stored in global variables do not persist between restarts. They must be refreshed the first time they are hit on a Lasso page. Global variables are used by default unless a -Session parameter is specified.

To cache content with an expiration:

Surround the portion of your page that you wish to cache with the [Cache] ... [/Cache] container tags using the -Expires parameter. In the example below, the output of the data source query surrounded by the [Cache] ... [/Cache] tags will be stored in a global variable, and then output consistently with each page refresh (without performing the data source query again) until the cache expires 3600 seconds later.

```
[Cache:
  -Name='Cache_Name',
  -Expires=3600]

  [Inline: -Database='Contacts', -SQL='Select * from people where ID < 3']
    [Field:'First_Name'] [Field:'Last_Name'] - [Field:'Company']<br>
  [/Inline]

[/Cache]

➜ John Doe - OmniPilot
  Jane Doe - OmniPilot
```

To cache content with no expiration:

Use the [Cache] ... [/Cache] tags without the -Expires parameter. The example below shows a cached data source query that never expires. This means that the first result set out put by the contained [Inline] ... [/Inline] tags will be the results that are always output.

```
[Cache:
  -Name='Cache_Name']

  [Inline: -Database='Contacts', -SQL='Select * from Contacts.People']
    [Field:'First_Name'] [Field:'Last_Name'] - [Field:'Company']<br>
  [/Inline]
```

```
[/Cache]

➜ John Doe - OmniPilot
  Jane Doe - OmniPilot
```

To cache content to a session instead of a global variable:

Use the [Cache] ... [/Cache] tags with the -Session parameter. This stores the cached data in a session instead of a global variable, which means that the cached data will expire when the session expires. In the example below, a [Date] tag cached will expire in three hours when the session expires.

```
[Session_Start: -Name='Session_Name', -Expires=3600]

[Cache:
  -Name='Cache_Name',
  -Session='Session_Name']

[Date]

[/Cache]

➜ 9/29/2003 19:13:00
```

To cache content to both a session and a global variable:

Use the [Cache] ... [/Cache] tags with both the -Session and -UseGlobal parameters. This will store the data in both a session and a global variable for maximum control over the cache. In the example below, a [Date] tag cached will never expire unless both the cache is dumped in Lasso Administration and the end-user deletes their session cookie..

```
[Session_Start: -Name='Session_Name']

[Cache:
  -Name='Cache_Name',
  -Session='Session_Name',
  -UseGlobal]

[Date]

[/Cache]

➜ 9/29/2003 19:13:00
```

To conditionally refresh a page:

Use the [Cache] ... [/Cache] tags with the -Condition parameter. The example below conditionally refreshes the cache if the value of [Action_Param:'Cache'] is equal to Yes.

```
[Cache:
  -Name='Cache_Name',
  -Condition=((Action_Param:'Cache') == 'Yes')]

[Date]

[/Cache]

➜ 9/29/2003 21:57:00
```

To create a secure cache:

Use the [Cache] ... [/Cache] tags with the optional -Key parameter. The example below creates a cache named Cache_Name with a key value of password. Only subsequent cache tags containing the parameter -Key='password' will be able to access this cache.

```
[Cache:
  -Name='Cache_Name',
  -Expires=3600,
  -Key='password']

  [Inline: -Database='Contacts', -SQL='Select * from people where ID < 3']
    [Field:'First_Name'] [Field:'Last_Name'] - [Field:'Company']<br>
  [/Inline]

[/Cache]
```

Caching LDML Objects

LDML 8 also includes tags that can cache LDML variables directly without having to use a container tag. When these tags are used, all instances of the LDML variable will be replaced by its cached value until the cache expires or is dumped.

Table 10: LDML Object Cache Tags

Tag	Description
[Cache_Object]	Caches the value of a named LDML variable. Uses the same parameters as the [Cache] ... [/Cache] tags, but requires a -Content parameter that specifies the name of an LDML object variable to be cached. This tag always returns the object value to the page.
[Cache_Store]	Works the same as [Cache_Object], except it does not return a value to the page. Also, the optional -Conditional parameter cannot be used with [Cache_Store].

To cache an LDML object and return its value to the page:

Use the [Cache_Object] tag. The example below adds a variable named Data to the cache. Whenever the Data variable is called while the cache is not expired, any instance of the Data variable will be replaced with its cached value.

```
[Var:'Data'='This is some data']
[Cache_Object: -Name='Cache_Name', -Expires=3600, -Content=$Data]
```

➜ This is some data

To cache an LDML object without returning its value to the page:

Use the [Cache_Store] tag. The example below adds a variable named Data to the cache. Whenever the Data variable is called while the cache is not expired, any instance of the Data variable will be replaced with its cached value.

```
[Var:'Data'='This is some data']
[Cache_Store: -Name='Cache_Name', -Expires=3600, -Content=$Data]
```

Cache Control Tags

Additional cache control tags allow values stored in caches to be programmatically fetched and emptied. These tags are described in *Table 11: Cache Control Tags*.

Table 11: Cache Control Tags

Tag	Description
[Cache_Fetch]	Outputs the contents of a Lasso cache. Requires a -Name parameter, which specifies the name of the cache. An optional -Session parameter specifies the session that contains the cache, if applicable.
[Cache_Empty]	Clears a specified cache. The cached contents will be forced to reload at the next page load. Requires a -Name parameter which specifies the name of the cache. An optional -Session parameter specifies the session that contains the cache, if applicable.

To return the contents of a cache:

Use the [Cache_Fetch] tag, where the name of the cache to return is specified in the -Name parameter. The example below returns the value of a cached [Date] tag in a cache named Cache_Name.

```
[Cache_Fetch: -Name='Cache_Name']
```

```
➜ 09/29/2003 19:13:00
```

To empty a cache:

Use the [Cache_Empty] tag, where the name of the cache to empty is specified in the -Name parameter.

```
[Cache_Empty: -Name='Cache_Name']
```

Controlling Caches in Lasso Administration

The Lasso global administrator has global control over all caches stored on the Lasso Professional 8 server. The *Utility > Cache* section of Lasso Administration provides information about all current caches, allows caches to be reset, and allows preferences for the caching mechanism to be set.

For more information, see the Site *Administration Utilities* chapter in the Lasso Professional 8 Setup Guide.

Note: Caches stored in sessions are not visible in Lasso Administration and do not maintain statistics.

Server Push

The [Server_Push] tag can be used to progressively download HTML content to a client that supports progressive downloads. All data in the format file up until the location of the [Server_Push] tag is sent to the client, but processing of the page continues normally. Multiple [Server_Push] tags can be used to send a page to a client in as many segments as desired.

Note: Some Web servers do not support [Server_Push]. These Web servers buffer all output from Lasso and stream it to the Web clients themselves.

Most Web browsers will accept progressive downloads only if the [Server_Push] tag is placed outside of any HTML container tags (except for <html> ... </html> and <body> ... </body>). In particular, the [Server_Push] tag should not be used within <table> ... </table> tags.

Lasso buffers the output of container tags such as [Loop] ... [/Loop], [If] ... [Else] ... [/If] and [Records] ... [/Records]. The [Sever_Push] tag can only be used outside of any container tags.

Warning: The [Server_Push] tag is incompatible with the [Header] ... [/Header], [Content_Type], [Redirect_URL], [Cookie_Set], and [Session_Start: -UseCookie] tags. These tags should not be used on pages which are being sent progressively using [Server_Push].

Table 12: Server Push Tag

Tag	Description
[Server_Push]	Instructs Lasso to send as much of the current format file to the client as possible.

To progressively download a page:

Use the [Server_Push] tag to send sections of the page as they are finished processing. The following example uses a [Server_Push] to force the first part of the page to download, then performs a search using [Inline] ... [/Inline] tags. The header of the page should be visible while the search completes.

```
<h2>Search Results</h2>
[Server_Push]
[Inline: -Database='Contacts',
    -Table='People',
    -KeyField='ID',
    -FindAll]
  [Records]
    <br>[Field: 'First_Name'] [Field: 'Last_Name']
  [/Records]
[/Inline]
```

Header Tags

The header tags allow the contents of the HTTP response header to be modified before the results of the current format file are served to the visitor. In addition to the tags described in *Table 13: Header Tags*, the [Cookie_Set] tag, [Redirect_URL] tag, and [Server_Push] tag also alter the HTTP response header.

The [Content_Type] and [Header] ... [/Header] tags should be included as the first tags in a format file whenever possible. This ensures that any additional tags that modify the HTTP response header will modify the header defined by these tags.

Lasso uses the character set specified in the [Content_Type] tag to determine how to encode the results of processing a format file before transmitting them to the client's Web browser. By default Lasso will transmit all results in the Unicode single-byte standard UTF-8. See below for examples of how to set the character set to something different.

Table 13: Header Tags

Tag	Description
[Content_Type]	Sets the MIME type of the current HTTP response.
[Header] ... [/Header]	Sets the HTTP header of the response to the contents of the container tags.

Content Type

Use the [Content_Type] tag to set the MIME type for a format file and to set the character set which will be used to transmit the results to the client. The client's Web browser will use this content type and character set to determine how to display the returned data to the client. The [Content_Type] tag should be one of the first tags within a format file.

To set the content type of a format file:

- The following example shows how to return HTML data in a format file encoded using UTF-8. This is the default state for the [Content_Type] tag.

```
[Content_Type: 'text/html; charset=UTF-8']
```

- The following example shows how to return HTML data using the Latin-1 (ISO 8859-1) character set. Some older browsers or other Web clients may expect data to be in this character set..

```
[Content_Type: 'text/html; charset=iso-8859-1']
```

- The following example shows how to return XML data in a format file with the text/xml MIME type and UTF-8 character set. See the *XML* chapter for more information.

```
[Content_Type: 'text/xml; charset=utf-8']
```

- The following example shows how to return WML data in a format file with the text/vnd.wap.wml MIME type and UTF-8 character set. This tag is used when serving data to WAP browsers. See the *Wireless Devices* chapter for more information.

```
[Content_Type: 'text/vnd.wap.wml; charset=utf-8']
```

Header Tag

If the [Header] ... [/Header] tags are used within a format file, then the HTTP response header will be set to the contents of the tags. This is a low-level tag that should only be used by developers who are familiar with the structure of HTTP response headers.

The [Content_Type], [Redirect_URL], [Server_Push] and [Cookie_Set] tags can all be used to modify portions of the HTTP response header. These tags are the preferred method for modifying the portions of the header that they affect.

Rules for use of the [Header] ... [/Header] tags:

- The literal string HTTP must appear within the [Header] ... [/Header] tags. Everything before this literal string will be removed from the HTTP response header.
- The first line of a header is a status line that has the following form, HTTP/Version Status_Code Status_Message. For example a soft redirect using the HTTP/1.0 standard is specified as follows.

  ```
  HTTP/1.0 302 FOUND
  ```

- Carriage returns within the [Header] ... [/Header] tags will be replaced by carriage return/line feed pairs.
- All [Header] ... [/Header] tags must end with an empty line. No empty lines are allowed within the [Header] ... [/Header] tags except for the required last line.
- No spaces are allowed at the start of any line within the [Header] ... [/Header] tags.

Headers set using the [Header] ... [/Header] tags must follow the standards defined by the World Wide Web Consortium. Please see their documentation of the HTTP standard for more information.

```
http://www.w3c.org/
```

To redirect a user to another URL:

Use the [Header] ... [/Header] tags. The following header will redirect the client to the URL specified on the Location line. The URI line is included for compatibility with older browsers. Notice that the destination is U R I, not U R L.

```
[Header]
HTTP/1.0 302 FOUND
Location: http://www.example.com/default.lasso
URI: http://www.example.com/default.lasso
Server: Lasso Professional 8

[/Header]
```

Note: The [Redirect_URL] tag, documented earlier in this chapter, can also be used to redirect a visitor to a different URL.

To submit a form without reloading the page:

Use the [Header] ... [/Header] tags with a 204 partial content response. The 204 response instructs the client's Web browser that an action has been taken, but that the current rendered page should not be altered. The user can then enter another item into the form and submit it again.

```
[Header]
HTTP/1.0 204
Server: Lasso Professional 8

[/Header]
```

To request authentication information from a client:

Use the [Header] ... [/Header] tags with a 401 unauthorized response. The 401 response instructs the client's Web browser that authentication is required to access the desired resource. The WWW-Authenticate line in the header names the realm which the user is attempting to access so subsequent requests for authentication information will properly retrieve stored passwords as defined by the features of the client's Web browser. The following code asks for authentication for a realm named Example.

```
[Header]
HTTP/1.0 401
WWW-Authenticate: Basic realm="Example"
Server: Lasso Professional 8

[/Header}
```

The first time a client's Web browser receives this response it will check for a stored password or prompt the client to enter a username and password for the specified realm. If the client's Web browser receives the same response again (or sometimes after several authentication attempts) it will assume that the user is not authorized to access the page in question.

Note: See the *Authentication Tags* section of the *Control Tags* chapter for information about LDML tags that automatically prompt for authentication information.

Request Tags

Lasso includes a number of tags that return information about the current HTTP request. These tags can be used to inspect the URL, GET arguments, POST arguments, form method, or even the raw HTTP request. These tags are summarized in *Table 14: Request Tags*.

Table 14: Request Tags

Tag	Description
[Client_ContentLength]	Returns the length in characters of the current POST parameters.
[Client_ContentType]	Returns the MIME type requested by the current HTTP request.
[Client_FormMethod]	Returns the method used to load the current page, either GET or POST.
[Client_GETArgs]	Returns a string containing all the arguments passed along with the URL in the current request.
[Client_GETParams]	Returns a pair array containing an element for each parameter passed along with the URL in the current request.
[Client_Headers]	Returns the text of the HTTP request which called this page.
[Client_Password]	Returns the password specified by the current client.
[Client_POSTArgs]	Returns a string containing all the arguments passed along with the URL as a POST parameter in the current request.
[Client_POSTParams]	Returns a pair array containing an element for each parameter passed along with the URL as a POST parameter in the current request.
[Client_Username]	Returns the username specified by the current client.
[Response_FilePath]	Returns the path to the file which is being served from the Web server root.
[Response_LocalPath]	Returns the path to the Web server root.
[Response_Path]	Returns the folder from which the current file is being served relative to the Web server root.
[Response_Realm]	Returns the name of the current realm reported by the Web server.

To provide a link to the current Web page:

The [Response_FilePath] tag can be used to provide a link that reloads the current Web page. The following example provides a simple link that reloads the current Web page without any GET or POST parameters.

```
<a href="[Response_FilePath]"> Reload this page </a>
```

To display the current GET parameters:

Use the [Loop] ... [/Loop] tags and the [Array->Get] tag to loop through the [Client_GetParams] array. The results are shown for the following URL: http://www.example.com/default.lasso?name1=value1&name2=value2.

```
[Loop: (Client_GetParams)->Size]
  [Variable: 'GET_Variable' = (Client_GetParams)->(Get: (Loop_Count))]
  <br>[Encode_HTML: $GET_Variable->First] = [Encode_HTML: $GET_Variable-
>Second]
[/Loop]

➜ <br>name1 = value1
  <br>name2 = value2
```

The same methodology can be used for the output of the [Client_PostParams] tag.

To accept only POST parameters:

Check the [Client_FormMethod] tag to see whether it equals GET or POST. If it is not set to the desired value then redirect the client to another page. The following code redirects the user to error.lasso if the current page is not loaded with POST parameters.

```
[If: (Client_FormMethod) != 'POST']
  [Redirect_URL: 'http://www.example.com/error.lasso']
[/If]
```

Note: It is possible to load a page with both POST and GET parameters so a complete solution needs to check that a POST form method was used and scan the GET parameters.

Client Tags

Lasso includes a number of tags that return information about the current client including what type of browser they are using and where their client machine is located. These tags are summarized in *Table 15: Client Tags*.

Table 15: Client Tags

Tag	Description
[Client_Address]	Returns the host name of the current client.
[Client_Browser]	Returns the type of browser used by the current client.
[Client_IP]	Returns the IP address of the current client.
[Client_Type]	Returns the type of browser used by the current client.

Note: Lasso also includes a set of [WAP_...] tags that return information about clients using WAP browsers. See the ***Wireless Devices*** chapter for more information.

To check whether a client is using a specific browser:

The [Client_Browser] tag can be used to return the type of browser the client is using. The following example checks whether the browser type contains Netscape and displays an appropriate message if it does.

```
[If: (Client_Browser) >> 'Netscape']
  <br>You are using a supported Netscape browser.
[Else]
  <br>You are using an unsupported browser of type: [Client_Browser].
[/If]
```

Server Tags

Lasso provides a number of tags which return information about the current Web server. The information returned by the tags in *Table 16: Server Tags* can be used to determine whether a page is being served normally or securely or to output information to log files.

Table 16: Server Tags

Tag	Description
[Server_Name]	Returns the name of the current server.
[Server_Port]	Returns the port which the current request is being served. Usually 80 for normal HTTP requests or 443 for secure HTTPS requests.

To check whether a page is being served securely:

Check the output of the [Server_Port] tag. Most Web servers serve normal HTTP traffic on port 80 and secure, SSL encrypted HTTPS traffic on port 443. The following example displays a reassuring message if the page is being served securely or a warning if the page is not being served securely.

```
[If: (Server_Port) == 80]
  <p><font color="red">Warning: this page is not being served securely.</font>
[Else: (Server_Port) == 443]
  <p><font color="blue">Don't panic: this page was served securely.</font>
[Else]
  <p><font color="yellow">Caution: this page is served from an unknown port.</font>
[/If]
```

To log information about server requests to a log file:

Use the [Server_...] and [Client_...] tags to return information about the current visitor and what page they are visiting. The following code will log the current date and time, the visitor's IP address, the name of the server

and the page they were loading, and the GET and POST parameters that were specified.

```
[Log: 'E://Logs/LassoLog.txt']
[Date]
[Client_IP] [Server_Name] [Response_FilePath]
[Client_GETArgs] [Client_POSTArgs]
[/Log]
```

See the *Files and Logging* chapter for more information a about the [Log] ... [/Log] tags.

45

Chapter 45
XML-RPC

Lasso can host XML-RPC methods or can call methods hosted on other servers..

- *Overview* introduces XML-RPC methods.
- *Calling a Remote Procedure* discusses how to use the [XML_RPCCall] tag to call a remote procedure.
- *Creating Procedures* instructions can be found in the *Custom Tags* chapter.
- *Processing an Incoming Call* discusses the low-level details of how incoming XML-RPC calls can be processed.

Overview

XML-RPC is a standard which allows remote procedure calls to be made between different servers on the Internet. A remote procedure call is similar to a CGI call (i.e. via [Include_URL]) to a different machine on the Internet, but by passing the parameters of the procedure call and results in a standard XML format, XML-RPC is more flexible than traditional CGIs.

One way to think of XML-RPC in Lasso is that it is a method of calling an LDML tag which happens to be located on a different Web server. Lasso can act as both ends of an XML-RPC call, enabling two Lasso servers to communicate with each other, or communication can be established between a Lasso server and another server that supports XML-RPC.

The first part of this section documents how to use the [XML_RPCCall] tag to make remote procedure calls. This technique is sufficent to make use of XML-RPC methods that are available on other servers. The second part of

this section documents the low-level [XML_RPC] object and its methods for calling and responding to XML-RPC requests.

Calling a Remote Procedure

A remote procedure can be called using the [XML_RPCCall] tag. This tag uses the low-level XML-RPC data type to create a remote procedure call and to evaluate the results.

Table 1: [XML_RPCCall] Tag

Tag	Description
[XML_RPCCall]	Calls a remote procedure and returns the result. Accepts three parameters. -Host is the URL of the remote host. -Method is the method to be called. -Params is an array of parameters to be passed to the remote server.

The -Host parameter defaults to the current host. The -Method parameter is required, but defaults to Test.Echo for testing purposes. The -Params parameter is only required if the method requires parameters.

Errors are returned from the tag through the [Error_CurrentError]. This tag will report [Error_NoError] if no error occured. Otherwise it will print out a detailed error message. The result of the [XML_RPCCall] tag when an error occurs is always Null.

To call a remote procedure:

Use the [XML_RPCCall] tag. In the following example the Test.Echo method on the current Lasso server is called. This method simply echoes its parameters back to the caller. The path to have Lasso process an incoming XML-RPC request is /Lasso/RPC.LassoApp.

```
[XML_RPCCall: -Host='http://127.0.0.1/Lasso/RPC.LassoApp',
  -Method='Test.Echo', -Params='Hello World!']

➜ Hello World!
```

To list all available methods on a server:

Lasso supports a number of built-in XML-RPC methods. These are listed in the *XML-RPC Built-In Methods* table. A list can be obtained direct from Lasso using the XML-RPC method System.ListMethods. Sample output is shown below.

```
[XML_RPCCall: -Host='http://127.0.0.1/Lasso/RPC.LassoApp',
  -Method='System.ListMethods']
```

```
➜ (Array: (System.ListMethods), (System.MethodHelp), (System.MethodSignature),
  (System.MultiCall), (Test.Echo),
```

To call multiple methods on a server:

The System.MultiCall method can be used to call multiple methods on a remote server in a single request. This enables several XML-RPC methods to be called without the overhead of making individual HTTP connections to the remote server.

The following example performs two Test.Echo calls in a single System.MultiCall method.

```
[XML_RPCCall: -Host='http://127.0.0.1/Lasso/RPC.LassoApp',
  -Method='System.MultiCall', -Params=(Array:
    (Map: 'MethodName'='Test.Echo', 'Params'='Hello World!'),
    (Map: 'MethodName'='Test.Echo', 'Params'='Hello Again.'))]

➜ (Array: (Array: 'Hello World!'), (Array: 'Hello Again.'))
```

Note that the results are returned as an array with the return value of each particular method as an element.

Built-In Methods

Lasso supports a number of built-in XML-RPC methods which most XML-RPC processors are expected to have available. These built-in methods are implemented in Startup.LassoApp located in the LassoStartup folder.

Table 2: XML-RPC Built-In Methods

Method	Description
System.ListMethods	Returns an array of method names available on the server.
System.MethodHelp	Requires a method name as parameter. Returns a description of what the method does.
System.MethodSignature	Returns an error message since Lasso does not support message signatures.
System.MultiCall	Requires an array of maps each with a MethodName and a Params element. Returns an array of results for each of the individual methods.
Test.Echo	Echoes the parameters back to the caller.

Note: Lasso also defines a series of validator methods used to test the XML-RPC functionality for proper adherence to the standard.

XML-RPC and Built-In Data Types

Lasso automatically translates between XML-RPC data types and Lasso's built-in data types. The *XML-RPC and Built-In Data Types* table provides details about how data types are converted. Since Lasso performs two way conversions XML-RPC calls to a Lasso server can be made without concern for data type conversions.

Table 3: XML-RPC and Built-In Data Types

XML-RPC Data Type	Lasso Equivalent
<i4> or <int>	Integer. XML-RPC supports only 32-bit signed integers.
<double>	Decimal. Double precision floating point number.
<boolean>	Boolean.
<dateTime.iso8601>	Date. Lasso automatically parses and formats XML-RPC date/times.
<string> or <base64>	String. Lasso stores both character and binary data in the string data type.
<struct>	Map. Individual <member> tags become elements of the map with <name> as the key and <value> as the value.
<array>	Array. Each <value> becomes an element of the array.

Note: Lasso supports 64-bit signed integers and greater floating point precision than many XML-RPC servers.

XML-RPC Data Type

Table 4: XML-RPC Data Type

Tag	Description
[XML_RPC]	Creates an XML_RPC object. Accepts an array of parameters for an outgoing XML_RPC call or for an incoming XML_RPC call that is to be processed.

Lasso supports calling a remote procedure through the [XML_RPC] data type. An instance of the [XML_RPC] data type is created with the parameters for the XML-RPC call, then the [XML_RPC->Call] tag is used to initiate the call and retrieve the results.

Table 5: [XML_RPC] Call Tag

Tag	Description
[XML_RPC->Call]	Calls a remote procedure. Requires two parameters. -URI is the location of the remote server. -Method is the name of the method to call on the remote server. The parameters of the request come from the XML_RPC object.

Note: The -URI parameter stands for Universal Resource Identifier.

To call a remote procedure using XML-RPC:

This example calls a remote procedure GetPrice which is available on a remote Lasso server at http://rpc.example.com/RPC.LassoApp. The remote procedure returns the price for a product based on name. The example has three steps: creating the [XML_RPC] object, calling the remote procedure, and interpreting the results.

1 Store the parameters for the remote procedure call in an [XML_RPC] parameter. The GetPrice procedure requires a single parameter which is the name of a product.

```
[Variable: 'MyRPC' = (XML_RPC: 'Widget')]
```

2 Call the GetPrice remote procedure on the desired server. The results are stored in a variable MyResults.

```
[Variable: MyResults= $MyRPC->(Call: -Method='GetPrice',
  -URI='http://rpc.example.com/RPC.LassoApp')]
```

3 Process the results. The first element of the returned array will be the price for the product. Finally, the price is displayed.

```
[Variable: 'Price' = $myResults->(Get: 1)]
$[Variable: 'Price']
```

➜ $35.95

Creating Procedures

See the *Custom Tags* chapter for information about creating XML-RPC procedures that external Web application servers can call.

Processing an Incoming Call

Lasso can processing incoming remote procedure calls in two ways.

- A custom tag can be created using the [Define_Tag] ... [/Define_Tag] tags with the -RPC parameter. The custom tag will be automatically made available through the RPC.LassoApp.
- Any Lasso format file can be used as the target for remote procedure calls. The methods of the [XML_RPC] data type can be used to interpret and process incoming calls.

The use of custom tags is the easiest way to process incoming remote procedure calls. Lasso handles the process of interpreting the method and parameters of each call and automatically returns the results to the caller. All XML-RPC calls are made to a single URL, i.e. http://www.example.com/RPC.LassoApp, making it easy to document what remote procedure calls the server supports.

Note: The creation of custom tags is covered in detail in the *Custom Tags* chapter.

To process remote procedure calls using a custom tag:

This example demonstrates how to create the GetPrice procedure used in the calling example. A custom tag named [GetPrice] will be created which accepts a single parameter, searches the Products table of a Store database, and returns the result. The -RPC parameter in the opening [Define_Tag] tag ensures that this procedure will be available through RPC.LassoApp.

```
[Define_Tag: 'GetPrice', -RPC, -Requires='Product']
  [Inline: -Search,
      -Database='Store',
      -Table='Products',
      'Product'=#Product]
    [Return: (Field: 'Price')]
  [/Inline]
[/Define_Tag]
```

The tag can be called from a remote Lasso Professional 8 server using the [XML-RPC] tags. A call to the GetPrice remote procedure on the server at http://rpc.example.com/ would look like as follows.

```
[Variable: MyResults= (XML_RPC: 'Widget')->(Call: -Method='GetPrice',
  -URI='http://rpc.example.com/RPC.LassoApp')]
[Variable: 'Price' = $myResults->(Get: 1)]
$[Variable: 'Price']
```

➜ $35.95

If more control is required beyond that provided by the built-in XML-RPC processing capabilities of Lasso then a custom format file can be created which processes incoming XML-RPC requests using the method of the [XML_RPC] data type directly.

An incoming XML-RPC request appears as a CGI call with POST parameters. An [XML_RPC] object should be initialized with the array of POST parameters from the [Client_POSTArgs] tag. The method and parameters of the incoming XML-RPC request can then be fetched with the member tags detailed in the *[XML_RPC] Processing Tags* table.

Table 6: [XML_RPC] Processing Tags

Tag	Description
[XML_RPC->GetMethod]	Returns the method for an incoming XML_RPC request.
[XML_RPC->GetParams]	Returns an array of parameters for an incoming XML_RPC request.
[XML_RPC->Response]	Returns a response to an incoming [XML_RPC] request. Accepts two parameters. -Full is either True or False and determines whether full headers should be returned. -Fault is either True or False and determines whether an error response is returned.

To process an incoming XML-RPC request on a custom format file:

There are three steps to process an incoming XML-RPC request. First, the incoming request is parsed and the method and parameters are extracted. Second, the method and parameters are processed. Finally, the results are formatted and returned to the caller.

1 The incoming XML-RPC request is processed by passing [Client_POSTArgs] to the [XML_RPC] tag. The method and parameters of the incoming request are then extracted with the [XML_RPC->GetMethod] and [XML_RPC->GetParams] tags.

```
[Variable: 'myRPC' = (XML_RPC: (Client_POSTArgs))]
[Variable: 'myMethod' = $myRPC->GetMethod]
[Variable: 'myParameters' = $myRPC->GetParameters]
```

2 Since a single format file might process many different XML-RPC methods [Select] ... [Case] ... [/Select] tags are used to determine what code to process.

```
[Select: $myMethod]
  [Case: 'GetPrice']
    [Inline: -Search,
        -Database='Store',
        -Table='Products',
```

```
            'Product'=$MyParameters]
          [Variable: 'Response'=(Field: 'Price')]
        [/Inline]
    [/Select]
```

3 The response is sent back to the caller of the remote procedure by outputting the result of the [XML_RPC->Response] tag with the results of the remote procedure. -Full is set to True so full HTTP headers will be returned to the caller and -Fault is set to False indicating that the XML-RPC call was successful.

```
[Variable: 'myRPC' = (XML_RPC: $Result)]
[ $myRPC->(Response: -Full=True, -Fault=False)]
```

46

Chapter 46
SOAP

This chapter includes information about calling SOAP procedures hosted on remote machines and on hosting SOAP procedures through Lasso.

- *Overview* introduces SOAP procedures.
- *Calling SOAP Procedures* discusses how remote SOAP procedures can be defined as custom tags within Lasso and easily called.
- *Defining SOAP Procedures* discusses how SOAP procedures can be defined within Lasso so remote servers can call them.
- *Low-Level Details* includes information about how SOAP procedures are called and served.

Overview

The Simple Object Access Protocol (SOAP) is a standard which allows remote procedure calls to be made between different servers on the Internet. A remote procedure call is similar to a CGI call (i.e. via [Include_URL]) to a different machine on the Internet, but by passing the parameters of the procedure call and results in a standard XML format, SOAP is more flexible than traditional CGIs.

Lasso's support for SOAP allows remote procedures to be represented locally by automatically created custom tags. These SOAP tags are called just like any tags in LDML, but they do all the work of formatting parameters, accessing the remote machine, and parsing the results automatically.

Similarly, Lasso can provide remote procedures that other machines can call. Remote procedures can be created as custom tags using the -SOAP parameter. Once a tag is defined it will be available as a remote procedure automatically.

This chapter includes information about calling SOAP remote procedures hosted on other machines than Lasso Service, information about creating procedures in Lasso that other machines can call, and low-level details about how Lasso processes both incoming and ougtoing SOAP calls.

Note: The ***XML-RPC*** chapter includes information about how to create and call remote procedures using XML-RPC..

Methodology

Web application servers which provide SOAP procedures that others can call through the Internet usually publish information about the available SOAP procedures in two ways.

- Human readable documentation is provided for manual coders. This documentation usually includes details about what procedures are available, what parameters they take, and what return values they have. This documentation often has examples of the XML SOAP envelopes that are passed back and forth to make a procedure call.
- WSDL is an XML format that describes the available SOAP procedures in a machine readable format. Lasso publishes a WSDL document for the procedures that are made available by Lasso Service at the following URL.

  http://www.exmaple.com/RPC.LassoApp?WSDL

  In order to call remote SOAP procedures Lasso parses the WSDL document from the remote server in order to determine the proper structure for the SOAP envelope.

It may often be necessary to examine both the human readable documentation provided by a remote application server and the raw XML of the WSDL document in order to determine exactly what requirements the remote server has for SOAP calls.

Calling a SOAP Procedure

This section shows the low-level details of how a SOAP procedure is called. The following section *Calling SOAP Procedures* includes instructions for how to use Lasso's built-in SOAP support to making calling SOAP procedures as easy as calling a custom tag.

A SOAP procedure on a rremote server is called by formatting a SOAP envelope using the appropriate XML tags and namespaces. The envelope is then sent as POST data to the remote application server and a result SOAP envelope is parsed.

For example, a SOAP envelope for a procedure that suggests spelling corrections for a word might look as follows (this is adapted from the Google API). The XML is made more complicated by the inclusion of required XML namespace parameters, but the basic structure is a SOAP envelope that includes a SOAP body which defines the remote procedure being called doSpellingSuggestion and includes the parameter for this procedure in <phrase> ... </phrase>.

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV='http://schemas.xmlsoap.org/soap/envelope/'
    xmlns:xsi='http://www.w3.org/1999/XMLSchema-instance'
    xmlns:xsd='http://www.w3.org/1999/XMLSchema'>
  <SOAP-ENV:Body>
    <ns1:doSpellingSuggestion
        xmlns:ns1='urn:GoogleSearch'
        SOAP-ENV:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
      <phrase xsi:type='xsd:string'>bleu wrld</phrase>
    </ns1:doSpellingSuggestion>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This SOAP envelope is sent to the remote server as POST data. This can be accomplished in lasso using [Include_URL] with the -PostData parameter. The response that comes back is itself a SOAP envelope. The following response contains a single spelling suggestion as doSpellingSuggestionResponse in a <return> ... </return> tag.

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:doSpellingSuggestionResponse
        xmlns:ns1="'urn:GoogleSearch'"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <return xsi:type="xsd:string">OmniPilot</return>
    </ns1:doSpellingSuggestionResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

If an error occurs while processing the SOAP procedure then a SOAP fault is sent instead of a successful response. The SOAP fault includes an error code, error string, and error factor.

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
```

```
        <faultcode>-1</faultcode>
        <faultstring>Unknown Error</faultstring>
        <faultactor>Unknown Error</faultactor>
      </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

The built-in tools in *Calling SOAP Procedures* allow most of these details to be hidden. Calling a remote SOAP procedure is made as easy as calling a custom tag. However, it is important to understand how SOAP procedures are being called in order to be able to troubleshoot SOAP communications.

Processing a SOAP Procedure

This section shows the low-level details of how a SOAP procedure is processed. The following section *Defining SOAP Procedures* includes instructions for how to use Lasso's built-in SOAP support to making defining and processing SOAP procedures as easy as creating a custom tag.

A SOAP procedure is processed by parsing the incoming SOAP envelope and SOAP body and determing what procedure is being referenced and what parameters are being passed. The incoming SOAP envelope can be found in [Client_PostParams]. If the procedure is not defined on the Lasso server or invalid parameters are passed then a SOAP fault must be returned.

If the procedure is defined then it is called with the parameters and the results are returned with appropriate formatting in a SOAP envelope with a SOAP body. The results are usually returned using [File_Serve] to serve XML data or by using [Content_Type: 'text/xml'].

See the preceding section for examples of an incoming SOAP envelope, a SOAP response envelope, and a SOAP fault response.

The built-in tools in *Defining SOAP Procedures* allow most of these details to be hidden. Defining and processing a remote SOAP procedure is made as easy as creating a custom tag. However, it is important to understand how SOAP procedures are normally processed in order to be able to troubleshoot SOAP communications.

Calling SOAP Procedures

Remote SOAP procedures are represented in Lasso by SOAP tags which are created using the [SOAP_DefineTag] tag. Once a SOAP tag has been defined it can be called like any other tag in Lasso. This section describes how to create SOAP tags and how to use them to easily call remote procedures.

The information required to create a SOAP tag needs to be collected as follows:

- Use the documentation from the remote Web application server to determine what procedures are available. Note that procedure names are usually case sensitive.
- Check what format the parameters of the desired procedure needs to be in. Are the parameters sent as simple string, integer, or decimal types or do they have XSD types?
- Determine the location of the server's WSDL document. This is usually a URL. For example, the location of a Lasso server's WSDL document is http://www.example.com/RPC.LassoApp?WSDL. The WSDL document tells Lasso how to call the SOAP procedures available on the server.
- How are the results returned from the remote SOAP procedure? A set of tags can be used to automatically parse the results and return just the desired values.

This information is used throughout the rest of this section in order to define a local SOAP tag.

Defining a SOAP Tag

The first step in defining a SOAP tag is to fetch the WSDL document for the server that provides the desired SOAP procedure. Consult the documentation for the SOAP procedure to find out what this URL is. It typically looks something like the following example or http://www.example.com/RPC.LassoApp?WSDL for Lasso servers.

The WSDL document is fetched using [Include_URL] and then parsed using the [XML] tag. The result is stored in a variable so this WSDL document can be used multiple times if necessary.

```
[Var('WSDL' = xml(include_url('http://www.example.com/morse.asmx?WSDL')))]
```

Note: It is sometimes desirable to fetch the WSDL document once and store it locally either by pasting it directly into the Lasso code or by storing it in a local file. Then this local copy of the WSDL can be referenced rather than requiring it to be fetched live each time.

The SOAP tag is defined using [SOAP_DefineTag]. The tag requires four parameters: the name of the tag to be created and the namespace in which to create it, the WSDL document we defined above, and the OperationName (SOAP procedure name) to call. In addition, the tag has two optional parameters a map of default parameters and an array of procedures for processing the return values.

The following example references a procedure MsgtoMorse on the remote server identified by the WSDL fetched above. A tag named [Ex_MsgtoMorse] is created by this tag (Ex_ is the namespace and MsgtoMorse is the tag name). The -Defaults parameter allows a map of default values for parameters to be specified. The -Procs parameter uses an XPath to extract the text of the result. See below for more details about how to process return values.

```
[SOAP_DefineTag(
  -LocalTagName='MsgtoMorse',
  -Namespace='Ex_',
  -WSDL=$WSDL,
  -OperationName='MsgtoMorse',
  -Defaults=(Map: -msg='Default'),
  -Procs=array(proc_extractOne("//text()"))]
```

Now the tag can be called the same as any built-in or custom tag. For example, the following tag call would contact the remote server and return the morse code for the example text. The parameter here is passed as a simple string. Some SOAP procedures require parameters to be passed as a particular pocedure specific type. See below for more details about how to format parameters.

```
[Ex_MsgtoMorse: 'I love Lasso!']

➜ .. / .-.. --- ...- . / .-.. .- ... ... ---
```

SOAP Parameters

The parameters which are passed to the SOAP procedure are determined by three factors. Lasso steps through each parameter which is specified in the WSDL for the specified tag and checks the following locations for the specific parameter:

- If the parameter is specified in the SOAP tag call as a keyword/value then its value is used. For example, in the following tag call a parameter named msg is passed explicitly.

  ```
  [Ex_MsgtoMorse: -msg='The message to be encoded!']
  ```

- If the parameter has a value in the -Defaults map from the [SOAP_DefineTag] call then that value is used. For example, in the above tag definition the -msg parameter is given a value of default. Defaults are

very useful since they allow parameters which do not often change to be hard-coded into the tag definition.

- If the parameter is not found in the SOAP tag call or in the defaults then the next unnamed parameter from the SOAP tag call is used as the parameter value. This allows simple SOAP tags that accept only a single parameter to be called as follows. Note, however, for this to work the default would need to be removed from the tag definition!

  [Ex_MsgtoMorse: 'The message to be encoded!']

If a SOAP procedure does not have all the required parameters or if they are of the wrong type then the SOAP procedure will fail with a SOAP fault. See the following sections for details about how to create parameters using procedure specific types.

Parameter Types

Lasso will automatically cast its built-in data types to appropriate formats for use as SOAP parameters. Procedures that expect string, decimal, integer, or other simple data types can often be called without any special preparations.

However, many SOAP procedures requires that parameters be specified using a specific XML type. Lasso allows these parameters to be created using the [XSD_GenerateType] tag. The [XSD_GenerateType] tag requires two parameters: the name of the namespace for which it is generating a type and the name of the type (or element) that is to be created.

Within a WSDL document a pair of <types> ... </types> tags will define a number of types each named in <element> ... </element> tags. For example, the following WSDL fragment (which has been simplified by removing some namespace identifiers) defines a complex type named MsgtoMorse which contains a single string value.

```
<types>
  <schema targetNamespace="http://www.example.com">
    <element name="MsgtoMorse">
      <complexType>
        <sequence>
          <element minOccurs="0" maxOccurs="1" name="msg" type="string" />
        </sequence>
      </complexType>
    </element>
  </schema>
</types>
```

Lasso can use this information from the WSDL to create a template for the MsgtoMorse data type. The following code creates an instance of MsgtoMorse in the variable $message. The msg part of the created type is then set to a string value.

```
[Var('message' = XSD_GenerateType('http://www.example.com', 'MsgToMorse'))]
[$msg->msg = 'I love Lasso!']
```

Finally, this created parameter can be passed to the [Ex_MsgtoMorse] tag. The tag will format the parameter properly so the remote server can parse the data in the type it expected.

```
[Ex_MsgtoMorse: $msg]
```

The particulars of which types will be available within a given WSDL and whether the remote server will require that parameters be specified using custom types will vary depending on the SOAP procedures being used. It is necessary to consult the documentation of the SOAP procedure and sometimes to view the source of the WSDL document to determine what is required.

Note: The [XSD_GenerateType] tag can only be called after the WSDL for the specified namespace has been loaded by defining a tag using [SOAP_DefineTag] with the appropriate -WSDL parameter.

Processing Return Values

The return value from a SOAP procedure is always a SOAP envelope with a SOAP body or a SOAP fault. By default Lasso automatically trys to translate the returned values from a SOAP procedure to built-in data types, but it is often necessary to use the -Procs parameter to specify an array of processors which can extract the desired values from the SOAP procedure's return value.

The -Procs parameter accepts an array of custom tag references or data types. Each element of the array is invoked in turn. The SOAP body is passed to the first processor as a string. Each subsequent processor is passed the result of the previous processor. This allows complex transformations to be performed on the return value of a SOAP procedure by linking processors.

The following table lists all of the built-in processors including details about how to call each one.

Table 1: Built-In Processors

Processor	Description
Proc_Null	Performs no action. This can be used to override the default behavior. This processor results in the raw XML of the SOAP body being returned. Called as \Proc_Null.
Proc_Extract	Executes [XML->Extract] with an XPath parameter. Called as (Proc_Extract: 'XPath').
Proc_ExtractOne	Executes [XML->ExtractOne] with an XPath parameter. Called as (Proc_ExtractOne: 'XPath').
Proc_Get	Executes ->Get with an integer position parameter. Called as (Proc_Get: #).
Proc_XML	Converts the data to the XML type. Called as \Proc_XML.
Proc_Convert	Converts the data to a built-in Lasso type instance. Called as \Proc_Convert.
Proc_ConvertOne	Converts the first child of XML data to a built-in type instance. Called \Proc_ConvertOne.
Proc_ConvertBody	Converts each child of XML data to a built-in type instance. Returns an array of type instances. This is the default processor. Called as \Proc_ConvertBody.
Proc_XSLT	Executes [XML->Transform] on the data with an XSLT parameter. Called as (Proc_XSLT: 'XSLT').
Proc_RegExp	Executes [String_FindRegExp] on the data with a regular expression parameter. Called as (Proc_RegExp: 'Regular Expression').

Some common processor arrays are included here as examples.

- Extracting the text value of the body can be accomplished using Pro_Extract with an XPath of //text(). This will extract the text value of every sub-element of body.

```
-Procs=Array(Proc_Extract("//text()"))
```

- Extracting the text value of the first element body can be accomplished using Pro_ExtractOne with an XPath of //text(). This will extract the text value of only the first sub-element of body.

```
-Procs=Array(Proc_Extract("//text()"))
```

- The raw XML value of the body can be returned using Proc_Null. This will extract the text value of only the first sub-element of body.

```
-Procs=Array(Proc_Null)
```

Defining SOAP Procedures

Custom tags can be automatically made available to remote servers by specifying the -SOAP parameter when the tag is created. Any tag which is specified as a remote procedure call will be accessible through RPC.LassoApp which is located in the LassoStartup folder. The LassoApp handles all of the translation of parameters and the return value to and from XML.

SOAP tags require that each required and optional parameter be assigned a type using the -Type parameter and that the return type of the tag be specified using the -ReturnType parameter. The parameter and return types are used to automatically translate incoming SOAP requests into appropriate Lasso data types and to properly describe the return value.

When called, remote procedure call tags will be executed using the permissions of the Anonymous user. If the tags require additional permissions a username and password must be written into an [Inline] ... [/Inline] container within the tag or the tag must accept a username and password as parameters.

Tags are called within the context of a page load of the RPC.LassoApp. Tags can access global variables, but will not be able to access any page variables from the page where they were defined. RPC and SOAP tags function essentially as asynchronous tags described elsewhere in this chapter.

To create a SOAP tag:

Use the -SOAP parameter in the opening [Define_Tag] tag. In the following example a method Example.Repeat is created which returns baseString repeated multiplier number of times. Both -Required parameters are followed by -Type parameters and the -ReturnType for the tag is specified.

```
[Define_Tag: 'Example.Repeat', -SOAP,
    -Required='baseString', -Type='string',
    -Required='multiplier', -Type='integer,
    -ReturnType='string']
  [Return: (#baseString * #multiplier)]
[/Define_Tag]
```

The tag can be called from a remote server server that supports SOAP. This tag could be called from within Lasso using [SOAP_DefineTag] to reference the procedure name and automatically generated WSDL file for Lasso Service.

```
[Var: 'WSDL' = XML(Include_URL('http://www.example.com/RPC.LassoApp?WSDL'))]
[SOAP_DefineTag:
    -LocalTagName='Repeat',
    -Namespace='Ex_',
    -WSDL=$WSDL,
    -OperationName='Example.Repeat']
```

Now the SOAP tag [Ex_Repeat] can be called. The request will be routed to Lasso Service and the answer will be returned.

```
[Ex_Repeat: 'String', 4]
```

```
➜ StringStringStringString
```

Of course, the ultimate advantage is that the SOAP procedure can be defined on one Lasso Service machine using [Define_Tag] with the -SOAP parameter and then the procedure can be called as a custom tag from any other Lasso Service machine by creating a SOAP tag with [SOAP_DefineTag].

Low-Level Details

This section details the low-level details of calling a SOAP procedure and responding to a SOAP procedure. It is generally preferable to use the high-level interfaces defined earlier in this chapter if possible, but these implementation details can be useful if one needs to provide compatibility with a server that has a non-standard SOAP implementation or for debugging.

Calling a Remote SOAP Procedure

A remote procedure can be called using the [Include_URL] tag with an appropriatly formated XML SOAP envelope. The SOAP envelope should be included in the documentation of the SOAP service and can be easily modified to include different parameters using Lasso's string tools.

To call a remote SOAP procedure:

This example calls a spell checking procedure that is available as a SOAP service. The input to the SOAP service are the words to be spell checked and the response includes one or more suggestions for proper spellings.

1 Format the SOAP envelope according to the documentation for the service. There is a lot of XML in the envelope, but only one part needs to be customized in order to craft different spell check requests. The words bleu wrld in the <phrase> ... </phrase> tags will be spell checked. Additional spell check requests can be created by changing these words and no other parts of the envelope.

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=\'http://schemas.xmlsoap.org/soap/envelope/\'
    xmlns:xsi=\'http://www.w3.org/1999/XMLSchema-instance\'
    xmlns:xsd=\'http://www.w3.org/1999/XMLSchema\'>
  <SOAP-ENV:Body>
    <ns1:doSpellingSuggestion
        xmlns:ns1=\'urn:GoogleSearch\'
        SOAP-ENV:encodingStyle=\'http://schemas.xmlsoap.org/soap/encoding/\'>
      <phrase xsi:type=\'xsd:string\'>bleu wrld</phrase>
    </ns1:doSpellingSuggestion>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

2 Store the envelope in a variable. Here the envelope is simply placed in a variable. Different SOAP requests can be constructed using search/replace on a template envelope or by appending the start of the envelope, the phrase to be checked, and the end of the envelope together.

```
[Variable: 'SOAP_Envelope' = '<SOAP-ENV:Envelope … ']
```

3 Use [Include_URL] to send the SOAP envelope to the appropriate URL. The SOAP envelope must be sent as a POST like form parameters using the -PostParams parameter of [Include_URL]. The content-type of the request must be set to text/xml using the -SendMIMEHeaders parameter. The result will be stored in a variable.

```
[Variable: 'Result' = (Include_URL: 'http://soap.example.com/SpellCheck/',
    -PostParams=$SOAP_Envelope,
    -SendMIMEHeaders=(Array: 'content-type'='text/xml'))]
```

4 The result will either include a fault code or proper response. First, check to see if it is a fault code and display an appropriate error message. The code below uses the [XML->Extract] tag with an XPath of //faultcode and //faultstring to extract these XML entities if they exist. The [Protect] ... [/Protect] tags will ensure that if there is no fault code the error is suppressed.

```
[Variable: 'XMLResult' = (XML: $Result)]
[Protect]
  [Variable: 'FaultCode' = $XMLResult->(Extract: '//faultcode')->(Get:1)->Contents]
  [Variable: 'FaultString' = $XMLResult->(Extract: '//faultstring')->(Get:1)->Contents]
[/Protect]
[Encode_HTML: $FaultString + ' (' + $FaultCode ')']
```

If an error occurs this will result in an error message like the following. This error indicates that one of the namespace entries in the SOAP envelope is missing.

```
➜ Unable to determine object id from call: is the method element namespaced?
  (SOAP-ENV:Server.BadTargetObjectURI)
```

5 Finally, the results of a successful operation are formatted for output. The results are extracted from the <return> tag using the [XML->Extract] tag with an XPath of //return. The resulting array is looped through to pull the contents out of each return tag and create the final output for the client.

```
[Variable: 'XMLResult' = (XML: $Result)]
[Variable: 'Output' = '']
[Protect]
  [Iterate: $XMLResult->(Extract: '//return'), (Var: 'temp')]
    [$Output += $Temp->Contents + ' ']
  [/Iterate]
[/Protect]
[Encode_HTML: 'Suggestions: ' + $Output]
```

T results of a successful SOAP call are displayed below. The suggested spelling is correct.

```
➜ Suggestions: OmniPilot
```

Note: This example is based on the Google spell checking service, but has been simplified to provide a more concise example.

Processing an Incoming SOAP Call

Lasso can processing incoming SOAP remote procedure calls in two ways.

- A custom tag can be created using the [Define_Tag] ... [/Define_Tag] tags with the -SOAP parameter. The custom tag will be automatically made available through the RPC.LassoApp. All parameters of the tag must have explicit types defined using -Type parameters and the return type of the tag must be defined using the -ReturnType parameter.
- Any Lasso format file can be used as the target for remote procedure calls through the GET method. The URL parameters are interpreted normally and a SOAP envelope is returned as a result.

The use of custom tags is the easiest way to process incoming SOAP remote procedure calls. Lasso handles the process of interpreting the method and parameters of each call and automatically returns the results to the caller. All SOAP calls are made to a single URL, i.e. http://www.example.com/RPC.LassoApp, making it easy to document what remote procedure calls the server supports.

Note: The creation of custom tags is covered in detail in the *Custom Tags* chapter.

To process SOAP remote procedure calls using a custom tag:

This example demonstrates how to create a GetPrice procedure. A custom tag named [GetPrice] will be created which accepts a single parameter, searches the Products table of a Store database, and returns the result.

The -SOAP parameter in the opening [Define_Tag] tag ensures that this procedure will be available through RPC.LassoApp as a SOAP procedure. The -Type parameter is required to specify the type of the Product parameter. The -ReturnType parameter specifies what type the return value of the tag will be.

```
[Define_Tag: 'GetPrice', -SOAP,
    -Requires='Product', -Type='String',
    -ReturnType='String']
  [Inline: -Search,
      -Database='Store',
      -Table='Products',
      'Product'=#Product]
    [Return: (Field: 'Price')]
  [/Inline]
[/Define_Tag]
```

The tag can be called from a remote Lasso Professional 8 server using the techniques documented earlier.

To process an incoming SOAP request in a custom format file:

If more control is required beyond that provided by the built-in XML-RPC processing capabilities of Lasso then a custom format file can be created which processes incoming SOAP requests directly. An incoming SOAP request appears either as a GET request with URL parameters or as a CGI call with a SOAP envelope in the POST parameters.

- SOAP procedures can be called using GET parameters in a URL. These parameters can be processed using [Action_Param] just like any URL parameters. The response to the SOAP procedure should be a properly formated SOAP envelope or SOAP fault code with a content type of text/xml.

  In the following code the GetPrice method is implemented for the following URL. In this example the price for a Widget will be returned.

  ```
  http://www.example.com/SOAP/GetPrice.Lasso?Product=Widget
  ```

  The code of the page performs the database search and returns a SOAP envelope with a <return> tag if the result is good or fault tags if the result is undefined.

  ```
  [Content_Type: 'text/xml']
  [Inline: -Search,
      -Database='Store',
      -Table='Products',
  ```

```
    'Product'=(Action_Param: 'Product')]
  [Variable: 'Response'=(Field: 'Price')]
[/Inline]
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
[If: $Response != '']
  <ns1:GetPrice
      xmlns:ns1="http://www.example.com/SOAP/GetPrice.Lasso"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <return xsi:type="xsd:string">[Variable: 'Response']</return>
  </ns1:doSpellingSuggestionResponse>
[Else]
  <SOAP-ENV:Fault>
    <faultcode>ERROR</faultcode>
    <faultstring>No Price Found</faultstring>
    <faultactor>[Response_FilePath]</faultactor>
  </SOAP-ENV:Fault>
[/If]
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

- SOAP procedures can be called by embedding a SOAP enveloped as the POST parameter in a Web request. The raw POST parameter can be fetched using [Client_PostArgs]. This returns a string that can be fed into the [XML] tag for further processing.

  In the following example this SOAP envelope will be considered a valid request. The product is contained in <product> tags and in this example the price for a Widget should be returned.

  ```
  <SOAP-ENV:Envelope
      xmlns:SOAP-ENV=\'http://schemas.xmlsoap.org/soap/envelope/\'
      xmlns:xsi=\'http://www.w3.org/1999/XMLSchema-instance\'
      xmlns:xsd=\'http://www.w3.org/1999/XMLSchema\'>
  <SOAP-ENV:Body>
    <ns1:GetPrice
        xmlns:ns1s=\'http://www.example.com/SOAP/GetPrice.Lasso\'
        SOAP-ENV:encodingStyle=\'http://schemas.xmlsoap.org/soap/encoding/\'>
      <product xsi:type=\'xsd:string\'>Widget</product >
    </ns1:GetPrice>
  </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
  ```

  The incoming request is parsed using the following code. At the end, the variable Product contains the name of the product whose price should be

returned. The [Protect] ... [/Protect] tags ensure that the absence of <product> tags does not cause a syntax error.

```
[Variable: 'SOAPEnvelope' = (Client_PostArgs)]
[Variable: 'XMLEnvelope' = (XML: $SOAPEnvelope)]
[Varialbe: 'Price' = '']
[Protect]
  [Variable: 'Price' = $XMLEnvelope->(Extract: '//product')->(Get:1)->Contents]
{/Protect]
```

The rest of the page performs the database search and returns a SOAP envelope with a <return> tag if the result is good or fault tags if the result is undefined.

```
[Content_Type: 'text/xml']
[Inline: -Search,
    -Database='Store',
    -Table='Products',
    'Product'=(Variable: 'Product')]
  [Variable: 'Response'=(Field: 'Price')]
[/Inline]
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
[If: $Response != '']
  <ns1:GetPrice
      xmlns:ns1="http://www.example.com/SOAP/GetPrice.Lasso"
      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <return xsi:type="xsd:string">[Variable: 'Response']</return>
  </ns1:doSpellingSuggestionResponse>
[Else]
  <SOAP-ENV:Fault>
    <faultcode>ERROR</faultcode>
    <faultstring>No Price Found</faultstring>
    <faultactor>[Response_FilePath]</faultactor>
  </SOAP-ENV:Fault>
[/If]
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

47

Chapter 47
Wireless Devices

This chapter describes how to create pages in the Wireless Markup Language (WML) which can be served to clients using Wireless Application Protocol (WAP) browsers.

- *Overview* introduces wireless devices.
- *Formatting WML* describes how to specify the MIME type and encode data for wireless browsers.
- *WAP Tags* describes the tags in LDML that allow the characteristics of a WAP client to be returned.
- *WML Example* shows how to create a page which a WAP client can use to search a database and retrieve the results.

Overview

Lasso provides support for serving data to cellular phones and personal digital assistants that support the Wireless Application Protocol (WAP) and the Wireless Markup Language (WML). Serving data to WAP devices (e.g. WAP browsers) is conceptually the same as serving pages to Web browsers, but there are some special considerations that need to be taken into account.

WAP devices require pages to be formatted using the XML-based Wireless Markup Language. Documentation of this language is beyond the scope of this manual. Please consult a book on WAP/WML for more information about how to create pages in WML. Since WML is based on XML, many XML books also contain information about WML.

Lasso does not serve pages to WAP browsers directly. Instead, most WAP browsers communicate with a gateway that contacts Lasso for WML pages and images. The gateway is responsible for performing some manipulation of WML pages and images to ensure they are formatted properly for the WAP browser. A WML-based Web site built using the tags described in this chapter will produce code that is very friendly to the gateway and ensures high fidelity of the site when it is viewed using a WAP browser.

Note: Since WML is based on XML all of the techniques in the following chapter on XML can be used on WML content.

Formatting WML

WAP browsers require pages to be sent using the MIME type of text/vnd.wap.wml and a UTF-8 character set. The [Content_Type] tag can be used to set the MIME type and character set of a page served by Lasso. This tag simply adjusts the header of the page served by Lasso, it does not perform any conversion of the data on the page.

