MarkLogic Server

Application Developer's Guide

MarkLogic 9
May, 2017

Last Revised: 9.0-7, September 2018

Copyright © 2019 MarkLogic Corporation. All rights reserved.

Table of Contents

Application Developer's Guide

1.0 Developing Applications in MarkLogic Server 17
1.1 Overview of MarkLogic Server Application Development 17
1.2 Skills Needed to Develop MarkLogic Server Applications 17
1.3 Where to Find Specific Information 18

2.0 Loading Schemas 21
2.1 Configuring Your Database 21
2.2 Loading Your Schema 22
2.3 Referencing Your Schema 23
2.4 Working With Your Schema 23
2.5 Validating XML and JSON Data Against a Schema 24
2.5.1 Validating Schemas using Schematron 24
2.5.2 Validating Schemas using the XQuery validate Expression 27
2.5.3 Validating JSON Documents against JSON Schemas 28

3.0 Understanding Transactions in MarkLogic Server 31
3.1 Terms and Definitions 32
3.2 Overview of MarkLogic Server Transactions 34
3.2.1 Key Transaction Attributes 35
3.2.2 Understanding Statement Boundaries 36
3.2.3 Single-Statement Transaction Concept Summary 38
3.2.4 Multi-Statement Transaction Concept Summary 39
3.3 Commit Mode 40
3.4 Transaction Type 41
3.4.1 Transaction Type Overview 41
3.4.2 Controlling Transaction Type in XQuery 42
3.4.3 Controlling Transaction Type in JavaScript 45
3.4.4 Query Transactions: Point-in-Time Evaluation 47
3.4.5 Update Transactions: Readers/Writers Locks 48
3.4.6 Example: Query and Update Transaction Interaction 51
3.5 Single vs. Multi-statement Transactions 51
3.5.1 Single-Statement, Automatically Committed Transactions 52
3.5.2 Multi-Statement, Explicitly Committed Transactions 52
3.5.3 Semi-Colon as a Statement Separator 57
3.6 Transaction Mode 59
3.6.1 Transaction Mode Overview 59
3.6.2 Auto Transaction Mode 61
3.6.3 Query Transaction Mode 61

3.6.4 Update Transaction Mode 62
3.6.5 Query-Single-Statement Transaction Mode 63
3.6.6 Multi-Auto Transaction Mode 64
3.7 Interactions with xdmp:eval/invoke 64
3.7.1 Isolation Option to xdmp:eval/invoke 64
3.7.2 Preventing Deadlocks 66
3.7.3 Seeing Updates From eval/invoke Later in the Transaction 68
3.7.4 Running Multi-Statement Transactions under xdmp:eval/invoke 69
3.8 Functions With Non-Transactional Side Effects 70
3.9 Reducing Blocking with Multi-Version Concurrency Control 71
3.10 Administering Transactions 71
3.11 Transaction Examples 72
3.11.1 Example: Multi-statement Transactions and Same-statement Isolation ...72
3.11.2 Example: Multi-Statement Transactions and Different-transaction Isolation 74
3.11.3 Example: Generating a Transaction Report With xdmp:host-status 75

4.0 Working With Binary Documents 77
4.1 Terminology 77
4.2 Loading Binary Documents 78
4.3 Configuring MarkLogic Server for Binary Content 78
4.3.1 Setting the Large Size Threshold 78
4.3.2 Sizing and Scalability of Binary Content 79
4.3.3 Selecting a Location For Binary Content 80
4.3.4 Monitoring the Total Size of Large Binary Data in a Forest 81
4.3.5 Detecting and Removing Orphaned Binaries 82
4.4 Developing Applications That Use Binary Documents 83
4.4.1 Adding Metadata to Binary Documents Using Properties 83
4.4.2 Downloading Binary Content With HTTP Range Requests 84
4.4.3 Creating Binary Email Attachments 86
4.5 Useful Built-ins for Manipulating Binary Documents 87

5.0 Importing XQuery Modules, XSLT Stylesheets, and Resolving Paths 89
5.1 XQuery Library Modules and Main Modules 89
5.1.1 Main Modules 89
5.1.2 Library Modules 90
5.2 Rules for Resolving Import, Invoke, and Spawn Paths 90
5.3 Module Caching Notes 92
5.4 Example Import Module Scenario 93

6.0 Library Services Applications 95
6.1 Understanding Library Services 95
6.2 Building Applications with Library Services 97
6.3 Required Range Element Indexes 97
6.4 Library Services API 98

6.4.1 Library Services API Categories 99
6.4.2 Managed Document Update Wrapper Functions 99
6.5 Security Considerations of Library Services Applications 100
6.5.1 dls-admin Role 100
6.5.2 dls-user Role 100
6.5.3 dls-internal Role 100
6.6 Transactions and Library Services 101
6.7 Putting Documents Under Managed Version Control 101
6.8 Checking Out Managed Documents 102
6.8.1 Displaying the Checkout Status of Managed Documents 102
6.8.2 Breaking the Checkout of Managed Documents 102
6.9 Checking In Managed Documents 103
6.10 Updating Managed Documents 103
6.11 Defining a Retention Policy 104
6.11.1 Purging Versions of Managed Document 104
6.11.2 About Retention Rules 105
6.11.3 Creating Retention Rules 105
6.11.4 Retaining Specific Versions of Documents 107
6.11.5 Multiple Retention Rules 108
6.11.6 Deleting Retention Rules 110
6.12 Managing Modular Documents in Library Services 111
6.12.1 Creating Managed Modular Documents 111
6.12.2 Expanding Managed Modular Documents 113
6.12.3 Managing Versions of Modular Documents 114

7.0 Transforming XML Structures With a Recursive typeswitch Expression 117
7.1 XML Transformations 117
7.1.1 XQuery vs. XSLT 117
7.1.2 Transforming to XHTML or XSL-FO 117
7.1.3 The typeswitch Expression 118
7.2 Sample XQuery Transformation Code 118
7.2.1 Simple Example 119
7.2.2 Simple Example With cts:highlight 120
7.2.3 Sample Transformation to XHTML 121
7.2.4 Extending the typeswitch Design Pattern 123

8.0 Document and Directory Locks 125
8.1 Overview of Locks 125
8.1.1 Write Locks 125
8.1.2 Persistent 125
8.1.3 Searchable 126
8.1.4 Exclusive or Shared 126
8.1.5 Hierarchical 126
8.1.6 Locks and WebDAV 126
8.1.7 Other Uses for Locks 126

8.2 Lock APIs 126
8.3 Example: Finding the URI of Documents With Locks 127
8.4 Example: Setting a Lock on a Document 128
8.5 Example: Releasing a Lock on a Document 128
8.6 Example: Finding the User to Whom a Lock Belongs 129

9.0 Properties Documents and Directories 131
9.1 Properties Documents 131
9.1.1 Properties Document Namespace and Schema 131
9.1.2 APIs on Properties Documents 133
9.1.3 XPath property Axis 134
9.1.4 Protected Properties 135
9.1.5 Creating Element Indexes on a Properties Document Element 135
9.1.6 Sample Properties Documents 135
9.1.7 Standalone Properties Documents 135
9.2 Using Properties for Document Processing 136
9.2.1 Using the property Axis to Determine Document State 136
9.2.2 Document Processing Problem 137
9.2.3 Solution for Document Processing 138
9.2.4 Basic Commands for Running Modules 139
9.3 Directories 139
9.3.1 Properties and Directories 140
9.3.2 Directories and WebDAV Servers 140
9.3.3 Directories Versus Collections 141
9.4 Permissions On Properties and Directories 141
9.5 Example: Directory and Document Browser 141
9.5.1 Directory Browser Code 142
9.5.2 Setting Up the Directory Browser 143

10.0 Point-In-Time Queries 145
10.1 Understanding Point-In-Time Queries 145
10.1.1 Fragments Stored in Log-Structured Database 145
10.1.2 System Timestamps and Merge Timestamps 146
10.1.3 How the Fragments for Point-In-Time Queries are Stored 146
10.1.4 Only Available on Query Statements, Not on Update Statements 147
10.1.5 All Auxiliary Databases Use Latest Version 147
10.1.6 Database Configuration Changes Do Not Apply to Point-In-Time Fragments 148
10.2 Using Timestamps in Queries 148
10.2.1 Enabling Point-In-Time Queries in the Admin Interface 148
10.2.2 The xdmp:request-timestamp Function 150
10.2.3 Requires the xdmp:timestamp Execute Privilege 150
10.2.4 The Timestamp Parameter to xdmp:eval, xdmp:invoke, xdmp:spawn ..150
10.2.5 Timestamps on Requests in XCC 151
10.2.6 Scoring Considerations 151

10.3 Specifying Point-In-Time Queries in xdmp:eval, xdmp:invoke, xdmp:spawn, and XCC 152
10.3.1 Example: Query Old Versions of Documents Using XCC 152
10.3.2 Example: Querying Deleted Documents 152
10.4 Keeping Track of System Timestamps 153
10.5 Rolling Back a Forest to a Particular Timestamp 155
10.5.1 Tradeoffs and Scenarios to Consider For Rolling Back Forests 155
10.5.2 Setting the Merge Timestamp 156
10.5.3 Notes About Performing an xdmp:forest-rollback Operation 156
10.5.4 General Steps for Rolling Back One or More Forests 157

11.0 System Plugin Framework 159
11.1 How MarkLogic Server Plugins Work 159
11.1.1 Overview of System Plugins 159
11.1.2 System Plugins versus Application Plugins 160
11.1.3 The plugin API 160
11.2 Writing System Plugin Modules 160
11.3 Password Plugin Sample 161
11.3.1 Understanding the Password Plugin 161
11.3.2 Modifying the Password Plugin 162

12.0 Using the map Functions to Create Name-Value Maps 165
12.1 Maps: In-Memory Structures to Manipulate in XQuery 165
12.2 map:map XQuery Primitive Type 165
12.3 Serializing a Map to an XML Node 166
12.4 Map API 166
12.5 Map Operators 167
12.6 Examples 167
12.6.1 Creating a Simple Map 168
12.6.2 Returning the Values in a Map 168
12.6.3 Constructing a Serialized Map 169
12.6.4 Add a Value that is a Sequence 169
12.6.5 Creating a Map Union 170
12.6.6 Creating a Map Intersection 171
12.6.7 Applying a Map Difference Operator 172
12.6.8 Applying a Negative Unary Operator 173
12.6.9 Applying a Div Operator 174
12.6.10 Applying a Mod Operator 175

13.0 Function Values 177
13.1 Overview of Function Values 177
13.2 xdmp:function XQuery Primitive Type 177
13.3 XQuery APIs for Function Values 178
13.4 When the Applied Function is an Update from a Query Statement 178
13.5 Example of Using Function Values 178

14.0 Reusing Content With Modular Document Applications 181
14.1 Modular Documents 181
14.2 XInclude and XPointer 182
14.2.1 Example: Simple id 183
14.2.2 Example: xpath() Scheme 183
14.2.3 Example: element() Scheme 183
14.2.4 Example: xmlns() and xpath() Scheme 184
14.3 CPF XInclude Application and API 184
14.3.1 XInclude Code and CPF Pipeline 184
14.3.2 Required Security Privileges—xinclude Role 185
14.4 Creating XML for Use in a Modular Document Application 185
14.4.1 <xi:include> Elements 186
14.4.2 <xi:fallback> Elements 186
14.4.3 Simple Examples 186
14.5 Setting Up a Modular Document Application 188

15.0 Controlling App Server Access, Output, and Errors 191
15.1 Creating Custom HTTP Server Error Pages 191
15.1.1 Overview of Custom HTTP Error Handling 191
15.1.2 Error Detail 192
15.1.3 Configuring Custom Error Handlers 193
15.1.4 Execute Permissions Are Needed On Error Handler Document for Modules Databases 194
15.1.5 Example: Custom Error Handler 194
15.2 Setting Up URL Rewriting for an HTTP App Server 195
15.2.1 Overview of URL Rewriting 195
15.2.2 Creating URL Rewrite Modules 197
15.2.3 Prohibiting Access to Internal URLs 199
15.2.4 URL Rewriting and Page-Relative URLs 199
15.2.5 Using the URL Rewrite Trace Event 200
15.3 Example: A Simple URL Rewriter 201
15.3.1 Create the Example App Server 201
15.3.2 Install the Example Content 202
15.3.3 Install the Example Application Module 202
15.3.4 Exercise the Example Application 203
15.3.5 Install the Rewriter 203
15.3.6 Configure the App Server to Use the Rewriter 204
15.3.7 Exercise the Rewriter 205
15.4 Outputting SGML Entities 206
15.4.1 Understanding the Different SGML Mapping Settings 206
15.4.2 Configuring SGML Mapping in the App Server Configuration 207
15.4.3 Specifying SGML Mapping in an XQuery Program 208
15.5 Specifying the Output Encoding 208
15.5.1 Configuring App Server Output Encoding Setting 208
15.5.2 XQuery Built-In For Specifying the Output Encoding 209

15.6 Specifying Output Options at the App Server Level 210

16.0 Creating an Interpretive XQuery Rewriter to Support REST Web Services ... 211
16.1 Terms Used in this Chapter 211
16.2 Overview of the REST Library 212
16.3 A Simple XQuery Rewriter and Endpoint 213
16.4 Notes About Rewriter Match Criteria 215
16.5 The options Node 217
16.6 Validating options Node Elements 219
16.7 Extracting Multiple Components from a URL 220
16.8 Handling Errors 222
16.9 Handling Redirects 222
16.10 Handling HTTP Verbs 224
16.10.1 Handling OPTIONS Requests 225
16.10.2 Handling POST Requests 227
16.11 Defining Parameters 228
16.11.1 Parameter Types 229
16.11.2 Supporting Parameters Specified in a URL 229
16.11.3 Required Parameters 230
16.11.4 Default Parameter Value 230
16.11.5 Specifying a List of Values 231
16.11.6 Repeatable Parameters 231
16.11.7 Parameter Key Alias 231
16.11.8 Matching Regular Expressions in Parameters with the match and pattern Attributes 232
16.12 Adding Conditions 234
16.12.1 Authentication Condition 235
16.12.2 Accept Headers Condition 235
16.12.3 User Agent Condition 235
16.12.4 Function Condition 236
16.12.5 And Condition 236
16.12.6 Or Condition 237
16.12.7 Content-Type Condition 237
16.13 Preparing to Run the Examples 237
16.13.1 Load the Example Data 237
16.13.2 Create the Example App Server 238

17.0 Creating a Declarative XML Rewriter to Support REST Web Services ... 241
17.1 Overview of the XML Rewriter 241
17.2 Configuring an App Server to use the XML Rewriter 242
17.3 Input and Output Contexts 242
17.3.1 Input Context 243
17.3.2 Output Context 244
17.4 Regular Expressions (Regex) 245

17.5 Match Rules 246
17.5.1 rewriter 247
17.5.2 match-accept 248
17.5.3 match-content-type 249
17.5.4 match-cookie 250
17.5.5 match-execute-privilege 251
17.5.6 match-header 252
17.5.7 match-method 254
17.5.8 match-path 255
17.5.9 match-query-param 258
17.5.10 match-role 260
17.5.11 match-string 261
17.5.12 match-user 262
17.6 System Variables 263
17.7 Evaluation Rules 265
17.7.1 add-query-param 266
17.7.2 set-database 267
17.7.3 set-error-format 268
17.7.4 set-error-handler 269
17.7.5 set-eval 270
17.7.6 set-modules-database 271
17.7.7 set-modules-root 272
17.7.8 set-path 272
17.7.9 set-query-param 273
17.7.10 set-transaction 274
17.7.11 set-transaction-mode 274
17.7.12 set-var 275
17.7.13 trace 276
17.8 Termination Rules 277
17.8.1 dispatch 277
17.8.2 error 279
17.9 Simple Rewriter Examples 280

18.0 Template Driven Extraction (TDE) 283
18.1 Security on TDE Documents 284
18.2 Template View Elements 286
18.3 JSON Template Structure 288
18.3.1 Collections 290
18.3.2 Directories 291
18.3.3 path-namespaces 291
18.3.4 Context 292
18.3.5 Variables 295
18.4 Template Dialect and Data Transformation Functions 296
18.4.1 Date and Time Functions 296
18.4.2 Logical Functions and Data validation 298
18.4.3 String Functions 298

18.4.4 Type Casting 299
18.4.5 Mathematical Functions 299
18.4.6 Miscellaneous Functions 299
18.5 Validating and Inserting a Template 300
18.6 Templates and Non-Conforming Documents 303
18.7 Enabling and Disabling Templates 303
18.8 Deleting Templates 304

19.0 Optic API for Multi-Model Data Access 305
19.1 Differences between the JavaScript and XQuery Optic APIs 307
19.2 Objects in an Optic Pipeline 308
19.3 Data Access Functions 311
19.3.1 fromView Examples 312
19.3.2 fromTriples Example 315
19.3.3 fromLexicons Examples 316
19.3.4 fromLiterals Examples 319
19.3.5 fromSQL Example 321
19.3.6 fromSPARQL Example 322
19.4 Kinds of Optic Queries 323
19.4.1 Basic Queries 323
19.4.2 Aggregates and Grouping 324
19.4.3 Row Joins 326
19.4.4 Document Joins 332
19.4.5 Union, Intersect, and Except 335
19.4.6 Document Queries 339
19.5 Processing Optic Output 340
19.6 Expression Functions For Processing Column Values 340
19.6.1 XQuery Libraries Required for Expression Functions 344
19.7 Functions Equivalent to Boolean, Numeric, and String Operators 346
19.8 Node Constructor Functions 348
19.9 Best Practices and Performance Considerations 350
19.10 Optic Execution Plan 350
19.11 Parameterizing a Plan 351
19.12 Exporting and Importing a Serialized Optic Query 352

20.0 Working With JSON 355
20.1 JSON, XML, and MarkLogic 355
20.2 How MarkLogic Represents JSON Documents 356
20.3 Traversing JSON Documents Using XPath 357
20.3.1 What is XPath? 358
20.3.2 Exploring the XPath Examples 358
20.3.3 Selecting Nodes and Node Values 359
20.3.4 Node Test Operators 360
20.3.5 Selecting Arrays and Array Members 362
20.4 Creating Indexes and Lexicons Over JSON Documents 364

20.5 How Field Queries Differ Between JSON and XML 365
20.6 Representing Geospatial, Temporal, and Semantic Data 366
20.6.1 Geospatial Data 366
20.6.2 Date and Time Data 367
20.6.3 Semantic Data 367
20.7 Character Set Restrictions 368
20.8 Document Properties 368
20.9 Serialization of Large Integer Values 368
20.10 Working With JSON in XQuery 369
20.10.1Constructing JSON Nodes 369
20.10.2Building a JSON Object from a Map 371
20.10.3Interaction With fn:data 371
20.10.4JSON Document Operations 372
20.10.5Example: Updating JSON Documents 373
20.10.6Searching JSON Documents 375
20.11 Working With JSON in Server-Side JavaScript 376
20.11.1Constructing JSON Nodes in JavaScript 377
20.11.2Updating JSON Documents from JavaScript 377
20.11.3Read-Only Access to JSON Document Contents 379
20.11.4Using Node Update Functions on JSON Documents 379
20.12 Converting JSON to XML and XML to JSON 381
20.12.1Conversion Philosophy 381
20.12.2Functions for Converting Between XML and JSON 382
20.12.3Understanding the Configuration Strategies For Custom Transformations 382
20.12.4Example: Conversion Using Basic Strategy 383
20.12.5Example: Conversion Using Full Strategy 383
20.12.6Example: Conversion Using Custom Strategy 385
20.13 Low-Level JSON XQuery APIs and Primitive Types 387
20.13.1Available Functions and Primitive Types 388
20.13.2Example: Serializing to a JSON Node 389
20.13.3Example: Parsing a JSON Node into a List of Items 389
20.14 Loading JSON Documents 390
20.14.1Loading JSON Document Using mlcp 391
20.14.2Loading JSON Documents Using the Java Client API 391
20.14.3Loading JSON Documents Using the Node.js Client API 391
20.14.4Loading JSON Using the REST Client API 391

21.0 Using Triggers to Spawn Actions 393
21.1 Overview of Triggers 393
21.1.1 Trigger Components 393
21.1.2 Databases Used By Triggers 394
21.2 Triggers and the Content Processing Framework 396
21.3 Pre-Commit Versus Post-Commit Triggers 397
21.3.1 Pre-Commit Triggers 397
21.3.2 Post-Commit Triggers 397

21.4 Trigger Events 398
21.4.1 Database Events 398
21.4.2 Data Events 398
21.5 Trigger Scope 399
21.6 Modules Invoked or Spawned by Triggers 400
21.6.1 Difference in Module Behavior for Pre- and Post-Commit Triggers 400
21.6.2 Module External Variables trgr:uri and trgr:trigger 401
21.7 Creating and Managing Triggers With triggers.xqy 401
21.8 Simple Trigger Example 402
21.9 Avoiding Infinite Trigger Loops (Trigger Storms) 404

22.0 Using Native Plugins 407
22.1 What is a Native Plugin? 407
22.2 How MarkLogic Server Manages Native Plugins 408
22.3 Building a Native Plugin Library 408
22.4 Packaging a Native Plugin 409
22.5 Installing a Native Plugin 410
22.6 Uninstalling a Native Plugin 411
22.7 Registering a Native Plugin at Runtime 411
22.8 Versioning a Native Plugin 412
22.9 Checking the Status of Loaded Plugins 413
22.10 The Plugin Manifest 414
22.11 Native Plugin Security Considerations 415
22.12 Native Plugin Example 416

23.0 Aggregate User-Defined Functions 417
23.1 What Are Aggregate User-Defined Functions? 417
23.2 In-Database MapReduce Concepts 417
23.2.1 What is MapReduce? 418
23.2.2 How In-Database MapReduce Works 418
23.3 Implementing an Aggregate User-Defined Function 419
23.3.1 Creating and Deploying an Aggregate UDF 419
23.3.2 Implementing AggregateUDF::map 420
23.3.3 Implementing AggregateUDF::reduce 422
23.3.4 Implementing AggregateUDF::finish 423
23.3.5 Registering an Aggregate UDF 424
23.3.6 Aggregate UDF Memory Management 425
23.3.7 Implementing AggregateUDF::encode and AggregateUDF::decode 427
23.3.8 Aggregate UDF Error Handling and Logging 428
23.3.9 Aggregate UDF Argument Handling 429
23.3.10Type Conversions in Aggregate UDFs 430

24.0 Redacting Document Content 433
24.1 Terms and Definitions 434
24.2 Introduction to Redaction 435

24.2.1 What is Redaction? 435
24.2.2 Express Redaction Requirements Through Rules 436
24.2.3 Apply Rules Using Multiple Interfaces 437
24.2.4 Protection of Redaction Logic 437
24.3 Example: Getting Started With Redaction 437
24.3.1 Installing the Source Documents 438
24.3.2 Installing the Rules 439
24.3.3 Understanding the Rules 440
24.3.4 Applying the Rules 441
24.4 Security Considerations 443
24.5 Defining Redaction Rules 445
24.5.1 Rule Definition Basics 445
24.5.2 Choosing a Redaction Strategy 447
24.5.3 Choosing a Redaction Function 448
24.5.4 Defining XML Namespace Prefix Bindings 449
24.5.5 Limitations on XPath Expressions in Redaction Rules 449
24.5.6 Defining Rules Usable on Multiple Document Formats 450
24.5.7 XML Rule Syntax Reference 452
24.5.8 JSON Rule Syntax Reference 454
24.6 Installing Redaction Rules 456
24.7 Applying Redaction Rules 457
24.7.1 Overview 458
24.7.2 Applying Rules Using mlcp 459
24.7.3 Applying Rules Using XQuery 459
24.7.4 Applying Rules Using JavaScript 460
24.7.5 No Guaranteed Ordering of Rules 461
24.8 Validating Redaction Rules 461
24.9 Built-in Redaction Function Reference 462
24.9.1 mask-deterministic 464
24.9.2 mask-random 467
24.9.3 conceal 470
24.9.4 redact-number 472
24.9.5 redact-us-ssn 474
24.9.6 redact-us-phone 477
24.9.7 redact-email 480
24.9.8 redact-ipv4 481
24.9.9 redact-datetime 483
24.9.10redact-regex 484
24.10 Example: Using the Built-In Redaction Functions 487
24.10.1Example Rule Summary 488
24.10.2Install the XML Rules 489
24.10.3Install the JSON Rules 492
24.10.4Apply the Rules 495
24.10.5Review the Results 496
24.11 User-Defined Redaction Functions 498
24.11.1Implementing a User-Defined Redaction Function 499

24.11.2Installing a User-Defined Redaction Function 500
24.12 Example: Using Custom Redaction Rules 503
24.12.1Example: Custom Redaction Using JavaScript 503
24.12.2Example: Custom Redaction Using XQuery 509
24.13 Using Dictionary-Based Masking 514
24.13.1Defining a Redaction Dictionary 514
24.13.2Installing a Redaction Dictionary 515
24.13.3Using a Redaction Dictionary 515
24.14 Example: Dictionary-Based Masking 516
24.14.1Install the Dictionaries 516
24.14.2Install the Rules 518
24.14.3Apply the Rules 521
24.15 Salting Masking Values for Added Security 523
24.16 Preparing to Run the Examples 526

25.0 Copyright 531
25.0 COPYRIGHT 531

26.0 Technical Support 533

1.0 Developing Applications in MarkLogic Server

This chapter describes application development in MarkLogic Server in general terms, and includes the following sections:

- Overview of MarkLogic Server Application Development
- Skills Needed to Develop MarkLogic Server Applications
- Where to Find Specific Information

This *Application Developer's Guide* provides general information about creating applications using MarkLogic Server. For information about developing search application using the powerful XQuery search features of MarkLogic Server, see the *Search Developer's Guide*.

1.1 Overview of MarkLogic Server Application Development

MarkLogic Server provides a platform to build application that store all kinds of data, including content, geospatial data, numeric data, binary data, and so on. Developers build applications using XQuery and/or Server-Side JavaScript to search the content and as a programming language in which to develop the applications. The applications can integrate with other environments via client APIs (Java, Node.js, and REST), via other web services, or via an XCC interface from Java. It is possible to create entire applications using only MarkLogic Server, programmed entirely in XQuery or Server-Side JavaScript.

This *Application Developer's Guide* focuses primarily on techniques, design patterns, and concepts needed to use XQuery or Server-Side JavaScript to build content and search applications in MarkLogic Server. If you are using the Java Client API, Node.js Client API, or the REST APIs, some of the concepts in this guide might also be helpful, but see the guides about those APIs for more specific guidance. For information about developing applications with the Java Client API, see the *Java Application Developer's Guide*. For information about developing applications with the REST API, see *REST Application Developer's Guide*. For information about developring applications with the Node.js Client API, see *Node.js Application Developer's Guide*.

1.2 Skills Needed to Develop MarkLogic Server Applications

The following are skills and experience useful in developing applications with MarkLogic Server. You do not need to have all of these skills to get started, but these are skills that you can build over time as you gain MarkLogic application development experience.

- Web development skills (xHTML, HTTP, cross-browser issues, CSS, Javascript, and so on), especially if you are developing applications which run on an HTTP App Server.
- Overall understanding and knowledge of XML.
- XQuery skills. To get started with XQuery, see the *XQuery and XSLT Reference Guide*.
- JavaScript skills. For information about Server-Side JavaScript in MarkLogic, see the *JavaScript Reference Guide*.
- Understanding of search engines and full-text queries.

- Java, if you are using the Java Client API or XCC to develop applications. For details, see the *Java Application Developer's Guide* or the *XCC Developer's Guide*.
- Node.js, if you use the Node.js Client to develop applications. For more details, see the *Node.js Application Developer's Guide*.
- General application development techniques, such as solidifying application requirements, source code control, and so on.
- If you will be deploying large-scale applications, administration on operations techniques such as creating and managing large filesystems, managing multiple machines, network bandwidth issues, and so on.

1.3 Where to Find Specific Information

MarkLogic Server includes a full set of documentation, available at http://docs.marklogic.com. This *Application Developer's Guide* provides concepts and design patterns used in developing MarkLogic Server applications. The following is a list of pointers where you can find technical information:

- For information about installing and upgrading MarkLogic Server, see the *Installation Guide*. Additionally, for a list of new features and any known incompatibilities with other releases, see the *Release Notes*.
- For information about creating databases, forests, App Servers, users, privileges, and so on, see the *Administrator's Guide*.
- For information on how to use security in MarkLogic Server, see *Security Guide*.
- For information on creating pipeline processes for document conversion and other purposes, see *Content Processing Framework Guide*.
- For syntax and usage information on individual XQuery functions, including the XQuery standard functions, the MarkLogic Server built-in extension functions for updates, search, HTTP server functionality, and other XQuery library functions, see the *MarkLogic XQuery and XSLT Function Reference*.
- For information about Server-Side JavaScript in MarkLogic, see the *JavaScript Reference Guide*.
- For information on using XCC to access content in MarkLogic Server from Java, see the *XCC Developer's Guide*.
- For information on how languages affect searches, see Language Support in MarkLogic Server in the *Search Developer's Guide*. It is important to understand how languages affect your searches regardless of the language of your content.
- For information about developing search applications, see the *Search Developer's Guide*.
- For information on what constitutes a transaction in MarkLogic Server, see "Understanding Transactions in MarkLogic Server" on page 31 in this *Application Developer's Guide*.

- For other developer topics, review the contents for this *Application Developer's Guide*.
- For performance-related issues, see the *Query Performance and Tuning Guide*.

2.0 Loading Schemas

MarkLogic Server has the concept of a *schema database*. The schema database stores schema documents that can be shared across many different databases within the same MarkLogic Server cluster. This chapter introduces the basics of loading schema documents into MarkLogic Server, and includes the following sections:

- Configuring Your Database
- Loading Your Schema
- Referencing Your Schema
- Working With Your Schema
- Validating XML and JSON Data Against a Schema

For more information about configuring schemas in the Admin Interface, see the "Understanding and Defining Schemas" chapter of the *Administrator's Guide*.

2.1 Configuring Your Database

MarkLogic Server automatically creates an empty schema database, named *Schemas*, at installation time.

Every document database that is created references both a schema database and a security database. By default, when a new database is created, it automatically references *Schemas* as its schema database. In most cases, this default configuration (shown in the following figure) will be correct:



In other cases, it may be desirable to configure your database to reference a different schema database. It may be necessary, for example, to be able to have two different databases reference different versions of the same schema using a common schema name. In these situations, simply select the database from the drop-down schema database menu that you want to use in place of the default *Schemas* database. Any database in the system can be used as a schema database.

In select cases, it may be efficient to configure your database to reference itself as the schema database. This is a perfectly acceptable configuration which can be set up through the same drop-down menu. In these situations, a single database stores both content and schema relevant to a set of applications.

> **Note:** To create a database that references itself as its schema database, you must first create the database in a configuration that references the default *Schemas* database. Once the new database has been created, you can change its schema database configuration to point to itself using the drop-down menu.

2.2 Loading Your Schema

HTTP and XDBC Servers connect to document databases. Document insertion operations conducted through those HTTP and XDBC Servers (using `xdmp:document-load`, `xdmp:document-insert` and the various XDBC document insertion methods) insert documents into the document databases connected to those servers.

This makes loading schemas slightly tricky. Because the system looks in the schema database referenced by the current document database when requesting schema documents, you need to make sure that the schema documents are loaded into the current database's *schema database* rather than into the current *document database*.

There are several ways to accomplish this:

1. You can use the Admin Interface's load utility to load schema documents directly into a schema database. Go to the Database screen for the schema database into which you want to load documents. Select the load tab at top-right and proceed to load your schema as you would load any other document.

2. You can create an XQuery program that uses the `xdmp:eval` built-in function, specifying the `<database>` option to load a schema directly into the current database's schema database:

```
xdmp:eval('xdmp:document-load("sample.xsd")', (),
        <options xmlns="xdmp:eval">
            <database>{xdmp:schema-database()}</database>
        </options>)
```

3. You can create an XDBC or HTTP Server that directly references the schema database in question as its document database, and then use any document insertion function to load one or more schemas into that schema database. This approach should not be necessary.

4. You can create a WebDAV Server that references the Schemas database and then drag-and-drop schema documents in using a WebDAV client.

2.3 Referencing Your Schema

Schemas are automatically invoked by the server when loading documents (for conducting content repair) and when evaluating queries (for proper data typing). For any given document, the server looks for a matching schema in the schema database referenced by the current document database.

1. If a schema with a matching target namespace is not found, a schema is not used in processing the document.

2. If one matching schema is found, that schema is used for processing the document.

3. If there are more than one matching schema in the schema database, a schema is selected based on the precedence rules in the order listed:

   a. If the `xsi:schemaLocation` or `xsi:noNamespaceSchemaLocation` attribute of the document root element specifies a URI, the schema with the specified URI is used.

   b. If there is an import schema prolog expression with a matching target namespace, the schema with the specified URI is used. Note that if the target namespace of the import schema expression and that of the schema document referenced by that expression do not match, the import schema expression is not applied.

   c. If there is a schema with a matching namespace configured within the current HTTP or XDBC Server's Schema panel, that schema is used. Note that if the target namespace specified in the configuration panel does not match the target namespace of the schema document, the Admin Interface schema configuration information is not used.

   d. If none of these rules apply, the server uses the first schema that it finds. Given that document ordering within the database is not defined, this is not generally a predictable selection mechanism, and is not recommended.

2.4 Working With Your Schema

It is sometimes useful to be able to explicitly read a schema from the database, either to return it to the outside world or to drive certain schema-driven query processing activities.

Schemas are treated just like any other document by the system. They can be inserted, read, updated and deleted just like any other document. The *difference* is that schemas are usually stored in a secondary schema database, not in the document database itself.

The most common activity developers want to carry out with schema is to read them. There are two approaches to fetching a schema from the server explicitly:

1. You can create an XQuery that uses `xdmp:eval` with the `<database>` option to read a schema directly from the current database's schema database. For example, the following expression will return the schema document loaded in the code example given above:

```
xdmp:eval('doc("sample.xsd")', (),
  <options xmlns="xdmp:eval">
    <database>{xdmp:schema-database()}</database>
  </options>)
```

   The use of the `xdmp:schema-database` built-in function ensures that the `sample.xsd` document is read from the current database's schema database.

2. You can create an XDBC or HTTP Server that directly references the schema database in question as its document database, and then submit any XQuery as appropriate to read, analyze, update or otherwise work with the schemas stored in that schema database. This approach should not be necessary in most instances.

Other tasks that involve working with schema can be accomplished similarly. For example, if you need to delete a schema, an approach modeled on either of the above (using `xdmp:document-delete("sample.xsd")`) will work as expected.

2.5 Validating XML and JSON Data Against a Schema

This section describes two ways to validate your schemas:

- Validating Schemas using Schematron
- Validating Schemas using the XQuery validate Expression
- Validating JSON Documents against JSON Schemas

2.5.1 Validating Schemas using Schematron

You can use the Schematron feature in MarkLogic to validate your XML and JSON documents against schemas. Schematron is a rule based validation language expressed in XM that uses XPath to make assertions about the presence or absence of patterns in XML trees.

Schematron is an open source project on Github and licensed under MIT. MarkLogic supports the latest version of Schematron, called the "skeleton" XSLT implementation of ISO Schematron. See the Schematron XQuery and JavaScript API reference documentation for more information.

The open source XSLT based Schematron implementation can be found at:

https://github.com/Schematron/schematron.

For example, to use Schematron to validate an XML schema, do the following:

1. Open Query Console, and use the following query to insert the example schema document into the Schemas database.

   Note: The `queryBinding="xslt2"` attribute in the schema file directs Schematron to make use of the xslt 2.0 engine.

```
xdmp:document-insert("/userSchema.sch",
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron"
            queryBinding="xslt2" schemaVersion="1.0">
<sch:title>user-validation</sch:title>
<sch:phase id="phase1">
  <sch:active pattern="structural"></sch:active>
</sch:phase>
<sch:phase id="phase2">
  <sch:active pattern="co-occurence"></sch:active>
</sch:phase>
<sch:pattern id="structural">
  <sch:rule context="user">
    <sch:assert test="@id">user element should have an id
attribute</sch:assert>
    <sch:assert test="count(*) = 5">
          user element should have 5 child elements: name, gender,
          age, score and result
    </sch:assert>
    <sch:assert test="score/@total">score element should have a total
attribute</sch:assert>
    <sch:assert test="score/count(*) = 2">score element should have two
child elements</sch:assert>
  </sch:rule>
</sch:pattern>
<sch:pattern id="co-occurence">
  <sch:rule context="score">
    <sch:assert test="@total = test-1 + test-2">
         total score must be a sum of test-1 and test-2 scores
    </sch:assert>
    <sch:assert test="(@total gt 30 and ../result = 'pass') or
              (@total le 30 and ../result = 'fail')" diagnostics="d1">
        if the score is greater than 30 then the result should be
        'pass' else 'fail'
    </sch:assert>
  </sch:rule>
</sch:pattern>
<sch:diagnostics>
<sch:diagnostic id="d1">the score does not match with the
result</sch:diagnostic>
</sch:diagnostics>
</sch:schema>)
```

2. Switch Query Console to the Documents database and use the following `schematron:put` query to compile the `userSchema.sch` Schematron document and insert the generated validator XSLT into the Modules database.

```
xquery version "1.0-ml";

import module namespace schematron =
"http://marklogic.com/xdmp/schematron"
      at "/MarkLogic/schematron/schematron.xqy";

let $params := map:map()
let $_put := map:put($params, 'phase', '#ALL')
let $_put := map:put($params, 'terminate', fn:false())
let $_put := map:put($params, 'generate-fired-rule', fn:true())
let $_put := map:put($params, 'generate-paths', fn:true())
let $_put := map:put($params, 'diagnose', fn:true())
let $_put := map:put($params, 'allow-foreign', fn:false())
let $_put := map:put($params, 'validate-schema', fn:true())
return schematron:put("/userSchema.sch", $params)
```

3. In the Documents database, insert a document to be validated against the `userSchema.sch` schema.

```
xdmp:document-insert("user001.xml",
<user id="001">
  <name>Alan</name>
  <gender>Male</gender>
  <age>14</age>
  <score total="90">
    <test-1>50</test-1>
    <test-2>40</test-2>
  </score>
  <result>fail</result>
</user>)
```

4. In the Documents database, call the `schematron:validate` function to validate the `user001.xml` document against the `userSchema.sch` schema.

```
xquery version "1.0-ml";

import module namespace schematron =
"http://marklogic.com/xdmp/schematron"
      at "/MarkLogic/schematron/schematron.xqy";

schematron:validate(fn:doc("user001.xml"),
                    schematron:get("/userSchema.sch"))
```

2.5.2 Validating Schemas using the XQuery validate Expression

You can also use the XQuery `validate` expression to check if an element is valid according to a schema. For details on the validate expression, see Validate Expression in the *XQuery and XSLT Reference Guide* and see the W3C XQuery recommendation (http://www.w3.org/TR/xquery/#id-validate).

If you want to validate a document before loading it, you can do so by first getting the node for the document, validate the node, and then insert it into the database. For example:

```
xquery version "1.0-ml";

(:
   this will validate against the schema if it is in scope, but
   will validate it without a schema if there is no in-scope schema
:)
let $node := xdmp:document-get("c:/tmp/test.xml")
return
try { xdmp:document-insert("/my-valid-document.xml",
        validate lax { $node } )
    }
catch ($e) { "Validation failed: ",
             $e/error:format-string/text() }
```

The following uses strict validation and imports the schema from which it validates:

```
xquery version "1.0-ml";
import schema "my-schema" at "/schemas/my-schema.xsd";

(:
   this will validate against the specified schema, and will fail
   if the schema does not exist (or if it is not valid according to
   the schema)
:)
let $node := xdmp:document-get("c:/tmp/test.xml")
return
try { xdmp:document-insert("/my-valid-document.xml",
        validate strict { $node } )
    }
catch ($e) { "Validation failed: ",
             $e/error:format-string/text() }
```

2.5.3 Validating JSON Documents against JSON Schemas

You can use the `xdmp:json-validate` function to validate a JSON document against a JSON schema in the Schemas database. For example, the following JSON schema is in the Schemas database at the URL, `/schemas/example.json`:

```
{
  "language": "zxx",
  "$schema": "http://json-schema.org/draft-07/schema#",
  "properties": {
    "count": { "type":"integer", "minimum":0 },
    "items": { "type":"array",
               "items": {"type":"string", "minLength":1 } }
  }
}
```

You can validate the following node against the `example.json` schema as follows:

```
xdmp:json-validate(
  object-node{ "count": 3, "items": array-node{12} },
  "/schemas/example.json" )
```

You can also use the `xdmp:json-validate-node` function to validate JSON documents against ad hoc schema nodes. For example:

```
xdmp:json-validate-node(
  object-node{ "count": 3, "items": array-node{12} },
  object-node{
    "properties": object-node{
    "count": object-node{ "type":"integer", "minimum":0 },
    "items": object-node{ "type":"array",
       "items": object-node{"type":"string", "minLength":1 }
             }
    }
  }
)
```

3.0 Understanding Transactions in MarkLogic Server

MarkLogic Server is a transactional system that ensures data integrity. This chapter describes the transaction model of MarkLogic Server, and includes the following sections:

- Terms and Definitions
- Overview of MarkLogic Server Transactions
- Commit Mode
- Transaction Type
- Single vs. Multi-statement Transactions
- Transaction Mode
- Interactions with xdmp:eval/invoke
- Functions With Non-Transactional Side Effects
- Reducing Blocking with Multi-Version Concurrency Control
- Administering Transactions
- Transaction Examples

For additional information about using multi-statement and XA/JTA transactions from XCC Java applications, see the *XCC Developer's Guide*.

3.1 Terms and Definitions

Although transactions are a core feature of most database systems, various systems support subtly different transactional semantics. Clearly defined terminology is key to a proper and comprehensive understanding of these semantics. To avoid confusion over the meaning of any of these terms, this section provides definitions for several terms used throughout this chapter and throughout the MarkLogic Server documentation. The definitions of the terms also provide a useful starting point for describing transactions in MarkLogic Server.

Term	Definition
statement	An XQuery main module, as defined by the W3C XQuery standard, to be evaluated by MarkLogic Server. A main module consists of an optional prolog and a complete XQuery expression. A Server-Side JavaScript program (or "script") is considered a single statement for transaction purposes. Statements are either *query statements* or *update statements*, determined statically through static analysis prior to beginning the statement evaluation.
query statement	A statement that contains no update calls. Query statements have a read-consistent view of the database. Since a query statement does not change the state of the database, the server optimizes it to hold no locks or lightweight locks, depending on the type of the containing transaction.
update statement	A statement with the potential to perform updates (that is, it contains one or more update calls). A statement can be categorized as an update statement whether or not the statement performs an update at runtime. Update statements run with *readers/writers locks*, obtaining locks as needed for documents accessed in the statement.
transaction	A set of one or more statements which either all fail or all succeed. A transaction is either an *update transaction* or a *query (read-only) transaction*, depending on the transaction type and the kind of statements in the transaction. A transaction is either a *single-statement transaction* or a *multi-statement transaction*, depending on the commit mode at the time it is created.

Term	Definition
transaction mode	Controls the transaction type and the commit semantics of newly created transactions. If you need to control the transaction type and/or commit semantics of a transaction, you should usually set them individually, rather than setting transaction mode. For details, see "Transaction Mode" on page 59.
single-statement transaction	Any transaction created in `auto` commit mode. Single-statement transactions always contain only one statment and are automatically committed on successful completion or rolled back on error.
multi-statement transaction	A transaction created in explicit commit mode, consisting of one or more statements which *commit* or *rollback* together. Changes made by one statement in the transaction are visible to subsequent statements in the same transaction prior to commit. Multi-statement transactions must be explicitly committed by calling `xdmp:commit` or `xdmp.commit`.
query transaction	A transaction which cannot perform any updates; a read-only transaction. A transaction consisting of a single *query statement* in `auto` update mode, or any transaction created with `query` transaction type. Attempting to perform an update in a query transaction raises `XDMP-UPDATEFUNCTIONFROMQUERY`. Instead of acquiring locks, query transactions run at a particular system timestamp and have a read-consistent view of the database.
update transaction	A transaction that can perform updates (make changes to the database). A transaction consisting of a single update statement in `auto` commit mode, or any transaction created with `update` transaction type. Update transactions run with *readers/writers locks*, obtaining locks as needed for documents accessed in the transaction.
commit	End a transaction and make the changes made by the transaction visible in the database. Single-statement transactions are automatically committed upon successful completion of the statement. Multi-statement transactions are explicitly committed using `xdmp:commit`, but the commit only occurs if and when the calling statement successfully completes.
rollback	Immediately terminate a transaction and discard all updates made by the transaction. All transactions are automatically rolled back on error. Multi-statement transactions can also be explicitly rolled back using `xdmp:rollback`, or implicitly rolled back due to timeout or reaching the end of the session without calling `xdmp:commit`.

Term	Definition
system timestamp	A number maintained by MarkLogic Server that increases every time a change or a set of changes occurs in any of the databases in a system (including configuration changes from any host in a cluster). Each fragment stored in a database has system timestamps associated with it to determine the range of timestamps during which the fragment is valid.
readers/writers locks	A set of read and write locks that lock documents for reading and update at the time the documents are accessed. MarkLogic Server uses readers/writers locks during update statements. Because update transactions only obtain locks as needed, update statements always see the latest version of a document. The view is still consistent for any given document from the time the document is locked. Once a document is locked, any update statements in other transactions wait for the lock to be released before updating the document. For more details, see "Update Transactions: Readers/Writers Locks" on page 48.
program	The expanded version of some XQuery code that is submitted to MarkLogic Server for evaluation, such as a query expression in a `.xqy` file or XQuery code submitted in an `xdmp:eval` statement. The program consists not only of the code in the calling module, but also any imported modules that are called from the calling module, and any modules they might call, and so on.
session	A "conversation" with a database on a MarkLogic Server instance. The session encapsulates state information such as connection information, credentials, and transaction settings. The precise nature of a session depends on the context of the conversation. For details, see "Sessions" on page 56.
request	Any invocation of a program, whether through an App Server, through a task server, through `xdmp:eval`, or through any other means. In addition, certain client calls to App Servers (for example, loading an XML document through XCC, downloading an image through HTTP, or locking a document through WebDAV) are also requests.

3.2 Overview of MarkLogic Server Transactions

This section summarizes the following key transaction concepts in MarkLogic Server for quick reference.

- Key Transaction Attributes

- Understanding Statement Boundaries
- Single-Statement Transaction Concept Summary
- Multi-Statement Transaction Concept Summary

The remainder of the chapter covers these concepts in detail.

3.2.1 Key Transaction Attributes

MarkLogic supports the following transaction models:

- Single-statement transactions, which are automatically committed at the end of a statement. This is the default transaction model in MarkLogic.
- Multi-statement transactions, which can span multiple requests or statements and must be explicitly committed.

Updates made by a statement in a multi-statement transaction are visible to subsequent statements in the same transaction, but not to code running outside the transaction.

An application can use either or both transaction models. Single statement transactions are suitable for most applications. Multi-statement transactions are powerful, but introduce more complexity to your application. Focus on the concepts that match your chosen transactional programming model.

In addition to being single or multi-statement, transactions are typed as either update or query. The transaction type determines what operations are permitted and if, when, and how locks are acquired. By default, MarkLogic automatically detects the transaction type, but you can also explicitly specify the type.

The transactional model (single or multi-statement), commit mode (auto or explicit), and the transaction type (auto, query, or update) are fixed at the time a transaction is created. For example, if a block of code is evaluated by an `xdmp:eval` (XQuery) or `xdmp.eval` (JavaScript) call using `same-statement` isolation, then it runs in the caller's transaction context, so the transaction configuration is fixed by the caller, even if the called code attempts to change the settings.

The default transaction semantics vary slightly between XQuery and Server-Side JavaScript. The default behavior for each language is shown in the following table, along with information about changing the behavior. For details, see "Transaction Type" on page 41.

Langauge	Default Transaction Behavior	Alternative
XQuery	single-statement, auto-commit, with auto-detection of transaction type	Use the `update` prolog option to explicitly set the transaction type to `auto`, `update` or `query`. Use the `commit` option to set the commit mode to `auto` (single-statement) or `explicit` (multi-statement). Similar controls are available through options on functions such as `xdmp:eval` and `xdmp:invoke`.
Server-Side JavaScript	single-statement, auto-commit, with query transaction type	Use the `declareUpdate` function to explicitly set the transaction type to update and control whether the commit mode is auto (single-statement) or explicit (multi-statement). Auto detection of transaction type is not available. Similar controls are available through options on functions such as `xdmp.eval` and `xdmp.invoke`.

A statement can be either a query statement (read only) or an update statement. In XQuery, the first (or only) statement type determines the transaction type unless you explicitly set the transaction type. The statement type is determined through static analysis. In JavaScript, query statement type is assumed unless you explicitly set the transaction to update.

In the context of transactions, a "statement" has different meanings for XQuery and JavaScript. For details, see "Understanding Statement Boundaries" on page 36.

3.2.2 Understanding Statement Boundaries

Since transactions are often described in terms of "statements", you should understand what constitutes a statement in your server-side programming language:

- Transactional Statements in XQuery
- Transactional Statements in Server-Side JavaScript

3.2.2.1 Transactional Statements in XQuery

In XQuery, a statement for transaction purposes is one complete XQuery statement that can be executed as a main module. You can use the semi-colon separator to include multiple statements in a single block of code.

For example, the following code block contains two statements:

```
xquery version "1.0-ml";
xdmp:document-insert('/some/uri/doc.xml', <data/>);
(: end of statement 1 :)

xquery version "1.0-ml";
fn:doc('/some/uri/doc.xml');
(: end of statement 2 :)
```

By default, the above code executes as two auto-detect, auto-commit transactions.

If you evaluate this code as a multi-statement transaction, both statements would execute in the same transaction; depending on the evaluation context, the transaction might remain open or be rolled back at the end of the code since there is no explicit commit.

For more details, see “Semi-Colon as a Statement Separator” on page 57.

3.2.2.2 Transactional Statements in Server-Side JavaScript

In JavaScript, an entire script or main module is considered a statement for transaction purposes, no matter how many JavaScript statements it contains. For example, the following code is one transactional “statement”, even though it contains multiple JavaScript statements:

```
'use strict';
declareUpdate();
xdmp.documentInsert('/some/uri/doc.json', {property: 'value'});
console.log('I did something!');
// end of module
```

By default, the above code executes in a single transaction that completes at the end of the script. If you evaluate this code in the context of a multi-statement transaction, the transaction remains open after completion of the script.

3.2.3 Single-Statement Transaction Concept Summary

If you use the default model (single-statement, auto-commit), it is important to understand the following concepts:

Single-Statement Transaction Concept	Where to Learn More
Statements run in a transaction.	Single vs. Multi-statement Transactions
A transaction contains exactly one statement.	Single-Statement, Automatically Committed Transactions
Transactions are automatically committed at the end of every statement.	Single-Statement, Automatically Committed Transactions
Transactions have either update or query type. • Query transactions use a system time-stamp instead of locks. • Update transactions acquire locks.	Transaction Type
In XQuery, transaction type can be detected by MarkLogic, or explicitly set. In JavaScript, transaction type is assumed to be query unless you explicitly set it to update. Auto detection is not available.	Transaction Type
Updates made by a statement are not visible until the statement (transaction) completes.	Update Transactions: Readers/Writers Locks
In XQuery, semi-colon can be used as a statement/transaction separator to include multiple statements in a main module. Each JavaScript program is considered a statement for transaction purposes, regardless how many JavaScript statements it contains.	Understanding Statement Boundaries Single-Statement, Automatically Committed Transactions Semi-Colon as a Statement Separator

3.2.4 Multi-Statement Transaction Concept Summary

If you use multi-statement transactions, it is important to understand the following concepts:

Multi-Statement Transaction Concept	Where to Learn More
Statements run in a transaction.	Single vs. Multi-statement Transactions
A transaction contains one or more statements that either all succeed or all fail.	Multi-Statement, Explicitly Committed Transactions
Multi-statement transactions must be explicitly committed using `xdmp:commit` (XQuery) or `xdmp.commit` (JavaScript).	Multi-Statement, Explicitly Committed Transactions Committing Multi-Statement Transactions
Rollback can be implicit or explicit. For explicit rollback, use `xdmp:rollback` (XQuery) or `xdmp.rollback` (JavaScript).	Multi-Statement, Explicitly Committed Transactions Rolling Back Multi-Statement Transactions
Transactions have either update or query type. • Query transactions use a system time-stamp instead of acquiring locks • Update transactions acquire locks.	Transaction Type
In XQuery, transaction type can be detected by MarkLogic, or explicitly set. In JavaScript, transaction type is assumed to be query unless you explicitly set it to update. Auto detection is not available.	Transaction Type
Transactions run in a session.	Sessions
Sessions have a transaction mode that affects the following: • transaction type • commit semantics • how many statements a transaction can contain	Transaction Mode
Setting the commit mode to explicit always creates a multi-statement transaction, explit-commit transaction.	Single vs. Multi-statement Transactions Multi-Statement, Explicitly Committed Transactions

Multi-Statement Transaction Concept	Where to Learn More
Updates made by a statement are not visible until the statement completes.	Update Transactions: Readers/Writers Locks
Updates made by a statement are visible to subsequent statements in the same transaction while the transaction is still open.	Multi-Statement, Explicitly Committed Transactions
In XQuery, semi-colon can be used as a statement separator to include multiple statements in a transaction.	Understanding Statement Boundaries Semi-Colon as a Statement Separator

3.3 Commit Mode

A transaction can run in either auto or explicit commit mode.

The default behavior for a single-statement transaction is auto commit, which means MarkLogic commits the transaction at the end of a statement, as defined in "Understanding Statement Boundaries" on page 36.

Explicit commit mode is intended for multi-statement transactions. In this mode, you must explicitly commit the transaction by calling `xdmp:commit` (XQuery) or `xdmp.commit` (JavaScript), or explicitly roll back the transaction by calling `xdmp:rollback` (XQuery) or `xdmp.rollback` (JavaScript). This enables you to leave a transaction open across multiple statements or requests.

You can control the commit mode in the following ways:

- Set the XQuery prolog option `xdmp:commit` to auto or explicit.
- Call the JavaScript function `declareUpdate` with the `explicitCommit` option. Note that this affects both the commit mode and the transaction type. For details, see "Controlling Transaction Type in JavaScript" on page 45.
- Set the `commit` option when evaluating code with `xdmp:eval` (XQuery), `xdmp.eval` (JavaScript), or another function in the eval/invoke family. See the table below for complete list of functions supporting this option.
- Call `xdmp.set-transaction-mode` (XQuery) or `xdmp.setTransactionMode` (JavaScript). Note that this sets both the commit mode and the query type. For details, see "Transaction Mode" on page 59.

The following functions support `commit` and `update` options that enable you to control the commit mode (explicit or auto) and transaction type (update, query, or auto). For details, see the function reference for `xdmp:eval` or `xdmp.eval`.

XQuery	JavaScript
`xdmp:eval`	`xdmp.eval`
`xdmp:javascript-eval`	`xdmp.xqueryEval`
`xdmp:invoke`	`xdmp.invoke`
`xdmp:invoke-function`	`xdmp.invokeFunction`
`xdmp:spawn`	`xdmp.spawn`
`xdmp:spawn-function`	

3.4 Transaction Type

This section covers the following information related to transaction type. This information applies to both single-statement and multi-statement transactions.

- Transaction Type Overview
- Controlling Transaction Type in XQuery
- Controlling Transaction Type in JavaScript
- Query Transactions: Point-in-Time Evaluation
- Update Transactions: Readers/Writers Locks
- Example: Query and Update Transaction Interaction

3.4.1 Transaction Type Overview

Transaction type determines the type of operations permitted by a transaction and whether or not the transaction uses locks. Transactions have either update or query type. Statements also have query or update type, depending on the type of operations they perform.

Update transactions and statements can perform both query and update operations. Query transactions and statements are read-only and may not attempt update operations. A query transaction can contain an update statement, but an error is raised if that statement attempts an update operation at runtime; for an example, see “Query Transaction Mode” on page 61.

MarkLogic Server determines transaction type in the following ways:

- Auto: (XQuery only) MarkLogic determines the transaction type through static analysis of the first (or only) statement in the transaction. Auto is the default behavior in XQuery.
- Explicit: Your code explicitly specifies the transaction type as update or query through an option, a call to `xdmp:set-transaction-mode` (XQuery) or `xdmp.setTransactionMode` (JavaScript), or by calling `declareUpdate` (JavaScript only).

For more details, see "Controlling Transaction Type in XQuery" on page 42 or "Controlling Transaction Type in JavaScript" on page 45.

Query transactions use a system timestamp to access a consistent snapshot of the database at a particular point in time, rather than using locks. Update transactions use readers/writers locks. See "Query Transactions: Point-in-Time Evaluation" on page 47 and "Update Transactions: Readers/Writers Locks" on page 48.

The following table summarizes the interactions between transaction types, statements, and locking behavior. These interactions apply to both single-statement and multi-staement transactions.

Transaction Type	Statement	Behavior
query	query	Point-in-time view of documents. No locking required.
	update	Runtime error.
update	query	Read locks acquired, as needed.
	update	Readers/writers locks acquired, as needed.

3.4.2 Controlling Transaction Type in XQuery

You do not need to explicitly set transaction type unless the default auto-detection is not suitable for your application. When the transaction type is "auto" (the default), MarkLogic determines the transaction type through static analysis of your code. In a multi-statement transaction, MarkLogic examines only the first statement when auto-detecting transaction type.

To explicitly set the transaction type:

- Declare the `xdmp:update` option in the XQuery prolog, or
- Call `xdmp:set-transaction-mode` prior to creating transactions that should run in that mode, or
- Set the `update` option in the options node passed to functions such as `xdmp:eval`, `xdmp:invoke`, or `xdmp:spawn`.

Use the `xdmp:update` prolog option when you need to set the transaction type before the first transaction is created, such as at the beginning of a main module. For example, the following code runs as a multi-statement update transaction because of the prolog options:

```
declare option xdmp:commit "explicit";
declare option xdmp:update "true";

let $txn-name := "ExampleTransaction-1"
return (
  xdmp:set-transaction-name($txn-name),
  fn:concat($txn-name, ": ",
    xdmp:host-status(xdmp:host())
      //hs:transaction[hs:transaction-name eq $txn-name]
        /hs:transaction-mode)
  );
  xdmp:commit();
```

For more details, see xdmp:update and xdmp:commit in the *XQuery and XSLT Reference Guide*.

Setting transaction mode with `xdmp:set-transaction-mode` affects both the commit semantics (auto or explicit) and the transaction type (auto, query, or update). Setting the transaction mode in the middle of a transaction does not affect the current transaction. Setting the transaction mode affects the transaction creation semantics for the entire session.

The following example uses `xdmp:set-transaction-mode` to demonstrate that the currently running transaction is unaffected by setting the transaction mode to a different value. The example uses `xdmp:host-status` to examine the mode of the current transaction. (The example only uses `xdmp:set-transaction-name` to easily pick out the relevant transaction in the `xdmp:host-status` results.)

```
xquery version "1.0-ml";

declare namespace hs="http://marklogic.com/xdmp/status/host";

(: The first transaction created will run in update mode :)
declare option xdmp:commit "explicit";
declare option xdmp:update "true";

let $txn-name := "ExampleTransaction-1"
return (
  xdmp:set-transaction-name($txn-name),
  xdmp:set-transaction-mode("query"),  (: no effect on current txn :)
  fn:concat($txn-name, ": ",
    xdmp:host-status(xdmp:host())
      //hs:transaction[hs:transaction-name eq $txn-name]
        /hs:transaction-mode)
  );

(: complete the current transaction :)
xdmp:commit();
```

```
(: a new transaction is created, inheriting query mode from above :)
declare namespace hs="http://marklogic.com/xdmp/status/host";
let $txn-name := "ExampleTransaction-2"
return (
  xdmp:set-transaction-name($txn-name),
    fn:concat($txn-name, ": ",
      xdmp:host-status(xdmp:host())
        //hs:transaction[hs:transaction-name eq $txn-name]
          /hs:transaction-mode)
  );
```

If you paste the above example into Query Console, and run it with results displayed as text, you see the first transaction runs in update mode, as specified by xdmp:transaction-mode, and the second transaction runs in query mode, as specified by `xdmp:set-transaction-mode`:

```
ExampleTransaction-1: update
ExampleTransaction-2: query
```

You can include multiple option declarations and calls to `xdmp:set-transaction-mode` in your program, but the settings are only considered at transaction creation. A transaction is implicitly created just before evaluating the first statement. For example:

```
xquery version "1.0-ml";
declare option xdmp:commit "explicit";
declare option xdmp:update "true";

(: begin transaction :)
"this is an update transaction";
xdmp:commit();
(: end transaction :)

xquery version "1.0-ml";
declare option xdmp:commit "explicit";
declare option xdmp:update "false";

(: begin transaction :)
"this is a query transaction";
xdmp:commit();
(: end transaction :)
```

The following functions support `commit` and `update` options that enable you to control the commit mode (explicit or auto) and transaction type (update, query, or auto). For details, see the function reference for `xdmp:eval` or `xdmp.eval`.

XQuery	JavaScript
`xdmp:eval`	`xdmp.eval`
`xdmp:javascript-eval`	`xdmp.xqueryEval`
`xdmp:invoke`	`xdmp.invoke`
`xdmp:invoke-function`	`xdmp.invokeFunction`
`xdmp:spawn`	`xdmp.spawn`
`xdmp:spawn-function`	

3.4.3 Controlling Transaction Type in JavaScript

By default, Server-Side JavaScripts runs in a single-statement, auto-commit, query transaction. You can control transaction type in the following ways:

- Use the `declareUpdate` function to set the transaction type to update and/or specify the commit semantics, or
- Set the `update` option in the options node passed to functions such as `xdmp.eval`, `xdmp.invoke`, or `xdmp.spawn`; or
- Call `xdmp.setTransactionMode` prior to creating transactions that should run in that mode.

3.4.3.1 Configuring a Transaction Using declareUpdate

By default, JavaScript runs in `auto` commit mode with `query` transaction type. You can use the `declareUpdate` function to change the transaction type to `update` and/or the commit mode from `auto` to `explicit`.

MarkLogic cannot use static analysis to determine whether or not JavaScript code performs updates. If your JavaScript code makes updates, one of the following requirements must be met:

- You call the `declareUpdate` function to indicate your code will make updates.
- The caller of your code sets the transaction type to one that permits updates.

Calling `declareUpdate` with no arguments is equivalent to auto commit mode and update transaction type. This means the code can make updates and runs as a single-statement transaction. The updates are automatically commited when the JavaScript code completes.

You can also pass an `explicitCommit` option to `declareUpdate`, as shown below. The default value of `explicitCommit` is false.

```
declareUpdate({explicitCommit: boolean});
```

If you set `explicitCommit` to `true`, then your code starts a new multi-statement update transaction. You must explicitly commit or rollback the transaction, either before returning from your JavaScript code or in another context, such as the caller of your JavaScript code or another request executing in the same transaction.

For example, you might use `explicitCommit` to start a multi-statement transaction in an ad-hoc query request through XCC, and then subsequently commit the transaction through another request.

If the caller sets the transaction type to update, then your code is not required to call `declareUpdate` in order to perform updates. If you do call `declareUpdate` in this situation, then the resulting mode must not conflict with the mode set by the caller.

For more details, see declareUpdate Function in the *JavaScript Reference Guide*.

3.4.3.2 Configuring Transactions in the Caller

The following are examples of cases in which the transaction type and commit mode might be set before your code is called:

- Your code is called via an eval/invoke function such as the XQuery function `xdmp:javascript-eval` or the JavaScript functions `xdmp.eval`, and the caller specifies the `commit`, `update`, or `transaction-mode` option.
- Your code is a server-side import transformation for use with the mlcp command line tool.
- Your code is a server-side transformation, extension, or other customization called by the Java, Node.js, or REST Client APIs. The pre-set mode depends on the operation which causes your code to run.
- Your code runs in the context of an XCC session where the client sets the commit mode and/or transaction type.

The following functions support `commit` and `update` options that enable you to control the commit mode (explicit or auto) and transaction type (update, query, or auto). For details, see the function reference for `xdmp:eval` (XQuery) or `xdmp.eval` (JavaScript).

XQuery	JavaScript
`xdmp:eval`	`xdmp.eval`
`xdmp:javascript-eval`	`xdmp.xqueryEval`
`xdmp:invoke`	`xdmp.invoke`
`xdmp:invoke-function`	`xdmp.invokeFunction`
`xdmp:spawn`	`xdmp.spawn`
`xdmp:spawn-function`	

3.4.4 Query Transactions: Point-in-Time Evaluation

Query transactions are read-only and never obtain locks on documents. This section explores the following concepts related to query transactions:

- System Timestamps and Fragment Versioning
- Query Transactions Run at a Timestamp (No Locks)
- Query Transactions See Latest Version of Documents Up To Timestamp of Transaction

3.4.4.1 System Timestamps and Fragment Versioning

To understand how transactions work in MarkLogic Server, it is important to understand how documents are stored. Documents are made up of one or more fragments. After a document is created, each of its fragments are stored in one or more stands. The stands are part of a forest, and the forest is part of a database. A database contains one or more forests.

Each fragment in a stand has system timestamps associated with it, which correspond to the range of system timestamps in which that version of the fragment is valid. When a document is updated, the update process creates new versions of any fragments that are changed. The new versions of the fragments are stored in a new stand and have a new set of valid system timestamps associated with them. Eventually, the system merges the old and new stands together and creates a new stand with only the latest versions of the fragments. Point-in-time queries also affect which versions of fragments are stored and preserved during a merge. After the merge, the old stands are deleted.

The range of valid system timestamps associated with fragments are used when a statement determines which version of a document to use during a transaction. For more details about merges, see Understanding and Controlling Database Merges in the *Administrator's Guide*. For more details on how point-in-time queries affect which versions of documents are stored, see "Point-In-Time Queries" on page 145.

3.4.4.2 Query Transactions Run at a Timestamp (No Locks)

Query transactions run at the system timestamp corresponding to transaction creation time. Calls to `xdmp:request-timestamp` return the same system timestamp at any point during a query transaction; they never return the empty sequence. Query transactions do not obtain locks on any documents, so other transactions can read or update the document while the transaction is executing.

3.4.4.3 Query Transactions See Latest Version of Documents Up To Timestamp of Transaction

When a query transaction is created, MarkLogic Server gets the current system timestamp (the number returned when calling the `xdmp:request-timestamp` function) and uses only the latest versions of documents whose timestamp is less than or equal to that number. Even if any of the documents that the transaction accesses are updated or deleted outside the transaction while the transaction is open, the use of timestamps ensures that all statements in the transaction always see a consistent view of the documents the transaction accesses.

3.4.5 Update Transactions: Readers/Writers Locks

Update transactions have the potential to change the database, so they obtain locks on documents to ensure transactional integrity. Update transactions run with readers/writers locks, not at a timestamp like query transactions. This section covers the following topics:

- Identifying Update Transactions
- Locks Are Acquired on Demand and Held Throughout a Transaction
- Visibility of Updates

3.4.5.1 Identifying Update Transactions

When MarkLogic creates a transaction in auto-detect mode, the transaction type is determined through static analysis of the first (or only) statement in the transaction. If MarkLogic detects the potential for updates during static analysis, then the transaction is considered an update transaction.

Depending on the specific logic of the transaction, it might not actually update anything, but a transaction that MarkLogic determines to be an update transaction always runs as an update transaction, not a query transaction.

For example, the following transaction runs as an update transaction even though the `xdmp:document-insert` can never occur:

```
if ( 1 = 2 )
then ( xdmp:document-insert("fake.xml", <a/>) )
else ()
```

In a multi-statement transaction, the transaction type always corresponds to the transaction type settings in effect when the transaction is created. If the transaction type is explicitly set to `update`, then the transaction is an update transaction, even if none of the contained statements perform updates. Locks are acquired for all statements in an update transaction, whether or not they perform updates.

Similarly, if you use auto-detect mode and MarkLogic determines the first statement in a multi-statement transaction is a query statement, then the transaction is created as a query transaction. If a subsequent statement in the transaction attempts an update operation, MarkLogic throws an exception.

Calls to `xdmp:request-timestamp` always return the empty sequence during an update transaction; that is, if `xdmp:request-timestamp` returns a value, the transaction is a query transaction, not an update transaction.

3.4.5.2 Locks Are Acquired on Demand and Held Throughout a Transaction

Because update transactions do not run at a set timestamp, they see the latest view of any given document at the time it is first accessed by any statement in the transaction. Because an update transaction must successfully obtain locks on all documents it reads or writes in order to complete evaluation, there is no chance that a given update transaction will see “half” or “some” of the updates made by some other transactions; the statement is indeed transactional.

Once a lock is acquired, it is held until the transaction ends. This prevents other transactions from updating the read locked document and ensures a read-consistent view of the document. Query (read) operations require read locks. Update operations require readers/writers locks.

When a statement in an update transaction wants to perform an update operation, a readers/writers lock is acquired (or an existing read lock is converted into a readers/writers lock) on the document. A readers/writers lock is an exclusive lock. The readers/writers lock cannot be acquired until any locks held by other transactions are released.

Lock lifetime is an especially important consideration in multi-statement transactions. Consider the following single-statement example, in which a readers/writers lock is acquired only around the call to `xdmp:node-replace`:

```
  (: query statement, no locks needed :)
fn:doc("/docs/test.xml");
  (: update statement, readers/writers lock acquired :)
xdmp:node-replace(fn:doc("/docs/test.xml")/node(), <a>hello</a>);
```

```
  (: readers/writers lock released :)
  (: query statement, no locks needed :)
fn:doc("/docs/test.xml");
```

If the same example is rewritten as a multi-statement transaction, locks are held across all three statements:

```
declare option xdmp:transaction-mode "update";

  (: read lock acquired :)
fn:doc("/docs/test.xml");
  (: the following converts the lock to a readers/writers lock :)
xdmp:node-replace(fn:doc("/docs/test.xml")/node(), <a>hello</a>);
  (: readers/writers lock still held :)
fn:doc("/docs/test.xml");

  (: after the following statement, txn ends and locks released :)
xdmp:commit()
```

3.4.5.3 Visibility of Updates

Updates are only visible within a transaction after the updating statement completes; updates are not visible within the updating statement. Updates are only visible to other transactions after the updating transaction commits. Pre-commit triggers run as part of the updating transaction, so they see updates prior to commit. Transaction model affects the visibility of updates, indirectly, because it affects when commit occurs.

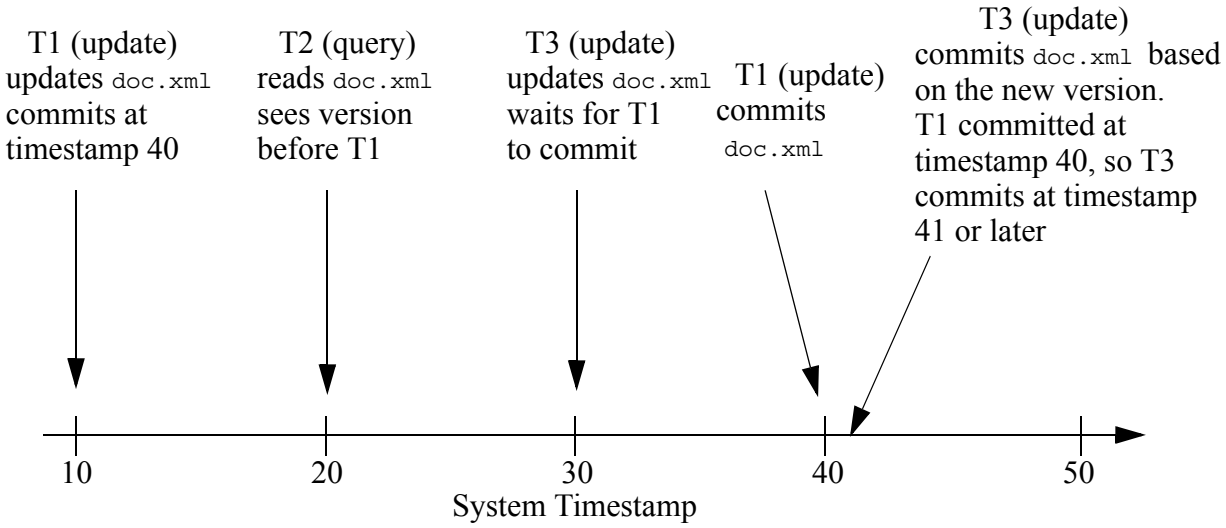
In the default single-statement transaction model, the commit occurs automatically when the statement completes. To use a newly updated document, you must separate the update and the access into two single-statement transactions or use multi-statement transactions.

In a multi-statement transaction, changes made by one statement in the transaction are visible to subsequent statements in the same transaction as soon as the updating statement completes. Changes are not visible outside the transaction until you call `xdmp:commit`.

An update statement cannot perform an update to a document that will conflict with other updates occurring in the same statement. For example, you cannot update a node and add a child element to that node in the same statement. An attempt to perform such conflicting updates to the same document in a single statement will fail with an `XDMP-CONFLICTINGUPDATES` exception.

3.4.6 Example: Query and Update Transaction Interaction

The following figure shows three different transactions, T1, T2, and T3, and how the transactional semantics work for each one:



Assume T1 is a long-running update transaction which starts when the system is at timestamp 10 and ends up committing at timestamp 40 (meaning there were 30 updates or other changes to the system while this update statement runs).

When T2 reads the document being updated by T1 (`doc.xml`), it sees the latest version that has a system timestamp of 20 or less, which turns out to be the same version T1 uses before its update.

When T3 tries to update the document, it finds that T1 has readers/writers locks on it, so it waits for them to be released. After T1 commits and releases the locks, then T3 sees the newly updated version of the document, and performs its update which is committed at a new timestamp of 41.

3.5 Single vs. Multi-statement Transactions

This section discusses the details of and differences between the two transaction programming models supported by MarkLogic Server, single-statement and multi-statement transactions. The following topics are covered:

- Single-Statement, Automatically Committed Transactions
- Multi-Statement, Explicitly Committed Transactions
- Semi-Colon as a Statement Separator

3.5.1 Single-Statement, Automatically Committed Transactions

By default, all transactions in MarkLogic Server are single-statement, auto-commit transactions. In this default model, MarkLogic creates a transaction to evaluate each statement. When the statement completes, MarkLogic automatically commits (or rolls back, in case of error) the transaction, and then the transaction ends.

In Server-Side JavaScript, a JavaScript program (or "script") is considered a single "statement" in the transactional sense. For details, see "Understanding Statement Boundaries" on page 36.

In a single statement transaction, updates made by a statement are not visible outside the statement until the statement completes and the transaction is committed.

The single-statement model is suitable for most applications. This model requires less familiarity with transaction details and introduces less complexity into your application:

- Statement and transaction are nearly synonymous.
- The server determines the transaction type through static analysis.
- If the statement completes successfully, the server automatically commits the transaction.
- If an error occurs, the server automatically rolls back any updates made by the statement.

Updates made by a single-statement transaction are not visible outside the statement until the statement completes. For details, see "Visibility of Updates" on page 50.

Use the semi-colon separator extension in XQuery to include multiple single-statement transactions in your program. For details, see "Semi-Colon as a Statement Separator" on page 57.

> **Note:** In Server-Side JavaScript, you need to use the `declareUpdate()` function to run an update. For details, see "Controlling Transaction Type in JavaScript" on page 45.

3.5.2 Multi-Statement, Explicitly Committed Transactions

When a transaction is created in a context in which the commit mode is set to "explicit", the transaction will be a multi-statement transaction. This section covers the following related topics:

- Characteristics of Multi-Statement Transactions
- Committing Multi-Statement Transactions
- Rolling Back Multi-Statement Transactions
- Sessions

For details on setting the transaction type and commit mode, see "Transaction Type" on page 41.

For additional information about using multi-statement transactions in Java, see "Multi-Statement Transactions" in the *XCC Developer's Guide*.

3.5.2.1 Characteristics of Multi-Statement Transactions

Using multi-statement transactions introduces more complexity into your application and requires a deeper understanding of transaction handling in MarkLogic Server. In a multi-statement transaction:

- In XQuery, semi-colon acts as a separator between statements in the same transaction.
- In Server-Side JavaScript, the entire program (script) is considered a single transactional statement, regardless how many JavaScript statements it contains.
- Each statement in the transaction sees changes made by previously evaluated statements in the same transaction.
- The statements in the transaction either all commit or all fail.
- You must use `xdmp:commit` (XQuery) or `xdmp.commit` (JavaScript) to commit the transaction.
- You can use `xdmp:rollback` (XQuery) or `xdmp.rollback` (JavaScript) to abort the transaction.

A multi-statement transaction is bound to the database in which it is created. You cannot use a transaction id created in one database context to perform an operation in the same transaction on another database.

The statements in a multi-statement transaction are serialized, even if they run in different requests. That is, one statement in the transaction completes before another one starts, even if the statements execute in different requests.

A multi-statement transaction ends only when it is explicitly committed using `xdmp:commit` or `xdmp.commit`, when it is explicitly rolled back using `xdmp:rollback` or `xdmp.rollback`, or when it is implictly rolled back through timeout, error, or session completion. Failure to explicitly commit or roll back a multi-statement transaction might retain locks and keep resources tied up until the transaction times out or the containing session ends. At that time, the transaction rolls back. Best practice is to always explicitly commit or rollback a multi-statement transaction.

The following example contains 3 multi-statement transactions (because of the use of the `commit` prolog option). The first transaction is explicitly committed, the second is explicitly rolled back, and the third is implicitly rolled back when the session ends without a commit or rollback call. Running the example in Query Console is equivalent to evaluating it using `xdmp:eval` with `different transaction` isolation, so the final transaction rolls back when the end of the query is reached because the session ends. For details about multi-statement transaction interaction with sessions, see “Sessions” on page 56.

```
xquery version "1.0-ml";

declare option xdmp:commit "explicit";
(: Begin transaction 1 :)
xdmp:document-insert('/docs/mst1.xml', <data/>);
(: This statement runs in the same txn, so sees /docs/mst1.xml :)
```

```
xdmp:document-insert('/docs/mst2.xml', fn:doc('/docs/mst1.xml'));
xdmp:commit();
(: Transaction ends, updates visible in database :)

declare option xdmp:commit "explicit";
(: Begin transaction 2 :)
xdmp:document-delete('/docs/mst1.xml');
xdmp:rollback();
(: Transaction ends, updates discarded :)

declare option xdmp:commit "explicit";
(: Begin transaction 3 :)
xdmp:document-delete('/docs/mst1.xml');
(: Transaction implicitly ends and rolls back due to
 :   reaching end of program without a commit :)
```

As discussed in “Update Transactions: Readers/Writers Locks” on page 48, multi-statement update transactions use locks. A multi-statement update transaction can contain both query and update operations. Query operations in a multi-statement update transaction acquire read locks as needed. Update operations in the transaction will upgrade such locks to read/write locks or acquire new read/write locks if needed.

Instead of acquiring locks, a multi-statement query transaction uses a system timestamp to give all statements in the transaction a read consistent view of the database, as discussed in “Query Transactions: Point-in-Time Evaluation” on page 47. The system timestamp is determined when the query transaction is created, so all statements in the transaction see the same version of accessed documents.

3.5.2.2 Committing Multi-Statement Transactions

Multi-statement transactions are explicitly committed by calling `xdmp:commit`. If a multi-statement update transaction does not call `xdmp:commit`, all its updates are lost when the transaction ends. Leaving a transaction open by not committing updates ties up locks and other resources.

Once updates are committed, the transaction ends and evaluation of the next statement continues in a new transaction. For example:

```
xquery version "1.0-ml";

declare option xdmp:commit "explicit";

(: Begin transaction 1 :)
xdmp:document-insert('/docs/mst1.xml', <data/>);
(: This statement runs in the same txn, so sees /docs/mst1.xml :)
xdmp:document-insert('/docs/mst2.xml', fn:doc('/docs/mst1.xml'));
xdmp:commit();
(: Transaction ends, updates visible in database :)
```

Calling `xdmp:commit` commits updates and ends the transaction only after the calling statement successfully completes. This means updates can be lost even after calling `xdmp:commit`, if an error occurs before the committing statement completes. For this reason, it is best practice to call `xdmp:commit` at the end of a statement.