To specify a format file contains WML:

Use the following tag as the very first line of any files which will be served to WAP browsers. Notice that the tag accepts only a single parameter, the charset argument which is appended to the MIME type argument with a semi-colon ;.

```
[Content_Type: 'text/vnd.wap.wml; charset=utf-8']
```

To serve WML:

WML data can be served using the [XML_Serve] tag with the optional -Type parameter set to text/vnd.wap.wml. When the [XML_Serve] tag is used all processing of the current page halts and the parameter of the tag is returned as the contents of the page. This is useful to prevent any stray comments or characters from being sent to WML browsers.

The following example serves some simple WML data in place of the current format file. No tags after the [XML_Serve] tag will be processed.

```
[Variable: 'WMLData' = '<?xml version="1.0" encoding="utf-8" ?>
  <wml>
    <card name="card_one">
      <p><b>Hello WAP user!</b></p>
    </card>
  </wml>']

[XML_Serve: $WMLData, -Type='text/vnd.wap.wml']
```

To format WML:

The data served by Lasso should be formatted using WML. Most WML pages have the following format, an <?XML ... ?> declaration followed by <wml> ... </wml> tags that surround one or more <card> ... </card> tags. The contents of the <card> ... </card> tags are formatted like tiny HTML pages. The following example shows a WML file with a single card.

```
[Content_Type: 'text/vnd.wap.wml; charset=utf-8']
<?xml version="1.0" encoding="utf-8" ?>
<wml>
  <card name="card_one">
    <p><b>Hello WAP user!</b></p>
  </card>
</wml>
```

Most HTML text-formatting tags can be used to format WML pages although the actual set of tags supported may differ from browser to browser. Tables can be used to format data into columns. All tags in WML have an opening and a closing tag. All paragraph tags ... must be closed. A tag which opens and then closes immediately can be written with a slash before the trailing angle bracket,
</br> can be written <br/>.

Every parameter of a tag must have a value. For example, the <input> tag for a check box takes a parameter checked="" rather than the simple checked parameter which HTML allows.

```
<input type="checkbox" name="Field_Name" value="Value" checked="">
```

To specify WML links:

Links can be included using the anchor convention to link to cards within the same document or a different document. The following code would create a link to the card defined above if it were inserted into another card in the same document.

```
<a href="#card_one"> Link to card one </a>
```

If the card defined above was saved in a document named default_wml.lasso then the following link inserted into a card in another document would link directly to it. Both the name of the document and the name of the card are included in the link.

```
<a href="default_wml.lasso#card_one"> Link to card one </a>
```

To specify WML forms:

Forms can be included in WML documents using most of the form input tags. Since WAP browser screens are usually very small, only a few form elements can usually be shown on screen at the same time. Also, since most WAP browsers have limited text capabilities it is often desirable to

place options in <select> ... </select> tags rather than having the client type them in. The following code shows a form that contains a single button. When the form is submitted, the card Card_One in default_wml.lasso is returned as the result.

```
<form action="default_wml.lasso#card_one" method="POST">
  <p><input type="submit" name="-Nothing" value="Submit Form"></p>
</form>
```

To encode data for WML:

The data displayed in WML pages should be XML encoded. The [Encode_Set] ... [/Encode_Set] tags can be used to change the default encoding for all substitution tags in an entire WML page. The following example shows a WML page with an enclosing set of [Encode_Set] ... [/Encode_Set] tags. The value of the [Variable] tag will be XML encoded, ensuring that it displays properly in a WAP browser.

```
[Content_Type: 'text/vnd.wap.wml; charset=utf-8']
<?xml version="1.0" encoding="utf-8">
[Encode_Set: -EncodeXML]
  <wml>
    <card name="card_one">
      <p>[Variable: 'WML_Data']</p>
    </card>
  </wml>
[/Encode_Set]
```

Tags which return XML tags should not have their values encoded. Tags which return XML data require an -EncodeNone encoding keyword in order to ensure that the angle brackets and other markup characters are not encoded into XML entities. The following example shows a variable that returns an entire <card> ... </card>. The [Variable] tag has an -EncodeNone keyword so the angle brackets within the WML data are not encoded.

```
[Content_Type: 'text/vnd.wap.wml; charset=utf-8']
[Variable: 'WML_Data' = '<card name="card_one"><p>Hello WAP user!</card>']
<?xml version="1.0" encoding="utf-8">
[Encode_Set: -EncodeXML]
  <wml>
    [Variable: 'WML_Data', -EncodeNone]
  </wml>
[/Encode_Set]
```

WAP Tags

LDML 8 includes a set of tags that return information about WAP clients. These tags allow a Lasso developer to determine if the current client is using a WAP browser and to determine the size of the screen and how many buttons the browser supports.

The tags are summarized in *Table 1: WAP Tags*. None of the tags return a value if the current client is not using a WAP browser or if the WAP browser does not report the appropriate information in their WAP request. The [WAP_IsEnabled] tag should always be used first to determine if the client is a WAP browser before the other tags are used.

Table 1: WAP Tags

Tag	Description
[WAP_IsEnabled]	Returns True if the current client is using a WAP enabled browser.
[WAP_MaxButtons]	Returns the number of buttons supported by the current client's WAP browser.
[WAP_MaxColumns]	Returns the number of text columns in the screen of the current client's WAP browser.
[WAP_MaxHorzPixels]	Returns the width of the screen in pixels of the current client's WAP browser.
[WAP_MaxRows]	Returns the number of text lines in the screen of the current client's WAP browser.
[WAP_MaxVertPixels]	Returns the height of the screen in pixels of the current client's WAP browser.
[XML_Serve]	Returns WML data in place of the current format file. The first parameter is the WML data to be served. -Type parameter should be set to text/vnd.wap.wml

To display a different page if a client is WAP enabled:

Use the [WAP_IsEnabled] tag to check whether a client is using a WAP browser or not. The following code returns the file default_wml.lasso if the user is using a WAP browser or the file default_html.lasso if they are using a normal Web browser.

```
[If: (WAP_IsEnabled)]
  [Content_Type: 'text/vnd.wap.wml; charset=utf-8']
  [Include: 'default_wml.lasso']
[Else]
  [Include: 'default_html.lasso']
[/If]
```

To choose a graphic based on the size of a WAP browser screen:

Use the [WAP_MaxHorzPixels] and [WAP_MaxVertPixels] tags to determine the size of the client's screen. The following example displays a different graphic if the client's screen is less than 72 pixels in height or width, if it is less than 144 pixels in height or width, or if it is larger.

```
[if: (WAP_MaxHorzPixels) <= 72 || (WAP_MaxVertPixels) <= 72]
  <img src="small_logo.gif" height="36" width="36"/>
[Else: (WAP_MaxHorzPixels) <= 144 || (WAP_MaxVertPixels) <= 144]
  <img src="medium_logo.gif" height="72" width="72"/>
[Else]
  <img src="large_logo.gif" height="144" width="144"/>
[/If]
```

WML Example

The following example shows how to create a page that allows a client to search a database through a WAP browser. The client will be able to search a database named Contacts for either the First_Name or Last_Name and will receive a list of Phone_Numbers in response.

The example is given first in a square bracket version using marked up WML code. The second version uses LassoScript and the [XML_Serve] tag to serve programmatically created WML.

Square Bracket Version

The initial page default.lasso includes a check to see whether the client is using a WAP browser or not. If they are not using a WAP browser then they are forwarded to an error page using the [Redirect_URL] tag.

```
[If: (WAP_IsEnabled) == False]
  [Redirect_URL: 'errror.lasso']
[/If]
```

The remainder of the initial page is a card called form that contains an HTML form which allows the user to search the database for either a First_Name or a Last_Name. When the form is submitted the results card of response.lasso is returned.

```
[Content_Type: 'text/vnd.wap.wml; charset=utf-8']
<?xml version="1.0" encoding="utf-8">
[Encode_Set: -EncodeXML]
  <wml>
    <card name="form">
      <form action="response.lasso#results" method="POST">
          First: <input type="text" name="First_Name" value=""/>
```

```
            <br/>Last: <input type="text" name="Last_Name" value=""/>
            <br/><input type="submit" name="-Nothing" value="Submit"/>
        </form>
      </card>
    </wml>
[/Encode_Set]
```

The results page response.lasso contains an [Inline] ... [/Inline] that performs the actual search. It retrieves the values for First_Name and Last_Name using [Action_Param] tags. The search results are sorted first by Last_Name, then by First_Name. None of the [Field] tags require encoding keywords since the default encoding for the page is set to XML encoding using [Encode_Set] ... [/Encode_Set] tags. An error message is returned if no records are found. A link is provided to return to the search page default.lasso so a new search can be performed.

```
[Content_Type: 'text/vnd.wap.wml; charset=utf-8']
<?xml version="1.0" encoding="utf-8">
[Encode_Set: -EncodeXML]
  <wml>
    <card name="results">
      [Inline: -Database='Contacts',
          -Table='People',
          -KeyField='ID',
          'First_Name' = (Action_Param: 'First_Name'),
          'Last_Name' = (Action_Param: 'Last_Name'),
          -SortField='Last_Name',
          -SortField='First_Name',
          -Search]
        [If: (Found_Count) <= 0]
          <br/>No phone numbers were found.
        [/If]
        [Records]
          <br/>[Field: 'First_Name'] [Field: 'Last_Name'] [Field: 'Phone_Number']
        [/Records]
      [/Inline]
      <br/><a href="default.lasso#form"> Search Again </a>
    </card>
  </wml>
[/Encode_Set]
```

LassoScript Version

The initial page default.lasso includes a check to see whether the client is using a WAP browser or not. If they are not using a WAP browser then they are forwarded to an error page using the [Redirect_URL] tag.

```
<?LassoScript
  If: (WAP_IsEnabled) == False;
    Redirect_URL: 'errror.lasso';
  /If;
?>
```

The remainder of the initial page is a card called form that contains an HTML form which allows the user to search the database for either a First_Name or a Last_Name. When the form is submitted the results card of response.lasso is returned.

```
<?LassoScript
  Variable: 'WML_Content' = <?xml version="1.0" encoding="utf-8">
  $WML_Content += '<wml><card name="form">';
  $WML_Content += '<form action="response.lasso#results" method="POST">';
  $WML_Content += 'First: <input type="text" name="First_Name" value=""/>';
  $WML_Content += '<br/>Last: <input type="text" name="Last_Name" value=""/>';
  $WML_Content += '<br/><input type="submit" name="-Nothing" value="Submit"/>';
  $WML_Content += '</form></card></wml>';

  XML_Serve: $WML_Content, -Type='text/vnd.wap.wml';
?>
```

The results page response.lasso contains an [Inline] ... [/Inline] that performs the actual search. The actual response is collected in the WML_Content variable. The [Field] tags have encoding explicitly set.

```
<?LassoScript
  Variable: 'WML_Content' = <?xml version="1.0" encoding="utf-8">';
  $WML_Content += '<wml><card name="results">';
  Inline: -Search, -Database='Contacts', -Table='People', -KeyField='ID',
      'First_Name' = (Action_Param: 'First_Name'),
      'Last_Name' = (Action_Param: 'Last_Name'),
      -SortField='Last_Name', -SortField='First_Name';
    If: (Found_Count) <= 0;
      $WML_Content += '<br/>No phone numbers were found.';
    /If;
    Records;
      $WML_Content += '<br/>' + (Field: 'First_Name', -EncodeXML) + ' ';
      $WML_Content += (Field: 'Last_Name', -EncodeXML) + ' ';
      $WML_Content += (Field: 'Phone_Number', -EncodeXML);
    /Records;
  /Inline;
  $WML_Content += '<br/><a href="default.lasso#form"> Search Again </a>';
  $WML_Content += '</card></wml>';

  XML_Serve: $WML_Content, -Type='text/vnd.wap.wml';
?>
```

VIII

Section VIII
LDML API

This section includes instructions for extending the functionality of Lasso by creating compiled LassoApps and with custom tags, custom data types, and custom data source connectors written in LDML.

- *Chapter 48: LDML Introduction* includes general information about extending Lasso's functionality.
- *Chapter 49: LassoApps* includes an introduction to LassoApps and instructions for compiling and serving LassoApps.
- *Chapter 50: Custom Tags* discusses how to create new tags in LDML including substitution tags, asynchronous tags, and remote procedures.
- *Chapter 51: Custom Types* discusses how to create new data types in LDML including sub-classing and symbol overloading.
- *Chapter 52: LDML Data Sources* discusses how to create new data sources in LDML.

Lasso can also be extended using C/C++ or Java. See the following sections on the *Lasso C/C++ API* (LCAPI) or *Lasso Java API* (LJAPI) for more information..

48

Chapter 48
LDML Introduction

This chapter presents a road map to the different ways that Lasso can be extended using LDML.

- *Overview* describes the different methods that can be used to extend Lasso and how programming in LDML differs from programming in C/C++ using LCAPI or Java using LJAPI..
- *LassoApps* describes how Lasso solutions can be packaged as compiled LassoApps for easy distribution and installation..
- *Custom Tags* describes how new tags can be created in LDML including looping container tags and asynchronous tags..
- *Custom Types* describes how new data types can be created in LDML including sub-classing built-in types, overloading built-in symbols, and more.
- *LDML Data Sources* describes how new data sources can be created in LDML by impementing a custom data type and registering it with Lasso at startup.

Overview

Lasso provides a number of different mechanisms through which new functionality can be added directly to the language. Lasso has long been extensible through C/C++ and Java APIs, but is now fully extensible using LDML itself to create new functionality.

Deciding which set of APIs or tags to use to extend Lasso depends on the programming skills of the developer and on what functionality is being implemented.

- **LDML** – Extending Lasso in LDML as documented in this section is the easiest for most Lasso developers. It does not require any skills beyond a knowledge of LDML and it does not require any tools beyond Lasso. Using the techniques described in the chapters that follow new tags, data types, and data source connectors can be implemented entirely in LDML.
- **LCAPI** – The Lasso C/C++ Application Programming Interface allows new tags, data types, and data source modules to be created in C/C++. The primary advantages of coding in C/C++ are that the speed of the executed code will be the best possible and that the developer can access many C/C++ code libraries to make implementing some APIs very easy. The drawback is that C/C++ modules must be compiled separately (and may require different support libraries) for each platform that Lasso supports.
- **LJAPI** – The Lasso Java Application Programming Interface allows new tags, data types, and data source modules to be created in Java. The primary advantages of coding in Java are the wide selection of libraries available for the language and the fact that Java modules generally work cross-platform without modification.

LassoApps

LassoApps are entire Lasso solutions that are compiled into a single file. LassoApps can include both LDML pages and images. Since the LassoApp is shipped as a single file it is very easy to install in an end-user's system. And, since LassoApps are compiled the end-user is not able to see the raw source that makes up the solution.

LassoApps can be used in four ways. They can be used in place of LDML pages by placing the LassoApp in the Web server root and calling it like a Web page. LassoApps can be placed in the LassoApps folder for a site and then loaded using a virtual URL. LassoApps can be placed in LassoStartup and used to execute code when Lasso initializes. Finally, LassoApps can be used as tag libraries by placing them in the LassoLibraries folder for a site.

See the following chapter on *LassoApps* for additional information.

Custom Tags

Custom tags are new LDML tags which are written in LDML using the [Define_Tag] ... [/Define_Tag] tags. Custom tags which are defined within an LDML page are immediately available below where they are defined. Custom tags can also be defined in LassoStartup to be made available to all

pages execute in Lasso. Or, custom tags can be defined in a tag library in LassoLibraries to be loaded on-demand.

Custom tags can be used to implement any of the built-in tag types including substitution tags, process tags, container tags, looping container tags, asynchronous process tags, or privileged tags.

See the following chapter on *Custom Tags* for additional information.

Custom Types

Custom types are new data types which are written in LDML using the [Define_Type] ... [/Define_Type] tags. Custom types can inherit from built-in types or from other custom types. Custom types can overload built-in symbols such as + - * / == >> and feature automatic type conversions when required.

See the following chapter on *Custom Types* for additional information.

LDML Data Sources

New data sources can be implemented entirely in LDML. The data sources are implemented as a custom type that defines a special set of member tags. The custom type is registered at start by placing the code which defines it in LassoStartup. The data source is then available in Lasso Administration and in any [Inline] ... [/Inline] tags the same as any built-in data source or data source implemented in C/C++ or Java.

See the following chapter on *LDML Data Sources* for additional information.

49

Chapter 49
LassoApps

This chapter discusses how to develop, build, serve, and administer LassoApps.

- *Overview* describes LassoApps and their benefits for distributing Lasso-based solutions.
- *Administration* explains how to enable LassoApp serving and how LassoApps are cached.
- *Serving LassoApps* explains how LassoApps are served including how to serve LassoApps from the Web server root and the LassoApps folder in the Lasso Professional 8 application folder.
- *Preparing Solutions* documents how to prepare a Lasso-based solution for conversion into a LassoApp.
- *Building LassoApps* explains how to use LassoApp Builder in Lasso Site Administration or the [LassoApp_Create] tag to build a LassoApp.
- *Tips and Techniques* provides helpful information about how to create professional quality LassoApps.

Overview

LassoApps allow entire Lasso-based solutions, including format files and image files, to be packaged into a single archive file with a .LassoApp extension. A compiled LassoApp can be easily distributed and executed on any machine running Lasso Professional 8.

LassoApps offer the following benefits:

- **Performance** – LassoApps are loaded into RAM and cached for efficient serving. All format files within the LassoApp are pre-parsed and served

without additional disk accesses. LassoApp solutions generally provide better performance than their non-LassoApp counterparts.

- **Size** – LassoApps are stored efficiently as a single file. The overhead associated with multiple format and image files is reduced. Redundant data within format files is optimized so only a single copy of duplicate strings is stored.
- **Flexibility** – LassoApps can be served from within the Web server root as normal format files or they can be served from within the LassoApps folder in the Lasso Professional 8 application folder. LassoApps can also be loaded at startup or used as tag libraries..
- **Security** – The code within a LassoApp is stored securely in a byte-compiled form. It is not possible to extract format files and code from a LassoApp.
- **Portability** – A LassoApp is an ideal way to distribute a solution by copying and installing a single file with all internal paths intact. LassoApps are fully cross platform. They can be created on either Mac OS X or Windows 2000 and then deployed on either platform without modifications.

LassoApps are stored in a custom binary file format with a .LassoApp extension. LassoApps can be created programmatically using the [LassoApp_Create] tag or through the LassoApp Builder located in Lasso Site Administration.

LassoApps can be used for any of the following purposes:

- **Packaged Solutions** – LassoApps enable developers to create packaged solutions that can be easily installed by end-users and served by any copy of Lasso Professional 8. LassoApps are placed in the Web serving folder and referenced like a Lasso-based format file or served from the LassoApps folder in the Lasso Professional 8 application folder.
- **Client Solutions** – LassoApps enable developers to deliver solutions to clients in a convenient, secure package. This is ideal so clients can evaluate the functionality of a solution without requiring access to the source code. LassoApps are placed in the Web serving folder and referenced like a Lasso-based format fileor served from the LassoApps folder in the Lasso Professional 8 application folder.
- **Tag Libraries** – LassoApps can define a set of tags in a particular namespace and be installed into the LassoLibraries folder in the Lasso Professional 8 application folder. Lasso will automatically load the LassoApp and all the tags it contains if any of the tags within it are called.
- **Startup Libraries** – LassoApps can be installed into the LassoStartup folder in the Lasso Professional 8 application folder. The

default page of the LassoApp will be executed as a library when Lasso Service starts up and can define custom tags or perform initialization code.

- **Secure Includes** – LassoApps can be included into other format files using the [Include] or [Library] tag. LassoApps can be used to define custom tags or to provide HTML code in a secure manner.

See the sections that follow for information about enabling LassoApps within Lasso Site Administration, preparing an existing solution for compilation as a LassoApp, and detailed instructions about building LassoApps.

Table 1: LassoApp Tags

Tag	Description
[LassoApp_Create]	Creates a LassoApp. Requires three parameters: the -Root of the LassoApp, the -Entry page or default page, and the -Result path where to write the completed LassoApp.
[LassoApp_Dump]	Removes a LassoApp from the cache. Removes a specific LassoApp if a name is specified or all LassoApps if no name is specified.
[LassoApp_Link]	Defines a link to a file within a LassoApp. This tag must be used to mark all links in HTML anchor, form, and image tags and format file references in [Include] and [Library] tags.
-ResponseLassoApp	Returns a specific page from a LassoApp.

Default LassoApps

Lasso Professional 8 relies on LassoApps for all of its administration interfaces, online documentation, and server start-up code. Lasso Professional 8 ships with the following LassoApps.

- **ServerAdmin.LassoApp** – The Lasso Server Administration interface pre-installed in the LassoApps folder within the Lasso Professional 8 application folder. This LassoApp is used to configure Lasso's server-wide preferences and to create sites.
- **SiteAdmin.LassoApp** – The Lasso Site Administration interface pre-installed in the LassoApps folder within the Lasso Professional 8 application folder. This LassoApp is used to configure Lasso Security, to establish the global preferences of a site, to browse existing databases, to monitor the email and event queues, and to create new databases and LassoApps.

- **DatabaseBrowser.LassoApp** – The database browser allows site visitors to browse through any databases that they have permission to access. This LassoApp is pre-installed in the LassoApps folder within the Lasso Professional 8 application folder.
- **GroupAdmin.LassoApp** – The Group Administration interface is pre-installed in the LassoApps folder within the Lasso Professional 8 application folder. This LassoApp allows group administrators to create users and assign them to groups and for users to change their passwords.
- **LDMLReference.LassoApp** – The LDML Reference is the definitive source for information about each tag in Lasso Dynamic Markup Language. This LassoApp is pre-installed in the LassoApps folder within the Lasso Professional 8 application folder.
- **RPC.LassoApp** – This LassoApp responds to incoming remote procedure calls using the XML-RPC format. This LassoApp is pre-installed in the LassoApps folder within the Lasso Professional 8 application folder.
- **Startup.LassoApp** – This LassoApp defines custom tags and performs initialization for Lasso Security, the email sender, and the event queue. This LassoApp is installed in the LassoStartup folder and must be present for Lasso Service to start.

The code for each of these LassoApps can be found within the *Documentation Folder > 2 - Language Guide > LassoApps* folder. This code is provided as-is without any warranty or support.

Warning: Do not compile LassoApps with the same name as the OmniPilot supplied LassoApps (e.g. Startup.LassoApp or SiteAdmin.LassoApp). OmniPilot cannot provide any support for customized versions of these LassoApps or for Lasso Professional 8 installations which make use of customized versions of these LassoApps.

Administration

This section discusses how to enable or disable LassoApp support and how administer the LassoApp cache using LDML tags and within Lasso Site Administration.

Enabling LassoApp Support

Lasso Site Administration includes a global setting to enable or disable LassoApp support. This setting can be found in the *Setup > Global Settings > LassoApps* section of Lasso Site Administration.

When LassoApp support is disabled only the LassoApps which ship with Lasso Professional 8 can be served (including Admin.LassoApp, GroupAdmin.LassoApp, LDMLReference.LassoApp, and Startup.LassoApp in the LassoStartup folder.

Please see the *Site Utilities* chapter of the Lasso Professional 8 Setup Guide for more information about enabling or disabling LassoApp support.

LassoApp Cache

LassoApps are cached in RAM for efficient serving. Each LassoApp only needs to be read from disk once and from then on is served from high-speed memory. LassoApps are read from disk automatically the first time they are called so there is no need to pre-load them (unless the fastest performance is required on the first load).

Since LassoApps are only read from disk the first time they are called it is necessary to ask Lasso to dump any LassoApps that need to be re-read from disk. For example, this is necessary if a new version of a LassoApp is copied into the Web serving folder.

LassoApps can be removed from the cache using the *Cache* page in the *Setup* > *Global Settings* > *LassoApps* section of Lasso Site Administration. See the *Site Utilities* chapter of the Lasso Professional 8 Setup Guide for more information. LassoApps can also be removed from the cache programatically using the following steps.

To remove a LassoApp from the cache:

Use the [LassoApp_Dump] tag with the name of the LassoApp. The following example shows how to remove a LassoApp named MySolution.LassoApp from the cache. The LassoApp will be read from disk the next time the LassoApp is called.

```
[LassoApp_Dump: 'MySolution.LassoApp']
```

To remove all LassoApps from the cache:

Use the [LassoApp_Dump] tag without any parameters The following example shows how to remove all LassoApps from the cache. Each LassoApp will be read from disk the next time it is called.

```
[LassoApp_Dump]
```

To preload a LassoApp into the cache:

LassoApps can be preloaded into the cache by calling them from a Web browser or by using the [Include_URL] tag. The following example shows how to preload a LassoApp named MySolution.LassoApp using [Include_URL].

```
[Include_URL: 'http://www.example.com/Lasso/MySolution.LassoApp']
```

If a LassoApp will be used frequently on the server it can be preloaded using the [Event_Schedule] tag in a format file in LassoStartup. The following code would preload a LassoApp named MySolution.LassoApp five minutes after Lasso Service is started. The delay is specified so the other initialization steps have a chance to complete before the LassoApp is loaded.

```
[Event_Schedule: -URL='http://www.example.com/MySolution.LassoApp',
  -Delay=5]
```

Serving LassoApps

LassoApps can be served the same way as Lasso format files. They can be served from the Web server root or the LassoApps folder in the Lasso Professional 8 application folder, included in other format files, or placed in the LassoStartup folder and executed at startup. This section includes information about how to use LassoApps in each of these situations.

Web Serving Folder

LassoApps which are placed in the Web serving folder are served like any Lasso-based format files. When they are referenced by name in HTML anchor tags, HTML form actions, [Include] or [Library] tags, or as the target of a -Response... tag. The entry page for the LassoApp is always the page that is served.

Since LassoApps are cached, only one copy of each named LassoApp can be served from a single site in Lasso Professional 8. If a second LassoApp with the same name is called the cached copy of the first LassoApp will be served in its place. It is important to ensure that multiple copies of the same LassoApp are identical or unexpected results can occur.

LassoApps Folder

LassoApps which are placed in the LassoApps folder in the Lasso Professional 8 application folder are served when they are referenced by name in HTML anchor tags, HTML form actions, [Include] or [Library] tags, or as the target of a -Response... tag. The entry page for the LassoApp is always the page that is served.

Since LassoApps are cached, only one copy of each named LassoApp can be served from a single site in Lasso Professional 8. If a second LassoApp with the same name is called the cached copy of the first LassoApp will be served in its place. It is important to ensure that multiple copies of the same LassoApp are identical or unexpected results can occur.

LassoApp Links

The links in the entry page must be marked with the [LassoApp_Link] tag in order to reference other files contained within the LassoApp. See the section on *Preparing Solutions* for more details.

The [LassoApp_Link] tag modifies internal links to be of the form LassoAppName.FileNumber.LassoApp. For example, the link to the entry page of a LassoApp named MySolution.LassoApp would be formated as follows in the source of the LassoApp.

```
<a href="[LassoApp_Link: 'default.lasso']"> Entry Page </a>
```

After the LassoApp is compiled, this link will be changed to the following code. The number referenced in the link is determined when the LassoApp is compiled. This number should not be relied on since it may change if the LassoApp is recompiled.

```
<a href="MySolution.0.LassoApp"> Entry Page </a>
```

The conversion of links marked [LassoApp_Link] is handled automatically. No further action beyond marking internal links with the [LassoApp_Link] tag is required. The site visitor will be able to visit any pages which can be reached from the entry page within the LassoApp and will be able to view any linked images within the LassoApp.

To reference pages in a LassoApp from outside the LassoApp:

Individual pages within a LassoApp can be referenced using the -ResponseLassoApp tag as a parameter to the LassoApp name. For example, the entry page (e.g. default.lasso) of the MySolution.LassoApp LassoApp could be referenced explicitly using the following link.

```
<a href="MySolution.LassoApp?-ResponseLassoApp=default.lasso"> Entry Page </a>
```

The path specified for the -ResponseLassoApp tag should be relative to the folder which was compiled into the LassoApp. The -ResponseLassoApp tag should not be used as part of a database action or to specify the response file for a database action. It should only be used to return a specific format file or image file from within a LassoApp.

Note: By using this technique, even files and images within a LassoApp which cannot be reached from the entry page can be viewed if the visitor knows the path to the file they want to view within the LassoApp.

Database Action Responses

The entry page of a LassoApp can be used as the response to a database action by specifying the path to the LassoApp as the parameter for any of

the -Response... command tags. The following form returns the entry file of MySolution.LassoApp as the response to a -FindAll action.

```
<form action="Action.Lasso" method="POST">
  <input type="hidden" name="-FindAll" value="">
  <input type="hidden" name="-Database" value="Contacts">
  <input type="hidden" name="-Table" value="People">
  <input type="hidden" name="-Response" value="MySolution.LassoApp">
  <input type="submit" name="-FindAll" value="Find All People">
</form>
```

Note: The -ResponseLassoApp tag cannot be used in conjunction with a database action to return a particular page from within a LassoApp. Only the entry page of a LassoApp can be returned as the result of a database action.

Lasso Libraries Folder

A LassoApp can define a set of custom tags which are all in the same namespace for on-demand loading. The LassoApp should be named with the same name as the namespace of the tags. For example, a LassoApp named Example.LassoApp could define a set of custom tags in the Example_ namespace [Example_TagOne], [Example_TagTwo], etc.

When Lasso is asked to execute a tag that has not yet been defined it checks in the LassoLibraries folder for a tag library with the name of the namespace of the desired tag. The entry page of a matching LassoApp will be loaded defining all of the tags within.

Note: All of the tags must be defined within the entry page of the LassoApp.

Lasso Startup Folder

The entry page of a LassoApp can be executed when Lasso Service starts up by placing the LassoApp file within the LassoStartup folder inside the Lasso Professional 8 application folder. The entry file can include as many other files within the LassoApp as it needs in order to perform the desired actions. For example, the Startup.LassoApp LassoApp located in the LassoStartup folder executes code which defines a number of custom tags (e.g. [Email_Send], [Include_URL]) in Lasso Professional 8. Because Startup.LassoApp is located in the LassoStartup folder, these custom tags are automatically available upon startup.

Preparing Solutions

Any Lasso-based solution can be compiled into a LassoApp following these preparation instructions. These steps require changes to be made to each format file which needs to link to another file within the LassoApp and requires files that need to remain user customizable to be stored and referenced outside the LassoApp.

The following steps need to be performed to prepare a solution for compilation as a LassoApp.

- The entire solution must be contained in a single folder including all format files and image files which will be compiled into the LassoApp. The folder should only contain text and GIF or JPEG image files.
- The solution must have a single entry point. One file will be loaded when the LassoApp is called, this file must reference other files within the LassoApp either through HTML links, HTML form actions, redirects or [Include] tags.
- All links to files or images within the LassoApp must be marked with the [LassoApp_Link] tag. This tag changes relative paths to a LassoApp specific format.

Preparing Links

The biggest change required to make most solutions ready to be compiled as a LassoApp is to mark all of the links which reference other files within the solution with the [LassoApp_Link] tag. All HTML anchor ... , image <img>, and form <form> ... </form> tags which reference other files within the LassoApp need to be marked as well as [Include] and [Library] tags. The [LassoApp_Link] tag is processed when the solution is compiled into a LassoApp.

Named anchors, links to targets within the same file, mailto links to email addresses, and links to Web sites on other servers do not need to be marked with the [LassoApp_Link] tag.

The [LassoApp_Link] tag can be safely used in any Lasso solution whether it is compiled into a LassoApp or not. When used in a non-compiled solution the [LassoApp_Link] simply returns the specified link value unchanged.

Note: The [LassoApp_Link] tag cannot be used within custom tags or custom data types. Since a custom tag could be called from a different LassoApp than the one in which it is defined (e.g. if a custom tag is defined in the LassoStartup folder, there is no way for Lasso to determine to which LassoApp the [LassoApp_Link] tag should refer. See the end of this section for tips on working with custom tags within LassoApps.

To prepare links to other files within the LassoApp:

- Anchor tags which reference other files within the LassoApp need to be marked with the [LassoApp_Link] tag. The [LassoApp_Link] tag will accept any relative path which is legal within an HTML anchor tag including those which contain ../ to reference files higher in the folder structure. The following example shows an HTML anchor tag that references a file named default.lasso contained in a folder named People.

  ```
  <a href="People/default.lasso"> People Page </a>
  ```

  After being marked with the [LassoApp_Link] tag this anchor tag appears as follows.

  ```
  <a href="[LassoApp_Link: 'People/default.lasso']"> People Page </a>
  ```

  Note: Do not mark named anchors, links to targets within the same file, mailto links to email addresses, or links to Web sites on other servers with the [LassoApp_Link] tag.

- Image tags should be marked with the [LassoApp_Link] tag if the referenced image is contained within the compiled LassoApp. The following example shows an HTML image tag that references a file named boat.gif contained in a folder named Images.

  ```
  <img src="Images/boat.gif" height="288" width="288">
  ```

  After being marked with the [LassoApp_Link] tag this anchor tag appears as follows.

  ```
  <img src="[LassoApp_Link: 'Images/boat.gif']" height="288" width="288">
  ```

- The action parameter for HTML <form> tags should be marked with the [LassoApp_Link] tag if it reference a format file explicitly. The following example shows an HTML <form> tag that references a file named result.lasso which is contained in the same folder as the current page.

  ```
  <form action="result.lasso" method="POST">
    ...
  </form>
  ```

  After being marked with the [LassoApp_Link] tag this HTML <form> tag appears as follows.

  ```
  <form action="[LassoApp_Link: 'result.lasso']" method="POST">
    ...
  </form>
  ```

- If an HTML <form> tag references Action.Lasso as its action then the value parameter for the appropriate <input> tag for the -Response command tag should be marked with the [LassoApp_Link] tag. The following example shows an HTML <form> tag that references Action.Lasso. The response

for the form is specified as response.lasso in a hidden input for the -Response command tag.

```
<form action="Action.Lasso" method="POST">
  <input type="hidden" name="-Response" value="response.lasso">
  ...
</form>
```

After being marked with the [LassoApp_Link] tag the hidden input appears as follows.

```
<form action="Action.Lasso" method="POST">
  <input type="hidden" name="-Response"
    value="[LassoApp_Link: 'response.lasso']">
  ...
</form>
```

- The file parameter for an [Include] or [Library] tag needs to be marked using the [LassoApp_Link] tag. The following examples show an [Include] tag for a file named include.lasso and a [Library] tag for a file library.lasso.

  ```
  [Include: 'include.lasso']

  [Library: 'library.lasso']
  ```

  After being marked with the [LassoApp_Link] tag the tags appear as follows.

  ```
  [Include: (LassoApp_Link: 'include.lasso')]

  [Library: (LassoApp_Link: 'library.lasso')]
  ```

- The response parameter for a [Link_...] tag needs to be marked using the [LassoApp_Link] tag. For example, the [Link_DetailURL] tag accepts a -Response parameter which specifies the format file that should be returned when the link is selected. The following example shows a [Link_DetailURL] tag used within an HTML anchor <a> tag.

  ```
  <a href="[Link_DetailURL: -Response='response.lasso', -Table='People']"> ... </a>
  ```

  After being marked with the [LassoApp_Link] tag, the [Link_DetailURL] tag appears as follows.

  ```
  <a href="[Link_DetailURL: -Response=(LassoApp_Link: 'response.lasso'),
    -Table='People']"> ... </a>
  ```

  Notice that only the name of the response page is marked with the [LassoApp_Link] tag, not the entire href attribute of the anchor tag.

To reference files within a LassoApp from a custom tag:

The [LassoApp_Link] tag cannot be used within custom tags and custom data types. The following techniques can be used to reference files within a LassoApp from custom tags or custom data types.

- References to files can be stored in variables and referenced by variable name within a custom tag. In the following example a reference to a file include.lasso is stored in a variable named IncludeFile. This variable is then referenced within a custom tag.

```
[Variable: 'IncludeFile' = (LassoApp_Link: 'include.lasso')]
…
[Define_Tag: 'myInclude']
  [Return: (Include: $IncludeFile)]
[/Define_Tag]
```

- References to LassoApp files can be passed into custom tags as parameters. In the following example a reference to a file include.lasso is passed as a parameter to a custom tag.

```
[Define_Tag: 'myInclude', -Required='IncludeFile']
  [Return: (Include: #IncludeFile)]
[/Define_Tag]
…
[myInclude: (LassoApp_Link: 'include.lasso')]
```

Building LassoApps

LassoApps can be built programmatically using the [LassoApp_Create] tag or can be built using LassoApp Builder provided in the *Build > LassoApp Builder* section of Lasso Site Administration.

Lasso Site Administration

In order to build a LassoApp using LassoApp Builder, the folder containing the files which will be compiled into the LassoApp must be placed in the Admin/BuildLassoApps folder within the Lasso Professional 8 application folder.

The name of the folder to be converted to a LassoApp is selected from a pop-up menu and the path to the entry file within the folder is specified. Any errors which occur are reported within the interface. If successful, the completed LassoApp is placed within the Admin/BuildLassoApps folder. The completed LassoApp has the same name as the folder that was selected with .LassoApp appended.

See the *Site Administration Utlities* chapter of the Lasso Professional 8 Setup Guide for complete documentation of LassoApp Builder.

To create a LassoApp using LassoApp Builder:

1 Place all of the files which will be compiled into the LassoApp into a single folder. The folder should only contain Lasso format files and image files. All of the format files should have been prepared following the instructions in the *Preparing Solutions* section of this chapter.

For example, place the format files within a folder named MySolution. This folder contains the entry file default.lasso, a folder of included sub-files, and a folder of images.

Note: All of the files within the source folder will be compiled into the LassoApp even if some of the files are never referenced. In order to create the smallest LassoApps possible, any files which are not needed should be removed from the source folder prior to compiling a LassoApp

2 Note the location of the MySolution folder within the Web server root or place the folder MySolution into the Admin/BuildLassoApps folder within the Lasso Professional 8 application folder.

3 Load Lasso Site Administration in a Web browser and go to the *Build > LassoApp Builder* section.

http://www.example.com/SiteAdmin.LassoApp

4 Enter the path to the MySolution folder or Choose MySolution from the pop-up menu and ensure that the entry file is default.lasso. Select the *Create LassoApp* button to create the LassoApp in the Admin/BuildLassoApps folder. Or, select the *Download LassoApp* button to download the created LassoApp through the Web browser.

Note: If the name of the source folder is not present in the pop-up menu select the Refresh button.

5 If any errors are reported, correct them within the format files of the solution and then return to Lasso Site Administration to build the LassoApp again. The LassoApp Builder must complete without any errors in order for a LassoApp file to be created.

6 The completed LassoApp will be in the Admin/BuildLassoApps folder named MySolution.LassoApp or will be downloaded through the Web browser. This file should be copied into the Web serving folder and can then be loaded through a Web browser. If this solution were placed at the root of the Web serving folder it could be loaded through the following URL.

http://www.example.com/MySolution.LassoApp

[LassoApp_Create] Tag

In order to build a LassoApp using the [LassoApp_Create] tag the files which will be compiled into a LassoApp need to be placed in a single folder on the same machine as Lasso Service. The source folder and destination file path for the LassoApp will both be specified using fully qualified, platform-specific paths on the same machine as Lasso Service.

The parameters for the [LassoApp_Create] tag are detailed in *Table 2: [LassoApp_Create] Tag Parameters*. An example of using the tag to create a LassoApp follows. The [LassoApp_Create] tag will return 0 if it is successful creating a LassoApp or an error message otherwise. The tag will replace an existing LassoApp file if the -Result parameter specifies a file that already exists.

Table 2: [LassoApp_Create] Tag Parameters

Parameter	Description
-Root	The folder which contains the files that will be compiled into the LassoApp. Should be specified using a fully qualified, platform-specific path.
-Entry	The default format file within the LassoApp which will be loaded when the LassoApp is called. Should be specified relative to the root folder.
-Result	The destination file name for the created LassoApp. Should be specified using a fully qualified, platform-specific path and must end in the file suffix .LassoApp.

To create a LassoApp using the [LassoApp_Create] tag:

1 Place all of the files which will be compiled into the LassoApp into a single folder. The folder should only contain Lasso format files and image files. All of the format files should have been prepared following the instructions in the *Preparing Solutions* section of this chapter.

This folder contains the entry file default.lasso, a folder of included sub-files, and a folder of images. Determine the platform-specific, fully qualified path to this folder.

For example, if the folder is located at the root of the Web serving folder on a Windows 2000 machine then the root path would be as follows.

```
C:\\InetPub\wwwroot\MySolution\
```

If the folder is located at the root of the Web serving folder on a Mac OS X machine then the root path would be as follows.

```
/Library/WebServer/Documents/MySolution/
```

3 Create a format file which contains the following [LassoApp_Create] tag. This tag will build a LassoApp named MySolution.LassoApp stored at the same location as the root folder defined above. The entry file for the LassoApp will be default.lasso immediately inside the MySolution folder.

The [LassoApp_Create] tag for Windows 2000 would be as follows.

```
[LassoApp_Create: -Root='C:\\InetPub\wwwroot\MySolution\',
  -Entry='default.lasso',
  -Result='C:\\InetPub\wwwroot\MySolution.LassoApp\']
```

The [LassoApp_Create] tag for Mac OS X would be as follows.

```
[LassoApp_Create: -Root='/Library/WebServer/Documents/MySolution/',
  -Entry='default.lasso',
  -Result="/Library/WebServer/Documents/MySolution.LassoApp']
```

4 If any errors are reported, correct them within the format files of the solution and then reload the format file to build the LassoApp again.

5 The completed LassoApp should have been created within the Web serving root and can be loaded through the following URL.

```
http://www.example.com/MySolution.LassoApp
```

Tips and Techniques

This section presents a number of tips and techniques which can make creating professional quality LassoApps easier.

Naming Conventions

LassoApps should be named with the identifier of the company that created the LassoApp followed by the name of the solution. For example, if OmniPilot shipped a phone book LassoApp it could be named OP_PhoneBook.LassoApp. This ensures that the LassoApp name will not conflict with LassoApps created by other companies.

Warning: Do not compile LassoApps with the same name as the OmniPilot supplied LassoApps (e.g. Startup.LassoApp or Admin.LassoApp). OmniPilot cannot provide any warranty or support for customized versions of these LassoApps or for Lasso Professional 8 installations which make use of customized versions of these LassoApps.

Run-Time Errors

Errors which occur when a LassoApp is executing are reported the same way they are for any Lasso format files. It is important to thoroughly test a

LassoApp to ensure that all errors are caught and properly reported to the site visitor. The [Protect] ... [/Protect], [Handle] ... [/Handle] and [Fail] tags can be used to trap for errors and handle them so that the errors are not reported to the site visitor.

Auto-Building Databases

If a LassoApp requires a database table to store solution-specific data it can be created automatically by the LassoApp using the [Database_Create...] tags. Using this technique ensures that a LassoApp can be shipped as a single file and cuts down on the installation required by the end-user.

- LassoApps can safely create tables in the Site database within any installation of Lasso Professional 8. This database is the appropriate place to store both preferences and solution-specific data.
- Tables created in the Site database should follow a naming convention which includes the name of the LassoApp in each table name. For example, a LassoApp named MySolution.LassoApp could create tables named MySolution_Preferences and MySolution_Data. Using a clear naming convention ensures that the global administrator knows why individual tables were created and ensures that different LassoApps do not create tables with the same name.
- If necessary, the LassoApp may need to ask for additional permissions in order to create new tables or to gain access to the tables that have been created. See the section on *Lasso Security* below for more information.
- Always check to make sure that a table does not exist before creating a new table. A LassoApp should never overwrite data in the Site table without explicitly ensuring that the administrator wants to do so.

Lasso Security

LassoApps are executed with the permissions of the current site visitor the same as any Lasso format files. If a LassoApp needs to have access to databases, tables, or tags that can be secured in Lasso Site Administration then it should check that the appropriate permissions are present before executing.

Tags

If a LassoApp requires access to tags which can be secured in Lasso Site Administration such as the [Admin_...] tags, [Database_Create...] tags, [File_...] tags, [Email_Send] or [Event_Schedule] tags, it should first check to be sure those tags are allowed by the current user before executing. The

following code will check to be sure the [Email_Send] tag is available and display an error message if it is not.

```
[If: (Lasso_TagExists: 'Email_Send') == False]
  <br>Error: The tag Email_Send is required in order for this LassoApp to execute.
  Please enable it within Lasso Site Administration before proceeding.
[/If]
```

LassoApps can be created even if the tags they require are not present when they are built and compiled. However, syntax errors will be reported when the LassoApp is served or executed.

Databases and Tables

If a LassoApp requires access to certain databases or tables it should first check to be sure they are available to the current user before executing. The following code will check to be sure the People table of the Contacts database is available.

```
[Inline: -Database='People', -Table='Contacts', -Show]
  [If: (Error_CurretError) != (Error_NoError) || (Field_Name: -Count) == 0]
    <br>Error: The People table of the Contacts database is required
      in order for this LassoApp to execute. Please enable it within Lasso
      Administration before proceeding.
  [/If]
[/Inline]
```

Groups and Users

The [Admin_...] tags can be used to create new users and assign them to a group. These tags are essential if Lasso Security is going to be used to handle multiple user accounts for a LassoApp. Since there is no tag to create a group and assign it permissions, the documentation for a LassoApp solution will need to walk a Lasso global administrator through creating a group with the proper name, assigning permissions, and creating a group administrator.

Lasso Startup

If code needs to be executed when Lasso Service starts up, then a LassoApp can be placed within the LassoStartup folder within the Lasso Professional 8 application folder. Usually, a solution that requires startup code would consist of two LassoApps, one that installs in LassoStartup and a second that defines the user interface for the solution.

50

Chapter 50
Custom Tags

This chapter introduces custom tags and shows how each can be created using LDML tags.

- *Overview* introduces the concepts behind custom tags including naming conventions, namespaces, parameter references, and error reporting.
- *Custom Tags* describes how to create custom tags including information about processing parameters and using local variables.
- *Container Tags* describes how to create custom container tags and looping container tags.
- *Web Services, Remote Procedure Calls, and SOAP* describes how to create tags that function as remote procedure calls through XML-RPC or SOAP and how to call those tags from another server.
- *Asynchronous Tags* describes how to create custom asynchronous process tags and background processes.
- *Overloading Tags* describes how to use criteria to determine which tag will execute and how to redefine built-in LDML tags.
- *Constants* describes how to create constants in LDML.
- *Libraries* describes how to package sets of custom tags for distribution including how to create on-demand tag libraries.

Overview

Lasso Professional 8 allows Web developers to extend Lasso Dynamic Markup Language (LDML) by creating custom tags programmed using LDML tags.

LDML custom tags have the following features:

- Custom tags operate just like built-in LDML substitution tags. They can be used in nested expressions, return data of any data type, and allow the use of encoding keywords.
- Custom process, substitution, or container tags can be created.
- They can be created in any Lasso format file and used instantly.
- They are written in LDML. No programming experience or knowledge of a programming language other than LDML is required.
- They can be collected into libraries of tags which can be loaded into any format file using the [Library] tag.
- Custom tags can be used as the target for remote procedure calls enabling communication between Web servers.
- Existing tags can be redefined.
- Tags can be defined with criteria for when they will run. This allows the same tag name to be used with different parameters and makes it easy to redefine tags for custom purposes.
- They can be defined in a format file or library within the LassoStartup folder, making them available to all pages processed by Lasso.
- Asynchronous tags allow operations to be performed in a separate thread so the current format file is served as fast as possible to the site visitor.

Custom data types can also be created in LDML. See the *Custom Types* chapter for more information.

Possible Uses

Custom tags can be used in any of the following ways:

- To define a new LDML tag that can be called like any built-in LDML tags.
- To reuse a portion of LDML code several times in the same format file.
- To create a macro which allows the same HTML code to be reused several times without being retyped.
- To structure the logic of complex calculations using local variables and tag parameters.
- To redefine and customize existing LDML tags.

- To defer processing of some code until after the visitor has already received the format file.
- To allow remote Web servers to make remote procedure calls to Lasso through XML-RPC.

Naming Conventions

Lasso Professional 8 has support for tag namespaces. All custom tags which are created by a developer should be defined in a namespace unique to the developer. For example, if OmniPilot was providing a custom tag which wrapped code with HTML bold tags it might be placed in the OP_ namespace and named [OP_Bold]. All of the tags in this guide will be defined in the Ex_ namespace meaning Example.

RPC Note: Tags which will be used for XML-RPC are typically named with a group named followed by a method, e.g. group.method.

Parameter References

All values are passed to and from custom tags by reference. This improves the speed and efficiency of custom tags by reducing the number of times that date needs to be copied. Parameter references make tags that perform operations on their parameters possible, but require careful programming in order to avoid unintended side-effects.

Lasso is an object-oriented system and every value in a given format file can be thought of as an object. Variables are simply references to objects and it is possible to have multiple references to the same object.

For example, the [Iterate] ... [/Iterate] tag accepts two parameters. The first is an array of values. The second is a variable that will be set as a reference to each element in the array in turn. The values are not copied out of the array, but the variable points to each value in turn. If the variable modifies the value then that new value is automatically modified in the array as well. This code modifies each element in an array to be uppercase.

```
[Var: 'myArray' = (Array: 'one','two','three')]
[Iterate: $myArray, (Var: 'myItem')]
  [Var: 'myItem' = (String_Uppercase: $myItem)]
[/Iterate]
[$myArray]
```

➜ Array: (ONE), (TWO), (THREE)

Custom tags work similarly. The following rules defined how values are passed to and from custom tags.

- All values passed into a custom tag are passed by reference. References are stored in local variables with the same name as the parameter and in a [Params] array.
- Any modifications of the values in the automatically created local variables or the [Params] array will result in the original values outside the custom tag being modified.
- It is recommended to use a set of uniquely named local variables within the custom tag so as not to interfere with the parameters passed by reference. The values of parameters can be copied into the local variables making their modifications safe.
- Local variables are created new for each custom tag call. References to local variables do not persist from tag call to tag call.
- All values are returned from a custom tag by reference. Normally this will be a reference to a local variable. Since a new set of local variables are created each time a tag is called the return value is safe.
- The return value can also be a reference to one of the input parameters or to a page or global variable. In this case any further modifications to the return value after the custom tag has returned will be reflected in the original value.

These rules are illustrated in the many examples that follow.

Error Reporting

Lasso has a flexible error reporting system which can be used to control the amount of information provided to the site visitor when an error occurs. Since custom tags are self-contained it is often desirable to develop and debug them independent of the site on which they are used.

The [Lasso_ErrorReporting] tag can be used with the -Local keyword to set the error reporting level for the current custom tag. Using this tag the error level can be set to Full while developing a tag in order to see detailed error messages.

```
[Lasso_ErrorReporting: 'Full', -Local]
```

Once the custom tag is debugged and ready for deployment the error reporting level can be adjusted to None in order to effectively suppress any details about the coding of the custom tag from being reported.

```
[Lasso_ErrorReporting: 'None', -Local]
```

See the *Error Controls* chapter in the Language Guide for additional details about the [Lasso_ErrorReporting] tag and other error control tags.

Custom Tags

Custom tags can be created in LDML using the [Define_Tag] ... [/Define_Tag] tags. The following table details the tags that are used to create custom tags. These tags are used to process the parameters of the custom tag and to return values from the custom tag.

Custom substitution and process tags can be created in any format file and will be available immediately. Custom container tags can only be created in the LassoStartup folder. See the section on *Libraries* for information about how to create libraries of tags, load tags in LassoStartup, and create tags which can be used by any format file.

It is not possible to create custom command tags using LDML. Command tags are implemented in data source modules. See the documentation on LCAPI later in this book for more information.

See the *Custom Types* chapter for information about creating custom data types and member tags.

Table 1: Tags For Creating Custom Tags

Tag	Description
[Define_Tag]	Defines a new substitution tag or a new member tag if used within a type definition. Requires a single parameter, the name of the tag to be defined. Other parameters are defined in Table 2: [Define_Tag] Parameters.
[Local]	Sets or retrieves the value of a local variable within a custom tag definition.
[Local_Defined]	Checks to see if a local variable has been defined within a custom tag definition.
[Local_Remove]	Removes a local variable.
[Locals]	Returns a map of all the local variables which have been defined within a custom tag definition.
[Params]	Returns an array of all the parameters which were passed to the custom tag.
[Params_Up]	Returns an array of all the parameters which were passed to the custom tag which called the current custom tag.
[Return]	Returns a value from a custom tag. No further processing is performed.
[Run_Children]	Process the contents of a custom tag created with the -Container option.
[Tag_Name]	Returns the name of the current tag.

The parameters for the [Define_Tag] ... [/Define_Tag] tags are detailed in the following table. The type of tag created, required parameters, return data type, and more are all specified in the opening [Define_Tag] tag.

Table 2: [Define_Tag] Parameters

Tag	Description
'Tag Name'	The name of the tag to be defined. Required.
-Namespace	The name of the namespace in which the tag is defined. Optional. If not specified then tags will be placed in the current namespace.
-Async	Specifies that the tag should be run asynchronously. Asynchronous tags cannot return a value. Optional.
-Container	Specifies that the tag is a container tag. [Run_Children] can be used if this parameter is specified. Optional. See also -Looping for looping container tags.
-Copy	Specifies that the preceding -Required or -Optional parameter should be copied rather than passed by reference.
-Criteria	Specifies the criteria under which the tag will run. If the criteria is not met then the next tag in the calling chain will be used instead. Optional.
-Description	A brief description of the tag. Can include calling instructions, author of the tag, etc. Optional.
-EncodeNone	Specifies that the return value of the tag should not be encoded by default. If this keyword is not specified then the return value will be HTML encoded by default.
-Looping	Specifies that the tag is a looping container tag. [Run_Children] can be used if this parameter is specified. Optional. See also -Container for non-looping container tags.
-Optional	Names an optional parameter of the tag. Optional.
-Priority	Requires the value 'High', 'Low', or 'Replace'. Specifies whether the tag should replace an existing tag with the same name or be placed before or after existing tags in the calling chain. Optional.
-Privileged	Specifies that the custom should run with the privileges of the current user rather than with the privileges of the user who ultimately calls the custom tag.
-Required	Names a required parameter of the tag. If the parameter is not specified then an error will result. Optional.
-ReturnType	Specifies the type of the return value of the tag. If a value of different type is returned then an error is generated.

-RPC	Specifies that the tag should be made available to remote Web servers as a remote procedure call. The tag can then be accessed through RPC.LassoApp.
-SOAP	Specifies that the tag should be made available to remote Web servers as a SOAP operation.The tag can then be accessed through RPC.LassoApp. The -Type and -ReturnType tags must be used to specify parameter and return types.
-Type	Specifies the type for the preceding -Required or -Optional parameter. If the tag is called with a parameter that is not of the proper type then an error is generated.

See the section on *Libraries* for information about how to create libraries of tags, load tags in LassoStartup, and create tags which can be used by any format file.

It is not possible to create custom command tags using LDML. See the *Custom Types* chapter for information about creating custom data types and member tags.

Substitution Tags

A new substitution tag is defined using the [Define_Tag] ... [/Define_Tag] container tag within an enclosed [Return] tag that defines the value of the tag. The opening [Define_Tag] tag requires the name of the new substitution tag to be defined. All of the LDML code between the two tags is stored and is executed each time the tag is called.

In the following example, a tag [Ex_EmailAddress] is defined which returns an example email address for John Doe, johndoe@example.com.

```
[Define_Tag: 'EmailAddress', -Namespace='Ex_']
  [Return: 'johndoe@example.com']
[/Define_Tag]
```

This tag can be called like any LDML substitution tag within the format file where the tag is defined. The following code calls this tag twice, once to provide the address for the HTML anchor tag and a second time to provide the text of the anchor.

```
<a href="[Ex_EmailAddress]"> [Ex_EmailAddress] </a>
➜ <a href="mailto:johndoe@example.com"> johndoe@example.com</a>
```

Process Tags

A new process tag is defined using the [Define_Tag] ... [/Define_Tag] container tags. The opening [Define_Tag] tag requires the name of the new process tag to be defined. All of the LDML code between the two tags is stored and is executed each time the tag is called. Since process tags do not return a value, the body of the tag should not contain a [Return] tag.

In the following example, a tag [Ex_SendEmail] is defined which sends an email to an example email address for John Doe, johndoe@example.com. The tag is defined within a LassoScript.

```
<?LassoScript
  Define_Tag: 'SendEmail', -Namespace='Ex_';
    Email_Send: -Host='mail.example.com',
      -To='johndoe@example.com',
      -From='lasso@example.com',
      -Subject='Sample Email',
      -Body='This email was sent from a custom tag.';
  /Define_Tag;
?>
```

This tag can be called like any LDML process tag within the format file where the tag is defined. The following code calls this [Ex_SendEmail] so an email will be sent to johndoe@example.com each time the page with this code is served by Lasso.

```
[Ex_SendEmail]
```

Privileged Tags

Custom tags normally run with the permissions of the user that calls the custom tag. Using the -Privileged keyword a custom tag will instead run with the permissions of the user who defined the custom tag.

This allows the execution of privileged actions to be written into custom tags. The privileged action can be performed without opening up general permission for performing similar actions to the end-users.

For example, a custom tag which is defined in LassoStartup that has the -Privileged keyword will always execute as the global administrator of the machine. Privileged custom tags can then be used to modify internal security settings or perform other actions that require global administrator permission.