The following example preserves updates even in the face of error because the statement calling `xdmp:commit` always completes.:

```
xquery version "1.0-ml";

declare option xdmp:commit "explicit";

(: transaction created :)
xdmp:document-insert("not-lost.xml", <data/>)
, xdmp:commit();
fn:error(xs:QName("EXAMPLE-ERROR"), "An error occurs here");
(: end of session or program :)

(: ==> Insert is retained because the statement
       calling commit completes sucessfully. :)
```

By contrast, the update in this example is lost because the error occurring in the same statement as the `xdmp:commit` call prevents successful completion of the committing statement:

```
xquery version "1.0-ml";

declare option xdmp:commit "explicit";

(: transaction created :)
xdmp:document-insert("lost.xml", <data/>)
, xdmp:commit()
, fn:error(xs:QName("EXAMPLE-ERROR"), "An error occurs here");
(: end of session or program :)

(: ==> Insert is lost because the statement
       terminates with an error before commit can occur. :)
```

Uncaught exceptions cause a transaction rollback. If code in a multi-statement transaction might raise an exception that should not abort the transaction, wrap the code in a try-catch block and take appropriate action in the catch handler. For example:

```
xquery version "1.0-ml";

declare option xdmp:commit "explicit";

xdmp:document-insert("/docs/test.xml", <a>hello</a>);
try {
  xdmp:document-delete("/docs/nonexistent.xml")
} catch ($ex) {
  (: handle error or rethrow :)
  if ($ex/error:code eq 'XDMP-DOCNOTFOUND') then ()
```

```
  else xdmp:rethrow()
}, xdmp:commit();
(: start of a new txn :)
fn:doc("/docs/test.xml")//a/text()
```

3.5.2.3 Rolling Back Multi-Statement Transactions

Multi-statement transactions are rolled back either implicitly (on error or when the containing session terminates), or explicitly (using `xdmp:rollback` or `xdmp.rollback)`. Calling `xdmp:rollback` immediately terminates the current transaction. Evaluation of the next statement continues in a new transaction. For example:

```
xquery version "1.0-ml";

declare option xdmp:commit "explicit";
                                              (: begin transaction :)
xdmp:document-insert("/docs/mst.xml", <data/>);
xdmp:commit()
, "this expr is evaluated and committed";
                                              (: end transaction :)
                                              (:begin transaction :)
declare option xdmp:commit "explicit";
xdmp:document-insert("/docs/mst.xml", <data/>);
xdmp:rollback()                               (: end transaction :)
, "this expr is never evaluated";
                                              (:begin transaction :)
"execution continues here, in a new transaction"
                                              (: end transaction :)
```

The result of a statement terminated with `xdmp:rollback` is always the empty sequence.

Best practice is to explicitly rollback when necessary. Waiting on implicit rollback at session end leaves the transaction open and ties up locks and other resources until the session times out. This can be a relatively long time. For example, an HTTP session can span multiple HTTP requests. For details, see “Sessions” on page 56.

3.5.2.4 Sessions

A session is a “conversation” with a database in a MarkLogic Server instance. A session encapsulates state about the conversation, such as connection information, credentials, and transaction settings. When using multi-statement transactions, you should understand when evaluation might occur in a different session because:

- transaction mode is an attribute of a session.
- uncommitted transactions automatically roll back when the containing session ends.

For example, since a query evaluated by `xdmp:eval` (XQuery) or `xdmp.eval` (JavaScript) with `different-transaction` isolation runs in its own session, it does not inherit the transaction mode setting from the caller. Also, if the transaction is still open (uncommitted) when evaluation reaches the end of the eval’d query, the transaction automatically rolls back.

By contrast, in an HTTP session, the transaction settings might apply to queries run in response to multiple HTTP requests. Uncommitted transactions remain open until the HTTP session times out, which can be a relatively long time.

The exact nature of a session depends on the "conversation" context. The following table summarizes the most common types of sessions encountered by a MarkLogic Server application and their lifetimes:

Session Type	Session Lifetime
HTTP An HTTP client talking to an HTTP App Server.	A session is created when the first HTTP request is received from a client for which no session already exists. The session persists across requests until the session times out.
XCC An XCC Java application talking to an XDBC App Server	A session is created when a Session object is instantiated and persists until the Session object is finalized, you call Session.close(), or the session times out.
Standalone query evaluated: • by `xdmp:eval` or `xdmp:invoke` with `different-transaction` isolation • by `xdmp:spawn` • as task on the Task Server)	A session is created to evaluate the eval/invoke/spawn'd query or task and ends when the query or task completes.

Session timeout is an App Server configuration setting. For details, see `admin:appserver-set-session-timeout` in *XQuery and XSLT Reference Guide* or the Session Timeout configuration setting in the Admin Interface for the App Server.

3.5.3 Semi-Colon as a Statement Separator

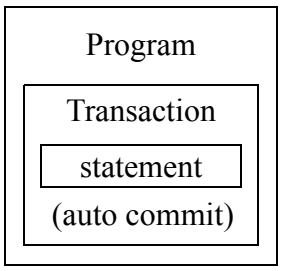
MarkLogic Server extends the XQuery language to include the semi-colon (;) in the XQuery body as a separator between statements. Statements are evaluated in the order in which they appear. Each semi-colon separated statement in a transaction is fully evaluated before the next statement begins.

In a single-statement transaction, the statement separator is also a transaction separator. Each statement separated by a semi-colon is evaluated as its own transaction. It is possible to have a program where some semi-colon separated parts are evaluated as query statements and some are evaluated as update statements. The statements are evaluated in the order in which they appear, and in the case of update statements, one statement commits before the next one begins.

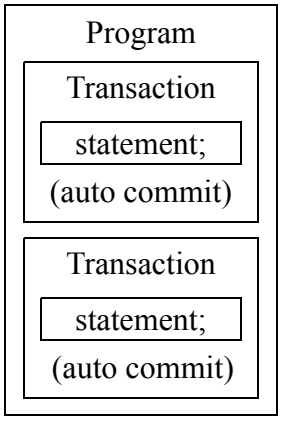
Semi-colon separated statements in `auto` commit mode (the default) are not multi-statement transactions. Each statement is a single-statement transaction. If one update statement commits and the next one throws a runtime error, the first transaction is not rolled back. If you have logic that requires a rollback if subsequent transactions fail, you must add that logic to your XQuery code, use multi-statement transactions, or use a pre-commit trigger. For information about triggers, see "Using Triggers to Spawn Actions" on page 393.

In a multi-statement transaction, the semi-colon separator does not act as a transaction separator. The semi-colon separated statements in a multi-statement transaction see updates made by previous statements in the same transaction, but the updates are not committed until the transaction is explicitly committed. If the transaction is rolled back, updates made by previously evaluated statements in the transaction are discarded.

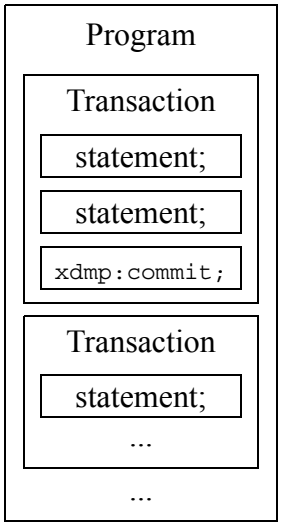
The following diagram contrasts the relationship between statements and transactions in single and multi-statement transactions:



Default model: A program containing one single statement, auto commit transaction.



Default model: A program containing multiple single statement transactions.



Multi-statement transactions: A program containing multiple, multi-statement transactions.

3.6 Transaction Mode

This section covers the following topics related to transaction mode:

- Transaction Mode Overview
- Auto Transaction Mode
- Query Transaction Mode
- Update Transaction Mode
- Query-Single-Statement Transaction Mode
- Multi-Auto Transaction Mode

3.6.1 Transaction Mode Overview

Transaction mode combines the concepts of commit mode (auto or explicit) and transaction type (auto, update, or query). The transaction mode setting is session wide. You can control transaction mode in the following ways:

- Call the `xdmp:set-transaction-mode` XQuery function or the `xdmp.setTransactionMode` JavaScript function.
- Deprecated: Use the `transaction-mode` option of `xdmp:eval` (XQuery) or `xdmp.eval` (JavaScript) or related functions eval/invoke/spawn functions. You should use the `commit` and update `options` instead.
- Deprecated: Use the XQuery prolog option `xdmp:transaction-mode`. You should use the xdmp:commit and xdmp:update XQuery prolog options instead.

You should generally use the more specific commit mode and transaction type controls instead of setting transaction mode. These controls provide finer grained control over transaction configuration.

For example, use the following table to map the `xdmp:transaction-mode` XQuery prolog options to the `xdmp:commit` and `xdmp:update` prolog options. For more details, see “Controlling Transaction Type in XQuery” on page 42.

`xdmp:transaction-mode` Value	Equivlaent `xdmp:commit` and `xdmp:update` Option Settings
`"auto"`	`declare option xdmp:commit "auto";` `declare option xdmp:update "auto";`
`"update-auto-commit"`	`declare option xdmp:commit "auto";` `declare option xdmp:update "true";`
`"query-single-statement"`	`declare option xdmp:commit "auto";` `declare option xdmp:update "false";`

xdmp:transaction-mode Value	Equivlaent xdmp:commit and xdmp:update Option Settings
"multi-auto"	declare option xdmp:commit "explicit"; declare option xdmp:update "auto";
"update"	declare option xdmp:commit "explicit"; declare option xdmp:update "true";
"query"	declare option xdmp:commit "explicit"; declare option xdmp:update "false";

Be aware that the xdmp:commit and xdmp:update XQuery prolog options affect only the next transaction created after their declaration; they do not affect an entire session. Use xdmp:set-transaction-mode or xdmp.setTransactionMode if you need to change the settings at the session level.

Use the following table to map between the transaction-mode option and the commit and update options for xdmp:eval and related eval/invoke/spawn functions.

transaction-mode Option Value	Equivalent commit and update Option Values
auto	commit: "auto" update: "auto"
update-auto-commit	commit: "auto" update: "true"
query-single-statement	commit: "auto" update: "false"
multi-auto	commit: "explicit" update: "auto"
update	commit: "explicit" update: "true"
query	commit "explicit" update "false"

Server-Side JavaScript modules use the declareUpdate function to indicate when the transaction mode is update-auto-commit or update. For more details, see “Controlling Transaction Type in JavaScript” on page 45.

To use multi-statement transactions in XQuery, you must explicitly set the transaction mode to multi-auto, query, or update. This sets the commit mode to “explicit” and specifies the transaction type. For details, see “Transaction Type” on page 41.

Selecting the appropriate transaction mode enables the server to properly optimize your queries. For more information, see "Multi-Statement, Explicitly Committed Transactions" on page 52.

The transaction mode is only considered during transaction creation. Changing the mode has no effect on the current transaction.

Explictly setting the transaction mode affects only the current session. Queries run under `xdmp:eval` or `xdmp.eval` or a similar function with `different-transaction` isolation, or under `xdmp:spawn` do not inherit the transaction mode from the calling context. See "Interactions with xdmp:eval/invoke" on page 64.

3.6.2 Auto Transaction Mode

The default transaction mode is `auto`. This is equivalent to "auto" commit mode and "auto" transaction type. In this mode, all transactions are single-statement transactions. See "Single-Statement, Automatically Committed Transactions" on page 52.

Most XQuery applications should use `auto` transaction mode. Using `auto` transaction mode allows the server to optimize each statement independently and minimizes locking on your files. This leads to better performance and decreases the chances of deadlock, in most cases.

Most Server-Side JavaScript applications should use `auto` mode for code that does not perform updates, and `update-auto-commit` mode for code that performs updates. Calling `declareUpdate` with no arguments activates `update-auto-commit` mode; for more details, see "Controlling Transaction Type in JavaScript" on page 45..

In `auto` transaction mode:

- All transactions are single-statement transactions, so a new transaction is created for each statement.
- Static analysis of the statement prior to evaluation determines whether the created transaction runs in update or query mode.
- The transaction associated with a statement is automatically committed when statement execution completes, or automatically rolled back if an error occurs.

The `update-auto-commit` differs only in that the transaction is always an update transaction.

In XQuery, you can set the mode to `auto` explicitly with `xdmp:set-transaction-mode` or the xdmp:transaction-mode prolog option, but this is not required unless you've previously explicitly set the mode to `update` or `query`.

3.6.3 Query Transaction Mode

Query transaction mode is equivalent to `explicit` commit mode plus `query` transaction type.

In XQuery, query transaction mode is only in effect when you explicitly set the mode using `xdmp:set-transaction-mode` or the xdmp:transaction-mode prolog option. Transactions created in this mode are always multi-statement transactions, as described in "Multi-Statement, Explicitly Committed Transactions" on page 52.

You cannot create a multi-statement query transaction from Server-Side JavaScript.

In query transaction mode:

- Transactions can span multiple statements.
- The transaction is assumed to be read-only, so no locks are acquired. MarkLogic executes all statements in the transaction as point-in-time queries, using the system timestamp at the start of the transaction.
- All statements in the transaction should functionally be query (read-only) statements. MarkLogic raises an error at runtime if an update operation is attempted.
- Transactions which are not explicitly committed using `xdmp:commit` roll back when the session times out. However, since there are no updates to commit, rollback is only distinguishable if an explicit rollback occurs before statement completion.

An update statement can appear in a multi-statement query transaction, but it must not actually make any update calls at runtime. If a transaction running in query mode attempts an update operation, `XDMP-UPDATEFUNCTIONFROMQUERY` is raised. For example, no exception is raised by the following code because the program logic causes the update operation not to run:

```
xquery version "1.0-ml";
declare option xdmp:transaction-mode "query";

if (fn:false())then
  (: XDMP-UPDATEFUNCTIONFROMQUERY only if this executes :)
  xdmp:document-insert("/docs/test.xml", <a/>)
else ();
xdmp:commit();
```

3.6.4 Update Transaction Mode

Update transaction mode is equivalent to explicit commit mode plus update transaction type.

In XQuery, update transaction mode is only in effect when you explicitly set the mode using `xdmp:set-transaction-mode` or the xdmp:transaction-mode prolog option. Transactions created in `update` mode are always multi-statement transactions, as described in "Multi-Statement, Explicitly Committed Transactions" on page 52.

In Server-Side JavaScript, setting `explicitCommit` to true when calling `declareUpdate` puts the transaction into update mode.

In update transaction mode:

- Transactions can span multiple statements.
- The transaction is assumed to change the database, so readers/writers locks are acquired as needed.
- Statements in an update transaction can be either update or query statements.
- Transactions which are not explicitly committed using `xdmp:commit` roll back when the session times out.

Update transactions can contain both query and update statements, but query statements in update transactions still acquire read locks rather than using a system timestamp. For more information, see "Update Transactions: Readers/Writers Locks" on page 48.

3.6.5 Query-Single-Statement Transaction Mode

The `query-single-statement` transaction mode is equivalent to auto commit mode plus query transaction type.

In XQuery, this transaction mode is only in effect when you explicitly set the mode using `xdmp:set-transaction-mode` or the xdmp:transaction-mode prolog option. Transactions created in this mode are always single-statement transactions, as described in "Single-Statement Transaction Concept Summary" on page 38.

You cannot explicitly create a single-statement query-only transaction from Server-Side JavaScript, but this is the default transaction mode when `declareUpdate` is not present.

In this transaction mode:

- All transactions are single-statement transactions, so a new transaction is created for each statement.
- The transaction is assumed to be read-only, so no locks are acquired. The statement is evaluated as a point-in-time query, using the system timestamp at the start of the transaction.
- An error is raised at runtime if an update operation is attempted by the transaction.
- The transaction is automatically committed when statement execution completes, or automatically rolled back if an error occurs.

An update operation can appear in a this type of transaction, but it must not actually make any updates at runtime. If a transaction running in query mode attempts an update operation, `XDMP-UPDATEFUNCTIONFROMQUERY` is raised.

3.6.6 Multi-Auto Transaction Mode

Setting transaction mode to `multi-auto` is equivalent to explicit commit mode plus auto transaction type. In this mode, all transactions are multi-statement transactions, and MarkLogic determines for you whether the transaction type is query or update.

In `multi-auto` transaction mode:

- Transactions can span multiple statements.
- Static analysis of the first statement in the transaction determines whether the transaction type is query or update.
- Transactions which are not explicitly committed using `xdmp:commit` or `xdmp.commit` roll back when the session times out.

In XQuery, `multi-auto` transaction mode is only in effect when you explicitly set the mode using `xdmp:set-transaction-mode` or the xdmp:transaction-mode prolog option.

There is no equivalent to `multi-auto` transaction mode for Server-Side JavaScript.

3.7 Interactions with xdmp:eval/invoke

The `xdmp:eval` and `xdmp:invoke` family of functions enable you to start one transaction from the context of another. The `xdmp:eval` XQuery function and the `xdmp.eval` JavaScript function submit a string to be evaluated. The `xdmp:invoke` XQuery function and the `xdmp.invoke` JavaScript function evaluate a stored module. You can control the semantics of eval and invoke with options to the functions, and this can subtly change the transactional semantics of your program. This section describes some of those subtleties and includes the following parts:

- Isolation Option to xdmp:eval/invoke
- Preventing Deadlocks
- Seeing Updates From eval/invoke Later in the Transaction
- Running Multi-Statement Transactions under xdmp:eval/invoke

3.7.1 Isolation Option to xdmp:eval/invoke

The `xdmp:eval` and `xdmp:invoke` XQuery functions and their JavaScript counterparts accept a set of options as an optional third parameter. The `isolation` option determines the behavior of the transaction that results from the eval/invoke operation, and it must be one of the following values:

- `same-statement`
- `different-transaction`

In `same-statement` isolation, the code executed by eval or invoke runs as part of the same statement and in the same transaction as the calling statement. Any updates done in the eval/invoke operation with `same-statement` isolation are not visible to subsequent parts of the calling statement. However, when using multi-statement transactions, those updates are visible to subsequent statements in the same transaction.

You may not perform update operations in code run under eval/invoke in `same-statement` isolation called from a query transaction. Since query transactions run at a timestamp, performing an update would require a switch between timestamp mode and readers/writers locks in the middle of a transaction, and that is not allowed. Statements or transactions that do so will throw `XDMP-UPDATEFUNCTIONFROMQUERY`.

You may not use `same-statement` isolation when using the `database` option of eval or invoke to specify a different database than the database in the calling statement's context. If your eval/invoke code needs to use a different database, use `different-transaction` isolation.

When you set the `isolation` to `different-transaction`, the code that is run by eval/invoke runs in a separate session and a separate transaction from the calling statement. The eval/invoke session and transaction will complete before continuing the rest of the caller's transaction. If the calling transaction is an update transaction, any committed updates done in the eval/invoke operation with `different-transaction` isolation are visible to subsequent parts of the calling statement and to subsequent statements in the calling transaction. However, if you use `different-transaction` isolation (which is the default isolation level), you need to ensure that you do not get into a deadlock situation (see "Preventing Deadlocks" on page 66).

The following table shows which isolation options are allowed from query statements and update statements.

Calling Statement	Called Statement (`xdmp:eval`, `xdmp:invoke`)			
	`same-statement` isolation		`different-transaction` isolation	
	query statement	update statement	query statement	update statement
query statement (timestamp mode)	Yes	Yes, if no update takes place. If an update takes place, throws exception.	Yes	Yes
update statement (readers/writers locks mode)	Yes (see Note)	Yes	Yes	Yes (possible deadlock if updating a document with any lock)

Note: This table is slightly simplified. For example, if an update statement calls a query statement with `same-statement` isolation, the "query statement" is actually run as part of the update statement (because it is run as part of the same transaction as the calling update statement), and it therefore runs with readers/writers locks, not in a timestamp.

3.7.2 Preventing Deadlocks

A deadlock is where two processes or threads are each waiting for the other to release a lock, and neither process can continue until the lock is released. Deadlocks are a normal part of database operations, and when the server detects them, it can deal with them (for example, by retrying one or the other transaction, by killing one or the other or both requests, and so on).

There are, however, some deadlock situations that MarkLogic Server cannot do anything about except wait for the transaction to time out. When you run an update statement that calls an `xdmp:eval` or `xdmp:invoke` statement, and the eval/invoke in turn is an update statement, you run the risk of creating a deadlock condition. These deadlocks can only occur in update statements; query statements will never cause a deadlock.

A deadlock condition occurs when a transaction acquires a lock of any kind on a document and then an eval/invoke statement called from that transaction attempts to get a write lock on the same document. These deadlock conditions can only be resolved by cancelling the query or letting the query time out.

To be completely safe, you can prevent these deadlocks from occurring by setting the `prevent-deadlocks` option to `true`, as in the following example:

```
xquery version "1.0-ml";
(: the next line ensures this runs as an update statement :)
declare option xdmp:update "true";
xdmp:eval("xdmp:node-replace(doc('/docs/test.xml')/a,
<b>goodbye</b>)",
          (),
          <options xmlns="xdmp:eval">
            <isolation>different-transaction</isolation>
            <prevent-deadlocks>true</prevent-deadlocks>
          </options>) ,
doc("/docs/test.xml")
```

This statement will then throw the following exception:

```
XDMP-PREVENTDEADLOCKS: Processing an update from an update with
  different-transaction isolation could deadlock
```

In this case, it will indeed prevent a deadlock from occurring because this statement runs as an update statement, due to the `xdmp:document-insert` call, and therefore uses readers/writers locks. In line 2, a read lock is placed on the document with URI `/docs/test.xml`. Then, the `xdmp:eval` statement attempts to get a write lock on the same document, but it cannot get the write lock until the read lock is released. This creates a deadlock condition. Therefore the `prevent-deadlocks` option stopped the deadlock from occurring.

If you remove the `prevent-deadlocks` option, then it defaults to `false` (that is, it will *allow* deadlocks). Therefore, the following statement results in a deadlock:

Warning This code is for demonstration purposes; if you run this code, it will cause a deadlock and you will have to cancel the query or wait for it to time out to clear the deadlock.

```
(: the next line ensures this runs as an update statement :)
if ( 1 = 2) then ( xdmp:document-insert("foobar", <a/>) ) else (),
doc("/docs/test.xml"),
xdmp:eval("xdmp:node-replace(doc('/docs/test.xml')/a,
<b>goodbye</b>)",
          (),
          <options xmlns="xdmp:eval">
            <isolation>different-transaction</isolation>
          </options>) ,
doc("/docs/test.xml")
```

This is a deadlock condition, and the deadlock will remain until the transaction either times out, is manually cancelled, or MarkLogic is restarted. Note that if you take out the first call to `doc("/docs/test.xml")` in line 2 of the above example, the statement will not deadlock because the read lock on `/docs/test.xml` is not called until after the `xdmp:eval` statement completes.

3.7.3 Seeing Updates From eval/invoke Later in the Transaction

If you are sure that your update statement in an eval/invoke operation does not try to update any documents that are referenced earlier in the calling statement (and therefore does not result in a deadlock condition, as described in "Preventing Deadlocks" on page 66), then you can set up your statement so updates from an eval/invoke are visible from the calling transaction. This is most useful in transactions that have the eval/invoke statement before the code that accesses the newly updated documents.

Note: If you want to see the updates from an eval/invoke operation later in your statement, the transaction must be an update transaction. If the transaction is a query transaction, it runs in timestamp mode and will always see the version of the document that existing before the eval/invoke operation committed.

For example, consider the following example, where `doc("/docs/test.xml")` returns `<a>hello</a>` before the transaction begins:

```
(: doc("/docs/test.xml") returns <a>hello</a> before running this :)
(: the next line ensures this runs as an update statement :)
if ( 1 = 2 ) then ( xdmp:document-insert("fake.xml", <a/>) ) else (),
xdmp:eval("xdmp:node-replace(doc('/docs/test.xml')/node(),
<b>goodbye</b>)", (),
          <options xmlns="xdmp:eval">
            <isolation>different-transaction</isolation>
            <prevent-deadlocks>false</prevent-deadlocks>
          </options>) ,
doc("/docs/test.xml")
```

The call to `doc("/docs/test.xml")` in the last line of the example returns `<a>goodbye</a>`, which is the new version that was updated by the `xdmp:eval` operation.

You can often solve the same problem by using multi-statement transactions. In a multi-statement transaction, updates made by one statement are visible to subsequent statements in the same transaction. Consider the above example, rewritten as a multi-statement transaction. Setting the transaction mode to `update` removes the need for "fake" code to force classification of statements as updates, but adds a requirement to call `xdmp:commit` to make the updates visible in the database.

```
declare option xdmp:transaction-mode "update";

(: doc("/docs/test.xml") returns <a>hello</a> before running this :)
xdmp:eval("xdmp:node-replace(doc('/docs/test.xml')/node(),
<b>goodbye</b>)", (),
          <options xmlns="xdmp:eval">
            <isolation>different-transaction</isolation>
            <prevent-deadlocks>false</prevent-deadlocks>
```

```
          </options>);
(: returns <a>goodbye</b> within this transaction :)
doc("/docs/test.xml"),
(: make updates visible in the database :)
xdmp:commit()
```

3.7.4 Running Multi-Statement Transactions under xdmp:eval/invoke

When you run a query using `xdmp:eval` or `xdmp:invoke` or their JavaScript counterparts with `different-transaction` isolation, or via `xdmp:spawn` or `xdmp.spawn`, a new transaction is created to execute the query, and that transaction runs in a newly created session. This has two important implications for multi-statement transactions evaluated with `xdmp:eval` or `xdmp:invoke`:

- Transaction mode is not inherited from the caller.
- Uncommitted updates are automatically rolled back when an eval/invoke'd or spawned query completes.

Therefore, when using multi-statement transactions in code evaluated under eval/invoke with `different-transaction` isolation or under `xdmp:spawn` or `xdmp.spawn`:

- Set the commit option to "explicit" in the options node if the transaction should run as a multi-statement transaction or use the XQuery `xdmp:commit` prolog option or JavaScript `declareUpdate` function to specify explicit commit mode.
- Always call `xdmp:commit` or `xdmp.commit` inside an eval/invoke'd multi-statement query if updates should be preserved.

Setting the commit mode in the XQuery prolog of the eval/invoke'd query is equivalent to setting it by passing an options node to `xdmp:eval/invoke` with `commit` set to explicit. Setting the mode through the options node enables you to set the commit mode without modifying the eval/invoke'd query.

For an example of using multi-statement transactions with `different-transaction` isolation, see "Example: Multi-Statement Transactions and Different-transaction Isolation" on page 74.

The same considerations apply to multi-statement queries evaluated using `xdmp:spawn` or `xdmp.spawn`.

Transactions run under `same-statement` isolation run in the caller's context, and so use the same transaction mode and benefit from committing the caller's transaction. For a detailed example, see "Example: Multi-statement Transactions and Same-statement Isolation" on page 72.

3.8 Functions With Non-Transactional Side Effects

Update transactions use various update built-in functions which, at the time the transaction commits, update documents in a database. These updates are technically known as *side effects*, because they cause a change to happen outside of what the statements in the transaction return. The side effects from the update built-in functions (`xdmp:node-replace`, `xdmp:document-insert`, and so on) are transactional in nature; that is, they either complete fully or are rolled back to the state at the beginning of the update statement.

Some functionsevaluate asynchronously as soon as they are called, whether called from an update transaction or a query transaction. These functions have side effects outside the scope of the calling statement or the containing transaction (*non-transactional* side effects). The following are some examples of functions that can have non-transactional side effects:

- `xdmp:spawn` (XQuery) or `xdmp.spawn` (JavaScript)
- `xdmp:http-get` (XQuery) or `xdmp.httpGet` (JavaScript)
- `xdmp:log` (XQuery) or `xdmp.log` (JavaScript)

When evaluating a module that performs an update transaction, it is possible for the update to either fail or retry. That is the normal, transactional behavior, and the database will always be left in a consistent state if a transaction fails or retries. However, if your update transaction calls a function with non-transactional side effects, that function evaluates even if the calling update transaction fails and rolls back.

Use care or avoid calling any of these functions from an update transaction, as they are not guaranteed to only evaluate once (or to not evaluate if the transaction rolls back). If you are logging some information with `xdmp:log` or `xdmp.log` in your transaction, it might or might not be appropriate for that logging to occur on retries (for example, if the transaction is retried because a deadlock is detected). Even if it is not what you intended, it might not do any harm.

Other side effects, however, can cause problems in updates. For example, if you use `xdmp:spawn` or `xdmp.spawn` in this context, the action might be spawned multiple times if the calling transaction retries, or the action might be spawned even if the transaction fails; the spawn call evaluates asyncronously as soon as it is called. Similarly, if you are calling a web service with `xdmp:http-get` or `xdmp.httpGet` from an update transaction, it might evaluate when you did not mean for it to evaluate.

If you do use these functions in updates, your application logic must handle the side effects appropriately. These types of use cases are usually better suited to triggers and the Content Processing Framework. For details, see “Using Triggers to Spawn Actions” on page 393 and the *Content Processing Framework Guide* manual.

3.9 Reducing Blocking with Multi-Version Concurrency Control

You can set the "multi-version concurrency control" App Server configuration parameter to `nonblocking` to minimize transaction blocking, at the cost of queries potentially seeing a less timely view of the database. This option controls how the timestamp is chosen for lock-free queries. For details on how timestamps affect queries, see "Query Transactions: Point-in-Time Evaluation" on page 47.

Nonblocking mode can be useful for your application if:

- Low query latency is more important than update latency.
- Your application participates in XA transactions. XA transactions can involve multiple participants and non-MarkLogic Server resources, so they can take longer than usual.
- Your application accesses a replica database which is expected to significantly lag the master. For example, if the master becomes unreachable for some time.

The default multi-version concurrency control is `contemporaneous`. In this mode, MarkLogic Server chooses the most recent timestamp for which any transaction is known to have committed, even if other transactions have not yet fully committed for that timestamp. Queries can block waiting for the contemporaneous transactions to fully commit, but the queries will see the most timely results. The block time is determined by the slowest contemporaneous transaction.

In `nonblocking` mode, the server chooses the latest timestamp for which all transactions are known to have comitted, even if there is a slightly later timestamp for which another transaction has committed. In this mode, queries do not block waiting for contemporaneous transactions, but they might not see the most up to date results.

You can run App Servers with different multi-version concurrency control settings against the same database.

3.10 Administering Transactions

The MarkLogic Server XQuery API include built-in functions helpful for debugging, monitoring, and administering transactions.

Use `xdmp:host-status` to get information about running transactions. The status information includes a `<transactions>` element containing detailed information about every running transaction on the host. For example:

```
<transactions xmlns="http://marklogic.com/xdmp/status/host">
  <transaction>
    <transaction-id>10030469206159559155</transaction-id>
    <host-id>8714831694278508064</host-id>
    <server-id>4212772872039365946</server-id>
    <name/>
    <mode>query</mode>
    <timestamp>11104</timestamp>
    <state>active</state>
```

```
    <database>10828608339032734479</database>
    <canceled>false</canceled>
    <start-time>2011-05-03T09:14:11-07:00</start-time>
    <time-limit>600</time-limit>
    <max-time-limit>3600</max-time-limit>
    <user>15301418647844759556</user>
    <admin>true</admin>
  </transaction>
  ...
</transactions>
```

In a clustered installation, transactions might run on remote hosts. If a remote transaction does not terminate normally, it can be committed or rolled back remotely using `xdmp:transaction-commit` or `xdmp:transaction-rollback`. These functions are equivalent to calling `xdmp:commit` and `xdmp:rollback` when `xdmp:host` is passed as the host id parameter. You can also rollback a transaction through the Host Status page of the Admin Interface. For details, see Rolling Back a Transaction in the *Administrator's Guide*.

Though a call to `xdmp:transaction-commit` returns immediately, the commit only occurs after the currently executing statement in the target transaction succesfully completes. Calling `xdmp:transaction-rollback` immediately interrupts the currently executing statement in the target transaction and terminates the transaction.

For an example of using these features, see "Example: Generating a Transaction Report With xdmp:host-status" on page 75. For details on the built-ins, see the *XQuery & XSLT API Reference*.

3.11 Transaction Examples

This section includes the following examples:

- Example: Multi-statement Transactions and Same-statement Isolation
- Example: Multi-Statement Transactions and Different-transaction Isolation

For an example of tracking system timestamp in relation to wall clock time, see "Keeping Track of System Timestamps" on page 153.

3.11.1 Example: Multi-statement Transactions and Same-statement Isolation

The following example demonstrates the interactions between multi-statement transactions and `same-statement` isolation, discussed in "Interactions with xdmp:eval/invoke" on page 64.

The goal of the sample is to insert a document in the database using `xdmp:eval`, and then examine and modify the results in the calling module. The inserted document should be visible to the calling module immediately, but not visible outside the module until transaction completion.

```
xquery version "1.0-ml";
declare option xdmp:transaction-mode "update";
```

```
(: insert a document in the database :)
let $query :=
  'xquery version "1.0-ml";
   xdmp:document-insert("/examples/mst.xml", <myData/>)
  '
return xdmp:eval(
  $query, (),
  <options xmlns="xdmp:eval">
    <isolation>same-statement</isolation>
  </options>);

(: demonstrate that it is visible to this transaction :)
if (fn:empty(fn:doc("/examples/mst.xml")//myData))
then ("NOT VISIBLE")
else ("VISIBLE");

(: modify the contents before making it visible in the database :)
xdmp:node-insert-child(doc('/examples/mst.xml')/myData, <child/>),
xdmp:commit()

(: result: VISIBLE :)
```

The same operation (inserting and then modifying a document before making it visible in the database) cannot be performed as readily using the default transaction model. If the module attempts the document insert and child insert in the same single-statement transaction, an `XDMP-CONFLICTINGUPDATES` error occurs. Performing these two operations in different single-statement transactions makes the inserted document immediately visible in the database, prior to inserting the child node. Attempting to perform the child insert using a pre-commit trigger creates a trigger storm, as described in "Avoiding Infinite Trigger Loops (Trigger Storms)" on page 404.

The eval'd query runs as part of the calling module's multi-statement update transaction since the eval uses `same-statement` isolation. Since transaction mode is not inherited by transactions created in a different context, using `different-transaction` isolation would evaluate the eval'd query as a single-statement transaction, causing the document to be immediately visible to other transactions.

The call to `xdmp:commit` is required to preserve the updates performed by the module. If `xdmp:commit` is omitted, all updates are lost when evaluation reaches the end of the module. In this example, the commit must happen in the calling module, not in the eval'd query. If the `xdmp:commit` occurs in the eval'd query, the transaction completes when the statement containing the xdmp:eval call completes, making the document visible in the database prior to inserting the child node.

3.11.2 Example: Multi-Statement Transactions and Different-transaction Isolation

The following example demonstrates how `different-transaction` isolation interacts with transaction mode for multi-statement transactons. The same interactions apply to queries executed with `xdmp:spawn`. For more background, see “Transaction Mode” on page 59 and “Interactions with xdmp:eval/invoke” on page 64.

In this example, `xdmp:eval` is used to create a new transaction that inserts a document whose content includes the current transaction id using `xdmp:transaction`. The calling query prints its own transaction id and the transaction id from the eval’d query.

```
xquery version "1.0-ml";

(: init to clean state; runs as single-statement txn :)
xdmp:document-delete("/docs/mst.xml");

(: switch to multi-statement transactions :)
declare option xdmp:transaction-mode "query";

let $sub-query :=
  'xquery version "1.0-ml";
   declare option xdmp:transaction-mode "update";       (: 1 :)

   xdmp:document-insert("/docs/mst.xml", <myData/>);

   xdmp:node-insert-child(
     fn:doc("/docs/mst.xml")/myData,
     <child>{xdmp:transaction()}</child>
   );
   xdmp:commit()                                        (: 2 :)
  '
return xdmp:eval($sub-query, (),
  <options xmlns="xdmp:eval">
    <isolation>different-transaction</isolation>
  </options>);

(: commit to end this transaction and get a new system
 : timestamp so the updates by the eval'd query are visible. :)
xdmp:commit();                                          (: 3 :)

(: print out my transaction id and the eval'd query transaction id :)
fn:concat("My txn id: ", xdmp:transaction()),           (: 4 :)
fn:concat("Subquery txn id: ", fn:doc("/docs/mst.xml")//child)
```

Setting the transaction mode in statement `(: 1 :)` is required because `different-transaction` isolation makes the eval’d query a new transaction, running in its own session, so it does not inherit the transaction mode of the calling context. Omitting xdmp:transaction-mode in the eval’d query causes the eval’d query to run in the default, `auto`, transaction mode.

The call to `xdmp:commit` at statement `(: 2 :)` is similarly required due to `different-transaction` isolation. The new transaction and the containing session end when the end of the eval'd query is reached. Changes are implicitly rolled back if a transaction or the containing session end without committing.

The `xdmp:commit` call at statement `(: 3 :)` ends the multi-statement query transaction that called `xdmp:eval` and starts a new transaction for printing out the results. This causes the final transaction at statement `(: 4 :)` to run at a new timestamp, so it sees the document inserted by xdmp:eval. Since the system timestamp is fixed at the beginning of the transaction, omitting this commit means the inserted document is not visible. For more details, see "Query Transactions: Point-in-Time Evaluation" on page 47.

If the query calling `xdmp:eval` is an update transaction instead of a query transaction, the `xdmp:commit` at statement `(: 3 :)` can be omitted. An update transaction sees the latest version of a document at the time the document is first accessed by the transaction. Since the example document is not accessed until after the `xdmp:eval` call, running the example as an update transaction sees the updates from the eval'd query. For more details, see "Update Transactions: Readers/Writers Locks" on page 48.

3.11.3 Example: Generating a Transaction Report With xdmp:host-status

Use the built-in `xdmp:host-status` to generate a list of the transactions running on a host, similar to the information available through the Host Status page of the Admin Interface.

This example generates a simple HTML report of the duration of all transactions on the local host:

```
xquery version "1.0-ml";

declare namespace html = "http://www.w3.org/1999/xhtml";
declare namespace hs="http://marklogic.com/xdmp/status/host";

<html>
  <body>
    <h2>Running Transaction Report for {xdmp:host-name()}</h2>
    <table border="1" cellpadding="5">
    <tr>
      <th>Transaction Id</th>
      <th>Database</th><th>State</th>
      <th>Duration</th>
    </tr>
    {
      let $txns:= xdmp:host-status(xdmp:host())//hs:transaction
      let $now := fn:current-dateTime()
      for $t in $txns
      return
        <tr>
          <td>{$t/hs:transaction-id}</td>
          <td>{xdmp:database-name($t/hs:database-id)}</td>
          <td>{$t/hs:transaction-state}</td>
          <td>{$now - $t/hs:start-time}</td>
```

```
            </tr>
        }
        </table>
      </body>
    </html>
```

If you paste the above query into Query Console and run it with HTML output, the query generates a report similar to the following:

Running Transaction Report for mumble.marklogic.com

Transaction Id	Database	State	Duration
6335215186646946533	Documents	active	PT1M15.666S
11159175550762420347	App-Services	active	PT0.666S
17156013413075008219	Documents	active	PT0.666S

Many details about each transaction are available in the `xdmp:host-status` report. For more information, see `xdmp:host-status` in the *XQuery & XSLT API Reference*.

If we assume the first transaction in the report represents a deadlock, we can manually cancel it by calling `xdmp:transaction-rollback` and supplying the transaction id. For example:

```
xquery version "1.0-ml";
xdmp:transaction-rollback(xdmp:host(), 6335215186646946533)
```

You can also rollback transactions from the Host Status page of the Admin Interface.

4.0 Working With Binary Documents

This section describes configuring and managing binary documents in MarkLogic Server. Binary documents require special consideration because they are often much larger than text or XML content. The following topics are included:

- Terminology
- Loading Binary Documents
- Configuring MarkLogic Server for Binary Content
- Developing Applications That Use Binary Documents
- Useful Built-ins for Manipulating Binary Documents

4.1 Terminology

The following table describes the terminology used related to binary document support in MarkLogic Server.

Term	Definition
small binary document	A binary document whose contents are managed by the server and whose size does not exceed the large size threshold.
large binary document	A binary document whose contents are managed by the server and whose size exceeds the large size threshold.
external binary document	A binary document whose contents are not managed by the server.
large size threshold	A database configuration setting defining the upper bound on the size of small binary documents. Binary documents larger than the threshold are automatically classified as large binary documents.
Large Data Directory	The per-forest area where the contents of large binary documents are stored.
static content	Content stored in the modules database of the App Server. MarkLogic Server responds directly to HTTP range requests (partial GETs) of static content. See “Downloading Binary Content With HTTP Range Requests” on page 84.
dynamic content	Dynamic content is content generated by your application, such as results returned by XQuery modules. MarkLogic Server does not respond directly to HTTP range requests (partial) GET requests for dynamic content. See “Downloading Binary Content With HTTP Range Requests” on page 84.

4.2 Loading Binary Documents

Loading small and large binary documents into a MarkLogic database does not require special handling, other than potentially explicitly setting the document format. See Choosing a Binary Format in the *Loading Content Into MarkLogic Server Guide*.

External binaries require special handling at load time because they are not managed by MarkLogic. For more information, see Loading Binary Documents.

4.3 Configuring MarkLogic Server for Binary Content

This section covers the MarkLogic Server configuration and administration of binary documents.

- Setting the Large Size Threshold
- Sizing and Scalability of Binary Content
- Selecting a Location For Binary Content
- Monitoring the Total Size of Large Binary Data in a Forest
- Detecting and Removing Orphaned Binaries

4.3.1 Setting the Large Size Threshold

The `large size threshold` database setting defines the maximum size of a small binary, in kilobytes. Any binary documents larger than this threshold are *large* and are stored in the Large Data Directory, as described in Choosing a Binary Format. The threshold has no effect on external binaries.

For example, a threshold of 1024 sets the size threshold to 1 MB. Any (managed) binary document larger than 1 MB is automatically handled as a large binary object.

The range of acceptable threshold values on a 64-bit machine is 32 KB to 512 MB, inclusive.

Many factors must be considered in choosing the large size threshold, including the data characteristics, the access patterns of the application, and the underlying hardware and operating system. Ideally, you should set the threshold such that smaller, frequently accessed binary content such as thumbnails and profile images are classified as *small* for efficient access, while larger documents such as movies and music, which may be streamed by the application, are classified as *large* for efficient memory usage.

The threshold may be set through the Admin Interface or by calling an admin API function. To set the threshold through the Admin Interface, use the `large size threshold` setting on the database configuration page.To set the threshold programmatically, use the XQuery built-in `admin:database-set-large-size-threshold`:

```
xquery version "1.0-ml";

import module namespace admin = "http://marklogic.com/xdmp/admin"
```

```
      at "/MarkLogic/admin.xqy";

let $config := admin:get-configuration()
return
  admin:save-configuration(
    admin:database-set-large-size-threshold(
        $config, xdmp:database("myDatabase"),2048)
```

When the threshold changes, the reindexing process automatically moves binary documents into or out of the Large Data Directory as needed to match the new setting.

4.3.2 Sizing and Scalability of Binary Content

This section covers the following topics:

- Determining the In Memory Tree Size
- Effect of External Binaries on E-node Compressed Tree Cache Size
- Forest Scaling Considerations

For more information on sizing and scalability, see the *Scalability, Availability, and Failover Guide* and the *Query Performance and Tuning Guide*.

4.3.2.1 Determining the In Memory Tree Size

The `in memory tree size` database setting should be at least 1 to 2 megabytes greater than the larger of the `large size threshold` setting or the largest non-binary document you plan to load into the database. That is, 1-2 MB larger than:

```
max(large-size-threshold, largest-expected-non-binary-document)
```

As described in “Selecting a Location For Binary Content” on page 80, the maximum size for small binary documents is 512 MB on a 64-bit system. Large and external binary document size is limited only by the maximum file size supported by the operating system.

To change the `in memory tree size` setting, see the Database configuration page in the Admin Interface or `admin:database-set-in-memory-limit` in the *XQuery and XSLT Reference Guide*.

4.3.2.2 Effect of External Binaries on E-node Compressed Tree Cache Size

If your application makes heavy use of external binary documents, you may need to increase the `compressed tree cache size` Group setting.

When a small binary is cached, the entire document is cached in memory. When a large or external binary is cached, the content is fetched into the compressed tree cache in chunks, as needed.

The chunks of a large binary are fetched into the compressed tree cache of the d-node containing the fragment or document. The chunks of an external binary are fetched into the compressed tree cache of the e-node evaluating the accessing query. Therefore, you may need a larger compressed tree cache size on e-nodes if your application makes heavy use of external binary documents.

To change the `compressed tree cache size`, see the Groups configuration page in the Admin Interface or `admin:group-set-compressed-tree-cache-size` in the *XQuery and XSLT Reference Guide*.

4.3.2.3 Forest Scaling Considerations

When considering forest scaling guidelines, include all types of binary documents in fragment count estimations. Since large and external binaries are not fully cached in memory on access, memory requirements are lower. Since large and external binaries are not copied during merges, you may exclude large and external binary content size from maximum forest size calculation.

For details on sizing and scalability, see Scalability Considerations in MarkLogic Server in the *Scalability, Availability, and Failover Guide*.

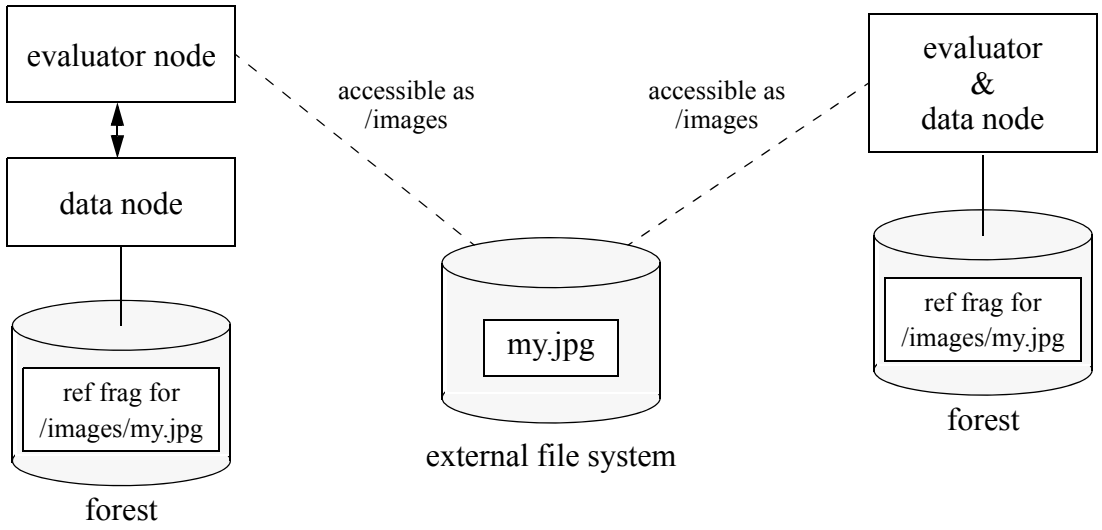
4.3.3 Selecting a Location For Binary Content

Each forest contains a Large Data Directory that holds the binary contents of all large binary documents in the forest. The default physical location of the Large Data Directory is inside the forest. The location is configurable during forest creation. This flexibility allows different hardware to serve small and large binary documents. The Large Data Directory must be accessible to the server instance containing the forest. to specify an arbitrary location for the Large Data Directory, use the `$large-data-directory` parameter of `admin:forest-create` or the `large data directory` forest configuration setting in the Admin Interface.

The external file associated with an external binary document should be located outside the forest containing the document. The external file must be accessible to any server instance evaluating queries that manipulate the document. That is, the external file path used when creating an `external-binary` node must be resolvable on any server instance running queries against the document.

External binary files may be shared across a cluster by placing them on a network shared file system, as long as the files are accessible along the same path from any e-node running queries against the external binary documents. The reference fragment containing the associated `external-binary` node may be located on a remote d-node that does not have access to the external storage.

The diagram below demonstrates sharing external binary content across a cluster with different host configurations. On the left, the evaluator node (e-node) and data node (d-node) are separate hosts. On the right, the same host serves as both an evaluator and data node. The database in both configurations contains an external binary document referencing `/images/my.jpg`. The JPEG content is stored on shared external storage, accessible to the evaluator nodes through the external file path stored in the external binary document in the database.



4.3.4 Monitoring the Total Size of Large Binary Data in a Forest

Use `xdmp:forest-status` or the Admin Interface to check the disk space consumed by large binary documents in a forest. The size is reported in megabytes. For more details on MarkLogic Server's monitoring capability, see the *Monitoring MarkLogic Guide*.

To check the size of the Large Data Directory using the Admin Interface:

1. Open the Admin Interface in your browser. For example, http:yourhost:8001.
2. Click Forests in the left tree menu. The Forest summary is displayed.
3. Click the name of a forest to display the forest configuration page.
4. Click the Status tab at the top to display the forest status page.
5. Observe the "Large Data Size" status, which reflects the total size of the contents of the large data directory.

The following example uses `xdmp:forest-status` to retrieve the size of the Large Data Directory:

```
xquery version "1.0-ml";
declare namespace fs = "http://marklogic.com/xdmp/status/forest";
fn:data(
```

```
xdmp:forest-status(
  xdmp:forest("samples-1"))/fs:large-data-size)
```

4.3.5 Detecting and Removing Orphaned Binaries

Large and external binary content may require special handling to detect and remove orphaned binary data no longer associated with a document in the database. This section covers the following topics related to managing orphaned binary content:

- Detecting and Removing Orphaned Large Binary Content
- Detecting and Removing Orphaned External Binary Content

4.3.5.1 Detecting and Removing Orphaned Large Binary Content

As discussed in Choosing a Binary Format in the *Loading Content Into MarkLogic Server Guide*, the binary content of a large binary document is stored in the Large Data Directory.

Normally, the server ensures that the binary content is removed when the containing forest no longer contains references to the data. However, content may be left behind in the Large Data Directory under some circumstances, such as a failover in the middle of inserting a binary document. Content left behind in the Large Data Directory with no corresponding database reference fragment is an orphaned binary.

If your data includes large binary documents, you should periodically check for and remove orphaned binaries. Use `xdmp:get-orphaned-binaries` and `xdmp:remove-orphaned-binary` to perform this cleanup. For example:

```
xquery version "1.0-ml";

for $fid in xdmp:forests()
  for $orphan in xdmp:get-orphaned-binaries($fid)
  return xdmp:remove-orphaned-binary($fid, $orphan)
```

4.3.5.2 Detecting and Removing Orphaned External Binary Content

Since the external file associated with an external binary document is not managed by MarkLogic Server, such documents may be associated with non-existent external files. For example, the external file may be removed by an outside agency. The XQuery API includes several builtins to help you check for and remove such documents in the database.

For example, to remove all external binary documents associated with the external binary file `/external/path/sample.jpg`, use `xdmp:external-binary-path`:

```
xquery version "1.0-ml";
for $doc in fn:collection()/binary()
where xdmp:external-binary-path($doc) = "/external/path/sample.jpg"
return xdmp:document-delete(xdmp:node-uri($doc))
```

To identify external binary documents with non-existent external files, use `xdmp:filesystem-file-exists`. Note, however, that `xdmp:filesystem-file-exists` queries the underlying filesystem, so it is a relatively expensive operation. The following example generates a list of document URIs for external binary documents with a missing external file:

```
xquery version "1.0-ml";
for $doc in fn:collection()/binary()
where xdmp:binary-is-external($doc)
return
  if (xdmp:filesystem-file-exists(xdmp:external-binary-path($doc)))
  then xdmp:node-uri($doc)
  else ()
```

4.4 Developing Applications That Use Binary Documents

This section covers the following topics of interest to developers creating applications that manipulate binary content:

- Adding Metadata to Binary Documents Using Properties
- Downloading Binary Content With HTTP Range Requests
- Creating Binary Email Attachments

4.4.1 Adding Metadata to Binary Documents Using Properties

Small, large, and external binary documents may be annotated with metadata using properties. Any document in the database may have an associated properties document for storing additional XML data. Unlike binary data, properties documents may participate in element indexing. For more information about using properties, see “Properties Documents and Directories” on page 131.

MarkLogic Server server offers the XQuery built-in, `xdmp:document-filter`, and JavaScript method, `xdmp.documentFilter`, to assist with adding metadata to binary documents. These functions extract metadata and text from binary documents as a node, each of whose child elements represent a piece of metadata. The results may be used as document properties. The text extracted contains little formatting or structure, so it is best used for search, classification, or other text processing.

For example, the following code creates properties corresponding to just the metadata extracted by `xdmp:document-filter` from a Microsoft Word document:

```
xquery version "1.0-ml";
let $the-document := "/samples/sample.docx"
return xdmp:document-set-properties(
  $the-document,
  for $meta in xdmp:document-filter(fn:doc($the-document))//*:meta
  return element {$meta/@name} {fn:string($meta/@content)}
)
```

The result properties document contains properties such as Author, AppName, and Creation_Date, extracted by `xdmp:document-filter`:

```
<prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
  <content-type>application/msword</content-type>
  <filter-capabilities>text subfiles HD-HTML</filter-capabilities>
  <AppName>Microsoft Office Word</AppName>
  <Author>MarkLogic</Author>
  <Company>Mark Logic Corporation</Company>
  <Creation_Date>2011-09-05T16:21:00Z</Creation_Date>
  <Description>This is my comment.</Description>
  <Last_Saved_Date>2011-09-05T16:22:00Z</Last_Saved_Date>
  <Line_Count>1</Line_Count>
  <Paragraphs_Count>1</Paragraphs_Count>
  <Revision>2</Revision>
  <Subject>Creating binary doc props</Subject>
  <Template>Normal.dotm</Template>
  <Typist>MarkLogician</Typist>
  <Word_Count>7</Word_Count>
  <isys>SubType: Word 2007</isys>
  <size>10047</size>
  <prop:last-modified>2011-09-05T09:47:10-07:00</prop:last-modified>
</prop:properties>
```

4.4.2 Downloading Binary Content With HTTP Range Requests

HTTP applications often use range requests (sometimes called partial GET) to serve large data, such as videos. MarkLogic Server directly supports HTTP range requests for *static binary content*. Static binary content is binary content stored in the modules database of the App Server. Range requests for *dynamic binary content* are not directly supported, but you may write application code to service such requests. Dynamic binary content is any binary content generated by your application code.

This section covers the following topics related to serving binary content in response to range requests:

- Responding to Range Requests with Static Binary Content
- Responding to Range Requests with Dynamic Binary Content

4.4.2.1 Responding to Range Requests with Static Binary Content

When an HTTP App Server receives a range request for a binary document in the modules database, it responds directly, with no need for additional application code. Content in the modules database is considered “static” content. You can configure an App Server to use any database as modules database, enabling MarkLogic to respond to directly to range requests for static binary content.

For example, suppose your database contains a large binary document with the URI "/images/really_big.jpg" and you create an HTTP App Server on port 8010 that uses this database as its modules database. Sending a GET request of the following form to port 8010 directly fetches the binary document:

```
GET http://host:8010/images/really_big.jpg
```

If you include a range in the request, then you can incrementally stream the document out of the database. For example:

```
GET http://host:8010/images/really_big.jpg
Range: bytes=0-499
```

MarkLogic returns the first 500 byes of the document `/images/really_big.jpg` in a `Partial Content` response with a 206 (Partial Content) status, similar to the following (some headers are omitted for brevity):

```
HTTP/1.0 206 Partial Content
Accept-Ranges: bytes
Content-Length: 500
Content-Range: bytes 0-499/3980
Content-Type: image/jpeg

[first 500 bytes of /images/really_big.jpg]
```

If the range request includes multiple non-overlapping ranges, the App Server responds with a 206 and a multi-part message body with media type "multipart/byteranges".

If a range request cannot be satisfied, the App Server responds with a 416 status (Requested Range Not Satisfiable).

The following request types are directly supported on static content:

- Single range requests
- Multiple range requests
- If-Range requests with an HTTP-date

If-Range requests with an entity tag are unsupported.

4.4.2.2 Responding to Range Requests with Dynamic Binary Content

The HTTP App Server does not respond directly to HTTP range requests for dynamic content. That is, content generated by application code. Though the App Server ignores range requests for dynamic content, your application XQuery code may still process the Range header and respond with appropriate content.

The following code demonstrates how to interpret a Range header and return dynamically generated content in response to a range request:

```
xquery version "1.0-ml";

(: This code assumes a simple range like 1000-2000; your :)
(: application code may support more complex ranges.     :)

let $data := fn:doc(xdmp:get-request-field("uri"))/binary()
let $range := xdmp:get-request-header("Range")
return
  if ($range)
  then
    let $range := replace(normalize-space($range), "bytes=", "")
    let $splits := tokenize($range, "-")
    let $start := xs:integer($splits[1])
    let $end := if ($splits[2] eq "")
                then xdmp:binary-size($data)-1
                else xs:integer($splits[2])
    let $ranges  :=
        concat("bytes ", $start, "-", $end, "/",
               xdmp:binary-size($data))
    return (xdmp:add-response-header("Content-Range", $ranges),
            xdmp:set-response-content-type("image/JPEG"),
            xdmp:set-response-code(206, "Partial Content"),
            xdmp:subbinary($data, $start+1, $end - $start + 1))
    else $data
```

If the above code is in an XQuery module fetch-bin.xqy, then a request such the following returns the first 100 bytes of a binary. (The `-r` option to the `curl` command specifies a byte range).

```
$ curl -r "0-99" http://myhost:1234/fetch-bin.xqy?uri=sample.jpg
```

The response to the request is similar to the following:

```
HTTP/1.1 206 Partial Content
Content-Range: bytes 0-99/1442323
Content-type: image/JPEG
Server: MarkLogic
Content-Length: 100

[first 100 bytes of sample.jpg]
```

4.4.3 Creating Binary Email Attachments

To generate an email message with a binary attachment, use xdmp:email and set the content type of the message to multipart/mixed. The following example generates an email message with a JPEG attachment:

```
xquery version "1.0-ml";

(: generate a random boundary string :)
let $boundary := concat("blah", xdmp:random())
let $newline := "&#13;&#10;"
let $content-type := concat("multipart/mixed; boundary=",$boundary)
```

```
let $attachment1 := xs:base64Binary(doc("/images/sample.jpeg"))
let $content := concat(
  "--",$boundary,$newline,
  $newline,
  "This is a test email with an image attached.", $newline,
  "--",$boundary,$newline,
  "Content-Type: image/jpeg", $newline,
  "Content-Disposition: attachment; filename=sample.jpeg", $newline,
  "Content-Transfer-Encoding: base64", $newline,
  $newline,
  $attachment1, $newline,
  "--",$boundary,"--", $newline)

return
  xdmp:email(
  <em:Message
    xmlns:em="URN:ietf:params:email-xml:"
    xmlns:rf="URN:ietf:params:rfc822:">
    <rf:subject>Sample Email</rf:subject>
    <rf:from>
      <em:Address>
        <em:name>Myself</em:name>
        <em:adrs>me@somewhere.com</em:adrs>
      </em:Address>
    </rf:from>
    <rf:to>
      <em:Address>
        <em:name>Somebody</em:name>
        <em:adrs>somebody@somewhere.com</em:adrs>
      </em:Address>
    </rf:to>
    <rf:content-type>{$content-type}</rf:content-type>
    <em:content xml:space="preserve">
      {$content}
    </em:content>
  </em:Message>)
```

4.5 Useful Built-ins for Manipulating Binary Documents

The following XQuery built-ins are provided for working with binary content. For details, see the *XQuery and XSLT Reference Guide*.

- `xdmp:subbinary`
- `xdmp:binary-size`
- `xdmp:external-binary`
- `xdmp:external-binary-path`
- `xdmp:binary-is-small`
- `xdmp:binary-is-large`
- `xdmp:binary-is-external`

In addition, the following XQuery built-ins may be useful when creating or testing the integrity of external binary content:

- `xdmp:filesystem-file-length`
- `xdmp:filesystem-file-exists`

5.0 Importing XQuery Modules, XSLT Stylesheets, and Resolving Paths

You can import XQuery into other XQuery and/or Server-Side JavaScript modules. Similarly, you can import XSLT stylesheets into other stylesheets, you can import XQuery modules into XSLT stylesheets, and you can import XSLT stylesheets into XQuery modules.

This chapter describes the two types of XQuery modules and specifies the rules for importing modules and resolving URI references. To import XQuery into Server-Side JavaScript modules, see Using XQuery Functions and Variables in JavaScript in the *JavaScript Reference Guide*.

This chapter covers the following topics:

- XQuery Library Modules and Main Modules
- Rules for Resolving Import, Invoke, and Spawn Paths
- Module Caching Notes
- Example Import Module Scenario

For details on importing XQuery library modules into XSLT stylesheets and vice-versa, see Notes on Importing Stylesheets With <xsl:import> and Importing a Stylesheet Into an XQuery Module in the *XQuery and XSLT Reference Guide*.

5.1 XQuery Library Modules and Main Modules

There are two types of XQuery modules (as defined in the XQuery specification, http://www.w3.org/TR/xquer//#id-query-prolog):

- Main Modules
- Library Modules

For more details about the XQuery language, see the *XQuery and XSLT Reference Guide*.

5.1.1 Main Modules

A main module can be executed as an XQuery program, and must include a query body consisting of an XQuery expression (which in turn can contain other XQuery expressions, and so on). The following is a simple example of a main module:

```
"hello world"
```

Main modules can have prologs, but the prolog is optional. As part of a prolog, a main module can have function definitions. Function definitions in a main module, however, are only available to that module; they cannot be imported to another module.

5.1.2 Library Modules

A library module has a namespace and is used to define functions. Library modules cannot be evaluated directly; they are imported, either from other library modules or from main modules with an `import` statement. The following is a simple example of a library module:

```
xquery version "1.0-ml";
module namespace hello = "helloworld";

declare function helloworld()
{
  "hello world"
};
```

If you insert the module into the modules database of your App Server or save it on the filesystem under the modules root directory of your App Server, then you can import the module and call the "helloworld" function.

For example, suppose you save the above module to the filesystem with the pathname `/my/app/helloworld.xqy`. If you configure an App Server to use "Modules" as the modules database and "/" as the modules root, then you can store the module in the modules database as follows:

```
xquery version "1.0-ml";
xdmp:eval('xdmp:document-load("/space/rest/helloworld.xqy")', (),
  <options xmlns='xdmp:eval'>
    <database>{xdmp:database('Modules')}</database>
  </options>)
```

The inserted module has the URI `/my/app/helloworld.xqy`. Now, you can import the module in a main module or library module and call the "helloworld" function as follows:

```
xquery version "1.0-ml";
import module namespace hw="helloworld" at "/my/app/helloworld.xqy";

hw:helloworld()
```

The same import statement works if you configure an App server to use the filesystem as the modules "database" and "/" as the modules root. In this case, the query imports the module from the filesystem instead of from the modules database.

5.2 Rules for Resolving Import, Invoke, and Spawn Paths

In order to call a function that resides in an XQuery library module, you need to import the module with its namespace. MarkLogic Server resolves the library paths similar to the way other HTTP and application servers resolve their paths. Similarly, if you use `xdmp:invoke` or `xdmp:spawn` to run a module, you specify access to the module with a path. These rules also apply to the path to an XSLT stylesheet when using `xdmp:xslt-invoke`, as well as to stylesheet imports in the `<xsl:import>` or `<xsl:include>` instructions.

The XQuery module that is imported/invoked/spawned can reside in any of the following places:

- In the Modules directory.
- In a directory relative to the calling module.
- Under the App Server root, which is either the specified directory in the Modules database (when the App Server is set to a Modules database) or the specified directory on the filesystem (when the App Server is set to find modules in the filesystem).

When resolving `import`/`invoke`/`spawn` paths, MarkLogic first resolves the root of the path, and then looks for the module under the Modules directory first and the App Server root second, using the first module it finds that matches the path.

The paths in `import`/`invoke`/`spawn` expressions are resolved as follows:

1. When an `import`/`invoke`/`spawn` path starts with a leading slash, first look under the Modules directory (on Windows, typically `c:\Program Files\MarkLogic\Modules`). For example:

   ```
   import module "foo" at "/foo.xqy";
   ```

   In this case, it would look for the module file with a namespace `foo` in `c:\Program Files\MarkLogic\Modules\foo.xqy`.

2. If the import/invoke/spawn path starts with a slash, and it is not found under the Modules directory, then start at the App Server root. For example, if the App Server root is `/home/mydocs/`, then the following import:

   ```
   import module "foo" at "/foo.xqy";
   ```

   will look for a module with namespace `foo` in `/home/mydocs/foo.xqy`.

   Note that you start at the App Server root, both for filesystem roots and Modules database roots. For example, in an App Server configured with a modules database and a root of `http://foo/`:

   ```
   import module "foo" at "/foo.xqy";
   ```

   will look for a module with namespace `foo` in the modules database with a URI `http://foo/foo.xqy` (resolved by appending the App Server root to `foo.xqy`).

3. If the import/invoke/spawn path does not start with a slash, first look under the Modules directory. If the module is not found there, then look relative to the location of the module

that called the function. For example, if a module at `/home/mydocs/bar.xqy` has the following import:

```
import module "foo" at "foo.xqy";
```

it will look for the module with namespace `foo` at `/home/mydocs/foo.xqy`.

Note that you start at the calling module location, both for App Servers configured to use the filesystem and for App Servers configured to use modules databases. For example, a module with a URI of `http://foo/bar.xqy` that resides in the modules database and has the following `import` statement:

```
import module "foo" at "foo.xqy";
```

will look for the module with the URI `http://foo/foo.xqy` in the modules database.

4. If the import/invoke/spawn path contains a scheme or network location, then the server throws an exception. For example:

```
import module "foo" at "http://foo/foo.xqy";
```

will throw an invalid path exception. Similarly:

```
import module "foo" at "c:/foo/foo.xqy";
```

will throw an invalid path exception.

5.3 Module Caching Notes

When XQuery modules (or XSLT files) are stored in the root for an App Server configured in MarkLogic Server, when they are first accessed, each module is parsed and then cached in memory so that subsequent access to the module is faster. If a module is updated, the cache is invalidated and each module for that App Server requires parsing again the next time it is evaluated. The module caching is automatic and therefore is transparent to developers. When considering the naming of modules, however, note the following:

- The best practice is to use a file extension for a module corresponding to `application/vnd.marklogic-xdmp` or `application/xslt+xml` mimetypes. By default, this includes the extensions `xqy`, `xq`, and `xslt`. You can add other extensions to these mimetypes using the mimetypes configuration in the Admin Interface.
- Any changes to modules that do not have a mimetype extension corresponding to `application/vnd.marklogic-xdmp` or `application/xslt+xml` will not invalidate the module cache, and therefore you must reload the cache on each host (for example, by restarting the server or modifying a module with the proper extension) to see changes in a module that does not have the correct extension.

5.4 Example Import Module Scenario

Consider the following scenario:

- There is an HTTP server with a root defined as `c:/mydir`.
- In a file called `c:/mydir/lib.xqy`, there is a library module with the function to import. The contents of the library module are as follows:

```
xquery version "1.0-ml";
module namespace hw="http://marklogic.com/me/my-module";

declare function hello()
{
"hello"
};
```

- In a file called `c:/mydir/main.xqy`, there is an XQuery main module that imports a function from the above library module. This code is as follows:

```
xquery version "1.0-ml";

declare namespace my="http://marklogic.com/me/my-module";
import module "http://marklogic.com/me/my-module" at "lib.xqy";

my:hello()
```

The library module `lib.xqy` is imported relative to the App Server root (in this case, relative to `c:/mydir`).

6.0 Library Services Applications

This chapter describes how to use Library Services, which enable you to create and manage versioned content in MarkLogic Server in a manner similar to a Content Management System (CMS). This chapter includes the following sections:

- Understanding Library Services
- Building Applications with Library Services
- Required Range Element Indexes
- Library Services API
- Security Considerations of Library Services Applications
- Transactions and Library Services
- Putting Documents Under Managed Version Control
- Checking Out Managed Documents
- Checking In Managed Documents
- Updating Managed Documents
- Defining a Retention Policy
- Managing Modular Documents in Library Services

6.1 Understanding Library Services

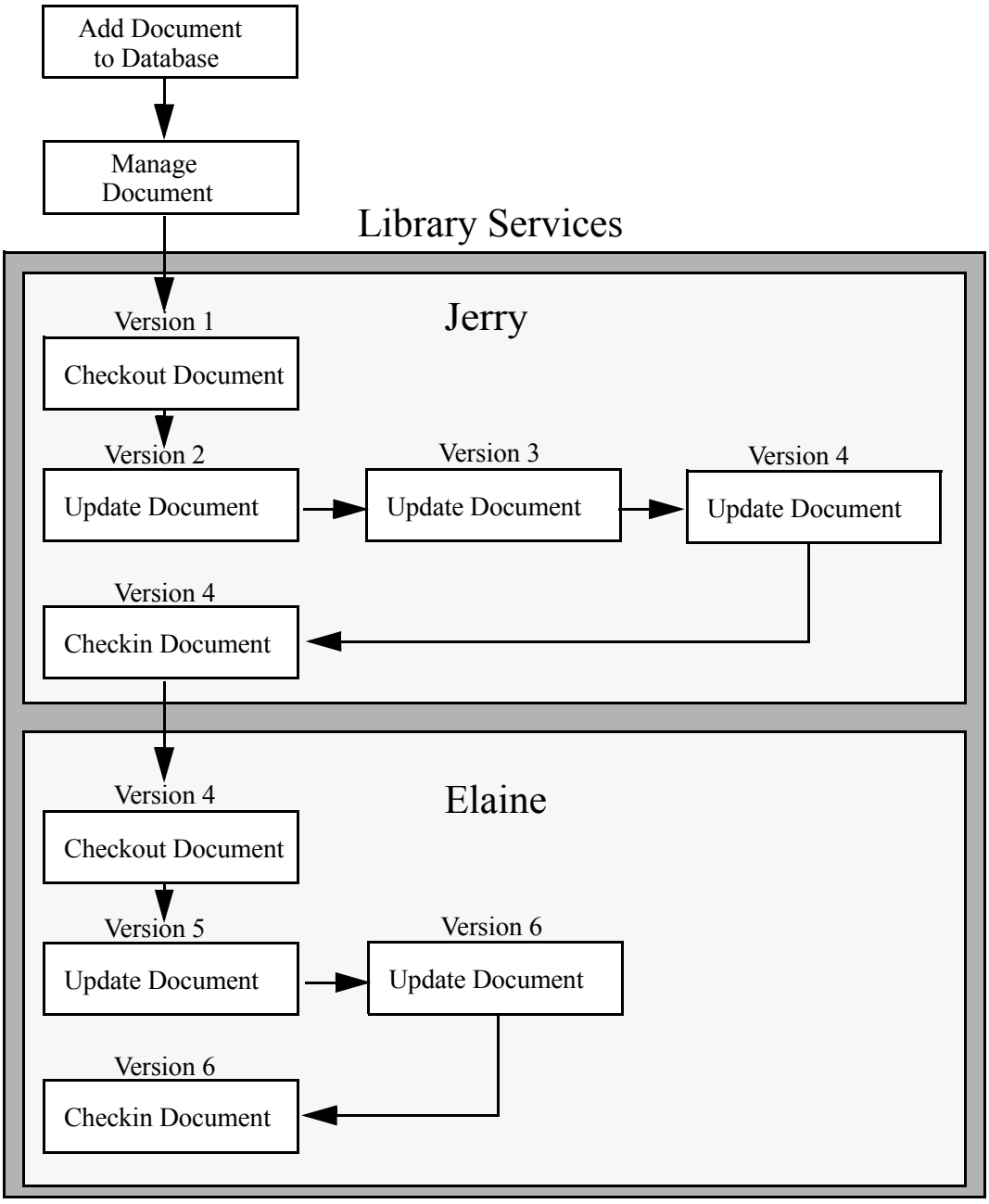
The Library Services enable you to create and maintain versions of managed documents in MarkLogic Server. Access to managed documents is controlled using a check-out/check-in model. You must first check out a managed document before you can perform any update operations on the document. A checked out document can only be updated by the user who checked it out; another user cannot update the document until it is checked back in and then checked out by the other user.

> **Note:** Documents must be stored in a database to be versioned. If a document is created by a CPF application, such as entity enrichment, modular documents, conversion, or a custom CPF application, then the document will only be versioned if the CPF application uses Library Services to insert it into the database. By default, the CPF applications supplied by MarkLogic do not create managed documents.

When you initially put a document under Library Services management, it creates Version 1 of the document. Each time you update the document, a new version of the document is created. Old versions of the updated document are retained according to your retention policy, as described in “Defining a Retention Policy” on page 104.

The Library Services include functions for managing modular documents so that various versions of linked documents can be created and managed, as described in “Managing Modular Documents in Library Services” on page 111.

The following diagram illustrates the workflow of a typical managed document. In this example, the document is added to the database and placed under Library Services management. The managed document is checked out, updated several times, and checked in by Jerry. Once the document is checked in, Elaine checks out, updates, and checks in the same managed document. Each time the document is updated, the previous versions of the document are purged according to the retention policy.



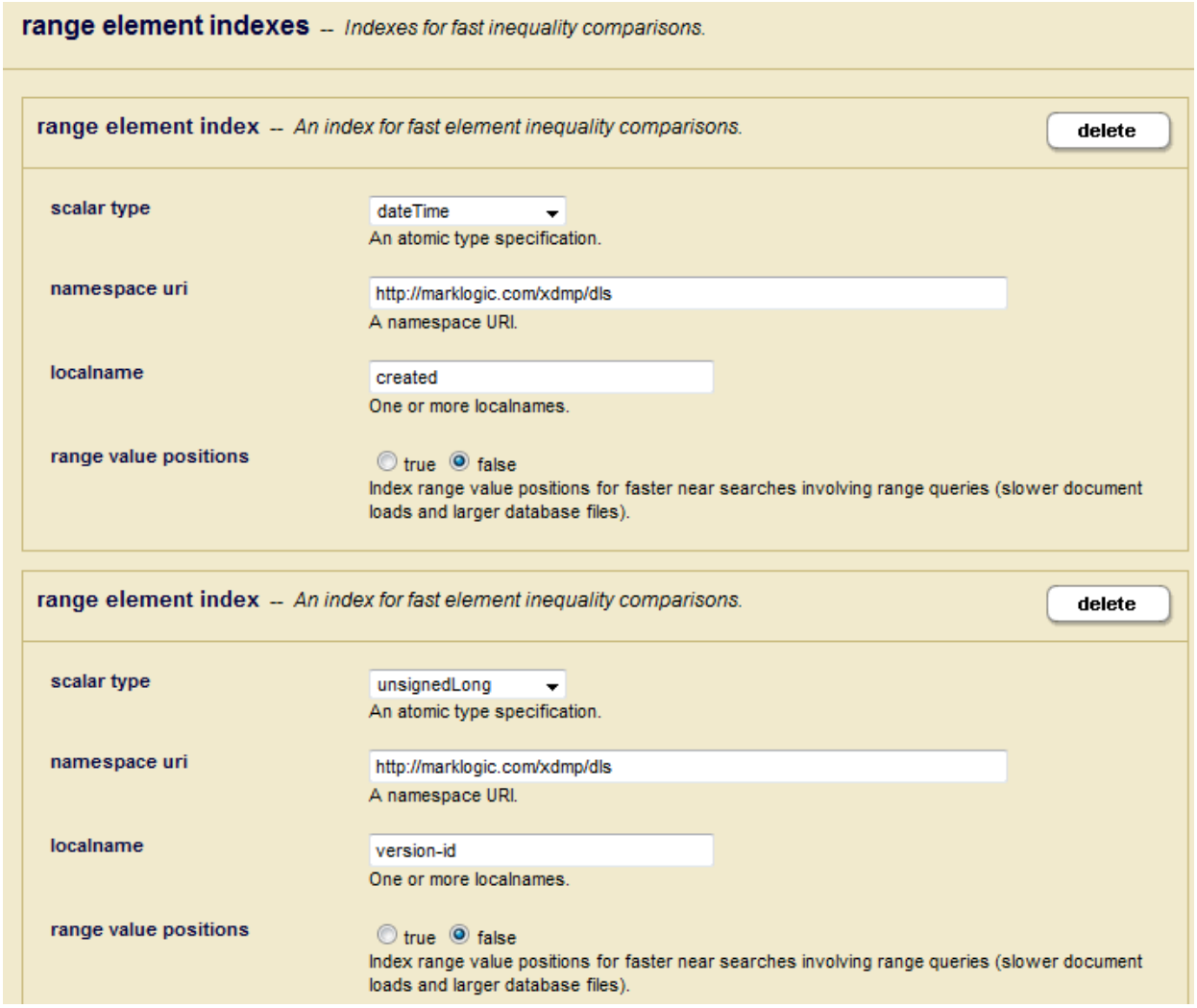
6.2 Building Applications with Library Services

The Library Services API provides the basic tools for implementing applications that store and extract specific drafts of a document as of a particular date or version. You can also use the Library Services API, along with the other MarkLogic Server APIs, to provide structured workflow, version control, and the ability to partition a document into individually managed components. The security API provides the ability to associate user roles and responsibilities with different document types and collections. And the search APIs provide the ability to implement powerful content retrieval features.

6.3 Required Range Element Indexes

The range element indexes shown in the table and figure below must be set for the database that contains the documents managed by the Library Services. These indexes are automatically set for you when you create a new database. However, if you want to enable the Library Services for a database created in an earlier release of MarkLogic Server, you must manually set them for the database.

Scalar Type	Namespace URI	Local Name	Range Value Position
dateTime	http://marklogic.com/xdmp/dls	created	false
unsignedLong	http://marklogic.com/xdmp/dls	version-id	false



6.4 Library Services API

This section describes the Library Services API and contains the following sections:

- Library Services API Categories
- Managed Document Update Wrapper Functions

6.4.1 Library Services API Categories

The Library Services functions are described in the *MarkLogic XQuery and XSLT Function Reference*. The Library Services functions fall into the following categories:

- Document management functions for putting documents under version management, checking documents in and out of version management, and so on. For usage information, see "Putting Documents Under Managed Version Control" on page 101, "Checking Out Managed Documents" on page 102 and "Checking In Managed Documents" on page 103.
- Document update functions for updating the content of documents and their properties. For usage information, see "Updating Managed Documents" on page 103 and "Managed Document Update Wrapper Functions" on page 99.
- Retention policy functions for managing when particular document versions are purged. For usage information, see "Defining a Retention Policy" on page 104.
- XInclude functions for creating and managing linked documents. For usage information, see "Managing Modular Documents in Library Services" on page 111.
- `cts:query` constructor functions for use by `cts:search`, Library Services XInclude functions, and when defining retention rules. For usage information, see "Defining a Retention Policy" on page 104.

6.4.2 Managed Document Update Wrapper Functions

All update and delete operations on managed documents must be done through the Library Services API. The Library Services API includes the following "wrapper" functions that enable you to make the same updates on managed documents as you would on non-managed document using their XDMP counterparts:

- `dls:document-add-collections`
- `dls:document-add-permissions`
- `dls:document-add-properties`
- `dls:document-set-collections`
- `dls:document-set-permissions`
- `dls:document-set-properties`
- `dls:document-remove-properties`
- `dls:document-remove-permissions`
- `dls:document-remove-collections`
- `dls:document-set-property`
- `dls:document-set-quality`

Note: If you only change the collection or property settings on a document, these settings will not be maintained in version history when the document is checked in. You must also change the content of the document to version changes to collections or properties.

6.5 Security Considerations of Library Services Applications

There are two pre-defined roles designed for use in Library Services applications, as well as an internal role that the Library Services API uses:

- dls-admin Role
- dls-user Role
- dls-internal Role

Note: Do not log in with the Admin role when inserting managed documents into the database or when testing your Library Services applications. Instead create test users with the `dls-user` role and assign them the various permissions needed to access the managed documents. When testing your code in Query Console, you must also assign your test users the `qconsole-user` role.

6.5.1 dls-admin Role

The `dls-admin` role is designed to give administrators of Library Services applications all of the privileges that are needed to use the Library Services API. It has the needed privileges to perform operations such as inserting retention policies and breaking checkouts, so only trusted users (users who are assumed to be non-hostile, appropriately trained, and follow proper administrative procedures) should be granted the `dls-admin` role. Assign the `dls-admin` role to administrators of your Library Services application.