Returning Values

In order for a custom tag to return a value it needs to use the [Return] tag. The parameter for the [Return] tag will be returned as the value of the custom tag and no further processing will be performed. A value of any type can be returned using the [Return] tag including simple decimal or integer numbers, strings, complex maps and arrays, or even custom types.

Custom tags can also return values by setting variables. See the section on *Page Variables* that follows for additional details.

The following custom tag returns a string that informs the site visitor of what day it is. If the current day is January 1st then Happy New Year! is returned. Note that if the conditional returns True then the [Return: 'Happy New Year!'] tag is executed and the tag is exited without executing the second [Return] tag that follows.

```
[Define_Tag: 'Greeting', -Namespace='Ex_']
  [If: (Date_GetDay) == 1 && (Date_GetMonth) == 1]
    [Return: 'Happy New Year!']
  [/If]
  [Return: 'The date is ' + (Server_Date: -Long) + '.']
[/Define_Tag]
```

When executed on any day other than the 1st of January this tag returns the current date.

```
[Ex_Greeting]
```

➜ The date is August 27, 2001.

Encoding

Encoding is handled automatically by Lasso when values are returned from a custom tag. Encoding follows the same rules as for built-in substitution tags. These rules are summarized below.

- If no encoding keyword is specified and the custom tag returns a string value then the tag follows the same rules as built-in substitution tags. The string value will be HTML encoded if it is output to the format file or will have no encoding applied if the tag is used as a sub-tag or in an expression.

  The following custom tag [Ex_String] would have HTML encoding applied.

  ```
  [Ex_String]
  ```

  ➜ Bold Text

  However, if the same tag is used as a sub-tag, no encoding is applied.

  ```
  [Variable: 'myString'=(Ex_String)]
  [Variable: 'myString', -EncodeNone]
  ```

```
➜ <b>Bold Text</b>
```

Note: If the tag is used within [Encode_Set] ... [/Encode_Set] tags then the default encoding which is set in the opening [Encode_Set] tag will be used instead of -EncodeHTML when the tag's value is output directly to a format file.

- If no encoding keyword is specified and the custom tag returns any data type other than string then no encoding is applied and the specified data type is returned.

  The following custom tag [Ex_Array] has no encoding applied since it returns an array.

```
[Ex_Array]

➜ (Array: (<b>Bold Text</b>))
```

  Note: Even if the tag is used within [Encode_Set] ... [/Encode_Set] tag, no encoding will by applied by default unless an explicit encoding keyword is specified.

- If an explicit encoding keyword other than -EncodeNone is specified then the return value from the tag is converted to a string and the specified encoding is applied. Use of an explicit encoding keyword guarantees that the value from the tag will be of data type string.

  The following custom tag [Ex_Array] has explicit HTML encoding applied.

```
[Ex_Array: -EncodeHTML]

➜ (Array: (&lt;b&gt;Bold Text&lt;/b&gt;))
```

  Note: The encoding keyword -EncodeNone instructs Lasso that no encoding is desired for a custom tag. For custom tags which return any data type other than string, -EncodeNone is equivalent to not specifying an encoding keyword.

Parameters

Custom tags can accept any mix of named or unnamed parameters. These parameters can be named using the -Required and -Optional parameters in the opening [Define_Tag] tag. Each parameter is automatically defined as a local variable within the tag. If a required parameter is omitted from a tag call then an error is generated. If an optional parameter is omitted then the local variable corresponding to that parameter will not be defined.

- **Named Parameters** – The -Required and -Optional parameters for a tag can be listed in any order. Each -Required parameter must have a matching keyword/value parameter in the parameters for the tag. Keywords must be preceded by a hyphen.

  The following example defines a tag [Ex_Note] which accepts two parameters. -Message is required and is the message to be displayed. -Font is an optional parameter that changes the font of the displayed message if it is specified, otherwise Arial is used.

```
[Define_Tag: 'Note', -Namespace='Ex_', -Required='Message', -Optional='Font']
  [If: (Local_Defined: 'Font') == False]
    [Local: 'Font' = 'Arial']
  [/If]
  [Return: '<font face="' + #Font + '"> ' + #Message + ' </font>']
[/Define_Tag]
```

  The parameters can be used in any order when the tag is called, but the -Message parameter must be present.

```
[Ex_Note: -Font='Helvetica', -Message='Hello World', -EncodeNone]
➜ <font face="Helvetica"> Hello World </font>

[Ex_Note: -Message="Hello World', -EncodeNone]
➜ <font face="Arial"> Hello World </font>
```

  Note: Extra named parameters passed into a custom tag will also create local variables automatically even if the -Required and -Optional parameters are not used.

- **Unnamed Parameters** – The -Required parameters for a tag should be listed in the order they will be specified in the tag followed by any optional parameters that may be specified. Each unnamed parameter of the tag will be assigned in order to the -Required or -Optional parameter in the corresponding position.

  The tag [Ex_Note] defined above accepts two parameters. The first parameter is required and is assigned the name Message. The second parameter is optional and is assigned the name Font if specified.

  When the tag is called at least one parameter must be specified. If a second parameter is specified it is used as the font for the message, otherwise the default font is used..

```
[Ex_Note: 'Hello World', 'Helvetica', -EncodeNone]
➜ <font face="Helvetica"> Hello World </font>

[Ex_Note: 'Hello World', -EncodeNone]
➜ <font face="Arial"> Hello World </font>
```

- **Combination Parameters** – A combination of named an unnamed parameters can be used. First, all keyword/value parameters are assigned to the -Required or -Optional parameters specified in the opening [Define_Tag] tag. Then, any remaining parameters are assigned in order to any -Required or -Optional parameters that have not yet been assigned values.

  For example, the tag [Ex_Note] defined above is called with one unnamed parameter and one keyword/value -Font parameter. First, the -Font parameter is assigned to the -Optional font parameter. Then, the unnamed parameter is assigned to the -Required message parameter.

  ```
  [Ex_Note: 'Hello World', -Font='Helvetica', -EncodeNone]
  ```

  ➜ Hello World

- **Parameters Types** – The type of each parameter can be specified by including a -Type parameter immediately after the -Required or -Optional parameter. When the tag is called if the specified parameter is not of the proper type then an error will be generated.

  The [Ex_Note] tag can be redefined to require that the -Message parameter be a string.

  ```
  [Define_Tag: 'Note', -Namespace='Ex_', -Required='Message', -Type='String',
      -Optional='Font']
    [If: (Local_Defined: 'Font') == False]
      [Local: 'Font' = 'Arial']
    [/If]
    [Return: '<font face="' + #Font + '"> ' + #Message + ' </font>']
  [/Define_Tag]
  ```

  Now if the tag is called with a decimal value for the -Message parameter an error will be generated..

  ```
  [Ex_Note: -Message=99, -EncodeNone]
  ```

  ➜ Syntax Error

Any tag defined with -Required and -Optional parameters can always be called with a combination of named and unnamed parameters. Documentation for custom tags should always specify how a tag should be called.

Parameters Array

If greater control is required over the parameters which are passed into a tag then the [Params] array can be inspected directly. This array contains one element for each parameter that is passed into a custom tag.

- **Simple Parameters** – Simple parameters are included as single elements within the array. Each parameter has the same data type as the literal or variable which was passed to the tag.
- **Name/Value Parameters** – Name/Value parameters are included as elements of the data type pair within the array. Each part of the pair has the same data type as the literal or variable which was passed to the tag.
- **Keyword Parameters** – Keyword parameters are included as string parameters. They should be distinguished by requiring that all keyword names start with a leading hyphen.
- **Keyword/Value Parameters** – Keyword/Value parameters are included as a pair with a string as the first element and the value as the second element. They should be distinguished by requiring that all keyword names start with a leading hyphen.
- **Encoding Keywords** – Encoding keywords are handled automatically by Lasso. They are not passed to custom tags within the [Params] array. Custom tags do not need to do anything special to take advantage of encoding nor is there any way to disable automatic encoding of returned string values.

The [Params_Up] tag is a special purpose tag that allows inspection of the [Params] array from the custom tag which called the current tag. This tag can only be used if the current tag was called from within a custom tag and can be used to create tags that change their values based on the parameters to the calling tags.

To inspect the parameters of a custom tag:

The [Params] array provides access to all the parameters of the current tag. The following example shows a custom tag [Ex_Echo] that outputs information about all the parameters that were passed to the tag by looping through the [Params] array.

```
[Define_Tag: 'Echo', -Namespace='Ex_']
  [Local: 'Output' = '']
  [Loop: (Params)->Size]
    [Local: 'Temp' = (Params)->(Get: (Loop_Count))]
    [If: #Temp->Type == 'pair']
      [#Output += '<br>Pair: ']
      [#Output += '<br> ' + #Temp->First->Type + ': ' + (#Temp->First)]
      [#Output += '<br> ' + #Temp->First->Type + ': ' + (#Temp->Second)]
    [Else]
      [#Output += '<br>' + #Temp->Type + ': ' + (#Temp)]
    [/If]
  [/Loop]
  [If: (#Output == '')]
```

```
    [#Output = '<br>No Parameters']
  [/If]
  [Return: #Output]
[/Define_Tag]
```

When this tag is called with different parameters the following output is created. Note that keywords are simply strings that start with a hyphen and that the -EncodeNone encoding keyword is not represented in the output.

```
[Ex_Echo: 'One', 'Two='Three', -Four, -Five='Six', -Seven=8, -Nine=1.0, -EncodeNone]

➜ String: One
  Pair:
    String: Two
    String: Three
  String: -Four
  Pair:
    String: -Five
    String: Six
  Pair:
    String: -Seven
    Integer: 8
  Pair:
    String -Nine
    Decimal: 1.0
```

To get the value of a keyword/value parameter:

The following custom tag uses the [Params->Find] tag to retrieve several named keyword/value parameters from the [Params] array. The tag [Ex_Greeting] accepts two parameters: -First which should have the first name of a person as its value and -Last which should have the last name of its person as its value. It returns a greeting to that person.

```
<?LassoScript
  Define_Tag: 'Greeting', -Namespace='Ex_';
    Local: 'First' = Params->(Find: '-First')->(Get: 1);
    Local: 'Last' = Params->(Find: '-Last')->(Get: 1);
    Return: 'Dear ' + #First + ' ' + #Last;
  /Define_Tag;
?>
```

When the tag is called it parses the two defined parameters and ignores all others.

```
[Ex_Greeting: -First='John', -Last='Doe'] ➜ Dear John Doe

[Ex_Greeting: -First='John', -Last='Doe', -Title='Mr.'] ➜ Dear John Doe
```

To get the value of all unnamed parameters:

The [Params] array provides access to all the parameters of the current tag. The following example shows a custom tag [Ex_Concatenate] that concatenates the value of all simple, unnamed parameters together and ignores all name/value and keyword/value parameters.

```
[Define_Tag: 'Concatenate', -Namespace='Ex_']
  [Local: 'Output' = '']
  [Loop: (Params)->Size]
    [Local: 'Temp' = (Params)->(Get: (Loop_Count))]
    [If: #Temp->Type != 'pair']
      [#Output += #Temp]
    [/If]
  [/Loop]
  [Return: #Output]
[/Define_Tag]
```

When this tag is called with different parameters the following output is created. Note that any named parameters are ignored and that the -EncodeNone encoding keyword is not represented in the output.

```
[Ex_Echo: 'One', 'Two='Three', -Four, -Five='Six', -Seven=8, -Nine=1.0, -EncodeNone]
```

➜ One-Four

To get the parameters from the calling tag:

The [Params_Up] tag provides access to the parameters of the calling tag. The following tag [Ex_UnnamedParams] returns an array of all unnamed parameters from the calling tag. This tag could be used to filter the [Params] array so only unnamed parameters remained.

```
[Define_Tag: 'UnnamedParams', -Namespace='Ex_']
  [Local: 'Output' = (Array)]
  [Loop: (Params_Up)->Size]
    [Local: 'Temp' = (Params_Up)->(Get: (Loop_Count))]
    [If: #Temp->Type != 'pair']
      [#Output->(Insert: #Temp)]
    [/If]
  [/Loop]
  [Return: #Output]
[/Define_Tag]
```

The [Ex_UnnamedParams] tag can now be used to rewrite the [Ex_Concatenate] custom tag by looping through the [Ex_UnnamedParams] array rather than through the [Params] array.

```
[Define_Tag: 'Concatenate', -Namespace='Ex_']
  [Local: 'Output' = '']
  [Local: 'Unnamed_Params' = (Ex_UnnamedParams)]
  [Loop: (#Unnamed_Params)->Size]
```

```
    [#Output +=  (#Unnamed_Params)->(Get: (Loop_Count))]
  [/Loop]
  [Return: #Output]
[/Define_Tag]
```

When this tag is called with different parameters the following output is created. Note that any named parameters are ignored and that the -EncodeNone encoding keyword is not represented in the output.

```
[Ex_Echo: 'One', 'Two='Three', -Four, -Five='Six', -Seven=8, -Nine=1.0, -EncodeNone]
```

➜ One-Four

Page Variables

Custom tags can set and retrieve the values of variables which are defined in the current format file. This provides a method of passing additional parameters to custom tags by setting pre-defined variables and a method of passing multiple values out of a custom tag.

Any use of page variables should be considered carefully. Local variables, which are defined in the following section, are usually sufficient for storing data required while executing a tag. If data needs to be stored between executions of a tag then it might be more efficient to create a custom data type. See the following section on *Custom Types* for more information.

If a custom tag must store values in page variables it should precede all variable names with the full name of the custom tag followed by an underscore. For example, the custom tag [Ex_Concatenate] would create variables named Ex_Concatenate_Value, Ex_Concatenate_Output, etc.

Local Variables

Each custom tag can create and manipulate its own set of local variables. These variables are separate from the page variables and are deleted when the custom tag returns. Using local variables ensures that the custom tag does not alter any variables which other custom tags or the page developer is relying on having a certain value.

For example, many developers will use the variable Temp to store temporary values. If a page developer is using the variable Temp and then calls a custom tag which also sets the variable Temp, then the value of the variable will be different than expected.

The solution is for the custom tag author to use a local variable named Temp. The local variable does not interfere with the page variable of the same name and is automatically deleted when the custom tag returns. In

the following example, a custom tag returns the sum of its parameters, storing the calculated value in Temp.

```
<?LassoScript
  Define_Tag: 'Sum', -Namespace='Ex_';
    Local: 'Temp'=0;
    Loop: (Params)->Size;
      Local: 'Temp'=(Local: 'Temp') + (Params)->(Get: Loop_Count);
    /Loop;
    Return: #Temp;
  /Define_Tag;
?>
```

The final reference to the local variable temp is as #Temp. The # symbol works like the $ symbol for page variables, allowing the variable value to be returned using shorthand syntax.

When this tag is called, it does not interfere with the page variable named Temp.

```
[Variable: 'Temp' = 'Important value:']
[Variable: 'Sum' = (Ex_Sum: 1, 2, 3, 4, 5)]
['<br>' + $Temp + ' ' + $Sum + '.']
```

➜
Important value: 15.

Parameter and Return Types

The -Type and -ReturnType parameters can be used to check that the parameters which are being passed to the tag are of the proper type before the tag is executed and that the return value of the tag is the proper type when the tag completes.

The -Type parameter is placed immediately after each -Required or -Optional parameter. The corresponding parameter must be of the specified type when the tag is executed or a syntax error is generated. Using these tags reduces the amount of double checking of types that is required within the body of the tag.

The -ReturnType parameter specifies the type that the returned value of the tag must be. If the tag attempts to return a value of a different type then an error is generated. Using this tag is useful as a double check for a tag that is always expected to return a certain data type. It makes enforcement of the return type explicit rather than relying on the custom tag author to ensure that the return type is always proper.

```
[Define_Tag: 'Ex_Bold', -Namespace='Ex_', -Required='theString', -Type='String',
    -ReturnType='String']
  [Return: '<b>' + #theString + '</b>']
[/Define_Tag]
```

If the [Ex_Bold] tag is called with a number then a syntax error will be returned. The following example first shows a successful call to the tag, then an unsuccessful call.

```
[Ex_Bold: 'Bold Text'] ➜ <b>Bold Text</b>

[Ex_Bold: 123.456] ➜ Syntax Error
```

Criteria

The -Criteria parameter allows custom tags to check certain conditions before any code in the tag is executed. Usually this is used to confirm that the appropriate parameters have been passed to the custom tag. If the criteria fails then a syntax error will be generated.

The -Criteria parameter requires a conditional expression. If the evaluated expression returns False then the tag execution is halted and an error is returned.

The code within the -Criteria are executed as if they were specified within the body of the [Define_Tag] ... [/Define_Tag]. Locals can be used to reference -Required or -Optional parameters and the [Params] array can be inspected. -Criteria can also inspect page variables.

To use criteria to check the parameters of a custom tag:

Specify the -Criteria parameter in the opening [Define_Tag] tag. If the condition in the criteria fails then the tag will not be executed. The following code checks to be sure that the tag's required parameter is a string.

```
[Define_Tag: 'Bold', -Namespace='Ex_', -Required='theString',
     -Criteria=(#theString->Type == 'string')]
  [Return: '<b>' + #theString + '</b>']
[/Define_Tag]
```

If the [Ex_Bold] tag is called with a number then a syntax error will be returned. The following example first shows a successful call to the tag, then an unsuccessful call.

```
[Ex_Bold: 'Bold Text'] ➜ <b>Bold Text</b>

[Ex_Bold: 123.456] ➜ Syntax Error
```

Error Control

Custom tags should use the -Required, -Optional, -Type, -ReturnType, and -Criteria parameters to ensure that the parameters of the tag are of the proper type and that the return value is of the proper type. These tags ensure that Lasso developers are alerted of errors when the page is first executed, rather than encountering obscure runtime errors later.

Errors can be returned from custom tags using the [Error_SetErrorMessage] and [Error_SetErrorCode] tags. A custom tag which is explicitly returning an error code should always return [Error_NoError] if no error occurred or an explicit error message otherwise.

Container Tags

A container tag can be created by specifying either the -Container or -Looping keyword within the opening [Define_Tag] tag. When the tag is used both an opening and a closing tag must be specified or an error will occur. The return value of the tag replaces the entire container tag. The contents of the container tag can be accessed using the [Run_Children] tag.

If the -Looping keyword is used the [Loop_Count] will be automatically changed when the custom tag is called. If the -Container keyword is used then the [Loop_Count] will not be modified by the container tag. This distinction allows both looping and simple container tags to be created.

Note: The output of a container tag is not encoded. This allows HTML to be output from container tags without requiring an -EncodeNone tag.

To create a simple container tag:

The following example creates a simple container tag [Ex_Font] ... [/Ex_Font] that wraps its parameters with an HTML <font> tag. The tag takes three optional parameters -Face, -Size, and -Color which correspond to the parameters of the HTML <font> tag.

```
[Define_Tag: 'Font', -Namespace='Ex_', -Container,
    -Optional='Face', -Optional='Size', -Optional='Color']
  [If: !(Local_Defined: 'Face')][Local: 'Face' = 'Verdana']
  [If: !(Local_Defined: 'Size')][Local: 'Size'= 1]
  [If: !(Local_Defined: 'Color')][Local: 'Color' = 'black']

  [Return: '<font face="' + #face + '" size="' + #size + '" color="' + #color + '"> ' +
    (Run_Children) + ' </font>']
[/Define_Tag]
```

A call to this tag appears like this. The -Face and -Color of the output are specified, but the -Size is left to the default of 1.

```
[Ex_Font: -Face='Helvetica', -Color='red'] My Message [/Ex_Font]

➜ <font face="Helvetica" size="1" color="red"> My Message </font>
```

To use the contents of the container tag multiple times:

The following example creates a tag [Ex_Link] that creates a pair of HTML anchor tags with the contents of the container used as both the URL to be followed and the text of the link. This could be used to automatically create hyperlinks out of URLs contained in text. The tag does not require any parameters.

```
[Define_Tag: 'Link', -Namespace='Ex_', -Container]
  [Return: '<a href="' + (Run_Children) + '"> ' + (Run_Children) + ' </a>']
[/Define_Tag]
```

A call to this tag appears like this. The specified URL is included in the results twice.

```
[Ex_Link] http://www.omnipilot.com [/Ex_Link]

➜ <a href="http://www.omnipilot.com"> http://www.omnipilot.com </a>
```

To create a looping container tag:

The following example creates a tag that loops ten times repeating its contents. The -Looping keyword is used in the [Define_Tag] tag to indicate that this is a looping tag rather than a simple container.

```
[Define_Tag: 'Loop10', -Namespace='Ex_', -Looping]
  [Local: #Output = '']
  [Loop: 10]
    [#Output += Run_Children]
  [/Loop]
  [Return: #Output]
[/Define_Tag]
```

A call to this tag appears like this. The specified contents of the tag is repeated ten times with the [Loop_Count] updated each time..

```
[Ex_Loop10] <br>This is loop [Loop_Count]. [/Ex_Loop10]

➜ <br>This is loop 1.
<br>This is loop 2.
…
<br>This is loop 10.
```

If the -Container keyword rather than the -Looping keyword had been used the tag still would have repeated its contents ten times, but the [Loop_Count] would have returned the same value for each repetition.

Web Services, Remote Procedure Calls, and SOAP

Lasso supports remote procedure calls through the XML-RPC and Simple Object Access Protocol (SOAP) standards. Both types of remote procedure calls allow one server on the Internet to call a function that is located on another server. The parameters of the function call and the results of the function call are transmitted between the servers using XML.

Custom tags can be automatically made available to remote servers by specifying the -RPC or -SOAP parameter when the tag is created. Any tag which is specified as a remote procedure call will be accessible through RPC.LassoApp which is located in the LassoStartup folder. The LassoApp handles all of the translation of parameters and the return value to and from XML.

SOAP tags additionally require that each required and optional parameter be assigned a type using the -Type parameter and that the return type of the tag be specified using the -ReturnType parameter. The parameter and return types are used to automatically translate incoming SOAP requests into appropriate Lasso data types and to properly describe the return value.

When called, remote procedure call tags will be executed using the permissions of the Anonymous user. If the tags require additional permissions a username and password must be written into an [Inline] ... [/Inline] container within the tag or the tag must accept a username and password as parameters.

Tags are called within the context of a page load of the RPC.LassoApp. Tags can access global variables, but will not be able to access any page variables from the page where they were defined. RPC and SOAP tags function essentially as asynchronous tags described elsewhere in this chapter.

Remote procedure calls are well suited to a number of different applications. See the *XML* chapter in the Lasso 8 Language Guide for more information. Some possible applications of remote procedure calls include:

- Serving news stories to remote servers. For example, creating a system where other Web sites can show the latest news stories automatically.
- Performing administrative tasks on remote servers. Tags can be created which perform periodic administrative tasks and then those tasks can be triggered using XML-RPC or SOAP calls.
- Integrating with remote systems that communicate via XML-RPC or SOAP. Both Windows 2000 and Mac OS X have systems for sending XML-RPC or SOAP calls and processing the results.

To create a remote procedure call tag:

Use the -RPC parameter in the opening [Define_Tag] tag. In the following example a method Example.Fortune is created which returns a random message each time it is called. Since the tag will not have access to page variables the array of messages is created inside the tag.

```
[Define_Tag: 'Example.Fortune', -RPC]
  [Local: 'Messages' = (Array: 'You will go on a long boat trip.',
    'You will meet a long lost friend',
    'You will strike it rich in the stock market')]
  [Local: 'Index' = (Math_Random: -Min=1, -Max=(#Messages->Size + 1))]
  [Return: #Messages->(Get: #Index)]
[/Define_Tag]
```

The tag can be called from a remote Lasso 8 server using the [XML-RPC] tags. A call to the Example.Fortune remote procedure on the server at http://www.example.com/ would look like as follows.

```
[Variable: 'Result' = XML_RPC->(Call: -Method='Example.Fortune',
  -URI='http://www.example.com/RPC.LassoApp')]
[Variable: 'Result']
```

The result will be one of the messages from the Messages array.

```
➜ You will meet a long lost friend.
```

To create a remote procedure call tag with complex data types:

The previous example demonstrated how a remote procedure call tag could be created and called using a simple tag which accepted no parameters and returned a string result. Remote procedure calls can be used with any number of parameters including any of Lasso's built-in data types such as array, map, boolean, integer, decimal, etc.

In the following example a method Example.TopStories is created that returns an array of formatted URLs for the top stories from a Web site. An optional -Count parameter allows the number of top stories to be returned to be specified. The top stories are found by finding all records in the Stories table of the News database and sorting the results first by Priority then by DateTime.

```
[Define_Tag: 'Example.TopStories', -Optional='Count']
  [Local: 'Results' = (Array)]
  [If: !(Local_Defined: 'Coun't)]
    [Local: 'Count' = 10]
  [/If]
  [Inline: -Findall,
      -Database='News',
      -Table='Stories',
      -SortField='Priority', -SortOrder='Descending',
      -SortField='DateTime', -SortOrder='Descending',
      -MaxRecords=#Count]
```

```
    [Records]
      [#Results->(Insert: '<a href=\"' + (Field: 'URL') + '\">' + (Field: 'Headline') + '</a>')]
    [/Records]

    [Return: #Results]
  [/Define_Tag]
```

The tag can be called from a remote Lasso 8 server using the [XML-RPC] tags. A call to the Example.TopStories remote procedure on the server at http://www.example.com/ which requests the top 3 stories would look like as follows.

```
[Variable: 'Result' = (XML_RPC: (Array: -Count=3))->(Call:
  -URI='http://www.example.com',
  -Method='Example.TopStories')]
[Variable: 'Result']
```

The result will be an array of the top three stories from the database each formatted as a URL linking to the page which contains the story.

```
➜ (Array: (<a href="http://www.example.com/story.lasso?id=106">Annual Results</a>),
    (<a href="http://www.example.com/story.lasso?id=105">Shareholder Meeting</a>),
    (<a href="http://www.example.com/story.lasso?id=104">Company Picnic!</a>))
```

To create a SOAP tag:

Use the -SOAP parameter in the opening [Define_Tag] tag. In the following example a method Example.Repeat is created which returns baseString repeated multiplier number of times. Both -Required parameters are followed by -Type parameters and the -ReturnType for the tag is specified.

```
[Define_Tag: 'Example.Repeat', -SOAP,
    -Required='baseString', -Type='string',
    -Required='multiplier', -Type='integer,
    -ReturnType='string']
  [Return: (#baseString * #multiplier)]
[/Define_Tag]
```

The tag can be called from a remote server that supports SOAP.

Asynchronous Tags

Asynchronous tags are LDML process tags that are executed in a separate thread from the main portion of the page. Lasso does not have to wait for completion of an asynchronous tag before processing and serving the remainder of the format file in which the tag is called.

Since asynchronous tags usually finish executing after a page has been served to the site visitor they cannot return values or modify the page variables for the format file from which they were called.

Asynchronous tags are usually used in one of the following situations:

- To perform database actions which are a side effect of loading a format file, but the results of which are not required for serving the file to the current site visitor.
- To create a background process that periodically checks for certain conditions and performs a database action or sends an email if that condition is met.

Asynchronous tags can be created using the [Define_Tag] ... [/Define_Tag] tags. Newly defined tags will be available below the point where they are defined in a format file. They can be used as many times as needed.

Note: There is no control over when an asynchronous tag will be executed. Depending on how busy the server is the tag may be executed immediately or may be delayed until after the current page is served to the client. The order of execution of asynchronous tags should never be assumed.

Defining Tags

A new asynchronous tag is defined using the [Define_Tag] ... [/Define_Tag] container tags. The opening [Define_Tag] tag requires the name of the new substitution tag to be defined and the second parameter should be -Async which specifies that the tag should be called asynchronously. All of the LDML code between the two tags is stored and is executed each time the tag is called.

In the following example, a tag [Ex_SendEmail] is defined which sends an email to an example email address for John Doe, johndoe@example.com. The tag is defined within a LassoScript and the second parameter is set to True to ensure that the tag will be called asynchronously.

```
<?LassoScript
  Define_Tag: 'SendEmail', -Namespace='Ex_', -Async;
    Email_Send: -Host='mail.example.com',
      -To='johndoe@example.com',
      -From='lasso@example.com',
      -Subject='Sample Email',
      -Body='This email was sent from a custom tag.';
  /Define_Tag;
?>
```

This tag can be called like any LDML process tag within the format file where the tag is defined. The following code calls this [Ex_SendEmail] so an email will be sent to johndoe@example.com each time the page with this code is loaded in a Web browser.

```
[Ex_SendEmail]
```

The code immediately following this tag is executed immediately without waiting for the tag to complete. The email will be queued for sending shortly after the page is finished executing and is served to the client.

Page Variables

None of the page variables which are defined when an asynchronous tag is called are available within the asynchronous tag. The only variables which are available to a custom asynchronous tag are server-wide global variables. Any values which are going to be used by an asynchronous tag should be set using the [Global] tag.

Calling Custom Tags

Only custom tags which are defined in the LassoStartup folder can be called by an asynchronous tag. Tags which are defined in the same format file as the asynchronous tag definition or call cannot be called by an asynchronous tag.

Custom tags can be defined within the body of an asynchronous tag if needed. These custom tags will be deleted as soon as the asynchronous tag finishes executing.

Background Processes

Asynchronous tags can be used to create background processes that continue to run independent of the visitors to a Lasso-powered Web site. An asynchronous tag will continue executing until the end of the tag body or a [Return] tag is reached. By putting an asynchronous tag into an infinite loop it will continue to run until Lasso Service is quit.

Warning: There is no way to stop an asynchronous tag from executing once it is started. Care should be taken to ensure that any background processes which are implemented with asynchronous tags are well behaved.

The [Sleep] tag can be used to pause execution of an asynchronous tag for a number of milliseconds. The asynchronous tag consumes virtually no resources while it is paused.

Most background processes are started by a format file within the LassoStartup folder. This ensures that the background process runs from when Lasso Service starts up until it is quit.

To create a background process:

Place the following code in a format file within the LassoStartup folder. This code will be executed the next time Lasso Service is started.

Two global variables are created. Since these variables are created in the LassoStartup folder they can be read and set from any page which is executed by Lasso. The first global variable Ex_Background_Pause can be used to pause the background task if it is set to True. The second global variable Ex_Background_Kill can be used to kill the background task if it is set to True.

```
[Global: 'Ex_Background_Pause' = False]
[Global: 'Ex_Background_Kill' = False]
```

These variables are not required to create a background task, but are useful for debugging and to kill a runaway task. By setting the appropriate variable to True in any format file the task can be paused or killed.

The task itself is defined in a [Define_Tag] ... [/Define_Tag]. Notice that the naming convention has the name of the tag which defines the task Ex_Background as the first part of the name of the variables associated with the task. The task contains a while loop that checks the Ex_Background_Kill variable and a conditional that checks the Ex_Background_Pause variable. After each execution, the tag pauses for 15 seconds (15000 milliseconds).

```
[Define_Tag: 'Background', -Namespace='Ex_', -Async]
  [While: (Variable: 'Ex_BackGround_Kill') != True]
    [If: (Variable: 'Ex_Background_Pause') != True)]
      ... Perform Task ...
    [/If]
    [Sleep: 15000]
  [/While]
[/Define_Tag]
```

The task is started by calling the [Ex_Background] tag immediately after it is defined. The task starts executing and does not stop until Lasso Service is quit or the variable Ex_Background_Kill is set to True.

```
[Ex_Background]
```

It is important not to call the [Ex_Background] tag more than once or else multiple instances of the background task will be created.

Background tasks can be made more robust by:

- Adding a variable which is set when the background task is executed so it cannot be executed again.
- Adding variables which control how long the background task sleeps.
- Outputting to the console window with [Log: -Window] ... [/Log] or to a log file in order to track the progress of a background task.

Overloading Tags

Lasso provides the ability to create several versions of a tag each with a criteria that dictates when it should be called. Tag overloading makes several advanced techniques possible.

- **Data Types** – Different tags can be created which operate only when their parameters are of a certain data type. The logic of each tag can be made simpler by removing laborious [Select] ... [Case] ... [/Select] statements.
- **Redefine Existing Tags** – Existing tags can be redefined with a specified criteria. The new version of the tag will be called only when the criteria is met, but the old version of the tag is still available. The source code for the original tag is not needed and even built-in tags can be redefined.
- **Debug Tags** – Tags can be created which output debugging information when a page variable is set appropriately. A page can be debugged and then all status messages can be suppressed by resetting the variable.

Note: Tags must reference the proper namespace in order to overload an existing tag. For example, the [Client_IP] tag is in the Client_ namespace so -Namespace='Client_' must be included in the new tag definition.

When a given tag is called, Lasso will check each tag with the same name in turn until the criteria of one of the tags is met. A tag with no criteria will always execute. All built-in tags will always execute when called.

The -Priority and -Criteria parameters of the [Define_Tag] tag will be discussed followed by examples of how to use those parameters to create systems of overloaded custom tags.

Important: In Lasso Professional 8 many built-in tags which comprise the core of the language can not be overloaded. See the LDML Reference for a complete list of tags that cannot be overloaded.

Priority

The placement of each custom tag in the list of tags in the calling chain can be specified using the -Priority parameter of the [Define_Tag] tag. The following three priorities are available.

- **Replace** – The tag will replace any tags of the same name. Only the newly defined tag will be called when a tag of the given name is called. This allows existing tags to be completely redefined. Aliases and synonyms of the replaced tag will not be redefined.

- **High** – The tag will be placed at the front of the calling chain. The criteria of this tag will be checked first to see if it can be called. If another tag is defined with high priority after this tag then that tag will actually be checked first.
- **Low** – The tag will be placed at the end of the calling chain. The criteria of this tag will be checked only after all other tags have been checked. If another tag is defined with low priority after this tag then that tag will actually be checked last. If the tag is placed after a built-in tag or a custom tag with no criteria then the tag will never be called.

Note: By default, tags have no priority. They must have unique names and will be the sole tag in the calling chain.

To replace a built-in tag:

A built-in tag can be replaced by creating a new tag that has a -Priority of Replace. This technique can be used to redefine a custom tag or to redefine a built-in tag. The definition for the new tag must reference the namespace in which the pre-existing tag is defined.

Note: OmniPilot does not support systems which have built-in tags replaced. It is always advisable to create new tag names rather than redefining existing tags.

For example, the [Form_Param] tag could be redefined so it only retrieved parameters that were sent using the Post method in an HTML form. This is done by inspecting the [Client_PostParams] tag and returning those items from the array that match the parameter to the tag. Note that the proper Form_ namespace must be referenced in order to redefine this built-in tag.

```
<?LassoScript
  Define_Tag: 'Param', -Namespace='Form_', -Priority='Replace', -Require='name';
    Local: 'id_array' = (Client_PostParams)->(Find: #name);
    Local: 'output' = '';
    Iterate: #id_array, (Local: 'id_item');
      #output += (Client_PostParams)->(Get: #id_item)->Second + '\r';
    /Iterate;
    #output->(RemoveTrailing: '\r');
    Return: #output;
  /Define_Tag;
?>
```

This tag can now be used anywhere on a page to get access to the parameters that were passed through a form using the Post method. Since the tag uses the [Client_PostParams] tag it can even be used within nested [Inline] tags.

If this tag is defined on a page then it will replace the [Form_Param] tag only until the end of the page. If this tag is defined in the LassoStartup folder then it will replace the [Form_Param] tag for all users of the site. The [Action_Param] tag is not modified by redefining the [Form_Param] tag even though they are aliases.

Criteria

If a tag has a -Criteria parameter defined then it will only be called when the specified criteria are met. If the criteria are not met then the next tag in the calling chain will be consulted or an error will be generated.

The -Criteria parameter should be a conditional expression that returns True of False. It is called within the environment of the tag being defined and has access to local variables created by the -Required and -Optional parameters and to the [Params] array. The -Criteria parameter can also reference page variables.

Required parameters specified by the -Required tag are checked prior to the -Criteria parameter. If a tag is missing a -Required parameter then a syntax error is returned and no further checking of the tags in the calling chain occurs. -Optional parameters should be used with appropriate an appropriate -Criteria expression to require parameters only on certain tags within a calling chain.

To execute a tag when it is called with a parameter of a given type:

Create the tag with a -Required parameter and a -Criteria expression that checks the type of the local defined by the -Required parameter. The following tag prints a formatted message only when it is called with a string parameter.

```
[Define_Tag: 'Print', -Namespace='Ex_',
    -Priority='High',
    -Required='myParam',
    -Criteria=(#myParam->Type == 'string')]
  [Return: '(String: \'' + #myParam + '\')]
[/Define_Tag]
```

When this tag is called with a string parameter the formatted output is generated, otherwise a syntax error is generated.

```
[Ex_Print: 'Text'] ➜ (String: 'Text')

[Ex_Print: 123.456] ➜ Syntax Error
```

Now, an additional tag can be added with the same name that executes when it is called with a parameter of a different data type. The following version of [Ex_Print] will be called when the parameter is of type decimal.

The -Priority of this tag is set to High ensuring that it is called before the other version of [Ex_Print] in the calling chain.

```
[Define_Tag: 'Print', -Namespace='Ex_',
    -Priority='High',
    -Required='myParam',
    -Criteria=(#myParam->Type == 'decimal')]
  [Return: '(Decimal: ' + #myParam + ')]
[/Define_Tag]
```

When this tag is called with a decimal parameter the formatted output is generated. When it is called with a string parameter the prior version of the tag is used and its formatted output is generated. If the tag is called with a parameter of a different data type then a syntax error is generated.

```
[Ex_Print: 'Text'] ➜ (String: 'Text')

[Ex_Print: 123.456] ➜ (Decimal: 123.456)

[Ex_Print: 123] ➜ Syntax Error
```

Additional tags can be created for each of the built-in data types: arrays, dates, maps, pairs, integers, boolean values, etc.

Rather than returning a syntax error when an unknown data type is specified as a parameter to the tag, a version of the tag can be created that accepts parameters of any type. The following version of [Ex_Print] is used for unknown data types. The -Priority is set to Low ensuring that this version of the tag is checked after all other versions of [Ex_Print] in the calling chain.

```
[Define_Tag: 'Print', -Namespace='Ex_',
    -Priority='Low',
    -Required='myParam']
  [Return: '(Unknown: ' + (String: #myParam) + ')]
[/Define_Tag]
```

When this tag is called with a parameter of type date for which no individual version of the tag has been created the Unknown output is generated.

```
[Ex_Print: (Date)] ➜ (Unknown: 5/15/2002 12:34:56)
```

The real power of this type of system of tags—which are only used when called with a parameter of a certain data type—is that it can be expanded by third parties to include their own custom data types. For example, if a new data type is created that represents currency then a new version of the [Ex_Print] tag could be created as well. The end-user will see that [Ex_Print] now works for the currency data type and doesn't have to be aware of the mechanism which has been used to extend this tag to the additional data type.

Constants

The [Define_Constant] tag allows a constant literal value to be declared in much the same way as a custom tag. A constant value works just like a tag except that when the name of the constant is referenced its value is simply returned.

Lasso defines built-in constants for many commonly used parameter values such as All, Eq, Neq, etc.

Table 3: [Define_Constant] Tag

Tag	Description
[Define_Constant]	Defines a constant. Requires two parameters: the name of the constant to be defined and the value for the constant.

To create a constant:

Use the [Define_Constant] tag. The defined constant can then be referenced as if it were a custom tag that returns the constant value or can be used as a parameter value without quotes. For example, the following code defines a constant MySiteName which is then output.

```
[Define_Constnat: 'MySiteName', 'www.example.com']

Welcome to [MySiteName]! Enjoy your stay!
➜ Welcome to www.example.com! Enjoy your stay!
```

Libraries

Libraries can be used to package custom tags and custom types into a format which is easy for any Lasso developer to incorporate into a Lasso-powered Web site.

The following types of libraries can be created:

- **On-Demand Tag Library** – A set of custom tag and custom type declarations can be stored in a format file or LassoApp and placed in the LassoLibraries folder in the Lasso Professional 8 application folder. The format file or LassoApp should have the same name (before the .Lasso or .LassoApp file suffix) as the namespace of the tags defined within. Subfolders can be used to define nested namespaces.
- **Library Format File** – A set of custom tag and custom type declarations can be stored in a format file and then included in any other Lasso format file using the [Library: 'library.lasso'] tag. This is a good way to create

and use a library file whose defined tags and types will only be needed on a few pages in a site.

- **LassoStartup Format File** – A set of custom tag and custom type declarations can be stored in a format file placed within the LassoStartup folder in the Lasso Professional 8 application folder. After Lasso Service is restarted all tags, types, and page variables which are defined within the format file will be available to all format files which are executed on the server.

51

Chapter 51
Custom Types

This chapter introduces custom data types and shows how they can be created using LDML tags.

- *Overview* introduces the concepts behind custom data types.
- *Custom Types* describes how to create new data types.
- *Member Tags* describes how to create member tags for a custom data type.
- *Prototypes* describes how to use prototypes to increase the speed of custom data types.
- *Callback Tags* describes how to use callback tags to perform instance initialization, how to store custom data in serialized types, and how to process arbitrary member tag names.
- *Symbol Overloading* describes how to use callback tags to overload the assignment, comparison, and mathematical operation symbols for any data type.
- *Inheritance* describes how custom data types can inherit instance variables and member tags from other custom data types or from built-in data types..
- *Libraries* describes how to package sets of custom types for distribution.

Overview

Lasso Professional 8 allows Web developers to extend Lasso Dynamic Markup Language (LDML) by creating custom data types programmed using LDML tags.

LDML custom data types have the following features:

- Tags for custom types operate just like built-in LDML member tags. They can be used in nested expressions, return data of any type, and allow the use of encoding keywords.
- Custom types are fully object-oriented. Custom types can inherit properties from other custom types.
- Custom types can provide support for the LDML comparison symbols and automatic casting.
- They can be created in any Lasso format file and used instantly.
- They are written in LDML. No programming experience or knowledge of a programming language other than LDML is required.
- They can be collected into libraries of tags which can be loaded into any format file using the [Library] tag.
- They can be defined in a format file or library within the LassoStartup folder, making them available to all pages processed by Lasso.

Naming Conventions

Lasso Professional 8 has support for tag namespaces. All custom types which are created by a developer should be defined in a namespace unique to the developer. For example, if OmniPilot was providing a custom type which implemented POP support it might be placed in the OP_ namesapce and named [OP_Pop]. All of the types in this guide will be defined in the Ex_ namespace meaning Example.

The member tags of a custom type do note need a prefix since member tags only need to be unique within each data type. In fact, it is recommended to use the same names as built-in member tags for custom member tags if the functionality is equivalent. For example, a custom data type might implement [Type->Get] and [Type->Size] member tags.

If either a member tag or instance variable of a custom tag starts with an underscore then the tag or variable will be transient in the data type. Transient member tags and instance variables will not be copied when the type is assigned to a different variable or serialized.

Error Reporting

Lasso has a flexible error reporting system which can be used to control the amount of information provided to the site visitor when an error occurs. Since custom types are self-contained it is often desirable to develop and debug them independent of the site on which they are used.

The [Lasso_ErrorReporting] tag can be used with the -Local keyword to set the error reporting level for each custom member tag. Using this tag the error level can be set to Full while developing a tag in order to see detailed error messages.

```
[Lasso_ErrorReporting: 'Full', -Local]
```

Once the custom type is debugged and ready for deployment the error reporting level can be adjusted to None in order to effectively suppress any details about the coding of the custom tag from being reported.

```
[Lasso_ErrorReporting: 'None', -Local]
```

See the *Error Controls* chapter in the Language Guide for additional details about the [Lasso_ErrorReporting] tag and other error control tags.

Custom Types

Custom data types can be created in LDML using the [Define_Type] ... [/Define_Type] tags. Newly defined types will be available below the point where they are defined in a format file.

See the section on *Libraries* for information about how to create libraries of types, load types in LassoStartup, and create types which can be used by any format file.

Table 1: Tags for Creating Custom Data Types

Tag	Description
[Define_Type]	Defines a new data type. Requires a single parameter, the name of the type to be defined. Optional second parameter defines a custom type which should be inherited from. Optional -Namespace parameter defined what namespace the custom type should be placed in. Optional -Prototype parameter specifies that the type should be defined with a prototype.
[Define_Tag]	Defines a new new member tag within a type definition.
[Local]	Sets or retrieves the value of a member variable within a custom type definition.
[Local_Defined]	Checks to see if a member variable has been defined within a custom type definition.
[Locals]	Returns a map of all the member variables which have been defined within a custom type definition.
[Params]	Returns an array of all the parameters which were passed to the custom tag.
[Self]	Returns a reference to the current data type instance.
[Self->Parent]	Returns a reference to the parent type for the current data type instance. For use within custom type declarations with inheritance.

Note: In addition to the listed tags all of the tags which are used for creating custom tags are used when creating member tags.

Defining a Type

A new data type is defined by specifying its name in the opening [Define_Type] tag. The body of the [Define_Type] ... [/Define_Type] tags contains code which will be executed each time a new instance of the data type is created.

For example, a new data type Ex_Dollar could be created which will store dollar amounts. The basic type definition is as follows. Each of the parts of this definition are discussed in more detail in the sections that follow.

```
[Define_Type: 'Dollar', -Namespace='Ex_']
  [Local: 'Amount' = 0]
  [Define_Tag: 'onCreate', -Optional='Amount']
    [If: (Local_Defined: 'Amount')]
      [Self->'Amount' = (Decimal: #Amount)]
      [Self->'Amount'->(SetFormat: -DecimalChar=',', -Precision=2)]
    [/If]
  [/Define_Tag]
```

```
    ... Member Tags ...
[/Define_Type]
```

The [Define_Type] ... [/Define_Type] tags define a tag with the same name as the data type. Each time a new instance of [Ex_Dollar] is created the [Ex_Dollar->onCreate] tag is called to initialize the instance.

The code within [Define_Type] ... [/Define_Type] should be used only to define instance variables and to create member tags. The code within [Ex_Dollar->onCreate] should be used to create a specific instance of the t ype based on the parameters passed to the [Ex_Dollar] tag.

Instance Variables

A data type can contain definitions for local variables within the [Define_Type] ... [/Define_Type] tags. These local variables are called instance variables since their values are stored separately for each instance of the data type which is created.

In the example above, the local variable Amount is created. This variable will store a dollar amount, the current value of the data type. Each time a new instance of [Ex_Dollar] is created, a new instance of the Amount instance variable will be created. For example, the following two lines create two variables each of which stores a value of type Ex_Dollar. Each stores its own independent Amount.

```
[Variable: 'Price' = (Ex_Dollar: 10)]
[Variable: 'Tax' = (Ex_Dollar: 0.93)]
```

Instance variables can be referenced explicitly by name using the member symbol -> with the name of the instance variable. The values for the Amount instance variable can be retrieved from each of the Ex_Dollar amounts defined above using the following code.

```
[(Variable: 'Price')->Amount] ➜ 10.00
[(Variable: 'Tax')->Amount] ➜ 0.93
```

The quotes around the variable name Amount can be omitted if the type does not define a tag with the same name as the member variable. Usually, $Price->'Amount' is equivalent to $Price->Amount.

Transient Variables

Instance variables which start with an underscore are transient variables that will not be copied when the instance is assigned to another variable or serialized. Transient variables should only be used to store static data that

does not need to be propagated to new instances of the data type and does not need to survive being stored in a session and retrieved.

The [Null->onSerialize] and [Null->onDeserialize] callbacks can be used to clean up or close any resources referenced by transient variables and to set them back up when the type is restored. The [Null->onAssign] callback can be used similarly when an instance is copied to another variable.

For example, in the prior example, the Amount variable can be renamed _Amount to make it transient in the data type.

```
[Define_Type: 'Dollar', -Namespace='Ex_']
  [Local: '_Amount' = 0]
  [(Local: '_Amount')->(SetFormat: -DecimalChar=',', -Precision=2)]

  [Define_Tag: 'onCreate', -Optional='Amount']
    [If: (Local_Defined: 'Amount')]
      [Self->'_Amount' = (Decimal: #Amount)]
      [Self->'_Amount'->(SetFormat: -DecimalChar=',', -Precision=2)]
    [/If]
  [/Define_Tag]

  … Member Tags …

[/Define_Type]
```

Member Tags

Built-In Member Tags

Each custom type can automatically make use of any of the tags of the null data type. These tags are detailed in *Table 2: Built-In Tags*. These tags are used by Lasso to provide information about data of any type and to provide efficient storage for custom data types. None of the tags in this table should be overriden by the custom data type.

In addition to these built-in member tags there are several tags that are defined as placeholders on the null data type. The [Null->Size] and [Null->SetFormat] tags are defined for every data type. [Null->Size] always returns 0 and [Null->SetFormat] is a placeholder that returns no value if called. Either of these tags can be overridden using custom member tags.

Note: It is desirable for custom data types to create custom [Type->Get] and [Type->Size] member tags so the [Iterate] ... [/Iterate] tags will function properly.

Table 2: Built-In Member Tags

Tag	Description
[Null->DetachReference]	Detaches the variable from the instance of the data type.
[Null->FreezeType]	Freezes the type of a variable. After calling this tag the current variable cannot be cast to another data type.
[Null->FreezeValue]	Freezes the value of a variable, essentially creating a read-only variable. After calling this tag the current variable cannot have its value changed.
[Null->Invoke]	This member tag normally calls the creator of the data type, but can be overridden to provide type specific behavior.
[Null->IsA]	Accepts a single parameter which is the name of a type. Returns True if the parameter matches the name of the current data type or any of its parent data types.
[Null->Parent]	Returns a references to the parent type of the current data type instance.
[Null->Properties]	Returns a pair which contains a map of all the instance variables and a map of all the member tags defined for the data type.
[Null->RefCount]	Returns the number of variables that currently point at the instance of the data type.
[Null->Serialize]	Returns a bit-stream representation for the data type. This tag can be used to store a custom data type in a database or to pass it from page to page as an action parameter.
[Null->Type]	Returns the type which was specified when the custom type was created.
[Null->Unserialize]	Accepts a single parameter which is a bit-stream created by [Null->Serialize]. This tag modifies the variable on which it is called by setting it to the custom data type represented by the bit-stream parameter.

See the Lasso 8 Language Guide for more information about using these tags.

Custom Member Tags

Each custom type can define member tags which can be called to modify the value stored in an instance of the custom type or to output values from an instance of the custom type.

Member tags are defined within the [Define_Type] ... [/Define_Type] tags for the custom type using [Define_Tag] ... [/Define_Tag] tags. The syntax for creating member tags is the same as that for creating custom tags. However,

member tags cannot be called asynchronously. The [Define_Tag] tag for a member tag should never have -Async as the value for the second parameter.

The [Self] tag allows member tags to reference the current instance of the data type. This allows member tags to call other defined member tags or to set or retrieve values stored in instance variables. See the example of defining a custom member tag below for more information. The [Self] tag also allows access to instance variables that are stored within the custom data type.

Custom member tags which are named starting with an underscore are transient member tags. These tags will not be copied when the data type instance is copied to another variable or serialized. The [Null->onAssign] and [Null->onDeserialize] callbacks can be used to redefine transient member tags.

To define custom member tags:

Two custom tags will be defined for the Ex_Dollar custom type. The [Ex_Dollar->Set] tag will accept a single parameter, cast it to decimal, and store it in the Amount instance variable. The [Ex_Dollar->Get] tag will simply return the value of the Amount instance variable formatted as a dollar amount.

- The [Ex_Dollar->Set] member tag is defined within the body of the [Define_Type] ... [/Define_Type] tags. It checks that there is at least one parameter in the [Params] array. The [Self] tag is a reference to the current instance of the Ex_Dollar data type, so the (Self->'Amount') statement is a reference to the Amount instance variable.

```
<?LassoScript
  Define_Tag: 'Set';
    If: (Params) && ((Params)->Size > 0);
      (Self->'Amount') = (Decimal: (Params)->(Get:1));
    /If;
  /Define_Tag;
?>
```

- The [Ex_Dollar->Get] member tag is defined within the body of the [Define_Type] ... [/Define_Type] tags. It appends a dollar sign $ to the value in the Amount instance variable and returns the value. The [Self] tag is a reference to the current instance of the Ex_Dollar data type, so the (Self->'Amount') statement is a reference to the Amount instance variable.

```
<?LassoScript
  Define_Tag: 'Get';
    Return: '$' + (Self->'Amount');
  /Define_Tag;
?>
```

To call a custom member tag:

Custom member tags are called in the same way that the member tags of the built-in data types are called. The Ex_Dollar type has two member tags [Ex_Dollar->Get] and [Ex_Dollar->Set]. They are used to set and retrieve dollar amounts in the following example.

```
[Variable: 'Price' = (Ex_Dollar: 100)]
<br>[(Variable: 'Price')->Get]
[(Variable: 'Price')->(Set: 19.95)]
<br>[(Variable: 'Price')->Get]

➜ <br>$100.00
<br>$19.95
```

Prototypes

Lasso can use type prototypes to dramatically increase the performance of custom types. When Lasso creates a new instance of a type it normally runs all of the code within the [Define_Type] ... [/Define_Type] tags in order to create instance variables and member tags. When the -Prototype keyword is used Lasso runs the code with the type definition once and stores a reference to the pre-compiled prototype. This prototype is copied each time an instance of the type is created.

Any data type which follows these guidelines can be used as a prototype. It is recommended that all custom data types be created as prototypes.

- The custom type definition must only contain [Local] tags to define instance variables and [Define_Tag] ... [/Define_Tag] tags to define member tags within the [Define_Type] ... [/Define_Type] tags.
- The -Prototype keyword must be referenced in the opening [Define_Type] tag.

In addition, it is possible to use the [Define_Prototype] tag to create a prototype out of any data type. This tag takes two parameters: the name of the desired prototype and a reference to a data type which will be used as the prototype. The data type is copied into the tag map as a prototype. Any time the prototype name is referenced a copy of the prototype will be made.

Table 3: Prototype Tag

Tag	Description
[Define_Prototype]	Installs a prototype in the tag map. Requires two parameters: the name by which the prototype will be referenced (the tag name) and a reference to the data type that will be copied as the prototype.

For example a map could be created that had some pre-defined values which would be used over and over again. This map can be installed as a prototype and then referenced as if it were a custom type.

```
[Var: 'Prototype_Map' = (Map: 'First_Name' = '', 'Last_Name'='')]
[Define_Prototype: 'Ex_Person', $Prototype_Map]

[Var: 'myMap' = (Ex_Person)]
```

Callback Tags

Each custom type can define a number of callback tags using the [Define_Tag] ... [/Define_Tag] tags within the [Define_Type] ... [/Define_Type] definition for the type. These callback tags will be executed with appropriate parameters when the data type is cast to another type, a new instance is created, or an instance is destroyed.

Table 3: Callback Tags details the tags that are available. These tag names are reserved. No member tags with these names should be defined. These tags are not normally called by a Lasso developer, they are called automatically by Lasso in the specified situation. Although there is no protection to prevent a Lasso developer from calling these tags directly, results should be considered undefined if they do.

The primary callback tags are shown in *Table 3: Callback Tags*. Additional callback tags allow the overriding of built-in symbols. These tags are described in the next section.

Table 4: Callback Tags

Tag	Description
[Null->onConvert]	Called when the instance is cast to a built-in data type. Accepts a single parameter, the name of the type to which the value should be converted (string, integer, or decimal). The return value should be the converted value or Null if no conversion was possible.
[Null->onCreate]	Called immediately after a new instance is created. This tag has full access to the variables and member tags defined within the [Define_Type] … [/Define_Type] tags.
[Null->onDestroy]	Called before the custom type is destroyed, usually at the end of the current format file or tag execution.
[Null->onSerialize]	Called when the custom type is serialized. The return value is stored with the serialized data and passed to [Null->onDeserialize].
[Null->onDeserialize]	Called when the custom type is deserialized. The return value of [Null->onSerialize] is passed as the first parameter to this tag.
[Null->_UnknownTag]	Called when an unknown member tag is referenced. The [Tag_Name] tag can be used to decide what tag name was referenced.

Note: These callback tags are not included in the LDML 8 tag list. They are intended to be called by Lasso automatically rather than being called like other member tags.

onCreate Callback

The [Null->onCreate] callback tag is called after a new instance of a type is created. It is called once for each instance of a type with any parameters that were passed to the tag that created the type.

For example, when the tag [Ex_Dollar] is used to create a new instance of the dollar type the following steps are performed.

1. The code within the [Define_Type] ... [/Define_Type] container is executed, creating all the custom tags and instance variables for the type.
2. The [Ex_Dollar->onCreate] tag is called with the parameters passed to the [Ex_Dollar] tag to set up the particular instance of the type.

Since the callback tag is called after the code within the [Define_Type] ... [/Define_Type] container is processed, the [Null->onCreate] tag has access to the [Self] tag and to each of the member tags which have been defined for the current type.

Order of operation:

A new instance of a custom type is created by calling the creator tag for the type which has the same name as the type. For example, to create a new Ex_Dollar type the [Ex_Dollar] tag must be called.

```
[Ex_Dollar: 10]
```

1. The body of the [Define_Type] ... [/Define_Type] tags for the Ex_Dollar type are executed. Local instance variables are defined and all member tags are defined.
2. If the [Ex_Dollar->onCreate] callback tag is defined then it is called.