6.5.2 dls-user Role

The `dls-user` role is a minimally privileged role. It is used in the Library Services API to allow regular users of the Library Services application (as opposed to `dls-admin` users) to be able to execute code in the Library Services API. It allows users, with document update permission, to manage, checkout, and checkin managed documents.

The `dls-user` role only has privileges that are needed to run the Library Services API; it does not provide execute privileges to any functions outside the scope of the Library Services API. The Library Services API uses the `dls-user` role as a mechanism to amp more privileged operations in a controlled way. It is therefore reasonably safe to assign this role to any user whom you trust to use your Library Services application. Assign the `dls-user` role to all users of your Library Services application.

6.5.3 dls-internal Role

The `dls-internal` role is a role that is used internally by the Library Services API, but you should not explicitly grant it to any user or role. This role is used to amp special privileges within the context of certain functions of the Library Services API. Assigning this role to users would give them privileges on the system that you typically do not want them to have; do not assign this role to any users.

6.6 Transactions and Library Services

The `dls:document-checkout`, `dls:document-update`, and `dls:document-checkin` functions must be executed in separate transactions. If you want to complete a checkout, update, and checkin in a single transaction, use the `dls:document-checkout-update-checkin` function.

6.7 Putting Documents Under Managed Version Control

In order to put a document under managed version control, it must be in your content database. Once the document is in the database, users assigned the `dls-user` role can use the `dls:document-manage` function to place the document under management. Alternatively, you can use the `dls:document-insert-and-manage` function to both insert a document into the database and place it under management.

When inserting a managed document, you should specify at least read and update permissions to the roles assigned to the users that are to manage the document. If no permissions are supplied, the default permissions of the user inserting the managed document are applied. The default permissions can be obtained by calling the `xdmp:default-permissions` function. When adding a collection to a document, as shown in the example below, the user will also need the `unprotected-collections` privilege.

For example, the following query inserts a new document into the database and places it under Library Services management. This document can only be read or updated by users assigned the `writer` and/or `editor` role and have permission to read and update the `http://marklogic.com/engineering/specs` collection.

```
(: Insert a new managed document into the database. :)
xquery version "1.0-ml";

import module namespace dls = "http://marklogic.com/xdmp/dls"
        at "/MarkLogic/dls.xqy";

dls:document-insert-and-manage(
        "/engineering/beta_overview.xml",
        fn:true(),
        <TITLE>Project Beta Overview</TITLE>,
        "Manage beta_overview.xml",
        (xdmp:permission("writer", "read"),
         xdmp:permission("writer", "update"),
         xdmp:permission("editor", "read"),
         xdmp:permission("editor", "update")),
        ("http://marklogic.com/engineering/specs"))
```

6.8 Checking Out Managed Documents

You must first use the `dls:document-checkout` function to check out a managed document before performing any update operations. For example, to check out the `beta_overview.xml` document, along with all of its linked documents, specify the following:

```
xquery version "1.0-ml";

import module namespace dls = "http://marklogic.com/xdmp/dls"
        at "/MarkLogic/dls.xqy";

dls:document-checkout(
        "/engineering/beta_overview.xml",
        fn:true(),
        "Updating doc")
```

You can specify an optional `timeout` parameter to `dls:document-checkout` that specifies how long (in seconds) to keep the document checked out. For example, to check out the `beta_overview.xml` document for one hour, specify the following:

```
dls:document-checkout(
        "/engineering/beta_overview.xml",
        fn:true(),
        "Updating doc",
        3600)
```

6.8.1 Displaying the Checkout Status of Managed Documents

You can use the `dls:document-checkout-status` function to report the status of a checked out document. For example:

```
dls:document-checkout-status("/engineering/beta_overview.xml")
```

Returns output similar to:

```
<dls:checkout xmlns:dls="http://marklogic.com/xdmp/dls">
  <dls:document-uri>/engineering/beta_overview.xml</dls:document-uri>
  <dls:annotation>Updating doc</dls:annotation>
  <dls:timeout>0</dls:timeout>
  <dls:timestamp>1240528210</dls:timestamp>
  <sec:user-id xmlns:sec="http://marklogic.com/xdmp/security">
    10677693687367813363
  </sec:user-id>
</dls:checkout>
```

6.8.2 Breaking the Checkout of Managed Documents

Users with `dls-admin` role can call `dls:break-checkout` to "un-checkout" documents. For example, if a document was checked out by a user who has since moved on to other projects, the Administrator can break the existing checkout of the document so that other users can check it out.

6.9 Checking In Managed Documents

Once you have finished updating the document, use the `dls:document-checkin` function to check it, along with all of its linked documents, back in:

```
dls:document-checkin(
        "/engineering/beta_overview.xml",
        fn:true() )
```

6.10 Updating Managed Documents

You can call the `dls:document-update` function to replace the contents of an existing managed document. Each time you call the `dls:document-update` function on a document, the document's version is incremented and a purge operation is initiated that removes any versions of the document that are not retained by the retention policy, as described in "Defining a Retention Policy" on page 104.

> **Note:** You cannot use node update functions, such as `xdmp:node-replace`, with managed documents. Updates to the document must be done in memory before calling the `dls:document-update` function. For information on how to do in-memory updates on document nodes, see "Transforming XML Structures With a Recursive typeswitch Expression" on page 117.

For example, to update the "Project Beta Overview" document, enter:

```
let $contents :=
<BOOK>
  <TITLE>Project Beta Overview</TITLE>
  <CHAPTER>
    <TITLE>Objectives</TITLE>
    <PARA>
    The objective of Project Beta, in simple terms, is to corner
    the widget market.
    </PARA>
  </CHAPTER>
</BOOK>

return
    dls:document-update(
       "/engineering/beta_overview.xml",
       $contents,
       "Roughing in the first chapter",
       fn:true())
```

> **Note:** The `dls:document-update` function replaces the entire contents of the document.

6.11 Defining a Retention Policy

A *retention policy* specifies what document versions are retained in the database following a purge operation. A retention policy is made up of one or more *retention rules*. If you do not define a retention policy, then none of the previous versions of your documents are retained.

This section describes:

- Purging Versions of Managed Document
- About Retention Rules
- Creating Retention Rules
- Retaining Specific Versions of Documents
- Multiple Retention Rules
- Deleting Retention Rules

6.11.1 Purging Versions of Managed Document

Each update of a managed document initiates a purge operation that removes the versions of that document that are not retained by your retention policy. You can also call `dls:purge` to purge all of the documents or `dls:document-purge` to run purge on a specific managed document.

You can also use `dls:purge` or `dls:document-purge` to determine what documents *would* be deleted by the retention policy without actually deleting them. This option can be useful when developing your retention rules. For example, if you change your retention policy and want to determine specifically what document versions will be deleted as a result, you can use:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:purge(fn:false(), fn:true())
```

6.11.2 About Retention Rules

Retention rules describe which versions of what documents are to be retained by the purge operation. When using `dls:document-update` or `dls:document-extract-part` to create a new version of a document, previous versions of the document that do not match the retention policy are purged.

You can define retention rules to keep various numbers of document versions, to keep documents matching a `cts-query` expression, and/or to keep documents for a specified period of time. Restrictions in a retention rule are combined with a logical AND, so that all of the expressions in the retention rule must be true for the document versions to be retained. When you combine separate retention rules, the resulting retention policy is an OR of the combined rules (that is, the document versions are retained if they are matched by any of the rules). Multiple rules do not have an order of operation.

Warning The retention policy specifies what is *retained*, not what is purged. Therefore, anything that does not match the retention policy is removed.

6.11.3 Creating Retention Rules

You create a retention rule by calling the `dls:retention-rule` function. The `dls:retention-rule-insert` function inserts one or more retention rules into the database.

For example, the following retention rule retains all versions of all documents because the empty `cts:and-query` function matches all documents:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert(
dls:retention-rule(
    "All Versions Retention Rule",
    "Retain all versions of all documents",
    (),
    (),
    "Locate all of the documents",
    cts:and-query(()) ) )
```

The following retention rule retains the last five versions of all of the documents located under the `/engineering/` directory:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert(
dls:retention-rule(
    "Engineering Retention Rule",
    "Retain the five most recent versions of Engineering docs",
    5,
    (),
    "Locate all of the Engineering documents",
    cts:directory-query("/engineering/", "infinity") ) )
```

The following retention rule retains the latest three versions of the engineering documents with "Project Alpha" in the title that were authored by Jim:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert(
dls:retention-rule(
    "Project Alpha Retention Rule",
    "Retain the three most recent engineering documents with
     the title 'Project Alpha' and authored by Jim.",
    3,
    (),
    "Locate the engineering docs with 'Project Alpha' in the
     title authored by Jim",
    cts:and-query((
       cts:element-word-query(xs:QName("TITLE"), "Project Alpha"),
       cts:directory-query("/engineering/", "infinity"),
       dls:author-query(xdmp:user("Jim")) )) ) )
```

The following retention rule retains the five most recent versions of documents in the "specs" collection that are no more than thirty days old:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert(
dls:retention-rule(
    "Specs Retention Rule",
    "Keep the five most recent versions of documents in the 'specs'
    collection that are 30 days old or newer",
    5,
    xs:duration("P30D"),
    "Locate documents in the 'specs' collection",
    cts:collection-query("http://marklogic.com/documents/specs") ) )
```

6.11.4 Retaining Specific Versions of Documents

The `dls:document-version-query` and `dls:as-of-query` constructor functions can be used in a retention rule to retain *snapshots* of the documents as they were at some point in time. A snapshot may be of specific versions of documents or documents as of a specific date.

For example, the following retention rule retains the latest versions of the engineering documents created before 5:00pm on 4/23/09:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert(
dls:retention-rule(
    "Draft 1 of the Engineering Docs",
    "Retain each engineering document that was update before
     5:00pm, 4/23/09",
    (),
    (),
    (),
    cts:and-query((
       cts:directory-query("/documentation/", "infinity"),
       dls:as-of-query(xs:dateTime("2009-04-23T17:00:00-07:00")) )) ))
```

If you want to retain two separate snapshots of the engineering documents, you can add a retention rule that contains a different `cts:or-query` function. For example:

```
cts:and-query((
    cts:directory-query("/documentation/", "infinity"),
    dls:as-of-query(xs:dateTime("2009-25-12T09:00:01-07:00")) ))
```

6.11.5 Multiple Retention Rules

In some organizations, it might make sense to create multiple retention rules. For example, the Engineering and Documentation groups may share a database and each organization wants to create and maintain their own retention rule.

Consider the two rules shown below. The first rule retains the latest 5 versions of all of the documents under the `/engineering/` directory. The second rule, retains that latest 10 versions of all of the documents under the `/documentation/` directory. The ORed result of these two rules does not impact the intent of each individual rule and each rule can be updated independently from the other.

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert((
dls:retention-rule(
    "Engineering Retention Rule",
    "Retain the five most recent versions of Engineering docs",
    5,
    (),
    "Apply to all of the Engineering documents",
    cts:directory-query("/engineering/", "infinity") ),

dls:retention-rule(
    "Documentation Retention Rule",
    "Retain the ten most recent versions of the documentation",
    10,
    (),
    "Apply to all of the documentation",
    cts:directory-query("/documentation/", "infinity") ) ))
```

As previously described, multiple retention rules define a logical OR between them, so there may be circumstances when multiple retention rules are needed to define the desired retention policy for the same set of documents.

For example, you want to retain the last five versions of all of the engineering documents, as well as all engineering documents that were updated before 8:00am on 4/24/09 and 9:00am on 5/12/09. The following two retention rules are needed to define the desired retention policy:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert((
dls:retention-rule(
    "Engineering Retention Rule",
    "Retain the five most recent versions of Engineering docs",
    5,
    (),
    "Retain all of the Engineering documents",
    cts:directory-query("/engineering/", "infinity") ),

dls:retention-rule(
    "Project Alpha Retention Rule",
    "Retain the engineering documents that were updated before
     the review dates below.",
    (),
    (),
    "Retain all of the Engineering documents updated before
     the two dates",
    cts:and-query((
      cts:directory-query("/engineering/", "infinity"),
      cts:or-query((
        dls:as-of-query(xs:dateTime("2009-04-24T08:00:17.566-07:00")),
        dls:as-of-query(xs:dateTime("2009-05-12T09:00:01.632-07:00"))
        ))
     )) ) ))
```

It is important to understand the difference between the logical OR combination of the above two retention rules and the logical AND within a single rule. For example, the OR combination of the above two retention rules is not same as the single rule below, which is an AND between retaining the last five versions and the as-of versions. The end result of this rule is that the last five versions are not retained and the as-of versions are only retained as long as they are among the last five versions. Once the revisions of the last five documents have moved past the as-of dates, the AND logic is no longer true and you no longer have an effective retention policy, so no versions of the documents are retained.

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-insert(
dls:retention-rule(
    "Project Alpha Retention Rule",
    "Retain the 5 most recent engineering documents",
    5,
    (),
    "Retain all of the Engineering documents updated before
     the two dates",
    cts:and-query((
      cts:directory-query("/engineering/", "infinity"),
      cts:or-query((
        dls:as-of-query(xs:dateTime("2009-04-24T08:56:17.566-07:00")),
        dls:as-of-query(xs:dateTime("2009-05-12T08:59:01.632-07:00"))
       )) )) ) )
```

6.11.6 Deleting Retention Rules

You can use the `dls:retention-rule-remove` function to delete retention rules. For example, to delete the “Project Alpha Retention Rule,” use:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-remove("Project Alpha Retention Rule")
```

To delete all of your retention rules in the database, use:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

dls:retention-rule-remove(fn:data(dls:retention-rules("*")//dls:name))
```

6.12 Managing Modular Documents in Library Services

As described in “Reusing Content With Modular Document Applications” on page 181, you can create modular documents from the content stored in one or more linked documents. This section describes:

- Creating Managed Modular Documents
- Expanding Managed Modular Documents
- Managing Versions of Modular Documents

6.12.1 Creating Managed Modular Documents

As described in “Reusing Content With Modular Document Applications” on page 181, you can create modular documents from the content stored in one or more linked documents. The `dls:document-extract-part` function provides a shorthand method for creating modular managed documents. This function extracts a child element from a managed document, places the child element in a new managed document, and replaces the extracted child element with an XInclude reference.

For example, the following function call extracts Chapter 1 from the “Project Beta Overview” document:

```
dls:document-extract-part("/engineering/beta_overview_chap1.xml",
    fn:doc("/engineering/beta_overview.xml")//CHAPTER[1],
    "Extracting Chapter 1",
    fn:true() )
```

The contents of `/engineering/beta_overview.xml` is now as follows:

```
<BOOK>
  <TITLE>Project Beta Overview</TITLE>
  <xi:include href="/engineering/beta_overview_chap1.xml"/>
</BOOK>
```

The contents of `/engineering/beta_overview_chap1.xml` is as follows:

```
<CHAPTER>
  <TITLE>Objectives</TITLE>
  <PARA>
   The objective of Project Beta, in simple terms, is to corner
   the widget market.
  </PARA>
</CHAPTER>
```

Note: The newly created managed document containing the extracted child element is initially checked-in and must be checked out before you can make any updates.

The `dls:document-extract-part` function can only be called once in a transaction for the same document. There may be circumstances in which you want to extract multiple elements from a document and replace them with XInclude statements. For example, the following query creates separate documents for all of the chapters from the "Project Beta Overview" document and replaces them with XInclude statements:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

declare namespace xi="http://www.w3.org/2001/XInclude";

let $includes := for $chap at $num in
   doc("/engineering/beta_overview.xml")/BOOK/CHAPTER

return (
   dls:document-insert-and-manage(
     fn:concat("/engineering/beta_overview_chap", $num, ".xml"),
     fn:true(),
     $chap),

   <xi:include href="/engineering/beta_overview_chap{$num}.xml"
    xmlns:xi="http://www.w3.org/2001/XInclude"/>
  )

let $contents :=
   <BOOK>
      <TITLE>Project Beta Overview</TITLE>
      {$includes}
   </BOOK>

return
   dls:document-update(
      "/engineering/beta_overview.xml",
      $contents,
      "Chapters are XIncludes",
      fn:true() )
```

This query produces a "Project Beta Overview" document similar to the following:

```
<BOOK>
   <TITLE>Project Beta Overview</TITLE>
   <xi:include href="/engineering/beta_overview_chap1.xml"
    xmlns:xi="http://www.w3.org/2001/XInclude"/>
   <xi:include href="/engineering/beta_overview_chap1.xml"
    xmlns:xi="http://www.w3.org/2001/XInclude"/>
   <xi:include href="/engineering/beta_overview_chap2.xml"
    xmlns:xi="http://www.w3.org/2001/XInclude"/>
</BOOK>
```

6.12.2 Expanding Managed Modular Documents

Modular documents can be "expanded" so that you can view the entire node, complete with its linked nodes, or a specific linked node. You can expand a modular document using `dls:node-expand`, or a linked node in a modular document using `dls:link-expand` .

> **Note:** When using the `dls:node-expand` function to expand documents that contain XInclude links to specific versioned documents, specify the `$restriction` parameter as an empty sequence.

For example, to return the expanded `beta_overview.xml` document, you can use:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

let $node := fn:doc("/engineering/beta_overview.xml")

return dls:node-expand($node, ())
```

To return the first linked node in the `beta_overview.xml` document, you can use:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

declare namespace xi="http://www.w3.org/2001/XInclude";

let $node := fn:doc("/engineering/beta_overview.xml")

return dls:link-expand(
               $node,
               $node/BOOK/xi:include[1],
               () )
```

The `dls:node-expand` and `dls:link-expand` functions allow you to specify a `cts:query` constructor to restrict what document version is to be expanded. For example, to expand the most recent version of the "Project Beta Overview" document created before 1:30pm on 4/6/09, you can use:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

let $node := fn:doc("/engineering/beta_overview.xml")

return dls:node-expand(
         $node,
         dls:as-of-query(
            xs:dateTime("2009-04-06T13:30:33.576-07:00")) )
```

6.12.3 Managing Versions of Modular Documents

Library Services can manage modular documents so that various versions can be created for the linked documents. As a modular document's linked documents are updated, you might want to take periodic snapshots of the entire node.

For example, as shown in "Creating Managed Modular Documents" on page 111, the "Project Beta Overview" document contains three chapters that are linked as separate documents. The following query takes a snapshot of the latest version of each chapter and creates a new version of the "Project Beta Overview" document that includes the versioned chapters:

```
xquery version "1.0-ml";
import module namespace dls="http://marklogic.com/xdmp/dls"
              at "/MarkLogic/dls.xqy";

declare namespace xi="http://www.w3.org/2001/XInclude";

(: For each chapter in the document, get the URI :)
let $includes :=
  for $chap at $num in doc("/engineering/beta_overview.xml")
//xi:include/@href

(: Get the latest version of each chapter :)
let $version_number :=
  fn:data(dls:document-history($chap)//dls:version-id)[last()]

let $version :=  dls:document-version-uri($chap, $version_number)

(: Create an XInclude statement for each versioned chapter :)
  return
    <xi:include href="{$version}"/>

(: Update the book with the versioned chapters :)
let $contents :=
  <BOOK>
    <TITLE>Project Beta Overview</TITLE>
    {$includes}
  </BOOK>

return
  dls:document-update(
    "/engineering/beta_overview.xml",
    $contents,
    "Latest Draft",
    fn:true() )
```

The above query results in a new version of the “Project Beta Overview” document that looks like:

```
<BOOK>
  <TITLE>Project Beta Overview</TITLE>
  <xi:include
href="/engineering/beta_overview_chap1.xml_versions/4-beta_overview_
chap1.xml" xmlns:xi="http://www.w3.org/2001/XInclude"/>
  <xi:include
href="/engineering/beta_overview_chap2.xml_versions/3-beta_overview_
chap2.xml" xmlns:xi="http://www.w3.org/2001/XInclude"/>
  <xi:include
href="/engineering/beta_overview_chap3.xml_versions/3-beta_overview_
chap3.xml" xmlns:xi="http://www.w3.org/2001/XInclude"/>
</BOOK>
```

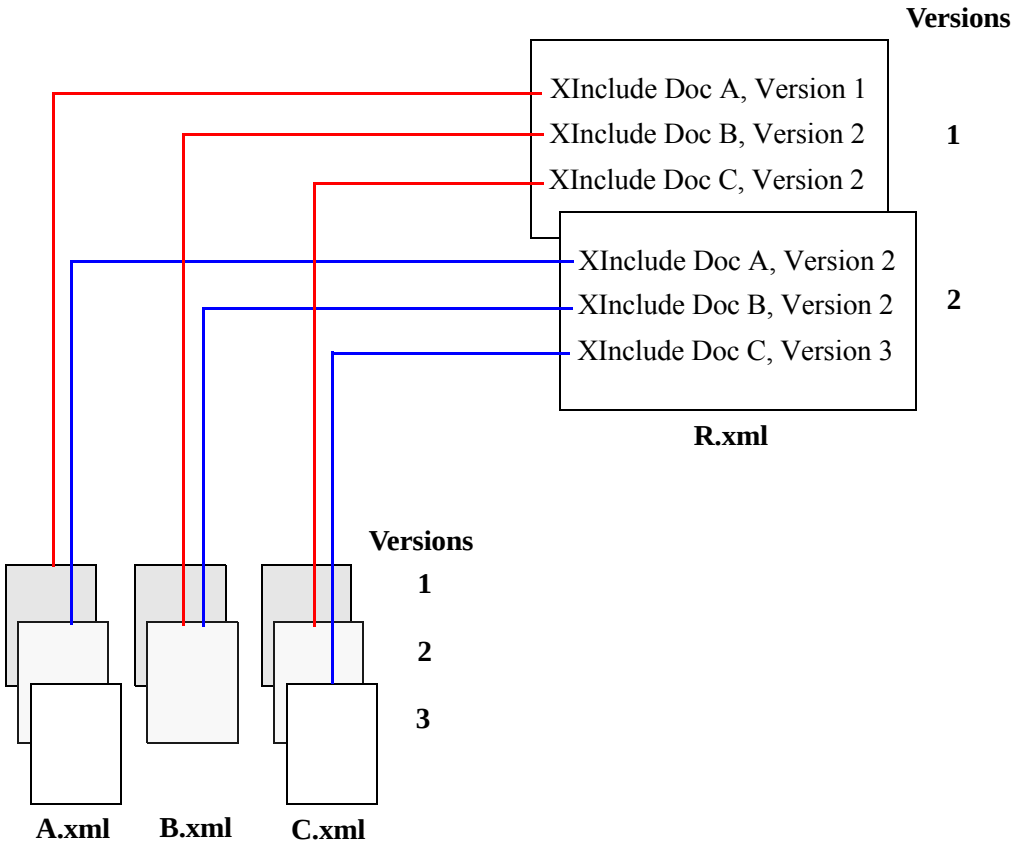
Note: When using the `dls:node-expand` function to expand modular documents that contain XInclude links to specific versioned documents, specify the `$restriction` parameter as an empty sequence.

You can also create modular documents that contain different versions of linked documents. For example, in the illustration below, Doc R.xml, Version 1 contains the contents of:

- Doc A.xml, Version 1
- Doc B.xml, Version 2
- Doc C.xml, Version 2

While Doc X, Version 2 contains the contents of:

- Doc A.xml, Version 2
- Doc B.xml, Version 2
- Doc C.xml, Version 3



7.0 Transforming XML Structures With a Recursive typeswitch Expression

A common task required with XML is to transform one structure to another structure. This chapter describes a design pattern using the XQuery `typeswitch` expression which makes it easy to perform complex XML transformations with good performance, and includes some samples illustrating this design pattern. It includes the following sections:

- XML Transformations
- Sample XQuery Transformation Code

7.1 XML Transformations

Programmers are often faced with the task of converting one XML structure to another. These transformations can range from very simple element name change transformations to extremely complex transformations that reshape the XML structure and/or combine it with content from other documents or sources. This section describes some aspects of XML transformations and includes the following sections:

- XQuery vs. XSLT
- Transforming to XHTML or XSL-FO
- The typeswitch Expression

7.1.1 XQuery vs. XSLT

XSLT is commonly used in transformations, and it works well for many transformations. It does have some drawbacks for certain types of transformations, however, especially if the transformations are part of a larger XQuery application.

XQuery is a powerful programming language, and MarkLogic Server provides very fast access to content, so together they work extremely well for transformations. MarkLogic Server is particularly well suited to transformations that require searches to get the content which needs transforming. For example, you might have a transformation that uses a lexicon lookup to get a value with which to replace the original XML value. Another transformation might need to count the number of authors in a particular collection.

7.1.2 Transforming to XHTML or XSL-FO

A common XML transformation is converting documents from some proprietary XML structure to HTML. Since XQuery produces XML, it is fairly easy to write an XQuery program that returns XHTML, which is the XML version of HTML. XHTML is, for the most part, just well-formed HTML with lowercase tag and attribute names. So it is common to write XQuery programs that return XHTML.

Similarly, you can write an XQuery program that returns XSL-FO, which is a common path to build PDF output. Again, XSL-FO is just an XML structure, so it is easy to write XQuery that returns XML in that structure.

7.1.3 The typeswitch Expression

There are other ways to perform transformations in XQuery, but the `typeswitch` expression used in a recursive function is a design pattern that is convenient, performs well, and makes it very easy to change and maintain the transformation code.

For the syntax of the `typeswitch` expression, see The typeswitch Expression in *XQuery and XSLT Reference Guide*. The `case` clause allows you to perform a test on the input to the `typeswitch` and then return something. For transformations, the tests are often what are called *kind tests*. A kind test tests to see what kind of node something is (for example, an element node with a given QName). If that test returns true, then the code in the `return` clause is executed. The `return` clause can be arbitrary XQuery, and can therefore call a function.

Because XML is an ordered tree structure, you can create a function that recursively walks through an XML node, each time doing some transformation on the node and sending its child nodes back into the function. The result is a convenient mechanism to transform the structure and/or content of an XML node.

7.2 Sample XQuery Transformation Code

This section provides some code examples that use the `typeswitch` expression. For each of these samples, you can cut and paste the code to execute against an App Server. For a more complicated example of this technique, see the Shakespeare Demo Application on developer.marklogic.com/code.

The following samples are included:

- Simple Example
- Simple Example With cts:highlight
- Sample Transformation to XHTML
- Extending the typeswitch Design Pattern

7.2.1 Simple Example

The following sample code does a trivial transformation of the input node, but it shows the basic design pattern where the `default` clause of the `typeswitch` expression calls a simple function which sends the child nodes back into the original function.

```
xquery version "1.0-ml";

(: This is the recursive typeswitch function :)
declare function local:transform($nodes as node()*) as node()*
{
for $n in $nodes return
typeswitch ($n)
  case text() return $n
  case element (bar) return <barr>{local:transform($n/node())}</barr>
  case element (baz) return <bazz>{local:transform($n/node())}</bazz>
  case element (buzz) return
     <buzzz>{local:transform($n/node())}</buzzz>
  case element (foo) return <fooo>{local:transform($n/node())}</fooo>
  default return <temp>{local:transform($n/node())}</temp>
};

let $x :=
<foo>foo
  <bar>bar</bar>
  <baz>baz
    <buzz>buzz</buzz>
  </baz>
  foo
</foo>
return
local:transform($x)
```

This XQuery program returns the following:

```
<fooo>
  foo
  <barr>bar</barr>
  <bazz>baz
    <buzzz>buzz</buzzz>
  </bazz>
  foo
</fooo>
```

7.2.2 Simple Example With cts:highlight

The following sample code is the same as the previous example, except it also runs `cts:highlight` on the result of the transformation. Using `cts:highlight` in this way is sometimes useful when displaying the results from a search and then highlighting the terms that match the `cts:query` expression. For details on `cts:highlight`, see Highlighting Search Term Matches in the *Search Developer's Guide*.

```
xquery version "1.0-ml";

(: This is the recursive typeswitch function :)
declare function local:transform($nodes as node()*) as node()*
{
for $n in $nodes return
typeswitch ($n)
  case text() return $n
  case element (bar) return <barr>{local:transform($n/node())}</barr>
  case element (baz) return <bazz>{local:transform($n/node())}</bazz>
  case element (buzz) return
     <buzzz>{local:transform($n/node())}</buzzz>
  case element (foo) return <fooo>{local:transform($n/node())}</fooo>
  default return <booo>{local:transform($n/node())}</booo>
};

let $x :=
<foo>foo
  <bar>bar</bar>
  <baz>baz
    <buzz>buzz</buzz>
  </baz>
  foo
</foo>
return
cts:highlight(local:transform($x), cts:word-query("foo"),
   <b>{$cts:text}</b>)
```

This XQuery program returns the following:

```
<fooo>
  <b>foo</b>
  <barr>bar</barr>
  <bazz>baz
    <buzzz>buzz</buzzz>
  </bazz>
  <b>foo</b>
</fooo>
```

7.2.3 Sample Transformation to XHTML

The following sample code performs a very simple transformation of an XML structure to XHTML. It uses the same design pattern as the previous example, but this time the XQuery code includes HTML markup.

```
xquery version "1.0-ml";
declare default element namespace "http://www.w3.org/1999/xhtml";

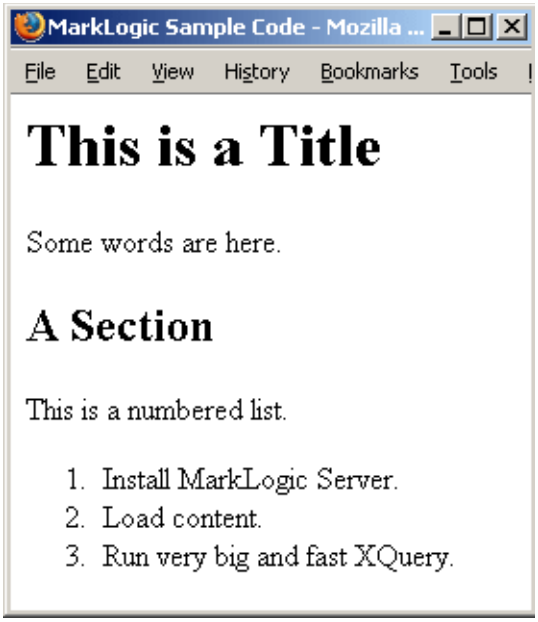
(: This is the recursive typeswitch function :)
declare function local:transform($nodes as node()*) as node()*
{
for $n in $nodes return
typeswitch ($n)
  case text() return $n
  case element (a) return local:transform($n/node())
  case element (title) return <h1>{local:transform($n/node())}</h1>
  case element (para) return <p>{local:transform($n/node())}</p>
  case element (sectionTitle) return
      <h2>{local:transform($n/node())}</h2>
  case element (numbered) return <ol>{local:transform($n/node())}</ol>
  case element (number) return <li>{local:transform($n/node())}</li>
  default return <tempnode>{local:transform($n/node())}</tempnode>
};

let $x :=
<a>
 <title>This is a Title</title>
 <para>Some words are here.</para>
 <sectionTitle>A Section</sectionTitle>
 <para>This is a numbered list.</para>
 <numbered>
   <number>Install MarkLogic Server.</number>
   <number>Load content.</number>
   <number>Run very big and fast XQuery.</number>
 </numbered>
</a>
return
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>MarkLogic Sample Code</title></head>
<body>{local:transform($x)}</body>
</html>
```

This returns the following XHTML code:

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>MarkLogic Sample Code</title>
  </head>
<body>
<h1>This is a Title</h1>
<p>Some words are here.</p>
<h2>A Section</h2>
<p>This is a numbered list.</p>
  <ol>
      <li>Install MarkLogic Server.</li>
      <li>Load content.</li>
      <li>Run very big and fast XQuery.</li>
  </ol>
</body>
</html>
```

If you run this code against an HTTP App Server (for example, copy the code to a file in the App Server root and access the page from a browser), you will see results similar to the following:



Note that the `return` clauses of the `typeswitch` `case` statements in this example are simplified, and look like the following:

```
case element (sectionTitle) return <h2>{local:passthru($x)}</h2>
```

In a more typical example, the `return` clause would call a function:

```
case element (sectionTitle) return local:myFunction($x)
```

The function can then perform arbitrarily complex logic. Typically, each case statement calls a function with code appropriate to how that element needs to be transformed.

7.2.4 Extending the typeswitch Design Pattern

There are many ways you can extend this design pattern beyond the simple examples above. For example, you can add a second parameter to the simple `transform` functions shown in the previous examples. The second parameter passes some other information about the node you are transforming.

Suppose you want your transformation to exclude certain elements based on the place in the XML hierarchy in which the elements appear. You can then add logic to the function to exclude the passed in elements, as shown in the following code snippet:

```
declare function transform($nodes as node()*, $excluded as element()*)
  as node()*
{
(: Test whether each node in $nodes is an excluded element, if so
   return empty, otherwise run the typeswitch expression.
:)
for $n in $nodes return
if ( some $node in $excluded satisfies $n )
then ( )
else ( typeswitch ($n) ..... )
};
```

There are plenty of other extensions to this design pattern you can use. What you do depends on your application requirements. XQuery is a powerful programming language, and therefore these types of design patterns are very extensible to new requirements.

8.0 Document and Directory Locks

This chapter describes locks on documents and directories, and includes the following sections:

- Overview of Locks
- Lock APIs
- Example: Finding the URI of Documents With Locks
- Example: Setting a Lock on a Document
- Example: Releasing a Lock on a Document
- Example: Finding the User to Whom a Lock Belongs

Note: This chapter is about document and directory locks that you set explicitly, not about transaction locks which MarkLogic sets implicitly. To understand transactions, see "Understanding Transactions in MarkLogic Server" on page 31.

8.1 Overview of Locks

Each document and directory can have a *lock*. A lock is stored as a locks document in a MarkLogic Server database. The locks document is separate from the document or directory to which it is associated. Locks have the following characteristics:

- Write Locks
- Persistent
- Searchable
- Exclusive or Shared
- Hierarchical
- Locks and WebDAV
- Other Uses for Locks

8.1.1 Write Locks

Locks are write locks; they restrict updates from all users who do not have the locks. When a user has an exclusive lock, no other users can get a lock and no other users can update or delete the document. Attempts to update or delete documents that have locks raise an error. Other users can still read documents that have locks, however.

8.1.2 Persistent

Locks are persistent in the database. They are not tied to a transaction. You can set locks to last a specified time period or to last indefinitely. Because they are persistent, you can use locks to ensure that a document is not modified during a multi-transaction operation.

8.1.3 Searchable

Because locks are persistent XML documents, they are therefore searchable XML documents, and you can write queries to give information about locks in the database. For an example, see "Example: Finding the URI of Documents With Locks" on page 127.

8.1.4 Exclusive or Shared

You can set locks as `exclusive`, which means only the user who set the lock can update the associated database object (document, directory, or collection). You can also set locks as `shared`, which means other users can obtain a shared lock on the database object; once a user has a `shared` lock on an object, the user can update it.

8.1.5 Hierarchical

When you are locking a directory, you can specify the depth in a directory hierarchy you want to lock. Specifying `"0"` means only the specified URI is locked, and specifying `"infinity"` means the URI (for example, the directory) and all of its children are locked.

8.1.6 Locks and WebDAV

WebDAV clients use locks to lock documents and directories before updating them. Locking ensures that no other clients will change the document while it is being saved. It is up to the implementation of a WebDAV client as to how it sets locks. Some clients set the locks to expire after a time period and some set them to last until they explicitly unlock the document.

8.1.7 Other Uses for Locks

Any application can use locks as part of its update strategy. For example, you can have a policy that a developer sets a lock for 30 seconds before performing an update to a document or directory. Locks are very flexible, so you can set up a policy that makes sense for your environment, or you can choose not to use them at all.

If you set a lock on every document and directory in the database, that can have the effect of not allowing any data to change in the database (except by the user who owns the lock). Combining a application development practice of locking and using security permissions effectively can provide a robust multi-user development environment.

8.2 Lock APIs

There are basically two kinds of APIs for locks: APIs to show locks and APIs to set/remove locks. For detailed syntax for these APIs, see the online XQuery Built-In and Module Function Reference.

The APIs to show locks are:

- `xdmp:document-locks`
- `xdmp:directory-locks`
- `xdmp:collection-locks`

The `xdmp:document-locks` function with no arguments returns a sequence of locks, one for each document lock. The `xdmp:document-locks` function with a sequence of URIs as an argument returns the locks for the specified document(s). The `xdmp:directory-locks` function returns locks for all of the documents in the specified directory, and the `xdmp:collection-locks` function returns all of the locks for documents in the specified collection.

You can set and remove locks on directories and documents with the following functions:

- `xdmp:lock-acquire`
- `xdmp:lock-release`

The basic procedure to set a lock on a document or a directory is to submit a query using the `xdmp:lock-acquire` function, specifying the URI, the scope of lock requested (`exclusive` or `shared`), the hierarchy affected by the lock (just the URI or the URI and all of its children), the owner of the lock, the duration of the lock

> **Note:** The `owner` of the lock is not the same as the `sec:user-id` of the lock. The `owner` can be specified as an option to `xdmp:lock-acquire`. If owner is not explicitly specified, then the owner defaults to the name of the user who issued the lock command. For an example, see "Example: Finding the User to Whom a Lock Belongs" on page 129.

8.3 Example: Finding the URI of Documents With Locks

If you call the XQuery built-in `xdmp:node-uri` function on a locks document, it returns the URI of the document that is locked. The following query returns a document listing the URIs of all documents in the database that have locks.

```
<root>
{
for $locks in xdmp:document-locks()
return <document-URI>{xdmp:node-uri($locks)}</document-URI>
}
</root>
```

For example, if the only document in the database with a lock has a URI `/document/myDocument.xml`, then the above query would return the following.

```
<root>
  <document-URI>/documents/myDocument.xml</document-URI>
</root>
```

8.4 Example: Setting a Lock on a Document

The following example uses the `xdmp:lock-acquire` function to set a two minute (120 second) lock on a document with the specified URI:

```
xdmp:lock-acquire("/documents/myDocument.xml",
                  "exclusive",
                  "0",
                  "Raymond is editing this document",
                  xs:unsignedLong(120))
```

You can view the resulting lock document with the `xdmp:document-locks` function as follows:

```
xdmp:document-locks("/documents/myDocument.xml")

=>

<lock:lock xmlns:lock="http://marklogic.com/xdmp/lock">
  <lock:lock-type>write</lock:lock-type>
  <lock:lock-scope>exclusive</lock:lock-scope>
  <lock:active-locks>
    <lock:active-lock>
      <lock:depth>0</lock:depth>
      <lock:owner>Raymond is editing this document</lock:owner>
      <lock:timeout>120</lock:timeout>
      <lock:lock-token>
       http://marklogic.com/xdmp/locks/4d0244560cc3726c
      </lock:lock-token>
      <lock:timestamp>1121722103</lock:timestamp>
      <sec:user-id xmlns:sec="http://marklogic.com/xdmp/security">
         8216129598321388485
      </sec:user-id>
    </lock:active-lock>
  </lock:active-locks>
</lock:lock>
```

8.5 Example: Releasing a Lock on a Document

The following example uses the `xdmp:lock-release` function to explicitly release a lock on a document:

```
xdmp:lock-release("/documents/myDocument.xml")
```

If you acquire a lock with no timeout period, be sure to release the lock when you are done with it. If you do not release the lock, no other users can update any documents or directories locked by the `xdmp:lock-acquire` action.

8.6 Example: Finding the User to Whom a Lock Belongs

Because locks are documents, you can write a query that finds the user to whom a lock belongs. For example, the following query searches through the `sec:user-id` elements of the lock documents and returns a set of URI names and user IDs of the user who owns each lock:

```
for $x in xdmp:document-locks()//sec:user-id
return  <lock>
           <URI>{xdmp:node-uri($x)}</URI>
           <user-id>{data($x)}</user-id>
        </lock>
```

A sample result is as follows (this result assumes there is only a single lock in the database):

```
<lock>
  <URI>/documents/myDocument.xml</URI>
  <user-id>15025067637711025979</user-id>
</lock>
```

9.0 Properties Documents and Directories

This chapter describes properties documents and directories in MarkLogic Server. It includes the following sections:

- Properties Documents
- Using Properties for Document Processing
- Directories
- Permissions On Properties and Directories
- Example: Directory and Document Browser

9.1 Properties Documents

A *properties document* is an XML document that shares the same URI with a document in a database. Every document can have a corresponding properties document, although the properties document is only created if properties are created. The properties document is typically used to store metadata related to its corresponding document, although you can store any XML data in a properties document, as long as it conforms to the properties document schema. A document typically exists at a given URI in order to create a properties document, although it is possible to create a document and add properties to it in a single transaction, and it is also possible to create a property where no document exists. The properties document is stored in a separate fragment to its corresponding document. This section describes properties documents and the APIs for accessing them, and includes the following subsections:

- Properties Document Namespace and Schema
- APIs on Properties Documents
- XPath property Axis
- Protected Properties
- Creating Element Indexes on a Properties Document Element
- Sample Properties Documents
- Standalone Properties Documents

9.1.1 Properties Document Namespace and Schema

Properties documents are XML documents that must conform to the `properties.xsd` schema. The `properties.xsd` schema is copied to the `<install_dir>/Config` directory at installation time.

The properties schema is assigned the `prop` namespace prefix, which is predefined in the server:

```
http://marklogic.com/xdmp/property
```

The following listing shows the `properties.xsd` schema:

```
<xs:schema targetNamespace="http://marklogic.com/xdmp/property"
 xsi:schemaLocation="http://www.w3.org/2001/XMLSchema XMLSchema.xsd
                     http://marklogic.com/xdmp/security security.xsd"
 xmlns="http://marklogic.com/xdmp/property"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xhtml="http://www.w3.org/1999/xhtml"
 xmlns:sec="http://marklogic.com/xdmp/security">

  <xs:complexType name="properties">
    <xs:annotation>
      <xs:documentation>
        A set of document properties.
      </xs:documentation>
      <xs:appinfo>
      </xs:appinfo>
    </xs:annotation>
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:any/>
    </xs:choice>
  </xs:complexType>

  <xs:element name="properties" type="properties">
    <xs:annotation>
      <xs:documentation>
        The container for properties.
      </xs:documentation>
      <xs:appinfo>
      </xs:appinfo>
    </xs:annotation>
  </xs:element>

  <xs:simpleType name="directory">
    <xs:annotation>
      <xs:documentation>
        A directory indicator.
      </xs:documentation>
      <xs:appinfo>
      </xs:appinfo>
    </xs:annotation>
    <xs:restriction base="xs:anySimpleType">
    </xs:restriction>
  </xs:simpleType>

  <xs:element name="directory" type="directory">
    <xs:annotation>
      <xs:documentation>
        The indicator for a directory.
      </xs:documentation>
      <xs:appinfo>
      </xs:appinfo>
    </xs:annotation>
  </xs:element>
```

```
  <xs:element name="last-modified" type="last-modified">
    <xs:annotation>
      <xs:documentation>
        The timestamp of last document modification.
      </xs:documentation>
      <xs:appinfo>
      </xs:appinfo>
    </xs:annotation>
  </xs:element>

  <xs:simpleType name="last-modified">
    <xs:annotation>
      <xs:documentation>
        A timestamp of the last time something was modified.
      </xs:documentation>
      <xs:appinfo>
      </xs:appinfo>
    </xs:annotation>
    <xs:restriction base="xs:dateTime">
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

9.1.2 APIs on Properties Documents

The APIs for properties documents are XQuery functions which allow you to list, add, and set properties in a properties document. The properties APIs provide access to the top-level elements in properties documents. Because the properties are XML elements, you can use XPath to navigate to any children or descendants of the top-level property elements. The properties document is tied to its corresponding document and shares its URI; when you delete a document, its properties document is also deleted.

The following APIs are available to access and manipulate properties documents:

- `xdmp:document-properties`
- `xdmp:document-add-properties`
- `xdmp:document-set-properties`
- `xdmp:document-set-property`
- `xdmp:document-remove-properties`
- `xdmp:document-get-properties`
- `xdmp:collection-properties`
- `xdmp:directory`
- `xdmp:directory-properties`

For the signatures and descriptions of these APIs, see the *MarkLogic XQuery and XSLT Function Reference*.

9.1.3 XPath property Axis

MarkLogic has extended XPath (available in both XQuery and XSLT) to include the *property axis*. The property axis (`property::`) allows you to write an XPath expression to search through items in the properties document for a given URI. These expression allow you to perform joins across the document and property axes, which is useful when storing state information for a document in a property. For details on this approach, see "Using Properties for Document Processing" on page 136.

The property axis is similar to the forward and reverse axes in an XPath expression. For example, you can use the `child::` forward axis to traverse to a child element in a document. For details on the XPath axes, see the XPath 2.0 specification and XPath Quick Reference in the *XQuery and XSLT Reference Guide*.

The property axis contains all of the children of the properties document node for a given URI.

The following example shows how you can use the property axis to access properties for a document while querying the document:

Create a test document as follows:

```
xdmp:document-insert("/test/123.xml",
  <test>
    <element>123</element>
  </test>)
```

Add a property to the properties document for the `/test/123.xml` document:

```
xdmp:document-add-properties("/test/123.xml",
  <hello>hello there</hello>)
```

If you list the properties for the `/test/123.xml` document, you will see the property you just added:

```
xdmp:document-properties("/test/123.xml")
=>
<prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
  <hello>hello there</hello>
</prop:properties>
```

You can now search through the property axis of the `/test/123.xml` document, as follows:

```
doc("/test/123.xml")/property::hello
=>
<hello>hello there</hello>
```

9.1.4 Protected Properties

The following properties are protected, and they can only be created or modified by the system:

- `prop:directory`
- `prop:last-modified`

These properties are reserved for use directly by MarkLogic Server; attempts to add or delete properties with these names fail with an exception.

9.1.5 Creating Element Indexes on a Properties Document Element

Because properties documents are XML documents, you can create element (range) indexes on elements within a properties document. If you use properties to store numeric or date metadata about the document to which the properties document corresponds, for example, you can create an element index to speed up queries that access the metadata.

9.1.6 Sample Properties Documents

Properties documents are XML documents that conform to the schema described in "Properties Document Namespace and Schema" on page 131. You can list the contents of a properties document with the `xdmp:document-properties` function. If there is no properties document at the specified URI, the function returns the empty sequence. A properties document for a directory has a single empty `prop:directory` element. For example, if there exists a directory at the URI `http://myDirectory/`, the `xdmp:document-properties` command returns a properties document as follows:

```
xdmp:document-properties("http://myDirectory/")
=>
<prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
  <prop:directory/>
</prop:properties>
```

You can add whatever you want to a properties document (as long as it conforms to the properties schema). If you run the function `xdmp:document-properties` with no arguments, it returns a sequence of all the properties documents in the database.

9.1.7 Standalone Properties Documents

Typically, properties documents are created alongside the corresponding document that shares its URI. It is possible, however, to create a properties document at a URI with no coresponding document at that URI. Such a properties document is known as a *standalone properties document*. To create a standalone properties document, use the `xdmp:document-add-properties` or `xdmp:document-set-properties` APIs, and optionally add the `xdmp:document-set-permissions`, `xdmp:document-set-collections`, and/or `xdmp:document-set-quality` APIs to set the permissions, collections, and/or quality on the properties document.

The following example creates a properties document and sets permissions on it:

```
xquery version "1.0-ml";

xdmp:document-set-properties("/my-props.xml", <my-props/>),
xdmp:document-set-permissions("/my-props.xml",
   (xdmp:permission("dls-user", "read"),
    xdmp:permission("dls-user", "update")))
```

If you then run `xdmp:document-properties` on the URI, it returns the new properties document:

```
xquery version "1.0-ml";

xdmp:document-properties("/my-props.xml")
(: returns:
<?xml version="1.0" encoding="ASCII"?>
<prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
  <my-props/>
  <prop:last-modified>2010-06-18T18:19:10-07:00</prop:last-modified>
</prop:properties>
:)
```

Similarly, you can pass in functions to set the collections and quality on the standalone properties document, either when you create it or after it is created.

9.2 Using Properties for Document Processing

When you need to update large numbers of documents, sometimes in multi-step processes, you often need to keep track of the current state of each document. For example, if you have a content processing application that updates millions of documents in three steps, you need to have a way of programatically determining which documents have not been processed at all, which have completed step 1, which have completed step 2, and so on.

This section describes how to use properties to store metadata for use in a document processing pipeline, it includes the following subsections:

- Using the property Axis to Determine Document State
- Document Processing Problem
- Solution for Document Processing
- Basic Commands for Running Modules

9.2.1 Using the property Axis to Determine Document State

You can use properties documents to store state information about documents that undergo multi-step processing. Joining across properties documents can then determine which documents have been processed and which have not. The queries that perform these joins use the `property::` axis (for details, see “XPath property Axis” on page 134).

Joins across the properties axis that have predicates are optimized for performance. For example, the following returns `foo` root elements from documents that have a property `bar`:

```
foo[property::bar]
```

The following examples show the types of queries that are optimized for performance (where `/a/b/c` is some XPath expression):

- Property axis in predicates:

  ```
  /a/b/c[property::bar]
  ```

- Negation tests on property axis:

  ```
  /a/b/c[not(property::bar = "baz")]
  ```

- Continuing path expression after the `property` predicate:

  ```
  /a/b/c[property::bar and bob = 5]/d/e
  ```

- Equivalent FLWOR expressions:

  ```
  for $f in /a/b/c
  where $f/property::bar = "baz"
  return $f
  ```

Other types of expressions will work but are not optimized for performance, including the following:

- If you want the bar property of documents whose root elements are `foo`:

  ```
  /foo/property::bar
  ```

9.2.2 Document Processing Problem

The approach outlined in this section works well for situations such as the following:

- "I have already loaded 1 million documents and now want to update all of them." The psuedo-code for this is as follows:

  ```
  for $d in fn:doc()
  return some-update($d)
  ```

  These types of queries will eventually run out of tree cache memory and fail.

- When iterative calls of the following form become progressively slow:

  ```
  for $d in fn:doc()[k to k+10000]
  return some-update($d)
  ```

For these types of scenarios, using properties to test whether a document needs processing is an effective way of being able to batch up the updates into manageable chunks.

9.2.3 Solution for Document Processing

This content processing technique works in a wide variety of situations This approach satisfies the following requirements:

- Works with large existing datasets.
- Does not require you to know before you load the datasets that you are going to need to further processing to them later.
- This approach works in a situations in which data is still arriving (for example, new data is added every day).
- Needs to be able to ultimately transition into a steady state "content processing" enabled environment.

The following are the basic steps of the document processing approach:

1. Take an iterative strategy, but one that does not become progressively slow.

2. Split the reprocessing activity into multiple updates.

3. Use properties (or lack thereof) to identify the documents that (still) need processing.

4. Repeatedly call the same module, updating its property as well as updating the document:

```
for $p in fn:doc()/root[not(property::some-update)][1 to 10000]
return some-update($d)
```

5. If there are any documents that still need processing, invoke the module again.

6. The psuedo-code for the module that processes documents that do not have a specific property is as follows:

```
let $docs := get n documents that have no properties
return
for $processDoc in $docs
return if (empty $processDoc)
       then ()
       else ( process-document($processDoc),
              update-property($processDoc) )
,
xdmp:spawn(process_module)
```

This psuedo-code does the following:

- gets the URIs of documents that do not have a specific property
- for each URI, check if the specific property exists
- if the property exists, do nothing to that document (it has already been updated)
- if the property does not exist, do the update to the document and the update to the property
- continue this for all of the URIs
- when all of the URIs have been processed, call the module again to get any new documents (ones with no properties)

7. (Optional) Automate the process by setting up a Content Processing Pipeline.

9.2.4 Basic Commands for Running Modules

The following built-in functions are needed to perform automated content processing:

- To put a module on Task Server Queue:

```
xdmp:spawn($database, $root, $path)
```

- To evaluate an entire module (similar to `xdmp:eval`, but for modules):

```
xdmp:invoke($path, $external-vars)

xdmp:invoke-in($path, $database-id, $external-vars)
```

9.3 Directories

Directories have many uses, including organizing your document URIs and using them with WebDAV servers. This section includes the following items about directories:

- Properties and Directories
- Directories and WebDAV Servers
- Directories Versus Collections

9.3.1 Properties and Directories

When you create a directory, MarkLogic Server creates a properties document with a `prop:directory` element. If you run the `xdmp:document-properties` command on the URI corresponding to a directory, the command returns a properties document with an empty `prop:directory` element, as shown in the following example:

```
xdmp:directory-create("/myDirectory/");

xdmp:document-properties("/myDirectory/")
=>
<prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
  <prop:directory/>
</prop:properties>
```

Note: You can create a directory with any unique URI, but the convention is for directory URIs to end with a forward slash (`/`). It is possible to create a document with the same URI as a directory, but this is not recommended; the best practice is to reserve URIs ending in slashes for directories.

Because `xdmp:document-properties` with no arguments returns the properties documents for all properties documents in the database, and because each directory has a `prop:directory` element, you can easily write a query that returns all of the directories in the database. Use the `xdmp:node-uri` function to accomplish this as follows:

```
xquery version "1.0-ml";

for $x in xdmp:document-properties()/prop:properties/prop:directory
return <directory-uri>{xdmp:node-uri($x)}</directory-uri>
```

9.3.2 Directories and WebDAV Servers

Directories are needed for use in WebDAV servers. To create a document that can be accessed from a WebDAV client, the parent directory must exist. The parent directory of a document is the directory in which the URI is the prefix of the document (for example, the directory of the URI `http://myserver/doc.xml` is `http://myserver/`). When using a database with a WebDAV server, ensure that the `directory creation` setting on the database configuration is set to `automatic` (this is the default setting), which causes parent directories to be created when documents are created. For information on using directories in WebDAV servers, see WebDAV Servers in the *Administrator's Guide*.

9.3.3 Directories Versus Collections

You can use both directories and collections to organize documents in a database. The following are important differences between directories and collections:

- Directories are hierarchical in structure (like a filesystem directory structure). Collections do not have this requirement. Because directories are hierarchical, a directory URI must contain any parent directories. Collection URIs do not need to have any relation to documents that belong to a collection. For example, a directory named `http://marklogic.com/a/b/c/d/e/` (where `http://marklogic.com/` is the root) requires the existence of the parent directories `d`, `c`, `b`, and `a`. With collections, any document (regardless of its URI) can belong to a collection with the given URI.
- Directories are required for WebDAV clients to see documents. In other words, to see a document with URI `/a/b/hello/goodbye` in a WebDAV server with `/a/b/` as the root, directories with the following URIs must exist in the database:

```
/a/b/

/a/b/hello/
```

Except for the fact that you can use both directories and collections to organize documents, directories are unrelated to collections. For details on collections, see Collections in the *Search Developer's Guide*. For details on WebDAV servers, see WebDAV Servers in the *Administrator's Guide*.

9.4 Permissions On Properties and Directories

Like any document in a MarkLogic Server database, a properties document can have permissions. Since a directory has a properties document (with an empty `prop:directory` element), directories can also have permissions. Permissions on properties documents are the same as the permissions on their corresponding documents, and you can list the permissions with the `xdmp:document-get-permissions` function. Similarly, you can list the permissions on a directory with the `xdmp:document-get-permissions` function. For details on permissions and on security, see *Security Guide*.

9.5 Example: Directory and Document Browser

Using properties documents, you can build a simple application that lists the documents and directories under a URI. The following sample code uses the `xdmp:directory` function to list the children of a directory (which correspond to the URIs of the documents in the directory), and the `xdmp:directory-properties` function to find the `prop:directory` element, indicating that a URI is a directory. This example has two parts:

- Directory Browser Code
- Setting Up the Directory Browser

9.5.1 Directory Browser Code

The following is sample code for a very simple directory browser.

```
xquery version "1.0-ml";
(:   directory browser
          Place in Modules database and give execute permission :)

declare namespace prop="http://marklogic.com/xdmp/property";

(: Set the root directory of your AppServer for the
   value of $rootdir :)
let $rootdir := (xdmp:modules-root())
(: take all but the last part of the request path, after the
   initial slash :)
let $dirpath := fn:substring-after(fn:string-join(fn:tokenize(
                xdmp:get-request-path(), "/")[1 to last() - 1],
                "/"), "/")
let $basedir := if ( $dirpath eq "" )
                then ( $rootdir )
                else fn:concat($rootdir, $dirpath, "/")
let $uri := xdmp:get-request-field("uri", $basedir)
return if (ends-with($uri, "/")) then
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
     <title>MarkLogic Server Directory Browser</title>
   </head>
   <body>
     <h1>Contents of {$uri}</h1>
<h3>Documents</h3>
{
   for $d in xdmp:directory($uri, "1")
   let $u := xdmp:node-uri($d)
   (: get the last two, and take the last non-empty string :)
   let $basename :=
     tokenize($u, "/")[last(), last() - 1][not(. = "")][last()]
   order by $basename
   return element p {
     element a {

(:  The following should work for all $basedir values, as long
    as the string represented by $basedir is unique in the
    document URI :)
       attribute href { substring-after($u,$basedir) },
       $basename
     }
   }
}
       <h3>Directories</h3>
{
   for $d in xdmp:directory-properties($uri, "1")//prop:directory
   let $u := xdmp:node-uri($d)
   (: get the last two, and take the last non-empty string :)
   let $basename :=
```

```
      tokenize($u, "/")[last(), last() - 1][not(. = "")][last()]
    order by $basename
    return element p {
      element a {
        attribute href { concat(
                                xdmp:get-request-path(),
                                "?uri=",
                                $u) },
        concat($basename, "/")
      }
    }
}
</body>
</html>
else doc($uri)

(: browser.xqy :)
```

This application writes out an HTML document with links to the documents and directories in the root of the server. The application finds the documents in the root directory using the `xdmp:directory` function, finds the directories using the `xdmp:directory-properties` function, does some string manipulation to get the last part of the URI to display, and keeps the state using the application server `request` object built-in XQuery functions (`xdmp:get-request-field` and `xdmp:get-request-path`).

9.5.2 Setting Up the Directory Browser

To run this directory browser application, perform the following:

1. Create an HTTP Server and configure it as follows:

   a. Set the Modules database to be the same database as the Documents database. For example, if the `database` setting is set to the database named `my-database`, set the `modules` database to `my-database` as well.

   b. Set the HTTP Server root to `http://myDirectory/`, or set the root to another value and modify the `$rootdir` variable in the directory browser code so it matches your HTTP Server root.

   c. Set the port to 9001, or to a port number not currently in use.

2. Copy the sample code into a file named `browser.xqy`. If needed, modify the `$rootdir` variable to match your HTTP Server root. Using the `xdmp:modules-root` function, as in the sample code, will automatically get the value of the App Server root.

3. Load the `browser.xqy` file into the Modules database at the top level of the HTTP Server root. For example, if the HTTP Server root is `http://myDirectory/`, load the `browser.xqy` file into the database with the URI `http://myDirectory/browser.xqy`. You can load the document either via a WebDAV client (if you also have a WebDAV server pointed to this root) or with the `xdmp:document-load` function.

4. Make sure the `browser.xqy` document has execute permissions. You can check the permissions with the following function:

   ```
   xdmp:document-get-permissions("http://myDirectory/browser.xqy")
   ```

   This command returns all of the permissions on the document. It should have "execute" capability for a role possessed by the user running the application. If it does not, you can add the permissions with a command similar to the following:

   ```
   xdmp:document-add-permissions("http://myDirectory/browser.xqy",
                                 xdmp:permission("myRole", "execute"))
   ```

   where `myRole` is a role possessed by the user running the application.

5. Load some other documents into the HTTP Server root. For example, drag and drop some documents and folders into a WebDAV client (if you also have a WebDAV server pointed to this root).

6. Access the `browser.xqy` file with a web browser using the host and port number from the HTTP Server. For example, if you are running on your local machine and you have set the HTTP Server port to 9001, you can run this application from the URL `http://localhost:9001/browser.xqy`.

You should see links to the documents and directories you loaded into the database. If you did not load any other documents, you will just see a link to the `browser.xqy` file.

10.0 Point-In-Time Queries

You can configure MarkLogic Server to retain old versions of documents, allowing you to evaluate a query statement as if you had travelled back to a point-in-time in the past. When you specify a timestamp at which a query statement should evaluate, that statement will evaluate against the newest version of the database up to (but not beyond) the specified timestamp.

This chapter describes point-in-time queries and forest rollbacks to a point-in-time, and includes the following sections:

- Understanding Point-In-Time Queries
- Using Timestamps in Queries
- Specifying Point-In-Time Queries in xdmp:eval, xdmp:invoke, xdmp:spawn, and XCC
- Keeping Track of System Timestamps
- Rolling Back a Forest to a Particular Timestamp

10.1 Understanding Point-In-Time Queries

To best understand point-in-time queries, you need to understand a little about how different versions of fragments are stored and merged out of MarkLogic Server. This section describes some details of how fragments are stored and how that enables point-in-time queries, as well as lists some other details important to understanding what you can and cannot do with point-in-time queries:

- Fragments Stored in Log-Structured Database
- System Timestamps and Merge Timestamps
- How the Fragments for Point-In-Time Queries are Stored
- Only Available on Query Statements, Not on Update Statements
- All Auxiliary Databases Use Latest Version
- Database Configuration Changes Do Not Apply to Point-In-Time Fragments

For more information on how merges work, see the "Understanding and Controlling Database Merges" chapter of the *Administrator's Guide*. For background material for this chapter, see "Understanding Transactions in MarkLogic Server" on page 31.

10.1.1 Fragments Stored in Log-Structured Database

A MarkLogic Server database consists of one or more forests. Each forest is made up of one or more stands. Each stand contains one or more fragments. The number of fragments are determined by several factors, including the number of documents and the fragment roots defined in the database configuration.

To maximize efficiency and improve performance, the fragments are maintained using a method analagous to a *log-structured filesystem*. A log-structured filesystem is a very efficient way of adding, deleting, and modifying files, with a garbage collection process that periodically removes obsolete versions of the files. In MarkLogic Server, fragments are stored in a log-structured database. MarkLogic Server periodically merges two or more stands together to form a single stand. This merge process is equivalent to the garbage collection of log-structured filesystems.

When you modify or delete an existing document or node, it affects one or more fragments. In the case of modifying a document (for example, an `xdmp:node-replace` operation), MarkLogic Server creates new versions of the fragments involved in the operation. The old versions of the fragments are marked as obsolete, but they are not yet deleted. Similarly, if a fragment is deleted, it is simply marked as obsolete, but it is not immediately deleted from disk (although you will no longer be able to query it without a point-in-time query).

10.1.2 System Timestamps and Merge Timestamps

When a merge occurs, it recovers disk space occupied by obsolete fragments. The system maintains a *system timestamp*, which is a number that increases everytime anything maintained by MarkLogic Server is changed. In the default case, the new stand is marked with the current timestamp at the time in which the merge completes (the *merge timestamp*). Any fragments that became obsolete prior to the merge timestamp (that is, any old versions of fragments or deleted fragments) are eliminated during the merge operation.

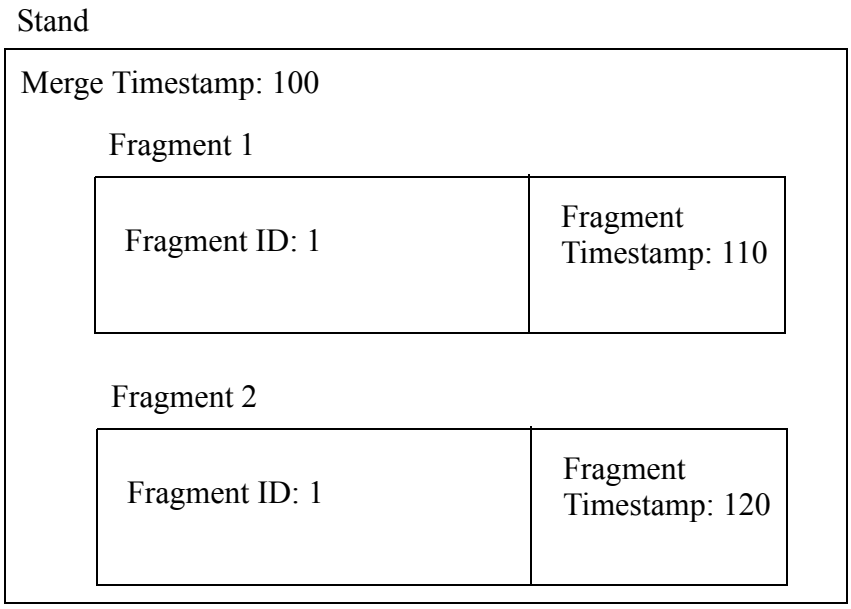
There is a control at the database level called the `merge timestamp`, set via the Admin Interface. By default, the `merge timestamp` is set to 0, which sets the timestamp of a merge to the timestamp corresponding to when the merge completes. To use point-in-time queries, you can set the `merge timestamp` to a static value corresponding to a particular time. Then, any merges that occur after that time will preserve all fragments, including obsolete fragments, whose timestamps are equal to or later than the specified `merge timestamp`.

The effect of preserving obsolete fragments is that you can perform queries that look at an older view of the database, as if you are querying the database from a point-in-time in the past. For details on setting the merge timestamp, see “Enabling Point-In-Time Queries in the Admin Interface” on page 148.

10.1.3 How the Fragments for Point-In-Time Queries are Stored

Just like any fragments, fragments with an older timestamp are stored in stands, which in turn are stored in forests. The only difference is that they have an older timestamp associated with them. Different versions of fragments can be stored in different stands or in the same stand, depending on if they have been merged into the same stand.

The following figure shows a stand with a merge timestamp of 100. Fragment 1 is a version that was changed at timestamp 110, and fragment 2 is a version of the same fragment that was changed at timestamp 120.



In this scenario, if you assume that the current time is timestamp 200, then a query at the current time will see Fragment 2, but not Fragment 1. If you perform a point-in-time query at timestamp 115, you will see Fragment 1, but not Fragment 2 (because Fragment 2 did not yet exist at timestamp 115).

There is no limit to the number of different versions that you can keep around. If the `merge timestamp` is set to the current time or a time in the past, then all subsequently modified fragments will remain in the database, available for point-in-time queries.

10.1.4 Only Available on Query Statements, Not on Update Statements

You can only specify a point-in-time query statement; attempts to specify a point-in-time query for an update statement will throw an exception. An update statement is any XQuery issued against MarkLogic Server that includes an update function (`xdmp:document-load`, `xdmp:node-replace`, and so on). For more information on what constitutes query statements and update statements, see "Understanding Transactions in MarkLogic Server" on page 31.

10.1.5 All Auxiliary Databases Use Latest Version

The auxiliary databases associated with a database request (that is, the Security, Schemas, Modules, and Triggers databases) all operate at the latest timestamp, even during a point-in-time query. Therefore, any changes made to security objects, schemas, and so on since the time specified in the point-in-time query are reflected in the query. For example, if the user you are running as was deleted between the time specified in the point-in-time query and the latest timestamp, then that query would fail to authenticate (because the user no longer exists).

10.1.6 Database Configuration Changes Do Not Apply to Point-In-Time Fragments

If you make configuration changes to a database (for example, changing database index settings), those changes only apply to the latest versions of fragments. For example, if you make index option changes and reindex a database that has old versions of fragments retained, only the latest versions of the fragments are reindexed. The older versions of fragments, used for point-in-time queries, retain the indexing properties they had at the timestamp in which they became invalid (that is, from the timestamp when an update or delete occured on the fragments). MarkLogic recommends that you do not change database settings and reindex a database that has the `merge timestamp` database parameter set to anything but 0.

10.2 Using Timestamps in Queries

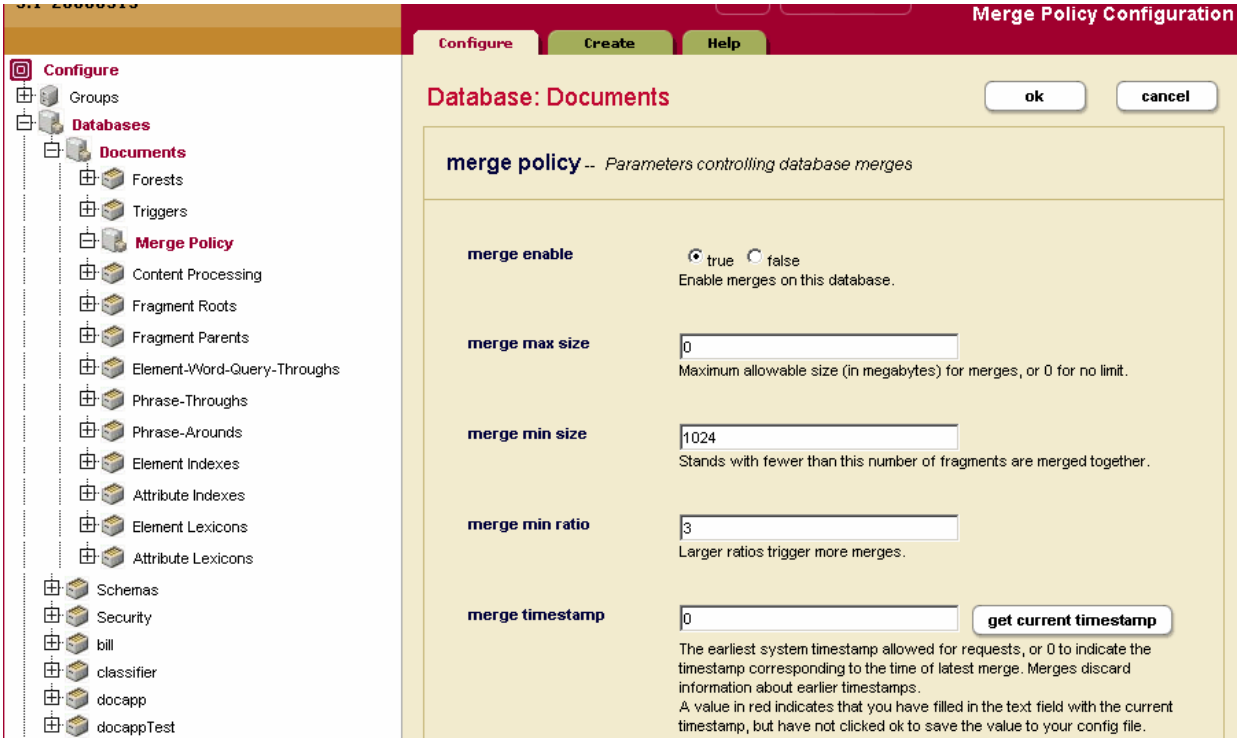
By default, query statements are run at the system timestamp in effect when the statement initiates. To run a query statement at a different system timestamp, you must set up your system to store older versions of documents and then specify the timestamp when you issue a point-in-time query statement. This section describes this general process and includes the following parts:

- Enabling Point-In-Time Queries in the Admin Interface
- The xdmp:request-timestamp Function
- Requires the xdmp:timestamp Execute Privilege
- The Timestamp Parameter to xdmp:eval, xdmp:invoke, xdmp:spawn
- Timestamps on Requests in XCC
- Scoring Considerations

10.2.1 Enabling Point-In-Time Queries in the Admin Interface

In order to use point-in-time queries in a database, you must set up merges to preserve old versions of fragments. By default, old versions of fragments are deleted from the database after a merge. For more information on how merges work, see the "Understanding and Controlling Database Merges" chapter of the *Administrator's Guide*.

In the Merge Policy Configuration page of the Admin Interface, there is a `merge timestamp` parameter. When this parameter is set to 0 (the default) and merges are enabled, point-in-time queries are effectively disabled. To access the Merge Policy Configuration page, click the Databases > *db_name* > Merge Policy link from the tree menu of the Admin Interface.



When deciding the value at which to set the `merge timestamp` parameter, the most likely value to set it to is the current system timestamp. Setting the value to the current system timestamp will preserve any versions of fragments from the current time going forward. To set the `merge timestamp` parameter to the current timestamp, click the `get current timestamp` button on the Merge Control Configuration page and then Click OK.

If you set a value for the `merge timestamp` parameter higher than the current timestamp, MarkLogic Server will use the current timestamp when it merges (the same behavior as when set to the default of 0). When the system timestamp grows past the specified `merge timestamp` number, it will then start using the `merge timestamp` specified. Similarly, if you set a `merge timestamp` lower than the lowest timestamp preserved in a database, MarkLogic Server will use the lowest timestamp of any preserved fragments in the database, or the current timestamp, whichever is lower.

You might want to keep track of your system timestamps over time, so that when you go to run point-in-time queries, you can map actual time with system timestamps. For an example of how to create such a timestamp record, see “Keeping Track of System Timestamps” on page 153.

Note: After the system merges when the `merge timestamp` is set to 0, all obsolete versions of fragments will be deleted; that is, only the latest versions of fragments will remain in the database. If you set the `merge timestamp` to a value lower than the current timestamp, any obsolete versions of fragments will not be available (because they no longer exist in the database). Therefore, if you want to preserve versions of fragments, you must configure the system to do so before you update the content.

10.2.2 The xdmp:request-timestamp Function

MarkLogic Server has an XQuery built-in function, `xdmp:request-timestamp`, which returns the system timestamp for the current request. MarkLogic Server uses the system timestamp values to keep track of versions of fragments, and you use the system timestamp in the `merge timestamp` parameter (described in "Enabling Point-In-Time Queries in the Admin Interface" on page 148) to specify which versions of fragments remain in the database after a merge. For more details on the `xdmp:request-timestamp` function, see the *MarkLogic XQuery and XSLT Function Reference*.

10.2.3 Requires the xdmp:timestamp Execute Privilege

In order to run a query at a timestamp other than the current timestamp, the user who runs the query must belong to a group that has the `xdmp:timestamp` execute privilege. For details on security and execute privileges, see *Security Guide*.

10.2.4 The Timestamp Parameter to xdmp:eval, xdmp:invoke, xdmp:spawn

The `xdmp:eval`, `xdmp:invoke`, and `xdmp:spawn` functions all take an `options` node as the optional third parameter. The options node must be in the `xdmp:eval` namespace. The options node has a `timestamp` element which allows you to specify a system timestamp at which the query should run. When you specify a `timestamp` value earlier than the current timestamp, you are specifying a point-in-time query.

The timestamp you specify must be valid for the database. If you specify a system timestamp that is less than the oldest timestamp preserved in the database, the statement will throw an `XDMP-OLDSTAMP` exception. If you specify a timestamp that is newer than the current timestamp, the statement will throw an `XDMP-NEWSTAMP` exception.

> **Note:** If the merge timestamp is set to the default of 0, and if the database has completed all merges since the last updates or deletes, query statements that specify any timestamp older than the current system timestamp will throw the `XDMP-OLDSTAMP` exception. This is because the merge timestamp value of 0 specifies that no obsolete fragments are to be retained.

The following example shows an `xdmp:eval` statement with a `timestamp` parameter:

```
xdmp:eval("doc('/docs/mydocument.xml')", (),
  <options xmlns="xdmp:eval">
    <timestamp>99225</timestamp>
  </options>)
```

This statement will return the version of the `/docs/mydocument.xml` document that existed at system timestamp 99225.

10.2.5 Timestamps on Requests in XCC

The `xdmp:eval`, `xdmp:invoke`, and `xdmp:spawn` functions allow you to specify timestamps for a query statement at the XQuery level. If you are using the XML Content Connector (XCC) libraries to communicate with MarkLogic Server, you can also specify timestamps at the Java.

In XCC for Java, you can set options to requests with the `RequestOptions` class, which allows you to modify the environment in which a request runs. The `setEffectivePointInTime` method sets the timestamp in which the request runs. The core design pattern is to set up options for your requests and then use those options when the requests are submitted to MarkLogic Server for evaluation. You can also set request options on the `Session` object. The following Java code snippet shows the basic design pattern:

```
// create a class and methods that use code similar to
// the following to set the system timestamp for requests

Session session = getSession();
BigInteger timestamp = session.getCurrentServerPointInTime();
RequestOptions options = new RequestOptions();

options.setEffectivePointInTime (timestamp);
session.setDefaultRequestOptions (options);
```

For an example of how you might use a Java environment to run point-in-time queries, see "Example: Query Old Versions of Documents Using XCC" on page 152.

10.2.6 Scoring Considerations

When you store multiple versions of fragments in a database, it will subtly effect the scores returned with `cts:search` results. The scores are calculated using document frequency as a variable in the scoring formula (for the default `score-logtfidf` scoring method). The amount of effect preserving older versions of fragments has depends on two factors:

- How many fragments have multiple versions.
- How many total fragments are in the database.

If the number of fragments with multiple versions is small compared with the total number of fragments in the database, then the effect will be relatively small. If that ratio is large, then the effect on scores will be higher.

For more details on scores and the scoring methods, see Relevance Scores: Understanding and Customizing in the *Search Developer's Guide*.

10.3 Specifying Point-In-Time Queries in xdmp:eval, xdmp:invoke, xdmp:spawn, and XCC

As desribed earlier, specifying a valid `timestamp` element in the `options` node of the `xdmp:eval`, `xdmp:invoke`, or `xdmp:spawn` functions initiates a point-in-time query. Also, you can use XCC to specify entire XCC requests as point-in-time queries. The query runs at the specified timestamp, seeing a version of the database that existed at the point in time corresponding to the specified timestamp. This section shows some example scenarios for point-in-time queries, and includes the following parts:

- Example: Query Old Versions of Documents Using XCC
- Example: Querying Deleted Documents

10.3.1 Example: Query Old Versions of Documents Using XCC

When making updates to content in your system, you might want to add and test new versions of the content before exposing the new content to your users. During this testing time, the users will still see the old version of the content. Then, when the new content has been sufficiently tested, you can switch the users over to the new content.

Point-in-time queries allow you to do this all within the same database. The only thing that you need to change in the application is the timestamps at which the query statements run. XCC provides a convenient mechanism for accomplishing this goal.

10.3.2 Example: Querying Deleted Documents

When you delete a document, the fragments for that document are marked as obsolete. The fragments are not actually deleted from disk until a merge completes. Also, if the `merge timestamp` is set to a timestamp earlier than the timestamp corresponding to when the document was deleted, the merge will preserve the obsolete fragments.

This example demonstrates how you can query deleted documents with point-in-time queries. For simplicity, assume that no other query or update activity is happening on the system for the duration of the example. To follow along in the example, run the following code samples in the order shown below.

1. First, create a document:

```
xdmp:document-insert("/docs/test.xml", <a>hello</a>))
```

2. When you query the document, it returns the node you inserted:

```
doc("/docs/test.xml")
(: returns the node <a>hello</a> :)
```

3. Delete the document:

```
xdmp:document-delete("/docs/test.xml")
```

4. Query the document again. It returns the empty sequence because it was just deleted.

5. Run a point-in-time query, specifying the current timestamp (this is semantically the same as querying the document without specifying a timestamp):

```
xdmp:eval("doc('/docs/test.xml')", (),
<options xmlns="xdmp:eval">
  <timestamp>{xdmp:request-timestamp()}</timestamp>
</options>)
(: returns the empty sequence because the document has been deleted :)
```

6. Run the point-in-time query at one less than the current timestamp, which is the old timestamp in this case because only one change has happened to the database. The following query statement returns the old document.