To define a [Null->onCreate] callback tag:

The Ex_Dollar data type is too simple to require an [Ex_Dollar->onCreate] callback tag. All the initialization which is needed is performed in the creator tag. However, for debugging purposes it might be nice to know each time an instance of the new data type is created. The following [Ex_Dollar->onCreate] tag logs the current value of the instance variable Amount each time a new instance of the data type is created.

```
[Define_Tag: 'onCreate']
  [Log: -Window] Create Ex_Dollar: [Self->'Amount'].[/Log]
[/Define_Tag]
```

onConvert Callback

The [Null->onConvert] callback tag is called when an instance of a custom type is cast to a built-in data type. This tag will be called when an instance of a custom type is used in an expression with built-in data types that requires an integer, decimal, or string value. Each custom type must support being cast to the string data type and should support being cast to the decimal or integer data types if possible.

The [Null->onConvert] callback is called with the name of the type to which the current instance is being converted (either string, integer, or decimal). If the name of the type is not recognized then the [Null->onConvert] tag should return Null. Lasso will attempt to convert the custom data type using another method or will throw an error.

To define a [Null->onConvert] callback tag:

The [Ex_Dollar->onConvert] callback tag is called when an Ex_Dollar amount is cast to a built-in data type. If the value is cast to a decimal or an integer then the callback tag will cast the value in the Amount instance variable to the appropriate data type. If the value is cast to a string then the [Ex_Dollar->Get] member tag which was defined previously will be called.

Otherwise, the callback tag will return Null instructing Lasso that the conversion is not supported.

```
<?LassoScript
  Define_Tag: 'onConvert';
    Local: 'Type' = (Params)->(Get: 1);
    If: (Local: 'Type') == 'String';
      Return: (Self->Get);
    Else: (Local: 'Type') == 'Integer';
      Return: (Integer: (Self->'Amount'));
    Else: (Local: 'Type') == 'Decimal';
      Return: (Decimal: (Self->'Amount'));
    /If;
    Return Null;
  /Define_Tag;
?>
```

In the following code a variable Price is set to a value of type Ex_Dollar. Then that variable is cast to different data types. Notice that the [Output] tag automatically casts all values to the string data type.

```
[Variable: 'Price' = (Ex_Dollar: 19.95)]
<br>[(Variable: 'Price')]
<br>[(Integer: (Variable: 'Price'))]
<br>[(Decimal: (Variable: 'Price'))]

➜ <br>$19.95
<br>20
<br>19.95
```

onDestroy Callback

The [Null->onDestroy] callback tag is the last member tag called for each instance of a custom type. The [Null->onDestroy] callback tag allows any cleanup code that needs to be performed to be executed before the tag is purged from memory. The [Null->onDestroy] tag is called once for each instance of a custom type.

The [Null->onDestroy] callback tag is called in the following instances.

- If a custom type literal is created and not stored in a variable, the instance is destroyed as soon as the current tag completes.

  ```
  [(Ex_Dollar: 10.0)]
  ```

- If a custom tag is created within the [Define_Tag] ... [/Define_Tag] tags of a custom tag declaration and stored in a local variable then the instance is destroyed as soon as the custom tag completes.
- If a custom tag is created within the [Define_Type] ... [/Define_Type] tags of a custom type or is stored in an instance variable within a custom type

then the instance is destroyed as soon as the custom type within which it is stored is destroyed.

- If a custom type is stored within a page variable then it will be destroyed as soon as the page finishes executing, but before it is served to the site visitor.

To define a [Null->onDestroy] callback tag:

The Ex_Dollar data type is too simple to require an [Ex_Dollar->onDestroy] callback tag. The instance variable Amount and each of the member tags will be destroyed automatically by Lasso. However, for debugging purposes it might be nice to know each time an instance of the new data type is destroyed. The following [Ex_Dollar->onDestroy] tag logs the current value of the instance variable Amount each time a new instance of the data type is destroyed.

```
[Define_Tag: 'onDestroy']
  [Log: -Window] Destroy Ex_Dollar: [Self->'Amount'].[/Log]
[/Define_Tag]
```

Unknown Tag Callback

The [Null->_UnknownTag] callback tag is called when a tag that does not exist for the current data type is referenced. This callback tag allows a custom data type to respond to member tags which are not explicitly created. The tag name which was called can be retrieved using the [Tag_Name] tag.

None of the callback tags are ever passed to the [Null->_UnknownTag] callback. Callback tags must be defined explicitly in order to be implemented.

Order of operation:

When a member tag is called on a custom type:

1. If a member tag with that name is defined then it is executed.
2. If an instance variable with that name is defined then its value is returned.
3. If no member tag or instance variable with that name is defined then the [Null->_UnknownTag] callback is executed.
4. If the unknown tag callback is not defined then an error is returned.

To define a [Null->_UnknownTag] callback tag:

The Ex_Dollar data type could implement a conversion to different currencies using the unknown tag callback.

Assume that there is a tag [Currency_Convert] which accepts a value, a -From parameter with the code for what currency to convert from, and a -To

parameter with the code for what currency to convert to. The tag uses data from a site on the Internet to get accureate real-time conversion rates.

Rather than coding in all currency codes explicitly and unknown tag callback can be used to pass any unknown member tags to the [Currency_Convert] tag. An error will be returned if the tag name is not a valid currency code.

```
[Define_Tag: '_UnknownTag']
  [Local: 'Code' = (Tag_Name)]
  [Local: 'Result' = (Currency_Convert: (Decimal: Self->'Amount'),
    -From='USD', -To=#Code)]
  [Return: (Decimal: #Result)]
[/Define_Tag]
```

The following code would now work to convert the U.S. currency represented by the [Ex_Dollar] type to U.K. Pounds represented by UKP.

```
[Variable: 'Price' = (Ex_Dollar: 19.95)]
<br>[(Variable: 'Price')->(UKP)]
```

➜
31.24

Symbol Overloading

Lasso allows complex expressions using math and string symbols to be specified as tag parameters. In addition, a set of assignment symbols allow a variable to be modified in place without returning a value. A list of common symbols is shown in *Table 4: Overloadable Symbols*.

Each data type can assign its own meanings to each of the symbols that Lasso provides. For example, the built-in integer and decimal data types use the + symbol for addition while the built-in string data type uses the + symbol for concatenation. In general it is wisest to match the common meanings of the symbols whenever possible. Ideally, the user will be able to use each data type's custom symbols interchangeably with the symbols provided by the built-in data types.

The meaning of corresponding assignment symbols, unary symbols, and binary symbols should be compatible whenever possible. The operation [(Variable: 'myVariable') += 'Value'] should be the same as the operation [Variable 'myVariable' = $myVariable + 'Value'].

Table 5: Overloadable Symbols

Symbol	Description
+	Unary/Binary symbol for addition or concatenation.
-	Unary/Binary symbol for subtraction or deletion.
*	Binary symbol for multiplication or repetition.
/	Binary symbol for division.
%	Binary symbol for modulus.
++	Unary increment symbol prefix or postfix.
--	Unary decrement symbol prefix or postifx.
==	Binary symbol for equality. Returns boolean.
!=	Binary symbol for inequality. Returns boolean.
>	Binary symbol for greater than. Returns boolean.
>=	Binary symbol for greater or equal. Returns boolean.
<	Binary symbol for less than. Returns boolean.
<=	Binary symbol for less or equal. Returns boolean.
>>	Binary symbol for contains. Returns boolean.
=	Assignment symbol.
+=	Addition assignment symbol.
-=	Subtraction assignment symbol.
*=	Multiplication assignment symbol.
/=	Division assignment symbol.
%=	Modulus assignment symbol.

Each of these symbols can be redefined or overloaded for a custom data type. The data type of the left parameter to a binary operator determines which tag is used to perform the operation. If a data type does not support the symbol then the parameter is cast to string and the string symbol is used instead.

Other symbols such as $, #, @ cannot be overloaded. These are core language constructs. The logical symbols ||, &&, and ! cannot be overloaded, but a custom behavior can be defined when a custom data type is cast to boolean.

Callback Tags

Each custom type can define a number of callback tags using the [Define_Tag] ... [/Define_Tag] tags within the [Define_Type] ... [/Define_Type] definition for the type. These callback tags will be executed with appropriate parameters when the data type is used in a complex expression.

Table 5: Comparison Callback Tags, *Table 6: Symbol Callback Tags*, and *Table 7: Assignment Callback Tags* detail the tags that are available. These tag names are reserved. No member tags with these names should be defined. These tags are not normally called by a Lasso developer, they are called automatically by Lasso in the specified situation. Although there is no protection to prevent a Lasso developer from calling these tags directly, results should be considered undefined if they do.

Table 6: Comparison Callback Tags

Tag	Description
[Null->onCompare]	Called when the current instance is used in a comparison expression. Accepts a single parameter, the value to be compared against. Should return 0 if the parameter is equal to the current instance, a positive number if the parameter is greater, or a negative number if the parameter is less. Called for the ==, !=, <, <=, >, >= symbols.
[Null->>>]	Called when an instance is used as the left parameter of a contains symbol. Accepts a single parameter which is the right parameter of the symbol. This tag should return True if the right parameter is contained in the current instance.

Note: These callback tags are not included in the LDML 8 tag list. They are intended to be called by Lasso automatically rather than being called like other member tags.

onCompare Callback

The [Null->onCompare] callback tag is called when an instance of a custom type is used as the left parameter of a comparison symbol ==, !=, <, <=, >, or >=. The callback tag is called with the value of the right parameter of the symbol. The result of the tag should be one of the following.

- **Equality** – If the value of the right parameter is equal to the value of the current instance of the custom type then the return value should be 0. This will evaluate to True for the ==, <=, and >= symbols.
- **Less Than** – If the value of the right parameter is less than the value of the current instance of the custom type then the return value should be any number less than 0. This will evaluate to True for the <, <=, and != symbols.
- **Greater Than** – If the value of the right parameter is greater than the value of the current instance of the custom type then the return value

should be any number greater than 0. This will evaluate to True for the >, >=, and != symbols.

If a comparison cannot be made then Null should be returned instead. Lasso will attempt to perform a cast in order to compare the two values instead. If no [Null->onCompare] callback tag is defined then Lasso will attempt to perform a cast in order to compare the two values as well.

The value of the left parameter determines the type of comparison which is used. If a custom type is used as the right parameter in a comparison expression and a built-in data type is used as the left parameter then the custom type is cast to the appropriate built-in data type and the values are compared.

Note: The [Array->Find] and [Array->Sort] member tags use comparisons to determine the found set or order of elements in the array. A custom data type will be searched or sorted according to the results of the [Null->onCompare] callback tag.

To define a [Null->onCompare] callback tag:

The [Ex_Dollar->onCompare] callback tag will simply cast any value that is assigned to it to the decimal data type then compare that value to the value stored in the Amount instance variable.

```
<?LassoScript
  Define_Tag: 'onCompare';
    Local: 'Temp' = (Decimal: (Params)->(Get: 1));
    If: (Local: 'Temp') == (Self->'Amount');
      Return: 0;
    Else: (Local: 'Temp') < (Self->'Amount');
      Return -1;
    Else: (Local: 'Temp') > (Self->'Amount');
      Return 1;
    /If;
  Return Null;
  /Define_Tag;
?>
```

In the following code a variable Price is set to a value of type Ex_Dollar. Then that variable is compared to different data types.

```
[Variable: 'Price' = (Ex_Dollar: 19.95)]
<br>[(Variable: 'Price') == (String: '19.95')]
<br>[(Variable: 'Price') == (Integer: 20)]
<br>[(Variable: 'Price') == (Decimal: 19.95)]

➜ <br>True
<br>False
<br>True
```

Contains Callback

The [Null->>>] callback tag is called when an instance of a custom type is used as the left parameter of a >> comparison symbol. The callback tag is called with the value of the right parameter of the symbol. The result of the tag should be one of the following.

- **True** – If the value of the right parameter is contained within the current instance.
- **False** – If the value of the right parameter is not contained within the current instance.

If the contains operation cannot be performed then Null should be returned instead. Lasso will attempt to perform a cast in order to perform the contains operation. If no [Null->>>] callback tag is defined then Lasso will attempt to perform a cast in order to perform the contains operation as well.

If a custom type is used as the right parameter in a contains expression and a built-in data type is used as the left parameter then the custom type is cast to the appropriate built-in data type and the values are compared.

To define a [Null->>>] callback tag:

The [Ex_Dollar->>>] callback tag will cast any value that is assigned to it to the string data type. If the output from [Ex_Dollar->Get] run on the [Self] tag contains the parameter then True is returned.

```
<?LassoScript
  Define_Tag: '>>';
    Local: 'Temp' = (String: (Params)->(Get: 1));
    Return: (Self->Get) >> #Temp;
  /Define_Tag;
?>
```

In the following code a variable Price is set to a value of type Ex_Dollar. Then that variable is checked to see if it contains $ which it does.

```
[Variable: 'Price' = (Ex_Dollar: 19.95)]
<br>[(Variable: 'Price') >> '$']

➜ <br>True
```

Table 7: Symbol Callback Tags

Tag	Description
[Null->+]	Called when an instance is used as the left parameter of an addition symbol. Accepts a single parameter which is the right parameter of the symbol. If no parameter is specified then the unary symbol is being used.
[Null->-]	Called when an instance is used as the left parameter of a subtraction symbol. Accepts a single parameter which is the right parameter of the symbol. If no parameter is specified then the unary symbol is being used.
[Null->*]	Called when an instance is used as the left parameter of a multiplication symbol. Accepts a single parameter which is the right parameter of the symbol.
[Null->/]	Called when an instance is used as the left parameter of a division symbol. Accepts a single parameter which is the right parameter of the symbol.
[Null->%]	Called when an instance is used as the left parameter of a modulus symbol. Accepts a single parameter which is the right parameter of the symbol.
[Null->++]	Called when an instance is used as the left or right parameter of a unary increment symbol.
[Null->--]	Called when an instance is used as the left or right parameter of a unary decrement symbol.

Note: These callback tags are not included in the LDML 8 tag list. They are intended to be called by Lasso automatically rather than being called like other member tags.

Symbol Callback Tags

The symbol callback tags are called whenever the custom data type is used as the left parameter to one of the built-in symbols +, -, *, /, or % or when the custom data type is used as the lone parameter to the + , ++, - or -- unary symbols. These tags usually return a value of the custom data type, but can return a value of any data type.

For the binary operators, the right parameter to the symbol is provided as the parameter of the callback function and could be of any data type. For the unary operators, no parameter is specified.

If no callback tag is defined for a given symbol then Lasso will attempt to cast values to string and will use the built-in string symbols.

To define a [Null->-] callback tag:

The [Ex_Dollar->-] callback tag will create a new [Ex_Dollar] data type. The value of the new type will be found by either subtracting a value from the Amount instance variable if a parameter is specified or by changing the sign of the Amount instance variable if no parameter is specified.

```
<?LassoScript
  Define_Tag: '-';
  If: (Params->Size > 0);
    Return: (Ex_Dollar: (Self->'Amount') - (Decimal: Params->(Get: 1)));
  Else;
    Return: (Ex_Dollar: (Self->'Amount') * (-1));
  /If;
  /Define_Tag;
?>
```

In the following code a variable Price is initialized with a value of 19.95. Then, 5.95 is subtracted from variable and the result is output. Notice that even though the amount subtracted is a decimal, the result is of type Ex_Dollar and outputs with proper formatting.

```
[Variable: 'Price' = (Ex_Dollar: 19.95)]
<br>[(Variable: 'Price') - 5.95]
➜ <br>$14.00
```

Table 8: Assignment Callback Tags

Tag	Description
[Null->onAssign]	Called when an assignment is made to the current instance from any other data type using the = symbol. This tag should return True if the assignment was successful.
[Null->+=]	Called when an instance is used as the left parameter of an addition assignment symbol. Accepts a single parameter which is the right parameter of the symbol. This tag should return true if the assignment was successful.
[Null->-=]	Called when an instance is used as the left parameter of a subtraction assignment symbol. Accepts a single parameter which is the right parameter of the symbol. This tag should return true if the assignment was successful.
[Null->*=]	Called when an instance is used as the left parameter of a multiplication assignment symbol. Accepts a single parameter which is the right parameter of the symbol. This tag should return true if the assignment was successful.
[Null->/=]	Called when an instance is used as the left parameter of a division assignment symbol. Accepts a single parameter which is the right parameter of the symbol. This tag should return true if the assignment was successful.
[Null->%=]	Called when an instance is used as the left parameter of a modulus assignment symbol. Accepts a single parameter which is the right parameter of the symbol. This tag should return true if the assignment was successful.

Note: These callback tags are not included in the LDML 8 tag list. They are intended to be called by Lasso automatically rather than being called like other member tags.

onAssign Callback

The [Null->onAssign] callback tag is called when an instance of a custom type is used as the left parameter of the assignment symbol =. The callback tag is called with the value of the right parameter of the symbol. The tag should attempt to store the value of the right parameter as the new value of the current instance of the custom type. It should return one of the following values.

- **True** – The callback tag should return True if the assignment was successful. This is the sign to Lasso that no further work needs to be done.
- **False** – If for any reason the assignment cannot be performed then the callback tag should return False. Lasso will instead attempt to cast the value of the right parameter to the data type of the left parameter and try the assignment again.

If no [Null->onAssign] callback tag is defined then Lasso will attempt to cast values to the current data type by calling the [Null->onConvert] tag of the right parameter of the assignment operator. For maximum compatibility, each data type should support at least all built-in data types for assignment and conversion.

To define a [Null->onAssign] callback tag:

The [Ex_Dollar->onAssign] callback tag will simply cast any value that is assigned to it to the decimal data type then store that value in the Amount instance variable. This mimics the behavior of the [Ex_Dollar->Set] member tag which was defined previously.

```
<?LassoScript
  Define_Tag: 'onAssign';
    (Self->'Amount') = (Decimal: (Params)->(Get: 1));
  /Define_Tag;
?>
```

In the following code a variable Price is initialized with a value of type Ex_Dollar. The variable is then assigned a string value 19.95 which is cast to a decimal value by the [Ex_Dollar->onAssign] tag called implicitly by Lasso to perform the assignment operator.

```
[Variable: 'Price' = (Ex_Dollar)]
[(Variable: 'Price') = '19.95']
<br>[(Variable: 'Price')->Get]
```

➜
$19.95

Assignment Symbols Callbacks

The [Null->+=], [Null->-=], [Null->*=], [Null->/=], and [Null->%] callback tags are called when an instance of a custom type is used as the left parameter of the corresponding assignment symbol +=, -=, *=, /=, or %=. The callback tag is called with the value of the right parameter of the symbol. The tag should attempt to perform the desired operation and store the value of the right parameter as the new value of the current instance of the custom type. It should return one of the following values.

- **True** – The callback tag should return True if the assignment was successful. This is the sign to Lasso that no further work needs to be done.
- **False** – If for any reason the assignment cannot be performed then the callback tag should return False. Lasso will instead attempt to cast the value of the right parameter to the data type of the left parameter and try the assignment again.

If no callback tag for a given assignment symbol is defined then Lasso will attempt to cast values to the current data type by calling the [Null->onConvert] tag of the right parameter of the assignment operator.

To define a [Null->+=] callback tag:

The [Ex_Dollar->=+] callback tag will simply cast any value that is assigned to it to the decimal data type and add that value to the Amount instance variable.

```
<?LassoScript
  Define_Tag: '+=';
    (Self->'Amount') += (Decimal: (Params)->(Get: 1));
  /Define_Tag;
?>
```

In the following code a variable Price is initialized with a value of type Ex_Dollar and a value of 19.95. Finally, the += symbol is used to add an additional 5.95 to the variable.

```
[Variable: 'Price' = (Ex_Dollar: '19.95')]
[(Variable: 'Price') += '5.95']
<br>[(Variable: 'Price')->Get]
```

➜
$19.95

Inheritance

Custom types can be created which inherit properties from other custom types. Each type which the custom type should inherit from is specified after the name of the custom type in the opening [Define_Type] tag. These are called parent types and the current type being defined is called a child type. Usually, only one parent type is specified.

All instance variables and member tags of the parent types are inherited by the child type. If the child type defines an instance variable or member tag with the same name as one of the parent types then the child's definition overrides the parent's definition.

Custom types can inherit properties from built-in data types. A custom type will inherit any member tags which the built-in type defines, but will not inherit any of the features that require callback functions. It will be necessary to create custom casting and assignment callbacks and to implement any symbols which are desired.

All custom data types inherit from the null data type. The tags of the null data type such as [Null->Type] can be used by any data type within Lasso. These tags can be overridden, but doing so can cause unexpected results.

The member tags and instance variables of the parent tag can be accessed using the [Parent] tag. This tag works like the [Self] tag, but returns the value of the current data type instance as it would be if it were of the parent type.

The creator tag [Null->onCreate] and destructor tag [Null->onDestroy] for each parent data type is called automatically when a new instance of the child data type is created.

To define a custom type that inherits from another custom type:

The Ex_Dollar type which is defined in this chapter only works with U.S. currency and outputs values using the dollar sign $. It is possible to create a sub-type that works with a different type of currency. For example, a new type Ex_UKPounds could be created which inherited from Ex_Dollar, but output values with a British pound symbol £. by overriding the [Ex_Dollar->Get] tag with a new [Ex_UKPounds->Get] tag.

The type is defined as inheriting from Ex_Dollar by specifying Ex_Dollar after the name of the new type in the opening [Define_Type] tag. All the member tags of Ex_Dollar are automatically defined as is the instance variable Amount.

The [Ex_UKPounds->Get] member tag is defined and overrides the equivalent [Ex_Dollar->Get] member tag. The [Self->Parent] tag is used to reference the Amount instance variable from the parent type.

```
<?LassoScript
  Define_Type: 'UKPounds', 'Ex_Dollar', -Namespace='Ex_';

    Define_Tag: 'Get';
      Return: '£' + (Self->Parent->'Amount');
    /Define_Tag;

  /Define_Type;
?>
```

The following example sets two variables, one to a value of Ex_Dollar type and the other to a value of Ex_UKPounds type, then outputs both values. The types are converted to strings when they are output and the appropriate [Ex_Dollar->Get] or [Ex_UKPounds->Get] tag is called to format the output.

```
[Variable: 'American'= (Ex_Dollar: 100)]
<br>[Variable: 'American']
[Variable: 'British'= (Ex_UKPounds: 100)]
<br>[Variable: 'British']

➜ <br>$100.00
<br>£100.00
```

Libraries

Libraries can be used to package custom tags and custom types into a format which is easy for any Lasso developer to incorporate into a Lasso-powered Web site.

The following types of libraries can be created:

- **On-Demand Tag Library** – A set of custom tag and custom type declarations can be stored in a format file or LassoApp and placed in the LassoLibraries folder in the Lasso Professional 8 application folder. The format file or LassoApp should have the same name (before the .Lasso or .LassoApp file suffix) as the namespace of the tags defined within. Sub-folders can be used to define nested namespaces.
- **Library Format File** – A set of custom tag and custom type declarations can be stored in a format file and then included in any other Lasso format file using the [Library: 'library.lasso'] tag. This is a good way to create and use a library file whose defined tags and types will only be needed on a few pages in a site.
- **LassoStartup Format File** – A set of custom tag and custom type declarations can be stored in a format file placed within the LassoStartup folder. After Lasso Service is restarted all tags, types, and page variables which are defined within the format file will be available to all format files which are executed on the server.

52

Chapter 52
LDML Data Sources

Data sources can be implemented entirely in LDML using the techniques and tags documented in this chapter.

- *Overview* describes the basic methodology of creating an LDML data source and how to install an LDML data source.
- *Data Source Tags* describes the tags that are available to help make an LDML data source.
- *Data Source Type* describes the data type that must be implemented to ccreate a new LDML data source.

Overview

Lasso provides the ability to create a data source enitely in LDML code. This makes it possible to easily implement entirely new types of data source modules without using either C/C++ or Java code.

- LDML data sources can use the net type to connect to a remote data source. Incoming data can be parsed using string or XML tags.
- LDML data sources can use XML-RPC or SOAP to connect to remote procedures. Searching remote application servers can be made as easy as searching local databases.
- LDML data sources can use the file tags to provide access to files on the local machine through standard Lasso actions. For example, the XML tags could be used to search local XML files.
- LDML data sources can be used as a wrapper around other Lasso data sources providing round-robin load balancing, intelligent fail over behavior, or caching.

An LDML data source is implemented as a custom type that must define certain member tags. The data source is registered with Lasso at Lasso Startup using the [DataSource_Register] tag and then appears within Lasso Administration along with the standard data sources, JDBC drivers, and any third-party data sources implemented in LCAPI or LJAPI.

The LDML data source must provide a list of databases to Lasso. When any of these database names are used within an inline action a new instance of the LDML data source type is instantiated and a member tag is called to perform the action. Once the action is performed the instance of the LDML data source type is deleted.

Data Source Register

The following table shows the tag that is used to register an LDML data source. This tag should be called in LassoStartup once for each LDML data source.

Table 1: Data Source Register

Tag	Description
[Datasource_Register]	Registers a data type as a new data source. The tag requires one parameter which is the name of the data source that implements the data source. The specified data source must implement the tags described in the following section.

To register an LDML data source:

Use the [Datasource_Register] tag with the name of the custom data source that implements the LDML data source. This tag should be called once by a file in the LassoStartup folder. In the following example a custom data source Ex_DataSource is regsitered.

```
<?LassoScript
  Define_Type: 'DataSource', -Namespace='Ex_';
    … Data Source Definition …
  /Define_Type;
  Datasource_Register: 'Ex_DataSource';
?>
```

Data Source Type

Each custom type that implements an LDML data source must define the following member tags. Lasso calls these tags in order to retrieve the list of databases, tables, schemas, or fields from the data source host and to perfrom database actions when an [Inline] is called with one of the data source's databases. Each of these member tags is described in more detail below.

Table 2: Data Source Member Tags

Tag	Description
Initialize	Called once immediately after the data source is registered. This allows the data source to perform an initialization routine if required.
Terminate	Called once immediately before Lasso Service is quit. This tag allows the data source to perform a global cleanup routine if required.
onCreate	Called when a new instance of the data source is created. The tag is passed an array of information about the host for the data source.
DatabaseNames	Called when Lasso needs a list of the databases that the data source provides. The return value must be an array of strings.
DatabaseExists	Pased a single parameter which is the name of a database. The return value should be True or False depending on whether this data source can handle the database.
SchemaNames	Passed a single parameter which is the name of a database. The return value should be an array of strings representing the schemas (if any) this data source supports.
TableNames	Passed a single parameter which is the name of a database. The return value should be an array of strings representing the table names (if any) this data source supports.
Info	Passed two parameters: the name of the database and name of the table. The return value should be an array of arrays for each field in the specified table. Each field array should contain four elements (field name, required true/false, type, protected true/false).
Action	Passed a single parameter which is an array of action parameters. The data source must interpret the action and provide an appropriate response.

Initialize and Terminate

The Initialize and Terminate tags are provided so that a data source can perform global initailization and cleanup routines if requires. Most data sources will probably not need to implement these.

For example, a data source that stores values in a global variable could set up the variable in the Initialize member tag.

```
<?LassoScript
  Define_Type: 'DataSource', -Namespace='Ex_';

    Define_Tag: 'Initialize;
      Global: 'Ex_DataSource_Storage' = (Map);
    /Define_Tag;

    … Additional Member Tags …
  /Define_Type;
?>
```

Note: If Lasso Service crashes the Terminate tag will never be called. A data source which relies on this tag being called could suffer from data loss if Lasso Service crashes.

onCreate and onDestroy

The onCreate tag is called when Lasso creates a new instance of a data source. This happens when an opening [Inline] tag references a database provided by the data source or when the data source is refreshed in Lasso Administration. The onDestroy tag is called when the data source instance is destroyed.

Lasso does not create new data source instances for nested inlines that reference the same data source host. A single data source instance may be asked to perform actions for a series of different databases and tables on the same host.

The onCreate tag is passed an array of information about the host defined in Lasso Administration for the referenced database. The elements of the array are defined in the following table.

Table 3: Host Information

Tag	Description
Host	Usually the URL of the desired data source host.
Port	The port number for the host.
Schema	The default schema for the host.
Username	The username for the host.
Password	The password for the host.

The information provided to onCreate is set in Lasso Administration. The fields do not have to be used for the purpose their label suggests. If an LDML data source does not require host or port information the documentation for the data source can instruct the end-user to enter whatever information is desired in Lasso Administration.

```
<?LassoScript
  Define_Type: 'DataSource', -Namespace='Ex_';

    Define_Tag: 'onCreate', -Required='HostInfo';

    /Define_Tag;

    … Additional Member Tags …
  /Define_Type;
?>
```

Usually, the data source type will need to store some of the information passed to the onCreate tag in order to remember what host the data source is connected to when an action occurs. For data sources that establish connections with remote hosts or open files the reference to the remote host or local files can sometimes serve as the state for subsequent database actions.

Database, Schema, and Table Names

The DatabaseNames, SchemaNames, and TableNames tags are all used by Lasso Administration to create the entries in Lasso Security for the data source. The database entries in particular are used to route [Inline] database actions to the LDML data source so must be accurate.

This tag will always be called after an onCreate tag which contains the host information for the current data source connection. The onCreate tag needs to store enough state so a subsequent call to DatabaseNames can list the databases for the desired host. Both the SchemaNames and TableNames tags must operate similarly in addition to being passed a specific database name as a parameter.

In the following example the DatabaseNames tag is hardcoded to return two databases Database_One and Database_Two. The TableNames tags returns Table_One and Table_Two for Database_One or Table_Two and Table_Three for Database_Two. In an actual LDML data source the database and table names would usually be generated based on what data was actually available on the remote data source.

```
<?LassoScript
  Define_Type: 'DataSource', -Namespace='Ex_';

    Define_Tag: 'DatabaseNames';
      Return: (Array: 'Database_One', 'Database_Two');
    /Define_Tag;

    Define_Tag: 'TableNames', -Required='Database';
      Select: #Database;
        Case: 'Database_One';
          Return: (Array: 'Table_One', 'Table_Two');
        Case: 'Database_Two';
          Return: (Array: 'Table_Three', 'Table_Four');
      /Select;
    /Define_Tag;

    … Additional Member Tags …
  /Define_Type;
?>
```

SchemaNames is not shown in the example above, but can be coded in exactly the same fashion as TableNames.

Field Info

The Info tag is used by Lasso Administration to create the entries in Lasso Security for the data source. It is also used after a database action to establish a correspondence between field names and the positions in the results array.

The Info tag is passed the name of the database and the name of the table. The return value from the Info tag is an array of arrays. Each element of the returned array represents one field with a four element array. All four elements are required including:

- **Field Name** – A stirng representing the name of the field.
- **Required** – A boolean value indicating if the field is required or not. Should be set to False by default.
- **Field Type** – A string reprsenting the type of the field. The types can be data source specific. They are displayed in the database browser and may be used by some solutions.

- **Protected** – A boolean value indicating if the field is read-only or not. Should be set to False by default.

A generic array for a field is shown below. A data source should use this at a minimum to ensure Lasso has all the information it needs about each field.

```
[Array: 'Field Name', False, 'Text', False]
```

In the following example the Info tag returns an array of fields Field_One, Field_Two, Field_Three, and Field_Four for Database_One and Table_One. The tag could be extended to return field info for the other databases and tables as well. In an actual LDML data source the field info would usually be generated based on what data was actually available on the remote data source.

```
<?LassoScript
  Define_Type: 'DataSource', -Namespace='Ex_';

    Define_Tag: 'Info', -Required='Database', -Required='Table';
      Select: #Database + '.' + #Table;
        Case: 'Database_One.Table_One';
          Return: (Array:
            (Array: 'Field_One', False, 'Text', False);
            (Array: 'Field_Two', False, 'Number', False);
            (Array: 'Field_Three', False, 'Date', False);
            (Array: 'Field_Four', False, 'Binary', False);
          );
        … Cases For Other Databases and Tables
      /Select;
    /Define_Tag;

    … Additional Member Tags …
  /Define_Type;
?>
```

Database Actions

The Action tag is called whenever an LDML data source is used in an inline tag by the end-user. The tag is passed an array of parameters which has the same content as [Action_Params] called inside of an [Inline] ... [/Inline] container tag.

The array of parameters will contain one action (listed below), one each of -Database, -Table, -KeyField, -MaxRecords, -SkipRecords, and -OperatorLogical tags, and additional parameters as specified by the user. It is the LDML data source's responsibility to interpret these parameters, decide what action to perform, and return appropriate results.

The list below explains how Lasso interprets each of the built-in database actions. For best results LDML data sources should try to match these meanings as close as possible. LDML data sources should also respect the -MaxRecords and -SkipRecords values if possible. However, it is not necessary for an LDML data source to implement every action or to provide exactly the same behavior as built-in data sources.

- **-Search** – The parameters specified with the action should be interpreted as search terms. -MaxRecords specifies the maximum number of records that should be returned and -SkipRecords specifies an offset into the found set at which to start returning records. See the section on *Result Sets* below for details about how to return records and set the [Found_Count] and [Total_Count].

  Lasso supports a number of optional parameters that an LDML data source should process if possible. These include -Op parameters that immediately precede name/value parameters and specify what search operator to use, -OpBegin and -OpEnd parameters that allow sophisticated And/Or groupings, -SortField and -SortOrder parameters, as well as -GroupBy, -Distinct, and -SortRandom parameters. The experience of the end-user will be richer if more of these parameters are provided by an LDML data source.

- **-FindAll** – The same as a -Search action, but none of the search parameters should be regarded. Other parameter like -MaxRecords, -SkipRecords, -SortField, etc. should still be processed.

- **-Random** – The same as a -Search action, but a random selection of records is returned. The behavior of -Random differs from data source to data source. It can either be based on -FindAll or -Search. Not all data sources support the -Random action.

- **-Add** – Add a record to the database. The parameters specify the record that is to be added. The result of an -Add action will often contain the single record that was just added to the database. Some data sources use -MaxRecords=0 to suppress returning this record.

- **-Update** – Update a single record within a database. A -KeyField and -KeyValue must in general be specified in order to determine what record to update. The parameters specify the new values for the updated record. The result of an -Update action will often contain the single record that was just updated within the database. Some data sources use -MaxRecords=0 to suppress returning this record.

- **-Delete** – Delete a single record from the database. A -KeyField and -KeyValue must in general be specified in order to determine what record to delete. The parameters are generally disregarded. The result of a -Delete action will usually be an empty set.

- **-Duplicate** – Duplicate a record within the database. A -KeyField and -KeyValue must in general be specified in order to determine what record to duplicate. The parameter may be disregarded or may be used to update the duplicated record. The result of a -Duplicate action will often contain the single record that was just added to the database. Some data sources use -MaxRecords=0 to suppress returning this record. Not all data sources support the -Duplicate action.
- **-SQL** – Execute a raw SQL statement on the data source. This action is used for data sources that support SQL, but may reasonably be used for raw database commands for any data source (e.g. XPath statements or data source specific low-level commands). Most of the parameters of the action are generally disregarded except for -MaxRecords and -SkipRecords.
- **-Nothing** – This is the default action if no other valid action is defined. An LDML data source should in general not perform any action when a -Nothing action is specified. However, some data sources will use a -Nothing action to send a keep-alive ping to a data source.

  Some data sources may define additional actions beyond those listed here. Those actions will be reported as a -Nothing action with the actual action specified within the parameters passed to the Action tag.

The actual implementation of each of the actions is up to the LDML data source developer. LDML data sources can run from simple implementations that support only a couple actions to full-fledged modules that support all of the rich set of actions and additional parameters that Lasso provides.

The example below uses a [Select] ... [Case] ... [/Select] tag to choose which action to perform based on the contents of the #Action_Params array. Any actions that are not supported by the data source are caught by the default -Nothing action option at the end. See the ***Result Sets*** section below for an example of how a -Search action can return results.

```
<?LassoScript
  Define_Type: 'DataSource', -Namespace='Ex_';

    Define_Tag: 'Action', -Required='Action_Params';
      Select: True;
        Case: (#Action_Params >> -Search);
          … Search Action (See Result Sets below for an example) …
        Case: (#Action_Params >> -FindAll);
          … FindAll Action …
        Case: (#Action_Params >> -Add);
          … Add Action …
        Case: (#Action_Params >> -Update);
          … Update Action …
        Case: (#Action_Params >> -Delete);
          … Delete Action …
```

```
            Case;
              … Nothing Action …
          /Select;
        /Define_Tag;

        … Additional Member Tags …
      /Define_Type;
    ?>
```

Result Sets

Most database actions return results to the end-user. These results are returned in the same way no matter if the action is a -Search or -Add.

The current result set is set using the [Action_AddRecord] tag once for each record in the result set. Usually, the entire result set is not returned, but only up to a maximum of -MaxRecords records starting at the offset defined by -SkipRecords. The [Action_AddRecord] tag requires one parameter which is an array of values for each field in a single returned record.

The field names returned by the Info tag should correspond to the same order as the results passed to the [Action_AddRecord] tag. This allows Lasso to return the proper field value for each [Field] tag.

Lasso will automatically calculate [Shown_Count], [Shown_First], and [Shown_Last] based on the number of times [Action_AddRecord] is called and the value for -SkipRecords. The value for [Found_Count] can be set by calling [Action_SetFoundCount]. In addition, some databases can set [Total_Count] to the total number of records in the database using [Action_SetTotalCount].

Finally, when processing an -Add or -Update action the current [RecordID_Value] can be set using [Action_SetRecordID]. This is separate from the key field value which is set automatically based on the value for -KeyField and the mapping from field names to field values. Some LDML data sources may want to set this value in order to return an internally generated ID that may be different from the key field value.

The following table shows the tags that are available for returning database action results.

Table 4: Result Set Tags

Tag	Description
[Action_AddRecord]	Adds a record to the found set after a database action. Requires one parameter which is an array of strings representing the results for one record of the found set.
[Action_AddInfo]	Reports the names of fields in the result set. Requires one parameter which is an array of arrays. Each field array needs name, required, type, and protection. The output of the ->Info member tag matchs the required parameter of this tag.
[Action_SetFoundCount]	Sets the number of records found in a database during a search action. Requires one integer parameter.
[Action_SetTotalCount]	Sets the total number of records that are in a database during a search action. Requires one integer parameter.
[Action_SetRecordID]	Sets the record ID value. This is the value returned by [RecordID_Value]. Requires one integer parameter.

The example below returns a set of eight records from a -Search action. It is hard-coded to return eight records and set the found count to 32 records and the total count to 256 records. In an actual LDML data source the field values would usually be generated based on what data was actually available on the remote data source.

```
<?LassoScript
  Define_Type: 'DataSource', -Namespace='Ex_';

    Define_Tag: 'Action', -Required='Action_Params';
      Select: True;
        Case: (#Action_Params >> -Search);
          [Action_SetFoundCount: 32]
          [Action_SetTotalCount: 256]
          [Action_AddInfo: Self->(Info:
            #Action_Params->(Find: -Database)->First->Second,
            #Action_Params->(Find: -Table)->First->Second)]
          [Action_AddRecord: (Array: 'One', 'Two', 'Three', 'Four')]
          [Action_AddRecord: (Array: 'Five', 'Six', 'Seven', 'Eight')]
          [Action_AddRecord: (Array: 'Nine', 'Ten', 'Eleven', 'Twelve')]
          [Action_AddRecord: (Array: 'Thirteen', 'Fourteen', 'Fifteen', 'Sixteen')]
          [Action_AddRecord: (Array: 'One', 'Two', 'Three', 'Four')]
          [Action_AddRecord: (Array: 'Nine', 'Ten', 'Eleven', 'Twelve')]
          [Action_AddRecord: (Array: 'Five', 'Six', 'Seven', 'Eight')]
          [Action_AddRecord: (Array: 'Thirteen', 'Fourteen', 'Fifteen', 'Sixteen')]
        … Cases For Other Actions …
      /Select;
    /Define_Tag;
```

```
    … Additional Member Tags …
  /Define_Type;
?>
```

IX

Section IX
Lasso C/C++ API

This section includes instructions for extending the functionality of Lasso by creating new tags, data types, and Web server connectors written in C/C++.

- *Chapter 53: LCAPI Introduction* includes general information about extending Lasso's functionality.
- *Chapter 54: LCAPI Tags* discusses how to create new tags in LCAPI including substitution tags, asynchronous tags, and remote procedures.
- *Chapter 55: LCAPI Data Types* discusses how to create new data types in LCAPI including sub-classing and symbol overloading.
- *Chapter 56: LCAPI Data Sources* discusses how to create new data sources in LCAPI.
- *Chapter 57: LCAPI References* includes information about each of the function calls available in LCAPI.

Lasso can also be extended using LDML or Java. See the preceding section on the *LDML API* or the following section on the *Lasso Java API* (LJAPI) for more information..

53

Chapter 53
LCAPI Introduction

This chapter provides an introduction to the Lasso C/C++ API (LCAPI) which allows new tags, data types, and data source connectors to be written in C/C++.

- *Overview* includes a description of what types of modules can be built with LCAPI.
- *Requirements* describes the basic system requirements for building LCAPI modules
- *Getting Started* includes a walktrhough of building a sample tag module on both Mac OS X and Windows.
- *Debugging* describes how the debugging tools can be used on an LCAPI module in either Mac OS X or Windows.
- *Frequently Asked Questions* includes a series of common questions that new users of LCAPI have and answers.

Overview

The Lasso C/C++ Application Programming Interface (LCAPI) lets you write C or C++ code to add new LDML substitution tags, data types, and data source connectors to Lasso Professional 8.

Writing tags in LCAPI offers advantages over LJAPI and custom LDML tags in speed and system performance. However, tags must be compiled separately for Windows 2000/XP and Mac OS X in order to support each platform. See the Custom Tags and Custom Types chapters for more information on writing custom tags in LDML. LCAPI is functionally similar to LJAPI. See the Lasso Java API chapter for more information about writing tags, data types, and data source connectors in Java using LJAPI.

This chapter provides a walk-through for building an example substitution tag, data source connector, and data type in LCAPI. Source code for the Lasso MySQL module as well as the code for the substitution tag, data type, and data source connector examples are included in the Lasso Professional 8/Documentation/3 - Language Guide/Examples/LCAPI folder on the hard drive.

Requirements

In order to write your own LDML substitution tags or data source connectors in C or C++, you need the following:

Windows

- Microsoft Windows 2000 or Microsoft Windows XP Professional.
- Microsoft Visual C++ .NET.
- Lasso Professional 8 for Windows 2000/XP.

Mac OS X

- Mac OS X 10.3 with GNU C++ compiler and linker (Dev Tools) installed.
- Lasso Professional 8 for Mac OS X.

Getting Started

This section provides a walk-through for building sample LCAPI tag modules in Windows 2000/XP and Mac OS X.

To build a sample LCAPI tag module in Windows 2000/XP:

1. Locate the following folder in the hard drive.

   C:\Program Files\OmniPilot Software\Lasso Professional 8\ Documentation\3 - Language Guide\Examples\LCAPI\Tags\MathFuncsTags

2. In the MathFuncsTags folder, double-click the MathFuncsCAPI.sln project file (you need Microsoft Visual C++ .NET in order to open it).
3. Choose Build > Build Solution to compile and make the MathFuncsCAPI.DLL.
4. After building, a Debug folder will have been created inside your MathFuncsCAPI project folder.
5. Open the MathFuncsTags/Debug folder and drag MathFuncsCAPI.DLL into the Lasso Professional 8/LassoModules folder on the hard drive.

6 Stop and then restart Lasso8Service.

7 New tags [Example_Math_Abs], [Example_Math_Sin] and [Example_Math_Sqrt] are now part of the LDML language.

8 Drag the sample Lasso format file called MathFuncsCAPI.lasso into your Web server root.

9 In a Web browser, view http://localhost/MathFuncsCAPI.lasso to see the new LDML tags in action.

To build a sample LCAPI tag module in Mac OS X:

1 Open a Terminal window.

2 Change the current folder to the Lasso Professional 8/Documentation folder using the following command:

```
cd /Library/Lasso Professional 8/Documentation/3 - Language Guide\Examples\
LCAPI\Tags\MathFuncsTags
```

3 Build the sample project using the provided makefile (you'll need to know a Mac OS X administrator password to use sudo).

```
sudo make
```

4 After building, a Mac OS X dynamic library file named MathFuncsCAPI.dylib will be in the current folder. This is the LCAPI module you'll install into the LassoModules folder.

5 Copy the newly-created module to the Lasso modules folder using the following command:

```
cp MathFuncsCAPI.dylib /Applications/Lasso Professional 8/LassoModules
```

6 Quit Lasso Service if it's running, so that the next time it starts up, it will load the new module you just built (you'll need to know a Mac OS X administrator password to use sudo).

```
sudo lasso8ctl stop
```

7 Start Lasso Service so it will load the new module.

```
sudo lasso8ctl start
```

New tags [Example_Math_Abs], [Example_Math_Sin] and [Example_Math_Sqrt] are now part of the LDML language.

8 Copy the sample Lasso format file called MathFuncsCAPI.lasso into your Web server document root.

9 Use a Web browser to view http://localhost/MathFuncsCAPI.lasso to see the new LDML tags in action.

Debugging

You can set breakpoints in your LCAPI DLLs or DYLIBs and perform source-level debugging for your own code. In order to set this up, add path information to your project so it knows where to load executables from. For this section, we will use the provided substitution tag project as the example.

To debug in Windows 2000/XP:

1. Select Processes... from the Debug main menu.
2. In the Processes window, select each instance of Lasso8Service.exe and choose to Attach.
3. Close the Processes window and set a breakpoint in the tagMathAbsFunc function.
4. Use a Web browser to access the sample http://localhost/MathFuncsCAPI.lasso. Visual Studio will stop at the location that the breakpoint was placed.

To debug in Mac OS X:

1. From a Terminal window, change folder into the example LCAPI source code folder by entering the following:

   ```
   cd /Applications/Lasso\ Professional\ 8/Documentation/3 - Language Guide\
   Examples\LCAPI\Tags\MathFuncsTags
   ```

2. Build with debug options turned on by entering the following:

   ```
   sudo make “DEBUG += -g3 -O0”
   ```

   Note: The last two characters of the command are a letter O followed by a zero.

3. Copy the built DYLIB into the LassoModules folder by entering the following:

   ```
   cp MathFuncsCAPI.dylib /Applications/Lasso\ Professional\ 8/LassoModules/
   ```

4. Change folder into the Lasso Professional 8/Tools folder:

   ```
   cd /Applications/Lasso\ Professional\ 8/Tools/
   ```

5. Restart Lasso Service. When it starts up, it will load the new module you just built (you’ll need to know a root password to use sudo).

   ```
   sudo lasso8ctl stop
   ```

6. Start the Lasso Service back up, so it will load the new module.

   ```
   sudo lasso8ctl start
   ```

7 Find out the process ID number of Lasso Service so you can attach to it later with GNU Debugger. Make a note of the process id for Lasso8Service.

```
ps aux | grep Lasso8Service
```

8 Start the GNU Debugger as a root user. You must be root in order to attach to the running Lasso Service process.

```
sudo gdb
```

9 From within GNU Debugger's command line, attach to the Lasso Service process ID by entering the following:

```
attach <type the process id from step 7 here>
```

10 Instruct GNU Debugger to break whenever the function tagMathAbsFunc is called by entering the following:

```
break tagMathAbsFunc
```

11 Use a Web browser to access the sample http://localhost/MathFuncsCAPI.lasso. An example Lasso format file is provided in the LCAPI folder; you must first copy it into your Web server's Documents folder, which is typically /Library/WebServer/Documents.

12 GNU Debugger breaks at the first line in tagMathAbsFunc() as soon as Lasso Service executes that tag in the format file

13 Type help in GNU Debugger for more information about using the GNU Debugger, or search for gdb tutorial on the Web for more in-depth tutorials.

Frequently Asked Questions

How do I install my custom tag?

Once you've compiled your tag module, you'll need to move the module to your installed Lasso Professional LassoModules folder, and then restart Lasso Service. Step-by-step instructions are available in the *Getting Started* section.

How do I return text from my custom tag?

Use either lasso_returnTagValueString to return UTF-8 data, or lasso_returnTagValueStringW to return UTF-16 data. Character data in other encoding methods can be returned by first allocating a string type using lasso_typeAllocStringConv and then returning it using lasso_returnTagValue.

How do I return binary data from my custom tag?

Use lasso_returnTagValueBytes to return binary data.

How do I prevent Lasso from automatically encoding text returned from my custom tag?

Make sure that your tag is registered with the flag_noDefaultEncoding flag. This flag is specified when you call lasso_registerTagModule at startup.

How do I debug my custom tag?

You can set breakpoints in your code and attach your module DLL to Lasso Service. Read the section on *Debugging* LCAPI modules.

How do I get parameters that were passed into my tag?

Most of the parameters passed into your custom tag can be retrieved using the lasso_getTagParam() and lasso_findTagParam() parameter info APIs. lasso_getTagParam() retrieves parameters by index and lasso_findTagParam() retrieves them by name. All parameters retrieved using those functions will be returned as strings. The access the parameters as Lasso type instances, use lasso_getTagParam2 and lasso_findTagParam2.

How do I get the value of unnamed parameters passed into my tag?

While there is no direct way to get unnamed parameters (how do you know what name to ask for?), you can enumerate through all the parameters by index, and then pick out the ones which do not have names. If, after retrieving a parameter, you discover that its data member is an empty string, then that means it is an unnamed parameter, and you can get its value from the name member. An example of this is in the substitution tag tutorial.

What's an auto_lasso_value_t and how do I use it?

It's a data structure which contains both a name and a value (a name/value pair). Many LCAPI APIs fill in this structure for you, and you can access the name and data members directly as null-terminated C-strings.

What is a lasso_type_t and how do I use it?

A lasso_type_t represents an instance of a Lasso type. Any Lasso type can be represented by a lasso_type_t, including strings, integers, or custom types. LassoCAPI provides many functions for allocating or manipulating lasso_type_t instances. All lasso_type_t instances encountered inside a LassoCAPI tag will be automatically garbage collected after the function returns. Therefore, a lasso_type_t instance should not be saved unless it if freed from the garbage collector using lasso_typeDetach.

How do I access variables from the LDML page I'm in?

You may need to get or even create LDML variables (the same variables that an LDML programmer makes when using the [var: 'fred'=12] variable syntax in a format file) from within your LCAPI module. You can retrieve a global variable, as long as it has already been assigned before your custom substitution tag is executed, by calling lasso_getVariable() with the variable's name. Using this method, one could directly set the __html_reply__ variable.

How do I return fatal and non-fatal error codes?

It is very important that your substitution tag return an error code of osErrNoErr (0) if nothing fatal happened. An example of a fatal error would be a missing required parameter, for instance. If you encounter a fatal error, then return a non-zero result code from your tag function, and the Lasso will stop processing the page at that point, and display an error page.

How do I write code that will compile easily across multiple operating systems?

While we cannot provide a complete cross-platform programming tutorial for you here, we can at least provide some guidance. The simplest way to make sure things compile across platforms is to make sure you use standard library functions (from stdio.h and stdlib.h) as much as possible: functions like strcpy(), malloc(), and strcmp() are always available on all platforms. Also note that Unix platforms are case-sensitive, so when you #include files, just make sure you keep the case the same as the file on disk. Finally, stay away from platform-specific functions, such as Windows APIs, which most often are not available on Unix platforms. Take a look at our Unix makefiles which are provided with the sample projects: notice the same source code is used for Windows, and all source files are saved with DOS-style cr/lf linebreaks so as not to confuse the Windows compilers. As a last resort, you can use #ifdef to show/hide portions of source code which are platform-specific.

54

Chapter 54
LCAPI Tags

This chapter includes information about creating tags in C/C++ using the Lasso C/C++ API (LCAPI).

- *Substitution Tag Operation* introduces the concepts behind substitution tags and how they are loaded and accessed through Lasso.
- *Substitution Tag Tutorial* documents a sample project that is shipped with Lasso including a walk-through of the sample code.

Substitution Tag Operation

When Lasso Professional first starts up, it looks for module files (Windows DLLs or Mac OS X DYLIBS) in its LassoModules folder. As it encounters each module, it executes that module's registerLassoModule() function once and only once. LCAPI developers must write code to register each of the new custom tag (or data source) function entry points in this registerLassoModule() function. The following function is required in every LCAPI module. It gets called once when Lasso Professional starts up.

```
void registerLassoModule()
{
  lasso_registerTagModule( "CAPITester", "testtag", myTagFunc,
    REG_FLAGS_TAG_DEFAULT, "simple test LCAPI tag" );
}
```

The preceding example registers a C function called myTagFunc to execute whenever the LDML [CAPITester_testtag] is encountered inside a Lasso script. The first parameter CAPITester is the namespace in which testtag will be placed..

Once the tag function is registered, Lasso will call it at appropriate times while parsing and executing Lasso scripts. The custom tag functions will not be called if none of the custom tags are encountered while executing a script. When Lasso Professional 8 encounters one of your custom tags, it will be called with two parameters: an opaque data structure called a "token", and an integer "action" (which is currently unused). LCAPI provides many function calls which you can use to get information about the environment, variables, parameters, etc., when provided with a token.

The passed-in token can also be used to acquire any parameters and to return a value from your custom tag function.

To build a basic custom tag function:

Enter the following code:

```
osError myTagFunc( lasso_request_t token, tag_action_t action )
{
  const char * retString = "Hello, World!";
  return lasso_returnTagValueString( token, retString, strlen(retString));
}
```

Below is the Lasso script needed to get the custom tag to execute:

```
Here's the custom tag:
[CAPITester_testtag]
<!-- This should display "Hello, World" when this page executes -->
```

This will produce the following output:

```
➜ Here's the custom tag:
Hello, World
```

Substitution Tag Tutorial

This section provides a walk-through of building an example tag to show how LCAPI features are used. This code will be most similar to the sample MathFuncsCAPI project, so in order to build this code, copy the MathFuncsCAPI project folder and edit the project files inside it.

The tag will simply display its parameters, and will look like the example below when called from a Lasso script.

Example of the sample tag's syntax:

```
[sample_tag: 'some text here', -option1='named param', -option2=12.5]
```

Notice the tag takes one unnamed parameter, one string parameter named -option1, and a numeric parameter named -option2. In general, Lasso does not care about the order in which you pass parameters, so plan to make

this tag as flexible as possible by not assuming anything about the order of parameters. The following variations should work exactly the same:

Example of sample tag with different ordered parameters:

```
[sample_tag: -option2=12.5, 'some text here', -option1='named param']

[sample_tag: -option2=12.5, -option1='named param', 'some text here']
```

Substitution Tag Module Code

Shown below is the code for the substitution tag module. This code is referenced in the *Substitution Tag Module Walk Through* section.

```
void registerLassoModule()
{
  lasso_registerTagModule( "sample", "tag", myTagFunc,
    REG_FLAGS_TAG_DEFAULT, "sample test" );
}

osError myTagFunc( lasso_request_t token, tag_action_t action )
{
  lasso_type_t retString = NULL, opt2 = NULL;
  lasso_typeAllocString(token, &retString, "", 0);
  auto_lasso_value_t  v;
  INITVAL(&v);

  if( lasso_findTagParam( token, "-option1", &v ) == osErrNoErr )
  {
    lasso_typeAppendString( token, "The value of -option1 is ", 25 );
    lasso_typeAppendString( token, v.data, v.dataSize );
  }

  if( lasso_findTagParam2( token, "-option2", &opt2 ) == osErrNoErr )
  {
    double  tempValue;
    char tempText[128];
    lasso_typeGetDecimal( token, opt2, &tempValue );
    sprintf( tempText, "%.15lg", tempValue );
    lasso_typeAppendString( token, " The value of -option2 is ", 26 );
    lasso_typeAppendString( token, tempText, strlen(tempText) );
  }

  int count = 0;
  lasso_getTagParamCount( token, &count );
```

```
  for (int i = 0; i < count; ++i)
  {
    lasso_getTagParam( token, i, &v );
    if ( v.dataSize == 0 )
    {
      lasso_typeAppendString( token, " The value of unnamed param is ", 31 );
      lasso_typeAppendString( token, v.name, v.nameSize );
    }
  }

  return lasso_returnTagValue(token, retString);
}
```

Substitution Tag Module Walk Through

This section provides a step-by-step walk through of the code for the substitution tag module.

To build a sample LCAPI tag module:

1 First, register the new tag in the required registerLassoModule() export function.

```
void registerLassoModule()
{
  lasso_registerTagModule( "sample", "tag", myTagFunc,
    REG_FLAGS_TAG_DEFAULT, "sample test" );
}
```

2 Implement myTagFunc, which gets called when [sample_tag] is encountered. All tag functions have this prototype. When the tag function is called, it's passed an opaque "token" data structure.

```
osError myTagFunc( lasso_request_t token, tag_action_t action )
{
```

The remainder of the code in the walk through includes the implementation for the myTagFunc function.

3 Allocate a string which will be this tag's return value.

```
lasso_type_t retString = NULL, opt2 = NULL;
lasso_typeAllocString(token, &retString, "", 0);
```

4 The auto_lasso_value_t variable named v will be our temporary variable for holding parameter values. Start off by initializing it.

```
auto_lasso_value_t  v;
INITVAL(&v);
```

5 Call lasso_FindTagParam() in order to get the value of the -option1 parameter. If it is found (no error while finding the named parameter), append some information about it to our return value string.

```
if( lasso_findTagParam( token, "-option1", &v ) == osErrNoErr )
{
  lasso_typeAppendString( token, "The value of -option1 is ", 25 );
  lasso_typeAppendString( token, v.data, v.dataSize );
}
```

6 Look for the other named parameter, -option2 and store its value into variable opt2. Because -option2 should be a decimal value, use lasso_findTagParam2, which will preserve the original data type of the value as opposed to converting it into a string like lasso_findTagParam will.

```
if( lasso_findTagParam2( token, "-option2", &opt2 ) == osErrNoErr )
{
```

7 Declare a temporary floating-point (double) value to hold the number passed in and then declare a temporary string to hold the converted number for display. Get the value of op2 as a decimal then print it to the tempText variable.

```
  double  tempValue;
  char tempText[128];
  lasso_typeGetDecimal( token, opt2, &tempValue );
  sprintf( tempText, "%.15lg", tempValue );
```

8 Append the parameter's information to the return string.

```
  lasso_typeAppendString( token, " The value of -option2 is ", 26 );
  lasso_typeAppendString( token, tempText, strlen(tempText) );
}
```

9 Now, we're going to look for the unnamed parameter. Because there's no way to ask for unnamed parameters, we're going to enumerate through all the parameters looking for one without a name. The integer count will hold the number of parameters found. Use lasso_getTagParamCount() to find out how many parameters were passed into our tag. The variable count now contains the number 3, if we were indeed passed three parameters.

```
int count = 0;

lasso_getTagParamCount( token, &count );

for (int i = 0; i < count; ++i) {
```

10 Use lasso_getTagParam() to retrieve a parameter by its index. If you design tags that require parameters to be in a particular order, then use this function to retrieve parameters by index, starting at index 0. If the parameter is unnamed, that means it's the one needed. Note that if the

user passes in more than one unnamed parameter, this loop will find all of them, and will ignore any named parameters.

```
lasso_getTagParam( token, i, &v );
if ( v.dataSize == 0 )
{
```

11 Again, append a descriptive line of text about the unnamed parameter and it's value. Notice that the name member of the variable is what holds the text we're looking for, and the data member is empty.

```
    lasso_typeAppendString( token, " The value of unnamed param is ", 31 );
    lasso_typeAppendString( token, v.name, v.nameSize );
  }
}
```

12 Returning an error code is very important. If you return a non-zero error code, then the interpreter will throw an exception indicating that this tag failed fatally and Lasso's standard page error routines will display an error message. For non-fatal errors, you can use lasso_setResultCode() and lasso_setResultMessage() to provide error codes for the caller; just make sure your tag function returns osErrNoErr from your function, otherwise Lasso's fatal error routines will be triggered.

```
  return lasso_returnTagValue(token, retString);
}
```

55

Chapter 55
LCAPI Data Types

This chapter includes information about creating data types in C/C++ using the Lasso C/C++ API (LCAPI).

- *Data Type Operation* discusses how new data types can be implemented in C/C++.
- *Data Type Tutorial* walks through a sample data type project.

Data Type Operation

Creating a new data type in LCAPI 8 is similar to creating a substitution tag. When Lasso Professional 8 starts up, it scans the LassoModules folder for module files (Windows DLLs or Mac OS X DYLIBS). As it encounters each module, it executes the registerLassoModule() function for that module. The developer registers the LCAPI data types or tags implemented by the module inside this function. Registering data type initializers differs from registering normal substitution tags in that the third parameter in lasso_registerTagMode is the value REG_FLAGS_TYPE_DEFAULT.

```
void registerLassoModule()
{
  lasso_registerTagModule( “test”, “type”, myTypeInitFunc,
    REG_FLAGS_TYPE_DEFAULT, “simple test LCAPI type” );
}
```

The prototype of a LCAPI type initializer is the same as a regular LCAPI substitution tag function. Lasso will call the type initializer each time a new instance of the type is created.

```
osError myTypeInitFunc( lasso_request_t token, tag_action_t action );
```

When the type initializer function is called, a new instance of the type is created using lasso_typeAllocCustom. This new instance will be created with no data or tag members.

```
osError myTypeInitFunc( lasso_request_t token, tag_action_t action )
{
  lasso_type_t theNewInstance = NULL;
  lasso_typeAllocCustom( token, &theNewInstance, “test_type” );
```

Once the type is created, new data and tag members can be added to it using lasso_typeAddMember. Data members can be of any type and should be allocated using any of the LCAPI type allocation calls. Tag members are allocated using lasso_typeAllocTag. LCAPI tag member functions are implemented just like any other LCAPI tag. In the example below, myTagMemberFunction is a function with the standard LCAPI tag prototype.

```
  const char * kStringData = “This is a string member.”;
  lasso_type_t stringMember = NULL;
  lasso_typeAllocString( token, &stringMember, kStringData, strlen(kStringData) );
  lasso_typeAddMember( token, theNewInstance, “member1”, stringMember );
  lasso_type_t tagMember = NULL;
  lasso_typeAllocTag( token, &tagMember, myTagMemberFunction );
  lasso_typeAddMember( token, theNewInstance, “member2”, tagMember );
```

The final step in creating a new LCAPI data type instance is to return the new type to Lasso as the tag's return value. After the type is returned, Lasso will complete the creation of the type by instantiating the new type's parent types.

```
  lasso_returnTagValue( token, theNewInstance );
  return osErrNoErr;
}
```

Data Type Tutorial

This tutorial walks through the main points of creating a custom data type using LCAPI 7. The resulting data type is a "file" type, and the ability to open, close, read and write to the file are implemented via the following member tags:

[File->Open] [File->Close] [File->Read] [File->Write]

Data Types Code

The example project and source files contain over 800 lines of code, and are located in the following folder:

Lasso Professional 8/Documentation/3 - Language Guide/Examples/LCAPI/Tags/CAPIFile

Do to the length of the project file (CAPIFile.cpp), the entire code is not shown here. The *Data Type Walk Through* section provides a conceptual overview of the operation behind the file type example, and describes the basic LCAPI functions used to implement it.

Note: This walk through is not fully up-to-date with the sample code in the documentation folder. The walk through should serve as a useful road map, but the sample code should be read separately to see how it has been updated for LCAPI 8.

Data Type Walk Through

This section provides a step-by-step conceptual walk through for building a custom file data type.

To build a custom data type:

1 The first step in creating a custom type is to register the type's initializer. Type initializers are registered in the same way that regular tag functions are registered. The only difference being that flag_typeInitializer should be passed for the fourth (flags) parameter.

This concept is illustrated in lines 95-129 of the CAPIFile.cpp file.

```
void registerLassoModule()
{
  ...
  lasso_registerTagModule("example", "file", file_init,
    REG_FLAGS_TYPE_DEFAULT, "Initializer for the file type.");
}
```

2 The registered type initializer will be called each time a new file type is created. In the above case, the LCAPI function file_init was registered as being the initializer. The prototype for file_init should look like any other LCAPI function.

This concept is illustrated in line 272 of the CAPIFile.cpp file.

```
osError file_init(lasso_request_t token, tag_action_t action)
```

3 The file_init function will now be called whenever the example_file type is used in a script. Within the type initializer, the type's member tags are added. Each member tag is implemented by its own LCAPI tag function. However, before members can be added, the new blank type must be created using lasso_typeAllocCustom.

lasso_typeAllocCustom can only be used within a properly registered type initializer. The value it produces should always be the return value of the tag as set by the lasso_returnTagValue function.

This concept is illustrated in lines 273-277 of the CAPIFile.cpp file.

```
{
  lasso_type_t file;
  ...
  lasso_typeAllocCustom(token, &file, KFileTypeName);
```

4 Once the blank type has been created, members can be added to it. LCAPI data types often need to store pointers to allocated structures or memory. LCAPI provides a means to accomplish this by using the lasso_setPtrMember and lasso_getPtrMember functions. These functions allow the developer to store a pointer with a specific name. The pointer is stored as a regular integer data member. The names of all pointer members should begin with an underscore. Naming a pointer as such will indicate to Lasso that it should not be copied when a copy is made of the data type instance. This LCAPI file type will store its private data in a structure called file_desc_t.

This concept is illustrated in lines 280-281 of the CAPIFile.cpp file.

```
file_desc_t * desc = new file_desc_t;
lasso_setPtrMember(token, file, kPrivateMember, desc);
```

5 Members are also added for open, close, read and write.

```
lasso_type_t mem;
lasso_typeAllocTag(token, &mem, file_open);
lasso_typeAddMember(token, file, "open", mem);

lasso_typeAllocTag(token, &mem, file_close);
lasso_typeAddMember(token, file, "close", mem);

lasso_typeAllocTag(token, &mem, file_read);
lasso_typeAddMember(token, file, "read", mem);
```

```
lasso_typeAllocTag(token, &mem, file_write);
lasso_typeAddMember(token, file, "write", mem);
```

This concept is illustrated in lines 286-295 of the CAPIFile.cpp file. The macro ADD_TAG is defined and used to avoid the more repetitive activities.

```
#define ADD_TAG(NAME, FUNC) {   lasso_type_t mem;\
  lasso_typeAllocTag(token, &mem, FUNC);\
  lasso_typeAddMember(token, file, NAME, mem);\
}

...

ADD_TAG(kMemOpen, file_open);
ADD_TAG(kMemClose, file_close);
ADD_TAG(kMemRead, file_read);
ADD_TAG(kMemWrite, file_write);
```

6 The final member tag to add is the onDestroy member. This tag will be called automatically by Lasso when the type goes away. Adding this tag will ensure that the file on disk is closed properly if the member tag function file_close is not called.

This concept is illustrated in line 309 of the CAPIFile.cpp file.

```
ADD_TAG(kMemOnDestroy, file_onDestroy);
```

7 At this point, the return value should be set. Keep in mind that the new file type is completely blank except for the members that were added above. No inherited members are available at this point. Inherited members are only added after the LCAPI type initializer returns.

This concept is illustrated in line 312 of the CAPIFile.cpp file.

```
lasso_returnTagValue(token, file);
```

8 There were no errors in the type initialization process, so return a "no error" code to Lasso, completing the type's initialization.

This concept is illustrated in line 313 of the CAPIFile.cpp file.

```
return osErrNoErr;
```

Note: For brevity, this example will not cover accepting parameters in the type's onCreate member tag. The full CAPIFile project illustrates accepting parameters in the onCreate member to open the file under various read and write permissions.

9 The new file type has now been initialized and made available to the caller in the script. The first member of the file type is [File->Open], which is implemented as the LCAPI function file_open.

This concept is illustrated in lines 365-366 of the CAPIFile.cpp file.

```
osError file_open(lasso_request_t token, tag_action_t action)
{
```

10 The first step in implementing a member tag is to acquire the "self" instance. The self is the instance upon which the member call was made.

This concept is illustrated in lines 367-370 of the CAPIFile.cpp file.

```
lasso_type_t self = NULL;
lasso_getTagSelf(token, &self);
if ( !self )
  return osErrInvalidParameter;
```

11 Once the self is successfully acquired and is not null, the rest of the member tag can proceed. This member tag accepts one parameter, which is the path to the file that will be opened. Since the path is a string value, it can be acquired using lasso_getTagParam. If the path parameter was not passed to the open member tag, an error should be returned and indicated to the user.

This concept is illustrated in lines 380-396 of the CAPIFile.cpp file.

```
// see what parameters we are being initialized with
int count;
lasso_getTagParamCount(token, &count);
if ( count < 2 )
{
  lasso_setResultMessage(token,
    "file->open requires at least a file path and open mode.");
  lasso_setResultCode(token, osErrInvalidParameter);
  return osErrInvalidParameter;
}
if ( count > 0 ) // we are given *at the least* a path
{
  // first param is going to be a string, so use the LCAPI 7 call to get it
  auto_lasso_value_t pathParam;
  pathParam.name = "";
  lasso_getTagParam(token, 0, &pathParam);
```

12 Now that the path parameter has been successfully acquired, permissions should be checked to make sure access to the file is permitted by Lasso security.

This concept is illustrated in lines 232-237 of the CAPIFile.cpp file.

```
if ( lasso_operationAllowed(token, op, const_cast<char*>(path)) != osErrNoErr )
{
  lasso_setResultMessage(token,
    "Permission to open the file was denied by Lasso security.");
  lasso_setResultCode(token, osErrNoPermission);
  return NULL;
}
```

13 If the current user has permission, the Lasso internal path should be converted to the platform specific path. This is a three-step process that begins with fully qualifying the path. This will ensure that relative paths are converted to root paths. The second step is to resolve the path. This converts root path to a complete path which will include the hard drive name, or /// if used on a Unix platform. The final step is to convert the path into a platform-specific format that will be understood by the platform-specific [File->Open] calls.

This concept is illustrated in lines 197-203 of the CAPIFile.cpp file.

```
{
  osPathname qualifiedPath;
  osPathname resolvedPath;
  lasso_fullyQualifyPath( token, inPath, qualifiedPath );
  lasso_resolvePath( token, qualifiedPath, resolvedPath );
  lasso_getPlatformSpecificPath( resolvedPath, outPath );
}
```

14 Once security is checked and the path is properly converted, the actual file can be opened using the file system calls supplied by the operating system.

This concept is illustrated in line 242 of the CAPIFile.cpp file.

```
FILE * f = fopen(xformPath, openMode);
```

15 The FILE pointer can now be retrieved using the lasso_typeGetCustomPtr LCAPI function. No error has occurred while opening the file, so complete the function call and return "no error".

This concept is illustrated in lines 426 of the CAPIFile.cpp file.

```
return osErrNoErr;
```

16 The remaining tag functions are implemented in a similar manner. Study the CAPIFile example for a more in-depth and complete example of how to properly construct custom data types in LCAPI 8.

56

Chapter 56
LCAPI Data Sources

This chapter includes information about creating data source connectors in C/C++ using the Lasso C/C++ API (LCAPI).

- *Data Source Connector Operation* discusses how data source connectors can be implemented in C/C++.
- *Data Source Connector Tutorial* walks through a sample project included with every Lasso installation.

Data Source Connector Operation

When Lasso Professional 8 starts up, it looks for module files (Windows DLLs or Mac OS X DYLIBS) in the LassoModules folder. As Lasso encounters each module, it executes the module's registerLassoModule() function once and only once. It is your job as an LCAPI developer to write code to register each of your new data source (or custom tag) function entry points in this registerLassoModule() function. Both substitution tags and data sources may be registered at the same time, and the code for them can reside in the same module. The only difference between registering a data source and a substitution tag is whether you call lasso_registerTagModule() or lasso_registerDSModule().

Data sources are a bit more complex than substitution tags because Lasso Service calls them with many different actions during the course of various database operations. Whereas a substitution tag only needs to know how to format itself, a data source needs to enumerate its tables, search through records, add new records, delete records, etc. Even so, this added complexity is easily handled with a single switch() statement, as you will see in the following tutorial.

Data Source Connectors and Lasso Administration

Once a custom data source connector module is registered by Lasso, it will appear in the *Setup > Data Sources > Connectors* section of Lasso Site Administration. If a connector appears here, then it has been installed correctly.

The administrator adds the data source connection information to the *Setup > Data Sources > Hosts* section of Lasso Site Administration, which sets the parameters by which Lasso connects to the data source via the connector. This information is stored in the Lasso_Internal Lasso MySQL database, where the connector can retrieve and use the data via function calls.

The data that the administrator can submit in the *Setup > Data Sources > Hosts* section of Lasso Site Administration includes the following:

- **Name** – The administrator-defined name of the data source host.
- **Connection URL** – The URL string required for Lasso to connect to a data source via the connector. This typically includes the IP address of the machine hosting the data source.
- **Connection Parameters** – Additional parameters passed with the Connection URL. This can include the TCP/IP port number of the data source.
- **Status** – Allows the administrator to enable or disable the connector in Lasso Professional 5.
- **Default Username** – The data source username required for Lasso to gain access to the data source.
- **Default Password** – The data source password required for Lasso to gain access to the data source.

The Connection URL, Connection Parameters, Default Username, and Default Password values are passed to the data source via the lasso_getDataHost function, which is described later in this chapter.

```
LCAPICALL osError lasso_getDataHost( lasso_request_t token,
    auto_lasso_value_t * host, auto_lasso_value_t * usernamepassword );
```

Data Source Connector Tutorial

This section provides a walk-through of an example data source to show how some of the LCAPI features are used. This code will be most similar to the sample SampleDataSource project, so if you want to actually build this code, then you should copy that project folder and edit the project files inside it.

The data source will simply display some simple text as each portion is called from an LDML inline which does a simple database search. It is not an effective or useful data source; it's meant to just provide an overview of what functions must be implemented. The sample data source will simulate a data source which has two databases, an Accounting database and a Customers database. Each of those databases in turn will report that it has a few tables within it. For a more complete example of a data source that is useful, look at the MySQLDataSource project.

Data Source Connector Code

Below is the code for the substitution tag module. Line numbers are provided to the left of each line of code, and are referenced in the *Data Source Connector Walk Through* section.

```
void registerLassoModule(
{
  lasso_registerDSModule( "SampleDatasource", sampleDS_func, 0 );
}
osError sampleDS_func( lasso_request_t token, datasource_action_t action, const
auto_lasso_value_t *param )
{
  osError   err = osErrNoErr;
  auto_lasso_value_t v1, v2;
  switch( action )
  {
    case datasourceInit:
      break;
    case datasourceTerm:
      break;
    case datasourceNames:
      lasso_addDataSourceResult( token, "Accounting" );
      lasso_addDataSourceResult( token, "Customers" );
      break;
    case datasourceExists:
      if( (strcmp( param->data, "Accounting" ) != 0)
        && (strcmp( param->data, "Customers" ) != 0) )
        err = osErrWebNoSuchObject;
      break;
    case datasourceTableNames:
    if( strcmp( param->data, "Accounting" ) == 0 ) {
        lasso_addDataSourceResult( token, "Payroll" );
        lasso_addDataSourceResult( token, "Payables" );
        lasso_addDataSourceResult( token, "Receivables" );
      }
      if( strcmp( param->data, "Customers" ) == 0 ) {
        lasso_addDataSourceResult( token, "ContactInfo" );
```

```
            lasso_addDataSourceResult( token, “ItemsPurchased” );
          }
          break;
        case datasourceSearch:
          lasso_getDataSourceName( token, &v1 );
          lasso_getTableName( token, &v2 );
          if( strcmp( v1.data, “Accounting” ) == 0 ) {
            int  count, i;
            lasso_getInputColumnCount( token, &count );
            for( i=0; i<count; i++ ) {
              auto_lasso_value_t      columnItem;
              lasso_getInputColumn( token, i, &columnItem );

            if( strcmp( v2.data, “Payroll” ) == 0 ) {
              char *row1[] = {“Samuel Goldwyn”, “1955-03-27”, “15000.00”};
              unsigned int sizes1[3] = {14, 10, 8};
              lasso_addColumnInfo( token, “Employee”, false, typeChar, kProtectionNone
);
              lasso_addColumnInfo( token, “StartDate”, false, typeDateTime,
kProtectionNone );
              lasso_addColumnInfo( token, “Wages”, false, typeDecimal,
kProtectionNone );
              lasso_addResultRow( token, (const char **)&row1, (unsigned int *)&sizes1,
(int)3 );
              lasso_setNumRowsFound( token, 1 );
            }
          }
          if( strcmp( v1.data, “Customers” ) == 0 ) {
          }
          break;
        case datasourceAdd:
          lasso_outputTagData( token, “datasourceAdd was called to append a
record<br>” );
          break;
        case datasourceUpdate:
          lasso_outputTagData( token, “datasourceUpdate was called to replace a
record<br>” );
          break;
        case datasourceDelete:
          lasso_outputTagData( token, “datasourceDelete was called to remove a
record<br>” );
          break;
        case datasourceInfo:
          lasso_outputTagData( token, “datasourceInfo was called<br>” );
          break;
        case datasourceExecSQL:
          lasso_outputTagData( token, “datasourceExecSQL was called<br>” );
```

```
        break;
    }
    return err;
}
```

Data Source Connector Walk Through

This section provides a step-by-step walk through for building the data source connector.

To build a sample LCAPI Data Source Connector:

1 Register the new data source in the required registerLassoModule() export function. It's similar to the way you register a substitution tag.

```
void registerLassoModule()
{
  lasso_registerDSModule( "SampleDatasource", sampleDS_func, 0 );
}
```

2 Now implement sampleDS_func, the function which gets called when any database operations are encountered.

```
osError sampleDS_func( lasso_request_t token, datasource_action_t action,
const auto_lasso_value_t *param )
```

All data source functions have this prototype. When your data source function is called, it's passed an opaque "token" data structure, an integer "action" telling it what it should do, and an optional parameter which sometimes contains extra information (like a database name) needed by the action being requested at that time.

3 Set a default error return value that indicates no error. Returning a non-zero value will cause the Lasso Professional engine to report a fatal error and stop processing the page.

```
{
  osError err = osErrNoErr;
  auto_lasso_value_t v1, v2;
  switch( action )
{
```

Declare a couple of temporary variables to be used later to retrieve important values such as database names and table names. This function gets called with various different actions as Lasso Professional requests information from our data source. This switch statement distinguishes between those various actions.

4 datasourceInit is called once when Lasso Professional starts up. This gives us a chance to initialize any communications with our database back-end, and set any global variables (including semaphores) we'll need later. This is called once when Lasso Professional starts up. Because this data source is so simple, it needs no special initialization calls.

```
case datasourceInit:
  break;
case datasourceTerm:
  break;
case datasourceNames:
    lasso_addDataSourceResult( token, "Accounting" );
    lasso_addDataSourceResult( token, "Customers" );
  break;
case datasourceExists:
  if( (strcmp( param->data, "Accounting" ) != 0)
    && (strcmp( param->data, "Customers" ) != 0) )
    err = osErrWebNoSuchObject;
  break;
```

datasourceTerm is called once when Lasso Professional shuts down. Because this data source is so simple, it needs no special shutdown code. Normally you would close your connection to your back-end data source and release any semaphores you created.

datasourceNames is called whenever Lasso Professional needs to get a list of databases which your data source provides access to. The developer must write code that discovers a list of all the databases your database 'knows about' and call lasso_addDataSourceResult() once for each found database, passing the name of the database. If the data source deals with five databases, then you would call lasso_addDataSourceResult() five times, once for each database name.

Because we are simulating a data source which knows about the Accounting and Customers databases, call lasso_addDataSourceResult() to add Accounting and Customers to the returned list of database names.

For datasourceExists, Lasso Professional is asking use if we know a particular database exists (meaning, do we control this database). The name of the database we should look up is passed in the C-string param->data. If we don't know about the database in question, then return osErrWebNoSuchObject. The conditional statement does a simple string comparison against our hard-coded database name Accounting, and then against our hard-coded database name Customers. If neither of the previous string comparisons matched, then return the error code osErrWebNoSuchObject indicating that we do not know anything about the requested database.

5 Lasso Professional will also need to call on the database tables once per database, passing the database name in the param->data value. datasourceTableNames enumerates the list of tables within that named database.

```
case datasourceTableNames:
  if( strcmp( param->data, “Accounting” ) == 0 ) {
    lasso_addDataSourceResult( token, “Payroll” );
    lasso_addDataSourceResult( token, “Payables” );
    lasso_addDataSourceResult( token, “Receivables” );
  }
```

The conditional statement checks to see if we are being asked about our Accounting database, and if so adds the Payroll table to the list of known tables by calling lasso_addDataSourceResult(), and so forth.

6 Next, Lasso Professional will need to check to see if there are inquiries regarding the Customers database.

```
if( strcmp( param->data, “Customers” ) == 0 ) {
  lasso_addDataSourceResult( token, “ContactInfo” );
  lasso_addDataSourceResult( token, “ItemsPurchased” );
}
  break;
```

Lasso Professional adds the ContactInfo table to the list of known tables by calling lasso_addDataSourceResult().Continue adding table names to the Customers database by calling lasso_addDataSourceResult(), this time for the ItemsPurchased table.

7 Use datasourceSearch to perform a search on the database.

```
case datasourceSearch:
  lasso_getDataSourceName( token, &v1 );
  lasso_getTableName( token, &v2 );
    if( strcmp( v1.data, “Accounting” ) == 0 ) {
      int  count, i;
      lasso_getInputColumnCount( token, &count );
        for( i=0; i<count; i++ ) {
        auto_lasso_value_t columnItem;
        lasso_getInputColumn( token, i, &columnItem );
        }
    }
```

All of the information (database and table names, search arguments, sort arguments, etc.) can be retrieved, and a search can be performed by calling various LCAPI functions such as lasso_getDataSourceName() and lasso_getTableName() to get the name of the database and table, respectively. A complete list of data source functions is here.

lasso_getDataSourceName asks Lasso Professional to give us the database name which is to be searched. This is often the value of the -Database parameter value in an inline tag. lasso_getTableName asks Lasso Professional to give us the table name to be searched. This is often the value from the -Layout or -Table parameter value from an inline tag.

The conditional statement checks to see if the database being searched is Accounting. If so, declare a couple of temporary integers, one for holding the number of search parameters. lasso_getInputColumnCount asks Lasso how many search fields (columns) were specified by the user for this search. For instance, if the LDML inline tag passed three different fields to be searched, then lasso_getInputColumnCount() returns 3.

Declare a temporary variable which will receive the name/value pair information from the next line of code. Retrieve the name/value text for the nth requested search parameter. For instance, an inline will fill the columnItem variable with the values Employee, fred the first time through the loop, and Wages, 15000 the second time through the loop.

```
[Inline: -Database='Accounting', -Table='Payroll', 'Employee'='fred',
'Wages'='15000']
```

8. Next, set a conditional statement to ask if the Payroll table is being searched. If so, we'll set up some fake hard-coded data in the next few lines of code. Declare an array of strings which represents the three fields we will return for this search. Declare an array of field sizes to match the lengths of the strings created on the previous line.

```
    if( strcmp( v2.data, "Payroll" ) == 0 ) {
      char *row1[] = {"Samuel Goldwyn", "1955-03-27", "15000.00"};
      unsigned int sizes1[3] = {14, 10, 8};
      lasso_addColumnInfo( token, "Employee", false, typeChar, kProtectionNone );
      lasso_addColumnInfo( token, "StartDate", false, typeDateTime,
kProtectionNone );
      lasso_addColumnInfo( token, "Wages", false, typeDecimal, kProtectionNone );
      lasso_addResultRow( token, (const char **)&row1, (unsigned int *)&sizes1(int)3
);lasso_setNumRowsFound( token, 1 );
    }
  }
  if( strcmp( v1.data, "Customers" ) == 0 ) {
  }
    break;
  case datasourceAdd:
    lasso_outputTagData( token, "datasourceAdd was called to append a
record<br>" );
    break;
  case datasourceUpdate:
      lasso_outputTagData( token, "datasourceUpdate was called to replace a
record<br>" );
```

```
        break;
    case datasourceDelete:
        lasso_outputTagData( token, "datasourceDelete was called to remove a
record<br>" );
        break;
    case datasourceInfo:
        lasso_outputTagData( token, "datasourceInfo was called<br>" );
        break;
    case datasourceExecSQL:
        lasso_outputTagData( token, "datasourceExecSQL was called<br>" );
        break;
    }
        return err;
    }
```

lasso_addColumnInfo tells LCAPI what the column names and data types are. Do this by calling lasso_addColumnInfo() once per column. In this line, the Employee column is described as text (typeChar) with no protection (kProtectionNone). In the next line, the StartDate column is described as date (typeDateTime) with no protection (kProtectionNone).

The last column Wages is described as being numeric (typeDecimal), with no protection (kProtectionNone). Now lasso_addResultRow() can be called as many times as there are rows of data to return. In this case, only one row is returned. Now LCAPI must be told how many total rows were found.

57

Chapter 57
Lasso Connector Protocol

This chapter documents Lasso Connector Protocol (LCP) and describes how to develop Lasso Web server connectors.

- *Overview* introduces Lasso Connector Protocol.
- *Requirements* includes platform specific development environment details.
- *Lasso Web Server Connectors* introduces the theory of operation behind creating Lasso Web server connectors using LCP.
- *Lasso Connector Operation* describes the theory and operation behind building Lasso Web server connectors.
- *Lasso Connector Protocol Reference* provides a reference of all commands and parameters used in LCP.

Overview

Lasso Web server connectors are small modules written specifically for a particular brand of Web server. Lasso Professional 8 initially includes connectors for Microsoft IIS (Intel architecture), Apple Mac OS X's Apache (PowerPC architecture), and 4D WebSTAR Server Suite V for Mac OS X. A connector for Red Hat Apache (Intel architecture) is also available.

The purpose of Lasso Connector Protocol (LCP) is to provide an efficient and platform-independent way of communication between a Lasso connector (client) and Lasso Service (server). Included are sample projects which give you full source code to the Web server connectors which ship with Lasso (e.g. Lasso Connector for IIS and Lasso Connector for Apache).

OmniPilot encourages developers to create and distribute new Web server connectors in order to give Lasso developers as many choices as possible for developing Lasso-based data-driven Web sites.

Requirements

In order to write your own Lasso Web server connector in C or C++, you will need the following:

Windows:

- Microsoft Windows 2000 or Windows XP Professional
- Microsoft Visual C++ .NET.
- Windows Lasso Professional version 8.0 or higher.

Mac OS:

- Mac OS X 10.3 with GNU C++ compiler and linker (Dev Tools) installed.
- Mac OS X Lasso Professional version 8.0 or higher.

Lasso Web Server Connectors

All modern Web servers have some form of suffix mapping, where they re-route HTTP requests to various modules based on their file suffix (e.g. .lasso). Modules have different names depending on which Web server you're using: ISAPI DLL, Apache Module, W*API plugin, etc. Once the Web server calls the Lasso Web server connector, it is the job of the Lasso Web server connector to collect all the information from a particular request, and pass it all along to a Lasso Service application that it's set up to talk to. Then it waits for Lasso Service to finish processing the request, and receives back some MIME headers and HTML body text. At this point it's the connector's job to pass the text back to the Web server, which in turn sends it back out to the requesting browser. All communication is via TCP/IP, so the Web server connector and Lasso Service may be on separate machines with different architectures.

Lasso Web server connectors also have another job, which is to decode and write out HTTP-upload files. As you examine the sample source code, you'll see that it interprets the incoming POST arguments, writes out temporary files, and passes a special list of filename arguments through to Lasso Service on the other side of the TCP connection.

Note: Only a single Lasso Web server connector can connect with Lasso Service at a time in Lasso Professional 8.

Getting Started

This section provides a walk-through for building a custom Web server connector in Windows 2000 and Mac OS X.

To build a sample Web server connector in Windows 2000/XP:

1. Browse to the Lasso Professional 8\Documentation\3 - Language Guide/Examples/LCAPI/Connectors/Lasso Connector for IIS folder on the hard drive.
2. Double-click the ISAPIConnector.sln project file — you need Microsoft Visual C++ .NET in order to open it.
3. Choose Build/Build Solution to compile and make the ISAPIConnector.dll.
4. After building, Debug and Release folders will have been created inside your ISAPIConnector project folder.
5. Open IIS Admin and shut down IIS (so that any previous ISAPIConnector.dll files will not be held open inside IIS).
6. Open the Lasso Connector for IIS/Debug folder and drag ISAPIConnector.dll into your Windows/System32 folder.
7. Restart IIS using the Services menu in the windows Control Panel.
8. Assuming you already have Lasso installed on this machine, your suffix mappings should all work, and Lasso should function just as it did before.
9. Use a Web browser to view http://your.Web.server/ and make sure the .lasso suffix mapping is still working.

To build a sample Web server connector in Mac OS X:

1. Open a Terminal window.
2. Change the current folder to the Documentation folder by entering the following:

```
cd /Applications/Lasso\ Professional\ 7/Documentation/3\ -\ Language\ Guide/Examples/LCAPI/Connectors/Lasso\ Connector\ for\ Apache
```

3. Build the sample project using the provided makefile. You must be logged in as the root user to run this command.

```
make
```

4. After building, a Mac OS X dynamic library file will be in the current folder: Lasso8ConnectorforApache.so. This is the module you'll install into the ApacheModules folder.

5 Copy the newly-created module to the LassoModules folder by entering the following:

```
cp LassoConnectorforApache.so /usr/libexec/httpd/
```

6 Logged in as root user, restart apache so it loads the new module.

```
su <enter root password here> apachectl restart
```

Assuming you already have Lasso installed on this machine, your suffix mappings should all work, and Lasso should function just as it did before.

7 In a Web browser, go to http://your.Web.server/ and try a few things to make sure the .lasso suffix mapping is still working.

Lasso Connector Operation

Communication between the Lasso Web server connector (client) and Lasso Service (server) is achieved by means of exchanging messages via a regular TCP/IP socket on port 14552. A typical session is initiated by a client and consists of the following steps:

1 Connect to Lasso Service host on port 14552.

2 Send the open request command.

3 Handle requests sent back from Lasso.

4 Repeat previous step until the close request command is received.

5 Close the connection.

All messages to and from Lasso Service begin with the LPCommandBlock structure, and are optionally followed by an arbitrary number of data bytes if needed. The LPCommandBlock structure is defined as follows:

```
typedef enum LPCommand;
typedef int LPRequestID;
typedef unsigned int LPSequenceNum;
typedef struct LPCommandBlock
{
  LPCommand  fCmd;
  int fResultCode;
  unsigned int fDataSize;
};
```

The meaning of each LPCommandBlock structure member is explained in the following table.

Table 1: LPCommandBlock Structure Members

Command	Description
fCmd	The command. For a list of currently defined commands see the Command Reference at the end of this chapter.
fResultCode	The result of the command. Used if the command is a reply.
fDataSize	The size of the additional command-specific data to follow (may be zero).

Lasso Connector Protocol Reference

LCP Commands

This section lists all of the commands used in LCP.

cmdProtoErr

Indicates that an error has occurred in the use of the protocol.

Data Required	Four-byte integer indicating the error code. Any additional data will be a textual description of what went wrong.
Sent By	Lasso Service
Reply	None

cmdCloseReq

Sent by Lasso Service when there is no more data to be sent to the Web browser.

Data Required	None
Sent By	Lasso Service or client.
Reply	None

cmdGetParam

Request to return the value of a "named" parameter - server/environment variable or an HTTP request.

Data Required	RequestParamKeyword as defined in RequestParams.h Then a four-byte integer indicating the size of the data for the argument. Multiple params may follow.
Sent By	Lasso Service
Reply	cmdGetParamRep

cmdGetParamRep

Returns the value of a "named" parameter, as requested by cmdGetParam command.

Data Required	RequestParamKeyword as defined in RequestParams.h, then a four-byte integer indicating the size of the character data for the requested param. If multiple params were requested, the data for each param should follow in the original order.
Sent By	client
Reply	None

cmdPushData

Push partially processed data to a Web browser.

Data Required	The data that should be sent to the web browser.
Sent By	LassoService
Reply	None

Named Parameters

The following table lists all named parameters used in LCP. These parameters are enumerated in the RequestParams.h file.

Table 2: Named Parameters

Parameter	Description
rpSearchArgKeyword	All text in URL after the question mark.
rpUserKeyword	Username sent from browser.
rpPasswordKeyword	Password sent from browser.
rpAddressKeyword	IP address of client browser.
rpPostKeyword	HTTP object body (form data, etc.).
rpMethodKeyword	GET or POST, depending on <form method>.
rpServerName	IP address of server on which the Web server is running.
rpServerPort	IP port this hit came to (80 is common, 443 for SSL).
rpScriptName	Relative path from server root to this Lasso format file.
rpContentType	MIME header sent from client browser.
rpContentLength	The length in bytes of the POST data sent from <form POST>.
rpReferrerKeyword	URL of referring page.
rpUserAgentKeyword	Browser name and type.
rpClientIPAddress	IP address of client browser.
rpFullRequestKeyword	All MIME headers, uninterpreted.

X

Section X
Lasso Java API

This section includes instructions for extending the functionality of Lasso by creating new tags, data types, and Web server connectors written in Java.

- *Chapter 58: LJAPI Introduction* includes general information about extending Lasso's functionality.
- *Chapter 59: LJAPI Tags* discusses how to create new tags in LJAPI including substitution tags, asynchronous tags, and remote procedures.
- *Chapter 60: LJAPI Data Types* discusses how to create new data types in LJAPI including sub-classing and symbol overloading.
- *Chapter 61: LJAPI Data Sources* includes information about how to create new data source in LJAPI.
- *Chapter 62: LJAPI Reference* includes information about each of the function calls available in LJAPI.

Lasso can also be extended using LDML or C/C++. See the preceding sections on the *LDML API* or the *Lasso C/C++ API* (LCAPI) for more information..

58

Chapter 58
LJAPI Introduction

This chapter provides an introduction to the Lasso Java API (LJAPI) which allows new tags, data types, and data source connectors to be written in Java.

- *Overview* introduces the Lasso Java API.
- *What's New* discusses what's new in this version of LJAPI.
- *LJAPI vs LCAPI* discusses when modules should be implemented in C/C++ versus Java.
- *Requirements* includes system requirements for building LJAPI modules.
- *Getting Started* includes basic information about how to build LJAPI modules.
- *Debugging* includes information about how to debug LJAPI modules within Lasso.
- *Frequently Asked Questions* includes several frequently asked questions and answers.

Overview

The Lasso Java Application Programming Interface (LJAPI) lets you write Java code to add new LDML tags, data source connectors, and data types to Lasso Professional 8. LJAPI is similar to LCAPI, but is tailored for the Java language.

Custom tags written in LJAPI instantly support each platform. One of the important reasons for developing LJAPI modules is an enormous class library included with each Java VM install, covering almost every single

programming need, from text processing to 2D/3D imaging to various network protocol implementations.

This chapter provides a walk-through for building an example substitution tag in LJAPI. Source code for the ZipCountTag module, as well as the code for the substitution tag, data source connector, and data type examples are included in the Lasso Professional 8/Documentation/4-ExtendingLasso/LJAPI folder on the hard drive.

What's New

Lasso Professional 8 includes some minor enhancements over the version of LJAPI that shipped with Lasso Professional 6. This section provides a quick summary of the history of LJAPI.

- **LJAPI** – LJAPI was introduced with Lasso Web Data Engine 3 in October 1998. Modules created using this version of LJAPI are generally compatible with all versions of Lasso from 3 through 6. This API is sometimes referred to as LJAPI 5.

  Lasso Professional 5 included some minor enhancements to LJAPI from Lasso WDE 3.x, but the API remained largly unchanged. Lasso Professional 5 also introduced the Lasso C/C++ API (LCAPI) for C/C++ programmers.

- **LJAPI 6** – Lasso Professional 6 included a complete rewrite of LJAPI. The most important change in LJAPI 6 is that it is now built upon LCAPI. Both API's share the same functionality and provide a single programming interface, making it easier for developers who wish to learn both APIs.

  Lasso Professional 6 shipped with support for both LJAPI 5 (and earlier) modules and LJAPI 6 modules.

- **LJAPI 7** – Lasso Professional 7 supports all modules created with LJAPI 6. There are some minor enhancements to the APIs, but no significant changes over the previous version. Lasso Professional 7 did not support modules written for LJAPI 5 or earlier.

- LJAPI 8 – Lasso Professional 8 supports all modules created with LJAPI 6 or 7. There are some minor enhancements to the APIs, but no significant changes over the previous version. Lasso Professional 8 does not support modules written for LJAPI 5 or earlier.

Each new release of Lasso brings enhancements to the LDML programming language, built-in data types, data sources, LCAPI, and more. While the basic API for LJAPI is not exepcted to change significantly post-LP6,

new releases of LJAPI may include support for any new features of Lasso or LCAPI that can be expressed in the API.

Modules written to the LJAPI 6 specifications should be compatible with Lasso Professional 8. Modules written to the LJAPI 7 specifications should be compatible with Lasso Professional 6 provided that no LP7-specific features are accessed. Modules written to the LJAPI 5 (or earlier) specifications will not work in Lasso Professional 8.

LJAPI vs. LCAPI

Developers who have experience creating LCAPI modules will find themselves familiar with the Lasso Java API. Similarly, Java developers who learn to use LJAPI 7 will find it easy to write LCAPI modules once they are ready to make a transition to a different language.

The following sections outlines a few basic differences between LCAPI 7 and LJAPI 7.

LJAPI is Object-Oriented

The majority of Lasso API functions must be aware of the current Lasso state in order to operate correctly. In order to solve the problem resulting from the non-OO nature of the C-based Lasso API, LCAPI introduced the token concept. When Lasso calls one of the methods implemented by an LCAPI module, it passes an opaque parameter of type lasso_request_t, which encapsulates the information about the current state of the request. The module then makes calls to Lasso while passing the token in the first parameter to every API function.

In LJAPI 7, the same state information is stored in an instance of the LassoCall Java class. All LJAPI 7 functions are implemented as members of the LassoCall class, which eliminates the need to pass a token parameter with each call.

This results in one of the most notable differences between LJAPI and LCAPI, in that LJAPI methods usually take one parameter less that their native LCAPI counterparts.

LJAPI Uses Shorter Function Names

In LCAPI, function names begin with the lasso_ prefix, reflecting the name space in which they reside. However, the corresponding LJAPI methods are implemented as members of LassoCall class. For this reason, the lasso_ prefix has been removed from all Java method names.

The following shows the lasso_getTagName function in LCAPI:

```
lasso_getTagName( lasso_request_t token, auto_lasso_value_t &name );
```

The following shows the equivalent getTagName method in LJAPI:

```
getTagName( LassoValue name );
```

Tokenless LCAPI Functions are Static Methods in LJAPI

There are few LCAPI functions that do not take the token state parameter. These functions are implemented in LJAPI as static methods of the LassoCall class:

The following shows the lasso_registerConstant function in LCAPI:

```
osError lasso_registerConstant( const char * name, lasso_type_t val );
```

The following shows the equivalent registerConstant method in LJAPI:

```
int LassoCall.registerConstant( String name, LassoTypeRef val );
```

LJAPI Does Not Use Function Pointers

Some LCAPI functions use a function pointer parameter of type lasso_tag_func. Since function pointers do not exist in Java, the corresponding LJAPI methods instead accept a pair of string parameters that specify the class and method name.

The following shows the lasso_typeAllocTag function in LCAPI:

```
osError lasso_typeAllocTag ( lasso_request_t token, lasso_type_t * outTag, lasso_
tag_func nativeTagFunction );
```

The following shows the equivalent typeAllocTag method in LJAPI:

```
int typeAllocTag ( LassoTypeRef outTag, String className, String methodName );
```

Requirements

In order to write your own LDML substitution tags, data source connectors, or custom data types in Java, you will need the following:

Windows

- Microsoft Windows 2000, Microsoft Windows XP Professional, or better.
- Java 2 SDK 1.4 or higher.
- Windows Lasso Professional 8 or higher.

Mac OS

- Mac OS X with Java 2 SDK installed (included).
- Mac OS X Lasso Professional 8 or higher.

Getting Started

This section provides a walk-through for building sample LJAPI tag modules in Windows 2000/XP and Mac OS X.

To build a sample LJAPI tag module using Apache Ant:

Apache Ant is a de-facto standard Java-based build tool, part of the Apache open-source initiative.

In order to build the sample code, you will need to install complete Ant package, downloadable from the following location:

http://ant.apache.org/

If you do not wish to install Apache Ant at this time, you can skip to the next section for instructions on building the code examples with the javac compiler tool.

Note: All LJAPI examples have been tested with the most recent stable version of the Ant tool (v1.5.2) available at the time of the Lasso Professional 8 release.

To build all included code examples:

1. Launch the command prompt (Windows 2000/XP), or open the Terminal application (Mac OS X)
2. Locate the following folder in the hard drive:

   Lasso Professional 8/Documentation/4-ExtendingLasso/LJAPI/Sample Code

3. Make this folder your current working directory.

   Windows:

   cd "C:\Program Files\OmniPilot Software\Lasso Professional 8\Documentation\4-ExtendingLasso\LJAPI\Sample Code"

   Mac OS X:

   cd "/Applications/Lasso Professional 8/Documentation/4-ExtendingLasso/LJAPI/Sample Code"

4. Invoke Ant tool by entering the "ant" command at the command prompt, optionally followed by the target (sub-project) name:

   ant <target-name>

Compiled LJAPI modules will be placed in the Modules (output) folder located inside the Sample Code directory.

5 To install sample LJAPI modules using the Ant tool, enter the following command at the command prompt:

```
ant install
```

Sample LJAPI modules can also be installed manually, by dragging one or more Java class/jar files from the Modules (output) folder to the LassoModules folder.

6 Restart Lasso Professional.

Please note that, when launched without an optional target name parameter (step 4), Ant will execute the default target defined in the "build.xml" descriptor file. This target has been pre-configured to compile all sample LJAPI modules. Individual modules can be also built separately by specifying one of the following target names on the command line: zipcount, zip, pdf, nntp, mysql, xml or docs.

Two special targets (clean and install) can be used for deleting the contents of the Modules (output) directory, and copying LJAPI modules to the LassoModules folder, respectively.

For further details, please see the contents of the build.xml descriptor file.

Alternately, you can also build the ZipCountTag module using the <javac> command-line tool included with Java SDK from Sun Microsystems.

To build ZipCountTag module using the <javac> command-line tool:

1 Launch the Windows 2000/XP command prompt.

2 Make the following folder your current directory.

```
C:\Program Files\OmniPilot Software\Lasso Professional 8\
Documentation\4-ExtendingLasso\LJAPI\Sample Code\Substitution Tags\ZipCountTag
```

3 Enter the path of the Java compiler tool javac, followed by the -classpath option keyword and the path to the LJAPI.jar file (contains all Java classes used by LJAPI modules), followed by the ZipCountTag module source file path:

```
javac -classpath ../../../../../../LassoModules/LJAPI.jar ZipCountTag.java
```

If Java SDK has been installed in the jdk1.4 folder, your command line might look like this:

```
C:\jdk1.4\bin\javac -classpath ..\LJAPI.jar ZipCountTag.java
```

4 After building, a ZipCountTag.class file will be created inside your ZipCountTag project folder.

5 Open the ZipCountTag folder and drag ZipCountTag.class into the Lasso Professional 8\LassoModules folder on the hard drive.

6 Stop and then restart Lasso Service.

7 The new tag [Zip_Count] is now part of the LDML language.

8 Drag the sample Lasso format file called ZipCountTag.lasso and the LJAPITest.zip test file into your Web server root.

9 In a Web browser, view http://localhost/ZipCountTag.lasso to see the new LDML tags in action.

To build a sample LJAPI tag module in Mac OS X:

1 Open a Terminal window.

2 Change the current folder to the Lasso Professional 8/Documentation folder using the following command:

```
cd /Applications/Lasso\ Professional\ 7/Documentation/4-ExtendingLasso/LJAPI/
Sample\ Code/Substitution\ Tags/ZipCountTag
```

3 Build the sample project using the provided makefile. This requires that you be logged in as the root user.

```
make
```

Alternatively, you can build the module by manually invoking the Java compiler:

```
javac -classpath ../../../../../../LJAPI.jar ZipCountTag.java
```

4 After building, a Java class file named ZipCountTag.class will be created in the current folder. This is the LJAPI module you'll install into the LassoModules folder.

5 Copy the newly-created module to the Lasso modules folder using the following command:

```
cp ZipCountTag.class /Applications/Lasso\ Professional\ 7/LassoModules
```

6 Quit Lasso Service if it's running, so that the next time it starts up, it will load the new module you just built (you'll need to know a root password to use sudo).

```
cd /Applications/Lasso\ Professional\ 7/Tools/
sudo ./stoplassoservice.command
```

7 Start the Lasso Service back up, so it will load the new module.

```
sudo ./startlassoservice.command
```

The new [Zip_Count] tag is now part of the LDML language.

8 Copy the sample Lasso format file called ZipCountTag.lasso and the LJAPITest.zip test file from your Lasso Professional 8/Documentation/4-ExtendingLasso/LJAPI/Tags/ZipCountTag folder into your Web server document root.

9 Use a Web browser to view http://localhost/ZipCountTag.lasso to see the new LDML tags in action.

Debugging

You can set breakpoints in your LJAPI class files and perform source-level debugging for your own code. In order to set this up, add path information to your project so it knows from where to load executables. For this section, we will use the provided substitution tag project as the example.

To set breakpoints in your LJAPI code:

1 Lasso Professional 8 allows you to specify Java Virtual Machine options used for launching JVM upon Lasso startup. These options are stored in the lasso_internal.global_prefs table as java_vm_options in the store_key field. To enable remote debugging on port 8000, add the following two options to the data column in the lasso_internal.global_pref table:

```
-Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n
```

2 After restarting Lasso Professional 8, launch JDB with the following option:

```
jdb -attach 8000
```

3 Once attached to the JVM, you can set the breakpoints, single-step through your code, catch exceptions, etc. Please note that you can store multiple JVM options in the same column. To monitor the GC activity, add -verbose:gc option, or use -verbose:jni to print JNI messages to the standard output.

For more information on the options available for your platform and JVM, please consult the JVM vendor documentation. For a list of non-standard options available for your JVM, review the Xusage.txt file:

Mac OS X:

```
/System/Library/Frameworks/JavaVM.framework/Home/lib/Xusage.txt
```

Windows:

```
<path-to-jvm.dll>/Xusage.txt
```

59

Chapter 59
LJAPI Tags

This chapter includes information about creating tags in Java using the Lasso Java API (LJAPI).

- *Substitution Tag Operation* discusses how to create new LDML tags in Java.
- *Substitution Tag Tutorial* walks through an example project that ships with every installation of Lasso.

Substitution Tag Operation

An LJAPI module is essentially a regular Java class file. When Lasso Professional 8 first starts up, it looks for module files (Windows DLLs or Mac OS X DYLIBS) in its LassoModules folder. As it encounters and loads an LJAPI 7 module, it launches the JVM and proceeds to scan the folder for other LJAPI modules. Upon finding a Java class file, Lasso attempts to determine if it is derived from the com.omnipilot.lassopro.JavaModule class. If it is, then Lasso loads the class while performing necessary instantiation and calls the registerLassoModule() function that is implemented in that class:

```
public void registerLassoModule()
```

At this point, the module must call the following method as many times as needed, once for each tag implemented by the module:

```
void registerTagModule( String moduleName,
                        String tagName,
                        String methodName,
                        int flags,
                        String description);
```

After a tag module is registered with Lasso Professional 8, it can provide information about the name of the tag and the name of the Java method that is implementing that tag. It also can provide a short description, and any special flags describing unique features implemented by that tag.

All registered information is later used for dispatching the task of executing a particular tag found in a .lasso format file to an appropriate LJAPI module, or executing a data source action.

For example, the following code tells Lasso to call the Java class called ZipCountTag whenever the LDML [Zip_Count] is encountered inside a .lasso format file. The first parameter of the registerTagModule method is the module name, the second is the tag name, and the third one is the name of the function implementing the tag. The last two parameters are the tag type flag and a short description:

```
public void registerLassoModule()
  {
    registerTagModule( "ZipCountTag", "zip_count", "myZipCountFunc",
    FLAG_SUBSTITUTION, "Count items in a zip file" );
  }
```

Below is the LDML needed in a Lasso format file in order to get the custom tag to execute:

```
<html>
  <body>
    Count of items in the LjapiTest.zip file:
    [Zip_Count:'LjapiTest.zip']
    <!-- This should display "2" when page executes -->
  </body>
</html>
```

This will produce the following:

```
➜ 2
```

Substitution Tag Tutorial

The following section provides a walk-through of building an example tag to show how LJAPI features are used. This code will be most similar to the sample ZipCountTag LJAPI project. In order to build this project, copy the ZipCountTag project folder and edit the project files inside it.

The module relies on a Java class library to do most of the work, particularly the java.util.zip package which provides a variety of functions for manipulating the contents of Zip files—standard compressed archives widely used on the Internet.

The [Zip_Count] tag implemented in the ZipCountTag LJAPI module simply displays the number of files and directories stored in a Zip file when called from an LDML format file.

Example sample tag LDML syntax:

```
[Zip_Count: -Zipfile='LJAPITest.zip', -FilesOnly]
```

Notice the required convention of placing a dash in front of all named parameters in order to make them easier to spot in the LDML code, and prevent ambiguities in the LDML parser. Notice the tag takes one string parameter named -Zipfile, and an optional keyword parameter named -FilesOnly. In general, LDML does not care about the order in which you pass parameters, so plan to make this tag as flexible as possible by not assuming anything about the order of parameters. The following variations should work exactly the same.

Example of sample tag with different ordered parameters:

```
[Zip_Count: -Zipfile='LJAPITest.zip', -FilesOnly]

[Zip_Count: -FilesOnly, -Zipfile='LJAPITest.zip']
```

Substitution Tag Module Code

Shown below is the code for the substitution tag module. Line numbers are provided to the left of each line of code, and are referenced in the *Substitution Tag Module Walk-Through* section.

Note: The line numbers shown refer to the line numbers of the code in the actual file being created, not as shown in this page. Some single lines of code may carry into two or more lines as shown on this page.

Substitution Tag Module Code

```
1  import com.omnipilot.lassopro.*;
2  import java.io.*;
3  import java.util.*;
4  import java.util.zip.*;
5  public class ZipCountTag extends LassoTagModule
6  {
7  public void registerLassoModule()
8  {
9    registerTagModule( "Zip", "zip_count", "myZipCountFunc",
10   FLAG_SUBSTITUTION, "Count items in a zip file" );
11 }
12
13   public int myZipCountFunc(LassoCall lasso, int action)
```

```
14    {
15      int err = ERR_NOERR;
16      try {
17        IntValue count = new IntValue();
18        err = lasso.getTagParamCount( count );
19        if ( err == ERR_NOERR && count.intValue() > 0 )
20        {
21          String zipName = null;
22          boolean filesOnly = false;
23          LassoValue param1 = new LassoValue();
24          LassoValue param2 = new LassoValue();
25          err = lasso.findTagParam( "-zipfile", param1 );
26          if ( err != ERR_NOERR || param1.name() == null )
27            lasso.getTagParam( 0, param1 );
28          if ( param1.name() == null || param1.name().length() == 0 )
29            return LassoErrors.InvalidParameter;
30          if ( count.intValue() > 1 &&
31              lasso.getTagParam( 1, param2 ) == ERR_NOERR )
32            filesOnly = param2.equalsIgnoreCase("-filesonly");
33          String filePath = lasso.fullyQualifyPath( param1.name() );
34          filePath = lasso.resolvePath( filePath );
35          filePath = lasso.getPlatformSpecificPath( filePath );
36          ZipFile zip = new ZipFile( filePath );
37          Enumeration enum = zip.entries();
38          ZipEntry entry = null;
39          int zipcount = 0;
40          while ( enum.hasMoreElements() )
41          {
42            entry = (ZipEntry)enum.nextElement();
43            if ( !filesOnly || !entry.isDirectory() )
44              ++zipcount;
45          }
46          err = lasso.outputTagData( Integer.toString( zipcount ) );
47          zip.close();
48        }
49      }
50      catch ( java.io.Exception e )
51      {
52        lasso.setResultMessage( e.getMessage() );
53        return LassoErrors.FileNotFound;
54      }
55      return err;
56    }
57  }
```

Substitution Tag Module Walk-Through

This section provides a step-by-step walk-through for building the substitution tag module.

To write a sample LJAPI tag module:

1 First, import com.omnipilot.lassopro.* classes as shown in line 1.

```
1  import com.omnipilot.lassopro.*;
2  import java.io.*;
3  import java.util.*;
4  import java.util.zip.*;
```

2 Define your class to be a subclass of the omnipilot.lasso.LassoSubstitutionTag class.

```
5  public class ZipCountTag extends LassoTagModule
```

3 Define the registerLassoModule method.

```
7  public void registerLassoModule() {
```

Every Lasso module must implement the registerLassoModule() method. This method will be called by Lasso at startup, giving your module a chance to register its tags.

4 Register the tags implemented by your module.

```
9  registerTagModule( "Zip", "zip_count", "myZipCountFunc",
10 FLAG_SUBSTITUTION, "Count items in a zip file" );
```

Call this method as many times as there are tags implemented in your module. This method takes five parameters: the module name, the name of LDML tag, the name of the Java method implemented by your module (to be called when the corresponding LDML tag is found on the page), any additional tag feature flags, and a brief tag description.

5 Define the tag formatting method with the same name as indicated in the third parameter of the corresponding registerTagModule call.

```
13 public int myZipCountFunc(LassoCall lasso, int action)
```

This is the method that does all the work. Every tag registered by your module can have its own formatting method. Its purpose is to perform an action based on the parameters passed to the tag and/or current request properties. Most substitution tags would output the data, although some may perform other actions such as setting page variables, manipulating files, etc.

When Lasso encounters one of the tags registered by your module, it creates new module instance and calls the corresponding method, passing the LassoCall object which then can be used by the module for calling back into Lasso.