```
xdmp:eval("doc('/docs/test.xml')", (),
<options xmlns="xdmp:eval">
  <timestamp>{xdmp:request-timestamp()-1}</timestamp>
</options>)
(: returns the deleted version of the document :)
```

10.4 Keeping Track of System Timestamps

The system timestamp does not record the actual time in which updates occur; it is simply a number that is increased each time an update or configuration change occurs in the system. If you want to map system timestamps with actual time, you need to either store that information somewhere or use the `xdmp:timestamp-to-wallclock` and `xdmp:wallclock-to-timestamp` XQuery functions. This section shows a design pattern, including some sample code, of the basic principals for creating an application that archives the system timestamp at actual time intervals.

> **Note:** It might not be important to your application to map system timestamps to actual time. For example, you might simply set up your merge timestamp to the current timestamp, and know that all versions from then on will be preserved. If you do not need to keep track of the system timestamp, you do not need to create this application.

The first step is to create a document in which the timestamps are stored, with an initial entry of the current timestamp. To avoid possible confusion of future point-in-time queries, create this document in a different database than the one in which you are running point-in-time queries. You can create the document as follows:

```
xdmp:document-insert("/system/history.xml",
<timestamp-history>
  <entry>
    <datetime>{fn:current-dateTime()}</datetime>
    <system-timestamp>{
(: use eval because this is an update statement :)
      xdmp:eval("xdmp:request-timestamp()")}
    </system-timestamp>
```

```
  </entry>
</timestamp-history>)
```

This results in a document similar to the following:

```
<timestamp-history>
  <entry>
    <datetime>2006-04-26T19:35:51.325-07:00</datetime>
    <system-timestamp>92883</system-timestamp>
  </entry>
</timestamp-history>
```

Note that the code uses `xdmp:eval` to get the current timestamp. It must use `xdmp:eval` because the statement is an update statement, and update statements always return the empty sequence for calls to `xdmp:request-timestamp`. For details, see "Understanding Transactions in MarkLogic Server" on page 31.

Next, set up a process to run code similar to the following at periodic intervals. For example, you might run the following every 15 minutes:

```
xdmp:node-insert-child(doc("/system/history.xml")/timestamp-history,
<entry>
  <datetime>{fn:current-dateTime()}</datetime>
  <system-timestamp>{
(: use eval because this is an update statement :)
   xdmp:eval("xdmp:request-timestamp()")}
  </system-timestamp>
</entry>)
```

This results in a document similar to the following:

```
<timestamp-history>
  <entry>
    <datetime>2006-04-26T19:35:51.325-07:00</datetime>
    <system-timestamp>92883</system-timestamp>
  </entry>
  <entry>
    <datetime>2006-04-26T19:46:13.225-07:00</datetime>
    <system-timestamp>92884</system-timestamp>
  </entry>
</timestamp-history>
```

To call this code at periodic intervals, you can set up a cron job, write a shell script, write a Java or dotnet program, or use any method that works in your environment. Once you have the document with the timestamp history, you can easily query it to find out what the system timestamp was at a given time.

10.5 Rolling Back a Forest to a Particular Timestamp

In addition to allowing you to query the state of the database at a given point in time, setting a merge timestamp and preserving deleted fragments also allows you to roll back the state of one or more forests to a timestamp that is preserved. To roll back one or more forests to a given timestamp, use the `xdmp:forest-rollback` function. This section covers the following topics about using xdmp:forest-rollback to roll back the state of one or more forests:

- Tradeoffs and Scenarios to Consider For Rolling Back Forests
- Setting the Merge Timestamp
- Notes About Performing an xdmp:forest-rollback Operation
- General Steps for Rolling Back One or More Forests

10.5.1 Tradeoffs and Scenarios to Consider For Rolling Back Forests

In order to roll a forest back to a previous timestamp, you need to have previously set a merge timestamp that preserved older versions of fragments in your database. Keeping deleted fragments around will make your database grow in size faster, using more disk space and other system resources. The advantage of keeping old fragments around is that you can query the older fragments (using point-in-time queries as described in the previous sections) and also that you can roll back the database to a previous timestamp. You should consider the advantages (the convenience and speed of bringing the state of your forests to a previous time) and the costs (disk space and system resources, keeping track of your system timestamps, and so on) when deciding if it makes sense for your system.

A typical use case for forest rollbacks is to guard against some sort of data-destroying event, providing the ability to get back to the point in time before that event without doing a full database restore. If you wanted to allow your application to go back to some state within the last week, for example, you can create a process whereby you update the merge timestamp every day to the system timestamp from 7 days ago. This would allow you to go back any point in time in the last 7 days. To set up this process, you would need to do the following:

- Maintain a mapping between the system timestamp and the actual time, as described in “Keeping Track of System Timestamps” on page 153.
- Create a script (either a manual process or an XQuery script using the Admin API) to update the merge timestamp for your database once every 7 days. The script would update the merge timestamp to the system timestamp that was active 7 days earlier.
- If a rollback was needed, roll back all of the forests in the database to a time between the current timestamp and the merge timestamp. For example:

```
xdmp:forest-rollback(
     xdmp:database-forests(xdmp:database("my-db")),
     3248432)
(: where 3248432 is the timestamp to which you want to roll back :)
```

Another use case to set up an environment for using forest rollback operations is if you are pushing a new set of code and/or content out to your application, and you want to be able to roll it back to the previous state. To set up this scenario, you would need to do the following:

- When your system is in a steady state before pushing the new content/code, set the merge timestamp to the current timestamp.
- Load your new content/code.
- Are you are happy with your changes?
    - If yes, then you can set the merge timestamp back to 0, which will eventually merge out your old content/code (because they are deleted fragments).
    - If no, then roll all of the forests in the database back to the timestamp that you set in the merge timestamp.

10.5.2 Setting the Merge Timestamp

As described above, you cannot roll back forests in which the database merge timestamp has not been set. By default, the merge timestamp is set to 0, which will delete old versions of fragments during merge operations. For details, see "System Timestamps and Merge Timestamps" on page 146.

10.5.3 Notes About Performing an xdmp:forest-rollback Operation

This section describes some of the behavior of `xdmp:forest-rollback` that you should understand before setting up an environment in which you can roll back your forests. Note the following about `xdmp:forest-rollback` operations:

- An `xdmp:forest-rollback` will restart the specified forest(s). As a consequence, any failed over forests will attempt to mount their primary host; that is, it will result in an un-failover operation if the forest is failed over. For details on failover, see High Availability of Data Nodes With Failover in the *Scalability, Availability, and Failover Guide* guide.
- Use caution when rolling back one or more forests that are in the context database (that is, forests that belong to the database against which your query is evaluating against). When in a forest in the context database, the `xdmp:forest-rollback` operation is run asyncronously. The new state of the forest is not seen until the forest restart occurs, Before the forest is unmounted, the old state will still be reflected. Additionally, any errors that might occur as part of the rollback operation are not reported back to the query that performs the operation (although, if possible, they are logged to the `ErrorLog.txt` file). As a best practice, MarkLogic recommends running `xdmp:forest-rollback` operations against forests not attached to the context database.

- If you do not specify all of the forests in a database to roll back, you might end up in a state where the rolled back forest is not in a consistent state with the other forests. In most cases, it is a good idea to roll back all of the forests in a database, unless you are sure that the content of the forest being rolled back will not become inconsistent if other forests are not rolled back to the same state (for example, if you know that all of content you are rolling back is only in one forest).
- If your database indexing configuration has changed since the point in time to which you are rolling back, and if you have reindexing enabled, a rollback operation will begin reindexing as soon as the rollback operation completes. If reindexing is not enabled, then the rolled backed fragments will remain indexed as they were at the time they were last updated, which might be inconsistent with the current database configuration.
- As a best practice, MarkLogic recommends running a rollback operation only on forests that have no update activitiy at the time of the operation (that is, the forests should be quiesced).

10.5.4 General Steps for Rolling Back One or More Forests

To roll back the state of one or more forests, perform the following general steps:

1. At the state of the database to which you want to be able to roll back, set the merge timestamp to the current timestamp.

2. Keep track of your system timestamps, as desribed in “System Timestamps and Merge Timestamps” on page 146.

3. Perform updates to your application as usual. Old version of document will remain in the database.

4. If you know you will not need to roll back to a time earlier, than the present, go back to step 1.

5. If you want to roll back, you can roll back to any time between the merge timestamp and the current timestamp. When you perform the rollback, it is a good idea to do so from the context of a different database. For example, to roll back all of the forests in the `my-db` database, perform an operation similar to the following, which sets the database context to a different one than the forests that are being rolled back:

```
xdmp:eval(
 'xdmp:forest-rollback(
    xdmp:database-forests(xdmp:database("my-db")),
    3248432)
 (: where 3248432 is the timestamp to which you want
    to roll back :)',
 (),
 <options xmlns="xdmp:eval">
   <database>{xdmp:database("Documents")}</database>
 </options>)
```

11.0 System Plugin Framework

This chapter describes the system plugin framework in MarkLogic Server, and includes the following sections:

- How MarkLogic Server Plugins Work
- Writing System Plugin Modules
- Password Plugin Sample

11.1 How MarkLogic Server Plugins Work

Plugins allow you to provide functionality to all of the applications in your MarkLogic Server cluster without the application having to call any code. This section describes the system plugin framework in MarkLogic Server and includes the following parts:

- Overview of System Plugins
- System Plugins versus Application Plugins
- The plugin API

11.1.1 Overview of System Plugins

Plugins are used to automatically perform some functionality before any request is evaluated. A plugin is an XQuery main module, and it can therefore perform arbitrary work. The plugin framework evaluates the main modules in the `<marklogic-dir>/Plugins` directory before each request is evaluated.

Consider the following notes about how the plugin framework works:

- After MarkLogic starts up, each module in the `Plugins` directory is evaluated before the first request against each App Server is evaluated on each node in the cluster. This process repeats again after the `Plugins` directory is modified.
- When using a cluster, any files added to the `Plugins` directory must be added to the `Plugins` directory on each node in a MarkLogic Server cluster.
- Any errors (for example, syntax errors) in a plugin module are thrown whenever any request is made to any App Server in the cluster (including the Admin Interface). It is therefore extremely important that you test the plugin modules before deploying them to the *`<marklogic-dir>`*`/Plugins` directory. If there are any errors in a plugin module, you must fix them before you will be able to successfully evalate any requests against any App Server.

- Plugins are cached and, for performance reasons, MarkLogic Server only checks for updates once per second, and only refreshes the cache after the `Plugins` directory is modified; it does not check for modifications of the individual files in the `Plugins` directory. If you are using an editor to modify a plugin that creates a new file (which in turn modifies the directory) upon each update, then MarkLogic Server will see the update within the next second. If your editor modifies the file in place, then you will have to touch the directory to change the modification date for the latest changes to be loaded (alternatively, you can restart MarkLogic Server). If you delete a plugin from the `Plugins` directory, it remains registered on any App Servers that have already evaluated the plugin until either you restart MarkLogic Server or another plugin registers with the same name on each App Server.

11.1.2 System Plugins versus Application Plugins

There are two types of plugins in MarkLogic Server: *system plugins* and *application plugins*.

System plugins use the built-in plugin framework in MarkLogic Server along with the `xdmp:set-server-field` and `xdmp:get-server-field` functions. As described in "Overview of System Plugins" on page 159, system plugins are stored in the `<marklogic-dir>/Plugins` directory and any errors in them are thrown on all App Servers in the cluster.

Application plugins are built on top of system plugins and are designed for use by applications. Application plugins are stored in the `<marklogic-dir>/Assets/plugins/marklogic/appservices` directory, and, unlike system plugins, they do not cause errors to other applications if the plugin code contains errors.

11.1.3 The plugin API

The `plugin:register` function is the mechanism that a plugin module uses to make plugin functionality available anywhere in a MarkLogic Server cluster. The other functions in the `plugin` API are used to implement the register capability. The `plugin` API uses server fields (the `xdmp:set-server-field` and `xdmp:get-server-field` family of functions) to register the ID and capabilities of each plugin. This API, in combination with the plugin framework that scans the `Plugins` directory, allows you to create functionality that is available to all App Servers in a MarkLogic Server cluster.

With the plugin API, you can register a set of plugins, and then you can ask for all of the plugins with a particular capability, and the functionality delivered by each plugin is available to your application. For details about the plugin API, see the *MarkLogic XQuery and XSLT Function Reference*.

11.2 Writing System Plugin Modules

A plugin module is just an XQuery main module, so in that sense, you can put any main module in the `Plugins` directory and you have a plugin. So it really depends what you are trying to accomplish.

Warning Any errors in a system plugin module will cause all requests to hit the error. It is therefore extremely important to test your plugins before deploying them in a production environment.

To use a system plugin, you must deploy the plugin main module to the `Plugins` directory. To deploy a plugin to a MarkLogic Server cluster, you must copy the plugin main module to the plugin directory of each host in the cluster.

Warning Any system plugin module you write should have a unique filename. Do not modify any of the plugin files that MarkLogic ships in the `<marklogic-dir>/Plugins` directory. Any changes you make to MarkLogic-installed files in the `Plugins` directory will be overridden after each upgrade of MarkLogic Server.

11.3 Password Plugin Sample

This section describes the password plugin and provides a sample of how to modify it, and contains the following parts:

- Understanding the Password Plugin
- Modifying the Password Plugin

11.3.1 Understanding the Password Plugin

One use case for a system plugin is to check passwords for things like number of characters, special characters, and so on. Included in the `<marklogic-dir>/Samples/Plugins` directory are sample plugin modules for password checking.

When a password is set using the security XQuery library (`security.xqy`), it calls the plugin to check the password using the plugin capability with the following URI:

```
http://marklogic.com/xdmp/security/password-check
```

When no plugins are registered with the above capability in the `<marklogic-dir>/Plugins` directory, then no other work is done upon setting a password. If you include plugins that register with the above `password-check` capability in the `<marklogic-dir>/Plugins` directory, then the module(s) are run when you set a password. If multiple plugins are registered with that capability, then they will all run. The order in which they run is undetermined, so the code should be designed such that the order does not matter.

There is a sample included that checks for a minimum length and a sample included that checks to see if the password contains digits. You can create your own plugin module to perform any sort of password checking you require (for example, check for a particular length, the existence of various special characters, repeated charatcers, upper or lower case, and so on).

Additionally, you can write a plugin to save extra history in the Security database user document, which stores information that you can use or update in your password checking code. The element you can use to store information for password checking applications is `sec:password-extra`. You can use the `sec:user-set-password-extra` and `sec:user-set-password-extra` functions (in `security.xqy`) to modify the `sec:password-extra` element in the user document. Use these APIs to create elements as children of the `sec:password-extra` element.

If you look at the `<marklogic-dir>/Samples/Plugins/password-check-minimum-length.xqy` file, you will notice that it is a main module with a function that returns empty on success, and an error message if the password is less than a minimum number of characters. In the body of the main module, the plugin is registered with a map that includes its capability (it could register several capabilities, but this only registers one) and a unique name (in this case, the name of the xqy file:

```
let $map := map:map(),
      $_ := map:put($map,
"http://marklogic.com/xdmp/security/password-check",
                    xdmp:function(xs:QName("pwd:minimum-length")))
return
  plugin:register($map, "password-check-minimum-length.xqy")
```

This registers the function `pwd:minimum-length` with the `http://marklogic.com/xdmp/security/password-check` capability, and this particular plugin is called each time a password is set.

Note: You should use a unique name to register your plugin (the second argument to `plugin:register`). If the name is used by another plugin, only one of them will end up being registered (because the other one will overwrite the registration).

If you want to implement your own logic that is performed when a password is checked (both on creating a user and on changing the password), then you can write a plugin, as described in the next section.

11.3.2 Modifying the Password Plugin

The following example shows how to use the sample plugins to check for a minimum password length and to ensure that it contains at least one numeric character.

Warning Any errors in a plugin module will cause all requests to hit the error. It is therefore extremely important to test your plugins before deploying them in a production environment.

To use and modify the sample password plugins, perform the following steps:

1. Copy the `<marklogic-dir>Samples/Plugins/password-check-*.xqy` files to the `Plugins` directory. For example:

```
cd /opt/MarkLogic/Plugins
cp ../Samples/Plugins/password-check-*.xqy .
```

If desired, rename the files when you copy them.

2. If you want to modify any of the files (for example, `password-check-minimum-length`), open them in a text editor.

3. Make any changes you desire. For example, to change the minimum length, find the `pwd:minimum-length` function and change the 4 to a 6 (or to whatever you prefer). When you are done, the body of the function looks as follows:

```
if (fn:string-length($password) < 6)
then "password too short"
else ()
```

This checks that the password contains at least 6 characters.

4. Optionally, if you have renamed the files, change the second parameter to `plugin:register` to the name you called the plugin files in the first step. For example, if you named the plugin file `my-password-plugin.xqy`, change the `plugin:register` call as follows:

```
plugin:register($map, "my-password-plugin.xqy")
```

5. Save your changes to the file.

Warning If you made a typo or some other mistake that causes a syntax error in the plugin, any request you make to any App Server will throw an exception. If that happens, edit the file to correct any errors.

6. If you are using a cluster, copy your plugin to the `Plugins` directory on each host in your cluster.

7. Test your code to make sure it works the way you intend.

The next time you try and change a password, your new checks will be run. For example, if you try to make a single-character password, it will be rejected.

12.0 Using the map Functions to Create Name-Value Maps

This chapter describes how to use the map functions and includes the following sections:

- Maps: In-Memory Structures to Manipulate in XQuery
- map:map XQuery Primitive Type
- Serializing a Map to an XML Node
- Map API
- Map Operators
- Examples

12.1 Maps: In-Memory Structures to Manipulate in XQuery

Maps are in-memory structures containing name-value pairs that you can create and manipulate. In some programming languages, maps are implemented using hash tables. Maps are handy programming tools, as you can conveniently store and update name-value pairs for use later in your program. Maps provide a fast and convenient method for accessing data.

MarkLogic Server has a set of XQuery functions to create manipulate maps. Like the `xdmp:set` function, maps have side-effects and can change within your program. Therefore maps are not strictly functional like most other aspects of XQuery. While the map is in memory, its structure is opaque to the developer, and you access it with the built-in XQuery functions. You can persist the structure of the map as an XML node, however, if you want to save it for later use. A map is a node and therefore has an identity, and the identity remains the same as long as the map is in memory. However, if you serialize the map as XML and store it in a document, when you retrieve it will have a different node identity (that is, comparing the identity of the map and the serialized version of the map would return false). Similarly, if you store XML values retrieved from the database in a map, the node in the in-memory map will have the same identity as the node from the database while the map is in memory, but will have different identities after the map is serialized to an XML document and stored in the database. This is consistent with the way XQuery treats node identity.

The keys take `xs:string` types, and the values take `item()*` values. Therefore, you can pass a string, an element, or a sequence of items to the values. Maps are a nice alternative to storing values an in-memory XML node and then using XPath to access the values. Maps makes it very easy to update the values.

12.2 map:map XQuery Primitive Type

Maps are defined as a `map:map` XQuery primitive type. You can use this type in function or variable definitions, or in the same way as you use other primitive types in XQuery. You can also serialize it to XML, which lets you store it in a database, as described in the following section.

12.3 Serializing a Map to an XML Node

You can serialize the structure of a map to an XML node by placing the map in the context of an XML element, in much the same way as you can serialize a `cts:query` (see Serializing a cts:query as XML in the Composing cts:query Expressions chapter of the *Search Developer's Guide*). Serializing the map is useful if you want to save the contents of the map by storing it in the database. The XML conforms to the `<marklogic-dir>/Config/map.xsd` schema, and has the namespace `http://marklogic.com/xdmp/map`.

For example, the following returns the XML serialization of the constructed map:

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
let $node := <some-element>{$map}</some-element>
return $node/map:map
```

The following XML is returned:

```
<map:map xmlns:map="http://marklogic.com/xdmp/map"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <map:entry key="1">
    <map:value xsi:type="xs:string">hello</map:value>
  </map:entry>
  <map:entry key="2">
    <map:value xsi:type="xs:string">world</map:value>
  </map:entry>
</map:map>
```

12.4 Map API

The map API is quite simple. You can create a map either from scratch with the `map:map` function or from the XML representation (`map:map`) of the map. The following are the map functions. For the signatures and description of each function, see the *MarkLogic XQuery and XSLT Function Reference*.

- `map:clear`
- `map:count`
- `map:delete`
- `map:get`
- `map:keys`
- `map:map`
- `map:put`

12.5 Map Operators

Map operators perform a similar function to set operators. Just as sets can be combined in a number of ways to produce another set, maps can be manipulated with map operators to create combined results. The following table describes the different map operators:

Map Operator	Description
`+`	The union of two maps. The result is the combination of the keys and values of the first map (Map A) and the second map (Map B). For an example, see "Creating a Map Union" on page 170.
`*`	The intersection of two maps (similar to a set intersection). The result is the key-value pairs that are common to both maps (Map A and Map B) are returned. For an example, see "Creating a Map Intersection" on page 171.
`-`	The difference between two maps (similar to a set difference). The result is the key-value pairs that exist in the first map (Map A) that do not exist in the second map (Map B) are returned. For an example, see "Applying a Map Difference Operator" on page 172. This operator also works as an unary negative operator. When it is used in this way, the keys and values become reversed. For an example, see "Applying a Negative Unary Operator" on page 173.
`div`	The inference that a value from a map matches the key of another map. The result is the keys from the first map (Map A), and values from the second map (Map B), where the value in Map A is equal to key in Map B. For an example, see "Applying a Div Operator" on page 174.
`mod`	The combination of the unary negative operation and inference between maps. The result is the reversal of the keys in the first map (Map A) and the values in Map B, where a value in Map A matches a key in Map B. In summary, `Map A mod Map B` is equivalent to `-Map A div Map B`. For an example, see "Applying a Mod Operator" on page 175.

12.6 Examples

This section includes example code that uses maps and includes the following examples:

- Creating a Simple Map
- Returning the Values in a Map
- Constructing a Serialized Map
- Add a Value that is a Sequence

- Creating a Map Union
- Creating a Map Intersection
- Applying a Map Difference Operator
- Applying a Negative Unary Operator
- Applying a Div Operator
- Applying a Mod Operator

12.6.1 Creating a Simple Map

The following example creates a map, puts two key-value pairs into the map, and then returns the map.

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
return $map
```

This returns a map with two key-value pairs in it: the key “1” has a value “hello”, and the key “2” has a value “world”.

12.6.2 Returning the Values in a Map

The following example creates a map, then returns its values ordering by the keys:

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
return
  for $x in map:keys($map)
  order by $x return
  map:get($map, $x)
(: returns hello world :)
```

12.6.3 Constructing a Serialized Map

The following example creates a map like the previous examples, and then serializes the map to an XML node. It then makes a new map out of the XML node and puts another key-value pair in the map, and finally returns the new map.

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
let $node := <some-element>{$map}</some-element>
let $map2 := map:map($node/map:map)
let $key := map:put($map2, "3", "fair")
return $map2
```

This returns a map with three key-value pairs in it: the key “1” has a value “hello”, the key “2” has a value “world”, and the key “3” has a value “fair”. Note that the map bound to the `$map` variable is not the same as the map bound to `$map2`. After it was serialized to XML, a new map was constructed in the `$map2` variable.

12.6.4 Add a Value that is a Sequence

The values that you can put in a map are typed as an `item()*`, which means you can add arbitrary sequences as the value for a key. The following example includes some string values and a sequence value, and then outputs each results in a `<result>` element:

```
let $map := map:map()
let $key := map:put($map, "1", "hello")
let $key := map:put($map, "2", "world")
let $seq := ("fair",
             <some-xml>
                 <another-tag>with text</another-tag>
             </some-xml>)
let $key := map:put($map, "3", $seq)
return
  for $x in map:keys($map) return
  <result>{map:get($map, $x)}</result>
```

This returns the following elements:

```
<result>fair
        <some-xml>
            <another-tag>with text</another-tag>
        </some-xml>
</result>
<result>world</result>
<result>hello</result>
```

12.6.5 Creating a Map Union

The following creates a union between two maps and returns the key-value pairs:

```
let $mapA := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>1</map:key>
            <map:value>1</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
        </map:entry>
</map:map>)
let $mapB := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>2</map:key>
            <map:value>2</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
        </map:map>)
return $mapA + $mapB
```

Any key-value pairs common to both maps are included only once. This returns the following:

```
<xml version="1.0" encoding="UTF-8">
<results warning="atomic item">
    <map:map xmlns:map="http://marklogic.com/xdmp/map"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <map:entry key="1">
            <map:value>1</map:value>
        </map:entry>
        <map:entry key="2">
            <map:value>2</map:value>
        </map:entry>
        <map:entry key="3">
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
    </map:map>
</results>
```

12.6.6 Creating a Map Intersection

The following example creates an intersection between two maps:

```
xquery version "1.0-ml";
let $mapA := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>1</map:key>
            <map:value>1</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
        </map:entry>
        </map:map>)
let $mapB := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>2</map:key>
            <map:value>2</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
</map:map>)
return $mapA * $mapB
```

The key-value pairs common to both maps are returned. This returns the following:

```
<xml version="1.0" encoding="UTF-8">
<results warning="atomic item">
    <map:map xmlns:map="http://marklogic.com/xdmp/map"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <map:entry key="3">
            <map:value>3</map:value>
        </map:entry>
    </map:map>
</results>
```

12.6.7 Applying a Map Difference Operator

The following example returns the key-value pairs that are in Map A but not in Map B:

```
let $mapA := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>1</map:key>
            <map:value>1</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
        </map:entry>
</map:map>)
let $mapB := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>2</map:key>
            <map:value>2</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
        </map:map>)
return $mapA - $mapB
```

This returns the following:

```
<xml version="1.0" encoding="UTF-8">
<results warning="atomic item">
    <map:map xmlns:map="http://marklogic.com/xdmp/map"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <map:entry key="1">
            <map:value>1</map:value>
        </map:entry>
    </map:map>
</results>
```

12.6.8 Applying a Negative Unary Operator

The following example uses the map difference operator as a negative unary operator to reverse the keys and values in a map:

```
xquery version "1.0-ml";
let $mapA := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>1</map:key>
            <map:value>1</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
        </map:entry>
        </map:map>)
let $mapB := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>2</map:key>
            <map:value>2</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
</map:map>)
return -$mapB
```

This returns the following:

```
<xml version="1.0" encoding="UTF-8">
<results warning="atomic item">
    <map:map xmlns:map="http://marklogic.com/xdmp/map"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <map:entry key="3.5">
            <map:value>3</map:value>
        </map:entry>
        <map:entry key="2">
            <map:value>2</map:value>
        </map:entry>
        <map:entry key="3">
            <map:value>3</map:value>
        </map:entry>
    </map:map>
</results>
```

12.6.9 Applying a Div Operator

The following example applies the inference rule that returns the keys from Map A and the values in Map B, where a value of Map A is equal to a key in Map B:

```
xquery version "1.0-ml";
let $mapA := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>1</map:key>
            <map:value>1</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
        </map:entry>
        </map:map>)
let $mapB := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>2</map:key>
            <map:value>2</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
</map:map>)
return $mapA div $mapB
```

This returns the following:

```
<xml version="1.0" encoding="UTF-8">
<results warning="atomic item">
    <map:map xmlns:map="http://marklogic.com/xdmp/map"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <map:entry key="3">
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
    </map:map>
</results>
```

12.6.10 Applying a Mod Operator

The following example perform two of the operations mentioned. First, the keys and values are reversed in Map A. Next, the inference rule is applied to match a value in Map A to a key in Map B and return the values in Map B.

```
xquery version "1.0-ml";
let $mapA := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>1</map:key>
            <map:value>1</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
        </map:entry>
        </map:map>)
let $mapB := map:map(
<map:map xmlns:map="http://marklogic.com/xdmp/map">
        <map:entry>
            <map:key>2</map:key>
            <map:value>2</map:value>
        </map:entry>
        <map:entry>
            <map:key>3</map:key>
            <map:value>3</map:value>
            <map:value>3.5</map:value>
        </map:entry>
</map:map>)
return $mapA mod $mapB
```

This returns the following:

```
<xml version="1.0" encoding="UTF-8">
<results warning="atomic item">
    <map:map xmlns:map="http://marklogic.com/xdmp/map"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <map:entry key="3.5">
            <map:value>3</map:value>
        </map:entry>
        <map:entry key="3">
            <map:value>3</map:value>
        </map:entry>
    </map:map>
</results>
```

13.0 Function Values

This chapter describes how to use function values, which allow you to pass function values as parameters to XQuery functions. It includes the following sections:

- Overview of Function Values
- xdmp:function XQuery Primitive Type
- XQuery APIs for Function Values
- When the Applied Function is an Update from a Query Statement
- Example of Using Function Values

13.1 Overview of Function Values

XQuery functions take parameters, and those parameters can be any XQuery type. Typically, parameters are strings, dates, numbers, and so on, and XQuery has many types to provide robust typing support. Sometimes, however, it is convenient to pass a pointer to a named function as a parameter to another function. These function pointers are known as *function values*, and they allow you to write code that can be more robust and more easily maintainable. Programming languages that support passing functions as parameters sometimes call those higher order functions. MarkLogic Server function values do most things that higher order functions in other languages do, except you cannot output a function and you cannot create anonymous functions; instead, you can output or input a function value, which is implemented as an XQuery primitive type.

You pass a function value to another function by telling it the name of the function you want to pass. The actual value returned by the function is evaluated dynamically during query runtime. Passing these function values allows you to define an interface to a function and have a default implementation of it, while allowing callers of that function to implement their own version of the function and specify it instead of the default version.

13.2 xdmp:function XQuery Primitive Type

Function values are defined as an `xdmp:function` XQuery primitive type. You can use this type in function or variable definitions, or in the same way as you use other primitive types in XQuery. Unlike some of the other MarkLogic Server XQuery primitive types (`cts:query` and `map:map`, for example), there is no XML serialization for the `xdmp:function` XQuery primitive type.

13.3 XQuery APIs for Function Values

The following XQuery built-in functions are used to pass function values:

- `xdmp:function`
- `xdmp:apply`

You use `xdmp:function` to specify the function to pass in, and `xdmp:apply` to run the function that is passed in. For details and the signature of these APIs, see the *MarkLogic XQuery and XSLT Function Reference*.

13.4 When the Applied Function is an Update from a Query Statement

When you apply a function using `xdmp:function`, MarkLogic Server does not know the contents of the applied function at query compilation time. Therefore, if the statement calling `xdmp:apply` is a query statement (that is, it contains no update expressions and therefore runs at a timestamp), and the function being applied is performing an update, then it will throw an `XDMP-UDATEFUNCTIONFROMQUERY` exception.

If you have code that you will apply that performs an update, and if the calling query does not have any update statements, then you must make the calling query an update statement. To change a query statement to be an update statement, either use the `xdmp:update` prolog option or put an update call somewhere in the statement. For example, to force a query to run as an update statement, you can add the following to your XQuery prolog:

```
declare option xdmp:update "true";
```

Without the prolog option, any update expression in the query will force it to run as an update statement. For example, the following expression will force the query to run as an update statement and not change anything else about the query:

```
if ( fn:true() )
then ()
else xdmp:document-insert("fake.xml", <fake/>)
```

For details on the difference between update statements and query statements, see “Understanding Transactions in MarkLogic Server” on page 31.

13.5 Example of Using Function Values

The following example shows a recursive function, `my:sum:sequences`, that takes an `xdmp:function` type, then applies that function call recursively until it reaches the end of the sequence. It shows how the caller can supply her own implementation of the `my:add` function to change the behavior of the `my:sum-sequences` function. Consider the following library module named `/sum.xqy`:

```
xquery version "1.0-ml";
module namespace my="my-namespace";

(: Sum a sequence of numbers, starting with the
```

```
   starting-number (3rd parameter) and at the
   start-position (4th parameter). :)
declare function my:sum-sequence(
  $fun as xdmp:function,
  $items as item()*,
  $starting-number as item(),
  $start-position as xs:unsignedInt)
as item()
{
  if ($start-position gt fn:count($items)) then $starting-number
  else
    let $new-value := xdmp:apply($fun,$starting-number,
                        $items[$start-position])
    return
      my:sum-sequence($fun,$items,$new-value,$start-position+1)
};

declare function my:add($x,$y) {$x+ $y};
(: /sum.xqy :)
```

Now call this function with the following main module:

```
xquery version "1.0-ml";
import module namespace my="my-namespace" at "/sum.xqy";

let $fn := xdmp:function(xs:QName("my:add"))
return my:sum-sequence($fn,(1 to 100), 2, 1)
```

This returns 5052, which is the sum of all of the numbers between 2 and 100.

If you want to use a different formula for adding up the numbers, you can create an XQuery library module with a different implementation of the same function and specify it instead. For example, assume you want to use a different formula to add up the numbers, and you create another library module named `/my.xqy` that has the following code (it multiplies the second number by two before adding it to the first):

```
xquery version "1.0-ml";
module namespace my="my-namespace";

declare function my:add($x,$y) {$x+ (2 * $y)};
(: /my.xqy :)
```

You can now call the `my:sum-sequence` function specifying your new implementation of the `my:add` function as follows:

```
xquery version "1.0-ml";
import module namespace my="my-namespace" at "/sum.xqy";

let $fn := xdmp:function(xs:QName("my:add"), "/my.xqy")
return my:sum-sequence($fn,(1 to 100), 2, 1)
```

This returns 10102 using the new formula. This technique makes it possible for the caller to specify a completely different implementation of the specified function that is passed.

14.0 Reusing Content With Modular Document Applications

This chapter describes how to create applications that reuse content by using XML that includes other content. It contains the following sections:

- Modular Documents
- XInclude and XPointer
- CPF XInclude Application and API
- Creating XML for Use in a Modular Document Application
- Setting Up a Modular Document Application

14.1 Modular Documents

A *modular document* is an XML document that references other documents or parts of other documents for some or all of its content. If you fetch the referenced document parts and place their contents as child elements of the elements in which they are referenced, then that is called *expanding* the document. If you expand all references, including any references in expanded documents (recursively, until there is nothing left to expand), then the resulting document is called the *expanded document*. The expanded document can then be used for searching, allowing you to get relevance-ranked results where the relevance is based on the entire content in a single document. Modular documents use the XInclude W3C recommendation as a way to specify the referenced documents and document parts.

Modular documents allow you to manage and reuse content. MarkLogic Server includes a Content Processing Framework (CPF) application that expands the documents based on all of the XInclude references. The CPF application creates a new document for the expanded document, leaving the original documents untouched. If any of the parts are updated, the expanded document is recreated, automatically keeping the expanded document up to date.

The CPF application for modular documents takes care of all of the work involved in expanding the documents. All you need to do is add or update documents in the database that have XInclude references, and then anything under a CPF domain is automatically expanded. For details on CPF, see the *Content Processing Framework Guide* guide.

Content can be reused by referencing it in multiple documents. For example, imagine you are a book publisher and you have boilerplate passages such as legal disclaimers, company information, and so on, that you include in many different titles. Each book can then reference the boilerplate documents. If you are using the CPF application, then if the boilerplate is updated, all of the documents are automatically updated. If you are not using the CPF application, you can still update the documents with a simple API call.

14.2 XInclude and XPointer

Modular documents use XInclude and XPointer technologies:

- XInclude: http://www.w3.org/TR/xinclude/
- XPointer: http://www.w3.org/TR/WD-xptr

XInclude provides a syntax for including XML documents within other XML documents. It allows you to specify a relative or absolute URI for the document to include. XPointer provides a syntax for specifying parts of an XML document. It allows you to specify a node in the document using a syntax based on (but not quite the same as) XPath. MarkLogic Server supports the XPointer framework, and the `element()` and `xmlns()` schemes of XPointer, as well as the `xpath()` scheme:

- `element()` Scheme: http://www.w3.org/TR/2002/PR-xptr-element-20021113/
- `xmlns()` Scheme: http://www.w3.org/TR/2002/PR-xptr-xmlns-20021113/
- `xpath()` Scheme, which is not a W3C recommendation, but allows you to use simple XPath to specify parts of a document.

The `xmlns()` scheme is used for namespace prefix bindings in the XPointer framework, the `element()` scheme is one syntax used to specify which elements to select out of the document in the XInclude `href` attribute, and the `xpath()` scheme is an alternate syntax (which looks much more like XPath than the `element()` scheme) to select elements from a document.

Each of these schemes is used within an attribute named `xpointer`. The `xpointer` attribute is an attribute of the `<xi:include>` element. If you specify a string corresponding to an `idref`, then it selects the element with that id attribute, as shown in “Example: Simple id” on page 183.

The examples that follow show XIncludes that use XPointer to select parts of documents:

- Example: Simple id
- Example: xpath() Scheme
- Example: element() Scheme
- Example: xmlns() and xpath() Scheme

14.2.1 Example: Simple id

Given a document `/test2.xml` with the following content:

```
<el-name>
  <p id="myID">This is the first para.</p>
  <p>This is the second para.</p>
</el-name>
```

The following selects the element with an `id` attribute with a value of `myID` from the `/test2.xml` document:

```
<xi:include href="/test2.xml" xpointer="myID" />
```

The expansion of this `<xi:include>` element is as follows:

```
<p id="myID" xml:base="/test2.xml">This is the first para.</p>
```

14.2.2 Example: xpath() Scheme

Given a document `/test2.xml` with the following content:

```
<el-name>
  <p id="myID">This is the first para.</p>
  <p>This is the second para.</p>
</el-name>
```

The following selects the second `p` element that is a child of the root element `el-name` from the `/test2.xml` document:

```
<xi:include href="/test2.xml" xpointer="xpath(/el-name/p[2])" />
```

The expansion of this `<xi:include>` element is as follows:

```
<p xml:base="/test2.xml">This is the second para.</p>
```

14.2.3 Example: element() Scheme

Given a document `/test2.xml` with the following content:

```
<el-name>
  <p id="myID">This is the first para.</p>
  <p>This is the second para.</p>
</el-name>
```

The following selects the second `p` element that is a child of the root element `el-name` from the `/test2.xml` document:

```
<xi:include href="/test2.xml" xpointer="element(/1/2)" />
```

The expansion of this `<xi:include>` element is as follows:

```
<p xml:base="/test2.xml">This is the second para.</p>
```

14.2.4 Example: xmlns() and xpath() Scheme

Given a document `/test2.xml` with the following content:

```
<pref:el-name xmlns:pref="pref-namespace">
  <pref:p id="myID">This is the first para.</pref:p>
  <pref:p>This is the second para.</pref:p>
</pref:el-name>
```

The following selects the first `pref:p` element that is a child of the root element `pref:el-name` from the `/test2.xml` document:

```
<xi:include href="/test2.xml"
            xpointer="xmlns(pref=pref-namespace)
                      xpath(/pref:el-name/pref:p[1])" />
```

The expansion of this `<xi:include>` element is as follows:

```
<pref:p id="myID" xml:base="/test2.xml"
   xmlns:pref="pref-namespace">This is the first para.</pref:p>
```

Note that the namespace prefixes for the XPointer must be entered in an xmlns() scheme; it does not inherit the prefixes from the query context.

14.3 CPF XInclude Application and API

This section describes the XInclude CPF application code and includes the following parts:

- XInclude Code and CPF Pipeline
- Required Security Privileges—xinclude Role

14.3.1 XInclude Code and CPF Pipeline

You can either create your own modular documents application or use the XInclude pipeline in a CPF application. For details on CPF, see the *Content Processing Framework Guide* guide. The following are the XQuery libraries and CPF components used to create modular document applications:

- The XQuery module library `xinclude.xqy`. The key function in this library is the `xinc:node-expand` function, which takes a node and recursively expands any XInclude references, returning the fully expanded node.
- The XQuery module library `xpointer.xqy`.
- The XInclude pipeline and its associated actions.

- You can create custom pipelines based on the XInclude pipeline that use the following `<options>` to the XInclude pipeline. These options control the expansion of XInclude references for documents under the domain to which the pipeline is attached:
  - `<destination-root>` specifies the directory in which the expanded version of documents are saved. This should be a directory path in the database, and the expanded document will be saved to the URI that is the concatenation of this root and the base name of the unexpanded document. For example, if the URI of the unexpanded document is `/mydocs/unexpanded/doc.xml`, and the destination-root is set to `/expanded-docs/`, then this document is expanded into a document with the URI `/expanded-docs/doc.xml`.
  - `<destination-collection>` specifies the collection in which to put the expanded version. You can specify multiple collections by specifying multiple `<destination-collection>` elements in the pipeline.
  - `<destination-quality>` specifies the document quality for the expanded version. This should be an integer value, and higher positive numbers increase the relevance scores for matches against the document, while lower negative numbers decrease the relevance scores. The default quality on a document is 0, which does not change the relevance score.
  - The default is to use the same values as the unexpanded source.

14.3.2 Required Security Privileges—xinclude Role

The XInclude code requires the following privileges:

- `xdmp:with-namespaces`
- `xdmp:value`

Therefore, any users who will be expanding documents require these privileges. There us a predefined role called `xinclude` that has the needed privileges to execute this code. You must either assign the xinclude role to your users or they must have the above execute privileges in order to run the XInclude code used in the XInclude CPF application.

14.4 Creating XML for Use in a Modular Document Application

The basic syntax for using XInclude is relatively simple. For each referenced document, you include an `<xi:include>` element with an `href` attribute that has a value of the referenced document URI, either relative to the document with the `<xi:include>` element or an absolute URI of a document in the database. When the document is expanded, the document referenced replaces the `<xi:include>` element. This section includes the following parts:

- <xi:include> Elements
- <xi:fallback> Elements
- Simple Examples

14.4.1 <xi:include> Elements

Element that have references to content in other documents are `<xi:include>` elements, where `xi` is bound to the `http://www.w3.org/2001/XInclude` namespace. Each `xi:include` element has an `href` attribute, which has the URI of the included document. The URI can be relative to the document containing the `<xi:include>` element or an absolute URI of a document in the database.

14.4.2 <xi:fallback> Elements

The XInclude specification has a mechanism to specify *fallback* content, which is content to use when expanding the document when the XInclude reference is not found. To specify fallback content, you add an `<xi:fallback>` element as a child of the `<xi:include>` element. Fallback content is optional, but it is good practice to specify it. As long as the `xi:include href` attributes resolve correctly, documents without `<xi:fallback>` elements will expand correctly. If an `xi:include href` attribute does not resolve correctly, however, and if there are no `<xi:fallback>` elements for the unresolved references, then the expansion will fail with an `XI-BADFALLBACK` exception.

The following is an example of an `<xi:include>` element with an `<xi:fallback>` element specified:

```
<xi:include href="/blahblah.xml">
  <xi:fallback><p>NOT FOUND</p></xi:fallback>
</xi:include>
```

The `<p>NOT FOUND</p>` will be substituted when expanding the document with this `<xi:include>` element if the document with the URI `/blahblah.xml` is not found.

You can also put an `<xi:include>` element within the `<xi:fallback>` element to fallback to some content that is in the database, as follows:

```
<xi:include href="/blahblah.xml">
  <xi:fallback><xi:include href="/fallback.xml" /></xi:fallback>
</xi:include>
```

The previous element says to include the document with the URI `/blahblah.xml` when expanding the document, and if that is not found, to use the content in `/fallback.xml`.

14.4.3 Simple Examples

The following is a simple example which creates two documents, then expands the one with the XInclude reference:

```
xquery version "1.0-ml";
declare namespace xi="http://www.w3.org/2001/XInclude";

xdmp:document-insert("/test1.xml", <document>
  <p>This is a sample document.</p>
  <xi:include href="test2.xml"/>
</document>);
```

```
xquery version "1.0-ml";

xdmp:document-insert("/test2.xml",
  <p>This document will get inserted where
     the XInclude references it.</p>);

xquery version "1.0-ml";
import module namespace xinc="http://marklogic.com/xinclude"
     at "/MarkLogic/xinclude/xinclude.xqy";

xinc:node-expand(fn:doc("/test1.xml"))
```

The following is the expanded document returned from the `xinc:node-expand` call:

```
<document>
  <p>This is a sample document.</p>
  <p xml:base="/test2.xml">This document will get inserted where
     the XInclude references it.</p>
</document>
```

The base URI from the URI of the included content is added to the expanded node as an `xml:base` attribute.

You can include fallback content as shown in the following example:

```
xquery version "1.0-ml";
declare namespace xi="http://www.w3.org/2001/XInclude";

xdmp:document-insert("/test1.xml", <document>
  <p>This is a sample document.</p>
  <xi:include href="/blahblah.xml">
    <xi:fallback><p>NOT FOUND</p></xi:fallback>
  </xi:include>
</document>);

xquery version "1.0-ml";

xdmp:document-insert("/test2.xml",
  <p>This document will get inserted where the XInclude references
it.</p>);

xquery version "1.0-ml";

xdmp:document-insert("/fallback.xml",
  <p>Sorry, no content found.</p>);

xquery version "1.0-ml";
import module namespace xinc="http://marklogic.com/xinclude"
     at "/MarkLogic/xinclude/xinclude.xqy";

xinc:node-expand(fn:doc("/test1.xml"))
```

The following is the expanded document returned from the `xinc:node-expand` call:

```
<document>
  <p>This is a sample document.</p>
  <p xml:base="/test1.xml">NOT FOUND</p>
</document>
```

14.5 Setting Up a Modular Document Application

To set up a modular documents CPF application, you need to install CPF and create a domain under which documents with XInclude links will be expanded. For detailed information about the Content Processing Framework, including procedures for how to set it up and information about how it works, see the *Content Processing Framework Guide* guide.

To set up an XInclude modular document application, perform the following steps:

1. Install Content Processing in your database, if it is not already installed. For example, if your database is named `modular`, In the Admin Interface click the Databases > modular > Content Processing link. If it is not already installed, the Content Processing Summary page will indicate that it is not installed. If it is not installed, click the Install tab and click install (you can install it with or without enabling conversion).

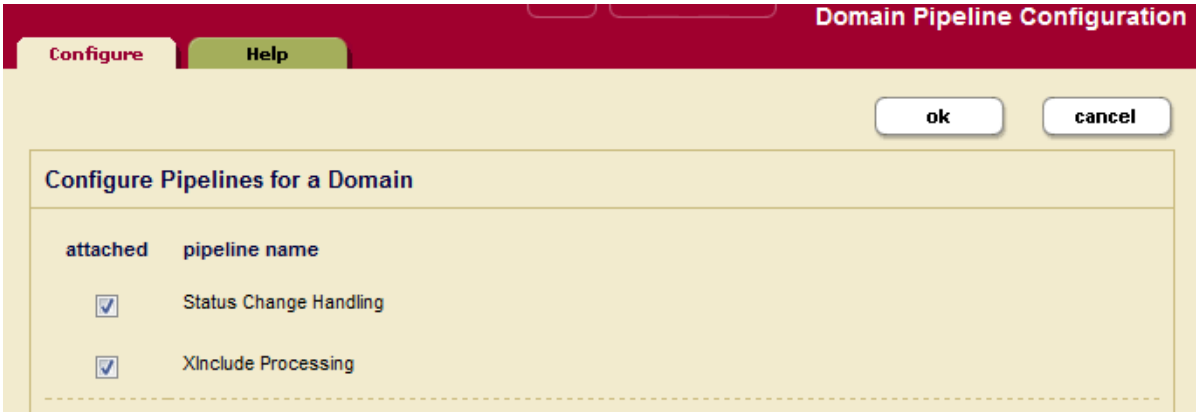
2. Click the domains link from the left tree menu. Either create a new domain or modify an existing domain to encompass the scope of the documents you want processed with the XInclude processing. For details on domains, see the *Content Processing Framework Guide* guide.

3. Under the domain you have chosen, click the Pipelines link from the left tree menu.

4. Check the `Status Change Handling` and `XInclude Processing` pipelines. You can also attach other pipelines or detach other pipelines, depending if they are needed for your application.

   Note: If you want to change any of the `<options>` settings on the `XInclude Processing` pipeline, copy that pipeline to another file, make the changes (make sure to change the value of the `<pipeline-name>` element as well), and load the pipeline XML file. It will then be available to attach to a domain. For details on the options for the XInclude pipeline, see “CPF XInclude Application and API” on page 184.

5. Click OK. The Domain Pipeline Configuration screen shows the attached pipelines.



Any documents with XIncludes that are inserted or updated under your domain will now be expanded. The expanded document will have a URI ending in `_expanded.xml`. For example, if you insert a document with the URI `/test.xml`, the expanded document will be created with a URI of `/test_xml_expanded.xml` (assuming you did not modify the XInclude pipeline options).

Note: If there are existing XInclude documents in the scope of the domain, they will not be expanded until they are updated.

15.0 Controlling App Server Access, Output, and Errors

MarkLogic Server evaluates XQuery programs against App Servers. This chapter describes ways of controlling the output, both by App Server configuration and with XQuery built-in functions. Primarily, the features described in this chapter apply to HTTP App Servers, although some of them are also valid with XDBC Servers and with the Task Server. This chapter contains the following sections:

- Creating Custom HTTP Server Error Pages
- Setting Up URL Rewriting for an HTTP App Server
- Example: A Simple URL Rewriter
- Outputting SGML Entities
- Specifying the Output Encoding
- Specifying Output Options at the App Server Level

15.1 Creating Custom HTTP Server Error Pages

This section describes how to use the HTTP Server error pages and includes the following parts:

- Overview of Custom HTTP Error Handling
- Error Detail
- Configuring Custom Error Handlers
- Execute Permissions Are Needed On Error Handler Document for Modules Databases
- Example: Custom Error Handler

15.1.1 Overview of Custom HTTP Error Handling

A custom HTTP Server error page is a way to redirect application exceptions to an error handler module. When any 400 or 500 HTTP exception is thrown (except for a 503 error), the error handler is evaluated and the results are returned to the client. Custom error pages typically provide more user-friendly messages to the end-user, but because the error page is generated by a code module, you can perform arbitrary work.

You can implement a custom error handler module in either XQuery or Server-Side JavaScript. The language you choose is independent from the language(s) in which you implement your application.

The error handler module can get the HTTP error code and the contents of the HTTP response using the `xdmp:get-response-code` (XQuery) or `xdmp.getResponseCode` (JavaScript) function.

The error handler module also has access to additional error details, including stack trace information, when available. For details, see “Error Detail” on page 192.

If the error is a 503 (unavailable) error, then the error handler is not invoked and the 503 exception is returned to the client.

If the error handler itself throws an exception, that exception is passed to the client with the error code from the error handler. It will also include a stack trace that includes the original error code and exception.

15.1.2 Error Detail

If you implement your error handler in XQuery, MarkLogic makes detailed information about the current error available to the error handler. If your handler is implemented in XQuery, MarkLogic makes the error detail available as an XML element. If the handler is implemented in Server-Side JavaScript object, MarkLogic makes the error detail available as a JSON node. See the following topics for details:

- XML Detail Format
- JavaScript Error Detail Format

15.1.2.1 XML Detail Format

An XQuery error handler receives detailed error information as an XML element node conforming to the `error.xsd` schema. The detail includes any exceptions thrown, line numbers, XQuery version (when appropriate), and a stack trace (when available).

An error handler accesses the error detail through a special `$error:errors` external variable that MarkLogic populates. To access the error details, include a declaration of the following form in your error handler:

```
declare variable $error:errors as node()* external;
```

The following is a sample error detail node, generated by an XQuery module with a syntax error that caused MarkLogic to raise an `XDMP-CONTEXT` exception:

```
<error:error xsi:schemaLocation="http://marklogic.com/xdmp/error
   error.xsd"
   xmlns:error="http://marklogic.com/xdmp/error"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <error:code>XDMP-CONTEXT</error:code>
  <error:name>err:XPDY0002</error:name>
  <error:xquery-version>1.0-ml</error:xquery-version>
  <error:message>Expression depends on the context where none
                 is defined</error:message>
  <error:format-string>XDMP-CONTEXT: (err:XPDY0002) Expression
    depends on the context where none is defined</error:format-string>
  <error:retryable>false</error:retryable>
  <error:expr/> <error:data/>
  <error:stack>
    <error:frame>
     <error:uri>/blaz.xqy</error:uri>
```

```
      <error:line>1</error:line>
      <error:xquery-version>1.0-ml</error:xquery-version>
    </error:frame>
  </error:stack>
</error:error>
```

15.1.2.2 JavaScript Error Detail Format

A Server-Side JavaScript error handler receives detailed error information through a variable named “error” that MarkLogic puts in the global scope. The detail includes any exceptions thrown, line numbers, XQuery version (when appropriate), and a stack trace (when available).

The following is an example error detail object resulting from a JavaScript module that caused MarkLogic to throw an `XDMP-DOCNOTFOUND` exception:

```
{ "code": "XDMP-DOCNOTFOUND",
  "name": "",
  "message": "Document not found",
  "retryable": "false",
  "data": [ ],
  "stack": "XDMP-DOCNOTFOUND:xdmp.documentDelete(\"nonexistent.json\")
      -- Document not found\n
      in /eh-ex/eh-app.sjs, at 3:7, in panic() [javascript]\n
      in /eh-ex/eh-app.sjs, at 6:0 [javascript]\n
      in /eh-ex/eh-app.sjs [javascript]",
  "stackFrames": [
    {
      "uri": "/eh-ex/eh-app.sjs",
      "line": "3",
      "column": "7",
      "operation": "panic()"
    }, {
      "uri": "/eh-ex/eh-app.sjs",
      "line": "6",
      "column": "0"
    }, {
      "uri": "/eh-ex/eh-app.sjs"
    }
  ]
}
```

15.1.3 Configuring Custom Error Handlers

To configure a custom error handler for an HTTP App Server, enter the path to the XQuery or Server-Side JavaScript module in the Error Handler field of an HTTP Server. If the path does not start with a slash (`/`), then it is relative to the App Server root. If it does start with a slash (`/`), then it follows the import rules described in “Importing XQuery Modules, XSLT Stylesheets, and Resolving Paths” on page 89.

15.1.4 Execute Permissions Are Needed On Error Handler Document for Modules Databases

If your App Server is configured to use a modules database (that is, it stores and executes its application modules in a database) then you should put an execute permission on the error handler module document. The execute permission is paired to a role, and all users of the App Server must have that role in order to execute the error handler; if a user does not have the role, then that user will not be able to execute the error handler module, and it will get a 401 (unauthorized) error instead of having the error be caught and handled by the error handler.

As a consequence of needing the execute permission on the error handler, if a user who is actually not authorized to run the error handler attempts to access the App Server, that user runs as the default user configured for the App Server until authentication. If authentication fails, then the error handler is called as the default user, but because that default user does not have permission to execute the error handler, the user is not able to find the error handler and a 404 error (not found) is returned. Therefore, if you want all users (including unauthorized users) to have permission to run the error handler, you should give the default user a role (it does not need to have any privileges on it) and assign an execute permission to the error handler paired with that role.

15.1.5 Example: Custom Error Handler

The following example is a very simple error handler that simply returns all of the error detail.

Language	Example

XQuery	`xquery version "1.0-ml";`
	
	`declare variable $error:errors as node()* external;`
	
	`xdmp:set-response-content-type("text/plain"),`
	`xdmp:get-response-code(),`
	`$error:errors`

Server-Side JavaScript	`var error;`
	`const resp = xdmp.getResponseCode().toArray();`
	
	`let response = 'Code: ' + resp[0] + '\nMessage: ' + resp[1];`
	`if (error != undefined) {`
	`  response += '\n' + error.toString();`
	`}`
	
	`xdmp.setResponseContentType('text/plain');`
	`response;`

In a typical error page, you would use some or all of the information to create a user-friendly representation of the error to display to users. Since you can write arbitrary code in the error handler, you can do a wide variety of things, such as sending an email to the application administrator or redirecting it to a different page.

15.2 Setting Up URL Rewriting for an HTTP App Server

This section describes how to use the HTTP Server URL Rewriter feature. For additional information on URL rewriting, see “Creating an Interpretive XQuery Rewriter to Support REST Web Services” on page 211.

This section includes the following topics:

- Overview of URL Rewriting
- Creating URL Rewrite Modules
- Prohibiting Access to Internal URLs
- URL Rewriting and Page-Relative URLs
- Using the URL Rewrite Trace Event

15.2.1 Overview of URL Rewriting

You can access any MarkLogic Server resource with a URL, which is a fundamental characteristic of Representational State Transfer (REST) services. In its raw form, the URL must either reflect the physical location of the resource (if a document in the database), or it must be of the form:

```
http://<dispatcher-program.xqy>?instructions=foo
```

Users of web applications typically prefer short, neat URLs to raw query string parameters. A concise URL, also referred to as a “clean URL,” is easy to remember, and less time-consuming to type in. If the URL can be made to relate clearly to the content of the page, then errors are less likely to happen. Also crawlers and search engines often use the URL of a web page to determine whether or not to index the URL and the ranking it receives. For example, a search engine may give a better ranking to a well-structured URL such as:

```
http://marklogic.com/technical/features.html
```

than to a less-structured, less-informative URL like the following:

```
http://marklogic.com/document?id=43759
```

In a “RESTful” environment, URLs should be well-structured, predictable, and decoupled from the physical location of a document or program. When an HTTP server receives an HTTP request with a well-structured, external URL, it must be able to transparently map that to the internal URL of a document or program.

The URL Rewriter feature allows you to configure your HTTP App Server to enable the rewriting of external URLs to internal URLs, giving you the flexibility to use any URL to point to any resource (web page, document, XQuery program and arguments). The URL Rewriter implemented by MarkLogic Server operates similarly to the Apache mod_rewrite module, except you write an XQuery or Server-Side JavaScript program to perform the rewrite operation.

The URL rewriting happens through an internal redirect mechanism so the client is not aware of how the URL was rewritten. This makes the inner workings of a web site's address opaque to visitors. The internal URLs can also be blocked or made inaccessible directly if desired by rewriting them to non-existent URLs, as described in “Prohibiting Access to Internal URLs” on page 199.

For an end to end example of a simple rewriter, see “Example: A Simple URL Rewriter” on page 201.

For information about creating a URL rewriter to directly invoke XSLT stylesheets, see Invoking Stylesheets Directly Using the XSLT Rewriter in the *XQuery and XSLT Reference Guide*.

Note: If your application code is in a modules database, the URL rewriter needs to have permissions for the default App Server user (nobody by default) to execute the module. This is the same as with an error handler that is stored in the database, as described in “Execute Permissions Are Needed On Error Handler Document for Modules Databases” on page 194.

15.2.2 Creating URL Rewrite Modules

This section describes how to create simple URL rewrite modules. For more robust URL rewriting solutions, see "Creating an Interpretive XQuery Rewriter to Support REST Web Services" on page 211.

You can implement a rewrite module in XQuery or Server-Side JavaScript. The language you choose for the rewriter implementation is independent of the implementation language of any module the rewriter may redirect to. For example, you can create a JavaScript rewriter that redirects a request to an XQuery application module, and vice versa.

You can use the pattern matching features in regular expressions to create flexible URL rewrite modules. For example, you want the user to only have to enter `/` after the scheme and network location portions of the URL (for example, `http://localhost:8060/`) and have it rewritten as `/app.xqy`:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`let $url := xdmp:get-request-url()`
	`return fn:replace($url,"^/$", "/app.xqy")`

Server-Side JavaScript	`const url = xdmp.getRequestUrl();` `url.replace(/^\/$/, '/app.xqy')`

The following example converts a portion of the original URL into a request parameter of a new dynamic URL:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`let $url := xdmp:get-request-url()`
	`return fn:replace($url,`
	`"^/product-([0-9]+)\.html$",`
	`"/product.xqy?id=$1")`

Server-Side JavaScript	`const url = xdmp.getRequestUrl();` `url.replace(/^\/product-(\d+).html$/, '\/product\?id=$1')`

The product ID can be any number. For example, the URL `/product-12.html` is converted to `/product.xqy?id=12` and `/product-25.html` is converted to `/product.xqy?id=25`.

Search engine optimization experts suggest displaying the main keyword in the URL. In the following URL rewriting technique you can display the name of the product in the URL:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`let $url := xdmp:get-request-url()`
	`return fn:replace($url,`
	`"^/product/([a-zA-Z0-9_-]+)/([0-9]+)\.html$",`
	`"/product.xqy?id=$2")`

Server-Side JavaScript	`const url = xdmp.getRequestUrl();` `url.replace(/^\/product\/([\w\d-]+)\/(\d+).html$/, '\/product\?id=$2')`

The product name can be any string. For example, `/product/canned_beans/12.html` is converted to `/product.xqy?id=12` and `/product/cola_6_pack/8.html` is converted to `/product.xqy?id=8`.

If you need to rewrite multiple pages on your HTTP server, you can create a URL rewrite script like the following:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`let $url := xdmp:get-request-url()`
	`let $url := fn:replace($url, "^/Shrew$", "/tame.xqy")`
	`let $url := fn:replace($url, "^/Macbeth$", "/mac.xqy")`
	`let $url := fn:replace($url, "^/Tempest$", "/tempest.xqy")`
	`return $url`

Server-Side JavaScript	`const url = xdmp.getRequestUrl();`
	`url.replace(/^\/Shrew$/, '/tame.xqy')`
	`.replace(/^\/Macbeth$/, '/mac.xqy')`
	`.replace(/^\/Tempest$/, '/tempest.xqy');`

15.2.3 Prohibiting Access to Internal URLs

The URL Rewriter feature also enables you to block user's from accessing internal URLs. For example, to prohibit direct access to `customer_list.html`, your URL rewrite script might look like the following:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`let $url := xdmp:get-request-url()`
	`return`
	`  if (fn:matches($url,"^/customer_list.html$"))`
	`  then "/nowhere.html"`
	`  else fn:replace($url,"^/price_list.html$", "/prices.html")`

Server-Side JavaScript	`const url = xdmp.getRequestUrl();`
	`url.match(/^\/customer_list.html/)`
	`  ? 'nowhere.html'`
	`  : url.replace(/^\/price_list.html$/, 'prices.html')`

Where `/nowhere.html` is a non-existent page for which the browser returns a "404 Not Found" error. Alternatively, you could redirect to a URL consisting of a random number generated using `xdmp:random` (XQuery) or xdmp.random (JavaScript), or some other scheme guaranteed to generate non-existent URLs.

15.2.4 URL Rewriting and Page-Relative URLs

You may encounter problems when rewriting a URL to a page that makes use of page-relative URLs because relative URLs are resolved by the client. If the directory path of the external URL used by the client differs from the internal URL at the server, then the page-relative links are incorrectly resolved.

If you are going to rewrite a URL to a page that uses page-relative URLs, convert the page-relative URLs to server-relative or canonical URLs. For example, if your application is located in `C:\Program Files\MarkLogic\myapp` and the page builds a frameset with page-relative URLs, like:

```
<frame src="top.html" name="headerFrame">
```

You should change the URLs to server-relative:

```
<frame src="/myapp/top.html" name="headerFrame">
```

or canonical:

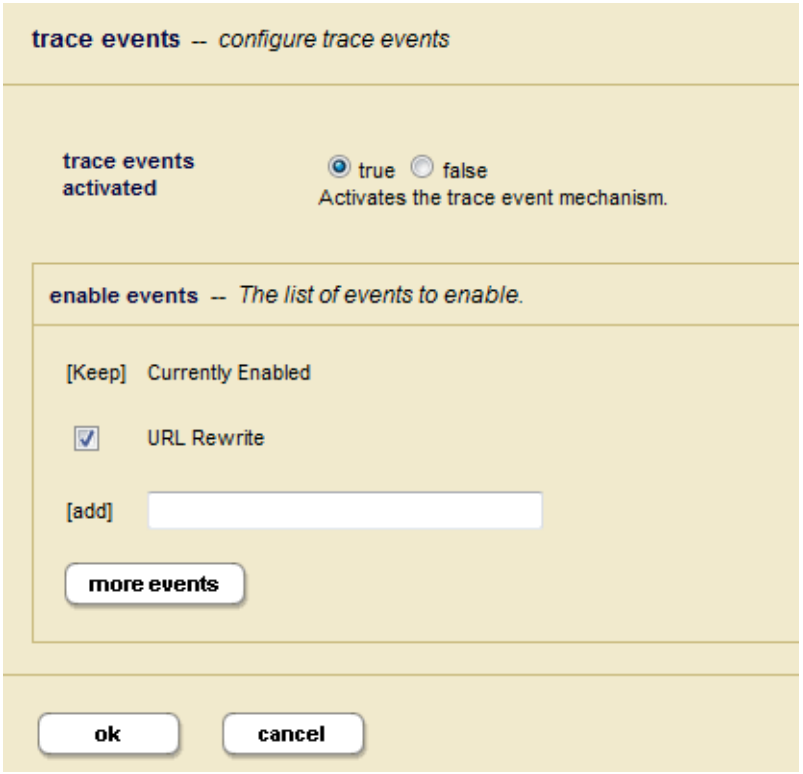
```
<frame src="http://127.0.0.1:8000/myapp/top.html" name="headerFrame">
```

15.2.5 Using the URL Rewrite Trace Event

You can use the URL Rewrite trace event to help you debug your URL rewrite modules. To use the URL Rewrite trace event, you must enable tracing (at the group level) for your configuration and set the event:

1. Log into the Admin Interface.
2. Select Groups > *group_name* > Diagnostics.

   The Diagnostics Configuration page appears.
3. Click the `true` button for `trace events activated`.
4. In the [add] field, enter: `URL Rewrite`
5. Click the OK button to activate the event.



After you configure the URL Rewrite trace event, when any URL Rewrite script is invoked, a line, like that shown below, is added to the `ErrorLog.txt` file, indicating the URL received from the client and the converted URL from the URL rewriter:

```
2009-02-11 12:06:32.587 Info: [Event:id=URL Rewrite] Rewriting URL
/Shakespeare to /frames.html
```

Note: The trace events are designed as development and debugging tools, and they might slow the overall performance of MarkLogic Server. Also, enabling many trace events will produce a large quantity of messages, especially if you are processing a high volume of documents. When you are not debugging, disable the trace event for maximum performance.

15.3 Example: A Simple URL Rewriter

This example walks you through creating a simple URL rewriter that enables you to use an "intuitive" URL to serve an XML document out of a MarkLogic database. The request for the documents is serviced by an example application module installed in MarkLogic. The example rewriter rewrites the external URL to reference the application module, internally.

Follow these steps to run the example:

- Create the Example App Server
- Install the Example Content
- Install the Example Application Module
- Exercise the Example Application
- Install the Rewriter
- Configure the App Server to Use the Rewriter
- Exercise the Rewriter

15.3.1 Create the Example App Server

This example requires you to create an HTTP App Server on which to exercise the sample rewriter. Do not run this example on the default port 8000 App Server as that App Server uses a special purpose MarkLogic rewriter.

The example assumes the existence of an HTTP App Server with the following characteristics. If you choose to use different settings, you will need to modify the subsequent instructions to match. For instructions on creating an HTTP App Server, see Creating a New HTTP Server in the *Administrator's Guide*.

Setting	Recommended Value
server name	rewriter-ex
root	/
port	8020
modules	Modules
database	Documents

Accept the defaults for all other configuration settings.

15.3.2 Install the Example Content

Run the following code in Query Console to insert the example document in the content database of your HTTP App Server (Modules).

Before running the code, set the Database to **Documents** and the Query Type as appropriate Query Console.

Language	Example

XQuery	`(: Insert example content to fetch using example app :)`
	`xquery version "1.0-ml";`
	`xdmp:document-insert('rewriter-ex.xml',`
	`  <example>This is the rewriter example document.</example>)`

Server-Side JavaScript	`// Insert example content to fetch using example app :)`
	`declareUpdate();`
	`xdmp.documentInsert('rewriter-ex.xml',`
	`  new NodeBuilder()`
	`  .addElement('example', 'This is the rewriter example document.')`
	`  .toNode());`

15.3.3 Install the Example Application Module

Run the following code in Query Console to insert the example application module into the modules database of your App Server (Modules).

Before running the code, set the Database to **Modules** and the Query Type as appropriate in Query Console.

Language	Example

XQuery	`(: Insert example app module with URI /rewriter-ex/app.xqy :)`
	`xquery version "1.0-ml";`
	`xdmp:document-insert('/rewriter-ex/app.xqy',`
	`  text {'xquery version "1.0-ml";`
	`   xdmp:set-response-content-type("text/xml"),`
	`   fn:doc("rewriter-ex.xml")'}`
	`)`

Server-Side JavaScript	`// Insert example app module with URI /rewriter-ex/app.sjs`
	`declareUpdate();`
	`xdmp.documentInsert('/rewriter-ex/app.sjs',`
	`  new NodeBuilder().addText(`
	`    'xdmp.setResponseContentType("text/xml");cts.doc("rewriter-ex.xml");'`
	`  ).toNode()`
	`);`

15.3.4 Exercise the Example Application

Use this step to confirm that the example application is properly installed.

If you used the XQuery example app, navigate to the following URL, assuming MarkLogic is installed on localhost:

```
http://localhost:8020/rewriter-ex/app.xqy
```

If you used the Server-Side JavaScript example app, navigate to the following URL, assuming MarkLogic is installed on localhost:

```
http://localhost:8020/rewriter-ex/app.sjs
```

The example document from “Install the Example Content” on page 202 should appear. If you get a 404 (Page Not Found) error, use Query Console to confirm that you correctly installed the example application module in the Modules database, and not in the Documents database.

15.3.5 Install the Rewriter

This step inserts an example rewriter into the modules database associated with your App Server (Modules). The example rewriter intercepts the inbound URL and use the `replace` function to change the request path to point to the example app module.

Run the following code in Query Console to insert the rewriter into the modules database. Set the Database to **Modules** and the Query Type as appropriate in Query Console.

Language	Example

XQuery	`(: Insert rewriter with URI /rewriter-ex/rewriter.xqy :)`
	`xquery version "1.0-ml";`
	`xdmp:document-insert('/rewriter-ex/rewriter.xqy',`
	`  text {'xquery version "1.0-ml";`
	`   fn:replace(`
	`     xdmp:get-request-url(),`
	`     "^/test-rewriter$","/rewriter-ex/app.xqy")'}`
	`)`

Server-Side JavaScript	`// Insert rewriter with URI /rewriter-ex/rewriter.sjs`
	`declareUpdate();`
	`const rewriter =`
	`  'const url = xdmp.getRequestUrl();' +`
	`  'url.replace(/^\\/test-rewriter$/, \'rewriter-ex/app.sjs\')';`
	`xdmp.documentInsert('/rewriter-ex/rewriter.sjs',`
	`  new NodeBuilder().addText(rewriter).toNode()`
	`);`

The example rewriter uses `xdmp:get-request-url` in XQuery and `xdmp.getRequestUrl` in JavaScript to access the portion of the URL following the scheme and network location (domain name or *host_name*:*port_number*). For example, if the original request URL is `http://localhost:8020/test-rewriter`, this function returns `/test-rewriter`.

Note that this `xdmp:get-request-rule` and `xdmp.getRequestUrl` also return any request parameters (fields). You rewriter can modify the request parameters. For example, you could add a parameter, changing the URL to `test-rewriter/someparam=value`. If you just want the request path (`/test-rewriter`, here), you can use `xdmp:get-request-path` (XQuery) or `xdmp.getRequestPath` (JavaScript).

You can create more elaborate URL rewrite modules, as described in “Creating URL Rewrite Modules” on page 197 and “Creating an Interpretive XQuery Rewriter to Support REST Web Services” on page 211.

15.3.6 Configure the App Server to Use the Rewriter

Now that you have installed the rewriter module, you can change the App Server configuration to reference it.

In the Admin Interface, go to the configuration page for the rewriter-ex App Server you created in “Create the Example App Server” on page 201.

Find the `url rewriter` configuration setting. Set the rewriter to one of the following paths, depending on whether you're using the XQuery or JavaScript example rewriter:

- `XQuery: /rewriter-ex/rewriter.xqy`
- `JavaScript: /rewriter-ex/rewriter.sjs`

Click OK at the top or bottom of the App Server configuration page to save your change.

You can also configure the rewriter for an App Server using the Admin library function `admin:appserver-set-url-rewriter`, or the the REST Management API.

15.3.7 Exercise the Rewriter

In your browser, navigate to the following URL:

```
http://localhost:8020/test-rewriter
```

Your request should return the same test document as when you queried the example application directly using `http://localhost:8020/rewriter-ex/rewriter.xqy`. or `http://localhost:8020/rewriter-ex/rewriter.sjs` in "Exercise the Example Application" on page 203.

Notice that the URL displayed in the browser remains `http://localhost:8020/test-rewriter`, even though it has been internally rewritten to `http://localhost:8020/rewriter-ex/app.xqy` (or `http://localhost:8020/rewriter-ex/app.sjs`, depending on your implementation language of choice).

15.4 Outputting SGML Entities

This section describes the SGML entity output controls in MarkLogic Server, and includes the following parts:

- Understanding the Different SGML Mapping Settings
- Configuring SGML Mapping in the App Server Configuration
- Specifying SGML Mapping in an XQuery Program

15.4.1 Understanding the Different SGML Mapping Settings

An SGML character entity is a name separated by an ampersand (`&`) character at the beginning and a semi-colon (`;`) character at the end. The entity maps to a particular character. This markup is used in SGML, and sometimes is carried over to XML. MarkLogic Server allows you to control if SGML character entities upon serialization of XML on output, either at the App Server level using the Output SGML Character Entites drop down list or using the `<output-sgml-character-entities>` option to the built-in functions `xdmp:quote` or `xdmp:save`. When SGML characters are mapped (for an App Server or with the built-in functions), any unicode characters that have an SGML mapping will be output as the corresponding SGML entity. The default is `none`, which does not output any characters as SGML entites.

The mappings are based on the W3C XML Entities for Characters specification:

- http://www.w3.org/TR/2008/WD-xml-entity-names-20080721/

with the following modifications to the specification:

- Entities that map to multiple codepoints are not output, unless there is an alternate single-codepoint mapping available. Most of these entities are negated mathematical symbols (`nrarrw` from `isoamsa` is an example).
- The `gcedil` set is also included (it is not included in the specification).

The following table describes the different SGML character mapping settings:

SGML Character Mapping Setting	Description
`none`	The default. No SGML entity mapping is performed on the output.
`normal`	Converts unicode codepoints to SGML entities on output. The conversions are made in the default order. The only difference between `normal` and the `math` and `pub` settings is the order that it chooses to map entities, which only affects the mapping of entities where there are multiple entities mapped to a particular codepoint.
`math`	Converts unicode codepoints to SGML entities on output. The conversions are made in an order that favors math-related entities. The only difference between `math` and the `normal` and `pub` settings is the order that it chooses to map entities, which only affects the mapping of entities where there are multiple entities mapped to a particular codepoint.
`pub`	Converts unicode codepoints to SGML entities on output. The conversions are made in an order favoring entities commonly used by publishers. The only difference between `pub` and the `normal` and `math` settings is the order that it chooses to map entities, which only affects the mapping of entities where there are multiple entities mapped to a particular codepoint.

Note: In general, the `<repair>full</repair>` option on `xdmp:document-load` and the `"repair-full"` option on `xdmp:unquote` do the opposite of the Output SGML Character Entites settings, as the ingestion APIs map SGML entities to their codepoint equivalents (one or more codepoints). The difference with the output options is that the output options perform only single-codepoint to entity mapping, not multiple codepoint to entity mapping.

15.4.2 Configuring SGML Mapping in the App Server Configuration

To configure SGML output mapping for an App Server, perform the following steps:

1. In the Admin Interface, navigate to the App Server you want to configure (for example, Groups > Default > App Servers > MyAppServer).

2. Select the Output Options page from the left tree menu. The Output Options Configuration page appears.

3. Locate the Output SGML Entity Characters drop list (it is towards the top).

4. Select the setting you want. The settings are described in the table in the previous section.

5. Click OK.

Codepoints that map to an SGML entity will now be serialized as the entity by default for requests against this App Server.

15.4.3 Specifying SGML Mapping in an XQuery Program

You can specify SGML mappings for XML output in an XQuery program using the `<output-sgml-character-entities>` option to the following XML-serializing APIs:

- `xdmp:quote`
- `xdmp:save`

For details, see the *MarkLogic XQuery and XSLT Function Reference* for these functions.

15.5 Specifying the Output Encoding

By default, MarkLogic Server outputs content in utf-8. You can specify a different output encodings, both on an App Server basis and on a per-query basis. This section describes those techniques, and includes the following parts:

- Configuring App Server Output Encoding Setting
- XQuery Built-In For Specifying the Output Encoding

15.5.1 Configuring App Server Output Encoding Setting

You can set the output encoding for an App Server using the Admin Interface or with the Admin API. You can set it to any supported character set (see Collations and Character Sets By Language in the Encodings and Collations chapter of the *Search Developer's Guide*).

To configure output encoding for an App Server using the Admin Interface, perform the following steps:

1. In the Admin Interface, navigate to the App Server you want to configure (for example, Groups > Default > App Servers > MyAppServer).

2. Select the Output Options page from the left tree menu. The Output Options Configuration page appears.

3. Locate the Output Encoding drop list (it is towards the top).

4. Select the encoding you want. The settings correspond to different languages, as described in the table in Collations and Character Sets By Language in the Encodings and Collations chapter of the *Search Developer's Guide*.

5. Click OK.

By default, queries against this App Server will now be output in the specified encoding.

15.5.2 XQuery Built-In For Specifying the Output Encoding

Use the following built-in functions to get and set the output encoding on a per-request basis:

- `xdmp:get-response-encoding`
- `xdmp:set-response-encoding`

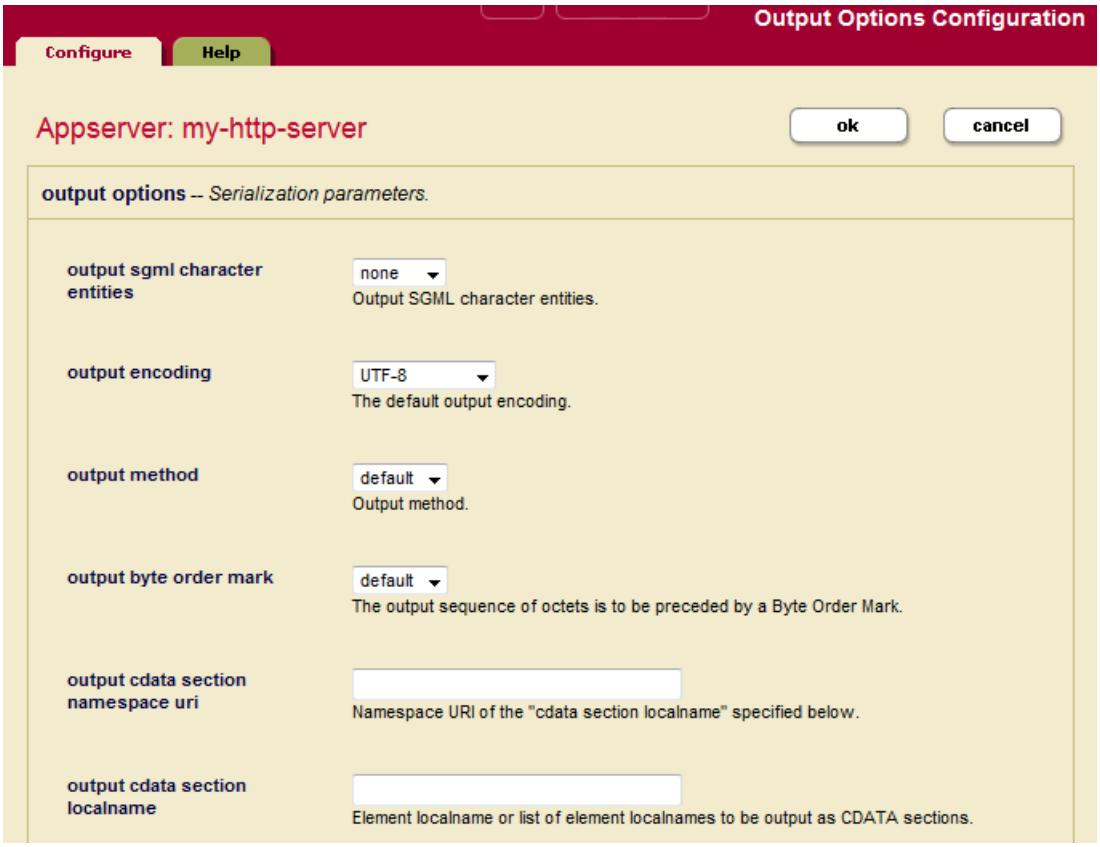
Additionally, you can specify the output encoding for XML output in an XQuery program using the `<output-encoding>` option to the following XML-serializing APIs:

- `xdmp:quote`
- `xdmp:save`

For details, see the *MarkLogic XQuery and XSLT Function Reference* for these functions.

15.6 Specifying Output Options at the App Server Level

You can specify defaults for an array of output options using the Admin Interface. Each App Server has an Output Options Configuration page.



This configuration page allows you to specify defaults that correspond to the XSLT output options (http://www.w3.org/TR/xslt20#serialization) as well as some MarkLogic-specific options. For details on these options, see xdmp:output in the *XQuery and XSLT Reference Guide*. For details on configuring default options for an App Server, see Setting Output Options for an HTTP Server in the *Administrator's Guide*.

16.0 Creating an Interpretive XQuery Rewriter to Support REST Web Services

The REST Library enables you to create RESTful functions that are independent of the language used in applications.

> **Note:** The procedures in this chapter assume you performed the steps described in "Preparing to Run the Examples" on page 237.

The topics in this section are:

- Terms Used in this Chapter
- Overview of the REST Library
- A Simple XQuery Rewriter and Endpoint
- Notes About Rewriter Match Criteria
- The options Node
- Validating options Node Elements
- Extracting Multiple Components from a URL
- Handling Errors
- Handling Redirects
- Handling HTTP Verbs
- Defining Parameters
- Adding Conditions

16.1 Terms Used in this Chapter

- *REST* stands for *Representational State Transfer*, which is an architecture style that, in the context of monitoring MarkLogic Server, describes the use of HTTP to make calls between a monitoring application and monitor host.
- A *Rewriter* interprets the URL of the incoming request and rewrites it to an internal URL that services the request. A rewriter can be implemented as an XQuery module as described in this chapter, or as an XML file as described in "Creating a Declarative XML Rewriter to Support REST Web Services" on page 241.
- An *Endpoint* is an XQuery module on MarkLogic Server that is invoked by and responds to an HTTP request.

16.2 Overview of the REST Library

The REST Library consists of a set of XQuery functions that support URL rewriting and endpoint validation and a MarkLogic REST vocabulary that simplifies the task of describing web service endpoints. The REST vocabulary is used to write declarative descriptions of the endpoints. These descriptions include the mapping of URL parts to parameters and conditions that must be met in order for the incoming request to be mapped to an endpoint.

The REST Library contains functions that simplify:

- Creating a URL rewriter for mapping incoming requests to endpoints
- Validating that applications requesting resources have the necessary access privileges
- Validating that incoming requests can be handled by the endpoints
- Reporting errors

The REST vocabulary allows you to use same description for both the rewriter and the endpoint.

When you have enabled RESTful access to MarkLogic Server resources, applications access these resources by means of a URL that invokes an endpoint module on the target MarkLogic Server host.

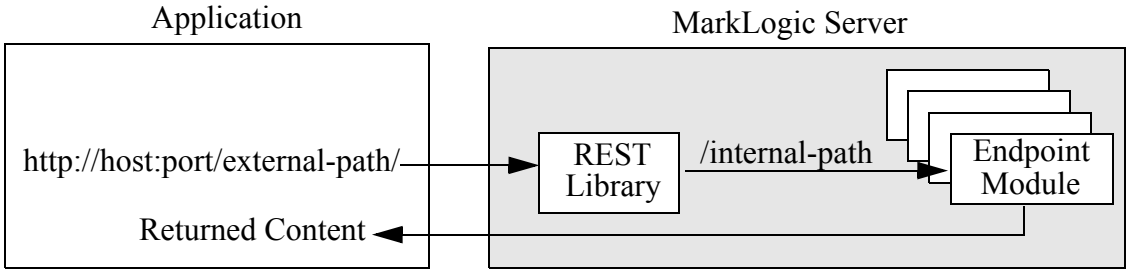
The REST library does the following:

1. Validates the incoming HTTP request.

2. Authorizes the user.

3. Rewrites the resource path to one understood internally by the server before invoking the endpoint module.

If the request is valid, the endpoint module executes the requested operation and returns any data to the application. Otherwise, the endpoint module returns an error message.



Note: The API signatures for the REST Library are documented in the *MarkLogic XQuery and XSLT Function Reference*. For additional information on URL rewriting, see "Setting Up URL Rewriting for an HTTP App Server" on page 195.

16.3 A Simple XQuery Rewriter and Endpoint

This section describes a simple rewriter script that calls a single endpoint.

Navigate to the `/<MarkLogic_Root>/bill` directory and create the following files with the described content.

Create a module, named `requests.xqy`, with the following content:

```
xquery version "1.0-ml";

module namespace
    requests="http://marklogic.com/appservices/requests";

import module namespace rest = "http://marklogic.com/appservices/rest"
    at "/MarkLogic/appservices/utils/rest.xqy";

declare variable $requests:options as element(rest:options) :=
  <options xmlns="http://marklogic.com/appservices/rest">
    <request uri="^/(.+)$" endpoint="/endpoint.xqy">
      <uri-param name="play">$1.xml</uri-param>
    </request>
  </options>;
```

Create a module, named `url_rewriter.xqy`, with the following content:

```
xquery version "1.0-ml";

import module namespace rest = "http://marklogic.com/appservices/rest"
     at "/MarkLogic/appservices/utils/rest.xqy";

import module namespace requests =
  "http://marklogic.com/appservices/requests"
     at "requests.xqy";

rest:rewrite($requests:options)
```

Create a module, named `endpoint.xqy`, with the following content:

```
xquery version "1.0-ml";

import module namespace rest = "http://marklogic.com/appservices/rest"
    at "/MarkLogic/appservices/utils/rest.xqy";

import module namespace requests =
  "http://marklogic.com/appservices/requests" at "requests.xqy";

let $request := $requests:options/rest:request
                  [@endpoint = "/endpoint.xqy"][1]

let $map  := rest:process-request($request)
let $play := map:get($map, "play")

return
  fn:doc($play)
```

Enter the following URL, which uses the `bill` App Server created in “Preparing to Run the Examples” on page 237:

```
http://localhost:8060/macbeth
```

The `rest:rewrite` function in the rewriter uses an `options` node to map the incoming request to an endpoint. The `options` node includes a `request` element with a `uri` attribute that specifies a regular expression and an `endpoint` attribute that specifies the endpoint module to invoke in the event the URL of an incoming request matches the regular expression. In the event of a match, the portion of the URL that matches `(.+)` is bound to the `$1` variable. The `uri-param` element in the `request` element assigns the value of the `$1` variable, along with the `.xml` extension, to the `play` parameter.