6 Define the variable to hold the result code returned by various LassoCall methods.

```
15  int err = ERR_NOERR;
```

7 Our [Zip_Count] LDML tag takes one required and one optional parameter. We need to make sure at least one parameter (filename) is present, otherwise we won't be able to continue.

```
17  IntValue count = new IntValue();
18  err = lasso.getTagParamCount( count );
19  if ( err == ERR_NOERR && count.intValue() > 0 )
```

8 Define the storage for the zip file name, optional -FilesOnly parameter, and LassoValue object to be used with various LassoCall methods.

```
21  String zipName = null;
22  boolean filesOnly = false;
23  LassoValue param = new LassoValue();
```

9 Our tag should be flexible enough to accept both named and unnamed versions of the required parameter. First, try to search for the parameter by a name.

```
25  err = lasso.findTagParam( "-zipfile", param1 );
```

10 If this fails, assume the first unnamed tag parameter to hold the file path name. Call getTagParam() with the index 0 (tag parameter numbering is zero-based).

```
26  if ( err != ERR_NOERR || param1.name() == null )
27    err = lasso.getTagParam( 0, param1 );
```

11 Next, make sure we've got a valid value. If the filename parameter contains an empty string, immediately return from our method, passing InvalidParameter result code back to Lasso.

```
28  if ( err != ERR_NOERR || param1.name().length() == 0 )
29    return LassoErrors.InvalidParameter;
```

12 Our tag also accepts an optional boolean parameter -FilesOnly, indicating that directories must be ignored while counting zip file items. If more than one parameter was supplied to our tag, try determining if it was the optional -FilesOnly parameter.

```
30  if ( count.intValue() > 1 &&
31       lasso.getTagParam( 1, param2 ) == ERR_NOERR )
32    filesOnly = param2.equalsIgnoreCase("-filesonly");
```

13 The path to the zip file is relative to the server root. In order to find out the actual location of the file, you can use a number of LassoCall class methods suited for converting a file path name into a fully qualified platform-specific path. fullyQualifyPath() turns a relative path into a from-the-server-root path. resolvePath() converts a from-the-root path into a full

internal path. Finally, getPlatformSpecificPath() will convert an internal path name into a platform-specific path name.

```
33 String filePath = lasso.fullyQualifyPath( param1.name() );
34 filePath = lasso.resolvePath( filePath );
35 filePath = lasso.getPlatformSpecificPath( filePath );
```

14 Now attempt to instantiate a ZipFile object using a platform-specific path name. Any exceptions thrown by the object constructor will be caught by the try/catch block wrapping our method's body.

```
36 ZipFile zip = new ZipFile( filePath );
```

15 Prepare to enumerate items in the zip file.

```
37 Enumeration enum = zip.entries();
```

16 Define the storage for holding the zip item count.

```
38 ZipEntry entry = null;
39 int zipcount = 0;
```

17 Iterate through the zip archive items, incrementing the counter for all items matching our criteria.

```
40 while ( enum.hasMoreElements() )
41 {
42   entry = (ZipEntry)enum.nextElement();
43   if ( !filesOnly || !entry.isDirectory() )
44     ++zipcount;
45 }
```

18 Output the resulting zip file item count.

```
46 err = lasso.outputTagData( Integer.toString( zipcount ) );
```

19 Close the zip file.

```
47 zip.close();
```

20 Make sure that any possible exceptions are handled correctly in your code. In this particular case, we simply pass the message retrieved from the Exception object back to Lasso, and return the FileNotFound error code. For a complete listing of error codes, see the variables defined in the LassoErrors class.

```
50 catch ( Exception e )
51 {
52   lasso.setResultMessage( e.getMessage() );
53   return LassoErrors.FileNotFound;
54 }
```

60

Chapter 60
LJAPI Data Types

This chapter includes information about creating data types in Java using the Lasso Java API (LJAPI).

- *Data Type Operation* discusses the fundamentals of implementing data types in Java.
- *Data Type Tutorial* walks through a sample project that is installed with Lasso Professional.

Data Type Operation

Among other new features, Lasso Professional 8 Java API introduces the ability to create custom data types in Java. Creating a new data type in LJAPI 7 is similar to creating a substitution tag. When Lasso Professional 8 starts up, it scans the LassoModules folder for module files (Windows DLLs or Mac OS X DYLIBS). As it encounters each module, it executes the registerLassoModule() function for that module. The developer may register new LJAPI data types implemented by the module inside this function.

Custom data types are analogous to objects used in many other programming languages. They can have properties (fields) and member tags (methods).

Data Type Tutorial

The following section provides a walk-through of building an example custom type to show how LJAPI features are used. This code will be most similar to the sample ZipType LJAPI project, so in order to build this code, copy the ZipType project folder and edit the project files inside it.

The module relies on a Java class library to do most of the work, particularly the java.util.zip package which provides variety of functions for manipulating the contents of ZIP files—standard compressed archives widely used on the Internet.

The resulting type will be a "zip" file with the ability to read data from a zip file given a path. The following member tags will be implemented:

Table 1: Type initializer and Member Tags

Name	Description
[Zip:'Pathname']	Type initializer. Creates new instance of a custom type.
[Zip->File]	Return the name of this Zip file.
[Zip->Count]	Return the count of entries in this file.
[Zip->Size]	Synonym for [Zip->Count].
[Zip->Enumerate]	Enumerates zip entries, allowing to iterate through stored items via consecutive calls to [Zip->Next].
[Zip->Next]	Advance to the next entry, returning True if more items are available.
[Zip->Position]	Current iterator position, i.e. the index.

The rest of the member tags are item accessors, operating on the entries stored in a zip file:

Table 2: Accessors

Name	Description
[Zip->Name]	Returns the name of an indexed entry.
[Zip->Get]	Synonym for [Zip->Name].
[Zip->Comment]	Zip entry comment.
[Zip->Date]	Returns the entry creation date.
[Zip->Crc]	Checksum, or 0xffffffff if not available.
[Zip->Method]	Compression method: DEFLATED or STORED.
[Zip->Extra]	Returns any extra data stored with the entry.
[Zip->GetData]	Returns uncompressed entry data.
[Zip->CSize]	Returns the size of the compressed data.
[Zip->USize]	Returns the size of uncompressed data.
[Zip->IsDir]	Returns True if the entry is a directory.

All zip entry accessor tags, except for [Zip->GetData], can take either one or zero parameters. An integer parameter can specify the index (position) of the entry in a zip file, while a string parameter could be used to locate an entry by its name. When no parameters are provided, a corresponding

action is performed on the "current" item, whose index can be obtained via the [Zip->Position] member tag.

Example sample tag LDML syntax:

The following shows an example of using a Zip custom type.

```
[Var:'zip' = zip:'/archive.zip']
[$zip->Count]
[$zip->Method]
[$zip->CSize]
[$zip->USize]
[While: $zip->Next]
  [$zip->CRC]
[/While]
```

Custom Data Type Module Code

Shown below is the code for the custom type tag module. Line numbers are provided to the left of each line of code, and are referenced in the *Custom Type Tag Module Walk-Through* section.

Note: The line numbers shown refer to the line numbers of the code in the actual file being created, not as shown in this page. Some single lines of code may carry into two or more lines as shown on this page.

Custom Data Type Module Code

```
 1  import com.omnipilot.lassopro.*;
 2  import java.util.*;
 3  import java.util.zip.*;
 4  import java.io.*;
 5  import java.text.DateFormat;
 6  public class ZipType extends LassoTagModule
 7  {
 8  static final DateFormat df =
 9    DateFormat.getDateTimeInstance(DateFormat.SHORT, DateFormat.MEDIUM);
10  static final String[] members = {
11    "File","Size","Count","Enumerate","Position","Next",
12    "GetData","Get","Name","Comment","Date","Crc",
13    "Method", "Extra", "CSize", "USize", "IsDir" };
14  ZipFile    zip = null;
15  Enumeration enum = null;
16  ZipEntry    entry = null;
17  int     index = 0;
18  public void registerLassoModule()
19  {
20    registerTagModule("ZipType", "zip", "format",
21         FLAG_SUBSTITUTION | FLAG_INITIALIZER, "zip custom type tag");
```

```
22  }
23  public int format(LassoCall lasso, int action)
24  {
25    int err = ERR_NOERR;
26    LassoValue param = new LassoValue();
27    String path name = null;
28    if (lasso.getTagParam(0, param) != ERR_NOERR ||
29       param.name().length() < 1)
30    {
31      lasso.setResultMessage("[Zip] invalid file path name parameter");
32      return LassoErrors.InvalidParameter;
33    }
34    try
35    {
36      IntValue count = new IntValue();
37      err = lasso.getTagParamCount(count);
38      if (err == ERR_NOERR && count.intValue() > 0)
39      {
40        String filePath = lasso.fullyQualifyPath(param.name());
41        filePath = lasso.resolvePath(filePath);
42        filePath = lasso.getPlatformSpecificPath(filePath);
43        this.zip = new ZipFile(filePath);
44        LassoTypeRef self = new LassoTypeRef();
45        if ((err = lasso.typeAllocCustom(self, "zip")) != ERR_NOERR)
46        {
47          lasso.setResultMessage("[Zip] couldn't create new zip type instance.");
48          return err;
49        }
50        LassoTypeRef ref = new LassoTypeRef();
51        String className = this.getClass().getName();
52        for (int i = 0; i < this.members.length; i++)
53        {
54          if ((err=lasso.typeAllocTag(ref, className, "memberFunc")) != ERR_NOERR ||
55             (err=lasso.typeAddMember(self, members[i], ref)) != ERR_NOERR)
56          {
57            lasso.setResultMessage("[Zip] error adding member: " + members[i]);
58            return err;
59          }
60        }
61        if (lasso.typeAllocTag(ref, className, "convertFunc") == ERR_NOERR)
62          lasso.typeAddMember(self, "onConvert", ref);
63        if (lasso.typeAllocTag(ref, className, "destroyFunc") == ERR_NOERR)
64          lasso.typeAddMember(self, "onDestroy", ref);
65        if ((err = lasso.typeSetCustomJavaObject(self, this)) != ERR_NOERR)
66        {
67          lasso.setResultMessage("[Zip] couldn't attach java object to a custom type");
68          return err;
69        }
```

```
70          err = lasso.returnTagValue(self);
71        }
72      }
73      catch (Exception e)
74      {
75        System.err.println(e.toString());
76        lasso.setResultMessage(e.getMessage());
77        return LassoErrors.FileNotFound;
78      }
79      return err;
80    }
81    public int destroyFunc(LassoCall lasso, int action)
82    {
83      if (this.zip != null)
84      {
85        try { this.zip.close(); }
86          catch (IOException e) {}
87        this.zip = null;
88      }
89      return ERR_NOERR;
90    }
91    public int convertFunc(LassoCall lasso, int action)
92    {
93      LassoValue param = new LassoValue();
94      if (lasso.getTagParam(0, param) == ERR_NOERR &&
95        param.name().equalsIgnoreCase("string"))
96      {
97        lasso.outputTagData("zip:(" + this.zip.getName() + ")");
98      }
99
100   return ERR_NOERR;
101   }
102   public int memberFunc(LassoCall lasso, int action)
103   {
104   LassoValue tag = new LassoValue();
105   LassoTypeRef out = new LassoTypeRef();
106   int err = lasso.getTagName(tag);
107   if (err != ERR_NOERR || tag.data().length() < 1)
108     return LassoErrors.InvalidParameter;
109   if (tag.data().equalsIgnoreCase("file"))
110     return lasso.outputTagData(zip.getName());
111   else if (tag.data().equalsIgnoreCase("size") ||
112         tag.data().equalsIgnoreCase("count"))
113   {
114     lasso.typeAllocInteger(out, zip.size());
115     return lasso.returnTagValue(out);
116   }
117   LassoValue param = new LassoValue();
```

```
ZipEntry item = this.entry;
if (lasso.getTagParam(0, param) == ERR_NOERR)
{
  if (param.type() == LassoValue.TYPE_INT)
  {
    try {
      int idx = Integer.parseInt(param.name());
      if (idx < 1 || idx > zip.size())
      {
        lasso.setResultMessage("[Zip] index out of range: " + idx);
        return LassoErrors.InvalidParameter;
      }
      else if (idx != index)
      {
        index = idx;
        Enumeration enum2 = zip.entries();
        while (enum2.hasMoreElements() && idx-- > 0)
          item = (ZipEntry)enum2.nextElement();
        entry = item;
      }
    } catch (NumberFormatException npe) {}
  }
  else if (param.type() == LassoValue.TYPE_CHAR)
    item = zip.getEntry(param.name());
}
String result = null;
if (tag.data().equalsIgnoreCase("name") ||
   tag.data().equalsIgnoreCase("get"))
  result = item.getName();
else if (tag.data().equalsIgnoreCase("comment"))
  result = item.getComment();
else if (tag.data().equalsIgnoreCase("crc"))
  result = Long.toHexString(item.getCrc());
else if (tag.data().equalsIgnoreCase("method"))
  result = (item.getMethod() == ZipEntry.DEFLATED ? "DEFLATED" : "STORED");
if (result != null)
  return lasso.outputTagData(result);
if (tag.data().equalsIgnoreCase("usize"))
  lasso.typeAllocInteger(out, item.getSize());
else if (tag.data().equalsIgnoreCase("csize"))
  lasso.typeAllocInteger(out, item.getCompressedSize());
else if (tag.data().equalsIgnoreCase("date"))
  lasso.typeAllocString(out, df.format(new Date(item.getTime())));
else if (tag.data().equalsIgnoreCase("isDir"))
  lasso.typeAllocBoolean(out, entry.isDirectory());
else if (tag.data().equalsIgnoreCase("position"))
  lasso.typeAllocInteger(out, index);
else if (tag.data().equalsIgnoreCase("enumerate"))
```

```
166 {
167   enum = zip.entries();
168   index = 0;
169 }
170 else if (tag.data().equalsIgnoreCase("getdata"))
171 {
172   int max = 0, skip = 0;
173
174   if (lasso.findTagParam("-skip", param) == ERR_NOERR)
175     skip = Integer.parseInt(param.data());
176   if (lasso.findTagParam("-max", param) == ERR_NOERR)
177     max = Integer.parseInt(param.data());
178   int count = 0;
179   int toRead = 1024;
180   if (max == 0 || max > item.getSize())
181     max = (int)item.getSize() - skip;
182   else if (max < 1024)
183     toRead = max;
184   try {
185     InputStream is = zip.getInputStream(item);
186     is.skip(skip);
187     byte b[] = new byte[toRead];
188     while ((count=is.read(b, 0, toRead)) > -1 && max > 0)
189     {
190       max -= count;
191       if (count > 0)
192         lasso.outputTagData(new String(b, 0, count));
193     }
194     is.close();
195   } catch (IOException ioe) {}
196 }
197 else if (tag.data().equalsIgnoreCase("next"))
198 {
199   boolean reset = (enum == null);
200   if (enum != null && !enum.hasMoreElements())
201     enum = null;
202   else if (reset)
203     enum = zip.entries();
204   boolean hasMore = (enum != null && enum.hasMoreElements());
205   lasso.typeAllocBoolean(out, hasMore);
206   if (hasMore)
207   {
208     entry = (ZipEntry)enum.nextElement();
209     if (reset)
210       index = 1;
211     else
212       index++;
213   }
```

```
214 }
215 if (!out.isNull())
216   return lasso.returnTagValue(out);
217 return err;
218 }
```

Custom Data Type Module Walk-Through

This section provides a step-by-step walk-through for building the custom type tag module.

To write a sample LJAPI tag module:

1 First, import com.omnipilot.lassopro.* classes as shown in line 1.

```
1  import com.omnipilot.lassopro.*;
2  import java.util.*;
3  import java.util.zip.*;
4  import java.io.*;
5  import java.text.DateFormat;
```

2 Define the class to be a subclass of the com.omnipilot.lassopro.LassoTagModule class.

```
6  public class ZipType extends LassoTagModule
```

3 Store the names of member tags implemented by our custom type in a String array variable.

```
10 static final String[] members = {
11   "File","Size","Count","Enumerate","Position","Next",
12   "GetData","Get","Name","Comment","Date","Crc",
13   "Method", "Extra", "CSize", "USize", "IsDir" };
```

4 Register the custom type initializer method, passing FLAG_INITIALIZER flag in the fourth parameter of the registerLassoModule method.

```
18 public void registerLassoModule()
19 {
20   registerTagModule("ZipType", "zip", "format",
21         FLAG_SUBSTITUTION | FLAG_INITIALIZER, "zip custom type tag");
22 }
```

5 Define main tag formatting method with the same name as specified in the third parameter of previously called registerTagModule method.

```
23 public int format(LassoCall lasso, int action)
```

6 Examine parameters passed to our type initializer and create new instance of a java.util.zip.ZipFile object, using resolved file path name.

```
40 String filePath = lasso.fullyQualifyPath(param.name());
41 filePath = lasso.resolvePath(filePath);
42 filePath = lasso.getPlatformSpecificPath(filePath);
43 this.zip = new ZipFile(filePath);
```

7 Create a new com.omnipilot.lassopro.LassoTypeRef variable to store a reference to the custom type, which is about to be created in the next step.

```
44 LassoTypeRef self = new LassoTypeRef();
```

8 Allocate new custom type instance, passing the LassoTypeRef variable and the type name to LassoCall.typeAllocCustom method.

```
45 if ((err = lasso.typeAllocCustom(self, "zip")) != ERR_NOERR)
46 {
47   lasso.setResultMessage("[Zip] couldn't create new zip type instance.");
48   return err;
49 }
```

9 Add member tags to the newly-allocated custom type. In our example, all member tags will be handled by the same Java method; however, LJAPI allows each member tag to have its own formatting method.

```
52 for (int i = 0; i < this.members.length; i++)
53 {
54   if ((err=lasso.typeAllocTag(ref, className, "memberFunc")) != ERR_NOERR ||
55       (err=lasso.typeAddMember(self, members[i], ref)) != ERR_NOERR)
56   {
57     lasso.setResultMessage("[Zip] error adding member: " + members[i]);
58     return err;
59   }
60 }
```

Note that adding the member tags to a custom type is a two-step process. First, an unnamed tag object is created and placed in a LassoTypeRef variable. In order to be successful, the second and third parameters in the LassoCall.typeAllocTag method must specify a valid class and method names used by Lasso for locating a formatting method in a Java class. Member tag methods have the same signature as a type initializer and regular substitution tag methods, and although not required they are most likely to be implemented in the same class with the main type initializer method.

Secondly, LassoCall.typeAddMember is used to add a reference to a newly-created tag (third parameter) to a custom type (first parameter), with the second parameter being a tag name.

10 Add all necessary callback methods, such as onConvert and onDestroy.

```
61 if (lasso.typeAllocTag(ref, className, "convertFunc") == ERR_NOERR)
62   lasso.typeAddMember(self, "onConvert", ref);
63 if (lasso.typeAllocTag(ref, className, "destroyFunc") == ERR_NOERR)
64   lasso.typeAddMember(self, "onDestroy", ref);
```

Callback methods are being triggered by the events that happen to a custom type in the course of its life. For example, when a type goes out of scope, its onDestroy tag method is called. When a custom type needs

to be converted to a different data type such as string or integer, its onConvert method is invoked.

Callbacks are added to the custom types in a similar fashion as the other members, with only constraint being their tag names, which must conform to established convention for naming callback tags. For a full list of intrinsic member tag names, see the Lasso 7 Language Guide.

11 Attach this module instance to a custom type.

```
65 if ((err = lasso.typeSetCustomJavaObject(self, this)) != ERR_NOERR)
66 {
67   lasso.setResultMessage("[Zip] couldn't attach java object to a custom type");
68   return err;
69 }
```

LassoCall.typeSetCustomJavaObject can be used to associate any private data with an instance of a custom type. Any Java object can be attached to a custom type and later retrieved with a call to a complimentary LassoCall.typeGetCustomJavaObject method. In the situation were associated object is an instance of the LassoTagModule subclass, Lasso will also try to invoke formatting methods on this object instead of creating a new instance (as it does for all substitution tag modules). Aside from producing much smaller overhead, this allows direct access to all instance (e.g. private) variables from any Java method implemented in that module.

12 Finally, return newly-generated custom type tag instance back to Lasso.

```
70 err = lasso.returnTagValue(self);
```

13 Implement formatting methods for onDestroy and onConvert callbacks.

```
81 public int destroyFunc(LassoCall lasso, int action)
82 {
83   if (zip != null)
84   {
85     try { zip.close(); }
86       catch (IOException e) {}
87     zip = null;
88   }
89   return ERR_NOERR;
90 }
```

14 In the case of onConvert callback, the first parameter passed to our method is the name of the type to which our custom Zip type should be converted to. If the desired type is a string, return the human-readable representation of the type, which consists of a type name and a zip file path name.

```
91 public int convertFunc(LassoCall lasso, int action)
92 {
93   LassoValue param = new LassoValue();
```

```
94    if (lasso.getTagParam(0, param) == ERR_NOERR &&
95      param.name().equalsIgnoreCase("string"))
96    {
97      lasso.outputTagData("zip:(" + this.zip.getName() + ")");
98    }
99
100 return ERR_NOERR;
101 }
```

15 Define our main member tag method memberFunc, that will take care of formatting over a dozen member tags. If tag name is File, return the full path name to the zip file.

```
109  if (tag.data().equalsIgnoreCase("File"))
110 return lasso.outputTagData(zip.getName());
```

16 If the member tag name is Count or Size, return an integer Zip entry count value.

```
111  else if (tag.data().equalsIgnoreCase("size") ||
112     tag.data().equalsIgnoreCase("count"))
113  {
114 lasso.typeAllocInteger(out, zip.size());
115 return lasso.returnTagValue(out);
116  }
```

17 Tags that output plain text can be processed first.

```
143  String result = null;
144  if (tag.data().equalsIgnoreCase("name") ||
145 tag.data().equalsIgnoreCase("get"))
146 result = item.getName();
147  else if (tag.data().equalsIgnoreCase("comment"))
148 result = item.getComment();
149  else if (tag.data().equalsIgnoreCase("crc"))
150 result = Long.toHexString(item.getCrc());
151  else if (tag.data().equalsIgnoreCase("method"))
152 result = (item.getMethod() == ZipEntry.DEFLATED ? "DEFLATED" : "STORED");
153  if (result != null)
154 return lasso.outputTagData(result);
```

18 Tags that return data types, such as integers or booleans, should allocate corresponding values using various LassoCall.typeAlloc... methods before passing them back to Lasso.

```
155  if (tag.data().equalsIgnoreCase("usize"))
156 lasso.typeAllocInteger(out, item.getSize());
157  else if (tag.data().equalsIgnoreCase("csize"))
158 lasso.typeAllocInteger(out, item.getCompressedSize());
159  else if (tag.data().equalsIgnoreCase("date"))
160 lasso.typeAllocString(out, df.format(new Date(item.getTime())));
161  else if (tag.data().equalsIgnoreCase("isDir"))
162 lasso.typeAllocBoolean(out, entry.isDirectory());
```

```
163  else if (tag.data().equalsIgnoreCase("position"))
164  lasso.typeAllocInteger(out, index);
```

19 The Enumerate member restarts a previously used enumeration. Unless it is called in a middle of iterating through the Zip entries, this tag has the same effect as calling Next for the very first time, or immediately after advancing past the very last enumerated item in a Zip file.

```
165  else if (tag.data().equalsIgnoreCase("enumerate"))
166  {
167  enum = zip.entries();
168  index = 0;
169  }
```

20 The GetData member tag reads uncompressed data from one of the zipped items. This tag accepts two optional parameters, -Skip and -Max, which are used to specify starting offset and maximum number of bytes to be read from the Zip archive entry.

```
170  else if (tag.data().equalsIgnoreCase("getdata"))
171  {
172  int max = 0;
173  int skip = 0;
174  if (lasso.findTagParam("-skip", param) == ERR_NOERR)
175    skip = Integer.parseInt(param.data());
176  if (lasso.findTagParam("-max", param) == ERR_NOERR)
177    max = Integer.parseInt(param.data());
178  int count = 0;
179  int toRead = 1024;
180  if (max == 0 || max > item.getSize())
181    max = (int)item.getSize() - skip;
182  else if (max < 1024)
183    toRead = max;
184  try {
185    InputStream is = zip.getInputStream(item);
186    is.skip(skip);
187    byte b[] = new byte[toRead];
188    while ((count=is.read(b, 0, toRead)) > -1 && max > 0)
189    {
190      max -= count;
191      if (count > 0)
192        lasso.outputTagData(new String(b, 0, count));
193    }
194    is.close();
195  } catch (IOException ioe) {}
196  }
```

21 The last member tag Next iterates through Zip archive entries, placing the internally maintained pointer at the next selected item. This tag provides fast sequential access to items stored in the Zip archive, and should be

used in concert with various accessor tags implemented in this module. When the end of the file is reached and no more items are available, the tag returns False and restarts the iteration, positioning the internal pointer immediately before the first Zip item.

```
197   else if (tag.data().equalsIgnoreCase("next"))
198   {
199   boolean reset = (enum == null);
200   if (enum != null && !enum.hasMoreElements())
201     enum = null;
202   else if (reset)
203     enum = zip.entries();
204   boolean hasMore = (enum != null && enum.hasMoreElements());
205   lasso.typeAllocBoolean(out, hasMore);
206   if (hasMore)
207   {
208     entry = (ZipEntry)enum.nextElement();
209     if (reset)
210       index = 1;
211     else
212       index++;
213   }
214   }
```

22 Finally, if any of the previous operations produced a valid result, pass the resulting value back to Lasso, returning an ERR_NOERR error code to flag a successful member tag execution.

```
215   if (!out.isNull())
216   return lasso.returnTagValue(out);
217   return err;
```

61

Chapter 61
LJAPI Data Sources

This chapter includes information about creating data source connectors in Java using the Lasso Java API (LJAPI).

- *Data Source Connector Operation* discusses the theory of creating data source connectors in Java.
- *Data Source Connector Tutorial* walks through a sample project that ships with every installtion of Lasso Professional.

Data Source Connector Operation

When Lasso Professional 8 starts up, it looks for module files (Windows DLLs or Mac OS X DYLIBS) in the LassoModules folder. As Lasso encounters each module, it executes the module's registerLassoModule() function once and only once. It is the job of the LJAPI developer to write code to register each new data source (or custom tag) methods in this registerLassoModule() function. Both substitution tags and data sources may be registered at the same time, and the code for them can reside in the same module. The only difference between registering a data source and a substitution tag is whether registerTagModule() or registerDSModule() is called.

Data sources are typically more complex than substitution tags because Lasso Service calls them with many different actions during the course of various database operations. Whereas a substitution tag only needs to know how to format itself, a data source needs to enumerate its tables, search through records, add new records, delete records, and more. Even so, this added complexity is easily handled with a single switch() statement, as you will see in the *Data Source Connector Tutorial* section of this chapter.

Data Source Connectors and Lasso Administration

Once a custom data source connector module is registered by Lasso, it will appear in the *Setup > Data Sources > Connectors* section of Lasso Site Administration. If a connector appears here, then it has been installed correctly.

The administrator adds the data source connection information to the *Setup > Data Sources > Hosts* section of Lasso Site Administration, which sets the parameters by which Lasso connects to the data source via the connector. This information is stored in the Lasso_Internal Lasso MySQL database, where the connector can retrieve and use the data via function calls.

The data that the administrator can submit in the *Setup > Data Sources > Hosts* section of Lasso Site Administration includes the following:

- **Name** – The administrator-defined name of the data source host.
- **Connection URL** – The URL string required for Lasso to connect to a data source via the connector. This typically includes the IP address of the machine hosting the data source.
- **Connection Parameters** – Additional parameters passed with the Connection URL. This can include the TCP/IP port number of the data source.
- **Status** – Allows the administrator to enable or disable the connector in Lasso Professional 8.
- **Default Username** – The data source username required for Lasso to gain access to the data source.
- **Default Password** – The data source password required for Lasso to gain access to the data source.

The Connection URL, Connection Parameters, Default Username, and Default Password values are passed to the data source via data source function methods in the com.omnipilot.lassopro.LassoCall class, which are described in the *LJAPI Class Reference* section of chapter.

Data Source Connector Tutorial

The following section provides a walk-through of an example data source to show how some of the LJAPI features are used. This code will be most similar to the sample NNTPDataSource project, which is provided with Lasso Professional 8 in the following folder.

Lasso Professional 8/Documentation/3 - Lasso Langage Guide/exammples/LJAPI/DataSourceConnectors/Nntp_ds

The example data source connector bridges a news (NNTP) server and Lasso Professional 8. Network News Transfer Protocol (NNTP) is used to read and post articles on Usenet news servers. This specific example has been tested with the Microsoft NNTP Service 5.0, and it provides a good start for any developer desiring to build a data source connector module supporting a large variety of other NNTP servers.

While an NNTP server is not exactly an RDBMS, there are some advantages to implementing the NNTP client as a data source connector. The hierarchy of a news storage is somewhat similar to that of a traditional RDBMS. News articles (rows) are organized in groups (tables), which in turn are parts of distributions (databases). However, due to a sheer number of news groups available on an average news server (2000-50000+), treating groups as database tables would put a big load on the internal Lasso security mechanism, which is required to keep track of permissions for every registered database table. Therefore, the hierarchy has been adopted to minimize the stress put on Lasso security.

The NNTP connector adds a single News database containing two static tables: Groups and Articles. Performing a search on the Group table returns a list of groups available on the server. Similarly, executing a query on the Articles table retrieves a range of articles from a specific newsgroup.

Updating groups or articles is not supported by the NNTP protocol, so only search and insert data source actions are implemented by this connector. SQL actions are also not supported, although it is possible to build a simple parser for translating SQL statements into commands understood by NNTP servers.

Data Source Connector Code

Below is the code for the data source module. Line numbers are provided to the left of each line of code, and are referenced in the *Data Source Connector Walk-Through section*.

Data Source Connector Code

```
1  import com.omnipilot.lassopro.*;
2  import java.net.*;
3  import java.io.*;
4  import java.util.*;
5  public class NNTP_DS extends LassoDSModule
6  {
7  Socket sock;
8  PrintStream printer;
9  BufferedReader reader;
10 String host = null;
```

```
11 int port = 0;
12 String user = null, pass = null;
13 String hostInfo = "";
14 Vector headers = new Vector(10);
15 int refsIdx = -1;
16 int xrefIdx = -1;
17 int bytesIdx = -1;
18 boolean useXpat = false;
19 String groupFilter = "";
20 String group = "";
21 String article = "";
22 int groupCount = -1;
23 int articleCount = -1;
24
25 public void registerLassoModule () {
26   registerDSModule("NNTP", "dsFunc", 0, "Lasso Connector for NNTP",
       "Simple Usenet client");
27 }
28 public int dsFunc (LassoCall lasso, int cmd, LassoValue value) {
29   int err = ERR_NOERR;
30   switch (cmd) {
31     case ACTION_INIT:
32       err = doInit(lasso);
33       break;
34     case ACTION_TERM:
35       err = doTerm(lasso);
36       break;
37     case ACTION_EXISTS:
38       if (!value.data().equalsIgnoreCase("News"))
39         err = LassoErrors.WebNoSuchObject;
40       break;
41     case ACTION_DB_NAMES:
42       err = doDBNames(lasso);
43       break;
44     case ACTION_TABLE_NAMES:
45       err = doTableNames(lasso, value.data());
46       break;
47     case ACTION_INFO:
48       err = doInfo(lasso, true);
49       break;
50     case ACTION_SEARCH:
51       err = doSearch(lasso);
52       break;
53   }
54   return err;
55 }
56 int doInit(LassoCall lasso) {
57   return ERR_NOERR;
```

```
58  }
59  int doTerm(LassoCall lasso) {
60    close();
61    return ERR_NOERR;
62  }
63  int doDBNames(LassoCall lasso) {
64    return lasso.addDataSourceResult("News");
65  }
66  int doTableNames(LassoCall lasso, String db) {
67    if (!db.equalsIgnoreCase("News"))
68      return -1;
69    lasso.addDataSourceResult("Groups");
70    lasso.addDataSourceResult("Articles");
71    return ERR_NOERR;
72  }
73  int doInfo(LassoCall lasso, boolean listAllCols) {
74    int err = ERR_NOERR;
75    LassoValue tbl = new LassoValue();
76    err = lasso.getTableName(tbl);
77    if (err != ERR_NOERR || tbl.data().length() == 0)
78      return LassoErrors.InvalidParameter;
79       if (!connect(lasso))
80      return LassoErrors.Network;
81    if (tbl.data().equalsIgnoreCase("Groups")) {
82      lasso.addColumnInfo("Group", 0,
            LassoValue.TYPE_CHAR, PROTECTION_READ_ONLY);
83      lasso.addColumnInfo("Last", 0,
            LassoValue.TYPE_INT, PROTECTION_READ_ONLY);
84      lasso.addColumnInfo("First", 0,
            LassoValue.TYPE_INT, PROTECTION_READ_ONLY);
85      lasso.addColumnInfo("AllowPost", 0,
            LassoValue.TYPE_CHAR, PROTECTION_READ_ONLY);
86    } else if (tbl.data().equalsIgnoreCase("Articles")) {
87      if (!this.headers.isEmpty()) {
88        String str;
89        int type, count = headers.size();
90        lasso.addColumnInfo("Number", 0,
            LassoValue.TYPE_INT, PROTECTION_READ_ONLY);
91        for (int i = 0; i < count; ++i) {
92          str = (String)this.headers.elementAt(i);
93          if (str.equalsIgnoreCase("Lines") ||
              str.equalsIgnoreCase("Bytes"))
94            type = LassoValue.TYPE_INT;
95          else if (str.equalsIgnoreCase("Date"))
96            type = LassoValue.TYPE_DATETIME;
97          else
98            type = LassoValue.TYPE_CHAR;
99          err = lasso.addColumnInfo((String)headers.elementAt(i), 0,
```

```
                type, PROTECTION_READ_ONLY);
100         }
101         if (listAllCols) {
102           lasso.addColumnInfo("Headers", 0,
                  LassoValue.TYPE_CHAR, PROTECTION_READ_ONLY);
103           lasso.addColumnInfo("Body", 0,
                  LassoValue.TYPE_CHAR, PROTECTION_READ_ONLY);
104         }
105       }
106     }
107     return err;
108   }
109   int doSearch(LassoCall lasso) {
110     int err = ERR_NOERR;
111     int skip = 0;
112     int max = 50;
113     int totalcount = 0;
114     String filter = "", reply = "";
115     LassoValue tbl = new LassoValue();
116     LassoValue val = new LassoValue();
117     IntValue ival = new IntValue();
118     if (lasso.getSkipRows(ival) == ERR_NOERR)
119       skip = ival.intValue();
120     if (lasso.getMaxRows(ival) == ERR_NOERR)
121       max = ival.intValue();
122     lasso.getTableName(tbl);
123     lasso.getInputColumnCount(ival);
124     if (!connect(lasso))
125       return LassoErrors.Network;
126     if ((err = doInfo(lasso, max == 1)) != ERR_NOERR)
127       return err;
128     try {
129       if (tbl.data().equalsIgnoreCase("GROUPS")) {
130         if (lasso.findInputColumn("group", val) == ERR_NOERR) {
131           if (val.type() == LassoOperators.OP_ENDS_WITH)
132             filter = '*' + val.data();
133           else if (val.type() == LassoOperators.OP_CONTAINS)
134             filter = '*' + val.data() + '*';
135           else if (val.type() == LassoOperators.OP_EQUALS)
136             filter = val.data();
137           else
138             filter = val.data() + '*';
139         }
140         this.printer.print("LIST ACTIVE " + filter + "\r\n");
141         reply=reader.readLine();
142         if (!reply.startsWith("2"))
143           return setError(lasso, reply);
144         if (!this.groupFilter.equalsIgnoreCase(filter)) {
```

```
145         this.groupFilter = filter;
146         this.groupCount = -1;
147       }
148       err = addGroups(lasso, skip, max);
149     } else if (tbl.data().equalsIgnoreCase("ARTICLES")) {
150       if (lasso.findInputColumn("-group", val) == ERR_NOERR ||
151         lasso.findInputColumn("group", val) == ERR_NOERR) {
152         if (val.data().length() > 0) {
153           if (!val.data().equalsIgnoreCase(this.group))
154             this.articleCount = -1;
155           this.group = val.data();
156         }
157       }
158       if (this.group == null || this.group.length() < 1) {
159         lasso.setResultMessage("Missing group parameter.");
160         return LassoErrors.InvalidParameter;
161       }
162       String id = null;
163       ival.setInt(0);
164       if (lasso.getRowID(ival) == ERR_NOERR && ival.intValue() != -1)
165         id = Integer.toString(ival.intValue());
166       else if (lasso.getPrimaryKeyColumn(val) == ERR_NOERR &&
167           (val.name().equalsIgnoreCase("message-id") ||
168            val.name().equalsIgnoreCase("number")))
169         filter = val.data();
170       else if (lasso.findInputColumn("message-id", val) == ERR_NOERR ||
171           lasso.findInputColumn("number", val) == ERR_NOERR)
172         filter = val.data();
173       if (this.articleCount == -1) {
174         err = selectGroup(lasso);
175         if (err != ERR_NOERR)
176           return err;
177       }
178       if (max == 1 && (filter == null || filter.length() < 1))
179         id = getRange(lasso, skip, 1);
180       if (filter.startsWith("<") || filter.indexOf('-') == -1)
181         id = filter;
182       if (id != null && id.length() > 1) { // detail
183         this.printer.print("ARTICLE " + id + "\r\n");
184         reply=reader.readLine();
185         if (!reply.startsWith("2"))
186           return setError(lasso, reply);
187         int idx=0, i=0, bytes=0;
188         String str;
189         String[] data = new String[headers.size()+3];
190         data[0] = reply.substring(4, reply.indexOf(' ', 4));
191         data[data.length-1] = ""; // body
192         data[data.length-2] = ""; // headers
```

```
193           while(!(reply=reader.readLine()).startsWith(".")) {
194             bytes += reply.length() + 2;
195             i = reply.indexOf(": ");
196             if (i != -1) { // header
197               str = reply.substring(0,i);
198               idx = this.headers.indexOf(str);
199               if (idx != -1) // known header
200                 data[idx+1] = reply.substring(i+2);
201               else
202                 data[data.length-2] += reply + '\r';
203             } else { // body
204               StringBuffer buf = new StringBuffer();
205               while (!(reply=reader.readLine()).startsWith(".")) {
206                 bytes += reply.length() + 2;
207                 buf.append(reply).append('\r');
208               }
209               data[data.length-1] = buf.toString();
210               data[this.bytesIdx] = Integer.toString(bytes + 2);
211               break;
212             }
213           }
214           if (data[0].equals("0") && this.refsIdx != -1) {
215             idx = data[this.xrefIdx].lastIndexOf(group);
216             if (idx != -1) {
217               str = data[this.xrefIdx].substring(idx + group.length() + 1);
218               if ((idx=str.indexOf(' ')) != -1)
219                 str = str.substring(0, idx);
220               data[0] = str;
221             }
222           }
223           err = lasso.addResultRow(data);
224         } else { // GET LIST
225           if (filter == null || filter.length() == 0)
226             filter = getRange(lasso, skip, max);
227           this.printer.print("XOVER " + filter + "\r\n");
228           reply=reader.readLine();
229           if (!reply.startsWith("2"))
230             return setError(lasso, reply);
231           while(err == ERR_NOERR && !(reply=reader.readLine()).startsWith("."))
232             err = lasso.addResultRow(split(reply, "\t"));
233         }
234         lasso.setNumRowsFound(this.articleCount);
235       }
236     } catch (Exception e) {
237       System.err.println(e.toString());
238       lasso.setResultMessage(e.getMessage());
239       err = LassoErrors.Network;
240       try { this.sock.close(); }
```

```
241         catch (Exception e2) {}
242       this.sock = null;
243     }
244     return err;
245   }
246   int setError(LassoCall lasso, String reply) {
247     int err = -1;
248     try {
249       err = Integer.parseInt(reply.substring(0, 3));
250       lasso.setResultMessage(reply.substring(4));
251     } catch (Exception e) {};
252     lasso.setResultCode(err);
253     return err;
254   }
255   String[] split (String str, String ch) {
256     int i = 0;
257     int numcols = headers.size() + 1;
258     String cols[] = new String[ numcols ];
259     StringTokenizer tok = new StringTokenizer(str, ch);
260     int count = tok.countTokens();
261     while (tok.hasMoreTokens()) {
262       if (i == this.refsIdx && numcols > count)
263         cols[i++] = ""; // empty References field
264       cols[i++] = tok.nextToken();
265     }
266     return cols;
267   }
268   boolean connect(LassoCall lasso) {
269     if (getHostInfo(lasso) != ERR_NOERR)
270       return false;
271     try
272     {
273       String reply;
274       if (this.sock != null) {
275         this.printer.print("MODE READER\r\n"); // probe the connnection
276         reply = reader.readLine();
277         if (!reply.startsWith("2")) {
278           this.sock.close();
279           this.sock = null;
280         }
281       }
282       if (this.sock == null) {
283         this.sock=new Socket(this.host,this.port);
284         this.reader=new BufferedReader(new InputStreamReader(this.sock.
getInputStream()), 2500);
285         this.printer=new PrintStream(new BufferedOutputStream(this.sock.getOutputStre
am(),2500),true);
286         this.hostInfo = this.reader.readLine();
```

```
287         login();
288         this.printer.print("MODE READER\r\n");
289         reader.readLine();
290         if (this.headers.isEmpty()) {
291           printer.print("LIST OVERVIEW.FMT\r\n");
292           reply = reader.readLine();
293           if (reply.startsWith("2")) {
294             int idx, i = 1;
295             while(!(reply=reader.readLine()).startsWith(".")) {
296               idx = reply.indexOf(':');
297               if (idx != -1)
298                 reply = reply.substring(0, idx);
299               this.headers.addElement(reply);
300               if (reply.equalsIgnoreCase("References"))
301                 this.refsIdx = i;
302               else if (reply.equalsIgnoreCase("Bytes"))
303                 this.bytesIdx = i;
304               else if (reply.equalsIgnoreCase("Xref"))
305                 this.xrefIdx = i;
306               ++i;
307             }
308           }
309         }
310       }
311     } catch (Exception e) {
312       lasso.setResultMessage(e.getMessage());
313       this.sock = null;
314       return false;
315     }
316     return true;
317   }
318   void close() {
319     this.host = null;
320     this.headers.clear();
321     this.groupCount = this.articleCount = -1;
322     this.refsIdx = this.xrefIdx = this.bytesIdx = -1;
323     try
324     {
325       this.printer.print("QUIT\r\n");
326       this.reader.close();
327       this.printer.close();
328       this.sock.close();
329       this.sock = null;
330     } catch (Exception e) {}
331   }
332   boolean login()
333   {
334     if (user != null && user.length() > 0) {
```

```
335       try
336       {
337         this.printer.print("AUTHINFO USER " + this.user + "\r\n");
338         this.printer.print("AUTHINFO PASS " + this.pass + "\r\n");
339         return (reply.startsWith("281"));
340       } catch (Exception e) {}
341     }
342     return false;
343   }
344   int getHostInfo(LassoCall lasso) {
345     int err = ERR_NOERR;
346     LassoValue hostPort = new LassoValue();
347     LassoValue userPass = new LassoValue();
348     err = lasso.getDataHost(hostPort, userPass);
349     if (err != ERR_NOERR ||
350         hostPort.name() == null ||
351         hostPort.name().length() == 0)
352       return err;
353     if (!hostPort.name().equalsIgnoreCase(this.host))
354       close();
355     this.host = hostPort.name();
356     this.user = userPass.name();
357     this.pass = userPass.data();
358     try {
359       this.port = Integer.parseInt(hostPort.data());
360     } catch (Exception e) {}
361
362     if (this.port == 0)
363       this.port = 119; // default NNTP port
364     return ERR_NOERR;
365   }
366   int addGroups(LassoCall lasso, int skip, int max) {
367     int err = ERR_NOERR;
368     int count = 0;
369     boolean getFirst = (max == 1 || this.group == null || this.group.length() < 1);
370     String reply = "";
371     try {
372       while ((skip-- > 0) && !(reply=reader.readLine()).startsWith("."))
373         count++;
374       if (!reply.startsWith(".")) {
375         String row[];
376         while (err == ERR_NOERR && !(reply=reader.readLine()).startsWith(".") &&
    (max-- > 0)) {
377           count++;
378           row = split(reply, " ");
379           if (getFirst) {
380             this.group = row[0];
381             err = lasso.addResultRow(row);
```

```
382             getFirst = false;
383           } else
384             err = lasso.addResultRow(row);
385         }
386         if (this.groupCount != -1) {
387           count = this.groupCount;
388           this.sock.close();
389           this.sock = null;
390         } else if (!reply.startsWith(".")) {
391           while (!(reader.readLine()).startsWith("."))
392             count++;
393           this.groupCount = count;
394         }
395       }
396     } catch (Exception e) { System.err.println(e.toString()); }
397     err = lasso.setNumRowsFound(count);
398     return err;
399   }
400   int selectGroup(LassoCall lasso) throws java.io.IOException {
401     this.printer.print("GROUP " + this.group + "\r\n");
402     String reply=reader.readLine();
403     if (!reply.startsWith("2"))
404       return setError(lasso, reply);
405     else
406       return ERR_NOERR;
407   }
408   String getRange(LassoCall lasso, int skip, int max) {
409     this.printer.print("LISTGROUP " + this.group + "\r\n");
410     StringBuffer result = new StringBuffer();
411     int count = 0;
412     try {
413       String reply=reader.readLine();
414       if (reply.startsWith("2")) {
415         String last = "";
416         while (!(reply=reader.readLine()).startsWith(".") && (skip-- > 0))
417           count++;
418         if (!reply.startsWith(".")) {
419           result.append(reply);
420           if (max != 1)
421             result.append("-");
422           while (!(reply=reader.readLine()).startsWith(".") && (--max > 0)) {
423             count++;
424             last = reply;
425           }
426           if (this.articleCount > -1) {
427             this.printer.println("QUIT\r\n");
428             this.sock.close();
429             this.sock = null;
```

```
430             count = this.articleCount;
431             if (connect(lasso))
432               selectGroup(lasso);
433           } else if (!reply.startsWith(".")) {
434             while (!(reader.readLine()).startsWith("."))
435               count++;
436             result.append(last);
437             this.articleCount = count;
438           }
439         }
440       }
441     } catch (Exception e) {} ;
442     return result.toString();
443   }
444   }
```

Data Source Connector Walk-Through

This section provides a step-by-step walk-through for building the described data source connector.

To build a sample LJAPI Data Source Connector:

1 First, import com.omnipilot.lassopro.* classes as shown in line 1.

```
1  import com.omnipilot.lassopro.*;
2  import java.net.*;
3  import java.io.*;
4  import java.util.*;
```

2 Define your module to be a subclass of the com.omnipilot.lassopro.LassoDSModule class:

```
6  public class NNTP_DS extends LassoDSModule {
```

Define the storage for global variables, which are objects used to communicate with an NNTP server, authentication and server info, etc.

```
7  Socket sock;
8  PrintStream printer;
9  BufferedReader reader;
...
```

3 Define the registerLassoModule method.

```
17  public void registerLassoModule() {
```

Every Lasso module must implement the registerLassoModule() method. This method will be called by Lasso at startup, giving your module a chance to register its data source(s).

4 Define your main data source method. This function gets called with various actions as Lasso Professional requests information from the data source. The method name should be identical to the string passed in the second parameter of the registerLassoModule() method.

```
28 public int dsFunc (LassoCall lasso, int cmd, LassoValue value)
```

5 Dispatch the action to corresponding Java method implemented in the module. The switch statement distinguishes between various actions. For a complete list of action constant values, see the LassoDSModule class reference.

```
30 switch (cmd) {
31   case ACTION_INIT:
32     err = doInit(lasso);
33     break;
34   case ACTION_TERM:
35     err = doTerm(lasso);
         break;
```

6 Among various actions that can be performed by a data source module the, ACTION_EXISTS command is sent by Lasso Professional to verify that a particular database exists on a specific host. If the name of the database being looked up is not known, the module must return a LassoErrors.WebNoSuchObject error:

```
37   case ACTION_EXISTS:
38     if (!value.data().equalsIgnoreCase("News"))
39       err = LassoErrors.WebNoSuchObject;
40     break;
```

7 Return the ERR_NOERR result code upon successful completion of the task. Returning a non-zero value will cause the Lasso Professional engine to report a fatal error and stop processing the page.

```
54 return err;
```

8 After successful registration, every data source module receives the ACTION_INIT command, which gives it a chance to establish connection with a data source or perform any other initialization tasks. Our module simply returns ERR_NOERR result code:

```
56 int doInit(LassoCall lasso) {
57   return ERR_NOERR;
58 }
```

9 Similarly, Lasso sends the ACTION_TERM command to all registered data source modules during its shutdown sequence. The sample data source uses this as a signal to close the connection to a NNTP server and perform additional clean-up tasks:

```
59 int doTerm(LassoCall lasso) {
60   close();
```

```
61    return ERR_NOERR;
62  }
```

10 The ACTION_DB_NAMES command is sent whenever Lasso Professional needs to get a list of databases which the data source provides access to. The developer must write code that discovers all the databases the module knows of, and call LassoCall.addDataSourceResult() once for each database it encounters:

```
65  int doDBNames(LassoCall lasso) {
66    return lasso.addDataSourceResult("News");
67  }
```

11 Whenever a data source module receives the ACTION_TABLE_NAMES command, it must examine the database name passed in the LassoValue parameter, and return the names of all tables available in the specified database:

```
68  int doTableNames(LassoCall lasso, String db) {
69    if (!db.equalsIgnoreCase("News"))
70      return -1;
71    lasso.addDataSourceResult("Groups");
72    lasso.addDataSourceResult("Articles");
73    return ERR_NOERR;
74  }
```

12 Lasso Professional sends the ACTION_INFO command when it needs to retrieve the information about columns contained in the result set. Inline tag actions like -FindAll and -Search usually return a result set containing certain number of rows/records, each consisting of one or more columns/fields. When data source module receives an ACTION_INFO command, it must call LassoCall.addColumnInfo() method once for each column stored in the result set.

```
73  int doInfo(LassoCall lasso, boolean listAllCols) {
74    int err = ERR_NOERR;
75    LassoValue tbl = new LassoValue();
76    err = lasso.getTableName(tbl);
77    if (err != ERR_NOERR || tbl.data().length() == 0)
78      return LassoErrors.InvalidParameter;
79      if (!connect(lasso))
80      return LassoErrors.Network;
81    if (tbl.data().equalsIgnoreCase("Groups")) {
82      lasso.addColumnInfo("Group", 0,
            LassoValue.TYPE_CHAR, PROTECTION_READ_ONLY);
83      lasso.addColumnInfo("Last", 0,
            LassoValue.TYPE_INT, PROTECTION_READ_ONLY);
84      lasso.addColumnInfo("First", 0,
            LassoValue.TYPE_INT, PROTECTION_READ_ONLY);
85      lasso.addColumnInfo("AllowPost", 0,
            LassoValue.TYPE_CHAR, PROTECTION_READ_ONLY);
```

```
86    } ...
```

13 The ACTION_SEARCH command is sent whenever Lasso Professional needs to perform the search action on a data source.

```
109  int doSearch(LassoCall lasso) {
```

14 All of the information about the current search parameters (database and table names, search arguments, sort arguments, etc.) can be retrieved by calling various LJAPI methods such as LassoCall.getDataSourceName() and LassoCall.getTableName(). Similarly, one can call getSkipRows() and getMaxRows() methods to retrieve the -SkipRecords and -MaxRecords inline parameter values. For a complete list of available methods, see LassoCall class reference.

```
117  IntValue ival = new IntValue();
118  if (lasso.getSkipRows(ival) == ERR_NOERR)
119    skip = ival.intValue();
120  if (lasso.getMaxRows(ival) == ERR_NOERR)
121    max = ival.intValue();
122  lasso.getTableName(tbl);
123  lasso.getInputColumnCount(ival);
```

15 The module needs to perform different actions depending on the search table name.

```
129  if (tbl.data().equalsIgnoreCase("GROUPS")) {
```

16 Some NNTP servers allow retrieval of newsgroup listings filtered by a matching pattern. The module builds the pattern string based on the value of the inline search operator (beginsWith, endsWith, etc.).

```
130  if (lasso.findInputColumn("group", val) == ERR_NOERR) {
131    if (val.type() == LassoOperators.OP_ENDS_WITH)
132      filter = '*' + val.data();
133    else if (val.type() == LassoOperators.OP_CONTAINS)
134      filter = '*' + val.data() + '*';
135    else if (val.type() == LassoOperators.OP_EQUALS)
136      filter = val.data();
137    else
138      filter = val.data() + '*';
139  }
```

17 In case the search is being performed on the ARTICLES table, we need to find out the name of a newsgroup before we can proceed any further.

```
149  } else if (tbl.data().equalsIgnoreCase("ARTICLES")) {
150    if (lasso.findInputColumn("-group", val) == ERR_NOERR ||
151    lasso.findInputColumn("group", val) == ERR_NOERR) {
```

18 Next, we check if an article number or message ID has been included in the search criteria, either as a primary keyfield, record ID, or as a named search field.

```
164  if (lasso.getRowID(ival) == ERR_NOERR && ival.intValue() != -1)
165    id = Integer.toString(ival.intValue());
```

```
166 else if (lasso.getPrimaryKeyColumn(val) == ERR_NOERR &&
167       (val.name().equalsIgnoreCase("message-id") ||
168        val.name().equalsIgnoreCase("number")))
169   filter = val.data();
170 else if (lasso.findInputColumn("message-id", val) == ERR_NOERR ||
171         lasso.findInputColumn("number", val) == ERR_NOERR)
172 filter = val.data();
```

19 If none of the above was found, yet the -MaxRecords inline parameter appears to limit the query results to a single row, we can try finding the desired article ID based on the current -SkipRecords value.

```
178 if (max == 1 && (filter == null || filter.length() < 1))
179   id = getRange(lasso, skip, 1);
```

20 If the article has been identified, proceed with retrieving the message in its entirety.

```
182 if (id != null && id.length() > 1) { // detail
183   this.printer.print("ARTICLE " + id + "\r\n");
184   reply=reader.readLine();
```

21 Otherwise, select the next group of news articles and retrieve their headers.

```
225 if (filter == null || filter.length() == 0)
226   filter = getRange(lasso, skip, max);
227 this.printer.print("XOVER " + filter + "\r\n");
228 reply=reader.readLine();
229 if (!reply.startsWith("2"))
230   return setError(lasso, reply);
```

22 The LassoCall.addResultRow() method is used to return the results of a data source action. It should be called as many times as there are records in the result set, once for each record.

LassoCall.addResultRow() method takes a single String array parameter. Each array element corresponds to a record column/field contained in the result set. The total number of array elements must be equal to the number of times LassoCall.addColumnInfo() method was called for this data source action. Since news article headers are transmitted in the form of a tab-delimited string, we use our custom split() method to convert the data to a String array, suitable for passing to addResultRow() method:

```
231 while(err == ERR_NOERR && !(reply=reader.readLine()).startsWith("."))
232   err = lasso.addResultRow(split(reply, "\t"));
```

23 Finally, implement a number of convenience methods, including the setError() routine used for standard error handling:

```
246 int setError(LassoCall lasso, String reply) {
247   int err = -1;
248   try {
249     err = Integer.parseInt(reply.substring(0, 3));
```

```
250       lasso.setResultMessage(reply.substring(4));
251     } catch (Exception e) {};
252     lasso.setResultCode(err);
253     return err;
254   }
```

62

Chapter 62
LJAPI Reference

This chapter provides a reference to all of the types and function in the Lasso Java API (LJAPI).

- *LJAPI Interface Reference* introduces the interfaces that are provided with LJAPI.
- *LJAPI Class Reference* documents every class this is provided with LJAPI.

LJAPI Interface Reference

This section provides a listing of all Java interfaces available for use in LJAPI 7. All variables, constructors, and methods for each interface are organized by category under each interface name.

com.omnipilot.lassopro.LassoJavaModule

This is the base interface implemented by both substitution tag and data source LJAPI modules. Upon Lasso Service startup, the registerLassoModule method is called for every Java module located inside the LassoModules folder. Each module returns information about their name, implemented tags or data sources, method names, etc.

Data source modules are instantiated only once and then used repeatedly to perform various data source actions. Tag modules are instantiated every time Lasso resolves a tag implemented by a LassoTagModule.

Methods

registerLassoModule()

This method must be defined in all LJAPI modules. Lasso calls this once at startup to allow a module to register its tags or data sources.

```
public void registerLassoModule ( );
```

Variables

ERR_NOERR

On success, every method must return ERR_NOERR result code.

```
public static final int ERR_NOERR
```

LJAPI Class Reference

This section lists all the Java classes available for use in LJAPI 7. All variables, constructors, and methods for each interface are organized alphabetically under each interface name, unless specified otherwise.

com.omnipilot.lassopro.FloatValue

Wrapper class for a primitive float or double type. Used for returning decimal values from the LassoCall.typeGetDecimal method.

Constructors

```
public FloatValue()
public FloatValue(float value)
public FloatValue(double value)
```

Methods

doubleValue()

Returns the value of a FloatValue object as a double.

```
public double doubleValue()
```

floatValue()

Returns the value of a FloatValue object as a float.

```
public float floatValue()
```

toString()

Converts an object to a string. Overrides toString() method in class Object.

```
public String toString()
```

com.omnipilot.lassopro.IntValue

Wrapper for primitive integer types. Used for returning values from LassoCall methods, which in C would require passing the pointer-type parameters: int*, long* and LP_TypeDesc*. In addition, this class provides methods for converting a 4-byte int (LP_TypeDesc type in LCAPI) to a String and back.