```
<rest:options xmlns="http://marklogic.com/appservices/rest">
  <rest:request uri="^/(.+)" endpoint="/endpoint.xqy">
    <uri-param name="play">$1.xml</uri-param>
  </rest:request>
</rest:options>
```

In the example rewriter module above, this `options` node is passed to the `rest:rewrite` function, which outputs a URL that calls the endpoint module with the parameter `play=macbeth.xml`:

```
/endpoint.xqy?play=macbeth.xml
```

The `rest:process-request` function in the endpoint locates the first `request` element associated with the `endpoint.xqy` module and uses it to identify the parameters defined by the rewriter. In this example, there is only a single parameter, `play`, but for reasons described in "Extracting Multiple Components from a URL" on page 220, when there are multiple `request` elements for the same endpoint, then the `request` element that extracts the greatest number of parameters from a URL should be listed in the `options` node ahead of those that extract fewer parameters.

```
let $request := $requests:options/rest:request
                  [@endpoint = "/endpoint.xqy"][1]
```

The `rest:process-request` function in the endpoint uses the `request` element to parse the incoming request and return a map that contains all of the parameters as typed values. The `map:get` function extracts each parameter from the map, which is only one in this example.

```
let $map  := rest:process-request($request)
let $play := map:get($map, "play")
```

16.4 Notes About Rewriter Match Criteria

The default behavior for the rewriter is to match the request against all of the criteria: URI, accept headers, content-type, conditions, method, and parameters. This assures that no endpoint will ever be called except in circumstances that perfectly match what is expected. It can sometimes, however, lead to somewhat confusing results. Consider the following request node:

```
<request uri="/path/to/resource" endpoint="/endpoint.xqy">
  <param name="limit" as="decimal"/>
</request>
```

An incoming request of the form:

```
/path/to/resource?limit=test
```

does not match that request node (because the limit is not a decimal). If there are no other request nodes which match, then the request will return 404 (not found).

That may be surprising. Using additional request nodes to match more liberally is one way to address this problem. However, as the number and complexity of the requests grows, it may become less attractive. Instead, the rewriter can be instructed to match only on specific parts of the request. In this way, error handling can be addressed by the called module.

The match criteria are specified in the call to `rest:rewrite`. For example:

```
rest:rewrite($options, ("uri", "method"))
```

In this case, only the URI and HTTP method will be used for the purpose of matching.

The criteria allowed are:

- `uri` = match on the URI
- `accept` = match on the accept headers
- `content-type` = match on the content type
- `conditions` = match on the conditions
- `method` = match on the HTTP method
- `params` = match on the params

The request must match all of the criteria specified in order to be considered a match.

16.5 The options Node

The REST Library uses an `options` node to map incoming requests to endpoints. The `options` node must be declared as type `element(rest:options)` and must be in the `http://marklogic.com/appservices/rest` namespace.

Below is a declaration of a very simple options node:

```
declare variable $options as element(rest:options) :=
  <rest:options xmlns="http://marklogic.com/appservices/rest">
    <rest:request uri="^/(.+)" endpoint="/endpoint.xqy">
      <uri-param name="play">$1.xml</uri-param>
    </rest:request>
  </rest:options>;
```

The table below summarizes all of the possible elements and attributes in an `options` node. On the left are the elements that have attributes and/or child elements. Attributes for an element are listed in the Attributes column. Attributes are optional, unless designated as '(required)'. Any first-level elements of the element listed on the left are listed in the Child Elements column. The difference between the `user-params="allow"` and `user-params="allow-dups"` attribute values is that `allow` permits a single parameter for a given name, and `allow-dups` permits multiple parameters for a given name.

Element	Attributes	Child Elements	Number of Children	For More Information

`options`	`user-params = ignore`
	`| allow`
	`| allow-dups`
	`| forbid`
		`request`	`0..n`	"A Simple XQuery Rewriter and Endpoint" on page 213

`request`	`uri=`*`string`*
	
	`endpoint=`*`string`*
	
	`user-params = ignore`
	`| allow`
	`| allow-dups`
	`| forbid`
		`uri-param`
		
		`param`
		
		`http`
		
		`auth`
		
		`function`
		
		`accept`
		
		`user-agent`
		
		`and`
		
		`or`
			`0..n`
			
			`0..n`
			
			`0..n`
			
			`0..n`
			
			`0..n`
			
			`0..n`
			
			`0..n`
			
			`0..n`
			
			`0..n`
				"A Simple XQuery Rewriter and Endpoint" on page 213
				
				"Extracting Multiple Components from a URL" on page 220
				
				"Adding Conditions" on page 234

Element	Attributes	Child Elements	Number of Children	For More Information
`uri-param`	`name=`*`string`* `(required)` `as=`*`string`*			"Extracting Multiple Components from a URL" on page 220 "Defining Parameters" on page 228

`param`	`name=`*`string`* `(required)`
	
	`as=`*`string`*
	
	`values=`*`string`*
	
	`match=`*`string`*
	
	`default=`*`string`*
	
	`required = true`
	`| false`
	
	`repeatable = true`
	`| false`
	
	`pattern = <regex>`
				"Defining Parameters" on page 228
				
				"Supporting Parameters Specified in a URL" on page 229.
				
				"Matching Regular Expressions in Parameters with the match and pattern Attributes" on page 232

`http`	`method = string (required)`
	
	`user-params = ignore`
	`| allow`
	`| allow-dups`
	`| forbid`
		`param`
		
		`auth`
		
		`function`
		
		`accept`
		
		`user-agent`
		
		`and`
		
		`or`
			`0..n`
			
			`0..n`
			
			`0..n`
			
			`0..n`
			
			`0..n`
			
			`0..n`
			
			`0..n`
				"Handling HTTP Verbs" on page 224
				
				"Adding Conditions" on page 234.

`auth`		`privilege`
		
		`kind`
			`0..n`
			
			`0..n`
				"Authentication Condition" on page 235

`function`	`ns=`*`string`* `(required)`
	
	`apply=`*`string`* `(required)`
	
	`at=`*`string`* `(required)`
				"Function Condition" on page 236

16.6 Validating options Node Elements

You can use the `rest:check-options` function to validate an `options` node against the REST schema. For example, to validate the `options` node defined in the `requests.xqy` module described in "A Simple XQuery Rewriter and Endpoint" on page 213, you would do the following:

```
xquery version "1.0-ml";

import module namespace rest = "http://marklogic.com/appservices/rest"
    at "/MarkLogic/appservices/utils/rest.xqy";

import module namespace requests =
  "http://marklogic.com/appservices/requests" at "requests.xqy";

rest:check-options($requests:options)
```

An empty sequence is returned if the `options` node is valid. Otherwise an error is returned.

You can also use the `rest:check-request` function to validate `request` elements in an `options` node. For example, to validate all of the `request` elements in the `options` node defined in the `requests.xqy` module described in "A Simple XQuery Rewriter and Endpoint" on page 213, you would do the following:

```
xquery version "1.0-ml";

import module namespace rest = "http://marklogic.com/appservices/rest"
    at "/MarkLogic/appservices/utils/rest.xqy";

declare option xdmp:mapping "false";

rest:check-request($requests:options/rest:request)
```

An empty sequence is returned if the `request` elements are valid. Otherwise an error is returned.

Note: Before calling the `rest:check-request` function, you must set `xdmp:mapping` to `false` to disable function mapping.

16.7 Extracting Multiple Components from a URL

An `options` node may include one or more `request` elements, each of which may contain one or more `uri-param` elements that assign parameters to parts of the request URL. The purpose of each `request` element is to detect a particular URL pattern and then call an endpoint with one or more parameters. Extracting multiple components from a URL is simply a matter of defining a `request` element with a regular expression that recognizes particular URL pattern and then binding the URL parts of interest to variables.

For example, you want expand the capability of the rewriter described in "A Simple XQuery Rewriter and Endpoint" on page 213 and add the ability to use a URL like the one below to display an individual act in a Shakespeare play:

```
http://localhost:8060/macbeth/act3
```

The `options` node in `requests.xqy` might look like the one below, which contains two `request` elements. The rewriter employs a "first-match" rule, which means that it tries to match the incoming URL to the `request` elements in the order they are listed and selects the first one containing a regular expression that matches the URL. In the example below, if an act is specified in the URL, the rewriter uses the first `request` element. If only a play is specified in the URL, there is no match in the first `request` element, but there is in the second `request` element.

> **Note:** The default parameter type is string. Non-string parameters must be explicitly typed, as shown for the `act` parameter below. For more information on typing parameters, see "Parameter Types" on page 229.

```
<options>
  <request uri="^/(.+)/act(\d+)$" endpoint="/endpoint.xqy">
    <uri-param name="play">$1.xml</uri-param>
    <uri-param name="act" as="integer">$2</uri-param>
  </request>
  <request uri="^/(.+)/?$" endpoint="/endpoint.xqy">
    <uri-param name="play">$1.xml</uri-param>
  </request>
</options>
```

When an act is specified in the incoming URL, the first `request` element binds `macbeth` and `3` to the variables `$1` and `$2`, respectively, and then assigns them to the parameters named, `play` and `act`. The URL rewritten by the `rest:rewrite` function looks like:

```
/endpoint.xqy?play=macbeth.xml&act=3
```

The following is an example endpoint module that can be invoked by a rewriter that uses the `options` node shown above. As described in "A Simple XQuery Rewriter and Endpoint" on page 213, the `rest:process-request` function in the endpoint uses the `request` element to parse the incoming request and return a map that contains all of the parameters as typed values. Each parameter is then extracted from the map by means of a `map:get` function. If the URL that invokes this endpoint does not include the `act` parameter, the value of the `$num` variable will be an empty sequence.

Note: The first `request` element that calls the `endpoint.xqy` module is used in this example because, based on the first-match rule, this element is the one that supports both the `play` and `act` parameters.

```
xquery version "1.0-ml";

import module namespace rest = "http://marklogic.com/appservices/rest"
    at "/MarkLogic/appservices/utils/rest.xqy";

import module namespace requests =
  "http://marklogic.com/appservices/requests" at "requests.xqy";

  let $request := $requests:options/rest:request
                    [@endpoint = "/endpoint.xqy"][1]

  let $map  := rest:process-request($request)
  let $play := map:get($map, "play")
  let $num  := map:get($map, "act")

return
  if (empty($num))
  then
    fn:doc($play)
  else
    fn:doc($play)/PLAY/ACT[$num]
```

16.8 Handling Errors

The REST endpoint library includes a `rest:report-error` function that performs a simple translation of MarkLogic Server error markup to HTML. You can invoke it in your modules to report errors:

```
try {
  let $params := rest:process-request($request)
  return
    ...the non-error case...
} catch ($e) {
  rest:report-error($e)
}
```

If the user agent making the request accepts text/html, a simple HTML-formatted response is returned. Otherwise, it returns the raw error XML.

You can also use this function in an error handler to process all of the errors for a particular application.

16.9 Handling Redirects

As shown in the previous sections of this chapter, the URL rewriter translates the requested URL into a new URL for dispatching within the server. The user agent making the request is totally unaware of this translation. As REST Librarys mature and expand, it is sometimes useful to use redirection to respond to a request by telling the user agent to reissue the request at a new URL.

For example, previous users accessed the macbeth play using the following URL pattern:

```
http://localhost:8060/Shakespeare/macbeth
```

You want to redirect the URL to:

```
http://localhost:8060/macbeth
```

The user can tell that this redirection happened because the URL in the browser address bar changes from the old URL to the new URL, which can then be bookmarked by the user.

You can support such redirects by adding a `redirect.xqy` module like this one to your application:

```
xquery version "1.0-ml";

import module namespace rest="http://marklogic.com/appservices/rest"
     at "/MarkLogic/appservices/utils/rest.xqy";

import module namespace requests =
  "http://marklogic.com/appservices/requests" at "requests.xqy";

(: Process requests to be handled by this endpoint module. :)
let $request := $requests:options/rest:request
                    [@endpoint = "/redirect.xqy"][1]

let $params  := rest:process-request($request)

(: Get parameter/value map from request. :)
let $query  := fn:string-join(
    for $param in map:keys($params)
    where $param != "__ml_redirect__"
    return
      for $value in map:get($params, $param)
      return
        fn:concat($param, "=", fn:string($value)),
  "&amp;")

(: Return the name of the play along with any parameters. :)
let $ruri   := fn:concat(map:get($params, "__ml_redirect__"),
                     if ($query = "") then ""
                     else fn:concat("?", $query))

(: Set response code and redirect to new URL. :)
return
  (xdmp:set-response-code(301, "Moved permanently"),
   xdmp:redirect-response($ruri))
```

In the `options` node in `requests.xqy`, add the following request elements to perform the redirect:

```
<request uri="^/shakespeare/(.+)/(.+)" endpoint="/redirect.xqy">
  <uri-param name="__ml_redirect__">/$1/$2</uri-param>
</request>
<request uri="^/shakespeare/(.+)" endpoint="/redirect.xqy">
  <uri-param name="__ml_redirect__">/$1</uri-param>
</request>
```

Your `options` node should now look like the one shown below. Note that the `request` elements for the `redirect.xqy` module are listed before those for the `endpoint.xqy` module. This is because of the "first-match" rule described in "Extracting Multiple Components from a URL" on page 220.

```
<options xmlns="http://marklogic.com/appservices/rest">
  <request uri="^/shakespeare/(.+)/(.+)" endpoint="/redirect.xqy">
    <uri-param name="__ml_redirect__">/$1/$2</uri-param>
  </request>
  <request uri="^/shakespeare/(.+)" endpoint="/redirect.xqy">
    <uri-param name="__ml_redirect__">/$1</uri-param>
  </request>
  <request uri="^/(.+)/act(\d+)" endpoint="/endpoint.xqy">
    <uri-param name="play">$1.xml</uri-param>
    <uri-param name="act" as="integer">$2</uri-param>
  </request>
  <request uri="^/(.+)$" endpoint="/endpoint.xqy">
    <uri-param name="play">$1.xml</uri-param>
  </request>
</options>
```

You can employ as many redirects as you want through the same `redirect.xqy` module by changing the value of the `__ml_redirect__` parameter.

16.10 Handling HTTP Verbs

A request that doesn't specify any verbs only matches HTTP GET requests. If you want to match other verbs, simply list them by using the `http` element with a `method` attribute in the `request` element:

```
<request uri="^/(.+?)/?$" endpoint="/endpoint.xqy">
  <uri-param name="play">$1.xml</uri-param>
  <http method="GET"/>
  <http method="POST"/>
</request>
```

This request will match (and validate) if the request method is either an HTTP GET or an HTTP POST.

The following topics describe use cases for mapping requests with verbs and simple endpoints that service those requests:

- Handling OPTIONS Requests
- Handling POST Requests

16.10.1 Handling OPTIONS Requests

You may find it useful to have a mechanism that returns the `options` node or a specific `request` element in the `options` node. For example, you could automate some aspects of unit testing based on the ability to find the `request` element that matches a given URL. You can implement this type of capability by supporting the `OPTIONS` method.

Below is a simple `options.xqy` module that handles requests that specify an `OPTIONS` method. If the request URL is /, the `options.xqy` module returns the entire `options` element, exposing the complete set of endpoints. When the URL is not /, the module returns the `request` element that matches the URL.

```
xquery version "1.0-ml";

import module namespace rest="http://marklogic.com/appservices/rest"
  at "/MarkLogic/appservices/utils/rest.xqy";

import module namespace requests =
  "http://marklogic.com/appservices/requests" at "requests.xqy";

(: Process requests to be handled by this endpoint module. :)
let $request := $requests:options/rest:request
                  [@endpoint = "/options.xqy"][1]

(: Get parameter/value map from request. :)
let $params  := rest:process-request($request)
let $uri     := map:get($params, "__ml_options__")
let $accept  := xdmp:get-request-header("Accept")
let $params  := map:map()

(: Get request element that matches the specified URL. :)
let $request := rest:matching-request($requests:options,
                                      $uri,
                                      "GET",
                                      $accept,
                                      $params)

(: If URL is '/', return options node. Otherwise, return request
   element that matches the specified URL. :)
return
  if ($uri = "/")
  then
    $requests:options
  else
    $request
```

Add the following `request` element to `requests.xqy` to match any HTTP request that includes an `OPTIONS` method.

```
<request uri="^(.+)$" endpoint="/options.xqy" user-params="allow">
  <uri-param name="__ml_options__">$1</uri-param>
  <http method="OPTIONS"/>
</request>
```

Open Query Console and enter the following query, replacing *name* and *password* with your login credentials:

```
xdmp:http-options("http://localhost:8011/",
<options xmlns="xdmp:http">
  <authentication method="digest">
    <username>name</username>
    <password>password</password>
  </authentication>
</options>)
```

Because the request URL is /, the entire `options` node should be returned. To see the results when another URL is used, enter the following query in Query Console:

```
xdmp:http-options("http://localhost:8011/shakespeare/macbeth",
<options xmlns="xdmp:http">
  <authentication method="digest">
    <username>name</username>
    <password>password</password>
  </authentication>
</options>)
```

Rather than the entire `options` node, the `request` element that matches the given URL is returned:

```
<request uri="^/shakespeare/(.+)" endpoint="/redirect.xqy"
  xmlns="http://marklogic.com/appservices/rest">
    <uri-param name="__ml_redirect__">/$1</uri-param>
</request>
```

You can use it by adding the following request to the end of your options:

```
<request uri="^(.+)$" endpoint="/options.xqy" user-params="allow">
  <uri-param name="__ml_options__">$1</uri-param>
  <http method="OPTIONS"/>
</request>
```

If some earlier request directly supports `OPTIONS` then it will have priority for that resource.

16.10.2 Handling POST Requests

You may want the ability to support the `POST` method to implement RESTful content management features, such as loading content into a database.

Below is a simple `post.xqy` module that accepts requests that include the `POST` method and inserts the body of the request into the database at the URL specified by the request.

```
xquery version "1.0-ml";

import module namespace rest="http://marklogic.com/appservices/rest"
       at "/MarkLogic/appservices/utils/rest.xqy";

import module namespace requests =
  "http://marklogic.com/appservices/requests" at "requests.xqy";

(: Process requests to be handled by this endpoint module. :)
let $request := $requests:options/rest:request
                  [@endpoint = "/post.xqy"][1]

(: Get parameter/value map from request. :)
let $params  := rest:process-request($request)
let $posturi := map:get($params, "_ml_post_")
let $type    := xdmp:get-request-header('Content-Type')

(: Obtain the format of the content. :)
let $format :=
  if ($type = 'application/xml' or ends-with($type, '+xml'))
  then
    "xml"
  else
    if (contains($type, "text/"))
    then "text"
    else "binary"

(: Insert the content of the request body into the database. :)
let $body := xdmp:get-request-body($format)

return
  (xdmp:document-insert($posturi, $body),
  concat("Successfully uploaded: ", $posturi, "&#10;"))
```

Add the following request element to `requests.xqy`. If the request URL is `/post/`*`filename`*, the rewriter will issue an HTTP request to the `post.xqy` module that includes the `POST` method.

```
<request uri="^/post/(.+)$" endpoint="/post.xqy">
  <uri-param name="_ml_post_">$1</uri-param>
  <http method="POST"/>
</request>
```

To test the `post.xqy` endpoint, open Query Console and enter the following query, replacing 'name' and 'password' with your MarkLogic Server login credentials:

```
let $document:= xdmp:quote(
  <html>
    <title>My Document</title>
    <body>
      This is my document.
    </body>
  </html>)

return
  xdmp:http-post("http://localhost:8011/post/mydoc.xml",
    <options xmlns="xdmp:http">
      <authentication method="digest">
        <username>name</username>
        <password>password</password>
      </authentication>
      <data>{$document}</data>
      <headers>
        <content-type>text/xml</content-type>
      </headers>
    </options>)
```

Click the Query Console Explore button and locate the `/mydoc.xml` document in the `Documents` database.

16.11 Defining Parameters

This section details the `uri-param` and `param` elements in a `request` element. The topics in this section are:

- Parameter Types
- Supporting Parameters Specified in a URL
- Required Parameters
- Default Parameter Value
- Specifying a List of Values
- Repeatable Parameters
- Parameter Key Alias
- Matching Regular Expressions in Parameters with the match and pattern Attributes

16.11.1 Parameter Types

By default, a parameter is typed as a string. Other types of parameters, such as integers or booleans, must be explicitly typed in the request element. Using the example `request` element from "Extracting Multiple Components from a URL" on page 220, the `act` parameter must be explicitly defined as an integer.

```
<request uri="^/(.+)/act(\d+)$" endpoint="/endpoint.xqy">
  <uri-param name="play">$1.xml</uri-param>
  <uri-param name="act" as="integer">$2</uri-param>
</request>
```

You can define a parameter type using any of the types supported by XQuery, as described in the specification, XML Schema Part 2: Datatypes Second Edition:

http://www.w3.org/TR/xmlschema-2/

16.11.2 Supporting Parameters Specified in a URL

The REST Library supports parameters entered after the URL path with the following format:

```
http://host:port/url-path?param=value
```

For example, you want the `endpoint.xqy` module to support a "scene" parameter, so you can enter the following URL to return Macbeth, Act 4, Scene 2:

```
http://localhost:8011/macbeth/act4?scene=2
```

To support the `scene` parameter, modify the first `request` element for the endpoint module as shown below. The `match` attribute in the `param` element defines a subexpression, so the parameter value is assigned to the `$1` variable, which is separate from the `$1` variable used by the `uri_param` element.

```
<request uri="^/(.+)/act(\d+)$" endpoint="/endpoint.xqy">
  <uri-param name="play">$1.xml</uri-param>
  <uri-param name="act" as="integer">$2</uri-param>
  <param name="scene" as="integer" match="(.+)">$1</param>
</request>
```

Rewrite the `endpoint.xqy` module as follows to add support for the `scene` parameter:

```
xquery version "1.0-ml";

import module namespace rest = "http://marklogic.com/appservices/rest"
    at "/MarkLogic/appservices/utils/rest.xqy";

import module namespace requests =
  "http://marklogic.com/appservices/requests" at "requests.xqy";

let $request := $requests:options/rest:request
                  [@endpoint = "/requests.xqy"][1]

let $map  := rest:process-request($request)
let $play := map:get($map, "play")
let $num  := map:get($map, "act")
let $scene := map:get($map, "scene")

return
    if (empty($num))
    then
      fn:doc($play)
    else if (empty($scene))
    then
      fn:doc($play)/PLAY/ACT[$num]
    else
      fn:doc($play)/PLAY/ACT[$num]/SCENE[$scene]
```

Now the rewriter and the endpoint will both recognize a `scene` parameter. You can define any number of parameters in a `request` element.

16.11.3 Required Parameters

By default parameters defined by the `param` element are optional. You can use the `required` attribute to make a parameter required. For example, you can use the `required` attribute as shown below to make the `scene` parameter required so that a request URL that doesn't have a `scene` will not match and an attempt to call the endpoint without a `scene` raises an error.

```
<param name="scene" as="integer" match="(.+)" required="true">
  $1
</param>
```

16.11.4 Default Parameter Value

You can provide a default value for a parameter. In the example below, a request for an act without a scene parameter will return scene 1 of that act:

```
<param name="scene" as="integer" match="(.+)" default="1">
  $1
</param>
```

16.11.5 Specifying a List of Values

For parameters like `scene`, you may want to specify a delimited list of values. For example, to support only requests for scenes 1, 2, and 3, you would do the following:

```
<param name="scene" as="integer" values="1|2|3" default="1"/>
```

16.11.6 Repeatable Parameters

You can mark a parameter as repeatable. For example, you want to allow a css parameter to specify additional stylesheets for a particular play. You might want to allow more than one, so you could add a css parameter like this:

```
<param name="css" repeatable="true"/>
```

In the rewriter, this would allow any number of css parameters. In the endpoint, there would be a single css key in the parameters map but its value would be a list.

16.11.7 Parameter Key Alias

There may be circumstances in which you want to interpret different key values in the incoming URL as a single key value.

For example, jQuery changes the key names if the value of a key is an array. So, if you ask JQuery to invoke a call with { "a": "b", "c": ["d", "e"] }, you get the following URL:

```
http://whatever/endpoint?a=b&amp;c[]=d&amp;c[]=e
```

You can use the `alias` attribute as shown below so that the map you get back from the `rest:process-request` function will have a key value of "c" regardless of whether the incoming URL uses `c=` or `c[]=` in the parameters:

```
<param name="c" alias="c[]" repeatable="true"/
```

16.11.8 Matching Regular Expressions in Parameters with the match and pattern Attributes

As shown in "Supporting Parameters Specified in a URL" on page 229, you can use the `match` attribute to perform the sort of match and replace operations on parameter values that you can perform on parts of the URL using the `uri` attribute in the `request` element. You can use the `pattern` attribute to test the name of the parameter. This section goes into more detail on the use of the `match` attribute and the `pattern` attribute. This section has the following parts:

- match Attribute
- pattern Attribute

16.11.8.1 match Attribute

The `match` attribute in the `param` element defines a subexpression with which to test the value of the parameter, so the captured group in the regular expression is assigned to the `$1` variable.

You can use the `match` attribute to translate parameters. For example, you want to translate a parameter that contains an internet media type and you want to extract part of that value using the `match` attribute. The following will translate `format=application/xslt+xml` to `format=xslt`.

```
<param name="format" match="^application/(.*?)(\+xml)?$">
    $1
</param>
```

If you combine matching in parameters with validation, make sure that you validate against the transformed value. For example, this parameter will never match:

```
<param name="test" values="foo|bar" match="^(.+)$">
    baz-$1
</param>
```

Instead, write it this way:

```
<param name="test" values="baz-foo|baz-bar" match="^(.+)$">
    baz-$1
</param>
```

In other words, the value that is validated is the transformed value.

16.11.8.2pattern Attribute

The `param` element supports a `pattern` attribute, which uses the specified regular expression to match the name of the parameter. This allows you to specify a regular expression for matching parameter names, for example:

```
pattern='xmlns:.+'

pattern='val[0-9]+'
```

Exactly one of `name` or `pattern` must be specified. It is an error if the name of a parameter passed to the endpoint matches more than one pattern.

16.12 Adding Conditions

You can add conditions, either in the body of the request, in which case they apply to all verbs, or within a particular verb. For example, the `request` element below contains an `auth` condition for the `POST` verb and a `user-agent` condition for both `GET` and `POST` verbs.

```
<request uri="^/slides/(.+?)/?$" endpoint="/slides.xqy">
    <uri-param name="play">$1.xml</uri-param>
    <http method="GET"/>
    <http method="POST">
      <auth>
        <privilege>http://example.com/privs/editor</privilege>
        <kind>execute</kind>
      </auth>
    </http>
    <user-agent>ELinks</user-agent>
</request>
```

With this request, only users with the specified execute privilege can `POST` to that URL. If a user without that privilege attempts to post, this request won't match and control will fall through to the next request. In this way, you can provide fallbacks if you wish.

In a rewriter, failing to match a condition causes the request not to match. In an endpoint, failing to match a condition raises an error.

The topics in this section are:

- Authentication Condition
- Accept Headers Condition
- User Agent Condition
- Function Condition
- And Condition
- Or Condition
- Content-Type Condition

16.12.1 Authentication Condition

You can add an `auth` condition that checks for specific privileges using the following format:

```
<auth>
  <privilege>privilege-uri</privilege>
  <kind>kind</kind>
</auth>
```

For example, the `request` element described for `POST` requests in "Handling POST Requests" on page 227 allows any user to load documents into the database. To restrict this `POST` capability to users with infostudio execute privilege, you can add the following `auth` condition to the request element:

```
<request uri="^/post/(.+)$" endpoint="/post.xqy">
  <uri-param name="_ml_post_">$1</uri-param>
  http method="POST">
    <auth>
      <privilege>
         http://marklogic.com/xdmp/privileges/infostudio
      </privilege>
      <kind>execute</kind>
    </auth>
  </http>
</request>
```

The privilege can be any specified execute or URL privilege. If unspecified, `kind` defaults to `execute`.

16.12.2 Accept Headers Condition

When a user agent requests a URL, it can also specify the kinds of responses that it is able to accept. These are specified in terms of media types. You can specify the media type(s) that are acceptable with the accept header.

For example, to match only user agent requests that can accept JSON responses, specify the following accept condition in the request:

```
<accept>application/json</accept>
```

16.12.3 User Agent Condition

You can also match on the user agent string. A request that specifies the `user-agent` shown below will only match user agents that identify as the ELinks browser.

```
<user-agent>ELinks</user-agent>
```

16.12.4 Function Condition

The `function` condition gives you the ability to test for arbitrary conditions. By specifying the namespace, local name, and module of a function, you can execute arbitrary code:

```
<function ns="http://example.com/module"
          apply="my-function"
          at="utils.xqy"/>
```

A request that specifies the function shown above will only match requests for which the specified function returns true. The function will be passed the URL string and the function condition element.

16.12.5 And Condition

An `and` condition must contain only conditions. It returns true if and only if all of its child conditions return true.

```
<and>
  ...conditions...
</and>
```

If more than one condition is present at the top level in a request, they are treated as they occurred in an `and`.

For example, the following condition matches only user agent requests that can accept responses in HTML from an ELinks browser:

```
<and>
  <accept>text/html</accept>
  <user-agent>ELinks</user-agent>
</and>
```

Note: There is no guarantee that conditions will be evaluated in any particular order or that all conditions will be evaluated.

16.12.6 Or Condition

An `or` condition must contain only conditions. It returns true if and only if at least one of its child conditions return true.

```
<or>
  ...conditions...
</or>
```

For example, the following condition matches only user agent requests that can accept responses in HTML or plain text:

```
<or>
  <accept>text/html</accept>
  <accept>text/plain</accept>
</or>
```

Note: There is no guarantee that conditions will be evaluated in any particular order or that all conditions will be evaluated.

16.12.7 Content-Type Condition

A `content-type` condition is a condition that returns true if the request has a matching content type. The `content-type` condition is allowed everywhere that conditions are allowed.

16.13 Preparing to Run the Examples

Before you can run the examples in this chapter, you must perform the following steps:

- Load the Example Data
- Create the Example App Server

16.13.1 Load the Example Data

The examples in this chapter assume you have the Shakespeare plays in the form of XML files loaded into a database. The easiest way to load the XML content into the `Documents` database is to do the following:

- Open Query Console and set the Database to `Documents`.
- Copy the query below into a Query Console window.
- Set the Query Type to XQuery.
- Click `Run` to run the query.

The following query loads the current database with the XML files in a zip file containing the plays of Shakespeare:

```
xquery version "1.0-ml";
```

```
import module namespace ooxml= "http://marklogic.com/openxml"
          at "/MarkLogic/openxml/package.xqy";

xdmp:set-response-content-type("text/plain"),
let $zip-file :=
 xdmp:document-get("http://www.ibiblio.org/bosak/xml/eg/shaks200.zip")

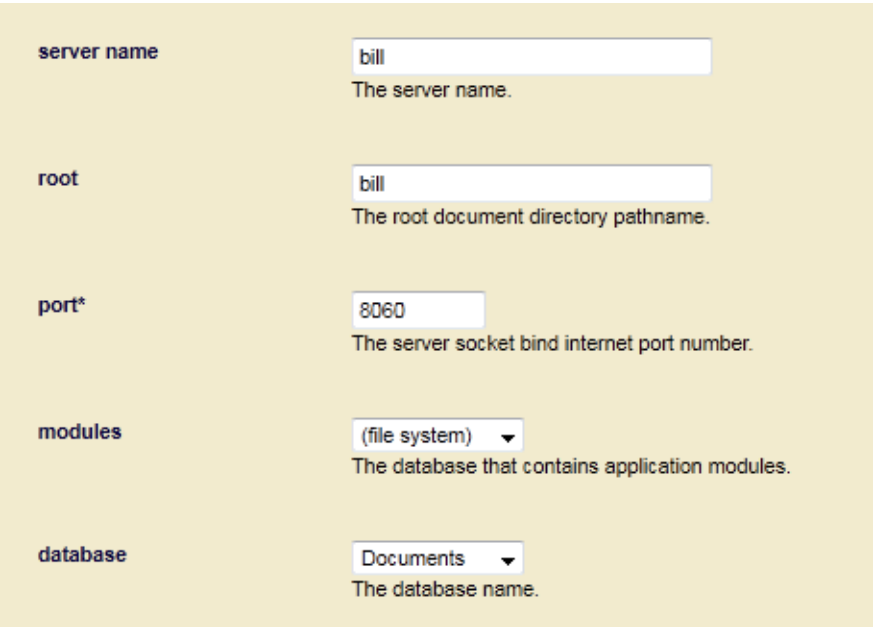
return for $play in ooxml:package-uris($zip-file)
  where fn:contains($play , ".xml")
    return (let $node := xdmp:zip-get ($zip-file, $play)
      return xdmp:document-insert($play, $node) )
```

Note: The XML source for the Shakespeare plays is subject to the copyright stated in the shaksper.htm file contained in the zip file.

16.13.2 Create the Example App Server

Follow this procedure to create an HTTP App Server with which to exercise the examples in this chapter.

1. In the Admin Interface, click the Groups icon in the left tree menu.
2. Click the group in which you want to define the HTTP server (for example, Default).
3. Click the App Servers icon on the left tree menu and create a new HTTP App Server.
4. Name the HTTP App Server `bill`, assign it port `8060`, specify `bill` as the root directory, and `Documents` as the database.



5. Create a new directory under the MarkLogic root directory, named `bill` to hold the modules you will create as part of the examples in this chapter.

17.0 Creating a Declarative XML Rewriter to Support REST Web Services

The Declarative XML Rewriter serves the same purpose as the Interpretive XQuery Rewriter described in "Creating an Interpretive XQuery Rewriter to Support REST Web Services" on page 211. The XML rewriter has many more options for affecting the request environment than the XQuery rewriter. However, because it is designed for efficiency, XML rewriter doesn't have the expressive power of the XQuery rewriter or access to the system function calls. Instead a select set of match and evaluation rules are available to support a large set of common cases.

The topics in this chapter are:

- Overview of the XML Rewriter
- Configuring an App Server to use the XML Rewriter
- Input and Output Contexts
- Regular Expressions (Regex)
- Match Rules
- System Variables
- Evaluation Rules
- Termination Rules
- Simple Rewriter Examples

17.1 Overview of the XML Rewriter

The XML rewriter is an XML file that contains rules for matching request values and preparing an environment for the request. If all the requested updates are accepted, then the request precedes with the updated environment, otherwise an error or warning is logged. The XQuery rewriter can only affect the request URI (Path and Query parameters). The XML rewriter, on the other hand, can change the content database, modules database, transaction ID, and other settings that would normally require an `eval-into` in an XQuery application. In some cases (such as requests for static content) the need for using XQuery code can be eliminated entirely for that request while still intercepting requests for dynamic content.

The XML rewriter enables XCC clients to communicate on the same port as REST and HTTP clients. You can also execute requests with the same features as XCC but without using the XCC library.

17.2 Configuring an App Server to use the XML Rewriter

To use an XML rewriter simply specify the XML rewriter (a file with an `.xml` extension) in the rewriter field for the server configuration of any HTTP server.

For example, the XML rewriter for the `App-Services` server at port `8000` is located in:

```
<marklogic-dir>/Modules/MarkLogic/rest-api/8000-rewriter.xml
```

17.3 Input and Output Contexts

The rewriter is invoked with a defined input context. A predefined set of modifications to the context is applied to the output context. These modifications are returned to the request handler for validation and application. The rewriter itself does not directly implement any changes to the input or output contexts.

The topics in this section are:

- Input Context
- Output Context

17.3.1 Input Context

The rewriter input context consists of a combination of matching properties accessible by the match rules described in “Match Rules” on page 246, or global system variables described in “System Variables” on page 263. When a matching rule for a property is evaluated, it produces locally scoped variables for the results of the match, which can be used by child rules.

The properties listed in the table below are available as the context of a match rule. Where "regex" is indicated, a match is done by a regular expression. Otherwise matches are “equals” of one or more components of the property.

Property / Description	Will Support Match by
path	regex
param	name [value]
HTTP header	name [value]
HTTP method	name in list
user	name or id
default user	is default user
execute privilege	in list

17.3.2 Output Context

The output context consists of values and actions that the rewriter is able (and allowed) to perform. These can be expressed as a set of context values and rewriter commands allowed on those values. Any of the output context properties can be omitted, in which case the corresponding input context is not modified. The simple case is no output from the rewriter and no changes to the input context. For example, if the output specifies a new database ID but it is the same as the current database, then no changes are required. The rewriter should not generate any conflicting output context, but it is ultimately up to the request handler to validate the changes for consistency as well as any other constraints, such as permissions. If the rewriter results in actions that are disallowed or invalid, such as setting to a nonexistent database or rewriting to an endpoint to which the user does not have permissions, then error handling is performed.

The input context properties, external path and external query, can be modified in the output context. There are other properties that can be added to the output context, such as to direct the request to a particular database or to set up a particular transaction, as shown in the table below.

Property	Description
path*	Rewritten path component of the URI
query*	Rewritten query parameters
module-database	Modules Database
root	Modules Root path
database	Database
eval	True if to evaluate path False for direct access
transaction	Transaction ID
transaction mode	Specify a query or update transaction mode.
error format	Specifies the error format for server generated errors

* These are modified from the input context.

17.4 Regular Expressions (Regex)

A common use case for paths in particular is the concept of "Match and Extract" (or "Match / Capture") using a regular expression.

As is the case with the regular expression rules for the `fn:replace` XQuery function, only the first (non overlapping) match in the string is processed and the rest ignored.

For example given the path shown below, you may want to both match the general form and at the same time extract components in one expression.

```
/admin/v2/meters/databases/12345/total/file.xqy
```

The following path match rule regex matches the above path and also extracts the desired components ("match groups") and sets them into the local context as numbered variables, as shown in the table below.

```
<match-path matches="/admin/v(.)/([a-z]+)/([a-z]+)/([0-9]+)/([a-z]+)/.+\.xqy">
```

Variable	Value
$0	/admin/v2/meters/databases/12345/total/file.xqy
$1	2
$2	meters
$3	databases
$4	12345
$5	total

The extracted values could then be used to construct output values such as additional query parameters.

> **Note:** No anchors (“^ .....$”) are used in this example, so the expression could also match a string, such as the one below and provide the same results.
>
> ```
> somestuff/admin/v2/meters/databases/12345/total/file.xqy/morestuff
> ```

Wherever a rule that matches a regex (indicated by the `matches` attribute) a flags option is allowed. Only the "i" flag (case insensitive) is currently supported.

17.5 Match Rules

Match rules control the evaluator execution flow. They are evaluated in several steps:

1. An Eval is performed on the rule to determine if it is a match

2. If it is a match, then the rule may produce zero or more "Eval Expressions" (local variables $*,$0 ... $n)

3. If it is a match then the evaluator descends into the match rule, otherwise the match is considered "not match" and the evaluator continues onto the next sibling.

4. If this is the last sibling then the evaluator "ascends" to the parent

Descending: When descending a match rule on match the following steps occur:

1. If "scoped" (attribute `scoped=true`) the current context (all in-scope user-defined variables and all currently active modification requests) is pushed.

2. Any Eval Expressions from the parent are cleared ($*,$0..$n) and replaced with the Eval Expressions produced by the matching node.

3. Evaluation proceeds at the first child node.

Ascending: When Ascending (after evaluating the last of the siblings) the evaluator Ascends to the parent node. The following steps occur:

1. If the parent was scoped (attribute `scoped=true`) then the current context is popped and replaced by the context of the parent node. Otherwise the context is left unchanged.

2. Eval Expressions ($*,$0...) popped and replaced with the parents in-scope eval expressions.

   Note: This is unaffected by the scoped attribute, Eval expressions are always scoped to only the immediate children of the match node that produced them.)

3. Evaluation proceeds at the next sibling of the parent node.

   Note: Ascending is a rare case and should be avoided, if possible.

The table below summarizes the match rules. A detailed description of each rule follows.

Element	Description
rewriter	Root element of the rewriter rule tree.
match-accept	Matches on an HTTP Accept header
match-content-type	Matches on an HTTP Content-Type header
match-cookie	Match on a cookie
match-execute-privilege	Match on the users execute privileges
match-header	Match on an HTTP Header
match-method	Match on the HTTP Method
match-path	Match on the request path
match-role	Match on the users assigned roles
match-string	Matches a string value against a regular expression
match-query-param	Match on a uri parameter (query parameter)
match-user	Match on a user name, id or default user

17.5.1 rewriter

Root element of the rewriter rule tree.

Attributes: none

Example:

Simple rewriter that redirects anything to `/home/**` to the module gohome.xqy otherwise passes through the request

```
<rewriter xmlns="http://marklogic.com/xdmp/rewriter">
     <match-path prefix="/home/">
           <dispatch>gohome.xqy</dispatch>
     </match-path>
</rewriter>
```

17.5.2 match-accept

Matches on the Accept HTTP Header.

Attributes

Name	Type	Required	Purpose
@any-of	list of strings	yes	Matches if the Accept header contains any of media types specified.
@scoped	boolean	no default **false**	Indicates this rule creates a new "scope" context for its children.
@repeated	boolean	no default **false**	If false then repeated matches are an immediate error.

Child Context Modifications

Variable	Type	Value
$0	string	The media types matched as a string
$*	list of strings	The media types matched as a List of String

Note: The match is performed as a case sensitive match against the literal strings of the type/subtype. No evaluation of expanding subtype, media ranges or quality factors are performed.

Example:

Dispatch to `/handle-text.xqy` if the media types `application/xml` or `text/plain` are specified in the Accept header.

```
<match-accept any-of="application/xml text/html">
      <dispatch>/handle-text.xqy</dispatch>
</match-accept>
```

17.5.3 match-content-type

Matches on the Content-Type HTTP Header.

Attributes

Name	Type	Required	Purpose
@any-of	list of strings	yes	Matches if the Content-Type header contains any of types specified.
@scoped	boolean	no default **false**	Indicates this rule creates a new "scope" context for its children.

Child Context Modifications

Variable	Type	Value
$0	string	The first types matched as a string.

Note: The match is performed as a case sensitive match against the literal strings of the type/subtype. No evaluation of expanding subtype, media ranges or quality factors are performed.

Example:

Dispatch to `/handle-text.xqy` if the media types `application/xml` or `text/plain` are specified in the Content-Type header.

```
<match-content-type any-of="application/xml text/html">
      <dispatch>/handle-text.xqy</dispatch>
</match-content-type>
```

17.5.4 match-cookie

Matches on a cookie by name. Cookies are an HTTP Header with a well-known structured format.

Attributes

Name	Type	Required	Purpose
@name	string	yes	Matches if the cookie of the specified name exists. Cookie names are matched in a case-insensitive manner.
@scoped	boolean	no default **false**	Indicates this rule creates a new "scope" context for its children.

Child Context Modifications

Variable	Type	Value
$0	string	The text value of the matching cookie.

Example:

Set the variable `$session` to the cookie value SESSIONID, if it exists:

```
<match-cookie name="SESSIONID">
       <set-var name="session">$0</set-var>
       ....
</match-cookie>
```

17.5.5 match-execute-privilege

Match on the users execute privileges

Attributes

Name	Type	Required	Purpose
@any-of	list of uris	no*	Matches if the user has at least one of the specified execute privileges
@all-of	list of uris	no*	Matches if the user has all of the specified execute privileges
@scoped	boolean	no default **false**	Indicates this rule creates a new "scope" context for its children.

* Exactly One of @any-of or @all-of of is required

Note: The execute privilege should be the URI not the name. See the example.

Child Context modifications:

Variable	Type	Value
$0	string	The matching privileges. For more than one match it is converted to a space delimited string
$*	list of strings	All of the matching privileges as a List of String

Example:

Dispatches if the user has either the `admin-module-read` or `admin-ui` privilege.

```
<match-execute-privilege
     any-of="http://marklogic.com/xdmp/privileges/admin-module-read
             http://marklogic.com/xdmp/privileges/admin-ui">
     <dispatch/>
</match-execute-privilege>
```

Note: In the XML format you can use newlines in the attribute

17.5.6 match-header

Match on an HTTP Header

Attributes

Name	Type	Required	Purpose
@name	string	yes	Matches if a header exists equal to the name. HTTP Header names are matched with a case insensitive string equals.
@value	string	no	Matches if a header exists with the name and value. The name is compared case insensitive, the value is case sensitive.
@matches	regex	no	Matches by regex
@flags	string	no	Optional regex flags. "i" for case insensitive.
@scoped	boolean	no default **false**	Indicates this rule creates a new "scope" context for its children.
@repeated	boolean	no default **false**	If `false` then repeated matches are an error.

Only one of `@value` or `@matches` is allowed but both may be omitted.

Child Context modifications:

If `@value` is specified, then `$0` is set to the matching value

If there is no `@matches` or `@value` attribute, then `$0` is the entire text content of the header of that name. If more than one header matches, then `@repeated` indicates if this is an error or allowed. If allowed (`true`), then `$*` is set to each individual value and `$0` to the space delimited concatenation of all headers. If `false` (default) multiple matches generates an error.

If `@matches` is specified then, as with match-path and match-string, `$0 .. $N` are the results of the regex match

Variable	Type	Value
$0	string	The value of the matched header
$1....$N	string	Each matching group

Example:

Adds a query-parameter if the User agent contains FireFox/20.0 or FireFox/21.0

```
<match-header name="User-Agent" matches="FireFox/2[01]\.0">
     <add-query-param name="do-firefox">yes</add-query-param>
     ...
</match-header>
```

17.5.7 match-method

Match on the HTTP Method

Attributes

Name	Type	Required	Purpose
@any-of	string list	yes	Matches if the HTTP method is one of the values in the list. Method names are Case Sensitive matches.
@scoped	boolean	no default **false**	Indicates this rule creates a new "scope" context for its children.

At least one method name must be specified.

Child Context modifications: none

The value of the HTTP method is a system global variable, `$_method`, as described in “System Variables” on page 263.

Example:

Dispatches if the method is either GET HEAD or OPTIONS AND if the user has the execute privilege http://marklogic.com/xdmp/privileges/manage

```
<match-method any-of="GET HEAD OPTIONS">
     <match-execute-privilege
         any-of="http://marklogic.com/xdmp/privileges/manage">
         <set-path>/history/endpoints/resources.xqy</set-path>
         <dispatch/>
     </match-execute-privilege>
</match-method>
```

17.5.8 match-path

Match on the request path. The "path" refers to the "path" component of the request URI as per RFC3986 [https://tools.ietf.org/html/rfc3986] . Simply, this is the part of the URL after the scheme, and authority section, starting with a "/' (even if none were given) up to but not including the Query Param seperator "?" and not including any fragement ("#").

The Path is NOT URL Decoded for the purposes of match-path, but query parameter values are decoded (as per HTTP specifications). This is intentional so that path components can contain what would otherwise be considered path component separates, and because HTTP specifications make the intent clear that path components are only to be encode when the 'purpose' of the path is ambiguous without encoding, therefore characters in a path are only supposed to be URL encoded in the case they are intended to NOT be considered as path separator compoents (or reserved URL characters).

For example, the URL:

```
http://localhost:8040//root%2Ftestme.xqy?name=%2Ftest
```

is received by the server as the HTTP request:

```
GET /root%2Ftestme.xqy?name=%2Ftest
```

This is parsed as:

```
PATH:  /root%2Ftestme.xqy
```

Query (name/value pairs decoded) : ("name" , "/test")

A match-path can be used to distinguish this from a URL such as:

```
http://localhost:8040//root/testme.xqy?name=%2Ftest
```

Which would be parsed as:

```
PATH:  /root/testme.xqy
```

For example, `<match-path matches="/root([^/].*)">` would match the first URL but not the second, even though they would decode to the same path.

When match results are placed into $0..$n then the default behavior is to decode the results so that in the above case, $1 would be "/testme.xqy". This is to handle consistency with other values which also are in decoded form, in particular when a value is set as a query parameter it is then *URL encoded* as part of the rewriting. If the value was in encoded form already it would be encoded twice resulting in the wrong value.

In the (rare) case where it is not desired for path-match to decode the results after a match the attribute @uri-decode can be specified and set to false.

Attributes

Name	Type	Required	Purpose
@matches	regex	no*	Matches if the path matches the regular expression
@prefix	string	no*	Matches if the path starts with the prefix (literal string)
@flags	string	no*	Optional regex flags. "i" for case insensitive.
@any-of	list of strings	no*	Matches if the path is one of the list of exact matches.
@url-decode	boolean	no default **true**	If true (default) then results are URL Decoded after extracted from the matching part of the path
@scoped	boolean	no default **false**	Indicates this rule creates a new "scope" context for its children.

Only one of @matches or @prefix or @any-of is allowed.

If supplied, @matches, @prefix, or @any-of must be non-empty.

@flags only applies to @matches (not @prefix or @any-of).

If none of @matches, @prefix or @any-of is specified then match-path matches all paths.

To match an empty path use matches="^$" (not matches="" which will match anything)

To match all paths omit @matches, @prefix and @any-of

Child Context modifications:

Variable	Type	Value
$0	string	The entire portion of the path that matched. For matches this is the full matching text. For @prefix this is the prefix pattern. For @any-of this is which of the strings in the list matched.
$1 ... $N	string	Only for @matches. The value of the numeric match group as defined by the XQuery function fn:replace()

Example:

```
<match-path
    matches="^/manage/(v2|LATEST)/meters/labels/([^F$*/?&amp;]+)/?$">
       <add-query-param name="version">$1</add-query-param>
       <add-query-param name="label-id">$2</add-query-param>
       <set-path>/history/endpoints/labels-item.xqy</set-path>
       ...
</match-path>
```

17.5.9 match-query-param

Match on a query parameter.

Query parameters can be matched exactly (by name and value quality) or partially (by only specifying a name match). For exact matches only one name/value pair is matched. For partial matches it is possible to match multiple name/value pairs with the same name when the query parameter has multiple parameters with the same name. The repeated attribute specifies if this is an error or not, the default (false) indicates repeated matching parameters are an error.

Attributes:

Name	Type	Required	Purpose
@name	string	yes	Matches if a query parameter exists with the name
@value	string	no	Matches if a query parameter exists with the name and value.
@scoped	boolean	no default **false**	Indicates this rule creates a new "scope" context for its children.
@repeated	boolean	no default **false**	If false then repeated matches are an immediate error.

Note: If a @value parameter that is present but empty is valid and is a match for the presence of query parameter with an empty value.

Child Context modifications:

Variable	Type	Value
$0	String	The value(s) of the matched query parameter. If the query parameter has multiple values that matched (due to multiple parameters of the same name) then the matched values are converted to a space delimited String.
$*	list of strings	A list of all the matched values as in $0 except as a List of String

Example:

If the query param user has the value "admin" verify AND they have execute privilege http://marklogic.com/xdmp/privileges/manage then dispatch to /admin.xqy.

```
<match-query-param name="user" value="admin">
      <match-execute-privilege
           any-of="http://marklogic.com/xdmp/privileges/manage">
           <dispatch>/admin.xqy</dispatch>
      /match-execute-privilege>
</match-query-param>
```

If the query parameter contains a transaction then set the transaction ID.

```
<match-query-param name="transaction">
      <set-transaction>$0</set-transaction>
      ...
</match-query-param>
```

Test for the existence of an empty query parameter.

For the URI: /query.xqy?a=has-value&b=&c=cvalue

This rule will set the value of "b" to "default" if it is empty.

```
<match-query-param name="b" value="">
      <set-query-param name="b" value="default"/>
</match-query-param>
```

See match-string for an example of multiple query parameters with the same name.

17.5.10 match-role

Match on the users assigned roles

Attributes

Name	Type	Required	Purpose
@any-of	list of role names (strings)	no*	Matches if the user has the at least one of the specified roles
@all-of	list of role names (strings)	no*	Matches if the user has all of the specified roles
@scoped	boolean	no default **false**	Indicates this rule creates a new "scope" context for its children.

* Exactly One of @any-of or @all-of is required

Child Context modifications:

Variable	Type	Value
$0	string	For any-of the first role that matched. Otherwise unset, (if all-of matched, its known what those roles are, the contents of @all-of).

Example:

Matches if the user has both of the roles infostudio-user AND rest-user

```
<match-role all-of="infostudio-user rest-user">
       ...
</match-role>
```

17.5.11 match-string

Matches a string expression against a regular expression. If the value matches then the rule succeeds and its children are descended.

This rule is intended to fill in gaps where the current rules are not sufficient or would be overly complex to implement additional regular expression matching to all rules. You should avoid using this rule unless it is absolutely necessary.

Attributes

Name	Type	Required	Purpose
@value	string	yes	The value to match against. May be a literal string or may be a single variable expression.
@matches	regex string	yes	Matches if the value matches the regular expression
@flags	string	no	Optional regex flags. "i" for case insensitive.
@scoped	boolean	no default **false**	Indicates this rule creates a new "scope" context for its children.
@repeated	boolean	false	If false then repeated matches are an error.

Child Context modifications:

Variable	Type	Value
$0	string	The entire portion of the value that matched.
$1 ... $N	string	The value of the numeric match group

See Regex (Regular Expressions)

Repeated matches: Regular expressions can match multiple non-overlapping portions of a string, if the regex is not anchored to the begin and end of the string.

17.5.12 match-user

Match on a user name or default user.

To match the user, use only one of @name.

To match if the user is the default user, use @default-user.

You can match both by supplying both a @name and a @default-user.

@default-user defaults to false.

If @default-user is false, no check is made for the value of @default-user.

Attributes

Name	Type	Required	Purpose
@name	string	no*	Matches the user name
@default-user	boolean (true | false)	no* default **false**	If true, matches if the user is the default user. If false, checks to see if @name matches the user name, but does not check the value of @default-user.
@scoped	boolean	no default **false**	Indicates this rule creates a new "scope" context for its children.

Child Context modifications: None: One of name or default-user available as system variables; See System Variables

Examples:

Matches the default user (note that there is no need to specify the “name’ attribute in this case):

```
<match-user default-user="true">
  ...
</match-user>
```

Matches the non-default user “grace”:

```
<match-user default-user="false" name="grace" >
  ...
</match-user>
```

17.6 System Variables

This section describes the predefined system variables that compose the initial input context. These are available in the context of any variable substitution expression.

System variables are used to substitute for the mechanism used by the XQuery rewriter which can get this information (and much more) by calling any of our XQuery APIs. The Declarative rewriter does not expose any API calls so in cases where the values may be needed in outputs they are made available as global variables. There is some overlap in these variables and the Match rules to simplify the case where you simply need to set a value but don't need to match on it. For example the set-database rule may want to set the database to the current modules database (to allow GET and PUT operations on module files). By supplying a system variable for the modules database ($_modules-database) there is no need for a matching rule on modules-database for the sole purpose of extracting the value.

System variables use a reserved prefix "_" to avoid accidental use in current or future code if new variables are added. Overwriting a system variable is only set in the current scope and does not produce changes to the system.

The period (".") is a convention that suggests the idea of property access but is really just part of the variable name. Where variables start with the same prefix but have ".<name>" as a suffix this is a convention that the name without the dot evaluates to the most useful value and the name with the dot specifics a specific property or type for that variable. For example $_database is the database Name, $_database.id is the database ID.

As noted in Variables and Types the actual type of all variable is a String (or List of String), the Type column in the table below is to indicate what range of values is possible for that variable. For example a database id originates as an unsigned long so can be safely used in any expression that expects a number.

Note:

- [*name*] means that *name* is optional.
- <*name*> means that *name* is not a predefined constant but is required

Variable	Type(s)	Desc / Notes
`$_cookie.<name>`	string	The value of the cookie <name>. Only the text value of the cookie is returned, not the extra metadata (path, domain, expires etc.). If the cookie does not exist evaluates as "" Cookie names matched and compared case insensitive. Future: may expose substructure of the cookie header

Variable	Type(s)	Desc / Notes
`$_database[.name]`	string	The name of the content database.
`$_database.id`	integer	The ID of the content database.
`$_defaultuser`	boolean	True if the authenticated user is the default user.
`$_method`	string	HTTP Method name.
`$_modules-database[.name]`	string	Modules database name. If no name is given, the file system is used for the modules.
`$_modules-database.id`	integer	Modules database ID. Set the ID to 0 to use the file system for modules.
`$_modules-root`	string	Modules root path.
`$_path`	string	The HTTP request path Not including the query string.
`$_query-param.<name>`	list of strings	The query parameters matching the name as a list of strings.
`$_request-url`	string	The original request URI, including the path and query parameters.
`$_user[.name]`	string	The user name.
`$_user.id`	integer	The user ID.

Set the filesystem for modules:

```
<set-database>$_modules-database</set-database>
```

Set the transaction to the cookie `TRANSACTION_ID`:

```
<set-transaction>$_cookie.TRANSACTION_ID</set-transaction>
```

17.7 Evaluation Rules

Eval rules have no effect on the execution control of the evaluator. They are evaluated when reached and only can affect the current context, not control the execution flow.

There are two types of eval rules: Set rules and assign rules.

Set Rules are rules that create a rewriter command (a request to change the output context in some way). Assign rules are rules that set locally scoped variables but do not produce any rewriter commands.

Variable and rewriter commands are placed into the current scope.

Element	Description
add-query-param	Adds a query parameter (name/value) to the query parameters
set-database	Sets the database
set-error-format	Sets the error format for system generated errors
set-error-handler	Sets the error handler
set-eval	Sets the evaluation mode (eval or direct)
set-modules-database	Sets the modules database
set-modules-root	Sets the modules root path
set-path	Sets the URI path
set-query-param	Sets a query parameter
set-transaction	Sets the transaction
set-transaction-mode	Sets the transaction mode
set-var	Sets a variable in the local scope
trace	Log a trace message

17.7.1 add-query-param

Adds (appends) a query parameter (name/value) to the query parameters

Attributes

Name	Type	Required	Purpose
@name	string	yes	Name of the parameter

Children:

Expression which evaluates to the value of the parameter

An empty element or list will still append a query parameter with an empty value (equivalent to a URL like http://company.com?a=)

If the expression is a List then the query parameter is duplicated once for each value in the list.

Example:

If the path matches then append to query parameters

- version= the version matched
- label-id =the label id matched

```
<match-path
     matches="^/manage/(v2|LATEST)/meters/labels/([^/?&amp;]+)/?$">
    <add-query-param name="version">$1</add-query-param>
    <add-query-param name="label-id">$2</add-query-param>
</match-path>
```

17.7.2 set-database

Sets the Database.

This will change the context Database for the remainder of request.

Attributes

Name	Type	Required	Purpose
@checked	boolean [true,1 | false,0]	no	If true then the eval-in privilege of the user is checked to verify the change is allowed.

Children:

An expression which evaluates to either a database ID or database name.

It is an immediate error to set the value using an expression evaluating to a list of values.

See Database (Name or ID) for a description of how Database references are interpreted.

Notes on @checked flag.

The @checked flag is interpreted during the rewriter modification result phase, by implication this means that only the last set-database that successfully evaluated before a dispatch is used.

If the @checked flag is true AND if the database is different than the App Server defined database then the user must have the eval-in privilege.

Examples:

Set the database to "SpecialDocuments":

```
<set-database>SpecialDocuments</set-database>
```

Set the database to the current modules database:

```
<set-database>$_modules-database</set-database>
```

17.7.3 set-error-format

Sets the error format used for all system generated errors. This is the format (content-type) of the body of error messages for a non-successful HTTP response.

This overwrites the setting from the application server configuration and takes effect immediately after validation of the rewriter rules have succeeded.

Attributes: None

Children: An expression which evaluates to one of the following error formats.

- html
- json
- xml
- compatible

The "compatible" format indicates for the system to match as closely as possible the format used in prior releases for the type of request and error. For example, if dispatch indicates "xdbc" then "compatible" will produce errors in the HTML format, which is compatible with XCC client library.

It is an immediate error to set the value using an expression evaluating to a list of values.

> **Note:** This setting does not affect any user defined error handler, which is free to output any format and body.

Example:

Set the error format for json responses

```
<set-error-format>json </set-database>
```

17.7.4 set-error-handler

Sets the error handler

Attributes: None

Children: An expression which evaluates to a Path (non blank String).

Example:

```
<set-error-handler >/myerror-handler.xqy</set-modules-root>
```

If error occurs during the rewriting process then the error handler which is associated with the application server is used for error handling. After a successful rewrite if the set-error-handler specifies a new error handler then it will be used for handling errors.

The modules database and modules root used to locate the error handler is the modules database and root in effect at the time of the error.

Setting the error handler to the empty string will disable the use of any user defined error handler for the remainder of the request.

It is an immediate error to set the value using an expression evaluating to a list of values.

For example, if in addition the set-modules-database rule was used, then the new error handler will be search for in the rewritten modules database (and root set with set-modules-root) otherwise the error handler will be searched for in the modules database configured in the app server.

17.7.5 set-eval

Sets the Evaluation mode (eval or direct).

The Evaluation mode is used in the request handler to determine if a path is to be evaluated (XQuery or JavaScript) or to be directly accessed (PUT/GET).

In order to be able to read and write to evaluable documents (in the modules database), the evaluation mode needs to be set to direct and the Database needs to be set to a Modules database.

Attributes: None

Children: An expression evaluating to either "eval" or "direct"

Example:

Forces a direct file access instead of an evaluation if the filename ends in .xqy

```
<match-path matches=".*\.xqy$">
     <set-eval>direct</set-eval>
</match-user>
```

17.7.6 set-modules-database

Sets the Modules database.

This sets the modules database for the request.

Attributes

Name	Type	Required	Purpose
@checked	boolean [true,1 | false,0] default **false**	no	If true then the permissions of the user are checked for the eval-in privilege verify the change is allowed.

Children:

An expression which evaluates to either a database ID or database name. An empty value, expression or expression evaluating to "0" indicates "Filesystem", otherwise the value is interpreted as a database Name, or ID.

See Database (Name or ID) for a description of how Database references are interpreted.

It is an immediate error to set the value using an expression evaluating to a list of values.

Notes on @checked flag.

The @checked flag is interpreted during the rewriter modification result phase, by implication this means that only the last set-database that successfully evaluated before a dispatch is used.

If the @checked flag is true AND if the database is different than the App Server defined modules database then the user must have the eval-in privilege.

Example:

Sets the database to "SpecialDocuments"

```
<match-user name="admin">
      <set-modules-database>SpecialModules</set-modules-database>
      ...
</match-user>
```

17.7.7 set-modules-root

Sets the modules root path

Attributes: None

Children: An expression which evaluates to a Path (non blank String).

It is an immediate error to set the value using an expression evaluating to a list of values.

Example:

Sets the modules root path to /myapp

```
<set-modules-root>/myapp</set-modules-root>
```

17.7.8 set-path

Sets the URI path for the request.

Often this is the primary use case for the rewriter.

Attributes: None

Children:

An expression which evaluates to a Path (non blank String).

It is an immediate error to set the value using an expression evaluating to a list of values.

Example:

If the user name is "admin" then set the path to /admin.xqy

Then if the method is either GET , HEAD, OPTIONS dispatch otherwise if the method is POST then set a query parameter "verified" to true and dispatch.

```
<match-user name="admin">
    <set-path>/admin.xqy</set-path>
    <match-method any-of="GET HEAD OPTIONS">
        <dispatch/>
    </match-method>
    <match-method any-of="POST">
        <set-query-param name="verified">true</set-query-param>
        <dispatch/>
    </match-method>
</match-user>
```

See 4.1.5.6.1for a way to set-path and dispatch in the same rule.

17.7.9 set-query-param

Sets (overwrites) a query parameter. If the query parameter previously existed all of its values are replaced with the new value(s).

Attributes

Name	Type	Required	Purpose
@name	string	yes	Name of the parameter

Children

An expression which evaluates to the value of the query parameter to be set. If the expression is a List then the query parameter is duplicated once for each value in the list.

An empty element, empty string value or empty list value will still set a query parameter with an empty value (equivalent to a URL like http://company.com?a=)

Examples:

If the user is admin then set the query parameter user to be admin, overwriting any previous values it may have had.

```
<match-user name="admin">
     <set-query-param name="user">admin</set-query-param>
</match-user>
```

Copy all the values from the query param "ids" to a new query parameter "app-ids" replacing any values it may have had.

```
<match-query-param name="ids">
     <set-query-param name="app-ids">$*</set-query-param>
</match-query-param>
```

This can be used to "pass through" query parameters by name when @include-request-query-params is specified in the <dispatch> rule.

The following rules will copy all query parameter (0 or more) named "special" to result without passing through other parameters.

```
<match-query-param name="special" repeated="true">
      <set-query-param name=" special">$*</set-query-param>
</match-query-param>
<dispatch include-request-query-params="false"/>
```

17.7.10 set-transaction

Sets the current transaction. If specified, set-transaction-mode must also be set.

Attributes: None

Children: An expression which evaluates to the transaction ID.

Example:

Set the transaction to the value of the cookie TRANSACTION_ID.

```
<set-transaction>$_cookie.TRANSACTION_ID</set-transaction>
```

Note: If the expression for set-transaction is empty, such as when the cookie doesn't exist, then the transaction is unchanged.

It is an immediate error (during rewriter parsing) to set the value using an expression evaluating to a list of values or to 0.

17.7.11 set-transaction-mode

Sets the transaction mode for the current transaction. If specified, set-transaction must also be set.

Attributes: None

Children: An expression evaluating to a transaction mode specified by exactly one of the strings

("auto" | "query" | "update")

Example:

Set the transaction mode to the value of the query param "trmode" if it exists.

```
<match-query-param name="trmode">
      <set-transaction-mode>$0</set-transaction-mode>
</match-query-param>
```

Note: It is an error if the value for transaction mode is not one of "auto," "query," or "update." It is also an error to set the value using an expression evaluating to a list of values.

17.7.12 set-var

Sets a variable in the local scope

This is an Assign Rule. It does not produce rewriter commands instead it sets a variable.

The assignment only affects the current scope (which is the list of variables pushed by the parent). The variable is visible to following siblings as well as children of following siblings.

Allowed user defined variable names must start with a letter and followed by zero or more letters, numbers, underscore or dash.

Specifically the name must match the regex pattern "[a-zA-Z][a-zA-Z0-9_-]*"

This implies that set-var cannot set either system defined variables, property components or expression variables.

Attributes

Name	Type	Required	Purpose
@name	string	yes	Name of the variable to set (without the "$")

Children:

An expression which is evaluated to value to set the variable.

Examples:

Sets the variable $dir1 to the first component of the matching path, and $dir2 to the second component.

```
<match-path matches="^/([a-z]+)/([a-z]+)/.*">
    <set-var name="dir1">$1</set-var>
    <set-var name="dir2">$2</set-var>
    ...
</match-path
```

If the Modules Database name contains the string "User" then set the variable usedb to the full name of the Modules DB.

```
<match-string  value="$_modules-database" matches=".*User.*">
    <set-var name="usedb">$0</set-var>
</match-string>
```

Matches all of the values of a query parameter named "ids" if any of them is fully numeric.

```
<match-query-param name="ids">
    <match-string  value="$*" matches="[0-9]+">
          ....
    </match-string>
</match-query-param>
```

17.7.13 trace

Log a trace message

The trace rule can be used anywhere an eval rule is allowed. It logs a trace message similar to `fn:trace`.

The event attribute specifies the Trace Event ID. The body of the trace element is the message to log.

Attributes

Name	Type	Required	Purpose
@event	string	yes	Specifies the trace event

Child Content: Trace message or expression.

Child Elements: None

Child Context modifications: None

Example:

```
<match-path prefix="/special">
    <trace event="AppEvent1">
        The following trace contains the matched path.
    </trace>
    <trace event="AppEvent2">
        $0
    </trace>
</match-path>
```

17.8 Termination Rules

Termination rules (dispatch, error) unconditionally stop the evaluator at the current rule. No further evaluation occurs. The dispatch rule will return out of the evaluator with all accumulated rewriter commands in scope. The error rule discards all command and returns with the error condition.

Element	Description
dispatch	Stop evaluation and dispatch with all rewrite commands
error	Terminates evaluation with an error

17.8.1 dispatch

Stop evaluation and dispatch with all rewrite commands.

The `dispatch` element is required as the last child of any match rule which contains no match rules.

Attributes

Name	Type	Required	Purpose
@include-request-query-params	boolean default **true**	no	If `true` then the original request query params are used as the initial set of query params before applying any rewrites
@xdbc	boolean default **false**	no	If `true` then the built-in XDBC handlers are used for the request.

The attribute `include-request-query-params` specifies whether the initial request query parameters are included in the rewriter result If `true` (or absent) then the rewriter modifications start with the initial query parameters and then are augmented (added or reset) by any set-query-param and add-query-param rules which are in scope at the time of dispatch.

If set to `false` then the initial request parameters are not included and only the parameters set or added by any set-query-param and add-query-param rules are included in the result.

If `xdbc` is specified and `true` then the built-in xdbc handlers will be used for the request. If xdbc support is enabled then the final path (possibly rewritten) MUST BE one of the paths supported by the xdbc built-in handlers.

Child Content:

Empty or an expression

Child Elements:

If the child element is not empty or blank then it is evaluated and used for the rewrite path.

Child Context modifications:

Examples:

```
<set-path>/a/path.xqy
     <dispatch/>
</set-path>
```

Is equivalent to:

```
<dispatch>/a/path.xqy</dispatch>
```

If the original URL is `/test?a=a&b=b`, the rewriter:

```
<set-query-param name="a">a1</set-query-param>
<dispatch include-request-query-params="false">/run.xqy</dispatch>
```

rewrites to path `/run.xqy` and the query parameters are:

```
a=a1
```

The following rewriter:

```
<set-query-param name="a">a1</set-query-param>
<dispatch>run.xqy</dispatch>
```

rewrites to path `/run.xqy` and the query parameters are:

```
a=a1
b=b
```

An example of a minimal rewriter rule that dispatches to XDBC is as follows:

```
<match-path  any-of="/eval /invoke /spawn /insert">
      <dispatch xdbc="true">$0</dispatch>
<match-path>
```

17.8.2 error

Terminate evaluation with an error.

The `error` rule terminates the evaluation of the entire rewriter and returns and error to the request handler. This error is then handled by the request handler, passing to the error-handler if there is one.

The code (optional) optional message data are supplied as attributes.

Attributes:

Name	Type	Required	Purpose
@code	string	yes	Specifies the error code
@data1	string	no	Error message, first part
@data2	string	no	Error message, second part
@data3	string	no	Error message, third part
@data4	string	no	Error message, fourth part
@data5	string	no	Error message, fifth part

Child Content:

None

Child Elements:

None

Child Context modifications: none

Example:

```
<error code="XDMP-BAD" data1="this" data2="that"/>
```

17.9 Simple Rewriter Examples

Some examples of simple rewriters:

Redirect a request by removing the prefix, `/dir`.

```
<rewriter xmlns="http://marklogic.com/xdmp/rewriter">
   <match-path matches="^/dir(/.+)">
      <dispatch>$1</dispatch>
   </match-path>
</rewriter>
```

For GET and PUT requests only, if the a query parameter named `path` is exactly `/admin` then redirect to `/private/admin.xqy` otherwise use the value of the parameter for the redirect.

If no `path` query parameter then do not change the request

```
<rewriter xmlns="http://marklogic.com/xdmp/rewriter">
   <match-method any-of="GET PUT">
      <!-- match by name/value -->
      <match-query-param name="path" value="/admin">
         <dispatch>/private/admin.xqy</dispatch>:
      </match-query-param>
      <!-- match by name use value -->
      <match-query-param name="path">
         <dispatch>$0</dispatch>:
      </match-query-param>
   </match-method>
</rewriter>
```

If a parameter named `data` is present in the URI then set the database to `UserData`. If a query parameter `module` is present then set the modules database to `UserModule`. If the path starts with `/users/` and ends with `/version<versionID>` then extract the next path component (`$1`), append it to `/app` and add a query parameter `version` with the `versionID`.

```
<rewriter xmlns="http://marklogic.com/xdmp/rewriter">
   <match-query-param name="data">
      <set-database>UserData</set-database>
   </match-query-param>
   <match-query-param name="module">
      <set-modules-database>UserModule</set-modules-database>
   </match-query-param>
   <match-path match="^/users/([^/]+)/version(.+)%">
      <set-path>/app/$1</set-path>
      <add-query-param name="version">$2</add-query-param>
   </match-path>
   <dispatch/>
</rewriter>
```

Match users by name and default user and set or overwrite a query parameter.

```
<rewriter xmlns="http://marklogic.com/xdmp/rewriter">
   <set-query-param name="default">
        default-user no match
   </set-query-param>
   <match-user name="admin">
       <add-query-param name="user">admin matched</add-query-param>
   </match-user>
   <match-user name="infostudio-admin">
       <add-query-param name="user">
           infostudio-admin matced
       </add-query-param>
   </match-user>
   <match-user default-user="true">
      <set-query-param name="default">
         default-user matched
      </set-query-param>
   </match-user>
   <dispatch>/myapp.xqy</dispatch>
</rewriter>
```

Matching cookies. This properly parses the cookie HTTP header structure so matches can be performed reliably. In this example, the SESSIONID cookie is used to conditionally set the current transaction.