Constructors

```
public IntValue()
public IntValue(int value)
public IntValue(long value)
```

Methods

byteValue()

Returns the value of an IntValue object as a 1-byte integer.

```
public byte byteValue()
```

shortValue()

Returns the value of an IntValue object as a 2-byte integer.

```
public short shortValue()
```

intValue()

Returns the value of an IntValue object as a 4-byte integer.

```
public int intValue()
```

longValue()

Returns the value of an IntValue object as an 8-byte integer.

```
public long longValue()
```

setByte()

Sets the value of an IntValue object to a 1-byte integer.

```
public void setByte( byte value )
```

setShort()

Sets the value of an IntValue object to a 2-byte integer.

```
public void setShort( short value )
```

setInt()

Sets the value of an IntValue object to a 4-byte integer.

```
public void setInt( int value )
```

setLong()

Sets the value of an IntValue object to an 8-byte integer.

```
public void setLong( long value )
```

toDescType()

Converts the lower 4 bytes of an IntValue value to a 4-char String.

```
public String toDescType()
```

toString()

Converts an object to a string. Overrides toString() method in class Object.

```
public String toString()
```

IntToFourCharString()

Static method used for converting an int to a 4-char String.

```
public static String IntToFourCharString( int value )
```

com.omnipilot.lassopro.LassoCall

Of all Java classes listed in this section, the LassoCall class is of the utmost importance. All the interaction between an LJAPI module and Lasso Professional 8 is achieved by means of invoking various methods implemented in the LassoCall class. These functions can be used to do any of the following: register your tags or data sources, allocate memory, return error messages, get tag or parameter information, get client/server environment information, output text, read/set MIME headers, access LDML variables, interpret/execute arbitrary LDML tags, store persistent data, check if the user is an administrator, perform data source functions, and safely access multiuser/multithreaded resources.

All class methods in this section are listed by their category.

Internal Value Methods

getLassoParam()

Fetches an internal server value such as path to LassoModules folder, name of the Lasso error log file, etc. For a full list of available parameters, please see the listing of constants defined in the LassoParams class.

```
public int getRequestParam( int key, LassoValue outResult );
```

getRequestParam()

Fetches an HTTP request value such as server port, cookies, root path, username, etc. For a full list of available parameters, please see the listing of constants defined in the LassoRequestParams class. Please note that some of these parameters may not be available on all HTTP servers.

```
public int getRequestParam( int key, LassoValue outResult );
```

Error Messages and Result Code Methods

setResultCode()

Sets the result code that can be displayed if the LDML programmer inserts [Error_CurrentError: -ErrorCode] into the format file after executing a custom LJAPI tag.

```
public int setResultCode( int err );
```

setResultMessage()

Sets the error message that can be displayed if the LDML programmer inserts [Error_CurrentError: -ErrorMessage] into the format file after executing a custom LJAPI tag.

```
public int setResultMessage( String msg );
```

Tag and Parameter Info Methods

getTagName()

Fetches the name of the tag that triggered this call (e.g. in the case of [my_tag: ...] the resulting value would be my_tag). This makes it possible to design a single tag function which can perform the duties of many different LDML tags, perhaps ones that all have similar functionality but different names.

```
public int getTagName( LassoValue result );
```

getTagParamCount()

Fetches the number of parameters that were passed to the tag. For instance, [my_tag: 'hello', -option=1, -hilite=false] will report that three parameters were passed (unnamed parameters are treated just like any other parameter).

```
public int getTagParamCount( IntValue result );
```

getTagParam()

Gets the name and value of a parameter given its index. Parameters are numbered left-to-right, starting at index 0: [my_tag: -param0='value0', -param1='value1', -param2=2].

```
public int getTagParam( int paramIndex, LassoValue result );
```

getTagParam2()

Get the parameter using the parameter index. This function differs from getTagParam( ) in that it preserves the actual type of the parameter instead of automatically converting it to a string. Keyword/value pairs are returned as a LASSO_PAIR type.

```
public int getTagParam2( int paramIndex, LassoTypeRef outValue );
```

tagParamIsDefined()

Returns ERR_NOERR if the parameter was defined. Otherwise, the parameter wasn't defined.

```
public int tagParamIsDefined( String paramName );
```

findTagParam()

Finds and fetches a tag parameter by name. A return value of ERR_NOERR means the parameter was found successfully.

```
public int findTagParam( String paramName, LassoValue result );
```

findTagParam2()

Finds and returns a tag parameter by name while preserving the original type. A returned value of ERR_NOERR means the parameter was successfully found.

```
public int findTagParam2( String paramName, LassoTypeRef outValue );
```

getTagEncoding()

Fetches the encoding method indicated for this tag. This is rarely used, because Lasso handles encoding and decoding for you.

```
public int getTagEncoding( IntValue method );
```

childrenRun()

Used to execute the contents of a container tag. Tags become containers when the FLAG_CONTAINER flag is used. The result parameter will contain the combined result data for all tags contained.

```
public int childrenRun( LassoTypeRef outValue );
```

runRequest()

Creates and runs a new LJAPI call on the given method (methodName of the className class). If there is already an active request on the current thread, the method will be run within the context of that thread. If there is no active request on the current thread, a new request will be created and run based on the global context. The tagAction parameter is passed to the methodName and can be used to signal or pass information to the function.

```
public static int runRequest( String className,
String methodName,
 int tagAction,
 int unused );
```

Output Methods

outputTagData()

Outputs any string data to the page. Lasso takes care of encoding, and this can be called as many times as needed. The second variant of this method is recommended for writing binary data.

```
public int outputTagData( String data );

public int outputTagData( byte[] data );
```

returnTagValue()

Specifies the return value for the tag. Note that only a single returnTagValue or outputTagData can be used from within a tag. returnTagValue is the prefered method for returning tag data as it allows data of any type to be returned (including binary data), while outputTagData is restricted to printable text data.

```
public int returnTagValue ( LassoTypeRef value );
```

Data Type Methods

typeAlloc()

This function will allocate a new type instance. The type is specified by the typeName parameter. An array of parameters can be passed to the type initializer. Types created through this function will be automatically destroyed after the LJAPI call has returned. In order to prevent this, typeDetach should be used.

```
public int typeAlloc ( String typeName,
                       LassoTypeRef[] params,
                       LassoTypeRef outType );
```

typeFree()

Attempts to free a type created using typeAlloc or any other method. The LassoCall variable may be null if the provided type has been detached using typeDetach.

```
public int typeFree ( LassoTypeRef inType );
```

typeDetach()

Prevents the type from being destroyed once the LJAPI call returns. Types that have been detached must eventually be destroyed using typeFree() (passing null in the LassoCall variable) or a memory leak will occur.

```
public int typeDetach( LassoTypeRef toDetach );
```

typeAllocNull()

This method allows new instances of LASSO_NULL data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned.

```
public int typeAllocNull ( LassoTypeRef outNull);
```

typeAllocString()

This method allows new instances of string data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned.

```
public int typeAllocString ( LassoTypeRef outString, String value);
```

typeAllocInteger()

This method allows new instances of integer data types to be allocated (Lasso integers are 8-byte signed INTs). Types allocated in this manner will be destroyed once the LJAPI call is returned.

```
public int typeAllocInteger ( LassoTypeRef outInteger, long value);
```

typeAllocDecimal()

This method allows new instances of decimal data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned.

```
public int typeAllocDecimal ( LassoTypeRef outDecimal, double value);
```

typeAllocPair()

This method allows new instances of pair data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned.

```
public int typeAllocPair ( LassoTypeRef outPair,
                           LassoTypeRef inFirst,
                           LassoTypeRef inSecond);
```

typeAllocReference()

This method allows new instances of reference data types to be allocated. Types allocated in this manner will be destroyed once the LCAPI call is returned.

```
public int typeAllocReference ( LassoTypeRef outRef,
                        LassoTypeRef referenced );
```

typeAllocTag()

This method allows new instances of tag data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned. Method methodName should have the same signature as TAG_METHOD_PROTOTYPE() method in the LassoTagModule class.

```
public int typeAllocTag ( LassoTypeRef outTag,
                String className,
                String methodName );
```

typeAllocArray()

This method allows new instances of array data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned.

```
public int typeAllocArray ( LassoTypeRef outArray,
                        LassoTypeRef[] inElements);
```

typeAllocMap()

This method allows new instances of map data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned.

Two versions of the same method are provided: in the first case the count of elements of the inElements array must be divisible by 2 and contain both keys and values (odd = key, even = value). In the second case, map keys and values must be passed in a separate parameters.

```
public int typeAllocMap ( LassoTypeRef outMap,
                LassoTypeRef[] inElements );
public int typeAllocMap ( LassoTypeRef outMap,
                LassoTypeRef[] inKeys,
                LassoTypeRef[] inValues );
```

typeAllocBoolean()

This method allows new instances of boolean data types to be allocated. Types allocated in this manner will be destroyed once the LJAPI call is returned.

```
public int typeAllocBoolean( LassoTypeRef outBool, boolean inValue );
```

typeGetBytes()

This method returns the data of a type instance as an array of bytes.

```
public bytes[] typeGetString( LassoTypeRef type);
```

typeGetString()

This method gets the data from a previously created string instance. When setting a value, the type is converted if required.

```
public int typeGetString( LassoTypeRef type, LassoValue outValue );
```

typeGetInteger()

This method gets the data from a previously created integer instance. When setting a value, the type is converted if required.

```
public int typeGetInteger( LassoTypeRef type, IntValue outValue );
```

typeGetDecimal()

This method gets the data from a previously created decimal instance. When setting a value, the type is converted if required.

```
public int typeGetDecimal( LassoTypeRef type, FloatValue outValue );
```

typeGetBoolean()

This method gets the data from a previously created boolean instance. When setting a value, the type is converted if required.

```
public int typeGetBoolean( LassoTypeRef type, BoolValue outValue );
```

typeSetBytes()

This method sets the data of a type instance. The type is converted if required.

```
public int typeSetBytes( LassoTypeRef type, byte[] value );
```

typeSetString()

This method sets the value of a previously created string type instance.

```
public int typeSetString( LassoTypeRef type, String value );
```

typeSetInteger()

This method sets the value of a previously created integer instance.

```
public int typeSetInteger( LassoTypeRef type, long value );
```

typeSetDecimal()

This method sets the value of a previously created decimal instance.

```
public int typeSetDecimal( LassoTypeRef type, double value );
```

typeSetBoolean()

This method sets the value of a previously created boolean instance.

```
public int typeSetBoolean( LassoTypeRef type, boolean value );
```

arrayGetSize()

This method gets the size of a previously created array instance.

```
public int arrayGetSize( LassoTypeRef array, IntValue outLen );
```

arrayGetElement()

This method gets an array element from a previously created array instance.

```
public int arrayGetElement( LassoTypeRef array,
                            int index,
                            LassoTypeRef outElement );
```

arraySetElement()

This method sets an array element in a previously created array instance.

```
public int arraySetElement( LassoTypeRef array,
                            int index,
                            LassoTypeRef element );
```

arrayRemoveElement()

This method removes an element from a previously created array instance.

```
public int arrayRemoveElement( LassoTypeRef array, int index );
```

mapGetSize()

This method gets the size of a previously created map instance.

```
public int mapGetSize( LassoTypeRef map, IntValue outLen );
```

mapFindElement()

This method finds an element in a previously created map instance stored under unique key.

```
public int mapFindElement( LassoTypeRef map,
                           LassoTypeRef key,
                           LassoTypeRef outElement );
```

mapGetElement()

This method gets an element from a previously created map instance using the element index.

```
public int mapGetElement( LassoTypeRef map,
                          int index,
                          LassoTypeRef outPair );
```

mapSetElement()

This function sets an element in a previously created map instance. If no elements were previously stored under the specified key, the element will be added to the map, otherwise the old element will be replaced by a new value.

```
public int mapSetElement( LassoTypeRef map,
                          LassoTypeRef key,
                          LassoTypeRef value );
```

mapRemoveElement()

This method removes an element from a previously created map instance.

```
public int mapRemoveElement( LassoTypeRef map, LassoTypeRef key );
```

pairGetFirst()

This method gets the first element from a previously created pair instance.

```
public int pairGetFirst( LassoTypeRef pair, LassoTypeRef outValue );
```

pairGetSecond()

This method gets the second element from a previously created pair instance.

```
public int pairGetSecond( LassoTypeRef pair, LassoTypeRef outValue );
```

pairSetFirst()

This method sets the first element in a previously created pair instance.

```
public int pairSetFirst( LassoTypeRef pair, LassoTypeRef first );
```

pairSetSecond()

This function sets the second element in a previously created pair instance.

```
public int pairSetSecond( LassoTypeRef pair, LassoTypeRef second );
```

typeGetMember()

This function is used to retrieve a member from a type instance. Members are searched by name with tag members searched first. Data members are searched if no tag member is found with the given name.

```
public int typeGetMember( LassoTypeRef fromType,
                          String named,
                          LassoTypeRef outMember );
```

typeGetProperties()

This method has two uses. If the targetType parameter is not null, it is used to get all data and tag members from a given type. They are returned as a pair of arrays in the outPair value. The first element of each pair is the map of data members for the type. The second element is the map of tag members. Each element in the array represents the members of each type inherited by the targetType.

If the targetType parameter is null, typeGetProperties will return an array containing the variable maps for the currently active request.

```
public int typeGetProperties ( LassoTypeRef targetType,
                               LassoTypeRef outPair );
```

typeGetName()

Retrieves the name of the target type.

```
public int typeGetName( LassoTypeRef target, LassoValue outName );
```

typeRunTag()

Used to to execute a given tag. The tag can be run given a specific name and parameters, and the return value of the tag can be accessed. If the tag is a member tag, the instance of which it is a member can be passed using the final parameter. The params, returnValue, and optionalTarget parameters may all be null.

A slightly modified version of the same method is provided for convenience puproses. It accepts a single LassoTypeRef parameter instead of a LassoTypeRef array.

```
public int typeRunTag ( LassoTypeRef tagType,
            String named,
            LassoTypeRef[] params,
            LassoTypeRef returnValue,
            LassoTypeRef optionalTarget );

public int typeRunTag ( LassoTypeRef tagType,
            String named,
            LassoTypeRef parameter,
            LassoTypeRef returnValue,
            LassoTypeRef optionalTarget );
```

typeAssign()

This performs an assignment of one type to another. The result will be the same as if the following had been executed in LDML:#left_hand_side = #right_hand_side

```
public int typeAssign( LassoTypeRef left_hand_side,
            LassoTypeRef right_hand_side);
```

typeStealValue()

This function transfers the data from one type to another type. Both types must be valid and pre-allocated. After the call, victim will still be valid, but will be of type null.

```
public int typeStealValue( LassoTypeRef thief, LassoTypeRef victim );
```

handleExternalConversion()

Converts a Lasso type into single-byte or binary data using the specific encoding name. The default for all database, column, table names should be "iso8859-1".

```
public byte[] handleExternalConversion(LassoTypeRef inInstance, String inEncoding);
```

handleInternalConversion()

Converts a single-byte or binary representation of a Lasso type back into an instance of that type.

```
public int handleInternalConversion( byte[] inData, String inEncoding, int
inClosestLassoType, LassoTypeRef outType );
```

typeInheritFrom

This function changes the inheritance structure of a type. Sets inNewParent to be the new parent of the child. Any parent that child currently has will be destroyed.

```
public int typeInheritFrom( LassoTypeRef inChild, LassoTypeRef inNewParent );
```

Custom Type Methods

typeAllocCustom()

This function is used within module methods that were registered as being a type initializer (FLAG_INITIALIZER). It initializes a blank custom type and sets the type's __type_name__ member to the provided value. The new type does not yet have a lineage and has no members added to it besides __type_name__. New data or tag members should be added using typeAddMember. The new custom type should be the return value of the type initializer. Any inherited members will be added to the type after the LJAPI call returns.

Warning: Do not call this unless you are in a type initializer. If you are not in a type initializer, the result will be a type that will never be fully initialized.

```
public int typeAllocCustom( LassoTypeRef outCustom, String name );
```

typeAddMember()

This is used to add new members to type instances. The member can be any sort of type including tags or other custom types.

```
public int typeAddMember( LassoTypeRef to,
                String named,
                LassoTypeRef member );
```

typeAllocFromProto()

Allocate a new type based on the given type. The given type's tag members will be referenced in the new type. No data members are added except for the typename member. Proto must be a custom type.

```
public int typeAllocFromProto( LassoTypeRef inProto, LassoTypeRef outType );
```

typeAllocOneOff()

Allocate a new type with the given name. The type does not have to have been registered as a type initializer or registered at all. The new type will have no tag or data members, but those may be added using the appropriate LCAPI call at any time. If no parent type is provided (a NULL pointer or empty string is passed in), type null will be the default. If a parent type is provided, it must have been a validly registered type initializer. onCreate will be called for the parent and beyond.

```
public int typeAllocOneOff( String inName, String inParentTypeName, LassoTypeRef
outType );
```

typeGetCustomJavaObject()

Custom types can have Java objects attached to them. The object can be retrieved at any point during the instance's lifetime. typeSetCustomJavaObject method retrieves the Java object associated with a custom type, or returns null if no object has been attached to this type.

```
public Object typeGetCustomJavaObject( LassoTypeRef type );
```

typeSetCustomJavaObject()

typeSetCustomJavaObject permits attachment of Java objects to custom types. Java object is retained until typeFreeCustomJavaObject is called, or the type is destroyed.

```
public int typeSetCustomJavaObject( LassoTypeRef type, Object object );
```

typeFreeCustomJavaObject()

Releases the Java object previously attached to a custom type. Must be called to free the Java object that is no longer needed, or to detach an old Java object before attaching a new one to the same custom type.

```
public int typeFreeCustomJavaObject( LassoTypeRef type );
```

Logging Function Methods

log()

Logs a message. The message goes to the prefered destination for the message level. Messages sent to a file are limited to 2048 bytes in length. Messages sent to the console are limited to 512 bytes in length. Messages sent to the database are limited a little less than 2048 bytes since the total length of the sql statement used to insert the message is limited to 2048 bytes. The msgLevel parameter must be one of the following: LOG_LEVEL_CRITICAL, LOG_LEVEL_WARNING, or LOG_LEVEL_DETAIL.

```
public static int log ( int msgLevel, String message );
```

logSetDestination()

Changes the system-wide log destination preference. You can log messages to more than one destination at a time by passing several flags in the destination parameter: FLAG_DEST_CONSOLE, FLAG_DEST_FILE, and/or FLAG_DEST_DATABASE.

```
public static int logSetDestination( int msgLevel, int destination );
```

MIME Header Methods

getResultHeader()

Retrieves current value of the result (HTTP) header. Part of the header that is returned to browsers is automatically built by Lasso, and can be modified or added to by LDML tags on the page. This function retrieves the current set of MIME headers that would be sent back to the browser if page processing were to stop now.

```
public int getResultHeader( LassoValue result );
```

setResultHeader()

Sets the result header, any data will be validated so as to be in the proper format.

```
public int setResultHeader( String header );
```

addResultHeader()

Simply appends the supplied data to the header, any data will be validated so as to be in the proper format.

```
public int addResultHeader( String data );
```

getCookieValue()

Retrieves a cookie value from the passed-in data sent by the client browser.

```
public int getCookieValue( String named, LassoValue value );
```

Page Variable Methods

getVariableCount()

Retrieves the number of array values which the named global variable has. Returns 1 if the global variable is not an array. Global variables are the same variables which you create in LDML statements, like [var: 'fred'=1234.56]. These variables last only as long as the current format file is executing; as soon as the hit gets sent back to the browser, these variables all get destroyed.

```
public int getVariableCount( String named, IntValue count );
```

getVariable()

Retrieves the value of the named global variable. If the global variable is an array, then the index specifies which array value to retrieve. If the global variable is not an array, then 0 is the only valid index. Array indices start at 0.

```
public int getVariable( String named, int index, LassoValue value );
```

getVariable2()

Retrieves the value of the named global variable while preserving the variable type.

```
public int getVariable2( String named, LassoTypeRef outValue );
```

setVariable()

Stores a new value into the named global variable. If the global variable is an array, then the 0-based index determines which array item to replace.

```
public int setVariable( String named, String value, int index );
```

setVariable2()

Stores a new global variable while preserving the type.

```
public int setVariable2( String named, LassoTypeRef inValue );
```

removeVariable()

Removes the specified variable (destroys it so it becomes undefined, as though it had never been created). If the named variable is an array, then you may pass in an index (0-based) to remove that array element. Once the array has 0 elements, then calling removeVariable on it will destroy the array itself.

```
public int removeVariable( String named, int index );
```

LDML Tag Interpreter Methods

formatBuffer()

Formats the supplied buffer and puts the resulting data in the data field of the LassoValue. The buffer should consist of plain text and bracketed Lasso tags.

```
public int formatBuffer( String buffer, LassoValue output );
```

Persistent Storage Tag Methods

storeHasData()

Returns ERR_NOERR if the data, specified by key, exists. The length of the stored data can be returned in the outLength parameter if you pass a valid IntValue object. You may pass null if you don't want to retrieve the length of the stored data.

```
public int storeHasData( String key, IntValue outLength );
```

storeGetData()

Fetches data that has been stored under the unique identifier key. The data will be returned in the data field of the LassoValue object.

```
public int storeGetData( String key, LassoValue outValue );
```

storePutData()

Adds the data to Lasso's storage. Key is the unique identifier for the data.

```
public int storePutData( String key, String data );
public int storePutData( String key, byte[] data );
```

Administration Methods

isAdministrator()

Returns ERR_NOERR if the current user has administrator privileges. This is useful for doing module administration that only the administrator should be able to do.

```
public int isAdministrator( );
```

Data Source Function Methods

getDSConnection()

This function accesses the current datasource connection.

```
public Object getDSConnection();
```

setDSConnection()

This function sets the current connection for the data source. May recurse to deliver the ACTION_CLOSE message if there is already a valid connection set.

```
public int setDSConnection( Object inConnection );
```

addDataSourceResult()

Sometimes Lasso Professional will query a data source function to return information, such as a list of database names or table names which the data source module controls. The module will call this function once for each name you add to the list, so if you have three database names you want to report back to Lasso Professional, you would call this function three times, once per database name.

```
public int addDataSourceResult( String data );
```

getDataSourceName()

Use this function when you want to ask Lasso Professional what database is being operated on. For instance, if you're being asked to perform a search, then you would call this function to retrieve the name of the database which Lasso Professional is asking you to search. It corresponds to the value of the parameter -Database='blah' passed to inlines. Optionally, you

can use the second (outUseHostDefault) parameter to determine whether the current database inherits its host default settings.

Note: Even though the name of the method is getDataSourceName, it really retrieves the database name. This is purely cosmetic, and just happens to be how the APIs were spelled when they were originally designed.

```
public int getDataSourceName( LassoValue outName,
                              BoolValue  outUseHostDefault,
                              LassoValue outUsernamePassword );
```

getDataHost()

Use this function when you want to ask Lasso Professional 8 what database host is being operated on. On return, LassoValue will contain the name and port of the database host.

```
public int getDataHost( LassoValue outHost,
                        LassoValue outUsernamePassword );
```

getDataHost2()

Same as getDataHost() but allows the usage of a host schema parameter for JDBC data sources.

```
public int getDataHost2( LassoValue outHost,
                         LassoValue outSchema,
                         LassoValue outUsernamePassword );
```

getSchemaName()

Use this function when you want to ask Lasso Professional what schema is being operated on for a JDBC data source. For instance, if you're being asked to perform a search, then you would call this function to retrieve the name of the schema which Lasso Professional is asking you to use for the search. It corresponds to the value of the parameter -Schema='blah' passed to inlines.

```
public int getSchemaName( LassoValue outName );
```

getTableName()

Use this function when you want to ask Lasso Professional what table is being operated on. For instance, if you're being asked to perform a search, then you would call this function to retrieve the name of the table which Lasso Professional is asking you to search. It corresponds to the value of the parameter -Layout='blah' or -Table='blah' passed to inlines.

```
public int getTableName( LassoValue outName );
```

getSkipRows()

You can ask Lasso Professional to tell you how many records should be skipped during a search by calling this function. It corresponds to the value of the -SkipRecords parameter in the inline search which is being executed at the moment your data source function is being called.

```
public int getSkipRows( IntValue outRows );
```

getMaxRows()

You can ask Lasso Professional to tell you the maximum number of records to be returned during a search by calling this function. It corresponds to the value of the -MaxRecords parameter in the inline search which is being executed at the moment your data source function is being called.

```
public int getMaxRows( IntValue outRows );
```

getPrimaryKeyColumn()

You can ask Lasso Professional to tell you which field is being used as the primary key. This value corresponds to the -KeyField parameter value used in the inline.

```
public int getPrimaryKeyColumn( LassoValue outColumn );
```

getInputColumnCount()

Tells how many fields were sent as parameters to the inline. For instance, if an LDML programmer wants to append a new record to a table, and passes in name, address, city, state, zip with values for each field, then this function will return the number 5 to indicate that five fields were passed to the inline. You can then retrieve the values of each of these parameters by calling getInputColumn by index, once per field. This function is smart enough to ignore parameters which are not fields, such as -Database, -Layout, etc.

```
public int getInputColumnCount( IntValue outCount );
```

getInputColumn()

Retrieve the name and value of field data parameters from the inline, starting at index zero. If five fields were entered into the inline, then you can retrieve each of their names and values by calling this function five times, once per field.

```
[Inline: -Database='MyDatabase', -Table='Main', 'MyFirstField'='Bill',
'MySecondField'='Ted', -Search]
```

In the above example, calling getInputColumn(0, v) will fill the v variable with v.name=MyFirstField, v.data=Bill. Notice it is smart enough to ignore well-known parameters such as -Table, thus only retrieving field information.

```
public int getInputColumn( int index, LassoValue outColumn );
```

getSortColumnCount()

Analogous to getInputColumnCount, this method retrieves the number of sort columns which were specified in the inline code. It basically counts how many -SortField parameters were passed. You can use this count to tell you how many times to enumerate through calls to getSortColumn.

```
public int getSortColumnCount( IntValue outCount );
```

getSortColumn()

Analogous to lasso_getInputColumn(), this function retrieves the names of sort parameters, starting at index zero. After calling this, the data field of outColumn variable will contain a String with the name of the sort field.

```
public int getSortColumn( int index, LassoValue outColumn );
```

getRowID()

Retrieves the current specified record ID (datasource-specific).

```
public int getRowID( IntValue outId );
```

setRowID()

Sets the record ID of the added record. After your custom LCAPI data source finishes adding a record to a database, it can call this function to let the caller know what the unique record ID of the added record was.

In FileMaker, this record ID is a standard feature of all records in its tables. In MySQL, this value is 0 unless there exists an AUTO_INCREMENT column. Results are not guaranteed for all database server software.

```
public int setRowID( int id );
```

findInputColumn()

Analogous to getInputColumn, except that it searches by name instead of index. If you already know the name of a field parameter you're interested in, then you can ask for the value of that parameter which was passed into the inline.

```
[Inline: -Database='MyDatabase', -Table='Main', 'MyFirstField'='Bill',
'MySecondField'='Ted', -Search]
```

In the example above, calling findInputColumn("MySecondField", outColumn) will fill the outColumn variable's data member with v.data=Ted.

```
public int findInputColumn( String name, LassoValue outColumn );
```

getLogicalOp()

Call this to retrieve the logical operator (OP_AND, OP_OR) which was passed to this inline. It corresponds to the value of -LogicalOperator passed into the inline. This function simply retrieves a single logical operator parameter. For more complex logical operations, with multiple operators, you will have to design a convention whereby you name your input fields

in some unique way, and then retrieve those custom logical operators using the getInputColumn function in a particular order that matches your convention.

```
public int getLogicalOp( IntValue outOp );
```

getReturnColumnCount()

Queries Lasso Professional to return the number of columns (fields) that are expected to be returned from a search operation. This counts how many -ReturnField parameters were encountered.

```
public int getReturnColumnCount( IntValue outCount );
```

getReturnColumn()

Once you know how many return columns are expected (from getReturnColumnCount), then you can enumerate through them to get their fieldnames. Use this information to retrieve field data from your database table, and populate the result rows when asked to perform a search operation.

```
public int getReturnColumn( int index, LassoValue outColumn );
```

addColumnInfo()

In order to return a row of data from your data source (perhaps as a result of a search), you must first indicate what the structure of the table columns is. Call this function for as many table columns as your database has, providing the fieldname, true/false if nulls are OK in this field, the field type (numeric, string, date, etc), and field protection (readonly, writeable, etc).

```
public int addColumnInfo( String name,
                          int nullOK,
                          int type,
                          int protection );
```

addResultRow()

Call this method once per row of records you want to return (perhaps from a search operation). You may choose to return an array of Strings, or construct an array of byte arrays that contain data for each of your fields (binary data is OK).

```
public int addResultRow( String[] columns );
public int addResultRow( byte[][] columns );
```

setNumRowsFound()

Corresponds to [Found_Count] in LDML. Call this when you know how many records your data source is going to return, and make sure you call addResultRow this many times in order to populate the rows.

```
public int setNumRowsFound( int num );
```

Semaphore Methods

createSem()

Creates a named semaphore sufficient for synchronizing multithreaded operations, which should be deleted after they are used. The Lasso Connector for MySQL example creates one of these at initialization time, and destroys it at terminate time.

```
public int createSem( String name );
```

destroySem()

Destroys a named semaphore that was created by the createSem method.

```
public int destroySem( String name );
```

acquireSem()

Attempts to acquire a lock on a semaphore, and waits until the owning thread has released the semaphore before acquiring the lock and continuing execution.

```
public int acquireSem( String name );
```

releaseSem()

Releases a locked semaphore so that other threads waiting for the semaphore can continue execution.

```
public int releaseSem( String name );
```

com.omnipilot.lassopro.LassoDSModule

Base class for all datasource modules. LassoDSModules are used to manipulate data sources. LassoDSModules are looked up by the datasource names they claim to support. They are instantiated once and used repeatedly by Lasso.

registerDSModule()

Your code must call this once at startup (from within your registerLassoModule() method) to register a data source with Lasso Professional. When Lasso encounters a data source request for moduleName, it calls the Java method methodName.

```
protected void registerDSModule( String datasourceName,
                                 String methodName,
                                 int flags,
                                 String moduleName,
                                 String description);
```

DS_METHOD_PROTOTYPE()

A prototype for all datasource action methods registered by registerDSModule. Since methods are being looked up by name, they must match exactly the values passed in a methodName parameter of the registerDSModule call.

```
public int DS_METHOD_PROTOTYPE( LassoCall lasso,
                                int action,
                                LassoValue data );
```

com.omnipilot.lassopro.LassoEncodings

Constants for the various text encoding methods.

ENCODE_BREAK

Static variable in class omnipilot.lasso.LassoTagEncodings.

```
public static final int ENCODE_BREAK
```

ENCODE_DEFAULT

Static variable in class com.omnipilot.lassopro.LassoEncodings.

```
public static final int ENCODE_DEFAULT
```

ENCODE_NONE

Static variable in class com.omnipilot.lassopro.LassoEncodings.

```
public static final int ENCODE_NONE
```

ENCODE_RAW

Static variable in class com.omnipilot.lassopro.LassoEncodings.

```
public static final int ENCODE_RAW
```

ENCODE_SMART

Static variable in class com.omnipilot.lassopro.LassoEncodings.

```
public static final int ENCODE_SMART
```

ENCODE_STRICT_URL

Static variable in class com.omnipilot.lassopro.LassoEncodings.

```
public static final int ENCODE_STRICT_URL
```

ENCODE_URL

Static variable in class com.omnipilot.lassopro.LassoEncodings.

```
public static final int ENCODE_URL
```

ENCODE_XML

Static variable in class com.omnipilot.lassopro.LassoEncodings.

```
public static final int ENCODE_XML
```

com.omnipilot.lassopro.LassoErrors

Constants for the various error codes which can be returned by your module.

NO_ERR

Static variable in class com.omnipilot.lassopro.LassoErrors.

public static final int NO_ERR

Assert

Static variable in class com.omnipilot.lassopro.LassoErrors.

public static final int Assert

StreamReadError

Could not write to stream.

public static final int StreamReadError

StreamWriteError

Could not read from stream.

public static final int StreamWriteError

Memory

Generic memory error.

public static final int Memory

InvalidMemoryObject

Invalid memory object.

public static final int InvalidMemoryObject

OutOfMemory

Not enough memory.

public static final int OutOfMemory

OutOfStackSpace

Stack overflow error.

public static final int OutOfStackSpace

CouldNotDisposeMemory

Error disposing an object.

public static final int CouldNotDisposeMemory

File

Generic file error.

public static final int File

FileInvalid

Trying to work with an invalid file.

public static final int FileInvalid

FileInvalidAccessMode

Trying to access a file in a mode that it doesn't support.

public static final int FileInvalidAccessMode

CouldNotCreateOrOpenFile

Could not create or open the file.

public static final int CouldNotCreateOrOpenFile

CouldNotCloseFile

Could not close the file.

public static final int CouldNotCloseFile

CouldNotDeleteFile

Could not delete the file.

public static final int CouldNotDeleteFile

FileNotFound

File does not exist.

public static final int FileNotFound

FileAlreadyExists

Trying to create a file that already exist.

public static final int FileAlreadyExists

FileCorrupt

File is corrupted.

public static final int FileCorrupt

VolumeDoesNotExist

Bad volume name.

public static final int VolumeDoesNotExist

DiskFull

No room left on disk.

public static final int DiskFull

DirectoryFull

No more items allowed in the directory.

public static final int DirectoryFull

IOError

I/O error.

public static final int IOError

InvalidPathname

Pathname is invalid.

public static final int InvalidPathname

InvalidFilename

Filename is invalid.

public static final int InvalidFilename

FileLocked

File is locked.

public static final int FileLocked

FileUnlocked

File is unlocked.

public static final int FileUnlocked

FileIsOpen

File is open.

public static final int FileIsOpen

FileIsClosed

File is closed.

public static final int FileIsClosed

BOF

Beginning of file reached.

public static final int BOF

EOF

End of file reached.

public static final int EOF

CouldNotWriteToFile

Unable to complete a write operation to the file.

public static final int CouldNotWriteToFile

CouldNotReadFromFile

Unable to complete a read operation from the file.

public static final int CouldNotReadFromFile

Resource

Unknown resource error.

public static final int Resource

ResNotFound

Resource not found.

public static final int ResNotFound

Network

Unknown networking error.

public static final int Network

InvalidUsername

The username supplied for the action is not valid.

public static final int InvalidUsername

InvalidPassword

The password supplied for the action is not valid.

public static final int InvalidPassword

InvalidDatabase

The database name supplied is not valid.

public static final int InvalidDatabase

NoPermission

General permissions error.

public static final int NoPermission

FieldRestriction

The specified action is restricted.

public static final int FieldRestriction

WebAddError

Add record error.

public static final int WebAddError

WebUpdateError

Update record error.

public static final int WebUpdateError

WebDeleteError

Delete record error.

public static final int WebDeleteError

InvalidParameter

An invalid parameter was passed to a function.

public static final int InvalidParameter

Overflow

Allocated memory was too small to hold the results.

public static final int Overflow

NilPointer

A pointer was null when it shouldn't have been.

public static final int NilPointer

UnknownError

Default when none of the cross-platform errors seem to fit.

public static final int UnknownError

FormattingLoopAborted

A looping tag was aborted; all looping tags must catch this exception.

public static final int FormattingLoopAborted

FormattingSyntaxError

Bad syntax used in a format file; parsing of the file was aborted.

public static final int FormattingSyntaxError

WebRequiredFieldMissing

Value missing for required field for Add.

public static final int WebRequiredFieldMissing

WebRepeatingRelatedField

Adding repeating related fields isn't supported.

public static final int WebRepeatingRelatedField

WebNoSuchObject

No records found.

public static final int WebNoSuchObject

WebTimeout

Operation timed out.

public static final int WebTimeout

WebActionNotSupported

Action not supported.

public static final int WebActionNotSupported

WebConnectionInvalid

The specified database was not found.

public static final int WebConnectionInvalid

WebModuleNotFound

The module was not found.

public static final int WebModuleNotFound

HTTPFileNotFound

The file was not found.

public static final int HTTPFileNotFound

DatasourceError

Third-party generic datasource error.

public static final int DatasourceError

com.omnipilot.lassopro.LassoOperators

Operator constants used throughout LJAPI.

Variables

OP_AND

Logical operator AND.

public static final int OP_AND

OP_ANY

Used for -Random database action.

public static final int OP_ANY

OP_BEGINS_WITH

Field search operator BW.

public static final int OP_BEGINS_WITH

OP_CONTAINS

Field search operator CN.

public static final int OP_CONTAINS

OP_DEFAULT

Same as OP_BEGINS_WITH.

public static final int OP_DEFAULT

OP_ENDS_WITH

Field search operator EW.

public static final int OP_ENDS_WITH

OP_EQUALS

Field search operator EQ.

public static final int OP_EQUALS

OP_GREATER_THAN

Field search operator GT.

public static final int OP_GREATER_THAN

OP_GREATER_THAN_EQUALS

Field search operator GTE.

public static final int OP_GREATER_THAN_EQUALS

OP_IN_FULL_TEXT

Field search operator FT.

public static final int OP_IN_FULL_TEXT

OP_IN_LIST

Static variable in class com.omnipilot.lassopro.LassoOperators.

public static final int OP_IN_LIST

OP_IN_REGEXP

Field search operator RX.

public static final int OP_IN_REGEXP

OP_LESS_THAN

Field search operator LT.

public static final int OP_LESS_THAN

OP_LESS_THAN_EQUALS

Field search operator LTE.

public static final int OP_LESS_THAN_EQUALS

OP_NO

Same as OP_NOT.

public static final int OP_NO

OP_NOT

Logical operator NOT.

public static final int OP_NOT

OP_NOT_BEGINS_WITH

Field search operator NBW.

public static final int OP_NOT_BEGINS_WITH

OP_NOT_CONTAINS

Field search operator NCN.

public static final int OP_NOT_CONTAINS

OP_NOT_ENDS_WITH

Field search operator NEW.

public static final int OP_NOT_ENDS_WITH

OP_NOT_EQUALS

Field search operator NEQ.

public static final int OP_NOT_EQUALS

OP_NOT_IN_LIST

Static variable in class com.omnipilot.lassopro.LassoOperators.

public static final int OP_NOT_IN_LIST

OP_NOT_IN_REGEXP

Field search operator NRX.

public static final int OP_NOT_IN_REGEXP

OP_OR

Logical operator OR.

public static final int OP_OR

com.omnipilot.lassopro.LassoParams

These constants signify the different parameters which can be retrieved from the LassoCall.getLassoParam method.

ModulesFolderPath

Path to the LassoModules folder.

public static final int ModulesFolderPath

StartupItemsFolderPath

Path to LassoStartup folder.

public static final int StartupItemsFolderPath

LassoErrorsFilePath

Path to Lasso error log file.

public static final int LassoErrorsFilePath

StorageHost

Location of Lasso MySQL datasource.

public static final int StorageHost

ScriptsRoot

Relative path to scripts root.

public static final int ScriptsRoot

ScriptsSiteRoot

Relative path to site scripts root (most likely includes ScriptsRoot).

public static final int ScriptsSiteRoot

com.omnipilot.lassopro.LassoTagModule

Base class for any tag module. Most tag modules output data onto the Web page, though some tags may perform other actions based on the parameters passed to them.

Every LassoTagModule must implement registerLassoModule method, and one or more methods with the same signature as TAG_METHOD_PROTOTYPE.

Lasso calls registerLassoModule once at startup to give the module a chance to register its tags. LassoTagModule must then call registerTagModule as many times as there are tags implemented by this module.

Variables

FLAG_INITIALIZER

Type initializer tags can have their own members.

public static final int FLAG_INITIALIZER

FLAG_SUBSTITUTION

Regular substitution tags.

public static final int FLAG_SUBSTITUTION

FLAG_ASYNC

Async tags run asynchronously in their own thread.

public static final int FLAG_ASYNC

FLAG_CONTAINER

Container tags have opening and closing. This flag will cause Lasso Professional to raise an error if the closing tag can't be found.

public static final int FLAG_CONTAINER

Methods

registerTagModule()

Use this method to register substitution tags implemented by your module. You should call registerTagModule as many times as there are tags implemented in your module.

moduleName parameter is the name of the module as returned by [Lasso_TagModuleName] LDML tag. tagName is the name of the custom LDML tag implemented by this module. One or more OR logical FLAG constants can be passed in the flags parameter to specify unique tag features. Finally, a description parameter can be used to provide optional tag info, such as brief description of the tag usage.

```
protected void registerTagModule( String moduleName,
                                  String tagName,
                                  String methodName,
                                  int flags,
                                  String description);
```

com.omnipilot.lassopro.LassoTypeRef

This class is used for creating and manipulating custom Lasso types. Unlike LassoValue or IntValue objects which store copies of the data, LassoTypeRef is merely a reference to a native object instance. Native objects exist for a fraction of a second while Lasso is processing a page, therefore the LassoTypeRef objects should never be stored or reused across multiple module invocations.

Variables

LASSO_ARRAY

The name of the built-in array type in Lasso Professional 8.

```
public static final String LASSO_ARRAY
```

LASSO_BOOLEAN

The name of the built-in boolean type in Lasso Professional 8.

```
public static final String LASSO_BOOLEAN
```

LASSO_DATE

The name of the built-in date type in Lasso Professional 8.

```
public static final String LASSO_DATE
```

LASSO_DECIMAL

The name of the built-in decimal type in Lasso Professional 8.

```
public static final String LASSO_DECIMAL
```

LASSO_INTEGER

The name of the built-in integer type in Lasso Professional 8.

```
public static final String LASSO_INTEGER
```

LASSO_MAP

The name of the built-in map type in Lasso Professional 8.

```
public static final String LASSO_MAP
```

LASSO_NULL

The name of the built-in null type in Lasso Professional 8.

```
public static final String LASSO_NULL
```

LASSO_PAIR

The name of the built-in pair type in Lasso Professional 8.

```
public static final String LASSO_PAIR
```

LASSO_STRING

The name of the built-in string type in Lasso Professional 8.

```
public static final String LASSO_STRING
```

LASSO_TAG

The name of the built-in tag type in Lasso Professional 8.

```
public static final String LASSO_TAG
```

Methods

isNull()

Returns true if this object does not refer to a valid type instance, which most likely would be a result of a failed LassoCall method.

```
public boolean isNull();
```

toString()

Returns string representation of the LassoTypeRef object. Overrides toString method in the class Object.

```
public String toString();
```

com.omnipilot.lassopro.LassoValue

Used for retrieving values from various LassoCall methods. Has name and data member variables of type String. The type member is set to one of the TYPE constants, reflecting the original type of the value before it was converted to string.

Variables

TYPE_ARRAY

Array type.

public static final int TYPE_ARRAY

TYPE_BLOB

Binary data.

public static final int TYPE_BLOB

TYPE_BOOLEAN

Boolean type.

public static final int TYPE_BOOLEAN

TYPE_CHAR

String type.

public static final int TYPE_CHAR

TYPE_CODE

Substitution tag code.

public static final int TYPE_CODE

TYPE_CUSTOM

Custom type.

public static final int TYPE_CUSTOM

TYPE_DATETIME

Date type.

public static final int TYPE_DATETIME

TYPE_DECIMAL

Decimal type.

public static final int TYPE_DECIMAL

TYPE_INT

Integer type.

public static final int TYPE_INT

TYPE_MAP

Map type.

public static final int TYPE_MAP

TYPE_NULL

Null type.

public static final int TYPE_NULL

TYPE_PAIR

Pair type.

```
public static final int TYPE_PAIR
```

TYPE_REFERENCE

Reference type.

```
public static final int TYPE_REFERENCE
```

Constructors

```
public LassoValue();
public LassoValue(int type);
public LassoValue(String data);
public LassoValue(String name, String data);
public LassoValue(String name, String data, int type);
```

Methods

data()

Returns the String object stored in the data field.

```
public String data();
```

name()

Returns the String object stored in the name field.

```
public String name();
```

setData()

Sets the value of the data field.

```
public String setData(String data);
```

setName()

Sets the value of the name field.

```
public String setName(String name);
```

setType()

Sets the value of the type field.

```
public int setType(int type);
```

toString()

Converts this object to String.

```
public String toString()
```

type()

Returns the original type of the data retrieved from one of the LassoCall methods: TYPE_CHAR for strings, TYPE_INT for integers, and so on.

For unnamed tag parameters, the type field is set to the type of the data stored in the data field. For named tag parameters, it reflects the type of the value member.

```
public int type();
```

com.omnipilot.lassopro.RequestParams

These constants signify the different parameters which can be retrieved from the LassoCall.getRequestParam method.

AddressKeyword

IP address of client browser.

```
public static final int AddressKeyword
```

ActionKeyword

Type of HTTP request (GET, POST, etc.).

```
public static final int ActionKeyword
```

ClientIPAddress

IP address of client browser.

```
public static final int ClientIPAddress
```

ContentLength

The length in bytes of the POST data sent from <form POST>.

```
public static final int ContentLength
```

ContentType

MIME header sent from client browser.

```
public static final int ContentType
```

FullRequestKeyword

All MIME headers, uninterpreted.

```
public static final int FullRequestKeyword
```

MethodKeyword

GET or POST, depending on <form method>.

```
public static final int MethodKeyword
```

PasswordKeyword

Password sent from browser.

```
public static final int PasswordKeyword
```

PostKeyword

HTTP object body (form data, etc.).

public static final int PostKeyword

ReferrerKeyword

URL of referring page.

public static final int ReferrerKeyword

ScriptName

Relative path from server root to a Lasso format file.

public static final int ScriptName

SearchArgKeyword

All text in URL after the question mark.

public static final int SearchArgKeyword

ServerName

IP address or host name of the server on which the Web server is running.

public static final int ServerName

ServerPort

IP port this hit came to (80 is common, 443 for SSL).

public static final int ServerPort

UserAgentKeyword

Browser name and type.

public static final int UserAgentKeyword

UserKeyword

Username sent from browser.

public static final int UserKeyword

A

Appendix A
LDML 8 Tag List

The LDML 8 tag list is not provided in this manual. Please see the online LDML 8 Reference for a full list of tags in Lasso Professional 8. The LDML 8 Reference allows the different tag categories to be browsed and for either preferred syntax tags to be viewed or legacy tags.

http://ldml.omnipilot.com

See the *Upgrading* section in this guide for lists of tags from earlier versions of Lasso that are not supported in Lasso Professional 8 and for specific tips regarding how to upgrade solutions that rely on these tags.

Use the *Index* at the end of this guide in order to find documentation about a particular tag within the manual. Every tag in Lasso Professional 8 should be documented in this guide and include an index entry.

B

Appendix B
Error Codes

This appendix contains a list of all known error codes that Lasso Professional 8, Lasso MySQL, or FileMaker Pro will return.

- *Lasso Professional 8 Error Codes* contains a list of all error codes which are generated by Lasso Professional 8.
- *Lasso MySQL Error Codes* contains a list of all error codes which are generated by Lasso MySQL or another MySQL data source.
- *FileMaker Pro Error Codes* contains a list of known error codes which are generated by FileMaker Pro when used as a data source.

In addition to the error codes described in this appendix, Lasso Professional 8 will report any unknown errors it receives from the operating system, Web server applications, or data source applications it communicates with. Please consult the documentation for the operating system and each application for more information about the error codes they may report.

For information about how to gracefully handle and recover from errors, please see the *Error Control* chapter.

Lasso Professional 8 Error Codes

The following *Table 1: Lasso Professional Error Codes* lists all of the native error codes of Lasso Professional 8. The error codes are listed in numerical order and are divided into general categories for easier reading. Many of the error codes descriptions contain helpful information about what to do to correct or prevent the error.

Table 1: Lasso Professional 8 Error Codes

Error Code	Description
0	No Error.
-609	The specified database was not found. Lasso could not find the specified database. This error usually occurs when a database is not open or not accessible by Lasso. Make sure the specified database is open.
-700	Could not find email format file. The format file specified by an -Email.Format command tag could not be found. Check the spelling of the file name. Make sure the path to the file is specified properly.
-701	All email tags must be assigned a value. In order for an email message to be sent, all five of the email parameters (-Email.Host, -Email.From, -Email.To, -Email.Subject, and -Email.Format) must be specified. Make sure you have specified values for all five parameters in your HTML form. Make sure the parameter names are spelled correctly.
Database Errors	
-800	Value missing for required field. The value of one or more required field was not specified. Make sure that all required fields are supplied with a value.
-801	Repeating related fields are not supported. An attempt to retrieve data from a repeating related field failed. Lasso does not support retrieving data from repeating related fields.
-802	Action not supported. The specified Lasso action is not supported by the specified database or data source.
-1712	Timeout. A database action timed out.
-1728	No records found. No records were found in the specified database.
-2000	The module was not found. The requested module was not found. Make sure that the module is located in the "Lasso Modules" folder and relaunch the Web server and/or Lasso.
-3000	A data source error has occured.

continued

Syntax Errors	
-9951	A syntax error occurred. Invalid or incorrect syntax was used. Correct the syntax.
-9952	A looping tag was aborted.
-9953	Unknown error.
Internal Errors	
-9954	A pointer was nil when it should not have been.
-9955	Overflow: Some memory passed to a function that was too small to hold the results.
-9956	An invalid parameter was passed to a function.
Action Errors	
-9957	Delete error. An error occurred while deleting a record from the specified database. Make sure that the database or data source is set to allow record deletion.
-9958	Update error. An error occurred while updating a record from the specified database. Make sure that the database or data source is set to allow records to be updated.
-9959	Add error. An error occurred while adding a record to the specified database. Make sure that the database or data source is set to allow records to be added.
-9960	Field restriction. A field security restriction prevented the action from being executed. Edit field security restrictions as configured within Lasso security.
Security Errors	
-9961	No permission. The current user is not allowed to perform the specified action. This could mean that a file suffix is not allowed by Lasso security. Edit user security permissions as configured within Lasso security.
-9962	Invalid database. The database or data source name is not valid.
-9963	Invalid password. The password supplied is not valid.
-9964	Invalid user name. The user name supplied is not valid.
-9965	Network error. An error occurred accessing the network connection. This error usually occurs while communicating with FileMaker Pro over TCP/IP. Try quitting and restarting the FileMaker Pro client.
-9966	Resource error.
-9967	Resource not found.

continued

File Errors	
-9968	Could not read from file.
-9969	Could not write to file.
-9970	End of file reached.
-9971	Beginning of file reached.
-9972	File is closed.
-9973	File already open with write permission.
-9974	File Unlocked.
-9975	File locked.
-9976	Invalid filename.
-9977	Invalid pathname.
-9978	I/O error.
-9979	Directory full.
-9980	Disk full.
-9981	Volume does not exist.
-9982	The file is corrupt.
-9983	File already exists.
-9984	Unauthorized file suffix or file not found.The error -9984 can be seen if you specify a format file with a file suffix which is not included in the Lasso Security settings. Also returned by file management tags.
-9985	Could not delete file.
-9986	Could not close file.
-9987	Could not create or open file.
-9988	Invalid access mode.
-9990	File error.

continued

Memory Errors	
-9991	Could not dispose memory.
-9992	Could not unlock memory.
-9993	Could not lock memory.
-9994	Lasso ran out of stack space. This error may occur when a Lasso format file contains too many deeply nested container tags. The [Variable] tag can be used in order to significantly reduce the number on nested tags in a format file.
-9995	Lasso ran out of memory. Increase the memory which is available to the server running Lasso.
-9996	Invalid memory object.
-9997	Memory error.
-9998	Error writing to stream.
-9999	Error reading from stream.

Lasso MySQL Error Codes

All of the known error codes in Lasso MySQL are listed in *Table 2: Lasso MySQL Error Codes*. Additional error codes may be reported if Lasso MySQL encounters an operating system error. If Lasso receives one of these error codes from Lasso MySQL or another MySQL data source then it will be passed on to the site visitor.

Table 2: Lasso MySQL Error Codes

Error Code	Description
1	Operation not permitted.
2	No such file or directory.
3	No such process.
4	Interrupted system call.
5	Input/output error.
6	Device not configured.
7	Argument list too long.
8	Exec format error.
9	Bad file descriptor.
10	No child processes.
11	Resource deadlock avoided.
12	Cannot allocate memory.
13	Permission denied.
14	Bad address.
15	Block device required.
16	Device busy.
17	File exists.
18	Cross-device link.
19	Operation not supported by device.
20	Not a directory.
21	Is a directory.
22	Invalid argument.
23	Too many open files in system.
24	Too many open files.
25	Inappropriate ioctl for device.

continued

26 – 50	
26	Text file busy.
27	File too large.
28	No space left on device.
29	Illegal seek.
30	Read-only file system.
31	Too many links.
32	Broken pipe.
33	Numerical argument out of domain.
34	Result too large.
35	Resource temporarily unavailable.
36	Operation now in progress.
37	Operation already in progress.
38	Socket operation on non-socket.
39	Destination address required.
40	Message too long.
41	Protocol wrong type for socket.
42	Protocol not available.
43	Protocol not supported.
44	Socket type not supported.
45	Operation not supported.
46	Protocol family not supported.
47	Address family not supported by protocol family.
48	Address already in use.
49	Can't assign requested address.
50	Network is down.

continued

51 – 75	
51	Network is unreachable.
52	Network dropped connection on reset.
53	Software caused connection abort.
54	Connection reset by peer.
55	No buffer space available.
56	Socket is already connected.
57	Socket is not connected.
58	Can't send after socket shutdown.
59	Too many references: can't splice.
60	Operation timed out.
61	Connection refused.
62	Too many levels of symbolic links.
63	File name too long.
64	Host is down.
65	No route to host.
66	Directory not empty.
67	Too many processes.
68	Too many users.
69	Disc quota exceeded.
70	Stale NFS file handle.
71	Too many levels of remote in path.
72	RPC struct is bad.
73	RPC version wrong.
74	RPC prog. not avail.
75	Program version wrong.

continued

76 – 150	
76	Bad procedure for program.
77	No locks available.
78	Function not implemented.
79	Inappropriate file type or format.
80	Authentication error.
81	Need authenticator.
82	Device power is off.
83	Device error.
84	Value too large to be stored in data type.
85	Bad executable (or shared library).
86	Bad CPU type in executable.
87	Shared library version mismatch.
88	Malformed Mach-O library file.
120	Didn't find key on read or update.
121	Duplicate key on write or update.
123	Someone has changed the row since it was read.
124	Wrong index given to function.
126	Index file is crashed / Wrong file format.
127	Record-file is crashed.
131	Command not supported by database.
132	Old database file.
133	No record read before update.
134	Record was already deleted (or record file crashed).
135	No more room in record file.
136	No more room in index file.
137	No more records (read after end of file).
138	Unsupported extension used for table.
139	Too big row (>= 16 M).
140	Wrong create options.
141	Duplicate unique key or constraint on write or update.
142	Unknown character set used.
143	Conflicting table definition between MERGE and mapped table.
144	Table is crashed and last repair failed.
145	Table was marked as crashed and should be repaired.

FileMaker Pro Error Codes

All of the known error codes for the FileMaker Pro Web Companion as of FileMaker Pro 5.5v3 are listed in *Table 3: FileMaker Pro Error Codes*. Additional error codes may be reported if FileMaker Pro encounters an operating system error. If Lasso receives one of these error codes from a FileMaker Pro data source, it will be passed on to the site visitor.

Table 3: FileMaker Pro Error Codes

Error Code	Description
-1	Unknown Error.
0	No Error.
1	User cancelled action.
2	Memory error.
3	Command is unavailable.
4	Command is unknown.
5	Command is invalid.
100 – 199	
100	File is missing.
101	Record is missing.
102	Field is missing.
103	Relation is missing.
104	Script is missing.
105	Layout is missing.
200 – 299	
200	Record access is denied.
201	Field cannot be modified.
202	Field access is denied.
203	No records in file to print or password doesn't allow print access.
204	No access to field(s) in sort order.
205	Cannot create new records; import will overwrite existing data.

continued

300 – 399	
300	The file is locked or in use.
301	Record is in use by another user.
302	Script definitions are in use by another user.
303	Paper size is in use by another user.
304	Password definitions are in use by another user.
305	Relationship or value list definitions are locked by another user.

400 – 499	
400	Find criteria is empty.
401	No records match the request.
402	Not a match field for a lookup.
403	Exceeding maximum record limit for demo.
404	Sort order is invalid.
405	Number of records specified exceeds number of records that can be omitted.
406	Replace/Reserialize criteria is invalid.
407	One or both key fields are missing (invalid relation).
408	Specified field has inappropriate data type for this operation.
409	Import order is invalid.
410	Export order is invalid.
411	Cannot perform delete because related records cannot be deleted.
412	Wrong version of FileMaker Pro used to recover file.

continued

500 – 599	
500	Date value does not meet validation entry options.
501	Time value does not meet validation entry options.
502	Number value does not meet validation entry options.
503	Value in field does not meet range validation entry options.
504	Value in field does not meet unique value validation entry options.
505	Value in field failed existing value validation test.
506	Value in field is not a member value of the validation entry option value list.
507	Value in field failed calculation test of validation entry option.
508	Value in field failed query value test of validation entry option.
509	Field requires a valid value.
510	Related value is empty or unavailable.