```
<rewriter xmlns="http://marklogic.com/xdmp/rewriter">
   <match-cookie name="SESSIONID">
      <set-transaction>$0</set-transaction>
   </match-cookie>
</rewriter>
```

User defined variables with local scoping. Set an initial value to the user variable “test”. If the patch starts with /test/ and contains atleast 2 more path components then reset the “test” variable to the first matching path, and add a query param “var1″ to the second matching path. If the role of the user also contains either “admin-builtins” or “app-builder” then rewrite to the path ‘/admin/secret.xqy’, otherwise add a query param “var2″ with the value of the “test” user variable and rewrite to “/default.xqy”

If you change the scoped attribute from true to false, (or remove it), then all the changes within that condition are discarded if the final dispatch to /admin/secret.xqy is not reached, leaving intact the initial value for the “test” variable, not adding the “var1″ query parameter and dispatching to /default.xqy

```
<rewriter xmlns="http://marklogic.com/xdmp/rewriter" >
   <set-var name="test">initial</set-var>
   <match-path matches="^/test/(\w+)/(\w+).*" scoped="true">
      <set-var name="test">$1</set-var>
      <set-query-param name="var1">$2</set-query-param>
      <match-role any-of="admin-builtins app-builder">
         <dispatch>/admin/secret.xqy</dispatch>
      </match-role>
   </match-path>
   <add-query-param name="var2">$test</add-query-param>
   <dispatch>/default.xqy</dispatch>
</rewriter>
```

18.0 Template Driven Extraction (TDE)

Template Driven Extraction (TDE) enables you to define a relational lens over your document data, so you can query parts of your data using SQL or the Optic API. Templates let you specify which parts of documents make up rows in a view. You can also use templates to define a semantic lens, specifying which values from a document make up triples in the triple index.

TDE enables you to generate rows and triples from ingested documents based on predefined templates that describe the following:

- The input data to match
- The data transformations that apply to the matched data
- The final data projections that are translated into indexed data.

TDE enables you to access the data in your documents in several ways, without changing the documents themselves. A relational lens is useful when you want to let SQL-savvy users access your data and when users want to create reports and visualizations using tools that communicate using SQL. It is also useful when you want to join entities and perform aggregates across documents. A semantic lens is useful when your documents contain some data that is naturally represented and queried as triples, using SPARQL.

TDE is applied during indexing at ingestion time and serves the following purposes:

- SQL/Relation indexing. TDE allows the user to map parts of an XML or JSON document into SQL rows. With a TDE template instance, users can create different rows and describe how each column in a row is constructed using the extracted data from a document. For details, see Creating Template Views in the *SQL Data Modeling Guide*.
- Custom Embedded Triple Extraction. TDE enables users to ingest triples that do not follow the `sem:triple` schema. A user can define many triple projections in a single template, where each projection specifies the different parts of a document that are mapped to subjects, predicates or objects. For details, see Using a Template to Identify Triples in a Document in the *Semantics Developer's Guide*.
- Entity Services Data Models. For details, see Creating and Managing Models in the *Entity Services Developer's Guide*.

TDE data is also used by the Optic API, as described in "Optic API for Multi-Model Data Access" on page 305.

> **Note:** The `tde-admin` role is required in order to insert a template into the schema database.

The main topics in this chapter are:

- Security on TDE Documents
- Template View Elements
- JSON Template Structure
- Template Dialect and Data Transformation Functions
- Validating and Inserting a Template
- Templates and Non-Conforming Documents
- Enabling and Disabling Templates
- Deleting Templates

18.1 Security on TDE Documents

Operations on template documents are controlled by:

The `http://marklogic.com/xdmp/tde` collection, which is a protected collection that contains TDE template documents.

The `tde-admin` role, which is required to access the TDE protected collection.

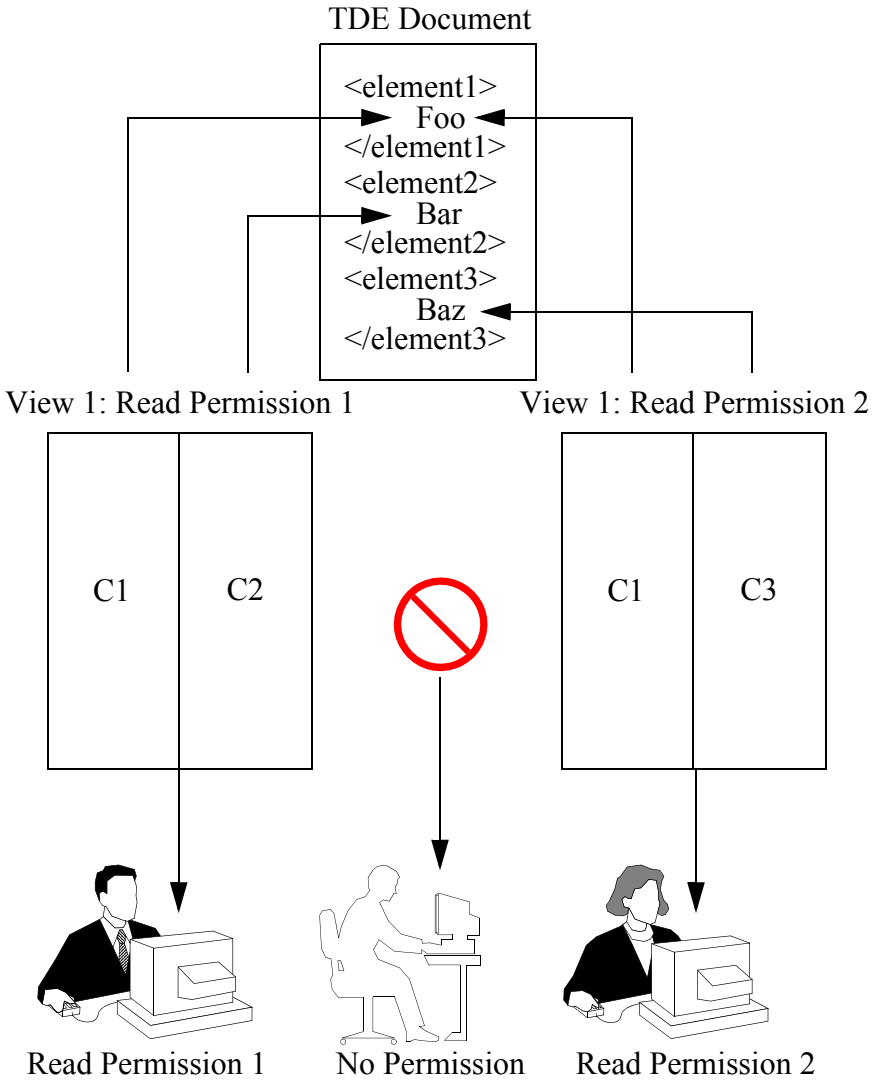
The `tde-view` role, which is required to view documents in the TDE protected collection. Access to views can be further restricted by setting additional permissions on the template documents that define the views. Since the same view can be declared in multiple templates loaded with different permissions, the access to views should be controlled at the column level as follows:

Column level read permissions are implicit and are derived from the read permissions set on the template documents. Permissions on a column are not required to be identical and are ORed together. A user with a role that has at least one of the read permissions set on a column will be able to see the column.

If a user does not have permissions on any of the view's columns, the view itself is not visible.

For example, as shown in the illustration below:

- Template document `TD1` creates view `View 1` with column `C1` and `C2`. Template document `TD1` was loaded with Read Permission 1.
- Template document `TD2` creates view `View 1` with column `C1` and `C3`. Template document `TD2` was loaded with Read Permission 2.
- Users with `Permission 1` have access to columns `C1` and `C2` at query time.
- Users with `Permission 2` have access to columns `C1` and `C3` at query time.
- Users without `Permission 1` or `Permission 2` will not have access to `View 1` or any of its columns.



With this design:

- Users can see columns referenced in templates they have access to.
- Users cannot see additional columns referenced in templates they do not have access to.

If a document in a TDE protected collection makes use of Element Level Security, under most circumstances only the unprotected elements will be extracted; the protected elements will be ignored. For details on Element Level Security, see Element Level Security in the *Security Guide*.

18.2 Template View Elements

A template contains the elements and child elements shown in the table below.

> **Note:** When creating a JSON template, substitute the dash (-) with an upper-case character. For example, `collections-and` becomes `collectionsAnd`. For the complete structure of a JSON template, see JSON Template Structure

Element	Description
`description`	Optional description of the template.

`collections`
`  collection`
`  collections-and`
`  collection`
	Optional collection scopes. Multiple collection scopes can be ORed or ANDed. See “Collections” on page 290.

`directories` `  directory`	Optional directory scopes. Multiple directory scopes are ORed together. See “Directories” on page 291.
`vars` `  var`	Optional intermediate variables extracted at the current context level. See “Variables” on page 295.

Element	Description

`rows`
`  row`
`   schema-name`
`   view-name`
`   view-layout`
`     sparse`
`     identical`
`   columns`
`     column`
`       name`
`       scalar-type`
`       val`
`       nullable`
`       default`
`       invalid-values`
`          ignore`
`          reject`
`       reindexing`
`          hidden`
`          visible`
`       collation`
	These elements are used for template views, as described in Creating Template Views in the *SQL Data Modeling Guide*.
	
	`rows` is a sequence of `row` descriptions and mappings, as described in Row in the *SQL Data Modeling Guide*.
	
	`columns` is sequence of `column` descriptions and mappings, as described in Columns in the *SQL Data Modeling Guide*.
	
	`scalar-type` is the type for the `val`. See “Type Casting” on page 299.

`triples`
`  triple`
`    subject`
`      val`
`      invalid-values`
`    predicate`
`      val`
`      invalid-values`
`    object`
`      val`
`      invalid-values`
	These elements are used for triple-extraction templates, as described in Using a Template to Identify Triples in a Document in the *Semantics Developer's Guide*.
	
	`triples` contains a sequence of triple extraction descriptions. Each `triple` description defines the data mapping for the `subject`, `predicate` and `object`.
	
	An extracted triple's graph cannot be specified through the template. The graph is implicitly defined by the document's collection similar to embedded triples.

`templates` `  template`	Optional sequence of sub-templates. For details, see Creating Views from Multiple Templates and Creating Views from Nested Templates in the *SQL Data Modeling Guide*.
`path-namespaces` `  path-namespace`	Optional sequence of namespace bindings. See “path-namespaces” on page 291.
`context` `  invalid-values`	The lookup node that is used for template activation and data extraction. See “Context” on page 292.
`enabled`	A boolean that specifies whether the template is enabled (`true`) or disabled (`false`). Default value is `true`.

The `context`, `vars`, and `columns` identify XQuery elements or JSON properties by means of path expressions. Path expressions are based on XPath, which is described in XPath Quick Reference in the *XQuery and XSLT Reference Guide* and "Traversing JSON Documents Using XPath" on page 357.

18.3 JSON Template Structure

Below is the structure of a view template in JSON.

```
{
  "template":{
    "description":"test template",
    "context":"context1",
    "pathNamespace":[
      {
        "prefix":"sem",
        "namespaceUri":"http://semantics"
      },
      {
        "prefix":"tde",
        "namespaceUri":"http://tde"
      }
    ],
    "collections":[
      "colc1",
      "colc4",
      { "collectionsAnd":["colc2","colc3"]},
      { "collectionsAnd":["colc5","colc6"]}
    ],
    "directories":["dir1","dir2"],
    "vars":[
      {
        "name":"myvar1",
        "val":"someVal"
      }
    ],
    "rows":[
      {
        "schemaName":"schemaA",
        "viewName":"viewA",
        "viewLayout":"sparse",
        "columns":[
          {
            "name":"A",
            "scalarType":"int",
            "val":"someVal",
            "nullable":false,
            "default":"'1'",
            "invalidValues":"ignore"
          },
          {
            "name":"B",
            "scalarType":"int",
```

```
          "val":"someVal",
          "nullable":true,
          "invalidValues":"ignore"
        }
      ]
    },
    {
      "schemaName": ...
      ...
    }
  ],
  "triples":[
    {
      "subject":{
        "val":"someVal",
        "invalidValues":"ignore"
      },
      "predicate":{
        "val":"someVal"
      },
      "object":{
        "val":"someVal"
      }
    },
    {
      "subject": ...
      ...
    }
  ],
  "templates":[
    {
      "context":"context2",
      "vars":[
        {
          "name":"myvar2",
          "val":"someval"
        }
      ],
      "rows":[
        {
          "schemaName":"schemaA",
          "viewName":"viewC",
          "viewLayout":"sparse",
          "columns":[
            {
              "name":"A",
              "scalarType":"string",
              "val":"someVal",
              "collation":"http://marklogic.com/collation/fr"
            },
            {
              "name":"B",
              "scalarType":"int",
              "val":"someVal"
```

```
                    }
                  ]
                }
              ]
            }
          ]
        }
      }
```

18.3.1 Collections

A `<collections>` section defines the scope of the template to be confined only to documents in specific collections. The `<collections>` section is a top level OR of a sequence of:

- `<collection>` that scope the template to a specific collection.
- `<collections-and>` that contains a sequence of `<collection>` that are ANDed together.

The following collection logical combinations are possible:

ORed collections:

```
<collections>
  <collection>A</collection>
  <collection>B</collection>
</collections>
```

ANDed collections:

```
<collections>
  <collections-and>
    <collection>A</collection>
    <collection>B</collection>
  </collections-and>
</collections>
```

OR of ANDed collections:

```
<collections>
  <collection>A</collection>
  <collection>B</collection>
  <collections-and>
    <collection>C</collection>
    <collection>D</collection>
  </collections-and>
  <collections-and>
    <collection>E</collection>
    <collection>F</collection>
  </collections-and>
</collections>
```

18.3.2 Directories

A `<directories>` section defines the scope of the template to be confined only to documents in specific directories. The `<directories>` section is a top level OR of a sequence of `<directory>` elements that scope the template to a specific directory.

18.3.3 path-namespaces

A `<path-namespaces>` section is a top level of one or more `<path-namespace>` elements, which contain:

- `<prefix>` the namespace prefix.
- `<namespace-uri>` the namespace URI.

For example, a path namespace binding can be specified in the template as follows:

```
<path-namespaces>
  <path-namespace>
    <prefix>wb</prefix>
    <namespace-uri>http://marklogic.com/wb</namespace-uri>
  </path-namespace>
</path-namespaces>
```

The namespace prefix definitions are stored in the template documents and not in the configuration of the target database. Otherwise, templates cannot be compiled into code without knowing the target database configuration that uses them.

18.3.4 Context

The `context` tag defines the lookup node that is used for template activation and data extraction. Path expressions occurring inside `vars`, `rows`, or `triples` are relative to the context element of their parent template. The `context` defines an anchor in the XML/JSON tree where data is collected by walking up and down the tree relative to the anchor. Any indexable path expression is valid in the `context` element, therefore predicates are allowed. The `context` element of a sub-template is relative to the `context` element of its parent template.

For example:

```
<context>/Employee</context>

<context>/MedlineCitation/Article</context>
```

For performance and security reasons, your path expressions are limited to a subset of XPath. For more details, see Template Driven Extraction (TDE) in the *XQuery and XSLT Reference Guide*.

You can specify an `invalid-values` element to control the behavior when the context expression cannot be evaluated. The possible `invalid-values` settings are:

- `ignore` — The extraction will be skipped for the node that resulted in an exception thrown during the evaluation of the `context` expression.
- `reject` — The server will generate an error when the document is inserted and reject the document. This is the default setting.

It is important to understand that context defines the node from which to extract a single row. If you want to extract multiple rows from the document, the context must be set to the parent element of those rows. For example, you have “order” documents that are as structured as follows:

```
<order>
  <order-num>10663</order-num>
  <order-date>2017-01-15</order-date>
  <items>
    <item>
      <product>SpeedPro Ultimate</product>
      <price>999</price>
      <quantity>1</quantity>
    </item>
    <item>
      <product>Ladies Racer Helmet</product>
      <price>115</price>
      <quantity>1</quantity>
    </item>
  </items>
</order>
```

Each order document contains one or more `<item>` nodes. You want to create a view template that extracts the `<product>`, `<price>`, and `<quantity>` values from each `<item>` node. A context of `/order` and column values, such as `items/item/product`, will trigger a single row extraction for the entire document, so the only way this will work is if the document has only one `<item>` node. To extract the content of all of the `<item>` nodes as multiple rows, the context must be `/order/items/item`. In this case, if you wanted to also extract `<order-num>`, the column value would be `../../order-num`.

Note: The `context` can be any path validated by `cts:valid-tde-context`. It may contain wildcards, such as '*', but, for performance reasons, wildcards should not be used unless their value outweighs the performance costs. It is best to use collection or directory scoping when wildcards are used in the context.

Below is the complete grammar for the Restricted XPath, including all the supported constructs.

```
RestrictedPathExpr ::= "/" |(PathExpr)* (("/" | "//") LeafExpr Predicates)
                     | SpecialFunctionExpr
SpecialFunctionExpr::= ( "fn:doc(" ArgsExpr ")" )
                     | ( "xdmp:document-properties(" ArgsExpr ")" )
                     | ( "xdmp:document-locks(" ArgsExpr ")" )
LeafExpr           ::= "(" UnionExpr ")" | LeafStep
PathExpr           ::= ("/" RelativePathExpr?) | ("//" RelativePathExpr)
                     | RelativePathExpr
RelativePathExpr   ::= UnionExpr | "(" UnionExpr ")"
UnionExpr          ::= GeneralStepExpr ("|" GeneralStepExpr)*
GeneralStepExpr    ::= ("/" | "//")? StepExpr (("./" | ".//")? StepExpr)*
StepExpr           ::= ForwardStep Predicates
ForwardStep        ::= (ForwardAxis AbbreviatedForwardStep)
                     | AbbreviatedForwardStep
AbbreviatedForwardStep ::= "." | ("@" NameTest) | NameTest | KindTest
LeafStep           ::= ("@"QName) | QName
NameTest           ::= QName | Wildcard
Wildcard           ::= "*" | "<" NCName ":" "*" ">" | "<" "*" ":" NCName ">"
QName              ::= PrefixedName | UnprefixedName
PrefixedName       ::= Prefix ":" LocalPart
UnprefixedName     ::= LocalPart
Prefix             ::= NCName
LocalPart          ::= NCName
NCName             ::= Name - (Char* ":" Char*)/* An XML Name, minus the ":" */
Name               ::= NameStartChar (NameChar)*

Predicates         ::= Predicate*
Predicate          ::= PredicateExpr | "[" Digit+ "]"
Digit              ::= [0-9]
PredicateExpr      ::= "[" PredicateExpr "and" PredicateExpr "]"
                     | "[" PredicateExpr "or" PredicateExpr  "]"
                     | "[" ComparisonExpr "]" | "[" FunctionExpr "]"
ComparisonExpr     ::= RelativePathExpr GeneralComp SequenceExpr
                     | RelativePathExpr ValueComp Literal
                     | PathExpr
FunctionExpr       ::= FunctionCall GeneralComp SequenceExpr
```

```
                    | FunctionCall ValueComp Literal
                    | FunctionCall
GeneralComp        ::= "=" | "!=" | "<" | "<=" | ">" | ">="
ValueComp          ::= "eq" | "ne" | "lt" | "le" | "gt" | "ge"
SequenceExpr       ::= Literal+
Literal            ::= NumericLiteral | StringLiteral

KindTest           ::= ElementTest
                    | AttributeTest
                    | CommentTest
                    | TextTest
                    | ArrayNodeTest
                    | ObjectNodeTest
                    | BooleanNodeTest
                    | NumberNodeTest
                    | NullNodeTest
                    | AnyKindTest
                    | DocumentTest
                    | SchemaElementTest
                    | SchemaAttributeTest
                    | PITest
TextTest           ::= "text" "(" ")"
CommentTest        ::= "comment" "(" ")"
AttributeTest       := "attribute" "(" (QNameOrWildcard ("," QName)?)? ")"
ElementTest        ::= "element" "(" (QNameOrWildcard ("," QName "?"?)?)? ")"

ArrayNodeTest      ::= "array-node" "(" NCName? ")"
ObjectNodeTest     ::= "object-node" "(" NCName? ")"
BooleanNodeTest    ::= "boolean-node" "(" ")"
NumberNodeTest     ::= "number-node" "(" ")"
NullNodeTest       ::= "null-node" "(" ")"
AnyKindTest        ::= "node" "(" ")"
DocumentTest       ::= "document-node" "(" (ElementTest | SchemaElementTest)
                                            ? ")"
SchemaElementTest  ::= "schema-element" "(" QName ")"
SchemaAttributeTest::= "schema-attribute" "(" QName ")"
PITest             ::= "processing-instruction" "( "(NCName | StringLiteral)
                                                  ? ")"
QNameOrWildcard    ::= QName | "*"
```

18.3.5 Variables

Variables are intermediate data projections needed for data transformation and are defined under `var` elements. Variables can reference other variables inside their transformation section `val`, for the cases where several intermediate projection/transformations are needed before the last projection into the column/triple. The expression inside the `val` code is relative to the context element of the current template in which the `var` is defined. See "Template Dialect and Data Transformation Functions" on page 296 for the types of expressions allowed in a `val`.

For example:

```
<context>/northwind/Orders/Order</context>

.......

<vars>
    <var>
      <name>OrderID</name>
      <val>./@OrderID</val>
    </var>
</vars>

.......

<column>
    <name>OrderID</name>
    <scalar-type>long</scalar-type>
    <val>$OrderID</val>
</column>
```

Note: You do not type variable values in the `var` description. Rather, the variable value is typed in the `column` description.

18.4 Template Dialect and Data Transformation Functions

Templates support a dialect using a subset of XQuery with limited functionalities where only a subset of functions are available.

The template dialect supports the following types of expressions described in the Expressions section of the An XML Query Language specification:

- Path Expressions
- Sequence Expressions
- Arithmetic Expressions
- Comparison Expressions
- Logical Expressions
- Conditional Expressions
- Expressions on SequenceTypes

More complex operations like looping, FLWOR statements, iterations, and XML construction are not supported within the dialect. The property axis `property::` is also not supported.

The supported XQuery functions are listed in the following sections:

- Date and Time Functions
- Logical Functions and Data validation
- String Functions
- Type Casting
- Mathematical Functions
- Miscellaneous Functions

Note: Templates only support XQuery functions. JavaScript functions are not supported.

18.4.1 Date and Time Functions

- `fn:adjust-date-to-timezone`
- `fn:adjust-dateTime-to-timezone`
- `fn:adjust-time-to-timezone`
- `fn:month-from-date`
- `fn:month-from-dateTime`
- `fn:months-from-duration`
- `fn:seconds-from-dateTime`
- `fn:seconds-from-duration`
- `fn:seconds-from-time`
- `fn:minutes-from-dateTime`
- `fn:minutes-from-duration`

- `fn:minutes-from-time`
- `fn:timezone-from-date`
- `fn:timezone-from-dateTime`
- `fn:timezone-from-time`
- `fn:year-from-date`
- `fn:year-from-dateTime`
- `fn:years-from-duration`
- `fn:day-from-date`
- `fn:day-from-dateTime`
- `fn:days-from-duration`
- `fn:format-date`
- `fn:format-dateTime`
- `fn:format-time`
- `fn:hours-from-dateTime`
- `fn:hours-from-duration`
- `fn:hours-from-time`
- `xdmp:dayname-from-date`
- `xdmp:quarter-from-date`
- `xdmp:week-from-date`
- `xdmp:weekday-from-date`
- `xdmp:yearday-from-date`
- `sql:dateadd`
- `sql:datediff`
- `sql:datepart`
- `sql:day`
- `sql:seconds`
- `sql:dayname`
- `sql:timestampadd`
- `sql:hours`
- `sql:timestampdiff`
- `sql:minutes`
- `sql:week`
- `sql:month`
- `sql:weekday`
- `sql:monthname`
- `sql:year`
- `sql:quarter`
- `sql:yearday`
- `xdmp:parse-dateTime`
- `xdmp:parse-yymmdd`

18.4.2 Logical Functions and Data validation

- `fn:boolean`
- `fn:empty`
- `fn:exists`
- `fn:false`
- `fn:not`
- `fn:true`

18.4.3 String Functions

- `fn:codepoint-equal`
- `fn:codepoints-to-string`
- `fn:compare`
- `fn:concat`
- `fn:contains`
- `fn:encode-for-uri`
- `fn:ends-with`
- `fn:escape-html-uri`
- `fn:escape-uri`
- `fn:format-number`
- `fn:insert-before`
- `fn:iri-to-uri`
- `fn:last`
- `fn:lower-case`
- `fn:matches`
- `fn:normalize-space`
- `fn:normalize-unicode`
- `fn:position`
- `fn:remove`
- `fn:replace`
- `fn:reverse`
- `fn:starts-with`
- `fn:string-join`
- `fn:string-length`
- `fn:string-to-codepoints`
- `fn:subsequence`
- `fn:substring`
- `fn:substring-after`
- `fn:substring-before`
- `fn:tokenize`
- `fn:translate`

- `fn:upper-case`

18.4.4 Type Casting

- `number`
- `string`
- `decimal`
- `integer`
- `long`
- `int`
- `short`
- `byte`
- `float`
- `double`
- `boolean`
- `date`
- `time`
- `dateTime`
- `gDay`
- `gMonth`
- `gYear`
- `gYearMonth`
- `gMonthDay`
- `duration`
- `dayTimeDuration`
- `yearMonthDuration`
- `castable-as`
- `anyURI`
- IRI (Internationalized Resource Identifier)

18.4.5 Mathematical Functions

- `fn:abs`
- `fn:round`
- `fn:ceiling`
- `fn:round-half-to-even`
- `fn:floor`

And all the math (http://marklogic.com/xdmp/math namespace) built-in functions except aggregate functions like variance and stddev.

18.4.6 Miscellaneous Functions

- `xdmp:node-uri`

- `xdmp:node-kind`
- `xdmp:path`
- `xdmp:type`
- `xdmp:node-metadata-value`
- `xdmp:node-metadata`
- `sem:uuid`
- `sem:uuid-string`
- `sem:bnode`
- `sem:datatype`
- `sem:sameTerm`
- `sem:lang`
- `sem:iri-to-QName`
- `sem:iri`
- `sem:QName-to-iri`
- `sem:unknown`
- `sem:unknown-datatype`
- `sem:invalid`
- `sem:invalid-datatype`
- `sem:typed-literal`
- `cts:point`
- `fn:head`
- `fn:tail`
- `fn:base-uri`
- `fn:document-uri`
- `fn:lang`
- `fn:local-name`
- `fn:name`
- `fn:namespace-uri`
- `fn:node-name`
- `fn:number`
- `fn:root`
- `fn:min`
- `fn:max`
- `fn:sum`
- `fn:count`
- `fn:avg`

18.5 Validating and Inserting a Template

Note: The `tde-admin` role is required in order to insert a template into the schema database.

Note: The default collation for string values in a TDE template is `codepoint`. If you are having problems joining columns that use a different collation, you will need to change the TDE template to use a matching collation, or change the appropriate range indexes to use `codepoint`.

Warning For best performance, it is recommended that you do not configure your content database to use the default Schemas database and instead create your own schemas database for your template documents. If you create multiple content databases to hold documents to be extracted by TDE, each content database should have its own schema database. Failure to do so may result in unexpected indexing behavior on the content databases.

You should always validate your template before inserting your view into a schema database. To validate your view, use the `tde:validate` function as follows:

```
let $viewTemplate :=
<template xmlns="http://marklogic.com/xdmp/tde">
  .....
</template>

return tde:validate($viewTemplate)
```

A valid template will return the following:

```
<map:map xmlns:map="http://marklogic.com/xdmp/map"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <map:entry key="valid">
     <map:value xsi:type="xs:boolean">true</map:value>
  </map:entry>
</map:map>
```

Note: Do not use `xdmp:validate` to validate your template, as this function may miss some validation steps.

After you have confirmed that the view template is valid, you can insert your view template into the schema database used by the content database holding the document data. You can use any method for inserting documents into the database to insert a view template, but you must insert the template document into the `http://marklogic.com/xdmp/tde` collection.

The `tde:template-insert` function is a convenience that validates the template, inserts the template document into the `tde` collection in the schema database (if executed on the content database) with the default permissions, and triggers a re-index of the database.

Note: When a template is inserted, only those document fragments affected by the template are re-indexed.

For example, to define and insert a view template, you would enter the following:

```
xquery version "1.0-ml";

import module namespace tde = "http://marklogic.com/xdmp/tde"
        at "/MarkLogic/tde.xqy";

let $ClinicalView :=
<template xmlns="http://marklogic.com/xdmp/tde">
  <description>populates patients' data</description>
  <context>/Citation/Article</context>
  <rows>
    <row>
      <schema-name>Medical2</schema-name>
      <view-name>Publications</view-name>
      <columns>
        <column>
          <name>ID</name>
          <scalar-type>long</scalar-type>
          <val>../ID</val>
        </column>
        <column>
          <name>ISSN</name>
          <scalar-type>string</scalar-type>
          <val>Journal/ISSN</val>
        </column>
      </columns>
    </row>
  </rows>
</template>

return tde:template-insert("/Template.xml", $ClinicalView)
```

If you use an alternative insert operation, you must explicitly insert the template document into the `http://marklogic.com/xdmp/tde` collection of the schema database used by your content database. For example:

```
return xdmp:document-insert(
      "/Template.xml",
      $ClinicalView,
      (),
      "http://marklogic.com/xdmp/tde")
```

18.6 Templates and Non-Conforming Documents

Once you have inserted a TDE template for a content database, an attempt to insert a document that does not conform to the template may generate and error.

"Doesn't conform" might mean that the template says you should have a `price` element at some path and the column is not nullable, and there's no default value. But the inserted document has no `price` element at that path, or perhaps there is a `price` in the document but it can't be cast to the type of the column.

If the document is already in the database and you add the template, you may not want to delete the non-conforming document, but you do want to be aware of its existence. If you set the log level to `debug`, then in the case where you added a template and some existing documents are non-conforming, you'll get an error in the error log for each document that doesn't get indexed. For details on setting the log level, see Understanding the Log Levels in the *Administrator's Guide*.

If the template is already in place and you try to insert the non-conforming document, there are two possible outcomes:

- The insert fails with an error
- The insert succeeds, but the row with the missing `price` column is skipped (it doesn't get added to the index)

You can control the outcome by setting `invalid-values` in the template to `reject` (reject the non-conforming document and throw an error) or `ignore` (allow the document insert and ignore that row for indexing purposes).

18.7 Enabling and Disabling Templates

Templates can be enabled and disabled by modifying the `<enabled>` flag on the template. Set the `<enabled>` flag to `true` to enable the template or `false` to disable,

For example, to disable the template set the `<enabled>` flag to `false`, as follows:

```
<template xmlns="http://marklogic.com/xdmp/tde">
    <context>/foo/bar</context>
    <enabled>false</enabled>
       ...
</template>
```

Reindexing will start automatically every time a template is enabled or disabled.

18.8 Deleting Templates

Template documents can be safely deleted once they have been disabled and after enough time has elapsed to make sure that the reindexing related to the disabled template has completed.

Accidental deletion of a template can be fixed by:

1. Reinserting the template in a disabled state.

2. Reusing the same template document URI for a new template.

3. Manually reindexing the database.

19.0 Optic API for Multi-Model Data Access

The MarkLogic Optic API makes it possible to perform relational operations on indexed values and documents. The Optic API is not a single API, but rather a set of APIs exposed within the XQuery, JavaScript, and Java languages.

The Optic API can read any indexed value, whether the value is in a range index, the triple index, or rows extracted by a template. The extraction templates, such as those used to create template views described in Creating Template Views in the *SQL Data Modeling Guide*, are a simple, powerful way to specify a relational lens over documents, making parts of your document data accessible via SQL. Optic gives you access to the same relational operations, such as joins and aggregates, over rows. The Optic API also enables document search to match rows projected from documents, joined documents as columns within rows, and dynamic document structures – all performed efficiently within the database and accessed programmatically from your application.

The Optic API allows you to use your data as-is and makes it possible to make use of MarkLogic document and search features using JavaScript or XQuery syntax, incorporating common SQL concepts, regardless of the structure of your data. Unlike SQL, Optic is well suited for building applications and accessing the full range of MarkLogic NoSQL capabilities. Because Optic is integrated into common application languages, it can perform queries within the context of broader applications that perform updates to data and process results for presentation to end users.

The Optic API supports:

- Joins – integrating documents that are frequently updated or that have many relations with a declarative query instead of with a denormalized write
- Grouping – summarizing aggregate properties over many documents
- Exact matches over repeated structures in documents
- Joining Triples – incorporating semantic triples to enrich row data or to link documents and rows
- Document Joins – returning the entire source document to provide context to row data
- Document Query – performing rich full text search to constrain rows in addition to relational filtering

As in the SQL and SPARQL interfaces, you can use the Optic API to build a query from standard operations such as `where`, `groupBy`, `orderBy`, `union`, and `join` but by expressing the operations through calls to JavaScript and XQuery functions. The Optic API enables you to work in the environment of the programming language, taking advantage of variables and functions for benefits such as modularizing plan construction and avoiding the parse errors and injection attacks associated with assembling a query by concatenating strings.

> **Note:** Unlike in SQL, column order is indeterminate in Optic. Notable exceptions of the sort order keys in `orderby` and grouping keys in `groupby`, which specify priority.

There is also an Optic Java Client API, which is described in Optic Java API for Relational Operations in the *Developing Applications With the Java Client API* guide.

This chapter has the following main sections:

- Differences between the JavaScript and XQuery Optic APIs
- Objects in an Optic Pipeline
- Data Access Functions
- Kinds of Optic Queries
- Processing Optic Output
- Expression Functions For Processing Column Values
- Functions Equivalent to Boolean, Numeric, and String Operators
- Node Constructor Functions
- Best Practices and Performance Considerations
- Optic Execution Plan
- Parameterizing a Plan
- Exporting and Importing a Serialized Optic Query

19.1 Differences between the JavaScript and XQuery Optic APIs

The XQuery Optic API and JavaScript Optic API are functionally equivalent. Each is adapted to the features and practices of their respective language conventions, but otherwise both are as consistent as possible and have the same performance profile. You should use the language that best suits your skills and programming environment.

The table below highlights the differences between the JavaScript and XQuery versions of the Optic API.

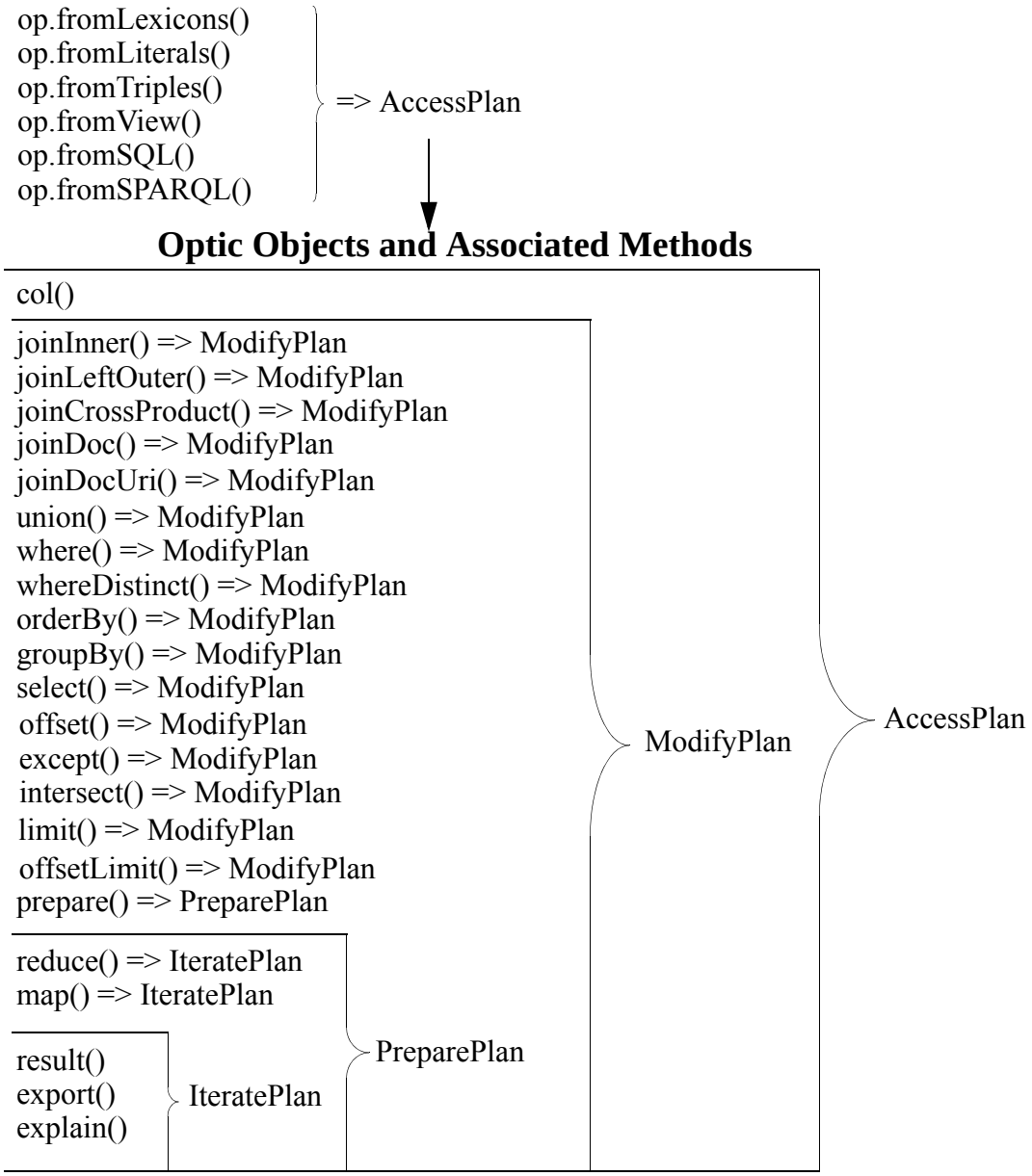
Characteristic	JavaScript	XQuery

Namespaces for proxy functions	Nested namespaces (such as `op.fn.min`)	A module in a separate namespace conforming to the following template (for a prefix, such as `ofn:min`):
		
		`import module namespace`
		`ofn="http://marklogic.com/optic/expression/fn"`
		`at "/MarkLogic/optic/optic-fn.xqy";`
		
		For details, see "XQuery Libraries Required for Expression Functions" on page 344.

Fluent object chaining	Methods that return objects	Functions take a state object as the first parameter and return a state object, enabling use of the XQuery => chaining operator. These black-box objects hold the state of the plan being built in the form of a map. Because these state objects might change in a future release, they should not be modified, serialized or persisted. Chained functions always create a new map instead of modifying the existing map.
Naming convention	camelCase	Hyphen-separated naming convention with the exception of proxy functions for a camelcase original function (such as the `fn:current-dateTime` function).
Unbounded parameters	Allowed	Supported as a single sequence parameter. The sole examples at present are the proxy functions for `fn:concat` and `sem:coalesce`.
Result types	Returns a sequence of objects, with the option to return a sequence of arrays	Returns a map of `sql:rows`, with the option to return an array consisting of a header and rows.

19.2 Objects in an Optic Pipeline

The following graphic illustrates the objects that are used as input and output by the methods in an Optic pipeline.



An Optic query creates a pipeline that applies a sequence of relational operations to a row set. The following are the basic characteristics of the functions and methods used in an Optic query:

- All data access functions (any `from*` function) produce an output row set in the form of an `AccessPlan` object.

- All modifier operations, such as `ModifyPlan.prototype.where`, take an input row set and produce an output row set in the form of a `ModifyPlan` object.
- All composer operations, such as `ModifyPlan.prototype.joinInner`, take two input row sets and produce one output row set in the form of a `ModifyPlan` object.
- The last output row set is the result of the plan.
- The order of operations is constrained only in that the pipeline starts with an accessor operation. For example, you can specify:
  - `select` before a `groupBy` that applies a formula to two columns to specify the input for a `sum` function.
  - `select` after a `groupBy` that applies a formula on the columns that are the output from two `sum` aggregates.

The following is simple example that selects specific columns from the rows in a view and outputs them in a particular order. The pipeline created by this query is illustrated below.

```
const op = require('/MarkLogic/optic');

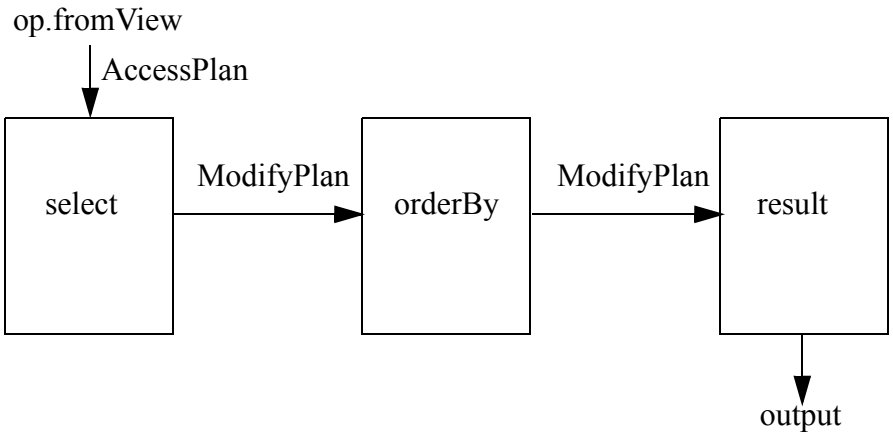
op.fromView('main', 'employees')
   .select(['EmployeeID', 'FirstName', 'LastName'])
   .orderBy('EmployeeID')
   .result();
```

1. The `op.fromView` function outputs an `AccessPlan` object that can be used by all of the API methods.

2. The `AccessPlan.prototype.select` method outputs a `ModifyPlan` object.

3. The `ModifyPlan.prototype.orderBy` method outputs another `ModifyPlan` object.

4. The `ModifyPlan.prototype.result` method consumes the `ModifyPlan` object and executes the plan.



The following example calculates the total expenses for each employee and returns the results in order of employee number.

```
const op = require('/MarkLogic/optic');
const employees = op.fromView('main', 'employees');
const expenses = op.fromView('main', 'expenses');

const Plan =
employees.joinInner(expenses, op.on(employees.col('EmployeeID'),
                                    expenses.col('EmployeeID')))
   .groupBy(employees.col('EmployeeID'), ['FirstName','LastName',
            op.sum('totalexpenses', expenses.col('Amount'))])
   .orderBy('EmployeeID')
Plan.result();
```

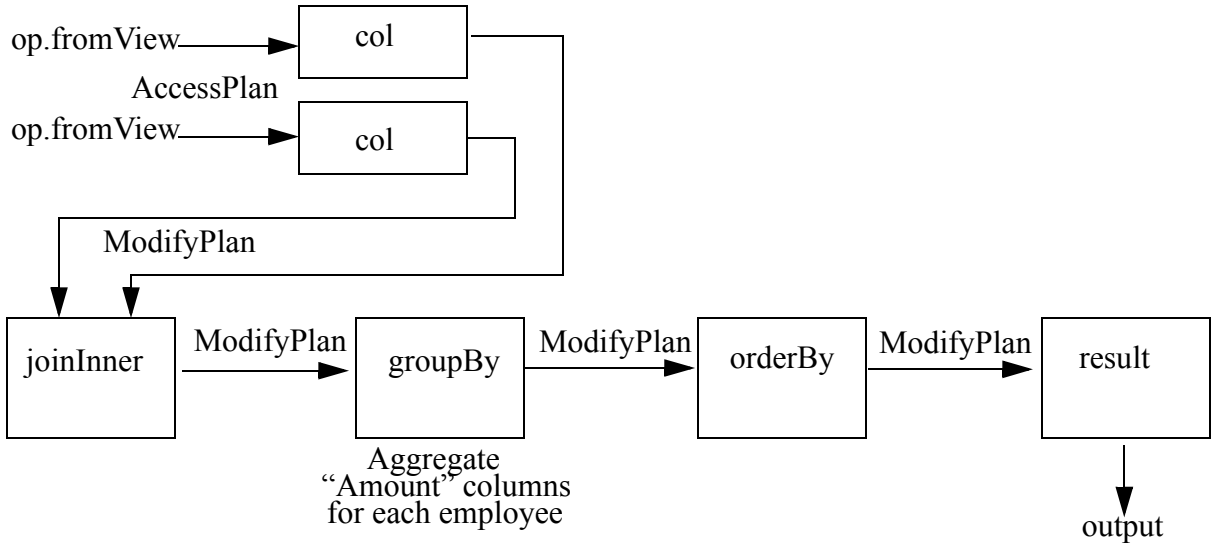
Note: The absence of `.select` is equivalent to a `SELECT *` in SQL, retrieving all columns in a view.

1. The `op.fromView` functions outputs `AccessPlan` objects that are used by the `op.on` function and `AccessPlan.prototype.col` methods to direct the `ModifyPlan.prototype.joinInner` method to join the row sets from both views, which then ouputs them as a single row set in the form of a `ModifyPlan` object.

2. The `ModifyPlan.prototype.groupBy` method calculates the total expenses for each employee and collapes the results into single rows.

3. The `ModifyPlan.prototype.orderBy` method sorts the results and outputs another `ModifyPlan` object.

4. The `ModifyPlan.prototype.result` method consumes the `ModifyPlan` object and executes the plan.



19.3 Data Access Functions

The following functions access data indexed as rows, triples, and lexicons, as well as literal row sets constructed in the program:

JavaScript	XQuery
`op.fromView`	`op:from-view`
`op.fromTriples`	`op:from-triples`
`op.fromLiterals`	`op:from-literals`
`op.fromLexicons`	`op:from-lexicons`
`op.fromSQL`	`op:from-sql`
`op.fromSPARQL`	`op:from-sparql`

The `op.fromView` function accesses indexes created by a template view, as described in Creating Template Views in the *SQL Data Modeling Guide*.

The `op.fromTriples` function accesses semantic triple indexes and abstracts them as rows and columns. Note, however, that the columns of rows from an RDF graph may have varying data types, which could affect joins.

The `op.fromLexicons` function dynamically constructs a view with columns on range-indexes, URI lexicons, and collection lexicons. Lexicons are often joined to enrich data indexed in views. Accessing lexicons from Optic may be useful if your application already has range indexes defined, or if URI or collection information is required for your query.

The `op.fromLiterals` function constructs a literal row set that is similar to the results from a SQL VALUES or SPARQL VALUES statement. This allows you to provide alternative columns to join with an existing view.

The `op.fromSQL` and `op.fromSPARQL` functions dynamically construct a row set based on a SELECT queries template views and triples, respectively

The following sections provide examples of the different data access functions:

- fromView Examples
- fromTriples Example
- fromLexicons Examples
- fromLiterals Examples
- fromSQL Example
- fromSPARQL Example

19.3.1 fromView Examples

Queries using `fromView` retrieve indexed rows exposed over documents. The examples in this section are based on documents and template views described in the SQL on MarkLogic Server Quick Start chapter in the *SQL Data Modeling Guide*.

List all of the employees in order of ID number.

JavaScript:

```
const op = require('/MarkLogic/optic');

op.fromView('main', 'employees')
   .select(['EmployeeID', 'FirstName', 'LastName'])
   .orderBy('EmployeeID')
   .result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

op:from-view("main", "employees")
   => op:select(("EmployeeID", "FirstName", "LastName"))
   => op:order-by("EmployeeID")
   => op:result()
```

You can use Optic to filter rows for specific data of interest. For example, the following query returns the ID and name for employee 3.

JavaScript:

```
const op = require('/MarkLogic/optic');

op.fromView('main', 'employees')
   .where(op.eq(op.col('EmployeeID'), 3))
   .select(['EmployeeID', 'FirstName', 'LastName'])
   .orderBy('EmployeeID')
   .result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

op:from-view("main", "employees")
   => op:where(op:eq(op:col("EmployeeID"), 3))
   => op:select(("EmployeeID", "FirstName", "LastName"))
   => op:order-by("EmployeeID")
   => op:result()
```

The following query returns all of the expenses and expense categories for each employee and return results in order of employee number. Because some information is contained only on the expense reports and some data is only in the employee record, a row join on `EmployeeID` is used to pull data from both sets of documents and produce a single, integrated row set.

JavaScript:

```
const op = require('/MarkLogic/optic');

const employees = op.fromView('main', 'employees');
const expenses = op.fromView('main', 'expenses');

const Plan =
   employees.joinInner(expenses, op.on(employees.col('EmployeeID'),
                       expenses.col('EmployeeID')))
   .select([employees.col('EmployeeID'), 'FirstName', 'LastName',
            'Category', 'Amount'])
   .orderBy(employees.col('EmployeeID'))
Plan.result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

let $employees := op:from-view("main", "employees")
let $expenses  := op:from-view("main", "expenses")

return $employees
   => op:join-inner($expenses, op:on(
          op:view-col("employees", "EmployeeID"),
          op:view-col("expenses",   "EmployeeID")))
   => op:select((op:view-col("employees", "EmployeeID"),
                 "FirstName", "LastName", "Category", "Amount"))
   => op:order-by(op:view-col("employees", "EmployeeID"))
   => op:result()
```

Locate employee expenses that exceed the allowed limit. The `where` operation in this example demonstrates the nature of the Optic chaining pipeline, as it applies to all of the preceding rows.

JavaScript:

```
const op = require('/MarkLogic/optic');

const employees = op.fromView('main', 'employees');
const expenses = op.fromView('main', 'expenses');
const expenselimit = op.fromView('main', 'expenselimit');

const Plan =
   employees.joinInner(expenses, op.on(employees.col('EmployeeID'),
                       expenses.col('EmployeeID')))
   .joinInner(expenselimit, op.on(expenses.col('Category'),
              expenselimit.col('Category')))
   .where(op.gt(expenses.col('Amount'), expenselimit.col('Limit')))
   .select([employees.col('EmployeeID'), 'FirstName', 'LastName',
            expenses.col('Category'), expenses.col('Amount'),
            expenselimit.col('Limit') ])
   .orderBy(employees.col('EmployeeID'))
Plan.result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

let $employees := op:from-view("main", "employees")
let $expenses  := op:from-view("main", "expenses")
let $expenselimit  := op:from-view("main", "expenselimit")

return $employees
   => op:join-inner($expenses, op:on(
                op:view-col("employees", "EmployeeID"),
                op:view-col("expenses", "EmployeeID")))
   => op:join-inner($expenselimit, op:on(
                op:view-col("expenses", "Category"),
                op:view-col("expenselimit", "Category")))
   => op:where(op:gt(op:view-col("expenses", "Amount"),
                op:view-col("expenselimit", "Limit")))
   => op:select((op:view-col("employees", "EmployeeID"),
                "FirstName", "LastName",
                op:view-col("expenses", "Category"),
                op:view-col("expenses", "Amount"),
                op:view-col("expenselimit", "Limit")))
   => op:order-by(op:view-col("employees", "EmployeeID"))
   => op:result()
```

19.3.2 fromTriples Example

The following example returns a list of the people who were born in Brooklyn in the form of a table with two columns, `person` and `name`. This is executed against the example dataset described in Loading Triples in the *Semantics Developer's Guide*.

JavaScript:

```
const op = require('/MarkLogic/optic');
// prefixer is a factory for sem:iri() constructors in a namespace
const resource   = op.prefixer('http://dbpedia.org/resource/');
const foaf   = op.prefixer('http://xmlns.com/foaf/0.1/');
const onto = op.prefixer('http://dbpedia.org/ontology/');

const person = op.col('person');

const Plan =
    op.fromTriples([
        op.pattern(person, onto('birthPlace'), resource('Brooklyn')),
        op.pattern(person, foaf("name"), op.col("name"))
    ])
Plan.result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

let $resource := op:prefixer("http://dbpedia.org/resource/")
let $foaf     := op:prefixer("http://xmlns.com/foaf/0.1/")
let $onto     := op:prefixer("http://dbpedia.org/ontology/")
let $person   := op:col("person")

return op:from-triples((
   op:pattern($person, $onto("birthPlace"), $resource("Brooklyn")),
   op:pattern($person, $foaf("name"), op:col("name"))))
   => op:result()
```

19.3.3 fromLexicons Examples

The `fromLexicons` function may be useful if you already have range indexes defined for use elsewhere in your application. This data access function enables you to incorporate lexicons as another source of data for your query pipeline.

The examples in this section operate on the documents described in Load the Data in the *SQL Data Modeling Guide*.

> **Note:** The `fromLexicons` function queries on range index names, rather than column names in a view. For example, for the `employee` documents, rather than query on `EmployeeID`, you create a range index, named `ID`, and query on `ID`.

First, in the database holding your data, create element range indexes for the following elements: `ID`, `Position`, `FirstName`, and `LastName`. For details on how to create range indexes, see Defining Element Range Indexes in the *Administrator's Guide*.

The following example returns the `EmployeeID` for each employee. The text, `myview`, is prepended to each column name.

JavaScript:

```
const op = require('/MarkLogic/optic');

const Plan =
op.fromLexicons(
       {EmployeeID: cts.elementReference(xs.QName('ID'))});
Plan.result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

op:from-lexicons(
   map:entry(
      "EmployeeID", cts:element-reference(xs:QName("ID"))),
   "myview")
=> op:result()
```

The following example returns the `EmployeeID`, `FirstName`, `LastName`, and the URI of the document holding the data for each employee.

JavaScript:

```
const op = require('/MarkLogic/optic');

const Plan =
op.fromLexicons({
            EmployeeID: cts.elementReference(xs.QName('ID')),
            FirstName: cts.elementReference(xs.QName('FirstName')),
            LastName: cts.elementReference(xs.QName('LastName')),
            URI: cts.uriReference()});
Plan.result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

op:from-lexicons(
map:entry("EmployeeID", cts:element-reference(xs:QName("ID")))
=> map:with("FirstName", cts:element-reference(xs:QName("FirstName")))
=> map:with("LastName", cts:element-reference(xs:QName("LastName")))
=> map:with("uri", cts:uri-reference()))
=> op:result()
```

Every view contains a fragment ID. The fragment ID generated from `op.fromLexicons` can be used to join with the fragment ID of a view. For example, the following returns the `EmployeeID`, `FirstName`, `LastName`, `Position`, and document `URI` for each employee.

JavaScript:

```
const op = require('/MarkLogic/optic');

const empldocid = op.fragmentIdCol('empldocid');
const uridocid = op.fragmentIdCol('uridocid');
const employees = op.fromView('main', 'employees', null, empldocid);
const DFrags = op.fromLexicons({'URI': cts.uriReference()},
                               null, uridocid)

const Plan =
    employees.joinInner(DFrags, op.on(empldocid, uridocid))
             .select(['URI', 'EmployeeID', 'FirstName',
                      'LastName', 'Position']);
Plan.result() ;
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

let $empldocid := op:fragment-id-col("empldocid")
let $uridocid := op:fragment-id-col("uridocid")
let $employees := op:from-view("main", "employees", (), $empldocid)
let $DFrags := op:from-lexicons(map:entry("URI", cts:uri-reference()),
                                        (), $uridocid)

return $employees
   => op:join-inner($DFrags, op:on($empldocid, $uridocid))
   => op:select((op:view-col("employees", "EmployeeID"),
                ("URI", "FirstName", "LastName", "Position")))
   => op:result()
```

19.3.4 fromLiterals Examples

The `fromLiterals` function enables you to dynamically generate rows based on run-time input of arrays and objects of strings. This data access function is helpful for testing and debugging.

Build a table with two rows and return the row that matches the `id` column value of 1:

JavaScript:

```
const op = require('/MarkLogic/optic');
op.fromLiterals([
            {id:1, name:'Master 1', date:'2015-12-01'},
            {id:2, name:'Master 2', date:'2015-12-02'}
            ])
   .where(op.eq(op.col('id'),1))
   .result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

op:from-literals(
    map:entry("columnNames",
              json:to-array(("id", "name", "date")))
    => map:with("rowValues", (
                json:to-array((  1,  "Master 1", "2015-12-01")),
                json:to-array((  2,  "Master 2", "2015-12-02")))))
    => op:where(op:eq(op:col("id"), 1))
    => op:result()
```

Build a table with five rows and return the average values for `group 1` and `group 2`:

JavaScript:

```
const op = require('/MarkLogic/optic');
op.fromLiterals([
            {group:1, val:2},
            {group:1, val:4},
            {group:2, val:3},
            {group:2, val:5},
            {group:2, val:7}
            ])
   .groupBy('group', op.avg('valAvg', 'val'))
   .orderBy('group')
   .result()
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

op:from-literals((
            map:entry("group", 1) => map:with("val", 2),
            map:entry("group", 1) => map:with("val", 4),
            map:entry("group", 2) => map:with("val", 3),
            map:entry("group", 2) => map:with("val", 5),
            map:entry("group", 2) => map:with("val", 7)
            ))
    => op:group-by("group", op:avg("valAvg", "val"))
    => op:order-by("group")
    => op:result()
```

19.3.5 fromSQL Example

The `fromSQL` function enables you to dynamically generate rows based on a SQL SELECT query from template views.

List all of the employees in the employees view:

JavaScript:

```
const op = require('/MarkLogic/optic');

op.fromSQL('SELECT employees.FirstName, employees.LastName \
                  FROM employees')
  .result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

op:from-sql('SELECT employees.FirstName, employees.LastName
                   FROM employees')
    => op:result()
```

19.3.6 fromSPARQL Example

The `fromSPARQL` function enables you to dynamically generate rows based on a SPARQL SELECT query from triples.

List all of the people born in Brooklyn:

JavaScript:

```
const op = require('/MarkLogic/optic');

op.fromSPARQL('SELECT employees.FirstName, employees.LastName \
                   FROM employees')
  .result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

op:from-sparql('PREFIX db: <http://dbpedia.org/resource/>
                PREFIX foaf: <http://xmlns.com/foaf/0.1/>
                PREFIX onto: <http://dbpedia.org/ontology/>
                SELECT ?person ?name
                    WHERE { ?person onto:birthPlace db:Brooklyn;
                    foaf:name ?name .}')
=> op:result()
```

19.4 Kinds of Optic Queries

This section describes some of the kinds of Optic queries. The examples in this section are based on documents and template views described in the SQL on MarkLogic Server Quick Start chapter in the *SQL Data Modeling Guide*.

The topics are:

- Basic Queries
- Aggregates and Grouping
- Row Joins
- Document Joins
- Union, Intersect, and Except
- Document Queries

19.4.1 Basic Queries

Begin using the Optic API by performing a basic query on a view over documents. Querying the view will return rows.

For example, the following lists all of the employee IDs and names in order of ID number.

JavaScript:

```
const op = require('/MarkLogic/optic');

op.fromView('main', 'employees')
   .select(['EmployeeID', 'FirstName', 'LastName'])
   .orderBy('EmployeeID')
   .result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

op:from-view("main", "employees")
   => op:select(("EmployeeID", "FirstName", "LastName"))
   => op:order-by("EmployeeID")
   => op:result()
```

19.4.2 Aggregates and Grouping

Use the MarkLogic Optic API to conveniently perform aggregate functions on values across documents. The following examples perform several operations to get a sense of basic statistics about employee expenses. For information on the `op.math.trunc` and `omath:trunc` proxy functions used in these examples, see "Expression Functions For Processing Column Values" on page 340.

Grouping in Optic differs from SQL. In SQL, the grouping keys are in the GROUP BY statement and the aggregates are separately declared in the SELECT. In an Optic group-by operation, the grouping keys are the first parameter and the aggregates are an optional second parameter. In this way, Optic enables you to aggregate sequences and arrays in a group-by operation and then call expression functions that operate on these sequences and arrays. For example, many of the `math:*` functions, described in "Expression Functions For Processing Column Values" on page 340, take a sequence.

In Optic, instead of applying aggregate functions to the group, a simple column can be supplied. Optic will sample the value of the column for one arbitrary row within the group. This can be useful when the column has the same value in every row within the group; for example, when grouping on a department number but sampling on the department name.

JavaScript:

```
const op = require('/MarkLogic/optic');

op.fromView('main', 'expenses')
  .groupBy(null, [
            op.count('ExpenseReports', 'EmployeeID'),
            op.min('minCharge', 'Amount'),
            op.avg('average', 'Amount'),
            op.max('maxCharge', 'Amount')
          ])
  .select(['ExpenseReports',
           'minCharge',
           op.as('avgCharge', op.math.trunc(op.col('average'))),
           'maxCharge'])
  .result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

import module namespace
     omath="http://marklogic.com/optic/expression/math"
        at "/MarkLogic/optic/optic-math.xqy";

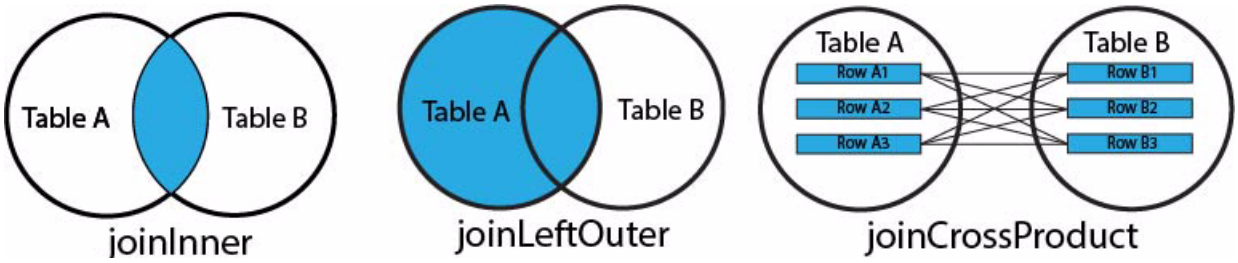
let $expenses  := op:from-view("main", "expenses")

return $expenses
   => op:group-by((), (
         op:count("ExpenseReports", "EmployeeID"),
         op:min("minCharge", "Amount"),
         op:avg("average", "Amount"),
         op:max("maxCharge", "Amount")
      ))
   => op:select(("ExpenseReports", "minCharge",
                op:as("avgCharge", omath:trunc(op:col("average"))),
                "maxCharge"))
   => op:result();
```

19.4.3 Row Joins

Optic supports the following types of row joins:

Method	Description
joinInner	Creates one output row set that concatenates one left row and one right row for each match between the keys in the left and right row sets.
joinLeftOuter	Creates one output row set with all of the rows from the left row set with the matching rows in the right row set, or NULL when there is no match.
joinCrossProduct	Creates one output row set that concatenates every left row with every right row.



The examples in this section join the employees and expenses views to return more information on employee expenses and their categories than what is available on individual documents.

19.4.3.1 joinInner

The following queries make use of the `AccessPlan.prototype.joinInner` and `op:join-inner` functions to return all of the expenses and expense categories for each employee in order of employee number. The join will supplement `employee` data with information stored in separate `expenses` documents. The inner join acts as a filter and will only include those `employees` with `expenses`.

JavaScript:

```
const op = require('/MarkLogic/optic');

const employees = op.fromView('main', 'employees');
const expenses = op.fromView('main', 'expenses');

const Plan =
   employees.joinInner(expenses, op.on(employees.col('EmployeeID'),
                       expenses.col('EmployeeID')))
    .select([employees.col('EmployeeID'), 'FirstName', 'LastName',
             expenses.col('Category'), 'Amount'])
    .orderBy(employees.col('EmployeeID'))
Plan.result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

let $employees := op:from-view("main", "employees")
let $expenses  := op:from-view("main", "expenses")

return $employees
   => op:join-inner($expenses, op:on(
          op:view-col("employees", "EmployeeID"),
          op:view-col("expenses",   "EmployeeID")))
   => op:select((op:view-col("employees", "EmployeeID"),
                 "FirstName", "LastName", "Category"))
   => op:order-by(op:view-col("employees", "EmployeeID"))
   => op:result()
```

Use the `AccessPlan.prototype.where` and `op:where` functions to locate employee expenses that exceed the allowed limit. Join the `employees`, `expenses`, and `category` limits to get a 360 degree view of employee expenses.

JavaScript:

```
const op = require('/MarkLogic/optic');

const employees = op.fromView('main', 'employees');
const expenses = op.fromView('main', 'expenses');
const expenselimit = op.fromView('main', 'expenselimit');

const Plan =
   employees.joinInner(expenses, op.on(employees.col('EmployeeID'),
                       expenses.col('EmployeeID')))
   .joinInner(expenselimit, op.on(expenses.col('Category'),
              expenselimit.col('Category')))
   .where(op.gt(expenses.col('Amount'), expenselimit.col('Limit')))
   .select([employees.col('EmployeeID'), 'FirstName', 'LastName',
            expenses.col('Category'), expenses.col('Amount'),
            expenselimit.col('Limit') ])
   .orderBy(employees.col('EmployeeID'))
Plan.result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

let $employees := op:from-view("main", "employees")
let $expenses  := op:from-view("main", "expenses")
let $expenselimit  := op:from-view("main", "expenselimit")

return $employees
   => op:join-inner($expenses, op:on(
                op:view-col("employees", "EmployeeID"),
                op:view-col("expenses", "EmployeeID")))
   => op:join-inner($expenselimit, op:on(
                op:view-col("expenses", "Category"),
                op:view-col("expenselimit", "Category")))
   => op:where(op:gt(op:view-col("expenses", "Amount"),
                op:view-col("expenselimit", "Limit")))
   => op:select((op:view-col("employees", "EmployeeID"),
                "FirstName", "LastName",
                op:view-col("expenses", "Category"),
                op:view-col("expenses", "Amount"),
                op:view-col("expenselimit", "Limit")))
   => op:order-by(op:view-col("employees", "EmployeeID"))
   => op:result()
```

19.4.3.2 joinLeftOuter

The following queries make use of the `AccessPlan.prototype.joinLeftOuter` and `op:join-left-outer` functions to return all of the expenses and expense categories for each employee in order of employee number, or null values for employees without matching expense records.

JavaScript:

```
const op = require('/MarkLogic/optic');

const employees = op.fromView('main', 'employees');
const expenses = op.fromView('main', 'expenses');

const Plan =
   employees.joinLeftOuter(expenses, op.on(employees.col('EmployeeID'),
                        expenses.col('EmployeeID')))
    .orderBy(employees.col('EmployeeID'))
Plan.result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

let $employees := op:from-view("main", "employees")
let $expenses  := op:from-view("main", "expenses")

return $employees
   => op:join-left-outer($expenses, op:on(
          op:view-col("employees", "EmployeeID"),
          op:view-col("expenses",   "EmployeeID")))
   => op:order-by(op:view-col("employees", "EmployeeID"))
   => op:result()
```

19.4.3.3 joinCrossProduct

The following queries make use of the `AccessPlan.prototype.joinCrossProduct` and `op:join-cross-product` functions to return all of the expenses and expense categories for each employee title (Position) in order of expense Category. If employees with a particular position do not have any expenses under a category, the reported expense is 0.

JavaScript:

```
const op = require('/MarkLogic/optic');

const employees = op.fromView('main', 'employees');
const expenses = op.fromView('main', 'expenses');

expenses.groupBy ('Category')
   .joinCrossProduct(employees.groupBy('Position'))
   .select(null, 'all')
   .joinLeftOuter(
         expenses.joinInner(employees,
                            op.on(employees.col('EmployeeID'),
                            expenses.col('EmployeeID'))
                            )
        .groupBy(['Category', 'Position'],
                 op.sum('rawExpense', expenses.col('Amount'))
                )
        .select(null, 'expensed'),
        [op.on(op.viewCol('expensed', 'Category'),
               op.viewCol('all', 'Category')),
         op.on(op.viewCol('expensed', 'Position'),
               op.viewCol('all', 'Position'))]
   )
   .select([op.viewCol('all', 'Category'),
            op.viewCol('all', 'Position'),
            op.as('expense', op.sem.coalesce(op.col('rawExpense'), 0))
           ])
   .orderBy(['Category', 'Position'])
   .result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";
import module namespace
     osem="http://marklogic.com/optic/expression/sem"
         at "/MarkLogic/optic/optic-sem.xqy";

let $employees := op:from-view("main", "employees")
let $expenses  := op:from-view("main", "expenses")
let $rawExpense  := op:col("rawExpense")

return $expenses
   => op:group-by('Category')
   => op:join-cross-product($employees => op:group-by("Position"))
   => op:select((), 'all')
   => op:join-left-outer(
         $expenses
         => op:join-inner($employees, op:on(
                op:col($employees, "EmployeeID"),
                op:col($expenses, "EmployeeID")
                ))
         => op:group-by(("Category", "Position"),
                  op:sum("rawExpense",  op:col($expenses, "Amount")))
         => op:select((), "expensed"),
           (op:on(op:view-col("expensed", "Category"),
                  op:view-col("all", "Category")),
            op:on(op:view-col("expensed", "Position"),
                  op:view-col("all", "Position")))
         )
   => op:select((op:view-col("all", "Category"),
                 op:view-col("all", "Position"),
                 op:as("expense",
                       osem:coalesce((op:col("rawExpense"), 0)))))
   => op:order-by(("Category", "Position"))
   => op:result();
```

19.4.4 Document Joins

The Optic API provides access not only to rows within views, but also to documents themselves.

Optic support the following types of document joins:

Method	Description
joinDoc	Joins the source documents for rows (especially when the source documents have detail that's not projected into rows). In this case, you should name the fragment id column and use it in the join
joinDocUri	Joins related documents based on document URIs. The `AccessPlan.prototype.joinDocUri` method provides a convenient way to join documents by their URIs. However, if you need more control (for example, left outer joins on related documents), you can use the explicit join with the `cts.uriReference` lexicon to get the fragment id and join the documents on the fragment id. After joining documents, you can use the `op.xpath` function to project or an `xdmp:*` function to add columns with the metadata for documents.

Note: You should minimize the number of documents retrieved by filtering or limiting rows before joining documents.

19.4.4.1 joinDoc

In the examples below, the 'employee' and 'expense' source documents are returned by the `AccessPlan.prototype.joinDoc` or `op:join-doc` function after the row data. The join is done on the document fragment ids returned by `op.fromView`.

JavaScript:

```
const op = require('/MarkLogic/optic');

const empldocid = op.fragmentIdCol('empldocid');
const expdocid = op.fragmentIdCol('expdocid');
const employees = op.fromView('main', 'employees', null, empldocid);
const expenses = op.fromView('main', 'expenses', null, expdocid);

const Plan =
   employees.joinInner(expenses, op.on(employees.col('EmployeeID'),
                        expenses.col('EmployeeID')))
            .joinDoc('Employee', empldocid)
            .joinDoc('Expenses', expdocid)
            .select([employees.col('EmployeeID'),'FirstName',
                     'LastName', expenses.col('Category'), 'Amount',
                     'Employee', 'Expenses'])
            .orderBy(employees.col('EmployeeID'))
Plan.result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

let $empldocid := op:fragment-id-col("empldocid")
let $expdocid := op:fragment-id-col("expdocid")
let $employees := op:from-view("main", "employees", (), $empldocid)
let $expenses := op:from-view("main", "expenses", (), $expdocid)

return $employees
   => op:join-inner($expenses, op:on(
               op:view-col("employees", "EmployeeID"),
               op:view-col("expenses", "EmployeeID")))
   => op:join-doc("Employee", $empldocid)
   => op:join-doc("Expenses", $expdocid)
   => op:select((op:view-col("employees", "EmployeeID"),
               "FirstName", "LastName",
               op:view-col("expenses", "Category"),
               op:view-col("expenses", "Amount"),
               "Employee", "Expenses"))
   => op:order-by(op:view-col("employees", "EmployeeID"))
   => op:result()
```

19.4.4.2 joinDocUri

The following examples show how the `AccessPlan.prototype.joinDocUri` or `op:join-doc-uri` function can be used to return the document URI along with the row data.

JavaScript:

```
const op = require('/MarkLogic/optic');
const empldocid = op.fragmentIdCol('empldocid');
const employees = op.fromView('main', 'employees', null, empldocid);

employees.joinDocUri(op.col('uri'), empldocid)
         .result();
```

XQuery:

```
xquery version "1.0-ml";

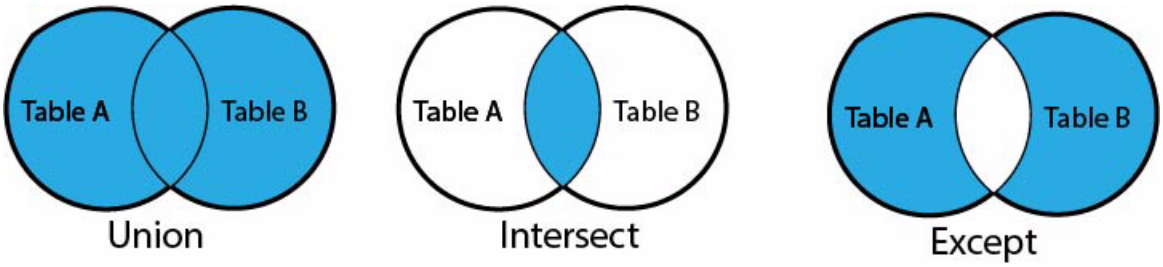
import module namespace op="http://marklogic.com/optic"
      at "/MarkLogic/optic.xqy";

let $empldocid := op:fragment-id-col("empldocid")
return op:from-view("main", "employees", (), $empldocid)
     => op:join-doc-uri(op:col("uri"), $empldocid)
     => op:result()
```

19.4.5 Union, Intersect, and Except

Optic supports the following ways to combine data into new rows:

Method	Description
union	Combines all of the rows from the input row sets. Columns that are present only in some input row sets effectively have a null value in the rows from the other row sets.
intersect	Creates one output row set from the rows that have the same columns and values in both the left and right row sets.
except	Creates one output row set from the rows that have the same columns in both the left and right row sets, but the column values in the left row set do not match the column values in the right row set.



The examples in this section operate on the employees and expenses views to return more information on employee expenses and their categories than what is available on individual documents.

19.4.5.1 union

The following queries make use of the `AccessPlan.prototype.union` and `op:union` functions to return all of the expenses and expense categories for each employee in order of employee number.

JavaScript:

```
const op = require('/MarkLogic/optic');

const employees = op.fromView('main', 'employees');
const expenses = op.fromView('main', 'expenses');

const Plan =
  employees.union(expenses)
           .whereDistinct()
           .orderBy([employees.col('EmployeeID')])
Plan.result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
      at "/MarkLogic/optic.xqy";

let $employees := op:from-view("main", "employees")
let $expenses  := op:from-view("main", "expenses")

return $employees
   => op:union($expenses)
   => op:where-distinct()
   => op:order-by(op:view-col("employees", "EmployeeID"))
   => op:result()
```

19.4.5.2 intersect

The following queries make use of the `AccessPlan.prototype.intersect` and `op:intersect` functions to return the matching columns and values in the tables, `tab1` and `tab2`.

Note: The `op.fromLiterals` function is used for this example because the data set does not contain redundant columns and values.

JavaScript:

```
const op = require('/MarkLogic/optic');

const tab1 = op.fromLiterals([
                     {id:1, val:'a'},
                     {id:2, val:'b'},
                     {id:3, val:'c'}
                     ]);

const tab2 = op.fromLiterals([
                     {id:1, val:'x'},
                     {id:2, val:'b'},
                     {id:3, val:'c'}
                     ]);

tab1.intersect(tab2)
        .orderBy('id')
        .result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
      at "/MarkLogic/optic.xqy";

let $tab1 := op:from-literals((
    map:entry("id", 1) => map:with("val", "a"),
    map:entry("id", 2) => map:with("val", "b"),
    map:entry("id", 3) => map:with("val", "c")
    ))

let $tab2  := op:from-literals((
    map:entry("id", 1) => map:with("val", "x"),
    map:entry("id", 2) => map:with("val", "b"),
    map:entry("id", 3) => map:with("val", "c")
    ))

return $tab1
    => op:intersect($tab2)
    => op:order-by("id")
    => op:result()
```

19.4.5.3 except

The following queries make use of the `AccessPlan.prototype.except` and `op:except` functions to return the columns and values in `tab1` that do not match those in `tab2`.

Note: The `op.fromLiterals` function is used for this example because the data set does not contain redundant columns and values.

JavaScript:

```
const op = require('/MarkLogic/optic');

const tab1 = op.fromLiterals([
                    {id:1, val:'a'},
                    {id:2, val:'b'},
                    {id:3, val:'c'}
                    ]);

const tab2 = op.fromLiterals([
                    {id:1, val:'x'},
                    {id:2, val:'b'},
                    {id:3, val:'c'}
                    ]);

tab1.except(tab2)
        .orderBy('id')
        .result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
      at "/MarkLogic/optic.xqy";

let $tab1 := op:from-literals((
    map:entry("id", 1) => map:with("val", "a"),
    map:entry("id", 2) => map:with("val", "b"),
    map:entry("id", 3) => map:with("val", "c")
    ))

let $tab2  := op:from-literals((
    map:entry("id", 1) => map:with("val", "x"),
    map:entry("id", 2) => map:with("val", "b"),
    map:entry("id", 3) => map:with("val", "c")
    ))

return $tab1
    => op:except($tab2)
    => op:order-by("id")
    => op:result()
```

19.4.6 Document Queries

The MarkLogic Optic API can be combined with other types of queries. Developers can restrict rows based on a document query, even if there are parts of the document that are not part of the row view. The following demonstrates the use of the `AccessPlan.prototype.where` and `op:where` functions to express a document query within the Optic API:

JavaScript:

```
const op = require('/MarkLogic/optic');

op.fromView('main', 'employees')
  .where(cts.andQuery([cts.wordQuery('Senior'),
                       cts.wordQuery('Researcher')]))
  .select(['FirstName', 'LastName', 'Position'])
  .result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

let $employees  := op:from-view("main", "employees")

return $employees
   => op:where(cts:and-query((cts:word-query("Senior"),
                              cts:word-query("Researcher"))))
   => op:select(("FirstName", "LastName", "Position"))
   => op:result()
```

19.5 Processing Optic Output

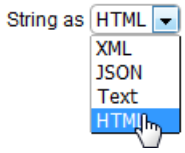
Optic JavaScript queries in Query Console output results in the form of serialized JSON objects. In most cases, you will want to have some code that consumes the Optic output. For example, the following query maps the Optic output to an HTML table.

```
const op = require('/MarkLogic/optic');

let keys = null;
const rowItr = op.fromView('main', 'employees')
  .map(row => {
    if (keys === null) {
      keys = Object.keys(row);
    }
    return `<tr>${keys.map(key => `<td>${row[key]}</td>`)}</tr>`;
  })
  .result();

const rows = Array.from(rowItr).join('\n');
const header = `<tr>${keys.map(key => `<th>${key}</th>`)}</tr>`;
const report = `<table>\n${header}\n${rows}\n</table>`;
report;
```

To view the output as a table in Query Console, select `HTML` from the `String as` menu.



19.6 Expression Functions For Processing Column Values

Optic supports expression functions that represent builtin functions to process column values returned by `op.col` and `op:col`. These include datatype constructors, datetime, duration, numeric, sequence, and string functions. Expression functions are both

- A proxy for a deferred call to a builtin function on the value of a column in each row.
- Nestable for powerful expressions that transform values.

For example, the `math.trunc` function is expressed by the `op.math.trunc` expression function in JavaScript and as `omath:trunc` in XQuery.

For example, the truncate to decimal portion of the returned `'average'` value, do the following:

```
op.math.trunc(op.col('average')) // JavaScript

omath:trunc(op:col('average')) (: XQuery :)
```

The list of JavaScript functions supported by expression functions is shown in the table below. Their XQuery equivalents are also supported, but you must import the respective module libraries listed in "XQuery Libraries Required for Expression Functions" on page 344.

Most every value processing built-in function you would want to use is listed below. In the unlikely event that you want to call a function that is not listed, the Optic API provides a general-purpose `op.call` constructor for deferred calls:

```
op.call(moduleUri, functionName, arg*) => expression

op.call({uri:..., name:..., args:*}) => expression
```

Use the `op.call` function with care because some builtins could adversely affect performance or worse. You cannot call JavaScript or XQuery functions using this function. Instead, provide a map or reduce function to postprocess the results.

Built-in Functions Supported by Optic Expression Functions

cts.tokenize	fn.roundHalfToEven	math.variance	xdmp.integerToOctal
cts.stem	fn.secondsFromDateTime	math.varianceP	xdmp.keyFromQName
fn.abs	fn.secondsFromDuration	rdf.langString	xdmp.lshift64
fn.adjustDateTimeToTimezone	fn.secondsFromTime	rdf.langStringLanguage	xdmp.md5
fn.adjustDateToTimezone	fn.startsWith	sem.bnode	xdmp.monthNameFromDate
fn.adjustTimeToTimezone	fn.string	sem.coalesce	xdmp.mul64
fn.analyzeString	fn.stringJoin	sem.datatype	xdmp.nodeCollections
fn.avg	fn.stringLength	sem.if	xdmp.nodeMetadata
fn.baseUri	fn.stringToCodepoints	sem.invalid	xdmp.nodeMetadataValue
fn.boolean	fn.subsequence	sem.invalidDatatype	xdmp.nodeKind
fn.ceiling	fn.substring	sem.iri	xdmp.nodePermissions
fn.codepointEqual	fn.substringAfter	sem.iriToQName	xdmp.nodeUri
fn.codepointsToString	fn.substringBefore	sem.isBlank	xdmp.not64
fn.compare	fn.sum	sem.isIRI	xdmp.octalToInteger
fn.concat	fn.tail	sem.isLiteral	xdmp.or64
fn.currentDateTime	fn.timezoneFromDate	sem.isNumeric	xdmp.parseDateTime
fn.currentDate	fn.timezoneFromDateTime	sem.lang	xdmp.parseYymmdd
fn.currentTime	fn.timezoneFromTime	sem.langMatches	xdmp.path

Built-in Functions Supported by Optic Expression Functions			
fn.contains	fn.tokenize	sem.QNameToIri	xdmp.position
fn.count	fn.translate	sem.random	xdmp.QNameFromKey
fn.dateTime	fn.true	sem.sameTerm	xdmp.quarterFromDate
fn.dayFromDate	fn.unordered	sem.timezoneString	xdmp.random
fn.dayFromDateTime	fn.upperCase	sem.typedLiteral	xdmp.resolveUri
fn.daysFromDuration	fn.yearFromDate	sem.unknown	xdmp.rshift64
fn.deepEqual	fn.yearFromDateTime	sem.unknownDatatype	xdmp.sha1
fn.defaultCollation	fn.yearsFromDuration	sem.uuid	xdmp.sha256
fn.distinctValues	json.array	sem.uuidString	xdmp.sha384
fn.documentUri	json.arraySize	spell.doubleMetaphone	xdmp.sha512
fn.empty	json.arrayValues	spell.levenshteinDistance	xdmp.step64
fn.encodeForUri	json.object	spell.romanize	xdmp.strftime
fn.endsWith	json.objectDefine	sql.bitLength	xdmp.timestampToWallclock
fn.escapeHtmlUri	json.subarray	sql.dateadd	xdmp.toJSON
fn.exists	json.toArray	sql.datediff	xdmp.type
fn.false	map.contains	sql.datepart	xdmp.urlDecode
fn.floor	map.count	sql.day	xdmp.urlEncode
fn.formatDate	map.entry	sql.dayname	xdmp.wallclockToTimestamp
fn.formatDateTime	map.get	sql.hours	xdmp.weekdayFromDate
fn.formatNumber	map.keys	sql.insert	xdmp.weekFromDate
fn.formatTime	map.map	sql.left	xdmp.xor64
fn.generateId	map.new	sql.minutes	xdmp.yeardayFromDate
fn.head	math.acos	sql.month	xs.anyURI
fn.hoursFromDateTime	math.asin	sql.monthname	xs.boolean
fn.hoursFromDuration	math.atan	sql.octetLength	xs.byte
fn.hoursFromTime	math.atan2	sql.quarter	xs.date
fn.implicitTimezone	math.ceil	sql.rand	xs.dateTime
fn.indexOf	math.correlation	sql.repeat	xs.dayTimeDuration
fn.inScopePrefixes	math.cos	sql.right	xs.decimal

Built-in Functions Supported by Optic Expression Functions			
fn.insertBefore	math.cosh	sql.seconds	xs.double
fn.iriToUri	math.cot	sql.sign	xs.duration
fn.lang	math.covariance	sql.space	xs.float
fn.localName	math.covarianceP	sql.timestampadd	xs.gDay
fn.localNameFromQName	math.degrees	sql.timestampdiff	xs.gMonth
fn.lowerCase	math.exp	sql.week	xs.gMonthDay
fn.matches	math.fabs	sql.weekday	xs.gYear
fn.max	math.floor	sql.year	xs.gYearMonth
fn.min	math.fmod	sql.yearday	xs.hexBinary
fn.minutesFromDateTime	math.frexp	xdmp.add64	xs.int
fn.minutesFromDuration	math.ldexp	xdmp.and64	xs.integer
fn.minutesFromTime	math.linearModel	xdmp.base64Decode	xs.language
fn.monthFromDate	math.linearModelCoeff	xdmp.base64Encode	xs.long
fn.monthFromDateTime	math.linearModelIntercept	xdmp.castableAs	xs.Name
fn.monthsFromDuration	math.linearModelRsquared	xdmp.crypt	xs.NCName
fn.name	math.log	xdmp.crypt2	xs.NMTOKEN
fn.namespaceUriFromQName	math.log10	xdmp.daynameFromDate	xs.negativeInteger
fn.namespaceUri	math.median	xdmp.decodeFromNCName	xs.nonNegativeInteger
fn.namespaceUriForPrefix	math.mode	xdmp.describe	xs.nonPositiveInteger
fn.nilled	math.modf	xdmp.diacriticLess	xs.normalizedString
fn.nodeName	math.percentile	xdmp.elementContentType	xs.numeric
fn.normalizeSpace	math.percentRank	xdmp.encodeForNCName	xs.positiveInteger
fn.normalizeUnicode	math.pi	xdmp.formatNumber	xs.QName
fn.not	math.pow	xdmp.fromJSON	xs.short
fn.number	math.radians	xdmp.getCurrentUser	xs.string
fn.prefixFromQName	math.rank	xdmp.hash32	xs.time
fn.QName	math.sin	xdmp.hash64	xs.token
fn.remove	math.sinh	xdmp.hexToInteger	xs.unsignedByte
fn.replace	math.sqrt	xdmp.hmacMd5	xs.unsignedInt

Built-in Functions Supported by Optic Expression Functions			
fn.resolveQName	math.stddev	xdmp.hmacSha1	xs.unsignedLong
fn.resolveUri	math.stddevP	xdmp.hmacSha256	xs.unsignedShort
fn.reverse	math.tan	xdmp.hmacSha512	xs.yearMonthDuration
fn.root	math.tanh	xdmp.initcap	
fn.round	math.trunc	xdmp.integerToHex	

19.6.1 XQuery Libraries Required for Expression Functions

In XQuery, the following libraries must be imported to use the expression functions for the respective built-in functions.

cts functions:

```
import module namespace octs="http://marklogic.com/optic/expression/cts"
    at "/MarkLogic/optic/optic-cts.xqy";
```

fn functions:

```
import module namespace ofn="http://marklogic.com/optic/expression/fn"
    at "/MarkLogic/optic/optic-fn.xqy";
```

json functions:

```
import module namespace ojson="http://marklogic.com/optic/expression/json"
    at "/MarkLogic/optic/optic-json.xqy";
```

map functions:

```
import module namespace omap="http://marklogic.com/optic/expression/map"
    at "/MarkLogic/optic/optic-map.xqy";
```

math functions:

```
import module namespace omath="http://marklogic.com/optic/expression/math"
    at "/MarkLogic/optic/optic-math.xqy";
```

rdf functions:

```
import module namespace ordf="http://marklogic.com/optic/expression/rdf"
    at "/MarkLogic/optic/optic-rdf.xqy";
```

sem functions:

```
import module namespace osem="http://marklogic.com/optic/expression/sem"
    at "/MarkLogic/optic/optic-sem.xqy";
```

spell functions:

```
import module namespace ospell="http://marklogic.com/optic/expression/spell"
    at "/MarkLogic/optic/optic-spell.xqy";
```

sql functions:

```
import module namespace osql="http://marklogic.com/optic/expression/sql"
    at "/MarkLogic/optic/optic-sql.xqy";
```

xdmp functions:

```
import module namespace oxdmp="http://marklogic.com/optic/expression/xdmp"
    at "/MarkLogic/optic/optic-xdmp.xqy";
```

xs functions:

```
import module namespace oxs="http://marklogic.com/optic/expression/xs"
    at "/MarkLogic/optic/optic-xs.xqy";
```

Expression functions can be nested for powerful expressions that transform values. For example:

```
.select(['countUsers', 'minReputation',
       op.as('avgReputation', op.math.trunc(op.col('aRep'))), 'maxReputation',
       op.as('locationPercent',
             op.fn.formatNumber(op.xs.double(
                 op.divide(op.col('locationCount'),
                           op.col('countUsers'))),'##%'))
])
```

19.7 Functions Equivalent to Boolean, Numeric, and String Operators

Function	SPARQL	SQL	Comments
eq(valueExpression, valueExpression) => booleanExpression eq({left:..., right:...}) => booleanExpression	=	= ==	In expressions, the call should pass a `op.col` value to identify a column.
gt(valueExpression, valueExpression) => booleanExpression gt({left:..., right:...}) => booleanExpression	>	>	
ge(valueExpression, valueExpression) => booleanExpression ge({left:..., right:...}) => booleanExpression	>=	>=	
lt(valueExpression, valueExpression) => booleanExpression lt({left:..., right:...}) => booleanExpression	<	<	
le(valueExpression, valueExpression) => booleanExpression le({left:..., right:...}) => booleanExpression	<=	<=	
ne(valueExpression, valueExpression) => booleanExpression ne({left:..., right:...}) => booleanExpression	!=	!=	
and(booleanExpression+) => booleanExpression and({list:...}) => booleanExpression	&&	AND	

Function	SPARQL	SQL	Comments
or(booleanExpression+) => booleanExpression or({list:...}) => booleanExpression	||	OR	
not(booleanExpression) => booleanExpression not({condition:...}) => booleanExpression	!	NOT	
case(whenExpression+, valueExpression) => valueExpression case({list:..., otherwise:...}) => valueExpression	IF	CASE WHEN ELSE	
when(booleanExpression, valueExpression) => whenExpression when({condition:..., value:...}) => whenExpression		WHEN	
isDefined(col) => booleanExpression isDefined({column: ...}) => booleanExpression	BOUND	IS NULL	
add(numericExpression, numericExpression) => numericExpression add({left:..., right:...}) => numericExpression	+	+	A column must be named with an `op.col` value.
divide(numericExpression, numericExpression) => numericExpression divide({left:..., right:...}) => numericExpression	/	/	

Function	SPARQL	SQL	Comments
modulo(numericExpression, numericExpression) => numericExpression modulo({left:..., right:...}) => numericExpression		%	
multiply(numericExpression, numericExpression) => numericExpression multiply({left:..., right:...}) => numericExpression	*	*	
subtract(numericExpression, numericExpression) => numericExpression subtract({left:..., right:...}) => numericExpression	-	-	

Note: Expressions that use rows returned from a subplan (similar to SQL or SPARQL EXISTS) are not supported in the initial release.

19.8 Node Constructor Functions

Optic provides node constructor functions that enable you to build tree structures. Node constructor functions can:

- Create JSON objects whose properties come from column values or XML elements whose content or attribute values come from column values.
- Insert documents or nodes extracted via `op.xpath` into constructed nodes.
- Create JSON arrays from aggregated arrays of nodes or XML elements from aggregated sequences of nodes.

The table below summarizes the Optic node constructor functions. For details on each function, see the Optic API reference documentation.

Function	Description
`op.jsonArray`	Constructs a JSON array with the specified JSON nodes as items.
`op.jsonBoolean`	Constructs a JSON boolean node with a specified value.
`op.jsonDocument`	Constructs a JSON document with the root content, which must be exactly one JSON object or array node.
`op.jsonNull`	Constructs a JSON null node.
`op.jsonNumber`	Constructs a JSON number node with a specified value.
`op.jsonObject`	Constructs a JSON object with the specified properties. The properties argument is constructed with the prop() function.
`op.jsonString`	Constructs a JSON text node with the specified value.
`op.prop`	Specifies a key expression and value content for a JSON property of a JSON object.
`op.xmlAttribute`	Constructs an XML attribute with a name and atomic value.
`op.xmlComment`	Constructs an XML comment with an atomic value.
`op.xmlDocument`	Constructs an XML document with a root content.
`op.xmlElement`	Constructs an XML element with a name, zero or more attribute nodes, and child content.
`op.xmlPI`	Constructs an XML processing instruction with an atomic value.
`op.xmlText`	Constructs an XML text node.
`op.xpath`	Extracts a sequence of child nodes from a column with node values.

For example, the following query constructs JSON documents, like the one shown below:

```
const op = require('/MarkLogic/optic');
const employees = op.fromView('main', 'employees');

employees.select(op.as('Employee', op.jsonDocument(
    op.jsonObject([op.prop('ID and Name',
                           op.jsonArray([
                              op.jsonNumber(op.col('EmployeeID')),
                              op.jsonString(op.col('FirstName')),
                              op.jsonString(op.col('LastName'))
                  ])),
                  op.prop('Position',
                           op.jsonString(op.col('Position')))
    ])
)))
.result();
```

This query will produce output that looks like the following:

```
{
  "Employee": {
    "ID and Name": [
      42,
      "Debbie",
      "Goodall"
    ],
    "Position": "Senior Widget Researcher"
  }
}
```

19.9 Best Practices and Performance Considerations

Optic does not have a default/implicit limit for the rows or documents returned. Creating plans that return large result sets, such as tens of thousands of rows, may perform poorly. Should you experience performance problems it is a best practice to page the results using the `AccessPlan.prototype.offsetLimit` method or a combination of `AccessPlan.prototype.offset` and `AccessPlan.prototype.limit` methods.

19.10 Optic Execution Plan

An Optic Execution Plan expresses a logical dataflow with a sequence of atomic operations. You use the Optic API to build up a plan definition, creating and modifying objects in a pipeline and then executing the plan with the `PreparePlan.prototype.result` function.

You can use the `PreparePlan.prototype.explain` function to view or save an execution plan. The execution plan definition consists of operations on a row set. These operations fall into the following categories:

- data access – an execution plan can read a row set from a view, graph, or literals where a view can access the triple index or the cross-product of the co-occurrence of range index values in documents.
- row set modification – an execution plan can filter with `where`, `order by`, `group`, project with `select`, and `limit` a row set to yield a modified row set.
- row set composition – an execution plan can combine multiple row sets with `join`, `union`, `intersect`, or `except` to yield a single row set.
- row result processing – an execution plan can specify operations to perform on the final row set including mapping or reducing.

When a view is opened as an execution plan, it has a special property that has an object with a property for each column in the view. The name of the property is the column name and the value of the property is a name object. To prevent ambiguity for columns with the same name in different views, the column name for a view column is prefixed with the view name and a separating period.

The execution plan result can be serialized to CSV, line-oriented XML or JSON, depending on the output mime type. For details on how to read an execution plan, see Execution Plan in the *SQL Data Modeling Guide*.

19.11 Parameterizing a Plan

You use the `op.param` function to create a placeholder that can be substituted for any value. You must specify the value of the parameter when executing the plan.

Because the plan engine caches plans, parameterizing a plan executed previously is more efficient than submitting a new plan.

For example, the following query uses a `start` and `length` parameter to set the `offsetLimit` and an `increment` parameter to increment the value of `EmployeeID`.

```
const op = require('/MarkLogic/optic');

const employees = op.fromView('main', 'employees');

employees.offsetLimit(op.param('start'), op.param('length'))
   .select(['EmployeeID',
            op.as('incremented', op.add(op.col('EmployeeID'),
                                        op.param('increment')))])
   .result(null, {start:1, length:2, increment:1});
```

19.12 Exporting and Importing a Serialized Optic Query

You can use the `IteratePlan.prototype.export` method or `op:export` function to export a serialized form of an Optic query. This enables the plan to be stored as a file and later imported by the `op.import` or `op:import` function or to be used by the `/v1/rows` REST call as a payload. You can recreate the source code used to create an exported plan by means of the `op.toSource` or `op:to-source` function.

For example, to export an Optic query to a file, do the following:

JavaScript:

```
const op = require('/MarkLogic/optic');

const EmployeePlan =
   op.fromView('main', 'employees')
     .select(['EmployeeID', 'FirstName', 'LastName'])
     .orderBy('EmployeeID')
const planObj = EmployeePlan.export();

xdmp.documentInsert("plan.json", planObj)
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

let $plan := op:from-view("main", "employees")
   => op:select(("EmployeeID", "FirstName", "LastName"))
   => op:order-by("EmployeeID")
   => op:export()

return xdmp:document-insert("plan.json",  xdmp:to-json($plan))
```

To import an Optic query from a file and output the results, do the following:

JavaScript:

```
const op = require('/MarkLogic/optic');

op.import(cts.doc('plan.json').toObject())
  .result();
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

op:import(fn:doc("plan.json")/node())
   => op:result()
```

To view the source code representation of a plan, do the following:

JavaScript:

```
const op = require('/MarkLogic/optic');

op.toSource(cts.doc('plan.json'))
```

XQuery:

```
xquery version "1.0-ml";

import module namespace op="http://marklogic.com/optic"
     at "/MarkLogic/optic.xqy";

op:to-source(fn:doc("plan.json"))
```

20.0 Working With JSON

This chapter describes how to work with JSON in MarkLogic Server, and includes the following sections:

- JSON, XML, and MarkLogic
- How MarkLogic Represents JSON Documents
- Traversing JSON Documents Using XPath
- Creating Indexes and Lexicons Over JSON Documents
- How Field Queries Differ Between JSON and XML
- Representing Geospatial, Temporal, and Semantic Data
- Serialization of Large Integer Values
- Character Set Restrictions
- Document Properties
- Working With JSON in XQuery
- Working With JSON in Server-Side JavaScript
- Converting JSON to XML and XML to JSON
- Low-Level JSON XQuery APIs and Primitive Types
- Loading JSON Documents

20.1 JSON, XML, and MarkLogic

JSON (JavaScript Object Notation) is a data-interchange format originally designed to pass data to and from JavaScript. It is often necessary for a web application to pass data back and forth between the application and a server (such as MarkLogic Server), and JSON is a popular format for doing so. JSON, like XML, is designed to be both machine- and human-readable. For more details about JSON, see json.org.

MarkLogic Server supports JSON documents. You can use JSON to store documents or to deliver results to a client, whether or not the data started out as JSON. The following are some highlights of the MarkLogic JSON support:

- You can perform document operations and searches on JSON documents within MarkLogic Server using JavaScript, XQuery, or XSLT. You can perform document operations and searches on JSON documents from client applications using the Node.js, Java, and REST Client APIs.
- The client APIs all have options to return data as JSON, making it easy for client-side application developers to interact with data from MarkLogic.

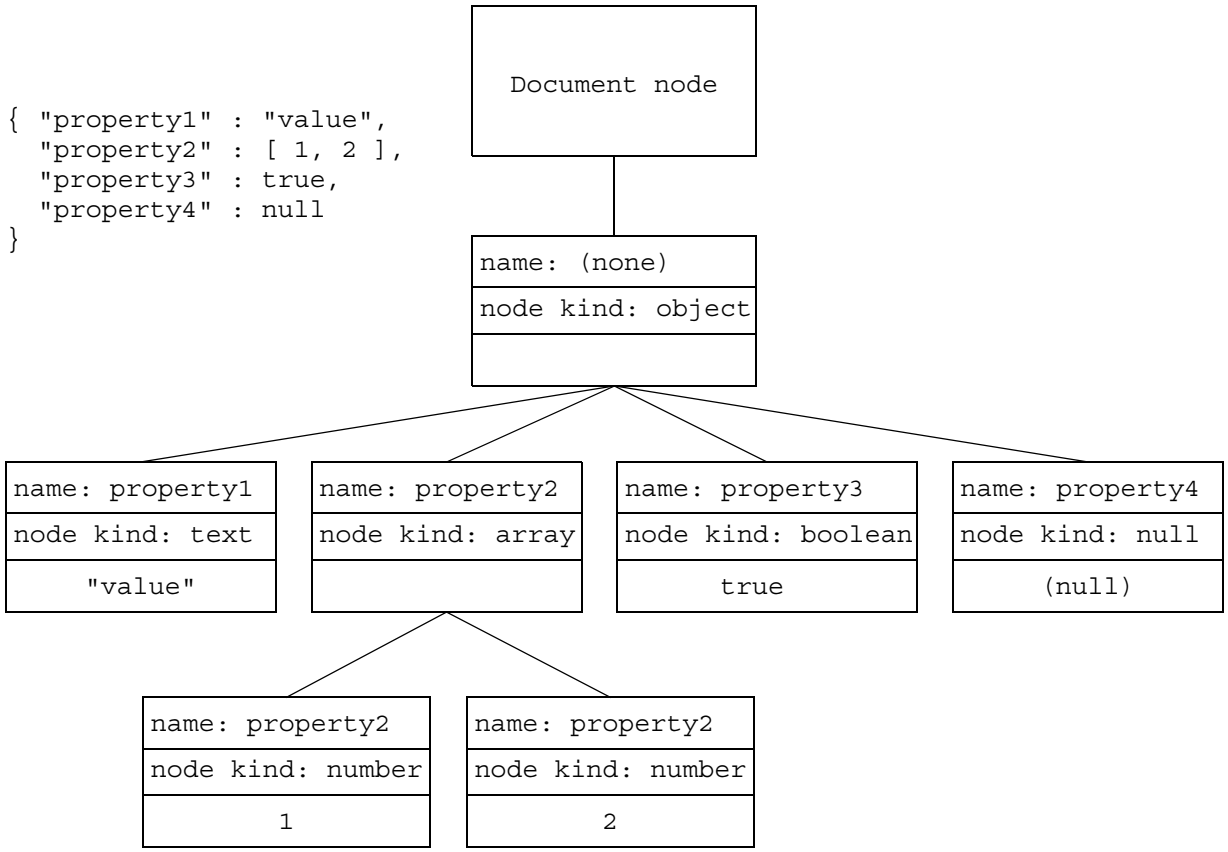
- The REST Client API and the REST Management API accept both JSON and XML input. For example, you can specify queries and configuration information in either format.
- The MarkLogic client APIs provide full support for loading and querying JSON documents. This allows for fine-grained access to the JSON documents, as well as the ability to search and facet on JSON content.
- You can easily transform data from JSON to XML or from XML to JSON. There is a rich set of APIs to do these transformations with a large amount of flexibility as to the specification of the transformed XML and/or the specification of the transformed JSON. The supporting low-level APIs are built into MarkLogic Server, allowing for extremely fast transformations.

20.2 How MarkLogic Represents JSON Documents

MarkLogic Server models JSON documents as a tree of nodes, rooted at a document node. Understanding this model will help you understand how to address JSON data using XPath and how to perform node tests. When you work with JSON documents in JavaScript, you can often handle the contents like a JavaScript object, but you still need to be aware of the differences between a document and an object.

For a JSON document, the nodes below the document node represent JSON objects, arrays, and text, number, boolean, and null values. Only JSON documents contain object, array, number, boolean, and null node types.

For example, the following picture shows a JSON object and its tree representation when stored in the database as a JSON document. (If the object were an in-memory construct rather than a document, the root document node would not be present.)



The name of a node is the name of the innermost JSON property name. For example, in the node tree above, "property2" is the name of both the array node and each of the array member nodes.

```
fn:node-name(fn:doc($uri)/property2/array-node()) ==> "property2"
fn:node-name(fn:doc($uri)/property2[1]) ==> "property2"
```

Nodes which do not have an enclosing property are unnamed nodes. For example, the following array node has no name, so neither do its members. Therefore, when you try to get the name of the node in XQuery using fn:node-name, an empty sequence is returned.

```
let $node := array-node { 1, 2 }
return fn:node-name($node//number-node[. eq 1])
==> an empty sequence
```

20.3 Traversing JSON Documents Using XPath

This section describes how to access parts of a JSON document or node using XPath. You can use XPath on JSON data anywhere you can use it on XML data, including from JavaScript and XQuery code.

The following topics are covered:

- What is XPath?

- Exploring the XPath Examples
- Selecting Nodes and Node Values
- Node Test Operators
- Selecting Arrays and Array Members

20.3.1 What is XPath?

XPath is an expression language originally designed for addressing nodes in an XML data structure. In MarkLogic Server, you can use XPath to traverse JSON as well as XML. You can use XPath expressions for constructing queries, creating indexes, defining fields, and selecting nodes in a JSON document.

XPath is defined in the following specification:

http://www.w3.org/TR/xpath20/#id-sequence-expressions

For more details, see XPath Quick Reference in the *XQuery and XSLT Reference Guide*.

20.3.2 Exploring the XPath Examples

XPath expressions can be used in many different contexts, including explicit node traversal, query construction, and index configuration. As such, many of the XPath examples in this chapter include just an XPath expression, with no execution context.

If you want to use node traversal to explore what is selected by an XPath expression, you can use one of the following patterns as a template in Query Console:

Language	Template

XQuery	`xquery version "1.0-ml";`
	`let $node := xdmp:unquote(`*`json_literal`*`)`
	`return xdmp:describe($node/`*`xpath_expr`*`)`

Server-Side JavaScript	`const node = xdmp.toJSON(`*`jsObject`*`);` `xdmp.describe(node.xpath(`*`xpathExpr`*`));`

The results are wrapped in a call to `xdmp:describe` in XQuery and `xdmp.describe` in JavaScript to clearly illustrate the result type, independent of Query Console formatting.

Note that in XQuery you can apply an XPath expression directly to a node, but in JavaScript, you must use the `Node.xpath` method. For example, `$node/a` vs. `node.xpath('/a/b')`. Note also that the `xpath` method returns a Sequence in JavaScript, so you may need to iterate over the results when using this method in your application.

For example, if you want to explore what happens if you apply the XPath expression "/a" to a JSON node that contains the data {"a": 1}, then you can run one following examples in Query Console:

Language	Template

XQuery	`xquery version "1.0-ml";`
	`let $node := xdmp:unquote('{"a": 1}')`
	`return xdmp:describe($node/a)`

Server-Side JavaScript	`const node = xdmp.toJSON({a: 1});` `xdmp.describe(node.xpath('/a'));`

20.3.3 Selecting Nodes and Node Values

In most cases, an XPath expression selects one or more nodes. Use `data()` to access the value of the node. For example, contrast the following XPath expressions. If you have a JSON object node containing `{ "a" : 1 }`, then first expression selects the number node with name "a", and the second expression selects the value of the node.

```
(: XQuery :)
$node/a ==> number-node { 1 }
$node/a/data() ==> 1

// JavaScript
node.xpath('/a') ==> number-node { 1 }
node.xpath('/a/data()') ==> 1
```

You can use node test operators to limit selected nodes by node type or by node type and name; for details, see "Node Test Operators" on page 360.

A JSON array is treated like a sequence by default when accessed with XPath. For details, see "Selecting Arrays and Array Members" on page 362.

Assume the following JSON object node is in a variable named `$node`.

```
{ "a": {
    "b": "value",
    "c1": 1,
    "c2": 2,
    "d": null,
    "e": {
      "f": true,
      "g": ["v1", "v2", "v3"]
    }
} }
```

Then the table below demonstrates what node is selected by of several XPath expressions applied to the object. You can try these examples in Query Console using the pattern described in "Exploring the XPath Examples" on page 358.

XPath Expression	Result
`$node/a/b`	`"value"`

`$node/a/c1`	`A number node named "c1" with value 1:`
	
	`number-node{ 1 }`

`$node/a/c1/data()`	`1`
`$node/a/d`	`null-node { }`
`$node/a/e/f`	`boolean-node{ fn:true() }`
`$node/a/e/f/data()`	`true`

`$node/a/e/g`	`A sequence containing 3 values:`
	
	`("v1", "v2", "v3")`

`$node/a/e/g[2]`	`"v2"`

`$node/a[c1=1]`	`An object node equivalent to the following JSON:`
	
	`{`
	`  "b": "value",`
	`  "c1": 1,`
	`  "c2": 2,`
	`...`
	`}`

20.3.4 Node Test Operators

You can constrain node selection by node type using the following node test operators.

- `object-node()`
- `array-node()`
- `number-node()`
- `boolean-node()`
- `null-node()`
- `text()`

All node test operators accept an optional string parameter for specifying a JSON property name. For example, the following expression matches any boolean node named “a”:

```
boolean-node("a")
```

Assume the following JSON object is in the in-memory object `$node`.

```
{ "a": {
    "b": "value",
    "c1": 1,
    "c2": 2,
    "d": null,
    "e": {
      "f": true,
      "g": ["v1", "v2", "v3"]
    }
} }
```

Then following table contains several examples of XPath expressions using node test operators. You can try these examples in Query Console using the pattern described in “Exploring the XPath Examples” on page 358.

XPath Expression	Result

`$node//number-node()`
 	
`$node/a/number-node()`
	`A sequence containing two number nodes, one named "c1" and one named "c2":`
	
	`(number-node{1}, number-node{2})`

`$node//number-node()/data()`	`A sequence containing 2 numbers:`
	
	`(1,2)`

`$node/a/number-node("c2")`	`The number node named "c2":`
	
	`number-node{2}`

`$node//text()`	`("value", "v1", "v2", "v3")`

XPath Expression	Result
`$node/a/text("b")`	`"value"`

`$node//object-node()`	A sequence of object nodes equivalent to the following JSON objects:
	
	`{"a": {"b": "value", ... } }`
	`{"b": "value", "c1": 1, ...}`
	`{"f": true, "g": ["v1", "v2", "v3"]}`

`$node/a/e/array-node("g")`	An array node equivalent to the following JSON array: `[ "s1", "s2", "s3" ]`

`$node//node("g")`	An array node, plus a text node for each item in the array.
	
	`[ "s1", "s2", "s3" ]`
	`"s1"`
	`"s2"`
	`"s3"`

20.3.5 Selecting Arrays and Array Members

References to arrays in XPath expressions are treated as sequences by default. This means that nested arrays are flattened by default, so `[1, 2, [3, 4]]` is treated as `[1, 2, 3, 4]` and the `[]` operator returns sequence member values.

To access an array as an array rather than a sequence, use the `array-node()` operator. To access an item in an array rather than the associated node, use the `data()` operator.

> **Note:** Unlike native JavaScript arrays, sequence (array) indices in XPath expressions begin with 1, rather than 0. That is, an XPath expression such as `/someArray[1]` addresses the first item in a JSON array.

Note that the "descendant-or-self" axis ("//") can select both the array node and the array items if you are not explicit about the node type. For example, given a document of the following form:

```
{ "a" : [ 1, 2] }
```

The XPath expression `//node("a")` selects both the array node and two number nodes for the item values 1 and 2.

Assume the following JSON object is in the in-memory object `$node`.

```
{
  "a": [ 1, 2 ],
  "b": [ 3, 4, [ 5, 6 ] ],
```

```
    "c": [
      { "c1": "cv1" },
      { "c2": "cv2" }
    ]
  }
```

Then following table contains examples of XPath expressions accessing arrays and array members. You can try these examples in Query Console using the pattern described in "Exploring the XPath Examples" on page 358.

XPath Expression	Result
`$node/a`	A sequence of number nodes: `(number-node{1}, number-node{2})`
`$node/a/data()`	A sequence of numbers: `(1, 2)`
`$node/array-node("a")`	An array-node with number nodes children, equivalent to the following JSON array: `[1, 2]`
`$node/node("a")`	An array-node with number nodes children, equivalent to the following JSON array: `[1, 2]`
`$node//node("a")`	A sequence of nodes consisting of an array node, and a number node for each item in "a".
`$node/a[1]`	`number-node{1}`
`$node/a[1]/data()`	`1`
`$node/b/data()`	A sequence of numbers. The inner array is flattened when the value is converted to a sequence. `(3, 4, 5, 6)`
`$node/array-node("b")`	All array nodes with name "b". Equivalent to the following JSON array: `[3, 4, [5, 6]]`

XPath Expression	Result

`$node/array-node("b")/array-node()`	`All array nodes contained inside the array named "b". Equivalent to the following JSON array:`
	
	`[5, 6]`

`$node/b[3]`	`number-node{5}`

`$node/c`	`An object node equivalent to the following JSON object:`
	
	`( {"c1": "cv1"}, {"c2": "cv2"} )`

`$node/c[1]`	`An object node equivalent to the following JSON:`
	
	`{ "c1": "cv1" }`

`$node//array-node()/number-node()[data()=2]`	`All number nodes with the value 2 that are children of an array node.`
	
	`number-node{2}`

`$node//array-node()[number-node()/data()=2]`	`All array nodes that contain a number node with a value of 2.`
	
	`[1, 2]`

`$node//array-node()[./node()/text() = "cv2"]`	`All array nodes that contain a member with a text value of "cv2".`
	
	`[ {"c1": "cv1"}, {"c2": "cv2"} ]`

20.4 Creating Indexes and Lexicons Over JSON Documents

You can create path, range, and field indexes on JSON documents. For purposes of indexing, a JSON property (name-value pair) is roughly equivalent to an XML element. For example, to create a JSON property range index, use the APIs and interfaces for creating an XML element range index.

Indexing for JSON documents differs from that of XML documents in the following ways:

- JSON string values are represented as text nodes and indexed as text, just like XML text nodes. However, JSON number, boolean, and null values are indexed separately, rather than being indexed as text.
- Each JSON array member value is considered a value of the associated property. For example, a document containing `{"a":[1,2]}` matches a value query for a property "a" with a value of 1 and a value query for a property "a" with a value of 2.

- You cannot define fragment roots for JSON documents.
- You cannot define a phrase-through or a phrase-around on JSON documents.
- You cannot switch languages within a JSON document, and the `default-language` option on `xdmp:document-load` (XQuery) or `xdmp.documentLoad` (JavaScript) is ignored when loading JSON documents.
- No string value is defined for a JSON object node. This means that field value and field range queries do not traverse into object nodes. For details, see "How Field Queries Differ Between JSON and XML" on page 365.

For more details, see Range Indexes and Lexicons in the *Administrator's Guide*.

20.5 How Field Queries Differ Between JSON and XML

Field word queries work the same way on both XML and JSON, but field value queries and field range queries behave differently for JSON than for XML due to the indexing differences described in "Creating Indexes and Lexicons Over JSON Documents" on page 364.

A complex XML node has a string value for indexing purposes that is the concatenation of the text nodes of all its descendant nodes. There is no equivalent string value for a JSON object node.

For example, in XML, a field value query for "John Smith" matches the following document if the field is defined on the path `/name` and excludes "middle". The value of the field for the following document is "John Smith" because of the concatenation of the included text nodes.

```
<name>
  <first>John</first>
  <middle>NMI</middle>
  <last>Smith</last>
<name>
```

You cannot construct a field that behaves the same way for JSON because there is no concatenation. The same field over the following JSON document has values "John" and "Smith", not "John Smith".

```
{ "name": {
    "first": "John",
    "middle": "NMI",
    "last": "Smith"
}
```

Also, field value and field range queries do not traverse into JSON object nodes. For example, if a path field named "myField" is defined for the path `/a/b`, then the following query matches the document "my.json":

```
xdmp:document-insert("my.json",
  xdmp:unquote('{"a": {"b": "value"}}'));
cts:search(fn:doc(), cts:field-value-query("myField", "value"));
```

However, the following query will not match "my.json" because `/a/b` is an object node (`{"c":"example"}`), not a string value.

```
xdmp:document-insert("my.json",
  xdmp:unquote('{"a": {"b": {"c": "value"}}}'));
cts:search(fn:doc(), cts:field-value-query("myField", "value"));
```

To learn more about fields, see Overview of Fields in the *Administrator's Guide*.

20.6 Representing Geospatial, Temporal, and Semantic Data

To take advantage of MarkLogic Server support for geospatial, temporal, and semantic data in JSON documents, you must represent the data in specific ways.

- Geospatial Data
- Date and Time Data
- Semantic Data

20.6.1 Geospatial Data

Geospatial data represents a set of latitude and longitude coordinates defining a point or region. You can define indexes and perform queries on geospatial values. Your geospatial data must use one of the coordinate systems recognized by MarkLogic.

A point can be represented in the following ways in JSON:

- The coordinates in a GeoJSON object; see http://geojson.org. For example: `{"geometry": {"type": "Point", "coordinates": [37.52, 122.25]}}`
- A JSON property whose value is an array of numbers, where the first 2 members represent the latitude and longitude (or vice versa) and all other members are ignored. For example, the value of the `coordinates` property of the following object:`{"location": {"desc": "somewhere", "coordinates": [37.52, 122.25]}}`
- A pair of JSON properties, one whose value represents latitude, and the other whose value represents the longitude. For example: `{"lat": 37.52, "lon": 122.25}`
- A string containing two numbers separated by a space. For example, `"37.52 122.25"`.

You can create indexes on geospatial data in JSON documents, and you can search geospatial data using queries such as `cts:json-property-geospatial-query`, `cts:json-property-child-geospatial-query`, `cts:json-property-pair-geospatial-query`, and `cts:path-geospatial-query` (or their JavaScript equivalents). The Node.js, Java, and REST Client APIs support similar queries.

Only 2D points are supported.

Note that GeoJSON regions all have the same structure (a `type` and a `coordinates` property). Only the type property differentiates between kinds of regions, such as points vs. polygons. Therefore, when defining indexes for GeoJSON data, you should usually use a geospatial path range index that includes a predicate on `type` in the path expression.

For example, to define an index that covers only GeoJSON points (`"type": "Point"`), you can use a path expressions similar to the following when defining the index. Then, search using `cts:path-geospatial-query` or the equivalent structured query (see geo-path-query in the *Search Developer's Guide*).

```
/whatever/geometry[type="Point"]/array-node("coordinates")
```

20.6.2 Date and Time Data

MarkLogic Server uses date, time, and dateTime data types in features such as Temporal Data Management, Tiered Storage, and range indexes.

A JSON string value in a recognized date-time format can be used in the same contexts as the equivalent text in XML. MarkLogic Server recognizes the date and time formats defined by the XML Schema, based on ISO-8601 conventions. For details, see the following document:

http://www.w3.org/TR/xmlschema-2/#isoformats

To create range indexes on a temporal data type, the data must be stored in your JSON documents as string values in the ISO-8601 standard XSD date format. For example, if your JSON documents contain data of the following form:

```
{ "theDate" : "2014-04-21T13:00:01Z" }
```

Then you can define an element range index on `theDate` with `dateTime` as the "element" type, and perform queries on the `theDate` that take advantage of temporal data characteristics, rather than just treating the data as a string.

20.6.3 Semantic Data

You can load semantic triples into the database in any of the formats described in Supported RDF Triple Formats in the *Semantics Developer's Guide*, including RDF/JSON.

An embedded triple in a JSON document is indexed if it is in the following format:

```
{ "triple": {
    "subject": IRI_STRING,
    "predicate": IRI_STRING,
    "object": STRING_PRESENTATION_OF_RDF_VALUE
} }
```

For example:

```
{
  "my" : "data",
  "triple" : {
    "subject": "http://example.org/ns/dir/js",
    "predicate": "http://xmlns.com/foaf/0.1/firstname",
    "object": {"value": "John", "datatype": "xs:string"}
  }
}
```

For more details, see Loading Semantic Triples in the *Semantics Developer's Guide*.

20.7 Character Set Restrictions

You cannot use any characters in a JSON document that are forbidden characters in XML 1.1. The following characters are forbidden:

- 0x0
- 0xD800 - 0xDFFF
- 0xFFFE, 0xFFFF, and characters above 0x10FFF

20.8 Document Properties

A JSON document can have a document property fragment, but the document properties must be in XML.

20.9 Serialization of Large Integer Values

MarkLogic can represent integer values larger than JSON supports. For example, the xs:unsignedLong XSD type includes values that cannot be expressed as an integer in JSON.

When MarkLogic serializes an xs:unsignedLong value that is too large for JSON to represent, the value is serialized as a string. Otherwise, the value is serialized as a number. This means that the same operation can result in either a string value or a number, depending on the input.

For example, the following code produces a JSON object with one property value that is a number and one property value that is a string:

```
xquery version "1.0-ml";
object-node {
  "notTooBig": 1111111111111,
  "tooBig":11111111111111111
}
```

The object node created by this code looks like the following, where "notTooBig" is a number node and "tooBig" is a text node.

```
{"notTooBig":1111111111111, "tooBig":"11111111111111111"}
```

Code that works with serialized JSON data that may contain large numbers must account for this possibility.

20.10 Working With JSON in XQuery

This section provides tips and examples for working with JSON documents using XQuery. The following topics are covered:

- Constructing JSON Nodes
- Building a JSON Object from a Map
- Interaction With fn:data
- JSON Document Operations
- Example: Updating JSON Documents
- Searching JSON Documents

Interfaces are also available to work with JSON documents using Java, JavaScript, and REST. See the following guides for details:

- *JavaScript Reference Guide*
- *Node.js Application Developer's Guide*
- *Java Application Developer's Guide*
- *REST Application Developer's Guide*

20.10.1 Constructing JSON Nodes

The following element constructors are available for building JSON objects and lists:

- `object-node`
- `array-node`
- `number-node`
- `boolean-node`
- `null-node`
- `text`

Each constructor creates a JSON node. Constructors can be nested inside one another to build arbitrarily complex structures. JSON property names and values can be literals or XQuery expressions.

The table below provides several examples of JSON constructor expressions, along with the corresponding serialized JSON.

JSON	Constructor Expression(s)
`{ "key": "value" }`	`object-node { "key" : "value" }`
`{ "key" : 42 }`	`object-node {"key" : 42 }` `object-node { "key" : number-node { 42 } }`
`{ "key" : true }`	`object-node { "key" : fn:true() }` `object-node { "key" : boolean-node { "true" } }`
`{ "key" : null }`	`object-node { "key" : null-node { } }`

`{ "key" : {`
`"child1" : "one",`
`"child2" : "two"`
`} }`
	`object-node {`
	`"key" : object-node {`
	`"child1" : "one",`
	`"child2" : "two"`
	`}`
	`}`

`{ "key" : [1, 2, 3] }`	`object-node { "key" : array-node { 1, 2, 3 } }`

`{ "date" : "06/24/14"`	`object-node {`
	`"date" :`
	`fn:format-date(`
	`fn:current-date(),"[M01]/[D01]/[Y01]")`
	`}`

You can also create JSON nodes from string using `xdmp:unquote`. For example, the following creates a JSON document that contains `{"a": "b"}`.

```
xdmp:document-insert("my.json", xdmp:unquote('{"a": "b"}'))
```

You can also create a JSON document node using `xdmp:to-json`, which accepts as input all the nodes types you can create with a constructor, as well as a `map:map` representation of name-value pairs. For details, see “Building a JSON Object from a Map” on page 371 and “Low-Level JSON XQuery APIs and Primitive Types” on page 387.

```
xquery version "1.0-ml";
let $object := json:object()
let $array := json:to-array((1, 2, "three"))
let $dummy := (
  map:put($object, "name", "value"),
  map:put($object, "an-array", $array))
return xdmp:to-json($object)
==> {"name":"value", "an-array": [1,2,"three"]}
```

20.10.2 Building a JSON Object from a Map

You can create a JSON document with a JSON object root node by building up a `map:map` and then applying `xdmp:to-json` to the map. You might find this approach easier than using the JSON node constructors in some contexts.

For example, the following code creates a document node that contains a JSON object with one property with atomic type ("a"), one property with array type ("b"), and one property with object-node type:

```
xquery version "1.0-ml";
let $map := map:map()
let $_ := $map => map:with('a', 1)
               => map:with('b', (2,3,4))
               => map:with('c', map:map() => map:with('c1', 'one')
                                          => map:with('c2', 'two'))
return xdmp:to-json($map)
```

This code produces the following JSON document node:

```
{ "a":1,
  "b":[2, 3, 4],
  "c":{"c1":"one", "c2":"two"}
}
```

A `json:object` is a special type of `map:map` that represents a JSON object. You can combine map operations and `json:*` functions. The following example uses both `json:*` functions such as `json:object` and `json:to-array` and `map:map` operations like `map:with`.

```
xquery version "1.0-ml";
let $object := json:object()
let $array := json:to-array((1, 2, "three"))
let $_ := (
  map:put($object, "name", "value"),
  map:put($object, "an-array", $array))
return xdmp:to-json($object)
```

This code produces the following JSON document node:

```
{"name":"value", "an-array": [1,2,"three"]}
```

To use JSON node constructors instead, see "Constructing JSON Nodes" on page 369.

20.10.3 Interaction With fn:data

Calling `fn:data` on a JSON node representing an atomic type such as a `number-node`, `boolean-node`, `text-node`, or `null-node` returns the value. Calling `fn:data` on an `object-node` or `array-node` returns the XML representation of that node type, such as a `<json:object/>` or `<json:array/>` element, respectively.

Example Call	Result

`fn:data(`
`  object-node {"a": "b"}`
`)`
	`<json:object ...`
	`  xmlns:json="http://marklogic.com/xdmp/json">`
	`  <json:entry key="a">`
	`  <json:value>b</json:value>`
	`  </json:entry>`
	`</json:object>`

`fn:data(`
`  array-node {(1,2)}`
`)`
	`<json:array ...`
	`  xmlns:json="http://marklogic.com/xdmp/json">`
	`  <json:value xsi:type="xs:integer">1</json:value>`
	`  <json:value xsi:type="xs:integer">2</json:value>`
	`</json:array>`

`fn:data(`
`  number-node { 1 }`
`)`
	`1`

`fn:data(`
`  boolean-node { true }`
`)`
	`true`

`fn:data(`
`  null-node { }`
`)`
	`()`

You can probe this behavior using a query similar to the following in Query Console:

```
xquery version "1.0-ml";
xdmp:describe(
  fn:data(
    array-node {(1,2)}
))
```

In the above example, the `fn:data` call is wrapped in `xdmp:describe` to more accurately represent the in-memory type. If you omit the `xdmp:describe` wrapper, serialization of the value for display purposes can obscure the type. For example, the array example returns [1,2] if you remove the `xdmp:describe` wrapper, rather than a `<json:array/>` node.

20.10.4 JSON Document Operations

Create, read, update and delete JSON documents using the same functions you use for other document types, including the following builtin functions:

- `xdmp:document-insert`
- `xdmp:document-load`
- `xdmp:document-delete`

- `xdmp:node-replace`
- `xdmp:node-insert-child`
- `xdmp:node-insert-before`

Use the node constructors to build JSON nodes programmatically; for details, see "Constructing JSON Nodes" on page 369.

> **Note:** A node to be inserted into an object node must have a name. A node to be inserted in an array node can be unnamed.

Use `xdmp:unquote` to convert serialized JSON into a node for insertion into the database. For example:

```
xquery version "1.0-ml";
let $node := xdmp:unquote('{"name" : "value"}')
return xdmp:document-insert("/example/my.json", $node)
```

Similar document operations are available through the Java, JavaScript, and REST APIs. You can also use the mlcp command line tool for loading JSON documents into the database.

20.10.5 Example: Updating JSON Documents

The table below provides examples of updating JSON documents using `xdmp:node-replace`, `xdmp:node-insert`, `xdmp:node-insert-before`, and `xdmp:node-insert-after`. Similar capabilities are available through other language interfaces, such as JavaScript, Java, and REST.

The table below contains several examples of updating a JSON document.

Update Operation	Results	

Replace a string value in a name-value pair.
 	
`xdmp:node-replace(`
`  fn:doc("my.json")/a/b,`
`  text { "NEW" }`
`)`
	Before	`{"a":{"b":"OLD"}}`

	After	`{"a":{"b":"NEW"}}`

Replace a string value in an array.
 	
`xdmp:node-replace(`
`  fn:doc("my.json")/a[2],`
`  text { "NEW" }`
`)`
	Before	`{"a": ["v1","OLD","v3"] }`

	After	`{"a": ["v1", "NEW", "v3"] }`

Update Operation	Results	

Insert an object.
 	
`xdmp:node-insert-child(`
`  fn:doc("my.json")/a,`
`  object-node {"c": "NEW" }/c`
`)`
	Before	`{"a": {"b":"val"} }`

	After	`{ "a": {"b":"val","c":"NEW"} }`

Insert an array member.
 	
`xdmp:node-insert-child(`
`  fn:doc("my.json")/array-node("a"),`
`  text { "NEW" }`
`)`
	Before	`{ "a": ["v1", "v2"] }`

	After	`{ "a": ["v1", "v2", "NEW"] }`

Insert an object before another node.
 	
`xdmp:node-insert-before(`
`  fn:doc("my.json")/a/b,`
`  object-node { "c": "NEW" }/c`
`)`
	Before	`{ "a": {"b":"val"} }`

	After	`{ "a": {"c":"NEW","b":"val"} }`

Insert an array member before another member.
 	
`xdmp:node-insert-before(`
`  fn:doc("my.json")/a[2],`
`  text { "NEW" }`
`)`
	Before	`{ "a": ["v1", "v2"] }`

	After	`{ "a": ["v1", "NEW", "v2"] }`

Insert an object after another node.
 	
`xdmp:node-insert-after(`
`  fn:doc("my.json")/a/b,`
`  object-node { "c": "NEW" }/c`
`)`
	Before	`{ "a": {"b":"val"} }`

	After	`{ "a": {"b":"val","c":"NEW"} }`

Insert an array member after another member.
 	
`xdmp:node-insert-after(`
`  fn:doc("my.json")/a[2],`
`  text { "NEW" })`
	Before	`{ "a": ["v1", "v2"] }`

	After	`{ "a": ["v1", "v2", "NEW"] }`

Notice that when inserting one object into another, you must pass the named object node to the node operation. That is, if inserting a node of the form `object-node {"c": "NEW"}` you cannot pass that expression directly into an operation like `xdmp:node-insert-child`. Rather, you must pass in the associated named node, `object-node {"c": "NEW"}/c`.

For example, assuming `fn:doc("my.json")/a/b` targets a object node, then the following generates an `XDMP-CHILDUNNAMED` error:

```
xdmp:node-insert-after(
  fn:doc("my.json")/a/b,
  object-node { "c": "NEW" }
)
```

20.10.6 Searching JSON Documents

Searches generally behave the same way for both JSON and XML content, except for any exceptions noted here. This section covers the following search related topics:

- Available cts Query Functions
- cts Query Serialization

You can also search JSON documents with string query, structured query, and QBE through the client APIs. For details, see the following references:

- *Search Developer's Guide*
- *Node.js Application Developer's Guide*
- *Java Application Developer's Guide*
- *MarkLogic REST API Reference*

20.10.6.1 Available cts Query Functions

A name-value pair in a JSON document is called a property. You can perform CTS queries on JSON properties using the following query constructors and `cts:search`:

- `cts:json-property-word-query`
- `cts:json-property-value-query`
- `cts:json-property-range-query`
- `cts:json-property-scope-query`
- `cts:json-property-geospatial-query`
- `cts:json-property-child-geospatial-query`
- `cts:json-property-pair-geospatial-query`

You can also use the following lexicon functions:

- `cts:json-property-words`
- `cts:json-property-word-match`
- `cts:values`
- `cts:value-match`

Constructors for JSON index references are also available, such as `cts:json-property-reference`.

The Search API and MarkLogic client APIs (REST, Java, Node.js) also support queries on JSON documents using string and structured queries and QBE. For details, see the following:

- Querying Documents and Metadata in the *Node.js Application Developer's Guide*
- Searching in the *Java Application Developer's Guide*
- *Search Developer's Guide*
- Using and Configuring Query Features in the *REST Application Developer's Guide*

When creating indexes and lexicons on JSON documents, use the interfaces for creating indexes and lexicons on XML elements. For details, see "Creating Indexes and Lexicons Over JSON Documents" on page 364.

20.10.6.2cts Query Serialization

A CTS query can be serialized as either XML or JSON. The proper form is chosen based on the parent node and the calling language.

If the parent node is an XML element node, the query is serialized as XML. If the parent node is a JSON object or array node, the query is serialized as JSON. Otherwise, a query is serialized based on the calling language. That is, as JSON when called from JavaScript and as XML otherwise.

If the value of a JSON query property is an array and the array is empty, the property is omitted from the serialized query. If the value of a property is an array containing only one item, it is still serialized as an array.

20.11 Working With JSON in Server-Side JavaScript

When you access a JSON document in the database from Server-Side JavaScript, you get an immutable document object. You should usually manipulate JSON documents in Server-Side JavaScript as JavaScript objects or arrays.

MarkLogic provides a `toObject` method on JSON document nodes for easy conversion from a JSON node to its natural JavaScript representation. However, you still need to be aware of the document model described in "How MarkLogic Represents JSON Documents" on page 356.

See the following topics for more detail:

- Constructing JSON Nodes in JavaScript
- Updating JSON Documents from JavaScript
- Read-Only Access to JSON Document Contents
- Using Node Update Functions on JSON Documents

20.11.1 Constructing JSON Nodes in JavaScript

Use the `NodeBuilder` interface when you need to programmatically construct a JSON node. You must use the `NodeBuilder` interface to construct text, number, boolean, and null nodes. For example:

```
// construct a number node
const nb = new NodeBuilder();
nb.addNumber(42).toNode();

// construct a text node
const nb = new NodeBuilder();
nb.addText('someString').toNode();
```

Using a `NodeBuilder` is optional when passing a JSON object node or array node into a function that expects a node because MarkLogic implicitly converts native JavaScript object and array parameter values into JSON object nodes and array nodes. For example:

```
// Create a JSON document from a native JavaScript object
declareUpdate();
const nb = new NodeBuilder();
xdmp.documentInsert('some.json', {a: 1, b: 2});

// Create a JSON document from a native JavaScript array
declareUpdate();
const nb = new NodeBuilder();
xdmp.documentInsert('some.json', [1,2,3]);

// Create a JSON document from a constructed object node
declareUpdate();
const nb = new NodeBuilder();
xdmp.documentInsert('some.json', nb.addNode({a: 10, b: 20}).toNode());
```

For more details on programmatically constructing nodes, see NodeBuilder API in the *JavaScript Reference Guide*.

20.11.2 Updating JSON Documents from JavaScript

To make changes to a JSON document whose root node is a JSON object node or array node, you should convert the immutable document node into its mutable JavaScript representation using the following technique.

1. Use the `toObject` method of the document node to convert it into an in-memory JavaScript representation.

2. Apply your changes to the JavaScript object or array.

3. Update the JSON document using the JavaScript object or array.

The following example applies the toObject technique to a document with an object node root. The example inserts, updates, and deletes JSON properties on a mutable object, and then updates the original document using `xdmp.nodeReplace`.

```
declareUpdate();

// assume my.json contains {a: 1, b: 2, c: 3}

const doc = cts.doc('my.json');
let obj = doc.toObject();           // create mutable representation

obj.d = 4;                          // insert a new property
obj.a = 10;                         // update a property
delete obj.b;                       // delete a property

xdmp.nodeReplace(doc, obj);

// resulting document contains {a: 10, c: 3, d: 4}
```

The example uses `xdmp.nodeReplace` rather than `xdmp.documentInsert` to update the original document because `xdmp.nodeReplace` preserves document metadata such as collections and permissions. However, you can use whatever update/insert function meets the needs of your application.

You can use this technique even when the root node of the document is not an object node. The following example applies the same `toObject` technique to update a document with an array node as its root.

```
declareUpdate();

// assume myArr.json contains [1,2,3]

const doc = cts.doc('myArr.json');
let arr = doc.toObject();
arr[1] = 20;
xdmp.nodeReplace(doc, arr);

// Result: [1, 20, 3]
```

If you attempt to modify a JSON document node without converting it to its mutable JavaScript representation using `toObject`, you will get an error. For example, the following code would produce an error because it attempts to change the value of a property named “a” on the immutable document node:

```
declareUpdate();

const doc = cts.doc('my.json');
doc.a = 10;                        // error because doc is immutable
```

20.11.3 Read-Only Access to JSON Document Contents

You should usually use the technique described in "Updating JSON Documents from JavaScript" on page 377 to work with JSON document contents from Server-Side JavaScript. That is, use the `toObject` method to first convert the document node into its logical native JavaScript representation so that you can manipulate it in a natural way. For example:

```
// assume my.json contains an object node of the form {"child": 1}
const doc = cts.doc('my.json');
const obj = doc.toObject();           // convert to a JavaScript object

console.log('The value of child is: ' + obj.child);
```

This technique applies even if the root node of the document is not an object node. For example, the following code retrieves the first item from a JSON document whose root node is an array node:

```
// assume arr.json contains an array node of the form [1,2,3]
const doc = cts.doc('arr.json');
'The first array item value is: ' + doc.toObject()[0];
```

The following example uses a JSON document whose root node is a number node:

```
// assume num.json contains a number with the value 42
const doc = cts.doc('num.json');
'The answer is: ' + (doc.toObject() + 5)
```

If you cannot read the entire document into memory for some reason, you can also access its contents through the document node `root` property. For example:

```
const docNode = cts.doc('my.json');
console.log('The value of child is: ' + docNode.root.child);
```

However, using `toObject` is the recommended approach.

For more details, see Document Object in the *JavaScript Reference Guide*.

20.11.4 Using Node Update Functions on JSON Documents

In most cases, you should use the technique described "Updating JSON Documents from JavaScript" on page 377 to modify JSON documents from Server-Side JavaScript. If you cannot use that technique for some reason, MarkLogic provides the following functions for updating individual nodes within a JSON or XML document.

- `xdmp.nodeReplace`
- `xdmp.nodeInsertChild`
- `xdmp.nodeInsertBefore`
- `xdmp.nodeInsertAfter`

- `xdmp.nodeDelete`

You can only use the insert and replace functions in contexts in which you can construct a suitable node to insert or replace. For example, inserting or updating array items, or updating the value of an existing JSON property.

You cannot construct a node that represents just a JSON property, so you cannot use `xdmp.nodeInsertAfter`, `xdmp.nodeInsertChild`, or `xdmp.nodeInsertBefore` to insert a new JSON property into an object node. Instead, use the technique described in "Updating JSON Documents from JavaScript" on page 377.

To replace the value of an array node, you must address the array node, not one of the array items. For example, use a path expression with an `array-node` or `node` expression in its leaf step. For more details, see "Selecting Arrays and Array Members" on page 362.

Keep the following points in mind when passing new or replacement nodes into the update functions. For more details, see "Constructing JSON Nodes in JavaScript" on page 377.

- You are not required to programmatically construct object and array nodes because MarkLogic implicitly converts a native JavaScript object or array into its corresponding JSON node during parameter passing.
- Any other node type must be constructed. For example, use a `NodeBuilder` to create a number, boolean, text, or null node.

The following examples illustrate using the node update functions on JSON documents. For more information on using XPath on JSON documents, see "Traversing JSON Documents Using XPath" on page 357.

```
// Replace a non-array node with an object node
xdmp.nodeReplace(someDoc.xpath('/target'), {my: 'NewValue'});

// Replace a non-array node with an array node
xdmp.nodeReplace(someDoc.xpath('/target'), [10,20,30]);

// Replace a non-array node with a constructed node (here, a text node)
xdmp.nodeReplace(someDoc.xpath('/target'),
  new NodeBuilder().addText('newValue').toNode());

// Replace an array node with another array node
xdmp.nodeReplace(someDoc.xpath('/array-node("target")'), [10,20,30]);

// Replace the first item in an array with a number
xdmp.nodeReplace(someDoc.xpath('/target[1]'),
  new NodeBuilder().addNumber(42).toNode());

// Insert a new item after the first item in an array
xdmp.nodeInsertAfter(someDoc.xpath('/target[1]'),
  new NodeBuilder().addNumber(11).toNode());

// Insert a new item before the first item in an array
```

```
xdmp.nodeInsertAfter(someDoc.xpath('/target[1]'),
  new NodeBuilder().addNumber(10).toNode());

// Insert a new item at the end of an array
xdmp.nodeInsertAfter(someDoc.xpath('/array-node("target")'),
  new NodeBuilder().addNumber(20).toNode());

// Delete a non-array node
xdmp.nodeDelete(someDoc.xpath('/target'));

// Delete an array node
xdmp.nodeDelete(someDoc.xpath('/array-node("target")'));
```

20.12 Converting JSON to XML and XML to JSON

You can use MarkLogic APIs to seamlessly and efficiently convert a JSON document to XML and vice-versa without losing any semantic meaning. This section describes how to perform these conversions and includes the following parts:

The JSON XQuery library module converts documents to and from JSON and XML. To ensure fast transformations, it uses the underlying low-level APIs described in "Low-Level JSON XQuery APIs and Primitive Types" on page 387. This section describes how to use the XQuery library and includes the following parts:

- Conversion Philosophy
- Functions for Converting Between XML and JSON
- Understanding the Configuration Strategies For Custom Transformations
- Example: Conversion Using Basic Strategy
- Example: Conversion Using Full Strategy
- Example: Conversion Using Custom Strategy

20.12.1 Conversion Philosophy

To understand how the JSON conversion features in MarkLogic work, it is useful to understand the following goals that MarkLogic considered when designing the conversion:

- Make it easy and fast to perform simple conversions using default conversion parameters.
- Make it possible to do custom conversions, allowing custom JSON and/or custom XML as either output or input.
- Enable both fast key/value lookup and fine-grained search on JSON documents.
- Make it possible to perform semantically lossless conversions.

Because of these goals, the defaults are set up to make conversion both fast and easy. Custom conversion is possible, but will take a little more effort.

20.12.2 Functions for Converting Between XML and JSON

The main function to convert from JSON to XML is:

- XQuery: `json:transform-from-json`
- Server-Side JavaScript: `json.transformFromJson`

The main function to convert from XML to JSON is:

- XQuery: `json:transform-to-json`
- Server-Side JavaScript: `json.transformToJson`

For examples, see the following sections:

- Example: Conversion Using Basic Strategy
- Example: Conversion Using Full Strategy
- Example: Conversion Using Custom Strategy

20.12.3 Understanding the Configuration Strategies For Custom Transformations

There are three strategies available for JSON conversion:

- `basic`
- `full`
- `custom`

A *strategy* is a piece of configuration that tells the JSON conversion library how you want the conversion to behave. The `basic` conversion strategy is designed for conversions that start in JSON, and then get converted back and forth between JSON, XML, and back to JSON again. The `full` strategy is designed for conversion that starts in XML, and then converts to JSON and back to XML again. The `custom` strategy allows you to customize the JSON and/or XML output.

To use any strategy except the `basic` strategy, you can set and check the configuration options using the following functions:

- `json:config` (XQuery) or json.config (Server-Side JavaScript)
- `json:check-config`
- XQuery: `json:config` and `json:check-config`
- Server-Side JavaScript: `json.config` and `json.checkConfig`

For the `custom` strategy, you can tailer the conversion to your requirements. For details on the properties you can set to control the transformation, see `json:config` in the *MarkLogic XQuery and XSLT Function Reference*.

20.12.4 Example: Conversion Using Basic Strategy

The following uses the `basic` (which is the default) strategy for transforming a JSON string to XML and then back to JSON. You can also pass in a JSON object or array node.

```
xquery version '1.0-ml';
import module namespace json = "http://marklogic.com/xdmp/json"
    at "/MarkLogic/json/json.xqy";

declare variable $j :=  '{
    "blah":"first value",
    "second Key":["first item","second item",null,"third item",false],
    "thirdKey":3,
    "fourthKey":{"subKey":"sub value",
                 "boolKey" : true, "empty" : null }
    ,"fifthKey": null,
    "sixthKey" : []
}'  ;

let $x := json:transform-from-json( $j )
let $jx := json:transform-to-json( $x )
return ($x, $jx)

=>
<json type="object" xmlns="http://marklogic.com/xdmp/json/basic">
  <blah type="string">first value</blah>
  <second_20_Key type="array">
    <item type="string">first item</item>
    <item type="string">second item</item>
    <item type="null"/>
    <item type="string">third item</item>
    <item type="boolean">false</item>
  </second_20_Key>
  <thirdKey type="number">3</thirdKey>
  <fourthKey type="object">
    <subKey type="string">sub value</subKey>
    <boolKey type="boolean">true</boolKey>
    <empty type="null"/>
  </fourthKey>
  <fifthKey type="null"/>
  <sixthKey type="array"/>
</json>
{"blah":"first value",
 "second Key":["first item","second item",null,"third item",false],
 "thirdKey":3,
 "fourthKey":{"subKey":"sub value", "boolKey":true, "empty":null},
 "fifthKey":null, "sixthKey":[]}
```

20.12.5 Example: Conversion Using Full Strategy

The following uses the `full` strategy for transforming a XML element to a JSON string. The full strategy outputs a JSON string with properties named in a consistent way. To transform the XML into a JSON object node instead of a string, use `json:transform-to-json-object`.

Suppose the database contains the following XML document with the URI “booklist.xml”:

```
<BOOKLIST>
 <BOOKS>
  <ITEM CAT="MMP">
   <TITLE>Pride and Prejudice</TITLE>
   <AUTHOR>Jane Austen</AUTHOR>
   <PUBLISHER>Modern Library</PUBLISHER>
    <PUB-DATE>2002-12-31</PUB-DATE>
    <LANGUAGE>English</LANGUAGE>
    <PRICE>4.95</PRICE>
    <QUANTITY>187</QUANTITY>
    <ISBN>0679601686</ISBN>
    <PAGES>352</PAGES>
    <DIMENSIONS UNIT="in">8.3 5.7 1.1</DIMENSIONS>
    <WEIGHT UNIT="oz">6.1</WEIGHT>
  </ITEM>
 </BOOKS>
</BOOKLIST>
```

Then the following code converts the contents from XML to JSON and back again.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace json = "http://marklogic.com/xdmp/json"`
	`    at "/MarkLogic/json/json.xqy";`
	
	`let $c := json:config("full")`
	`            => map:with("whitespace", "ignore"),`
	`    $j := json:transform-to-json(fn:doc("booklist.xml", $c),`
	`    $xj := json:transform-from-json($j,$c)`
	`return ($j, $xj)`

JavaScript	`const json = require('/MarkLogic/json/json.xqy');`
	
	`let config = json.config('full');`
	`config.whitespace = 'ignore';`
	
	`const j = json.transformToJson(cts.doc('booklist.xml'), config);`
	`const xj = json.transformFromJson(j, config);`
	`[j, xj]`

The example produces the following output:

```
{"BOOKLIST": { "_children": [
  {"BOOKS": { "_children": [ {
    "ITEM": {
      "_attributes": { "CAT": "MMP" },
      "_children": [
        {"TITLE": { "_children": [ "Pride and Prejudice" ] } },
```

```
          {"AUTHOR": { "_children": [ "Jane Austen" ] } },
          {"PUBLISHER": { "_children": [ "Modern Library" ] } },
          {"PUB-DATE": { "_children": [ "2002-12-31" ] } },
          {"LANGUAGE": { "_children": [ "English" ] } },
          {"PRICE": { "_children": [ "4.95" ] } },
          {"QUANTITY": { "_children": [ "187" ] } },
          {"ISBN": { "_children": [ "0679601686" ] } },
          {"PAGES": { "_children": [ "352" ] } },
          {"DIMENSIONS": {
            "_attributes": { "UNIT": "in" },
            "_children": [ "8.3 5.7 1.1" ]
          }},
          {"WEIGHT": {
            "_attributes": { "UNIT": "oz" },
            "_children": [ "6.1" ]
          }
}]}}]}}]}}
<BOOKLIST>
  <BOOKS>
    <ITEM CAT="MMP">
      <TITLE>Pride and Prejudice</TITLE>
      <AUTHOR>Jane Austen</AUTHOR>
      <PUBLISHER>Modern Library</PUBLISHER>
      <PUB-DATE>2002-12-31</PUB-DATE>
      <LANGUAGE>English</LANGUAGE>
      <PRICE>4.95</PRICE>
      <QUANTITY>187</QUANTITY>
      <ISBN>0679601686</ISBN>
      <PAGES>352</PAGES>
      <DIMENSIONS UNIT="in">8.3 5.7 1.1</DIMENSIONS>
      <WEIGHT UNIT="oz">6.1</WEIGHT>
    </ITEM>
  </BOOKS>
</BOOKLIST>
```

20.12.6 Example: Conversion Using Custom Strategy

The following uses the custom strategy to carefully control both directions of the conversion. The example converts a Search API XML options node into JSON and back again. The REST Client API uses a similar approach to transform options nodes back and forth between XML and JSON.

The following code is an XQuery example. The equivalent Server-Side JavaScript example follows.

```
xquery version "1.0-ml";
import module namespace json = "http://marklogic.com/xdmp/json"
    at "/MarkLogic/json/json.xqy";

declare namespace search="http://marklogic.com/appservices/search"  ;
declare variable $doc :=
<search:options xmlns:search="http://marklogic.com/appservices/search">
  <search:constraint name="decade">
    <search:range facet="true" type="xs:gYear">
```

```
      <search:bucket ge="1970" lt="1980" name="1970s">1970s</search:bucket>
      <search:bucket ge="1980" lt="1990" name="1980s">1980s</search:bucket>
      <search:bucket ge="1990" lt="2000" name="1990s">1990s</search:bucket>
      <search:bucket ge="2000" name="2000s">2000s</search:bucket>
      <search:facet-option>limit=10</search:facet-option>
      <search:attribute ns="" name="year"/>
      <search:element ns="http://marklogic.com/wikipedia" name="nominee"/>
    </search:range>
  </search:constraint>
</search:options>
 ;

let $c := json:config("custom")
          => map:with("whitespace", "ignore")
          => map:with("array-element-names",
xs:QName("search:bucket"))
          => map:with("attribute-names",
                    ("facet","type","ge","lt","name","ns" ))
          => map:with("text-value", "label")
          => map:with("camel-case", fn:true())
          => map:with("element-namespace",
                      "http://marklogic.com/appservices/search")
let $j := json:transform-to-json($doc ,$c)
let $x := json:transform-from-json($j,$c)
return ($j, $x)
```

The following code is a Server-Side JavaScript example.

```
'use strict';
const json = require('/MarkLogic/json/json.xqy');

const doc = fn.head(xdmp.unquote(
`<search:options xmlns:search="http://marklogic.com/appservices/search">
  <search:constraint name="decade">
    <search:range facet="true" type="xs:gYear">
      <search:bucket ge="1970" lt="1980" name="1970s">1970s</search:bucket>
      <search:bucket ge="1980" lt="1990" name="1980s">1980s</search:bucket>
      <search:bucket ge="1990" lt="2000" name="1990s">1990s</search:bucket>
      <search:bucket ge="2000" name="2000s">2000s</search:bucket>
      <search:facet-option>limit=10</search:facet-option>
      <search:attribute ns="" name="year"/>
      <search:element ns="http://marklogic.com/wikipedia" name="nominee"/>
    </search:range>
  </search:constraint>
</search:options>`
)) ;

let config = json.config('custom');
config['whitespace'] = 'ignore';
config['array-element-names'] = Sequence.from([
  fn.QName('http://marklogic.com/appservices/search', 'search:bucket')
]);
config['attribute-names'] = Sequence.from([
```

```
  'facet', 'type', 'ge', 'lt', 'name', 'ns'
]);
config['text-value'] = 'label';
config['camel-case'] = true;
config['element-namespace'] = 'http://marklogic.com/appservices/search';

let j = json.transformToJson(doc ,config);
let x = json.transformFromJson(j,config);
[j, x]
```

The examples produce the following output:

```
{"options":
 {"constraint":
  {"name":"decade",
   "range":{"facet":true, "type":"xs:gYear",
     "bucket":[{"ge":"1970", "lt":"1980", "name":"1970s",
       "label":"1970s"},
    {"ge":"1980", "lt":"1990", "name":"1980s","label":"1980s"},
    {"ge":"1990", "lt":"2000", "name":"1990s", "label":"1990s"},
    {"ge":"2000", "name":"2000s", "label":"2000s"}],
    "facetOption":"limit=10",
    "attribute":{"ns":"", "name":"year"},
    "element":{"ns":"http:\/\/marklogic.com\/wikipedia",
       "name":"nominee"}
}}}}
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="decade">
    <range facet="true" type="xs:gYear">
      <bucket ge="1970" lt="1980" name="1970s">1970s</bucket>
      <bucket ge="1980" lt="1990" name="1980s">1980s</bucket>
      <bucket ge="1990" lt="2000" name="1990s">1990s</bucket>
      <bucket ge="2000" name="2000s">2000s</bucket>
      <facet-option>limit=10</facet-option>
      <attribute ns="" name="year"/>
      <element ns="http://marklogic.com/wikipedia" name="nominee"/>
    </range>
  </constraint>
</options>
```

20.13 Low-Level JSON XQuery APIs and Primitive Types

There are several JSON APIs that are built into MarkLogic Server, as well as several primitive XQuery/XML types to help convert back and forth between XML and JSON. The APIs do the heavy work of converting between an XQuery/XML data model and a JSON data model. The higher-level JSON library module functions use these lower-level APIs. If you use the JSON library module, you will likely not need to use the low-level APIs.

This section covers the following topics:

- Available Functions and Primitive Types

- Example: Serializing to a JSON Node
- Example: Parsing a JSON Node into a List of Items

20.13.1 Available Functions and Primitive Types

There are two APIs devoted to serialization of JSON properties: one to serialize XQuery to JSON, and one to read a JSON string and create an XQuery data model from that string:

- `xdmp:to-json`
- `xdmp:from-json`

These APIs make the data available to XQuery as a map, and serialize the XML data as a JSON string. Most XQuery types are serialized to JSON in a way that they can be round-tripped (serialized to JSON and parsed from JSON back into a series of items in the XQuery data model) without any loss, but some types will not round-trip without loss. For example, an `xs:dateTime` value will serialize to a JSON string, but that same string would have to be cast back into an `xs:dateTime` value in XQuery in order for it to be equivalent to its original. The high-level API can take care of most of those problems.

There are also a set of low-level APIs that are extensions to the XML data model, allowing lossless data translations for things such as arrays and sequences of sequences, neither of which exists in the XML data model. The following functions support these data model translations:

- `json:array`
- `json:array-pop`
- `json:array-push`
- `json:array-resize`
- `json:array-values`
- `json:object`
- `json:object-define`
- `json:set-item-at`
- `json:subarray`
- `json:to-array`

Additionally, there are primitive XQuery types that extend the XQuery/XML data model to specify a JSON object (`json:object`), a JSON array (`json:array`), and a type to make it easy to serialize an `xs:string` to a JSON string when passed to `xdmp:to-json` (`json:unquotedString`).

To further improve performance of the transformations to and from JSON, the following built-ins are used to translate strings to XML NCNames:

- `xdmp:decode-from-NCName`
- `xdmp:encode-for-NCName`

The low-level JSON APIs, supporting XQuery functions, and primitive types are the building blocks to make efficient and useful applications that consume and or produce JSON. While these APIs are used for JSON translation to and from XML, they are at a lower level and can be used for any kind of data translation. But most applications will not need the low-level APIs; instead use the XQuery library API (and the REST and Java Client APIs that are built on top of the it), described in "Converting JSON to XML and XML to JSON" on page 381.

For the signatures and description of each function, see the *MarkLogic XQuery and XSLT Function Reference*.

20.13.2 Example: Serializing to a JSON Node

The following code returns a JSON array node that includes a map, a string, and an integer.