600 –699	
600	Print error has occurred.
601	Combined header and footer exceed one page .
602	Body doesn't fit on a page for current column setup .
603	Print connection lost.

700 – 799	
700	File is of the wrong file type for import.
701	Data Access Manager can't find database extension file.
702	Data Access Manager was unable to open the session.
704	Data Access Manager failed when sending a query.
705	Data Access Manager failed when executing a query.
706	EPSF file has no preview image.
707	Graphic translator can not be found.
708	Can't import the file or need color computer.
709	QuickTime movie import failed.
710	Unable to update Quicktime file reference, read-only.
711	Import Translator can not be found.
712	XTND version is incompatible.
713	Couldn't initialize the XTND system.
714	Insufficient password privileges to allow the operation.

continued

800 – 899	
800	Unable to create file on disk.
801	Unable to create temporary file on System disk.
802	Unable to open file .
803	File is single user or host cannot be found .
804	File cannot be opened as read-only in its current state .
805	File is damaged; use Recover command.
806	File cannot be opened with this version of FileMaker Pro.
807	File is not a FileMaker Pro file or is severely damaged .
808	Cannot open file because of damaged access privileges
809	Disk/volume is full.
810	Disk/volume is locked.
811	Temporary file cannot be opened as FileMaker Pro file.
812	Cannot open the file because it exceeds host capacity.
813	Record Synchronization error on network.
814	File(s) cannot be opened because maximum number is open.
815	Couldn't open lookup file.
816	Unable to convert file.

900 – 999	
900	General spelling engine error.
901	Main spelling dictionary not installed.
902	Could not launch the Help system.
903	Command cannot be used in a shared file.
904	Command can only be used in a file hosted under FileMaker Server.
950	Adding repeating related fields is not supported.
951	An unexpected error occurred.
971	The user name is invalid.
972	The password is invalid.
973	The database is invalid.
974	Permission denied.
975	The field has restricted access.
976	Security is disabled.
977	Invalid client IP address (FileMaker Pro 5.x only).
978	The number of allowed guests has been exceeded (FileMaker Pro 5.x only).

JDBC Error Codes

All error codes specific to JDBC data sources are listed in *Table 4: JDBC Error Codes*. Additional error codes may be reported if the JDBC data source encounters an operating system error. If Lasso receives one of these error codes from a JDBC data source, it will be passed on to the site visitor.

Table 4: JDBC Error Codes

Error Code	Description
-11000	Invalid Token Error. Invalid Lasso state token passed from Java.
-10999	Null Parameter Error. One of the required parameters was Null.

C

Appendix C
Copyright Notice

Copyright © 1996-2004 OmniPilot Software, Inc.

This copyright notice applies to all source code, examples and documentation provided in the Lasso 8 Language Guide provided in the Lasso Professional 8 software product from OmniPilot Software, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE X CONSORTIUM BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Except as contained in this notice, the name of OmniPilot Software, Inc. shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization from OmniPilot Software, Inc.

Lasso, Lasso Professional, Lasso Studio, Lasso Dynamic Markup Language, LDML, Lasso Service, Lasso Connector, Lasso Web Data Engine, OmniPilot and OmniPilot Software are trademarks of OmniPilot Software, Inc.

D

Appendix D
Index

SYMBOLS

!
  Boolean not 332
!=
  Boolean inequality 332
  Mathematical inequality 488
  String inequality 455
“
  HTML delimiter 113
312, 905
  URL delimiter 113
$
  Page variable 303
  Regular expressions 471
%
  Mathematical modulus 486
  Symbol overloading 940
%=
  Mathematical modulus 487
&
  URL delimiter 113
&&
  Boolean and 332
‘
  String delimiter 112
( )
  Regular expressions 472
*
  Mathematical multiplication 486
  Regular expressions 472
  String repetition 454
  Symbol overloading 940
*=
  Mathematical multiplication 487
  String repetition 455
+
  Date Addition 515
  Mathematical addition 486
  Regular expressions 472
  String concatenation 454
  Symbol overloading 940
++
  Symbol overloading 940
+=
  String concatenation 455
’
  Tag delimiter 112
-
  Date Subtraction 515
  Keyword prefix 112
  Mathematical subtraction 486
  String deletion 454
  Symbol overloading 940
--
  Symbol overloading 940
-=
  Mathematical subtraction 487
  String deletion 455
->
  Member symbol 112
-Schema 274
.
  Regular expressions 471
/
  Mathematical division 486
  Symbol overloading 940
  URL delimiter 113

//
LassoScript 74
LassoScript comment 112
/=
Mathematical division 487
:
Naming related fields 253
Tag delimiter 112
;
LassoScript delimiter 74, 78
Tag delimiter 112
<
HTML delimiter 113
Mathematical less than 488
String order 455
<=
Mathematical less than or equal 488
String order 455
<?LassoScript 74
=
Mathematical assignment 487
Parameter delimiter 112
URL delimiter 113
Variable assignment 303
==
Boolean equality 332
Mathematical equality 488
String equality 455
>
Mathematical greater than 488
String order 455
>=
Mathematical greater than or equal 488
String order 455
>>
String contains 455
Symbol overloading 939
?
Regular expressions 472
URL delimiter 113
?>
LassoScript delimiter 74
@ 315
References 312
[ ]
Regular expressions 471
Tag delimiter 112
\
Escape Character 671
Line Endings 438, 586
Regular Expressions 471
^
Regular expressions 471
{ }
Compound Expressions 112
Compound expressions 78, 764
Regular expressions 472
|
Regular expressions 472
||
Boolean or 332
Logical expressions 110

A

Abbreviation 93
[Abort] 331
Absolute Paths 61
Accessing PDF File Information 662
Action.Lasso 53, 54, 880
HTML forms 139
Paths 62
[Action_Param] 144
Database searches 167
Inline actions 136
[Action_Params] 144
Displaying the current action parameters 145
HTML forms 137
Inline actions 137
Linking to data 189
Results schema 146
[Action_Statement] 133, 144
Action Errors 354
Action Methods 52
Action Parameters 143
-Add 198
Requirements 201
Adding Content to Table Cells 680
Adding Records 200
Classic Lasso 199
FileMaker Pro 201
Lasso MySQL 201
MySQL 201
Security 199
Using an HTML form 203
Using an inline 202
Using a URL 203
[Admin_ChangeUser] 732
[Admin_CreateUser] 732
[Admin_GroupAssignUser] 732
[Admin_GroupListUsers] 732
[Admin_GroupRemoveUser] 732
[Admin_ListGroups] 732
Administration Tags 732

AND 170
  Performing an and search 170
[Array] 523
[Array->Contains] 524
[Array->Difference] 524
[Array->Find] 524
[Array->FindPosition] 524
[Array->First] 524
[Array->ForEach] 525
[Array->Get] 327, 525
[Array->Insert] 525
[Array->InsertFirst] 525
[Array->InsertFrom] 525
[Array->InsertLast] 525
[Array->Intersection] 525
[Array->Iterator] 525
[Array->Join] 525
[Array->Last] 525
[Array->Merge] 525
  Parameters 531
[Array->Remove] 525
[Array->RemoveAll] 526
[Array->RemoveFirst] 526
[Array->RemoveLast] 526
[Array->Reserve] 526
[Array->Reverse] 526
[Array->ReverseIterator] 526
[Array->Second] 526
[Array->Size] 327
[Array->Size] 526
[Array->Sort] 526
[Array->SortWith] 526
[Array->Union] 526
[Array] 290
Arrays 101, 518, 522
  Automatic string casting 453
  Compressing an array 727
  Converting a string to an array 469
  Creating 523
  Creating an empty array 523
  Creating a pair array 534
  Finding an element 532
  Finding a pair within an array 534
  Getting an element 527
  Getting the size 526
  Inserting an element 528
  Iterating through an array 527
  Joining into a string 529
  Looping through an array 527
  Members tags 524
  Merging arrays 531
  Pair arrays 534
  Passing values into an inline 143
  Removing an element 529
  Setting an element 527
  Sorting 535
  Types 522
Array Parameters 142
Asynchrnonous Tools
  See Thread Tools
Asynchronous Tags 911
  See also Custom Tags
  Accessing variables 913
  Calling custom tags 913
  Creating background processes 913
[Auth] 730
[Auth_Admin] 730
Authentication 38

B

Background Processes 913
Barcodes 686
Base 64 Encoding 338
Binary Formats 58
Binary Operations 492
Bit Operations 492
Blowfish 720
  Seeds 721
  Storing data securely 722
[Boolean] 290, 332
  Data Type 331
  False 332
  Symbols 332
  True 332
Boolean Operations 492
bw 169
[Bytes->Append] 479
[Bytes->BeginsWith] 479
[Bytes->Contains] 478
[Bytes->EndsWith] 479
[Bytes->ExportString] 479
[Bytes->Find] 478
[Bytes->Get] 478
[Bytes->GetRange] 478
[Bytes->ImportString] 479
[Bytes->Position] 479
[Bytes->Remove] 479
[Bytes->RemoveLeading] 479
[Bytes->RemoveTrailing] 479
[Bytes->Replace] 478
[Bytes->SetPosition] 479
[Bytes->SetRange] 478
[Bytes->SetSize] 478

[Bytes->Size] 478
[Bytes->Split] 479
[Bytes->SwapBytes] 480
[Bytes->Trim] 479
[Bytes] 290, 478
Bytes Types 477
Byte Order Mark 50

C

[Cache] 819
[Cache_Empty] 823
[Cache_Fetch] 823
[Cache_Object] 822
[Cache_Store] 822
Caching 818
Callback Tags 930
[Case] 323
Casting
  String 453
Cellular Phones 857
CGI 806
Character Encoding 50, 163, 198
character set 825
Character Sets 37
[Checked] 223, 261
  Displaying selected values 265
Check Boxes 227
Cipher Tags 724
Classic Lasso
  Adding records 199
  Database searches 164
  Deleting records 199
  Tokens 140
  Updating records 199
[Client_Address] 830
[Client_Browser] 830
[Client_ContentLength] 829
[Client_ContentType] 829
[Client_CookieList] 814
[Client_Cookies] 814
[Client_FormMethod] 829
[Client_GETArgs] 829
[Client_GETParams] 829
[Client_Headers] 829
[Client_IP] 830
[Client_Password] 829
[Client_POSTArgs] 829
[Client_POSTParams] 829
[Client_PostParams] 916
[Client_Type] 830
[Client_Username] 829
Client Tags 830
cn 169
Color
  Creating a random color 497
Command Tags 88
  Action tags 134
  Email sending tags 779
Comments
  LassoScript 71, 75
Comparators 521, 566
Complex Expressions 107
Compound Data Types 517
  Common Tags 520
  How to Select 519
Compound Expressions 764
  Evaluation rules 78, 764
  Running compound expressions 79, 765
  Tag data type 760
[Compress] 726
Compression 726
  Compressing an array 727
  Compressing a string 726
Conditional Expressions 108
  Symbols 109
Conditional Logic 317
  Complex conditionals 320
  If else conditionals 318
  Iterations 329
  Loops 325
  Nested conditionals 320
  Select statements 322
  While loops 330
Configuration Tags 745
Connection Parameters 984, 1032
Connection URL 984, 1032
Container Fields 266
Container Tags 87
  See also Custom Tags
  Defining 907
  Encoding 907
  LassoScript 71, 74
  Link tags 187
[Content_Type] 36, 37, 826
  Serving images and multimedia 614
  Serving WML 858
  XML data 648
Content Type 826
Control Tags 729
[Cookie] 814
[Cookie_Set] 814
  Parameters 815
Cookies 37, 813

Checking for cookie support 818
Retrieving cookies 816
Setting cookies 814
Creating Barcodes 686
Creating PDF Documents 655, 660
Creating Tables 678
Creating Text Content 663
Credit Cards
Checking whether a number is valid 772
Criteria 906
[Currency] 498
Custom Errors
Using the [Protect] … [/Protect] tags 368
Custom Error Pages 355
Defining 359
Testing 360
Custom Tags 889
Creating background processes 913
Criteria 906, 917
Defining asynchronous tags 911
Defining container tags 907
Defining process tags 896
Defining substitution tags 895
Encoding 897, 907
Error control 906
Getting a parameter value 902
Inspecting parameters 901
Libraries 919
Local variables 904
Named parameters 899, 902
Naming conventions 891
Optional parameters 898
Overloading 915
Page variables 904
Parameters 898
Parameters array 900
Parameters of the calling tag 903
Possible uses 890
Priority 915
Redefining 915
Referencing LassoApp Files 881
Remote procedure calls 909
Required parameters 898
Returning values 895, 897
Tags 893
Tag data type 759
Unnamed parameters 899, 903
Using global variables 311
Using references 315
XML-RPC 838, 854
Custom Types 921
Assignment tags 942
Automatic type conversions 932
Callback tags 930
Calling custom member tags 929
Comparison tags 937
Contains tag 939
[Define_Type] 924
Defining an onAssign callback 943
Defining an onCompare callback 938
Defining an onConvert callback 932
Defining an onCreate callback 932, 934
Defining an onDestory callback 934
Defining an unknown tag callback 934
Defining a >> callback 939
Defining a Type 924
Defining custom member tags 928
Destructor tags 933
Inheritance 944
Initialization tags 931
Instance variables 925
Libraries 946
Member tags 926
Naming conventions 922
Symbol overloading 935, 940, 943
Tags 924
Tag module code 1019
Tag module walk-through 1024
Unknown tags 934

D

-Database 154
Database 60
[Database_ChangeField] 229
Parameters 232
[Database_CreateField] 229
Parameters 232
[Database_CreateTable] 229
Parameters 230
[Database_FMContainer] 266
[Database_Name] 144
[Database_NameItem] 151
[Database_Names] 151
[Database_Names] … [/Database_Names]
Listing available databases 152
[Database_RealName] 151, 155
[Database_RemoveField] 229
[Database_RemoveTable] 229
[Database_SchemaNameItem] 274
[Database_SchemaNames] 274
[Database_TableNameItem] 151
[Database_TableNames] 151

[Database_TableNames] … [/Database_TableNames]
Listing available tables 152
DatabaseBrowser.LassoApp 874
Databases
Listing available databases 152
Listing fields 152
Required fields 153
Database Actions 134
Action parameters 143
Displaying the current parameters 145
Error codes 1092
FileMaker Pro error codes 1100
Finding all records 135
HTML forms 136
Inline method 132
JDBC error codes 1104
Lasso MySQL error codes 1096
Response tags 139
Results 148
Searching for records 135
Tags 134
Tokens 140
Database Errors 354
Database Field Paths 63
Database Schema
Showing 150
Database Searches 162
Classic Lasso 164
Complex queries 172
Detail links 194
Displaying data 179
Displaying results from a named inline 181
Displaying results out of order 181
Displaying search results 180
Error reporting 163
Field operators 168
Finding all records 176
Finding random records 178
HTML forms 167
Limiting returned fields 175
Linking to data 182
Logical operators 170
Manipulating the found set 174
Navigation links 190
Operators 168
Performing a logical not search 171
Random sorting 179
Results 173
Returning part of a found set 175
Returning unique field values 177
Searching records 165
Security 164
Security command tags 165
Sorting links 193
Sorting results 173, 174
Specifying field operators 169
Specifying username and password 165
Using a logical and operator 170
Using a logical or operator 171
Using inline tags 166
Data Output 281
Data Source Connector Code 1033
Data Source Connector Operation 983, 1031
Data Source Connector Tutorial 984, 1032
Data Source Connector Walk-Through 1043
Data Source Host 984, 1032
Data Type
Boolean 331
Data Types 290
Casting 292, 294
Custom member tags 927
Decimal 484
Integer 484
Member tags 927
Returning the type of a variable 742
XML 632
XML-RPC 836
Data Type Operation 1017
Data Type Tutorial 1017
[Date] 290, 503
[Date->Add] 513
[Date->Day] 508
[Date->DayofWeek] 508
[Date->DayofYear] 508
[Date->Difference] 514
[Date->Format] 507
[Date->GMT] 508
[Date->Hour] 508
[Date->Millisecond] 508
[Date->Minute] 508
[Date->Month] 508
[Date->Second] 508
[Date->Set] 507
[Date->SetFormat] 507
[Date->Subtract] 513
[Date->Time] 508
[Date->Week] 508
[Date->Year] 508
[Date_Add] 512
[Date_Difference] 512
[Date_Format] 503
[Date_GetLocalTimeZone] 503
[Date_GMTToLocal] 503

[Date_LocalToGMT] 503
[Date_Maximum] 503
[Date_Minimum] 503
[Date_SetFormat] 503
[Date_Subtract] 512
Dates 99, 499
  Accessors 508
  Formatting 505
  Math Operations 511, 515
Date Data Type 499
Date Format Symbols 505, 507
Date Math Symbols 515
Date Math Tags 511, 513
Date Tags 500
Daylight Savings Time 500, 511
[Decimal] 290, 485
[Decimal->SetFormat] 489
  Parameters 490
Decimals 98, 484
  Assignment symbols 487
  Automatic string casting 453
  Casting 485
  Comparing values 488
  Comparison symbols 488
  Formatting 489
  Formatting as currency 490
  Member tags 489
  Random numbers 496
  Rounding values 495
  Scientific notation 490
  Substitution tags 494
  Trignometry 497
  Using assignment symbols 487
  Using mathematical symbols 486
[Decode_Base64] 341
[Decode_Hex] 341
[Decode_HTML] 341
[Decode_QHeader] 341
[Decode_QuotedPrintable] 341
[Decode_URL] 341
[Decompress] 726
[Define_Tag] 893, 924
  Asynchronous tags 912
  Container tags 907
  Criteria 906, 917
  Defining custom member tags 928
  Parameters 894
  Priority 915
  RPC 909
  XML-RPC 838, 854
[Define_Type] 924
  Defining a type 924
-Delete 198
  Requirements 209
Deleting Records 209
  Classic Lasso 199
  Deleting several records 210
  Security 199
  Using an inline tag 209
Delimiters 112
  LassoScript 74
Detail Links 182
  Inline Lasso 194
Displaying data 179
-Distinct 173, 218, 240
Documentation 29
Documentation Conventions 31
Document Type Definition 630, 631
Domain Name Server 771
Downloading Files 811, 812
Drawing Graphics Direct to PDF Pages 683
DTD 631
  See also Document Type Definition
-Duplicate 198
  Requirements 211
Duplicating Records 211
  Using an inline tag 212
[Duration] 510
[Duration->Day] 510
[Duration->Hour] 510
[Duration->Minute] 510
[Duration->Month] 510
[Duration->Second] 510
[Duration->Week] 510
[Duration->Year] 510
[Duration] 290
Durations 100, 499
  Math Operations 511, 515
Duration Data Type 499
Duration Math Tags 513
Duration Tags 509

E

[Else] 318, 319
  Complex conditionals 320
Email 777
  Attachments 783
  Command tags 779
  Composing 786
  Downloading 791
  HTML 781
  Multiple recipients 781
  Parsing 796

Examples 799
POP 791
Queuing 786
Sending 779
Sending a message 780
SMTP 789
Structure 796
[Email_Compose] 786
[Email_Compose->AddAttachment] 786
[Email_Compose->AddHTMLPart] 787
[Email_Compose->AddPart] 787
[Email_Compose->AddTextPart] 786
[Email_Compose->Data] 787
[Email_Compose->From] 787
[Email_Compose->Recipients] 787
[Email_Immediate] 787
[Email_Parse] 798
[Email_Parse->Body] 798
[Email_Parse->Data] 798
[Email_Parse->Get] 798
[Email_Parse->Header] 798
[Email_Parse->Headers] 798
[Email_Parse->Mode] 798
[Email_Parse->Size] 798
[Email_POP] 792
[Email_POP->Authorize] 793
[Email_POP->Cancel] 792
[Email_POP->Close] 792
[Email_POP->Delete] 792
[Email_POP->Get] 792
[Email_POP->Headers] 792
[Email_POP->NOOP] 793
[Email_POP->Retrieve] 792
[Email_POP->Size] 792
[Email_POP->UniqueID] 792
[Email_Queue] 787
[Email_Send] 779
Parameters 779, 783, 785
[Email_SMTP] 789
[Email_SMTP->Close] 789
[Email_SMTP->Command] 789
[Email_SMTP->Open] 789
[Email_SMTP->Send] 789
[Encode_Base64] 341
[Encode_Break] 341
HTML encoding 337
[Encode_Hex] 341
[Encode_HTML] 341
HTML encoding 337
[Encode_QHeader] 341
[Encode_QuotedPrintable] 341
[Encode_Set]
Encoding for WML 860
Setting encoding within a LassoScript 72, 76
[Encode_Set] … [/Encode_Set] 340
Setting default encoding 340
-EncodeSmart 339
HTML encoding 337
[Encode_Smart] 341
HTML encoding 337
[Encode_SQL] 341
[Encode_StrictURL] 341
URL encoding 338
[Encode_URL] 341
URL encoding 337
[Encode_XML] 341
XML encoding 337
-EncodeBreak 339
HTML encoding 337
-EncodeHTML 339
Default encoding 336
HTML encoding 337
-EncodeNone 339
-EncodeStrictURL 339
URL encoding 338
-EncodeURL 339
URL encoding 337
-EncodeXML 339
Encoding for WML 860
XML encoding 337
Encoding 335
Base 64 encoding 338
Container tags 907
Controls 339
Custom tags 897
Default encoding 336
HTML 337
HTML encoding 337
Keywords 338
LassoScripts 336
Rules for encoding 335
Substitution tags 336
Tags 340
URL encoding 337
Using encoding tags 342
WML 860
XML 648
Encoding Keywords 95
Encryption 720
BlowFish 720
Cipher Tags 724
MD5 hash function 720
Storing and checking passwords 723
Storing data securely 722

ENUM MySQL Data Type 222
EQ 169
[Error_CurrentError] 362
[Error_NoRecordsFound] 364
[Error_SetErrorCode] 362
[Error_SetErrorMessage] 362
Errors
  Types 354
Error Codes 1091
  FileMaker Pro 1100
  JDBC 1104
  Lasso MySQL 1096
Error Control 287, 353, 364
  Checking for an error 364
  Displaying the current error message 362
  Executing code if an error is encountered 366
  Fail tags 367
  Handle tags 365
  Outputting debugging messages 366
  Post-processing 366
  Protecting a portion of a page 368
  Protect tags 368
  Reporting an error 367
  Response tags 361
  Setting the current error message 362
  Standard error tags 363
  Tags 361, 364
Error Messages 355
  Built-In 355
  Custom 358
Error Pages 360
  Custom 358
Error Reporting 278
  Adding records 198
  Checking for an error 163
  Database searches 163
  Deleting records 198
  Displaying the current error 163
  Updating records 198
[Event_Schedule] 736
  Parameters 736
  Scheduled actions 56
Events 756
  Waiting for a signal 757
Event Administration 736
Event Tags 736
EW 169
Examples
  [Ex_Background] 914
  [Ex_Bold] 905, 906
  [Ex_Concatenate] 903
  [Ex_Echo] 901
  [Ex_EmailAddress] 895
  [Ex_Font] 907
  [Ex_Fortune] 850, 910, 911
  [Ex_Greeting] 897, 902
  [Ex_Link] 908
  [Ex_Note] 899, 900
  [Ex_Print] 917
  [Ex_SendMail] 896, 912
  [Ex_Sum] 311, 905
  [Ex_TopStories] 910
  [Ex_UnnamedParams] 903
  [Form_Param] 916
Example PDF Files 688
Expressions 102
Extensible Markup Language 631
Extensible Stylesheet Language 631

F

False 332
[Field] 180, 253
  Database searches 167
  Displaying results out of order 181
  Displaying search results 180
  Returning related fields 253
[Field_Name] 151, 327
  Listing fields 153
  Parameters 152
[Field_Names] 149, 151
Fields
  Paths 63
  Required fields 153
Field Operators 168
[File->Close] 589
[File->Delete] 589
[File->Get] 589
[File->GetPosition] 589
[File->IsOpen] 589
[File->MoveTo] 589
[File->Name] 589
[File->Open] 588
[File->Path] 589
[File->Read] 588
[File->SetMode] 588
[File->SetPosition] 588
[File->SetSize] 589
[File->Size] 589
[File->Write] 588
[File] 587
[File_Copy] 581
[File_Create] 581
[File_CreationDate] 581

[File_CurrentError] 581
[File_Delete] 581
[File_Exists] 581
[File_GetLineCount] 582
[File_GetSize] 581
[File_IsDirectory] 582
[File_ListDirectory] 582
[File_ModDate] 582
[File_Move] 582
[File_Read] 582
[File_ReadLine] 582
[File_Rename] 582
[File_Serve] 594
[File_SetSize] 582
[File_Stream] 595
[File_Uploads] 592
[File_Write] 582
FileMaker
  Container Fields 266
FileMaker Pro 245
  Adding a record through a portal 255
  Adding a record with repeating fields 258
  Adding records 201
  Checking for databases 249
  Compatibility tips 248
  Deleting a record through a portal 257
  Deleting repeating field values 259
  Displaying a value list 261
  Displaying data 252
  Duplicating a record 211
  Error codes 1100
  Executing a script 269
  Field operators 168
  Key fields 250
  Listing databases 249
  Logical operators 170
  Performance tips 247
  Portals 254
  Record IDs 250
  Referencing a record by ID 251
  Related fields 253
  Repeating fields 257
  Returning a random record 178
  Returning the current record ID 250
  Returning values from a repeating field 258
  Scripts 268
  Sorting records 251
  Terminology 246
  Updating a record within a portal 256
  Updating a record with repeating fields 259
  Value lists 260
  XML templates 649
Files 577
  Error codes 1094
  Management 58
  Paths 34, 578
  Security 580
  Tags 577
File Permissions 599
File Suffixes 287, 580
File Uploads 591
-FindAll 162
  Inline action 135
  Requirements 177
-FMScript 268
-FMScriptPre 268
-FMScriptPreSort 268
[Form_Param]
  See [Action_Param]
  Redefining 916
Format Files 47, 278
  Action methods 52
  Character Encoding 50
  Editing 50
  File management 58
  Functional types 51
  HTML form actions 53
  Inline actions 54
  Naming 49
  Output formats 57
  Post-processing 366
  Scheduled actions 55
  Securing 56
  Specificing paths 60
  Startup actions 56
  Storage types 48
  Unicode 50
  URL actions 52
Forms 299
  See also HTML Forms
Form Parameters 299
Form Tags
  Preparing LassoApps 880
[Found_Count]
  Displaying the current found count 148
FT 169, 216
FTP 811
[FTP_GetFile] 812
[FTP_GetListing] 812
[FTP_PutFile] 811
Full-Text Search 217

G

GET Method 42
GIF
  Serving image files 614
[Global] 308
[Global_Defined] 308
[Global_Remove] 308
[Globals] 308
Global Variables 307
  Defining at startup 308
  Overriding a value 310
  Retrieving a value 309
  Setting a value 309
  Using within custom tags 311
GroupAdmin.LassoApp 874
GT 169
GTE 169

H

[Handle] … [/Handle] 365
[Header] … [/Header] 826
Header Tags 825, 826
Hexadecimals 492
  Creating a random color 497
Host Name 34
  Looking up an IP address 771
HTML 631
  Email 781
  Encoding 337
  Output formats 57
[HTML_Comment] 281
HTML Delimiters 113
HTML Format Files 48
HTML Forms 41
  Actions 53
  Adding a record 203
  Creating a pop-up menu 262
  Creating radio buttons 263
  Executing a FileMaker Pro script 269
  Format files 139
  GET method 42
  Inline actions 136
  Input syntax 77
  POST method 42
  Response tags 139
  Searching databases 167
  Setting values 140
  Updating a record 206
HTTP 810
[HTTP_GetFile] 810
HTTPS 806, 808
HTTP Content and Controls 805
HTTP Delimiters 113
HTTP Requests 34
HTTP Responses 35
HyperText Markup Language 631

I

[If] 318, 319
  Complex conditionals 320
  Error control 319
  LassoScript 319
  Nested conditionals 320
Illegal Characters 113
[Image->AddComment] 603
[Image->Annotate] 609
[Image->Blur] 607
[Image->Comments] 602
[Image->Composite] 610
[Image->Contrast] 606
[Image->Crop] 605
[Image->Depth] 601
[Image->Describe] 602
[Image->Enhance] 607
[Image->File] 602
[Image->FlipH] 605
[Image->FlipV] 605
[Image->Format] 601
[Image->Height] 601
[Image->Modulate] 606
[Image->Pixel] 602
[Image->ResolutionH] 601
[Image->ResolutionV] 601
[Image->Rotate] 605
[Image->Scale] 605
[Image->Sharpen] 607
[Image->Width] 601
image/gif 614
image/jpeg 614
[Image] 598, 600
ImageMagick 598
Images 597
  Generating the path to a file 613
  MIME types 614
  Serving an image file 615
Image Formats 599
Image Tags
  Preparing LassoApps 880
[Include]
  Preparing LassoApps 881
[Include] 288
[Include_Raw]

Serving images and multimedia 614
[Include_Raw] 288
[Include_URL] 806
Parameters 807
Includes 285
Include Paths 286
Include URLs 806
Index 1107
Inheritance 944
[Inline] … [/Inline] 132
Actions 54
Action parameters 143
Adding a record 202
Array parameters 142
Checking for an error 364
Database actions 132
Deleting a record 209
Deleting several records 210
Displaying results from a named inline 181
Displaying search results 180
Duplicating a record 212
Executing a FileMaker Pro script 269
-FindAll action 135
Finding all records 177
HTML forms 136
Linking to data 189
Nesting tags 142
Passing array parameters 143
-Search action 135
Searching databases 166
Specifying field operators 169
Specifying username and password 165
Updating a record 205
Updating several records 208
-InlineName
Displaying results 181
Inline Lasso 132
Detail links 194
Navigation links 190
Sorting links 193
Inline Log Level 132
Inline Name 132
Inline Tag 132
Installation Problems 354
Integer
Substitution tags 494
[Integer] 484
[Integer->BitAnd] 491
[Integer->BitClear] 491
[Integer->BitFlip] 491
[Integer->BitNot] 491
[Integer->BitOr] 491
[Integer->BitSet] 491
[Integer->BitShiftLeft] 491
[Integer->BitShiftRight] 491
[Integer->BitTest] 491
[Integer->BitXOr] 491
[Integer->SetFormat] 491
Parameters 492
[Integer] 290
Integers 98, 484
Assignment symbols 487
Automatic string casting 453
Bit operations 492
Casting to integer 484
Comparing values 488
Comparison symbols 488
Formatting 491
Formatting as hexadecimals 492
Hexadecimal output 492
Member tags 491
Random numbers 496
Rounding numbers 495
Using assignment symbols 487
Using mathematical symbols 486
IP Address
Looking up a host name 772
ISO-8859-1 36, 50, 163, 198
ISO 8859-1 393
[Iterate]
Implementing for custom types 926
[Iterate] … [/Iterate] 329
Array elements 329
Iterating through an array 527
Iterating through a map 541, 563
String characters 330
Iterators 522, 571
iText 654

J

[Java_Bean] 699
JavaBeans 697
JavaScript
Not processing square brackets 740
JDBC 271
Certification 272
Data sources 271
Error codes 1104
Tips for usage 272
JDBC Schema Tags 273
JPEG
Serving image files 614

K

-KeyField
  Using with FileMaker Pro 201
  Using with MySQL 201
[KeyField_Name] 144
[KeyField_Value] 144
  Using with FileMaker Pro 205
  Using with MySQL 205
Keywords
  Encoding 338

L

[Lasso_CurrentAction] 144
[Lasso_DataSourceIsFileMaker] 249
[Lasso_DatasourceIsFileMaker] 745
[Lasso_DatasourceIsLassoMySQL] 745
[Lasso_DatasourceIsMySQL] 215
[Lasso_DatasourceIsMySQL] 745
[Lasso_DatasourceIsSQLite] 239
[Lasso_DatasourceModuleName] 745
Lasso_Internal Database 984, 1032
[Lasso_TagExists] 745
[Lasso_TagModuleName] 745
[Lasso_UniqueID] 773
[Lasso_Version] 745
LassoApp
  Removing all LassoApps from the cache 875
[LassoApp_Create]
  Building LassoApps 884
  Parameters 884
[LassoApp_Link] 877
  Preparing <form> Tags 880
  Preparing <img> Tags 880
  Preparing [Include] Tags 881
  Preparing [Library] Tags 881
  Preparing [Link_…] Tags 881
  Preparing Links 879
LassoApps 871
  Administration 874
  Auto-Building databases 886
  Benefits 871
  Building 882
  Cache 875
  Compiling 882
  DatabaseBrowser.LassoApp 874
  Database Action Responses 877
  Defaults 873
  Disabling 874
  Enabling 874
  GroupAdmin.LassoApp 874
  Lasso Administration 874
  Lasso Security 886
  Lasso Startup 887
  Lasso Startup folder 878
  LDMLReference.LassoApp 874
  Naming conventions 885
  Preloading 875
  Preparing links 879
  Preparing solutions 879
  Referencing files within a LassoApp 877
  Removing a LassoApp from the cache 875
  RPC.LassoApp 850, 874, 909
  Run-time errors 885
  Serving 876
  Startup.LassoApp 874
  Tags 873
  Tips and techniques 885
  Uses 872
  Using the [LassoApp_Create] tag 884
  Using the LassoApp Builder 883
LassoScript
  Compound expressions 78, 764
LassoScripts
  Comments 71, 75
  Container tags 71, 74
  Converting from square bracket syntax 76
  Delimiters 74
  Encoding 336
  Setting default encoding 72, 76, 340
  Single tag 74
  Suppressing output 71, 75
LassoStartup 913, 917, 920
Lasso 8 Documentation 29
Lasso Administration 882, 984, 1032
Lasso Java API
  Debugging 1008
  Getting started 1005
  Requirements 1004
Lasso MySQL
  Adding records 201
  Error codes 1096
  Field operators 168
  Logical operators 170
  Random sorting 179
  Returning unique field values 177
  SQL encoding 338
Lasso Security
  Databases and tables 887
  Groups and tables 887
  LassoApps 886
  Tags 886
Lasso Service 59
  Paths 63

Lasso Startup
  Defining global variables 308
Lasso Startup Folder 878
Lasso Web Server Connector 59
Latin-1 37, 50, 163, 198, 393
LDML 81, 279
  Converting to LassoScript 76
  Format files 49
  Tag categories 90
  Tag listing 1089
  Tag types 82
LDMLReference.LassoApp 874
LDML 8 Reference 115
  Browsing 122
  Components 116
  Navigation 117
  Searching 118
  Sections of the interface 117
  Tag detail 123
  Tag listing 126
LDML 8 Tag Language 81
Leap Years 500
libCURL 806, 810, 811
Libraries 919, 946
[Library]
  Preparing LassoApps 881
[Library] 288
Library Files 51, 286
Line Endings 586
[Link_CurrentAction] … [/Link_CurrentAction] 187
  Linking to the current record 194
[Link_CurrentActionURL] 186
[Link_Detail] … [/Link_Detail] 187
  Linking to the current record 194
[Link_DetailURL] 186
[Link_FirstGroup] … [/Link_FirstGroup] 187
  Creating sort links 193
[Link_FirstGroupURL] 186
[Link_FirstRecord] … [/Link_FirstRecord] 187
[Link_FirstRecordURL] 186
[Link_LastGroup] … [/Link_LastGroup] 187
[Link_LastGroupURL] 186
[Link_LastRecord] … [/Link_LastRecord] 187
[Link_LastRecordURL] 186
[Link_NextGroup] … [/Link_NextGroup] 187
  Creating next links 190
[Link_NextGroupURL] 186
[Link_NextRecord] … [/Link_NextRecord] 187
[Link_NextRecordURL] 186
[Link_PrevGroup] … [/Link_PrevGroup] 187
  Creating previous links 190
[Link_PrevGroupURL] 186
[Link_PrevRecord] … [/Link_PrevRecord] 187
[Link_PrevRecordURL] 186
Linking to Data 182
  Container tags 187
  Tag parameters 183
  URL tags 186
Linking to PDF Files 693
Link Tags
  Preparing LassoApps 881
[List] 535
[List->Contains] 536
[List->Difference] 536
[List->Find] 536
[List->First] 536
[List->ForEach] 536
[List->Insert] 536
[List->InsertFirst] 536
[List->InsertFrom] 537
Lists 518, 535
  Members tags 536
List Array 522
Literals 103
LJAPI 6 vs. LCAPI 6 1003
LJAPI Class Reference 1050
LJAPI Interface Reference 1049
[Local] 312, 893, 905, 924
  # symbol 312, 905
  Instance variables 926
[Local_Defined] 893, 924
[Local_Remove] 893
[Locale_Format] 498
[Locals] 893, 924
Local Variables 904
  # symbol 312, 905
Lock
  Controlling access to a resource 753
  Thread lock 752
[Log] 714
[Log_Always] 713
[Log_Critical] 713
[Log_Deprecated] 713
Log_Destination_Console 716
Log_Destination_Database 716
Log_Destination_File 716
[Log_Detail] 713
Log_Level_Critical 716
Log_Level_Deprecated 716
Log_Level_Detail 716
Log_Level_SQL 716
Log_Level_Warning 716
[Log_SetDestination] 715

[Log_SQL] 713
[Log_Warning] 713
Logging 711
  Changing log destination preferences 716
  Destination codes 716
  Message level codes 716
  Preferences 715
  Resetting log destination preferences 717
Logical Errors 354
Logical Expressions 109
  Symbols 110
Logical Operators 170
  Performing an and search 170
  Performing an or search 171
  Performing a not search 171
[Loop] … [/Loop] 325, 327
  Array elements 327
  Display field names 327
  Looping through an array 527
  Looping through a map 542, 564
  Parameters 325
[Loop_Abort] 326, 327, 331
[Loop_Continue] 327
[Loop_Count] 325, 327, 331
Lower Case
  Strings 462
LT 169
LTE 169

M

Mac OS X File Permissions 581
[Map] 539
[Map] 290
Maps 101, 518, 539
  Automatic string casting 453
  Comparison to pair arrays 543
  Creating a map 539, 562
  Displaying an element 543, 565
  Getting a value 541, 563
  Inserting an element 542, 564
  Iterating through a map 541, 563
  Looping through a map 542, 564
  Member tags 540
  Removing an element 543, 565
Matchers 521, 568
Math 483
  See also Decimals; *See also* Integers
  Addition 487
  Expressions 106
  Scientific notation 490
  Symbols 107, 486
  Trigonometry 497
[Math_Abs] 494
[Math_ACos] 497
[Math_Add] 494
[Math_ASin] 497
[Math_ATan] 497
[Math_ATan2] 497
[Math_Ceil] 494
  Rounding numbers 495
[Math_ConvertEuro] 494
[Math_Cos] 497
[Math_Div] 494
[Math_Exp] 497
[Math_Floor] 494
  Rounding numbers 495
[Math_Ln] 497
[Math_Log10] 497
[Math_Max] 494
[Math_Min] 494
[Math_Mod] 494
[Math_Mult] 494
[Math_Pow] 497
[Math_Random] 494
  Parameters 496
[Math_RInt] 494
  Rounding numbers 495
[Math_Roman] 494
[Math_Round] 494
  Rounding numbers 495
[Math_Sin] 497
[Math_Sqrt] 497
[Math_Sub] 494
[Math_Tan] 497
-MaxRecords 154, 173
[MaxRecords_Value] 144
MD5 720
MD5 Hash Function
  Storing and checking passwords 723
Member Tags 85, 104, 297, 926
  Built-in 927
  Custom 927
  Decimal tags 489
  Integer tags 491
Member Tag Types 298
MIME Type
  Image files 614
Miscellaneous Tags 771
Multimedia 597
  Generating the path to a file 613
  MIME types 614
  Serving a multimedia file 615
MySQL

See Lasso MySQL
Adding and updating records 221
Adding records 201
Creating fields 231
Creating tables 228
Data sources 213
Error codes 1096
Field operators 168
Field types 233
Logical operators 170
Random sorting 179
Returning unique field values 177
Searching records 216
Search command tags 218
Search field operators 216
Security 214
SQL encoding 338
Tags 215
Tips for usage 214

N

Named Inlines
Displaying results 181
[Namespace_Import] 708
[Namespace_Load] 708
[Namespace_Unload] 708
[Namespace_Using] 708
Namespaces 705
Scope 705
Search Order 706
Third-Party 707
Name Server 771
Naming Conventions
Custom tags 891, 922
RPC tags 891
Navigation Links 182
Inline Lasso 190
NEQ 169
Nesting Tags 141
[NoProcess] … [/NoProcess] 739
NOT 170
Performing a not search 171
-Nothing 143
NRX 169, 217
[NSLookup] 771
[Null]
Member tags 927
Null
Data type 741
Value 220, 222
[Null->_UnknownTag] 931
[Null->DetachReference] 315, 927
Detaching a reference 314
[Null->DetachReference] 741
[Null->Dump] 741
[Null->FreezeType] 927
[Null->FreezeType] 741
[Null->FreezeValue] 927
[Null->FreezeValue] 741
[Null->Invoke] 927
[Null->IsA] 927
[Null->onConvert] 931
[Null->onCreate] 931
[Null->onDeserialize] 931
[Null->onDestroy] 931
[Null->onSerialize] 931
[Null->Parent] 927
[Null->Properties] 927
[Null->Properties]
Finding a tag 760
[Null->Properties] 741
[Null->RefCount] 927
[Null->RefCount] 315
[Null->Serialize] 927
Compressing an array 727
[Null->Serialize] 742
[Null->Type] 927
[Null->Type] 290, 742
[Null->Unserialize] 927
[Null->UnSerialize] 742
[Null] 741

O

On-Demand Libraries 707
-OpBegin
Complex queries 172
-OpEnd
Complex queries 172
OpenSSL 806
Operating System Errors 354
-Operator 168
[Operator_LogicalValue] 144
-OperatorBegin 168
-OperatorEnd 168
-OperatorLogical 168
Performing an and search 170, 171
Operators
Database searches 168
Field operators 168
Logical operators 170
Optimizing Tables 235
[Option] 223, 261

Creating a pop-up menu 262
OR 170
Performing an or search 171
[Output] 281
Automatic string casting 453
[Output_None] 281
Suppressing LassoScript output 75
Outputting Values 281
Output Formats 57
Output Suppressing 282

P

Page Variables 743, 744, 904
[Pair] 290
Pairs 518, 544
Automatic string casting 453
Creating a pair 544
Displaying an element 545
Getting an element 545
Member tags 545
Setting an element 545
Pair Arrays 523, 534
Comparison to maps 543
Parameters
Array 900
Array objects 142
Inspecting 901
Named 899
Optional 898
Required 898
Unnamed 899, 903
[Params] 893, 924
Parameters array 900
[Params_Up] 893, 901
Parameters from calling tag 903
[Parent] 945
-Password 165
Paths
Absolute 61
Action.Lasso 62
Database fields 63
Lasso Service 63
Relative 61
Specifying 60
PDF 653
Output Formats 57
PDF, Introduction to Creating PDF Files 598, 654
[PDF_Barcode] 686
[PDF_Doc] 600, 655, 660
[PDF_Doc->AddChapter] 659
[PDF_Doc->AddCheckBox] 672
[PDF_Doc->AddComboBox] 672
[PDF_Doc->AddHiddenField] 673
[PDF_Doc->AddList] 669
[PDF_Doc->AddPage] 659
[PDF_Doc->AddPasswordField] 672
[PDF_Doc->AddRadioButton] 672
[PDF_Doc->AddRadioGroup] 672
[PDF_Doc->AddResetButton] 673
[PDF_Doc->AddSelectList] 673
[PDF_Doc->AddSubmitButton] 673
[PDF_Doc->AddText] 658, 667
[PDF_Doc->AddTextArea] 672
[PDF_Doc->AddTextField] 672
[PDF_Doc->Circle] 684
[PDF_Doc->Close] 663
[PDF_Doc->CurveTo] 684
[PDF_Doc->DrawArc] 684
[PDF_Doc->DrawText] 669
[PDF_Doc->GetColor] 662
[PDF_Doc->GetHeaders] 662
[PDF_Doc->GetMargins] 662
[PDF_Doc->GetPageNumber] 659
[PDF_Doc->GetSize] 662
[PDF_Doc->InsertPage] 661
[PDF_Doc->Line] 684
[PDF_Doc->MoveTo] 684
[PDF_Doc->Rect] 684
[PDF_Doc->SetColor] 684
[PDF_Doc->SetFont] 662
[PDF_Doc->SetLineWidth] 684
[PDF_Doc->SetPageNumber] 659
[PDF_Font] 664
[PDF_Font->GetColor] 665
[PDF_Font->GetEncoding] 665
[PDF_Font->GetFace] 665
[PDF_Font->GetPSFontName] 665
[PDF_Font->GetSize] 665
[PDF_Font->GetFullFontName] 666
[PDF_Font->GetSupportedEncodings 665
[PDF_Font->IsTrueType] 665
[PDF_Font->SetColor] 665
[PDF_Font->SetEncoding] 665
[PDF_Font->SetFace] 665
[PDF_Font->SetSize] 665
[PDF_Font->SetUnderline] 665
[PDF_Image] 682
[PDF_List->Add] 670
[PDF_Read->PageCount] 660
[PDF_Read->PageSize] 660
[PDF_Serve] 694
[PDF_Table] 678

[PDF_Table->GetAbsWidth] 679
[PDF_Table->GetColumnCount] 679
[PDF_Table->GetRowCount] 679
[PDF_Table->Insert] 680
[Percent] 498
Performance Tips
  FileMaker Pro 247
Personal Digital Assistants 857
Pipes 757
  Processing messages 758
POP 791
  Examples 794
  Methodology 793
Pop-Up Menu 226
Portable Document Format 653
[Portal] … [/Portal] 253
  Returning portal values 255
Portals
  Adding a record through a portal 255
  Deleting a record through a portal 257
  FileMaker Pro 254
  Updating a record within a portal 256
Port Number 34
Post-Lasso 51
Post-Processing 366
POST Method 42
Pre-Lasso 51
[PriorityQueue] 546
Priority Queue 518, 546
  Member Tags 547
[Process] 739
  Processing code stored in a field 740
  Processing code stored in a variable 740
Process Tags 84, 739
  See also Custom Tags
  Defining 896
Process Tools
  See Thread Tools
Programming Fundamentals 277
[Protect] … [/Protect] 368
Protocol 34

Q

[Queue] 550
Queue 518, 550
  Member Tags 551

R

-Random 162
  Requirements 178
Random Numbers 496
Read/Write Lock 754
  Controlling access to a resource 755
[RecordID_Value] 250
[Records] … [/Records] 180
  Database actions 132
  Database searches 167
  Displaying results from a named inline 181
  Displaying search results 180
[Records_Array] 148, 149
Record ID 250
[Redirect_URL] 331, 809
[Reference] 312, 315
References 312
  Detaching a reference 314
  Types 314
  Using with custom tags 315
[Referrer] 187
[Referrer_URL] 186
RegExp
  See Regular Expressions
Regular Expressions 216, 470
  Combination symbols 472, 473
  Finding expressions 474
  Matching symbols 471
  Replacement symbols 472
  Replacing expressions 473
Relative Paths 61
Remote Procedure Call 631
Remote Procedure Calls 909
  Naming conventions 891
[Repeating] … [/Repeating] 253
  Returning values from a repeating field 258
[Repeating_ValueItem] 253
Repeating Fields 257
  Adding a record 258
  Deleting values 259
  Returning values 258
  Updatinga a record 259
[Repetition] 326
  Two column display 328
Request Tags 828
[Required_Field] 151
  Parameters 153
-Response
  Action.Lasso 139
[Response_FilePath] 144, 829
[Response_LocalPath] 829
[Response_Path] 829
[Response_Realm] 829
-ResponseAnyError 360
-ResponseLassoApp 877
Response Tags 139

Command tags 140
Error control 361
Results
Database searches 173
[Return] 893
Returning Values 895, 897
-ReturnField 173
Limiting returned fields 176
RPC 631
RPC.LassoApp 850, 874, 909
[Run_Children] 893
Defining container tags 907
RX 169, 216

S

Scheduled Events 55
Scheduling Events 735
Schema 630, 631
[Schema_Name] 274
[Scientific] 498
Scientific Notation 490
Scope 705
Scripts
Executing a Script 269
FileMaker Pro 268
-Search 162
Inline Action 135
Requirements 166
[Search_Arguments] 144
[Search_Arguments] … [/Search_Arguments]
Displaying search arguments 147
[Search_FieldItem] 144
[Search_OperatorItem] 144
[Search_ValueItem] 144
Searching Databases
See Database Searches
Security
Adding records 199
Command tags 165
Database searches 164
Deleting records 199
Duplicating records 199
Error codes 1093
Violations 354
[Select] 323
Data type 323
[Selected] 223, 261
Displaying selected values 264
[Self] 924, 945
[Self->Parent] 924
Semaphore 753
Controlling access to a resource 754
Series 518
[Server_Port] 831
[Server_Push] 825
Server Push 824
Server Tags 831
Serving PDF Files 693
Serving PDF Files to Client Browsers 694
-Session 344
[Session_AddVariable] 344, 345
[Session_End] 345
[Session_ID] 345
[Session_RemoveVariable] 345
[Session_Start] 344, 345
Parameters 346
Sessions 343
Adding variables 348
Deleting 349
Example 350
Removing variables 348
Starting a session 346
Tags 345
Using cookies 347
Using links 347
[Set] 554
Set 554
Member Tags 555
Sets 518
SET MySQL Data Type 222
SGML 631
-Show 327
Listing fields 153
Listing required fields 153
Requirements 150
Showing database schema 150
[Shown_Count]
Displaying the current shown count 148
Simple Object Access Protocol 841
-SkipRecords 154, 173
[SkipRecords_Value] 144
[Sleep] 914
[Sleep] 739
Smart HTML Encoding 337
SMTP 789
[Sort_Arguments] 144
[Sort_Arguments] … [/Sort_Arguments]
Displaying sort arguments 147
[Sort_FieldItem] 144
[Sort_OrderItem] 144
-SortField 173
Sorting FileMaker Pro results 252
Sorting

Arrays 535
Sorting Links 182
Inline Lasso 193
Sorting Records
FileMaker Pro 251
-SortOrder 173
-SortRandom 173, 218
Specifying Paths 60
SQL
Encoding 338
SQLite
Creating tables 243
Searching records 240
Search command tags 240
Security 238
Tips for usage 237
[SQLite_CreateDB] 243
SQL Server
XML templates 650
SQL Statements 154
Square Brackets
Converting to LassoScript 76
SSL 806
[Stack] 558
Stack 557
Member Tags 558
Stacks 519
Standard Generalized Markup Language 631
Startup.LassoApp 874
Startup Actions 56
Statement Only Inline 132
Storage Array 523
Storage Types 48
[String] 453
[String->Split]
Creating an array 524
[String] 290
[String_FindRegExp]
Examples 474
[String_ReplaceRegExp]
Examples 473
Strings 97, 451
Assignment 454
Automatic casting 453
Casting values to string 453
Comparisons 457
Concatenation 455
Converting case 462
Converting to an array 469
Deleting a substring 456
Expressions 105
Extracting part of a string 464
Finding regular expressions 474
Joining an array 529
Lenth 464
Manipulation tags 460
Regular expressions 470
Repeating a string 456
Replacing regular expressions 473
Splitting a string into an array 524
Symbols 106, 454
Style Sheets 640
Sub-Tags 103
Submitting Form Data to Lasso-Enabled Databases 677
Substitution Tags 82
See also Custom Tags
Defining 895
Encoding 336
Module code 1011
Module walk-through 1013
Operation 1009
Tutorial 1010
Symbols 102, 295
symbol 312, 905
Assignment 296
Boolean 332
Math 486
Overloading 935, 940, 943
Strings 454
Synonym 93
Syntax 82
Syntax Errors 354
System.ListMethods 835
System.MethodHelp 835
System.MethodSignature 835
System.MultiCall 835

T

[Table_Name] 144
[Table_RealName] 151
[Table_RealName] 155
Tables
Listing available tables 152
Listing fields 152
Required fields 153
[Tag->asAsync] 761
[Tag->asType] 761
[Tag->Description] 761
[Tag->Eval] 761
Evaluating compound expressions 78, 764
[Tag->Invoke] 761
[Tag->Run] 761

Parameters 761
Running compound expressions 79, 765
[Tags]
Finding tags 759, 760
Tags
See Custom Tags
Categories and naming 90
Listing 1089
Naming conventions 91
Synonynms and abbreviations 93
[Tags] 743
Tag Data Type 759
Member tags 761
Running a tag 762
Tag Types 82
[TCP_Open] 620
[TCP_Send] 621
Templates
XML 649
Test.Echo 835
Text Formats 57
Text Format Files 49
Third-Party Namespaces 707
[Thread_Event] 756
Member Tags 757
[Thread_Event->Signal] 757
[Thread_Event->SignalAll] 757
[Thread_Event->Wait] 757
[Thread_Lock] 752
Member tags 753
[Thread_Lock->Lock] 753
[Thread_Lock->Unlock] 753
[Thread_Pipe] 756
Member Tags 758
[Thread_Pipe->Get] 758
[Thread_Pipe->Set] 758
[Thread_RWLock] 752
Member tags 755
[Thread_RWLock->ReadLock] 755
[Thread_RWLock->ReadUnlock] 755
[Thread_RWLock->WriteLock] 755
[Thread_RWLock->WriteUnlock] 755
[Thread_Semaphore] 752
[Thread_Semaphore->Decrement] 754
[Thread_Semaphore->Increment] 754
[Thread_Semaphore]
Member tags 754
Thread Tools
Communications 756
Controlling access to a resource 753, 754, 755
Events 756
Lock 752
Pipes 757
Processing messages 758
Read/write lock 754
Semaphore 753
Waiting for a signal 757
Time 499
Time Zone 500
[Token_Value] 144
Tokens 140
Transient 922
Transient Member Tags 928
Transient Variable 925
[TreeMap] 561
Tree Map 561
Member Tags 562
Tree Maps 519
Trigonometry 497
True 332

U

UCS Transformation Format 37
Unicode 37, 50, 163, 198, 825
-Unique
Returning unique field values 177
Unique ID 773
See also Sessions
Universal Character Set 37
Unknown Tag Callback 934
-Update 198
Requirements 204
Updating Records 204
Classic Lasso 199
Security 199
Updating several records 208
Using an HTML form 206
Using a URL 207
Using inline tags 205
Upgrading
Email command tags 779
Uploading Files 812
Upper Case
Strings 462
URLs 34, 299
Action.Lasso 139
Actions 52
Adding a record 203
Encoding 337
Executing a FileMaker Pro script 270
Format files 139
Link Tags 186

Parameters 41, 299
Response tags 139
Syntax 77
Updating a record 207
-UseLimit 173, 218, 240
-Username 165
Using Fonts 664
UTF-8 36, 37, 50, 163, 198, 825

V

[Valid_CreditCard] 772
[Valid_Email] 772
Validation Tags 772
[Valid_URL] 772
[Value_List] … [/Value_List] 223, 261
[Value_ListItem] 223, 261
Displaying selected values 263
Value Lists 222, 260
Creating a pop-up menu 262
Creating radio buttons 263
Displaying all values 261
Displaying selected values 263
[Var] 303
[Var_Defined] 303
[Var_Remove] 303
[Variable] 303
[Variable_Defined] 303
Variables 283, 303
See also Global Variables; *See also* Local Variables
Accessing in asynchronous tags 913
Checking 306
Creating 283, 304
Local 904
Page 904
Returning data types 291
Returning the type of a variable 742
Returning values 284, 305
Server-side 343
Setting 285, 305
[Variables] 743

W

WAP 857
Tags 861
[WAP_IsEnabled]
Checking to see if current browser is WAP enabled 861
Web Application Servers 43
Web Browsers 33
Authentication 38
Cookies 37
Web Companion 246
Web Servers 40
Connectors 59
Errors 354
Web Serving Folder
Serving LassoApps 876
[While] 330, 331
Wireless Application Protocol 857
See also WAP
Wireless Devices 857
Wireless Markup Language 631, 857
See also WML
WML 631, 857
Encoding 860
Example 862
Formatting 858, 859
Forms 859
Links 859
Output formats 57
Serving 858

X

XML 629, 631
See also XML-RPC
Attributes 634
Children 634, 638
Contents 635
Customizing templates 652
Data type 632
Descendants 640
Document type definition 630
Encoding 337, 648
Extracting tags using an XPath 638
Extracting tags using XPath 639
Formatting 648
Format files 48
Member tags 633, 643, 644, 645
Output formats 57
Parameters 638
Root tag 638
Schema 630
Serving 647
Templates 649
Transformations 640
Wireless Markup Language 857
XPath 630, 635
XML-RPC 630, 631, 909
Built-in data types 836
Built-in methods 835
Calling a remote procedure 834

Calling multiple methods 835
Calling remote procedured (low-level) 837
Custom Tags 838, 854
Data Type 836
Listing available methods 834
Naming conventions 891
Processing incoming requests 838, 853
Processing tags 839
[XML_Extract] 636
[XML_RPCCall] 834
[XML_Serve] 647
Serving WML 858
[XML_Transform] 641
XPath 630, 631, 635
Conditional expressions 639
Extracting XML Tags 638, 639
Simple expressions 637
XSL 631
Transforming XML data 641
XSLT 631, 640
XSL Transformations 631