```
let $map := map:map()
let $put := map:put($map, "some-prop", 45683)
let $string := "this is a string"
let $int := 123
return
xdmp:to-json(($map, $string, $int))

(:
returns:
[{"some-prop":45683}, "this is a string", 123]
:)
```

For details on maps, see "Using the map Functions to Create Name-Value Maps" on page 165.

20.13.3 Example: Parsing a JSON Node into a List of Items

Consider the following, which is the inverse of the previous example:

```
let $json :=
  xdmp:unquote('[{"some-prop":45683}, "this is a string", 123]')
return
xdmp:from-json($json)
```

This returns the following items:

```
json:array(
<json:array xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:json="http://marklogic.com/xdmp/json">
 <json:value>
  <json:object>
   <json:entry key="some-prop">
    <json:value xsi:type="xs:integer">45683
    </json:value>
   </json:entry>
  </json:object>
 </json:value>
```

```
 <json:value xsi:type="xs:string">this is a string
 </json:value>
 <json:value xsi:type="xs:integer">123</json:value>
</json:array>)
```

Note that what is shown above is the serialization of the `json:array` XML element. You can also use some or all of the items in the XML data model. For example, consider the following, which adds to the `json:object` based on the other values (and prints out the resulting JSON string):

```
xquery version "1.0-ml";
let $json :=
  xdmp:unquote('[{"some-prop":45683}, "this is a string", 123]')
let $items := xdmp:from-json($json)
let $put := map:put($items[1], xs:string($items[3]), $items[2])
return
($items[1], xdmp:to-json($items[1]))

(: returns the following json:array and JSON string:
json:object(
<json:object xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:json="http://marklogic.com/xdmp/json">
  <entry key="some-prop">
    <json:value xsi:type="xs:integer">45683</json:value>
  </entry>
  <entry key="123">
    <json:value xsi:type="xs:string">this is a string</json:value>
  </entry>
</json:object>)
{"some-prop":45683, "123":"this is a string"}

This query uses the map functions to modify the first json:object
in the json:array.
:)
```

In the above query, the first item (`$items[1]`) returned from the `xdmp:from-json` call is a `json:array`, and the you can use the `map` functions to modify the `json:array`, and the query then returns the modified `json:array`. You can treat a `json:array` like a map, as the main difference is that the `json:array` is ordered and the `map:map` is not. For details on maps, see "Using the map Functions to Create Name-Value Maps" on page 165.

20.14 Loading JSON Documents

This section provides examples of loading JSON documents using a variety of MarkLogic tools and interfaces. The following topics are covered:

- Loading JSON Document Using mlcp
- Loading JSON Documents Using the Java Client API
- Loading JSON Documents Using the Node.js Client API

- Loading JSON Using the REST Client API

20.14.1 Loading JSON Document Using mlcp

You can ingest JSON documents with mlcp just as you can XML, binary, and text documents. If the file extension is ".json", MarkLogic automatically recognizes the content as JSON.

For details, see Loading Content Using MarkLogic Content Pump in the *Loading Content Into MarkLogic Server Guide*.

20.14.2 Loading JSON Documents Using the Java Client API

The Java Client API enables you to interact with MarkLogic Server from a Java application. For details, see the Java Application Developer's Guide.

Use the class `com.marklogic.client.document.DocumentManager` to create a JSON document in a Java application. The input data can come from any source supported by the Java Client API handle interfaces, including a file, a string, or from Jackson. For details, see Document Creation in the *Java Application Developer's Guide*.

You can also use the Java Client API to create JSON documents that represent POJO domain objects. For details, see POJO Data Binding Interface in the *Java Application Developer's Guide*.

20.14.3 Loading JSON Documents Using the Node.js Client API

The Node.js Client API enables you to handle JSON data in your client-side code as JavaScript objects. You can create a JSON document in the database directly from such objects, using the DatabaseClient.documents interface.

For details, see Loading Documents into the Database in the *Node.js Application Developer's Guide*.

20.14.4 Loading JSON Using the REST Client API

You can load JSON documents into MarkLogic Server using REST Client API. The following example shows how to use the REST Client API to load a JSON document in MarkLogic.

Consider a JSON file names `test.json` with the following contents:

```
{
"key1":"value1",
"key2":{
        "a":"value2a",
        "b":"value2b"
       }
}
```

Run the following `curl` command to use the `documents` endpoint to create a JSON document:

```
curl --anyauth --user user:password -T ./test.json -D - \
  -H "Content-type: application/json" \
  http://my-server:5432/v1/documents?uri=/test/keys.json
```

The document is created and the endpoint returns the following:

```
HTTP/1.1 100 Continue

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest realm="public", qop="auth",
  nonce="b4475e81fe81b6c672a5
  d105f4d8662a", opaque="de72dcbdfb532a0e"
Server: MarkLogic
Content-Type: text/xml; charset=UTF-8
Content-Length: 211
Connection: close

HTTP/1.1 100 Continue

HTTP/1.1 201 Document Created
Location: /test/keys.json
Server: MarkLogic
Content-Length: 0
Connection: close
```

You can then retrieve the document from the REST Client API as follows:

```
$ curl --anyauth --user admin:password -X GET -D - \
  http://my-server:5432/v1/documents?uri=/test/keys.json
==>
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest realm="public", qop="auth",
  nonce="2aaee5a1d206cbb1b894
  e9f9140c11cc", opaque="1dfded750d326fd9"
Server: MarkLogic
Content-Type: text/xml; charset=UTF-8
Content-Length: 211
Connection: close

HTTP/1.1 200 Document Retrieved
vnd.marklogic.document-format: json
Content-type: application/json
Server: MarkLogic
Content-Length: 56
Connection: close

{"key1":"value1", "key2":{"a":"value2a", "b":"value2b"}}
```

For details about the REST Client API, see REST Application Developer's Guide.

21.0 Using Triggers to Spawn Actions

MarkLogic Server includes pre-commit and post-commit triggers. This chapter describes how triggers work in MarkLogic Server and includes the following sections:

- Overview of Triggers
- Triggers and the Content Processing Framework
- Pre-Commit Versus Post-Commit Triggers
- Trigger Events
- Trigger Scope
- Modules Invoked or Spawned by Triggers
- Creating and Managing Triggers With triggers.xqy
- Simple Trigger Example
- Avoiding Infinite Trigger Loops (Trigger Storms)

21.1 Overview of Triggers

Conceptually, a trigger listens for certain events (document create, delete, update, or the database coming online) to occur, and then invokes an XQuery module to run after the event occurs. The trigger definition determines whether the action module runs before or after committing the transaction which causes the trigger to fire.

Creating a robust trigger framework is complex, especially if your triggers need to maintain state or recover gracefully from service interruptions. Before creating your own custom triggers, consider using the Content Processing Framework. CPF provides a rich, reliable framework which abstracts most of the event management complexity from your application. For more information, see "Triggers and the Content Processing Framework" on page 396.

Note: Triggers run as the user performing the update transaction that caused the trigger. The programmer is free to call amped library functions in triggers if the use case requires certain roles to work correctly.

The only exception here is the database-online trigger, because in that case there is no triggering update transaction, and hence no user. For database-online trigger the user is specified by the trigger itself. Some customization of CPF installation scripts is required in order to insure that this event is run as an existing administrative user.

21.1.1 Trigger Components

A trigger definition is stored as an XML document in a database, and it contains information about the following:

- The event definition, which describes:
  - the conditions under which the trigger fires
  - the scope of the watched content
- The XQuery module to invoke or spawn when the event occurs.

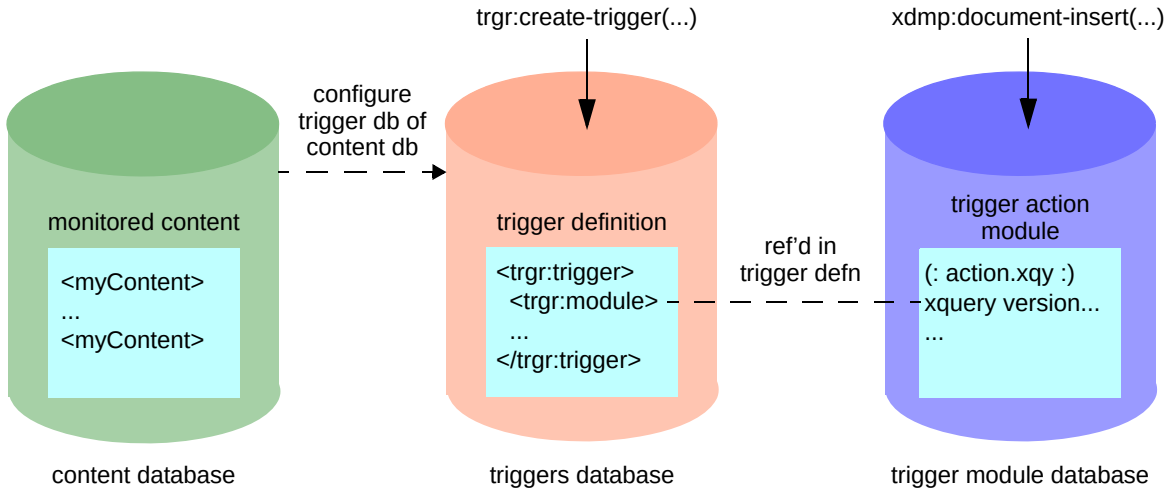
A trigger definition is created and installed by calling `trgr:create-trigger`. To learn more about trigger event definitions, see "Trigger Events" on page 398.

21.1.2 Databases Used By Triggers

A complete trigger requires monitored content, a trigger definition, and an action module. These components involve 3 databases:

- The content database monitored by the trigger.
- The triggers database, where the trigger definition is stored by `trgr:create-trigger`. This must be the triggers database configured for the content database.
- The module database, where the trigger action module is stored. This need not be the modules database configured for your App Server.

The following diagram shows the relationships among these databases and the trigger components:



Usually, the content, triggers and module databases are different physical databases, but there is no requirement that they be separate. A database named Triggers is installed by MarkLogic Server for your convenience, but any database may serve as the content, trigger, or module database. The choice is dependent on the needs of your application.

For example, if you want your triggers backed up with the content to which they apply, you might store trigger definitions and their action modules in your content database. If you want to share a trigger action module across triggers that apply to multiple content databases, you would use a separate trigger modules database.

Note: Most trigger API function calls must be evaluated in the context of the triggers database.

21.2 Triggers and the Content Processing Framework

The Content Processing Framework uses triggers to capture events and then set states in content processing pipelines. Since the framework creates and manages the triggers, you only need to configure the pipeline and supply the action modules.

In a pipeline used with the Content Processing Framework, a trigger fires after one stage is complete (from a document update, for example) and then the XQuery module specified in the trigger is executed. When it completes, the next trigger in the pipeline fires, and so on. In this way, you can create complex pipelines to process documents.

The Status Change Handling pipeline, installed when you install Content Processing in a database, creates and manages all of the triggers needed for your content processing applications, so it is not necessary to directly create or manage any triggers in your content applications.

When you use the Content Processing Framework instead of writing your own triggers:

- Actions may easily be chained together through pipelines.
- You only need to create and install your trigger action module.
- CPF handles recovery from interruptions for you.
- CPF automatically makes state available to your module and across stages of the pipeline.

Applications using the Content Processing Framework Status Change Handling pipeline do not need to explicitly create triggers, as the pipeline automatically creates and manages the triggers as part of the Content Processing installation for a database. For details, see the *Content Processing Framework Guide* manual.

21.3 Pre-Commit Versus Post-Commit Triggers

There are two ways to configure the transactional semantics of a trigger: pre-commit and post-commit. This section describes each type of trigger and includes the following parts:

- Pre-Commit Triggers
- Post-Commit Triggers

21.3.1 Pre-Commit Triggers

The module invoked as the result of a pre-commit trigger is evaluated as part of the same transaction that produced the triggering event. It is evaluated by invoking the module on the same App Server in which the triggering transaction is run. It differs from invoking the module with `xdmp:invoke` in one way, however; the module invoked by the pre-commit trigger sees the updates made to the triggering document.

Therefore, pre-commit triggers and the modules from which the triggers are invoked execute in a single context; if the trigger fails to complete for some reason (if it throws an exception, for example), then the entire transaction, including the triggering transaction, is rolled back to the point before the transaction began its evaluation.

This transactional integrity is useful when you are doing something that does not make sense to break up into multiple asynchronous steps. For example, if you have an application that has a trigger that fires when a document is created, and the document needs to have an initial property set on it so that some subsequent processing can know what state the document is in, then it makes sense that the creation of the document and the setting of the initial property occur as a single transaction. As a single transaction (using a pre-commit trigger), if something failed while adding the property, the document creation would fail and the application could deal with that failure. If it were not a single transaction, then it is possible to get in a situation where the document is created, but the initial property was never created, leaving the content processing application in a state where it does not know what to do with the new document.

21.3.2 Post-Commit Triggers

The task spawned as the result of a post-commit trigger is evaluated as a separate transaction. The task is compiled before the original transaction commits and is queued on the task server and run some time after the original transaction commits. Static errors that occur compiling a post-commit trigger task cause the original transaction to roll back. Dynamic errors that occur running a post-commit trigger task do not cause the original transaction to roll back. There is no guarantee that the post-commit trigger task will complete.

When a post-commit trigger spawns an XQuery module, it is put in the queue on the *task server*. The task server maintains this queue of tasks, and initiates each task in the order it was received. The task server has multiple threads to service the queue. There is one task server per group, and you can set task server parameters in the Admin Interface under Groups > *group_name* > Task Server.

Because post-commit triggers are asynchronous, the code that calls them must not rely on something in the trigger module to maintain data consistency. For example, the state transitions in the Content Processing Framework code uses post-commit triggers. The code that initiates the triggering event updates the property state before calling the trigger, allowing a consistent state in case the trigger code does not complete for some reason. Asynchronous processing has many advantages for state processing, as each state might take some time to complete. Asynchronous processing (using post-commit triggers) allows you to build applications that will not lose all of the processing that has already occurred should something happen in the middle of processing your pipeline. When the system is available again, the Content Processing Framework will simply continue the processing where it left off.

21.4 Trigger Events

The trigger event definition describes the conditions under which a trigger fires and the content to which it applies. There are two kinds of trigger events: data events and database events. Triggers can listen for the following events:

- document create
- document update
- document delete
- any property change (does *not* include MarkLogic Server-controlled properties such as `last-modified` and `directory`)
- specific (named) property change
- database coming online

21.4.1 Database Events

The only database event is a database coming online event. The module for a database online event runs as soon as the watched database comes online. A database online event definition requires only the name of the user under which the action module runs.

21.4.2 Data Events

Data events apply to changes to documents and properties. A trigger data event has the following parts:

- The trigger scope defines the set of documents to which the event applies. Use `trgr:*-scope` functions such as `trgr:directory-scope` to create this piece. For more information, see “Trigger Scope” on page 399.
- The content condition defines the triggering operation, such as document creation, update or deletion, or property modification. Use the `trgr:*-content` functions such as `trgr:document-content` to create this piece.

  To watch more than one operation, you must use multiple trigger events and define multiple triggers.

- The timing indicator defines when the trigger action occurs relative to the transaction that matches the event condition, either pre-commit or post-commit. Use `trgr:*-commit` functions such as `trgr:post-commit` to create this piece. For more information, see "Pre-Commit Versus Post-Commit Triggers" on page 397.

The content database to which an event applies is not an explicit part of the event or the trigger definition. Instead, the association is made through the triggers database configured for the content database.

Whether the module that the trigger invokes commits before or after the module that produced the triggering event depends upon whether the trigger is a pre-commit or post-commit trigger. Pre-commit triggers in MarkLogic Server listen for the event and then invoke the trigger module *before* the transaction commits, making the entire process a single transaction that either all completes or all fails (although the module invoked from a pre-commit trigger sees the updates from the triggering event).

Post-commit triggers in MarkLogic Server initiate after the event is committed, and the module that the trigger spawns is run in a separate transaction from the one that updated the document. For example, a trigger on a document update event occurs *after* the transaction that updates the document commits to the database.

Because the post-commit trigger module runs in a separate transaction from the one that caused the trigger to spawn the module (for example, the create or update event), the trigger module transaction cannot, in the event of a transaction failure, automatically roll back to the original state of the document (that is, the state *before* the update that caused the trigger to fire). If this will leave your document in an inconsistent state, then the application must have logic to handle this state.

For more information on pre- and post-commit triggers, see "Pre-Commit Versus Post-Commit Triggers" on page 397.

21.5 Trigger Scope

The *trigger scope* is the scope with which to listen for create, update, delete, or property change events. The scope represents a portion of the database corresponding to one of the trigger scope values: `document`, `directory`, or `collection`.

A `document` trigger scope specifies a given document URI, and the trigger responds to the specified trigger events only on that document.

A `collection` trigger scope specifies a given collection URI, and the trigger responds to the specified trigger events for any document in the specified collection.

A `directory` scope represents documents that are in a specified directory, either in the immediate directory (depth of `1`); or in the immediate or any recursive subdirectory of the specified directory. For example, if you have a directory scope of the URI `/` (a forward-slash character) with a depth of `infinity`, that means that any document in the database with a URI that begins with a

forward-slash character (`/`) will fire a trigger with this scope upon the specified trigger event. Note that in this directory example, a document called `hello.xml` is *not* included in this trigger scope (because it is not in the `/` directory), while documents with the URIs `/hello.xml` or `/mydir/hello.xml` are included.

21.6 Modules Invoked or Spawned by Triggers

Trigger definitions specify the URI of a module. This module is evaluated when the trigger is fired (when the event completes). The way this works is different for pre-commit and post-commit triggers. This section describes what happens when the trigger modules are invoked and spawned and includes the following subsections:

- Difference in Module Behavior for Pre- and Post-Commit Triggers
- Module External Variables trgr:uri and trgr:trigger

21.6.1 Difference in Module Behavior for Pre- and Post-Commit Triggers

For pre-commit triggers, the module is invoked when the trigger is fired (when the event completes). The invoked module is evaluated in an analogous way to calling `xdmp:invoke` in an XQuery statement, and the module evaluates synchronously in the same App Server as the calling XQuery module. The difference is that, with a pre-commit trigger, the invoked module sees the result of the triggering event. For example, if there is a pre-commit trigger defined to fire upon a document being updated, and the module counts the number of paragraphs in the document, it will count the number of paragraphs *after* the update that fired the trigger. Furthermore, if the trigger module fails for some reason (a syntax error, for example), then the entire transaction, including the update that fired the trigger, is rolled back to the state before the update.

For post-commit triggers, the module is spawned onto the task server when the trigger is fired (when the event completes). The spawned module is evaluated in an analogous way to calling `xdmp:spawn` in an XQuery statement, and the module evaluates asynchronously on the task server. Once the post-commit trigger module is spawned, it waits in the task server queue until it is evaluated. When the spawned module evaluates, it is run as its own transaction. Under normal circumstances the modules in the task server queue will initiate in the order in which they were added to the queue. Because the task server queue does not persist in the event of a system shutdown, however, the modules in the task server queue are not guaranteed to run.

21.6.2 Module External Variables trgr:uri and trgr:trigger

There are two external variables that are available to trigger modules:

- `trgr:uri as xs:string`
- `trgr:trigger as node()`

The `trgr:uri` external variable is the URI of the document which caused the trigger to fire (it is only available on triggers with data events, not on triggers with database online events). The `trgr:trigger` external variable is the trigger XML node, which is stored in the triggers database with the URI `http://marklogic.com/xdmp/triggers/`*`trigger_id`*, where *trigger_id* is the ID of the trigger. You can use these external variables in the trigger module by declaring them in the prolog as follows:

```
xquery version "1.0-ml";
import module namespace trgr='http://marklogic.com/xdmp/triggers'
   at '/MarkLogic/triggers.xqy';

declare variable $trgr:uri as xs:string external;
declare variable $trgr:trigger as node() external;
```

21.7 Creating and Managing Triggers With triggers.xqy

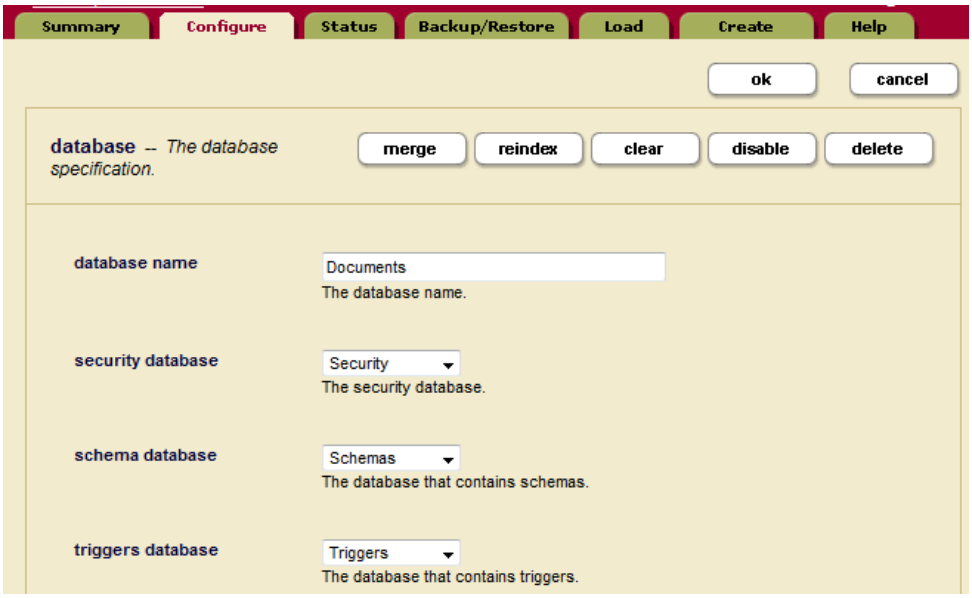
The `<install_dir>/Modules/MarkLogic/triggers.xqy` XQuery module file contains functions to create, delete, and manage triggers. If you are using the Status Change Handling pipeline, the pipeline takes care of all of the trigger details; you do not need to create or manage any triggers. For details on the trigger functions, see the *MarkLogic XQuery and XSLT Function Reference*.

For real-world examples of XQuery code that creates triggers, see the `<install_dir>/Modules/MarkLogic/cpf/domains.xqy` XQuery module file. For a sample trigger example, see “Simple Trigger Example” on page 402. The functions in this module are used to create the needed triggers when you use the Admin Interface to create a domain.

21.8 Simple Trigger Example

The following example shows a simple trigger that fires when a document is created.

1. Use the Admin Interface to set up the database to use a triggers database. You can specify any database as the triggers database. The following screenshot shows the database named `Documents` as the content database and `Triggers` as the triggers database.



2. Create a trigger that listens for documents that are created under the directory `/myDir/` with the following XQuery code. Note that this code must be evaluated against the triggers database for the database in which your content is stored.

```
xquery version "1.0-ml";
import module namespace trgr="http://marklogic.com/xdmp/triggers"
   at "/MarkLogic/triggers.xqy";

trgr:create-trigger("myTrigger", "Simple trigger example",
  trgr:trigger-data-event(
      trgr:directory-scope("/myDir/", "1"),
      trgr:document-content("create"),
      trgr:post-commit()),
  trgr:trigger-module(xdmp:database("Documents"),
                      "/modules/", "log.xqy"), fn:true(),
                      xdmp:default-permissions() )
```

   This code returns the ID of the trigger. The trigger document you just created is stored in the document with the URI `http://marklogic.com/xdmp/triggers/`*`trigger_id`*, where *trigger_id* is the ID of the trigger you just created.

3. Load a document whose contents is the XQuery module of the trigger action. This is the module that is spawned when the when the previously specified create trigger fires. For

this example, the URI of the module must be `/modules/log.xqy` in the database named `Documents` (from the `trgr:trigger-module` part of the `trgr:create-trigger` code above). Note that the document you load, because it is an XQuery document, must be loaded as a text document and it must have execute permissions. For example, create a trigger module in the `Documents` database by evaluating the following XQuery against the modules database for the App Server in which the triggering actions will be evaluated:

```
xquery version '1.0-ml';
(: evaluate this against the database specified
   in the trigger definition (Documents in this example)
:)
xdmp:document-insert("/modules/log.xqy",
  text{ "
xquery version '1.0-ml';
import module namespace trgr='http://marklogic.com/xdmp/triggers'
   at '/MarkLogic/triggers.xqy';

declare variable $trgr:uri as xs:string external;

xdmp:log(fn:concat('*****Document ', $trgr:uri, ' was created.*****'))"
}, xdmp:permission('app-user', 'execute'))
```

4. The trigger should now fire when you create documents in the database named `Documents` in the `/myDir/` directory. For example, the following:

```
xdmp:document-insert("/myDir/test.xml", <test/>)
```

will write a message to the `ErrorLog.txt` file similar to the following:

```
2007-03-12 20:14:44.972 Info: TaskServer: *****Document /myDir/test.xml
was created.*****
```

Note: This example only fires the trigger when the document is created. If you want it to fire a trigger when the document is updated, you will need a separate trigger with a `trgr:document-content` of `"modify"`.

21.9 Avoiding Infinite Trigger Loops (Trigger Storms)

If you create a trigger for a document to update itself, the result is an infinite loop, which is also known as a "trigger storm."

When a pre-commit trigger fires, its actions are part of the same transaction. Therefore, any updates performed in the trigger should not fire the same trigger again. To do so is to guarantee trigger storms, which generally result in an `XDMP-MAXTRIGGERDEPTH` error message.

In the following example, we create a trigger that calls a module when a document in the `/storm/` directory is modified the database. The triggered module attempts to update the document with a new child node. This triggers another update of the document, which triggers another update, and so on, ad infinitum. The end result is an `XDMP-MAXTRIGGERDEPTH` error message and no updates to the document.

To create a trigger storm, do the following:

1. In the `Modules` database, create a `storm.xqy` module to be called by the trigger:

```
xquery version "1.0-ml";

import module namespace trgr="http://marklogic.com/xdmp/triggers"
    at "/MarkLogic/triggers.xqy";

if (xdmp:database() eq xdmp:database("Modules"))
   then ()
   else error((), 'NOTMODULESDB', xdmp:database()) ,

xdmp:document-insert( '/triggers/storm.xqy', text {
   <code>
   xquery version "1.0-ml";
   import module namespace trgr='http://marklogic.com/xdmp/triggers'
       at '/MarkLogic/triggers.xqy';

    declare variable $trgr:uri as xs:string external;
    declare variable $trgr:trigger as node() external;

    xdmp:log(text {{
               'storm:',
               $trgr:uri,
               xdmp:describe($trgr:trigger)
               }}) ,

    let $root := doc($trgr:uri)/*
    return xdmp:node-insert-child(
               $root,
               element storm
               {{ count($root/*) }})
   </code>
} )
```

2. In the `Triggers` database, create the following trigger to call the `storm.xqy` module each time a document in the `/storm/` directory in the database is modified:

```
xquery version "1.0-ml";

import module namespace trgr="http://marklogic.com/xdmp/triggers"
    at "/MarkLogic/triggers.xqy";

if (xdmp:database() eq xdmp:database("Triggers"))
   then ()
   else error((), 'NOTTRIGGERSDB', xdmp:database()) ,

trgr:create-trigger(
     "storm",
     "storm",
     trgr:trigger-data-event(trgr:directory-scope("/storm/", "1"),
     trgr:document-content("modify"),
     trgr:pre-commit()),
     trgr:trigger-module(
         xdmp:database("Modules"),
         "/triggers/",
         "storm.xqy"),
     fn:true(),
     xdmp:default-permissions(),
     fn:true() )
```

3. Now insert a document twice into any database that uses `Triggers` as its triggers database:

```
xquery version "1.0-ml";

xdmp:document-insert('/storm/test', <test/> )
```

4. The second attempt to insert the document will fire the trigger, which should result in an `XDMP-MAXTRIGGERDEPTH` error message and repeated messages in `ErrorLog.txt` that look like the following:

```
2010-08-12 15:04:42.176 Info: Docs: storm: /storm/test
<trgr:trigger xmlns:trgr="http://marklogic.com/xdmp/triggers">
  <trgr:trigger-id>13904462711559236l4</trgr:trigger-id>
<trgr:trig...</trgr:trigger>
```

If you encounter similar circumstances in your application and it's not possible to modify your application logic, you can avoid trigger storms by setting the `$recursive` parameter in the `trgr:create-trigger` function to `fn:false()`. So your new trigger would look like:

```
trgr:create-trigger(
     "storm",
     "storm",
     trgr:trigger-data-event(trgr:directory-scope("/storm/", "1"),
     trgr:document-content("modify"),
     trgr:pre-commit()),
     trgr:trigger-module(
         xdmp:database("Modules"),
         "/triggers/",
         "storm.xqy"),
     fn:true(),
     xdmp:default-permissions(),
     fn:false() )
```

The result will be a single update to the document and no further recursion.

22.0 Using Native Plugins

A native plugin is a C++ dynamically loaded library that provides one or more plugin implementations to MarkLogic. This chapter covers how to create, install, and manage native plugins.

- What is a Native Plugin?
- How MarkLogic Server Manages Native Plugins
- Building a Native Plugin Library
- Packaging a Native Plugin
- Installing a Native Plugin
- Uninstalling a Native Plugin
- Registering a Native Plugin at Runtime
- Versioning a Native Plugin
- Checking the Status of Loaded Plugins
- The Plugin Manifest
- Native Plugin Security Considerations
- Native Plugin Example

22.1 What is a Native Plugin?

A native plugin is a dynamically linked library that contains one or more UDF (User Defined Function) implementations. When you package and deploy a native plugin in the expected way, MarkLogic distributes your code across the cluster and makes it available for execution through specific extension points.

The UDF interfaces define the extension points that can take advantage of a native plugin. MarkLogic currently supports the following UDFs:

- AggregateUDF: Use the map-reduce capabilities of MarkLogic to analyze values in lexicons and range indexes. For more details, see “Aggregate User-Defined Functions” on page 417.
- LexerUDF: Define a custom tokenizer for a language. For more details, see User-Defined Lexer Plugins in the *Search Developer’s Guide*.
- StemmerUDF: Define a custom stemmer for a language. For more details, see Using a User-Defined Stemmer Plugin in the *Search Developer’s Guide*.

The implementation requirement for each UDF varies, but they all use the native plugin mechanism for packaging, deployment, and version.

22.2 How MarkLogic Server Manages Native Plugins

Native plugins are deployed as dynamically loaded libraries that MarkLogic Server loads on-demand when referenced by an application. The User-Defined Functions (UDFs) implemented by a native plugin are identified by the relative path to the plugin and the name of the UDF. For a list of the supported kinds of UDFs, see "What is a Native Plugin?" on page 407.

When you install a native plugin library, MarkLogic Server stores it in the Extensions database. If the MarkLogic Server instance in which you install the plugin is part of a cluster, your plugin library is automatically propagated to all the nodes in the cluster.

There can be a short delay between installing a plugin and having the new version available. MarkLogic Server only checks for changes in plugin state about once per second. Once a change is detected, the plugin is copied to hosts with an older version.

In addition, each host has a local cache from which to load the native library, and the cache cannot be updated while a plugin is in use. Once the plugin cache starts refreshing, operations that try use a plugin are retried until the cache update completes.

MarkLogic Server loads plugins on-demand. A native plugin library is not dynamically loaded until the first time an application calls a UDF implemented by the plugin. A plugin can only be loaded or unloaded when no plugins are in use on a host.

22.3 Building a Native Plugin Library

Native plugins run in the same process context as the MarkLogic Server core, so you must compile and link your library in a manner compatible with the MarkLogic Server executable. Follow these basic steps to build your library:

- Compile your library with a C++ compiler and standard libraries compatible with MarkLogic Server. See the table below. This is necessary because C++ is not guaranteed binary compatible across compiler versions.
- Compile your C++ code with the options your platform requires for creating shared objects. For example, on Linux, compile with the `-fPIC` option.
- Build a 64-bit library (32-bit on Windows).

The sample plugin in *`marklogic_dir`*`/Samples/NativePlugins` includes a Makefile usable with GNU `make` on all supported platforms. You should use this makefile as the basis for building your own plugins as it includes all the required compiler options.

The makefile builds a shared library, generates a manifest, and zips up the library and manifest into an install package. The makefile is easily customized for your own plugin by changing a few `make` variables at the beginning of the file:

```
PLUGIN_NAME = sampleplugin
PLUGIN_VERSION = 0.1
PLUGIN_PROVIDER = MarkLogic
```

```
PLUGIN_DESCRIPTION = Example native plugin

PLUGIN_SRCS = \
  SamplePlugin.cpp
```

The table below shows the compiler and standard library versions used to build MarkLogic Server. You must build your native plugin with compatible tools.

Platform	Compiler
Linux	gcc 4.8.3
Windows	Microsoft Visual Studio 9 SP1
MacOS	gcc 4.2.1

22.4 Packaging a Native Plugin

You must package a native plugin into a zip file to install it. The installation zip file must contain:

- A C++ shared library implementing the plugin interface(s), such as `marklogic::AggregateUDF`, and the registration function `marklogicPlugin`.
- A plugin manifest file called `manifest.xml`. See "The Plugin Manifest" on page 414.
- Optionally, additional shared libraries required by the plugin implementation.

Including dependent libraries in your plugin zip file gives you explicit control over which library versions are used by your plugin and ensures the dependent libraries are available to all nodes in the cluster in which the plugin is installed.

The following example creates the plugin package `sampleplugin.zip` from the plugin implementation, `libsampleplugin.so`, a dependent library, `libdep.so`, and the plugin manifest.

```
$ zip sampleplugin.zip libsampleplugin.so libdep.so manifest.xml
```

If the plugin contents are organized into subdirectories, include the subdirectories in the paths in the manifest. For example, if the plugin components are organized as follows in the zip file:

```
$ unzip -l sampleplugin.zip
Archive:  sampleplugin.zip
  Length     Date   Time    Name
 --------    ----   ----    ----
    28261  06-28-12 12:54   libsampleplugin.so
      334  06-28-12 12:54   manifest.xml
        0  06-28-12 12:54   deps/
    28261  06-28-12 12:54   deps/libdep.so
 --------                   -------
    56856                   4 files
```

Then `manifest.xml` for this plugin must include `deps/` in the dependent library path:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin xmlns="http://marklogic.com/extension/plugin">
  <name>sampleplugin-name</name>
  <id>sampleplugin-id</id>
  ...
  <native>
    <path>libsampleplugin.so</path>
    <dependency>deps/libdep1.so</dependency>
  </native>
</plugin>
```

22.5 Installing a Native Plugin

After packaging your native plugin as described in "Packaging a Native Plugin" on page 409, install or update your plugin using the XQuery function `plugin:install-from-zip` or the Server-Side JavaScript function `plugin:installFromZip`.

For example, the following code installs a native plugin contained in the file `/space/plugins/sampleplugin.zip`. The relative plugin path in the Extensions directory is "native".

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace plugin =`
	`"http://marklogic.com/extension/plugin"`
	`  at "MarkLogic/plugin/plugin.xqy";`
	
	`plugin:install-from-zip("native",`
	`  xdmp:document-get("/space/plugins/sampleplugin.zip")/node())`

Server-Side JavaScript	`'use strict';`
	`declareUpdate();`
	`const plugin = require('/MarkLogic/plugin/plugin');`
	
	`plugin.installFromZip(`
	`  'native',`
	`  fn.head(`
	`  xdmp.documentGet('/space/plugins/sampleplugin.zip')).root);`

If the plugin was already installed on MarkLogic Server, the new version replaces the old.

An installed plugin is identified by its "path". The path is of the form *`scope/plugin-id`*, where *`scope`* is the first parameter to `plugin:install-from-zip`, and *`plugin-id`* is the ID in the `<id/>` element of the plugin manifest. For example, if the manifest for the above plugin contains `<id>sampleplugin-id</id>`, then the path is `native/sampleplugin-id`.

The plugin zip file can be anywhere on the filesystem when you install it, as long as the file is readable by MarkLogic Server. The installation process deploys your plugin to the Extensions database and creates a local on-disk cache inside your MarkLogic Server directory.

Installing or updating a native plugin on any host in a MarkLogic Server cluster updates the plugin for the whole cluster. However, the new or updated plugin may not be available immediately. For details, see "How MarkLogic Server Manages Native Plugins" on page 408.

22.6 Uninstalling a Native Plugin

To uninstall a native plugin, call the XQuery function `plugin:uninstall` or the Server-Side JavaScript function `plugin.uninstall`. In the first parameter, pass the scope with which you installed the plugin. In the second parameter, pass the plugin ID (the `<id/>` in the manifest). For example:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace plugin =`
	`"http://marklogic.com/extension/plugin"`
	`  at "MarkLogic/plugin/plugin.xqy";`
	
	`plugin:uninstall("native", "sampleplugin-id")`

Server-Side JavaScript	`'use strict';`
	`declareUpdate();`
	`const plugin = require('/MarkLogic/plugin/plugin');`
	
	`// Install a plugin package in the Extensions database under`
	`// the relative path 'my/native/plugin'.`
	`plugin.uninstall('native', 'sampleplugin-id');`

The plugin is removed from the Extensions database and unloaded from memory on all nodes in the cluster. There can be a slight delay before the plugin is uninstalled on all hosts. For details, see "How MarkLogic Server Manages Native Plugins" on page 408. There can be a slight delay

22.7 Registering a Native Plugin at Runtime

When you install a native plugin, it becomes available for use. The plugin is loaded on demand. When a plugin is loaded, MarkLogic Server uses a registration handshake to cache details about the plugin, such as the version and what UDFs the plugin implements.

Every C++ native plugin library must implement an `extern "C"` function called `marklogicPlugin` to perform this load-time registration. The function interface is:

```
using namespace marklogic;
extern "C" void marklogicPlugin(Registry& r) {...}
```

When MarkLogic Server loads your plugin library, it calls `marklogicPlugin` so your plugin can register itself. The exact requirements for registration depend on the interfaces implemented by your plugin, but should include at least the following:

- Register the version of your plugin by calling `marklogic::Registry::version`.
- Register the interface(s) your plugin implements by calling the appropriate `marklogic::Registry` registration method. For example, `Registry::registerAggregate` for implementations of `marklogic::AggregateUDF`.

Declare `marklogicPlugin` as required by your platform to make it accessible outside your library. For example, on Microsoft Windows, include the extended attribute `dllexport` in your declaration:

```
extern "C" __declspec(dllexport) void marklogicPlugin(Registry& r)...
```

For example, the following code registers two `AggregateUDF` implementations. For a complete example, see *marklogic_dir*`/Samples/NativePlugins`.

```
#include "MarkLogic.h"
using namespace marklogic;

class Variance : public AggregateUDF {...};
class MedianTest : public AggregateUDF {...};

extern "C" void marklogicPlugin(Registry& r)
{
    r.version();
    r.registerAggregate<Variance>("variance");
    r.registerAggregate<MedianTest>("median-test");
}
```

22.8 Versioning a Native Plugin

Your implementation of the registration function `marklogicPlugin` should include a call to `marklogic::Registry::version` to register your plugin version. MarkLogic Server uses this information to maintain plugin version consistency across a cluster.

When you deploy a new plugin version, both the old and new versions of the plugin can be present in the cluster for a short time. If MarkLogic Server detects this state when your plugin is used, MarkLogic Server reports `XDMP-BADPLUGINVERSION` and retries the operation until the plugin versions synchronize.

Calling `Registry::version` with no arguments uses a default version constructed from the compilation date and time (`__DATE__` and `__TIME__`). This ensures the version number changes every time you compile your plugin. The following example uses the default version number:

```
extern "C" void marklogicPlugin(Registry& r)
{
    r.version();
```

```
    ...
}
```

You can override this behavior by passing an explicit version to `Registry::version`. The version must be a numeric value. For example:

```
extern "C" void marklogicPlugin(Registry& r)
{
    r.version(1);
    ...
}
```

The MarkLogic Server native plugin API (`marklogic_dir/include/MarkLogic.h`) is also versioned. You cannot compile your plugin library against one version of the API and deploy it to a MarkLogic Server instance running a different version. If MarkLogic Server detects this mismatch, an `XDMP-BADAPIVERSION` error occurs.

22.9 Checking the Status of Loaded Plugins

Using the Admin Interface or the `xdmp:host-status` function, you can monitor which native plugin libraries are loaded into MarkLogic Server, as well as their versions and UDF capabilities.

> **Note:** Native plugin libraries are demand loaded when an application uses one of the UDFs implemented by the plugin. Plugins that are installed but not yet loaded will not appear in the host status.

To monitor loaded plugins using the Admin Interface:

1. In your browser, navigate to the Admin Interface: http://*yourhost*:8001.
2. Click the name of the host you want to monitor, either on the tree menu or the summary page. The host summary page appears.
3. Click the Status tab at the top right. The host status page appears.
4. Scroll down to the native plugin status section.

To examine loaded programatically, open Query Console and run a query similar to the following:

Language	Example

XQuery	`xquery version "1.0-ml";`
	`(: List native plugins loaded on this host :)`
	`xdmp:host-status(xdmp:host())//*:native-plugins`

Server-Side JavaScript	`'use strict';` `fn.head(xdmp.hostStatus(xdmp.host()))['native-plugins']`

You should see output similar to the following if there are plugins loaded. The XQuery code emits XML. The JavaScript code emits a JavaScript object (pretty-printed as JSON by Query Console). This output is the result of installing and loading the sample plugin in *MARKLOGIC_DIR*/Samples/NativePlugin, which implements several aggregate UDFs ("max", "min", etc.), a lexer UDF, and a stemmer UDF.

Language	Example

XQuery	`<native-plugins xmlns="http://marklogic.com/xdmp/status/host">`
	`  <native-plugin>`
	`  <path>native/sampleplugin/libsampleplugin.so</path>`
	`  <version>356528850</version>`
	`  <capabilities>`
	`  <aggregate>max</aggregate>`
	`  <aggregate>min_point</aggregate>`
	`  <aggregate>min</aggregate>`
	`  <aggregate>variance</aggregate>`
	`  <aggregate>median-test</aggregate>`
	`  <aggregate>max_dateTime</aggregate>`
	`  <aggregate>max_string</aggregate>`
	`  <lexer>sample_lexer</lexer>`
	`  <stemmer>sample_stemmer</stemmer>`
	`  </capabilities>`
	`  </native-plugin>`
	`</native-plugins>`

Server-Side JavaScript	`[{`
	`  "path":"native/sampleplugin/libsampleplugin.so",`
	`  "version":"356528850",`
	`  "capabilities":[`
	`  "max", "min_point", "min", "variance",`
	`  "median-test", "max_dateTime", "max_string",`
	`  "sample_lexer",`
	`  "sample_stemmer"]`
	`}]`

22.10 The Plugin Manifest

A native plugin zip file must include a manifest file called `manifest.xml`. The manifest file must contain the plugin name, plugin id, and a `<native>` element for each native plugin implementation library in the zip file. The manifest file can also include optional metadata such as provider and plugin description. For full details, see the schema in *MARKLOGIC_INSTALL_DIR*/Config/plugin.xsd.

Paths to the plugin library and dependent libraries must be relative.

You can use the same manifest on multiple platforms by specifying the native plugin library without a file extension or, on Unix, `lib` prefix. If this is the case, then MarkLogic Server forms the library name in a platform specific fashion, as shown below:

- Windows: Add a `.dll` extension
- Linux: Add a `lib` prefix and a `.so` extension
- Mac OS X: Add a `lib` prefix and a `.dylib` extension

The following example is the manifest for a native plugin with the ID “sampleplugin-id”, implemented by the shared library `libsampleplugin.so`.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin xmlns="http://marklogic.com/extension/plugin">
  <name>sampleplugin-name</name>
  <id>sampleplugin-id</id>
  <version>1.0</version>
  <provider-name>MarkLogic</provider-name>
  <description>Example native plugin</description>
  <native>
    <path>libsampleplugin.so</path>
  </native>
</plugin>
```

If the plugin package includes dependent libraries, list them in the `<native>` element. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin xmlns="http://marklogic.com/extension/plugin">
  <name>sampleplugin-name</name>
  ...
  <native>
    <path>libsampleplugin.so</path>
    <dependency>libdep1.so</dependency>
    <dependency>libdep2.so</dependency>
  </native>
</plugin>
```

22.11 Native Plugin Security Considerations

Administering (installing, updating or uninstalling) a native plugin requires the following:

- The `http://marklogic.com/xdmp/privileges/plugin-register` privilege, or
- The `application-plugin-registrar rol`e.

Loading and running a native plugin can be controlled in two ways:

- The `native-plugin` privilege (`http://marklogic.com/xdmp/privileges/native-plugin`) enables the use of all native plugins.
- You can define a plugin-specific privelege of the form `http://marklogic.com/xdmp/privileges/native-plugin/`*`plugin-path`* to enable users to use a specific privilege.

The *plugin-path* is same plugin library path you use when invoking the plugin. For example, if you install the following plugin and its manifest specifies the plugin path as "sampleplugin", then the plugin-specific privilege would be `http://marklogic.com/xdmp/privileges/native-plugin/native/sampleplugin`.

```
plugin:install-from-zip("native",
  xdmp:document-get("/space/udf/sampleplugin.zip")/node())
```

The plugin-specific privilege is not pre-defined for you. You must create it. However, MarkLogic Server will honor it if it is present.

22.12 Native Plugin Example

You can explore a sample native plugin through the source code and makefile in *MARKLOGIC_DIR*`/Samples/NativePlugins`. This example implements several kinds of UDF.

The sample Makefile will lead you through compiling, linking, and packaging the native plugin. The `README.txt` provides instructions for installing and exercising the plugin library.

23.0 Aggregate User-Defined Functions

This chapter describes how to create user-defined aggregate functions. This chapter includes the following sections:

- What Are Aggregate User-Defined Functions?
- In-Database MapReduce Concepts
- Implementing an Aggregate User-Defined Function

23.1 What Are Aggregate User-Defined Functions?

Aggregate functions are functions that take advantage of the MapReduce capabilities of MarkLogic Server to analyze values in lexicons and range indexes. For example, computing a sum or count over an element, attribute, or field range index. Aggregate functions are best used for analyses that produce a small number of results, rather than analyses that produce results in proportion to the number of range index values or the number of documents processed.

MarkLogic Server provides a C++ interface for defining your own aggregate functions. You build your aggregate user-defined functions (UDFs) into a dynamically linked library, package it as a *native plugin*, and install the plugin in MarkLogic Server. To learn more about native plugins, see "Using Native Plugins" on page 407.

A native plugin is automatically distributed throughout your MarkLogic cluster. When an application calls your aggregate UDF, your library is dynamically loaded into MarkLogic Server on each host in the cluster that participates in the analysis. To understand how your aggregate function runs across a cluster, see "How In-Database MapReduce Works" on page 418.

This chapter covers how to implement an aggregate UDF. For information on using aggregate UDFs, see Using Aggregate User-Defined Functions in the *Search Developer's Guide*.

23.2 In-Database MapReduce Concepts

MarkLogic Server uses In-Database MapReduce to efficiently parallelize analytics processing across the hosts in a MarkLogic cluster, and to move that processing close to the data.

This section covers the following topics:

- What is MapReduce?
- How In-Database MapReduce Works

You can explicitly leverage In-Database MapReduce efficiencies by using builtin and user-defined aggregate functions. For details, see Using Aggregate Functions in the *Search Developer's Guide*.

23.2.1 What is MapReduce?

MapReduce is a distributed, parallel programming model in which a large data set is split into subsets that are independently processed by passing each data subset through parallel map and reduce tasks. Usually, the map and reduce tasks are distributed across multiple hosts.

Map tasks calculate intermediate results by passing the input data through a map function. Then, the intermediate results are processed by reduce tasks to produce final results.

MarkLogic Server supports two types of MapReduce:

- In-database MapReduce distributes processing across a MarkLogic cluster when you use qualifying functions, such as builtin or user-defined aggregate functions. For details, see "How In-Database MapReduce Works" on page 418.
- External MapReduce distributes work across an Apache Hadoop cluster while using MarkLogic Server as the data source or result repository. For details, see the *MarkLogic Connector for Hadoop Developer's Guide*.

23.2.2 How In-Database MapReduce Works

In-Database MapReduce takes advantage of the internal structure of a MarkLogic Server database to do analysis close to the data. When you invoke an Aggregate User-Defined Function, MarkLogic Server executes it using In-Database MapReduce.

MarkLogic Server stores data in structures called forests and stands. A large database is usually stored in multiple forests. The forests can be on multiple hosts in a MarkLogic Server cluster. Data in a forest can be stored in multiple stands. For more information on how MarkLogic Server organizes content, see Understanding Forests in the *Administrator's Guide* and Clustering in MarkLogic Server in the *Scalability, Availability, and Failover Guide*.

In-Database MapReduce analysis works as follows:

1. Your application calls an In-Database MapReduce function such as `cts:sum-aggregate` or `cts:aggregate`. The e-node where the function is evaluated begins a MapReduce *job*.

2. The originating e-node distributes the work required by the job among the local and remote forests of the target database. Each unit of work is a *task* in the job.

3. Each participating host runs map tasks in parallel to process data on that host. There is at least one map task per forest that contains data needed by the job.

4. Each participating host runs reduce tasks to roll up the local per stand map results, then returns this intermediate result to the originating e-node.

5. The originating e-node runs reduce tasks to roll up the results from each host.

6. The originating e-node runs a "finish" operation to produce the final result.

23.3 Implementing an Aggregate User-Defined Function

You can create an aggregate user-defined function (UDF) by implementing a subclass of the `marklogic::AggregateUDF` C++ abstract class and deploying it as a native plugin.

The section covers the following topics:

- Creating and Deploying an Aggregate UDF
- Implementing AggregateUDF::map
- Implementing AggregateUDF::reduce
- Implementing AggregateUDF::finish
- Registering an Aggregate UDF
- Aggregate UDF Memory Management
- Implementing AggregateUDF::encode and AggregateUDF::decode
- Aggregate UDF Error Handling and Logging
- Aggregate UDF Argument Handling
- Type Conversions in Aggregate UDFs

23.3.1 Creating and Deploying an Aggregate UDF

An aggregate user-defined function (UDF) is a C++ class that performs calculations across MarkLogic range index values or index value co-occurrences. When you implement a subclass of `marklogic::AggregateUDF`, you write your own in-database map and reduce functions usable by an XQuery, Java, or REST application. The MarkLogic Server In-Database MapReduce framework handles distributing and parallelizing your C++ code, as described in "How In-Database MapReduce Works" on page 418.

> **Note:** An aggregate UDF runs in the same memory and process space as MarkLogic Server, so errors in your plugin can crash MarkLogic Server. Before deploying an aggregate UDF, you should read and understand "Using Native Plugins" on page 407.

To create and deploy an aggregate UDF:

1. Implement a subclass of the C++ class `marklogic::AggregateUDF`. See *marklogic_dir*`/include/MarkLogic.h` for interface details.

2. Implement an `extern "C"` function called `marklogicPlugin` to perform plugin registration. See "Registering a Native Plugin at Runtime" on page 411.

3. Package your implementation into a native plugin. See "Packaging a Native Plugin" on page 409.

4. Install the plugin by calling the XQuery function `plugin:install-from-zip`. See "Installing a Native Plugin" on page 410.

A complete example is available in *`marklogic_dir`*`/Samples/NativePlugins`. You should use the sample Makefile as the basis for building your plugin. For more details, see "Building a Native Plugin Library" on page 408.

The table below summarizes the key methods of `marklogic::AggregateUDF` that you must implement:

Method Name	Description
`start`	Initialize the state of a job and process arguments. Called once per job, on the originating e-node.
`map`	Perform the map calculations. Called once per map task (at least once per stand of the database containing target content). May be called on local and remote objects. For example, in a mean aggregate, calculate a sum and count per stand.
`reduce`	Perform reduce calculations, rolling up the map results. Called N-1 times, where N = # of map tasks. For example, in a mean aggregate, calculate a total sum and count across the entire input data set.
`finish`	Generate the final results returned to the calling application. Called once per job, on the originating e-node. For example, in a mean aggregate, calculate the mean from the sum and count.
`clone`	Create a copy of an aggregate UDF object. Called at least once per map task to create an object to execute your `map` and `reduce` methods.
`close`	Notify your implementation that a cloned object is no longer needed.
`encode`	Serialize your aggregate UDF object so it can be transmitted to a remote host in the cluster.
`decode`	Deserialize your aggregate UDF object after it has been transmitted to/from a remote host.

23.3.2 Implementing AggregateUDF::map

`AggregateUDF::map` has the following signature:

```
virtual void map(TupleIterator&, Reporter&);
```

Use the `marklogic::TupleIterator` to access the input range index values. Store your map results as members of the object on which `map` is invoked. Use the `marklogic::Reporter` for error reporting and logging; see "Aggregate UDF Error Handling and Logging" on page 428.

This section covers the following topics:

- Iterating Over Index Values with TupleIterator
- Controlling the Ordering of Map Input Tuples

23.3.2.1 Iterating Over Index Values with TupleIterator

The `marklogic::TupleIterator` passed to `AggregateUDF::map` is a sequence of the input range index values assigned to one map task. You can do the following with a `TupleIterator`:

- Iterate over the tuples using `TupleIterator::next` and `TupleIterator::done`. T
- Determine the number of values in each tuple using `TupleIterator::width`.
- Access the values in each tuple using `TupleIterator::value`.
- Query the type of a value in a tuple using `TupleIterator::type`.

If your aggregate UDF is invoked on a single range index, then each tuple contains only one value. If your aggregate UDF is invoked on N indexes, then each tuple represents one N-way co-occurrence and contains N values, one from each index. For more information, see Value Co-Occurrences Lexicons in the *Search Developer's Guide*.

The order of values within a tuple corresponds to the order of the range indexes in the invocation of your aggregate UDF. The first index contributes the first value in each tuple, and so on. Empty (null) tuple values are possible.

If you try to extract a value from a tuple into a C++ variable of incompatible type, MarkLogic Server throws an exception. For details, see "Type Conversions in Aggregate UDFs" on page 430.

In the following example, the `map` method expects to work with 2-way co-occurrences of `<name>` (string) and `<zipcode>` (int). Each tuple is a `(name, zipcode)` value pair. The name is the 0th item in each tuple; the zipcode is the 1st item.

```
#include "MarkLogic.h"
using namespace marklogic;
...
void myAggregateUDF::map(TupleIterator& values, Reporter& r)
{
    if (values.width() != 2) {
        r.error("Unexpected number of range indexes.");
        // does not return
    }
    for (; !values.done(); values.next()) {
        if (!values.null(0) && !values.null(1)) {
            String name;
```

```
            int zipcode;

            values.value(0, name);
            values.value(1, zipcode);
            // work with this tuple...
        }
    }
```

23.3.2.2 Controlling the Ordering of Map Input Tuples

MarkLogic Server passes input data to your map function through a `marklogic::TupleIterator`. By default, the tuples covered by the iterator are in descending order. You can control the ordering by overriding `AggregateUDF::getOrder`.

The following example causes input tuples to be delivered in ascending order:

```
#include "MarkLogic.h"
using namespace marklogic;
...
RangeIndex::getOrder myAggregateUDF::getOrder() const
{
    return RangeIndex::ASCENDING;
}
```

23.3.3 Implementing AggregateUDF::reduce

`AggregateUDF::reduce` folds together the intermediate results from two of your aggregate UDF objects. The object on which `reduce` is called serves as the accumulator.

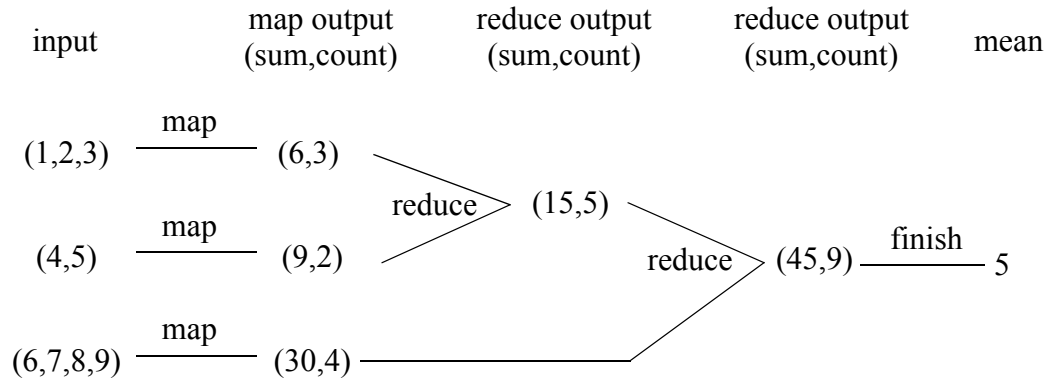
The `reduce` method has the following signature. Fold the data from the input `AggregateUDF` into the object on which reduce is called. Use the `Reporter` to report errors and log messages; see "Aggregate UDF Error Handling and Logging" on page 428.

```
virtual void reduce(const AggregateUDF*, Reporter&);
```

MarkLogic Server repeatedly invokes `reduce` until all the map results are folded together, and then invokes `finish` to produce the final result.

For example, consider an aggregate UDF that computes the arthimetic mean of a set of values. The calculation requires a sum of the values and a count of the number of values. The map tasks accumulate intermediate sums and counts on subsets of the data. When all reduce tasks complete, one object on the e-node contains the sum and the count. MarkLogic Server then invokes `finish` on this object to compute the mean.

For example, if the input range index contains the values 1-9, then the mean is 5 (45/9). The following diagram shows the map-reduce-finish cycle if MarkLogic Server distributes the index values across 3 map tasks as the sequences (1,2,3), (4,5), and (6,7,8,9):



The following code snippet is an aggregate UDF that computes the mean of values from a range index (sum/count). The `map` method (not shown) computes a sum and a count over a portion of the range index and stores these values on the aggregate UDF object. The `reduce` method folds together the sum and count from a pair of your aggregate UDF objects to eventually arrive at a sum and count over all the values in the index:

```
#include "MarkLogic.h"
using namespace marklogic;

class Mean : public AggregateUDF
{
public:
    void reduce(const AggregateUDF* o, Reporter& r)
        sum += o->sum;
        count += o->count;
    }

    // finish computes the mean from sum and count
    ....
protected:
    double sum;
    double count;
};
```

For a complete example, see *marklogic_dir*`/Samples/NativePlugin`.

23.3.4 Implementing AggregateUDF::finish

`AggregateUDF::finish` performs final calculations and prepares the output sequence that is returned to the calling application. Each value in the sequence can be either a simple value (int, string, DateTime, etc.) or a key-value map (`map:map` in XQuery). MarkLogic Server invokes `finish` on the originating e-node, once per job. MarkLogic Server invokes `finish` on the aggregate UDF object that holds the cumulative `reduce` results.

`AggregateUDF::finish` has the following signature. Use the `marklogic::OutputSequence` to record your final values or map(s). Use the `marklogic::Reporter` to report errors and log messages; see "Aggregate UDF Error Handling and Logging" on page 428.

```
virtual void finish(OutputSequence&, Reporter&);
```

Use `OutputSequence::writeValue` to add a value to the output sequence. To add a value that is a key-value map, bracket paired calls to `OutputSequence::writeMapKey` and `OutputSequence::writeValue` between `OutputSequence::startMap and OutputSequence::endMap`. For example:

```
void MyAggregateUDF::finish(OutputSequence& os, Reporter& r)
{
    // write a single value
    os.writeValue(int(this->sum/this-count));

    // write a map containing 2 key-value pairs
    os.startMap();
    os.writeMapKey("sum");
    os.writeValue(this->sum);
    os.writeMapKey("count");
    os.writeValue(this->count);
    os.endMap();
}
```

For information on how MarkLogic Server converts types between your C++ code and the calling application, see "Type Conversions in Aggregate UDFs" on page 430.

23.3.5 Registering an Aggregate UDF

You must register your Aggregate UDF implementation with MarkLogic Server to make it available to applications.

Register your implementation by calling `marklogic::Registry::registerAggregate` from `marklogicPlugin`. For details on `marklogicPlugin`, see "Registering a Native Plugin at Runtime" on page 411.

Calling `Registry::registerAggregate` gives MarkLogic Server a pointer to a function it can use to create an object of your UDF class. MarkLogic Server calls this function whenever an application invokes your aggregate UDF. For details, see "Aggregate UDF Memory Management" on page 425.

Call the template version of `marklogic::Registry::registerAggregate` to have MarkLogic Server use the default allocator and constructor. Call the virtual version to use your own object factory. The following code snippet shows the two registration interfaces:

```
// From MarkLogic.h
namespace marklogic {
```

```
typedef AggregateUDF* (*AggregateFunction)();
class Registry
{
public:
  // Calls new T() to allocate an object of your UDF class
  template<class T> void registerAggregate(const char* name);

  // Calls your factory func to allocate an object of your UDF class
  virtual void registerAggregate(const char* name, AggregateFunction);
  ...
};
}
```

The string passed to `Registry::registerAggregate` is the name applications use to invoke your plugin. For example, as the second parameter to `cts:aggregate` in XQuery:

```
cts:aggregate("pluginPath", "ex1", ...)
```

Or, as the value of the `aggregate` parameter to `/values/{name}` using the REST Client API:

```
GET /v1/values/theLexicon?aggregate=ex1&aggregatePath=pluginPath
```

The following example illustrates using the template function to register `MyFirstAggregate` with the name “ex1” and the virtual member function to register a second aggregate that uses an object factory, under the name “ex2”.

```
#include "MarkLogic.h"
using namespace marklogic;
...
AggregateUDF* mySecondAggregateFactory() {...}

extern "C" void marklogicPlugin(Registry& r)
{
    r.version();
    r.registerAggregate<MyFirstAggregate>("ex1");
    r.registerAggregate("ex2", &mySecondAggregateFactory);
}
```

23.3.6 Aggregate UDF Memory Management

This section gives an overview of how MarkLogic Server creates and destroys objects of your aggregate UDF class.

- Aggregate UDF Object Lifetime
- Using a Custom Allocator With Aggregate UDFs

23.3.6.1 Aggregate UDF Object Lifetime

Objects of your aggregate UDF class are created in two ways:

- When you register your plugin, the registration function calls `marklogic::Registry::registerAggregate`, giving MarkLogic Server a pointer to function that creates objects of your `AggregateUDF` subclass. This function is called when an application invokes one of your aggregate UDFs, prior to calling `AggregateUDF::start`.
- MarkLogic Server calls `AggregateUDF::clone` to create additional objects, as needed to execute map and reduce tasks.

MarkLogic Server uses `AggregateUDF::clone` to create the transient objects that execute your algorithm in map and reduce tasks when your UDF is invoked. MarkLogic Server creates at least one clone per forest when evaluating your aggregate function.

When a clone is no longer needed, such as at the end of a task or job, MarkLogic Server releases it by calling `AggregateUDF::close`.

The `clone` and `close` methods of your aggregate UDF may be called many times per job.

23.3.6.2 Using a Custom Allocator With Aggregate UDFs

If you want to use a custom allocator and manage your own objects, implement an object factory function and supply it to `marklogic::Registry::registerAggregate`, as described in "Registering an Aggregate UDF" on page 424.

The factory function is called whenever an application invokes your plugin. That is, once per call to `cts:aggregate` (or the equivalent). Additional objects needed to execute map and reduce tasks are created using `AggregateUDF::clone`.

The factory function must conform to the `marklogic::AggregateFunction` interface, shown below:

```
// From MarkLogic.h
namespace marklogic {

typedef AggregateUDF* (*AggregateFunction)();
}
```

The following example demonstrates passing an object factory function to `Registry::registerAggregate`:

```
#include "MarkLogic.h"
using namespace marklogic;
...
AggregateUDF* myAggregateFactory() { ... }

extern "C" void marklogicPlugin(Registry& r)
{
    r.version();
    r.registerAggregate("ex2", &myAggregateFactory);
}
```

The object created by your factory function and `AggregateUDF::clone` must persist until MarkLogic Server calls your `AggregateUDF::close` method.

Use the following entry points to control the allocation and deallocation of your your aggregate UDF objects:

- The `AggregateFunction` you pass to `Registry::registerAggregate`.
- Your `AggregateUDF::clone` implementation
- Your `AggregateUDF::close` implementation

23.3.7 Implementing AggregateUDF::encode and AggregateUDF::decode

MarkLogic Server uses `Aggregate::encode` and `Aggregate::decode` to serialize and deserialize your aggregate objects when distributing aggregate analysis across a cluster. These methods have the following signatures:

```
class AggregateUDF
{
public:
  ...
  virtual void encode(Encoder&, Reporter&) = 0;
  virtual void decode(Decoder&, Reporter&) = 0;
  ...
};
```

You must provide implementations of `encode` and `decode` that adhere to the following guidelines:

- Encode/decode the implementation-specific state on your objects.
- You can encode data members in any order, but you must be consistent between encode and decode. That is, you must decode members in the same order in which you encode them.

Encode/decode your data members using `marklogic::Encoder` and `marklogic::Decoder`. These classes provide helper methods for encoding and decoding the basic item types and an arbitrary sequence of bytes. For details, see *`marklogic_dir`*`/include/MarkLogic.h`.

The following example demonstrates how to encode/decode an aggregate UDF with 2 data members, `sum` and `count`. Notice that the data members are encoded and decoded in the same order.

```
#include "MarkLogic.h"

using namespace marklogic;

class Mean : public AggregateUDF
{
public:
  ...
```

```
  void encode(Encoder& e, Reporter& r)
    {
      e.encode(this->sum);
      e.encode(this->count);
    }
  void decode(Decoder& d, Reporter& r)
    {
      d.decode(this->sum);
      d.decode(this->count);
    }
  ...
protected:
  double sum;
  double count;
};
```

23.3.8 Aggregate UDF Error Handling and Logging

Use `marklogic::Reporter` to log messages and notify MarkLogic Server of fatal errors. Your code should not report errors to MarkLogic Server by throwing exceptions.

Report fatal errors using `marklogic::Reporter::error`. When you call `Reporter::error`, control does not return to your code. The reporting task stops immediately, no additional related tasks are created on that host, and the job stops prematurely. MarkLogic Server returns `XDMP-UDFERR` to the application. Your error message is included in the `XDMP-UDFERR` error.

Note: The job does not halt immediately. The task that reports the error stops, but other in-progress map and reduce tasks may still run to completion.

Report non-fatal errors and other messages using `marklogic::Reporter::log`. This method logs a message to the MarkLogic Server error log, `ErrorLog.txt`, and returns control to your code. Most methods of `AggregateUDF` have `marklogic::Reporter` input parameter.

The following example aborts the analysis if the caller does not supply a required parameter and logs a warning if the caller supplies extra parameters:

```
#include "MarkLogic.h"
using namespace marklogic;
...
void ExampleUDF::start(Sequence& arg, Reporter& r)
{
  if (arg.done()) {
    r.error("Required parameter not found.");
  }
  arg.value(target_);
  arg.next();
  if (!arg.done()) {
    r.log(Reporter::Warning, "Ignoring extra parameters.");
  }
}
```

23.3.9 Aggregate UDF Argument Handling

This section covers the following topics:

- Passing Arguments to an Aggregate UDF
- Processing Arguments in AggregateUDF::start
- Example: Passing Arguments to an Aggregate UDF

23.3.9.1 Passing Arguments to an Aggregate UDF

Arguments can only be passed to aggregate UDFs from XQuery. The Java and REST client APIs do not support argument passing.

From XQuery, pass an argument sequence in the 4th parameter of `cts:aggregate`. The following example passes two arguments to the "count" aggregate UDF:

```
cts:aggregate(
  "native/samplePlugin",
  "count",
  cts:element-reference(xs:QName("name"),
  (arg1,arg2))
```

The arguments reach your plugin as a `marklogic::Sequence` passed to `AggregateUDF::start`. For details, see "Processing Arguments in AggregateUDF::start" on page 429.

For a more complete example, see "Example: Passing Arguments to an Aggregate UDF" on page 430.

23.3.9.2 Processing Arguments in AggregateUDF::start

MarkLogic Server makes your aggregate-specific arguments available through a `marklogic::Sequence` passed to `AggregateUDF::start`.

```
class AggregateUDF
{
public:
  ...
  virtual void start(Sequence& arg, Reporter&) = 0;
  ...
};
```

The Sequence class has methods for iterating over the argument values (`next` and `done`), checking the type of the current argument (`type`), and extracting the current argument value as one of several native types (`value`).

Type conversions are applied during value extraction. For details, see "Type Conversions in Aggregate UDFs" on page 430.

If you need to propagate argument data to your `map` and `reduce` methods, copy the data to a data member of the object on which `start` is invoked. Include the data member in your `encode` and `decode` methods to ensure the data is available to remote map and reduce tasks.

23.3.9.3 Example: Passing Arguments to an Aggregate UDF

Consider an aggregate UDF that counts the number of 2-way co-occurrences where one of the index values matches a caller-supplied value. In the following example, the caller passes in the value 95008 to `cts:aggregate`:

```
xquery version "1.0-ml";
cts:aggregate("native/sampleplugin", "count",
    (cts:element-reference(xs:QName("zipcode"))
    ,cts:element-reference(xs:QName("name"))
    ),
    95008
 )
```

The `start` method shown below extracts the argument value from the input Sequence and stores it in the data member `ExampleUDF::target`. The value is automatically propagated to all tasks in the job when MarkLogic Server clones the object on which it invokes `start`.

```
using namespace marklogic;
...
void ExampleUDF::
start(Sequence& arg, Reporter& r)
{
  if (arg.done()) {
    r.error("Required argument not found.");
  } else {
    arg.value(this->target);
    arg.next();
    if (!arg.done()) {
      r.log(Reporter::Warning, "Ignoring extra arguments.");
    }
  }
}
```

23.3.10 Type Conversions in Aggregate UDFs

The MarkLogic native plugin API models XQuery values as equivalent C++ types, using either primitive types or wrapper classes. You should understand these type equivalences and the type conversions supported between them because values passed between your aggregate UDF and a calling application pass through the MarkLogic Server XQuery evaluator core even if the application is not implemented in XQuery.

- Where Type Conversions Apply
- Type Conversion Example
- C++ and XQuery Type Equivalences

23.3.10.1Where Type Conversions Apply

Your plugin interacts with native XQuery values in the following places:

- Arguments passed to your plugin from the calling application through `marklogic::Sequence`.
- Range index values passed to `AggregateUDF::map` through `marklogic::TupleIterator`.
- Results returned to the application by `AggregateUDF::finish` through `marklogic::OutputSequence`.

All these interfaces (`Sequence`, `TupleIterator`, `OutputSequence`) provide methods for either inserting or extracting values as C++ types. For details, see *`marklogic_dir`*`/include/Marklogic.h`.

Where the C++ and XQuery types do not match exactly during value extraction, XQuery type casting rules apply. If no conversion is available between two types, MarkLogic Server reports an error such as `XDMP-UDFBADCAST` and aborts the job. For details on XQuery type casting, see:

http://www.w3.org/TR/xpath-functions/#Casting

23.3.10.2Type Conversion Example

In this example, the aggregate UDF expects an integer value and the application passes in a string that can be converted to a numeric value using XQuery rules. You can extract the value directly as an integer. If the calling application passes in "12345":

```
(: The application passes in the arg "12345" :)
cts:aggregate("native/samplePlugin", "count", "12345")
```

Then your C++ code can safely extract the arg directly as an integral value:

```
// Your plugin can safely extract the arg as int
void YourAggregateUDF::start(Sequence& arg, Reporter& r)
{
    int theNumber = 0;
    arg.value(theNumber);
}
```

If the application instead passes a non-numeric string such "dog", the call to `Sequence::value` raises an exception and stops the job.

23.3.10.3C++ and XQuery Type Equivalences

The table below summarizes the type equivalences between the C++ and XQuery types supported by the native plugin API. All C++ class types below are declared in *`marklogic_dir`*`/include/MarkLogic.h`.

XQuery Type	C++ Type
xs:int	int
xs:unsignedInt	unsigned
xs:long	int64_t
xs:unsignedLong	uint64_t
xs:float	float
xs:double	double
xs:boolean	bool
xs:decimal	marklogic::Decimal
xs:dateTime	marklogic::DateTime
xs:time	marklogic::Time
xs:date	marklogic::Date
xs:gYearMonth	marklogic::GYearMonth
xs:gYear	marklogic::GYear
xs:gMonth	marklogic::GMonth
xs:gDay	marklogic::GDay
xs:yearMonthDuration	marklogic::YearMonthDuration
xs:dayTimeDuration	marklogic::DayTimeDuration
xs:string	marklogic::String
xs:anyURI	marklogic::String
cts:point	marklogic::Point
map:map	marklogic::Map
item()*	marklogic::Sequence

24.0 Redacting Document Content

Redaction is the process of eliminating or obscuring portions of a document as you read it from the database. For example, you can use redaction to eliminate or mask sensitive personal information such as credit card numbers, phone numbers, or email addresses from documents. This chapter describes redaction features you can use when reading a document from the database.

> **Note:** Advanced Security License option is required when using redaction.

This chapter covers the following topics:

- Terms and Definitions
- Introduction to Redaction
- Example: Getting Started With Redaction
- Security Considerations
- Defining Redaction Rules
- Installing Redaction Rules
- Applying Redaction Rules
- Validating Redaction Rules
- Built-in Redaction Function Reference
- Example: Using the Built-In Redaction Functions
- User-Defined Redaction Functions
- Example: Using Custom Redaction Rules
- Using Dictionary-Based Masking
- Example: Dictionary-Based Masking
- Salting Masking Values for Added Security
- Preparing to Run the Examples

24.1 Terms and Definitions

The following terms are used in this chapter:

Term	Definition
redaction	The process of modifying a document to obscure or conceal sensitive information. You can redact XML and JSON documents.
redaction rule	A specification of what portion of a document to redact and what function to use to make the modification. Rules can be defined in XML or JSON. For details, see “Defining Redaction Rules” on page 445.
rule document	A document containing exactly one redaction rule. Rule documents must be installed in the schema database and be part of a collection before you can use them to redact content. For details, see “Installing Redaction Rules” on page 456.
rule collection	A database collection that only includes rule documents. A rule must be part of a collection before you can use it to redact documents.
redaction function	A function used to modify content during redaction. A redaction rule must include a redaction function specification. MarkLogic provides several built-in redaction functions. You can also create user-defined redaction functions. For details, see “Built-in Redaction Function Reference” on page 462 and “User-Defined Redaction Functions” on page 498.
source document	A database document to which you apply one or more redaction rules. Redacting a document creates an in-memory copy. The source document is unmodified.
masking	A form of redaction in which the original value is replaced by a new value. The new value may be deterministic or random.
deterministic masking	A form of redaction in which the original value is replaced by a new value, and the same input always yields the same output. For an example, see “mask-deterministic” on page 464.
random masking	A form of redaction in which the original value is replaced by a new, random value. The same input does not result in the same output every time. For an example, see “mask-random” on page 467.
dictionary-based masking	A form of random or deterministic masking in which the new value is drawn from a user-defined dictionary. For details, see “Using Dictionary-Based Masking” on page 514.

Term	Definition
redaction dictionary	A specially formatted collection of values that can be used as a source for dictionary-based masking. Redaction dictionaries must be installed in the schemas database. You can define a dictionary using XML or JSON. For details, see "Defining a Redaction Dictionary" on page 514.
concealment	A form of redaction in which the original value is completely hidden. The XML element or JSON property containing the redacted value is usually hidden as well, depending on the semantics of the redaction operation. For an example, see "conceal" on page 470.

24.2 Introduction to Redaction

This section provides a brief overview of the redaction feature. The following topics are covered:

- What is Redaction?
- Express Redaction Requirements Through Rules
- Apply Rules Using Multiple Interfaces
- Protection of Redaction Logic

24.2.1 What is Redaction?

The redaction feature covered in this chapter is a read transformation you can apply to XML and JSON documents. A redacted document usually has selected portions removed, replaced, or obscured when it is read from the database. For example, you might use redaction to eliminate email addresses or obscure all but the last 4 digits of credit card numbers when exporting a document from MarkLogic.

Note: Using redaction requires the Advanced Security License option.

Redaction is best suited for granular data hiding when you're exporting content from the database. For granular, real-time, in-application information hiding use Element Level Security; for more details, see Element Level Security in the *Security Guide*. For document-level access control, use security features such as document permissions and URI privileges. For more details on these and other security features in MarkLogic, see the *Security Guide*.

Warning Redaction does not secure your documents within the database. For example, even if you redact a document when it is read, applications can still search or modify the content unless you properly secure the content with features such as document permissions and Element Level Security.

The table below describes some of the techniques you can use to redact your content. The details of what to redact and what techniques to apply depend on the requirements of your application. For details, see "Choosing a Redaction Strategy" on page 447.

Redaction Type	Variations	Description
masking	full	The original value is completely obscured. For example, `123-45-6789` becomes `###-##-####`.
	partial	A portion of the original value is retained. For example, `123-45-6789` becomes `###-##-6789`.
	deterministic	The same input always results in the same redacted output. For example, the value "12345" becomes "11111" everywhere it appears in content selected for redaction.
	random	Each input results in a random redacted value. For example, the value "12345" might be masked as "1a2f578" in one place and "30da61b" in another.
	dictionary-based	A form of random or deterministic masking in which the replacement value is drawn from a user-defined redaction dictionary.
concealment		The original value (and potentially the containing XML element or JSON property) is entirely removed. For example, if you conceal the value of `/a/b`, then `<a><b>12345</b></a>` might become `</a>`.

MarkLogic supports redaction through the mlcp command line tool and an XQuery library module in the `rdt` namespace. You can also use the library module with Server-Side JavaScript.

The redaction feature includes built-in redaction functions for common redaction tasks such as obscuring social security numbers and telephone numbers. You can also plug in your own redaction functions.

24.2.2 Express Redaction Requirements Through Rules

MarkLogic uses rule-based redaction. A redaction rule tells MarkLogic how to locate the content within a document that should be redacted and how to modify that portion. A rule expresses the business logic, independent of the documents to be redacted.

A key component of a redaction rule is a redaction function specification. This function is what modifies the input nodes selected by the rule. MarkLogic provides several built-in redaction functions that you can use in your rules. For example, there are built-in redaction functions for redacting Social Security numbers, telephone numbers, and email addresses. You can also define your own redaction functions.

For details, see “Defining Redaction Rules” on page 445.

Before you can apply a rule, you must intall it in the Schemas database as part of a rule collection. For details, see “Installing Redaction Rules” on page 456.

24.2.3 Apply Rules Using Multiple Interfaces

You can apply redaction rules when reading documents from MarkLogic using the following tools and interfaces:

- mlcp command line tool
- `rdt:redact` XQuery function
- `rdt.redact` Server-Side JavaScript function

The `rdt:redact` and `rdt.redact` functions are primarily intended for testing redaction rules.

For details, see “Applying Redaction Rules” on page 457.

24.2.4 Protection of Redaction Logic

It is important that you design and implement security policies that properly protect your rules, as well as your content.

The redaction workflow enables you to protect the business logic captured in a redaction rule independent of the documents to be redacted. For example, the user who generates redacted documents need not have privileges to modify or create rules, and the user who creates and administers rules need not have privileges to read or modify the content to be redacted.

For more details, see “Security Considerations” on page 443.

24.3 Example: Getting Started With Redaction

This section walks you through a simple example of defining, installing, and applying a redaction rule. The example uses the built-in redaction functions “redact-email” and “redact-us-phone”.

In this example, rules are installed and applied using Query Console. For a similar example based on mlcp, see Example: Using mlcp for Redaction in the *mlcp User Guide*.

The walkthrough covers the following steps:

1. Installing the Source Documents

2. Installing the Rules

3. Understanding the Rules

4. Applying the Rules

24.3.1 Installing the Source Documents

Use the procedure in this section to install the sample documents into the Documents database using XQuery and Query Console. Though this example uses XQuery, you do not need to be familiar with XQuery to successfully complete the exercise.

When you complete these steps, your Documents database should contain the following documents. The documents are also inserted in a collection named "gs-samples" for easy reference.

- `/redact-gs/sample1.xml`
- `/redact-gs/sample2.json`

Follow these steps to insert the sample documents:

1. Navigate to Query Console in your browser. For example, go to `http://localhost:8000/qconsole`.

2. Paste the following script into a new query tab in Query Console.

```
xquery version "1.0-ml";
xdmp:document-insert("/redact-gs/sample1.xml",
  <personal>
    <name>Little Bopeep</name>
    <summary>Seeking lost sheep. Please call 123-456-7890.</summary>
    <id>12-3456789</id>
  </personal>,
  <options xmlns="xdmp:document-insert">
    <permissions>{xdmp:default-permissions()}</permissions>
    <collections>
      <collection>gs-samples</collection>
    </collections>
  </options>);

xquery version "1.0-ml";
xdmp:document-insert("/redact-gs/sample2.json", xdmp:unquote('
  {"personal": {
     "name": "Jack Sprat",
     "summary": "Free nutrition advice! Call (234)567-8901 now!",
     "id": "45-6789123"
  }}
'),
  <options xmlns="xdmp:document-insert">
    <permissions>{xdmp:default-permissions()}</permissions>
```

```
    <collections>
      <collection>gs-samples</collection>
    </collections>
  </options>
);
```

3. Select Documents in the Database dropdown.

4. Select XQuery in the Query Type dropdown.

5. Click the Run button. The sample documents are installed.

6. Optionally, click the Explore (eyeglass) icon next to the Database dropdown to explore the database and confirm insertion of the sample documents.

24.3.2 Installing the Rules

Rules must be installed in the schemas database associated with your content database. Rules must also be part of a collection before you can use them. This section installs rules in the Schemas database, which is the default schemas database associated with the Documents database.

You can install rules using any document insert technique. This example uses XQuery and Query Console. You do not need to be familiar with XQuery to complete this exercise. For other rule installation options, see "Installing Redaction Rules" on page 456.

When you complete this exercise, your schemas database should contain one rule defined in XML one rule defined in JSON. The rules are inserted in a collection named "gs-rules". The XML rule uses the redact-us-phone built-in redaction function. The JSON rule uses the conceal built-in redaction function.

Follow these steps to install the rules. For an explanation of what the rules do, see "Understanding the Rules" on page 440.

1. Navigate to Query Console in your browser. For example, go to `http://localhost:8000/qconsole`.

2. Paste the following script into a new query tab in Query Console.

```
(: Apply redact-us-phone to //summary :)
xquery version "1.0-ml";
xdmp:document-insert("/rules/gs/redact-phone.xml",
  <rule xml:lang="zxx" xmlns="http://marklogic.com/xdmp/redaction">
    <description>Obscure phone numbers.</description>
    <path>//summary</path>
    <method>
      <function>redact-us-phone</function>
    </method>
    <options>
```

```
      <level>partial</level>
    </options>
  </rule>,
  <options xmlns="xdmp:document-insert">
    <permissions>{xdmp:default-permissions()}</permissions>
    <collections>
      <collection>gs-rules</collection>
    </collections>
  </options>
);

(: Apply conceal to //id :)
xquery version "1.0-ml";
xdmp:document-insert("/rules/gs/conceal-id.json", xdmp:unquote('
  { "rule": {
    "description": "Remove customer ids.",
    "path": "//id",
    "method": { "function": "conceal" }
  }}
'),
  <options xmlns="xdmp:document-insert">
    <permissions>{xdmp:default-permissions()}</permissions>
    <collections>
      <collection>gs-rules</collection>
    </collections>
  </options>
);
```

3. Select Schemas in the Database dropdown.

4. Select XQuery in the Query Type dropdown.

5. Click the Run button. The rule documents are installed with the URIs "/rules/gs/redact-phone.xml" and "/rules/gs/conceal-id.json". added to the "custom-rules" collection.

24.3.3 Understanding the Rules

The XML rule installed in "Installing the Rules" on page 439 has the following form:

```
<rule xml:lang="zxx" xmlns="http://marklogic.com/xdmp/redaction">
  <description>Obscure phone numbers.</description>
  <path>//summary</path>
  <method>
    <function>redact-us-phone</function>
  </method>
  <options>
    <level>partial</level>
  </options>
</rule>
```

The rule elements have the following effect:

- `description` - Optional metadata for informational purposes.
- `path` - Apply the redaction function specified by the rule to nodes selected by the path expression "//summary".
- `method` - Use the built-in redaction function `redact-us-phone` to redact the value in a `summary` XML element or JSON property. By default, this function replaces all digits in a phone number by the character "#". You can tell this is a built-in function because `method` has no `module` child.
- `options` - Pass a `level` parameter value of "partial" to redact-us-phone, causing the function to leave the last 4 digits of the value unchanged.

The expected result of applying this rule is that any text in the value of a node named "summary" that matches the pattern of a US phone number will be replaced. The replacement value uses the "#" number to replace all but the last 4 digits. For example, a value such as 123-456-7890 is redacted to ###-###-7890. For more details, see "redact-us-phone" on page 477.

The JSON rule installed in "Installing the Rules" on page 439 has the following form:

```
{ "rule": {
    "description": "Remove customer ids.",
    "path": "//id",
    "method": { "function": "conceal" }
}}
```

The rule properties have the following effect:

- `description` - Optional metadata for informational purposes.
- `path` - Apply the redaction function specified by the rule to nodes selected by the path expression "//id".
- `method` - Use the built-in redaction function `conceal` to redact the `id` XML element or JSON property. This function will hide the nodes selected by `path`. You can tell this is a built-in function because `method` has no `module` child.

The expected result of applying this rule is to remove nodes named `id`. For example, if `//id` selects an XML element or JSON property, the element or property does not appear in the redacted output. Note that, if `//id` selects array items in JSON, the items are eliminated, but the `id` property might remain, depending on the structure of the document. For more details, see "conceal" on page 470.

24.3.4 Applying the Rules

Follow the steps in this section to apply the rules in the collection "gs-rules" to the sample documents. This example applies the rules using Query Console. You can also use the mlcp command line tool to apply rules; for more details, see "Applying Redaction Rules" on page 457.

The user who applies the rules must have read permission on the source documents, the rule documents, and the rule collection. For more details, see "Security Considerations" on page 443.

1. Navigate to Query Console in your browser. For example, go to `http://localhost:8000/qconsole`.

2. If you want to use XQuery to apply the rules, perform the following steps:

   a. Paste the following script into a new query tab in Query Console:

   ```
   xquery version "1.0-ml";
   import module namespace rdt = "http://marklogic.com/xdmp/redaction"
       at "/MarkLogic/redaction.xqy";
   rdt:redact(fn:collection("gs-samples"), "gs-rules")
   ```

   b. Select XQuery in the Query Type dropdown.

3. If you want to use Server-Side JavaScript to apply the rules, perform the following steps:

   a. Paste the following script into a new query tab in Query Console:

   ```
   const rdt = require('/MarkLogic/redaction');
   rdt.redact(fn.collection('gs-samples'), ['gs-rules']);
   ```

   b. Select JavaScript in the Query Type dropdown.

4. Select Documents in the Databases dropdown.

5. Click the Run button. The rules in the "gs-rules" collection are applied to the documents in the "gs-samples" collection.

The following table shows the result of redacting the XML sample document. Notice that the telephone number in the `summary` element has been partially redacted by the `redact-us-phone` function. Also, the `id` element has been completely hidden by the `conceal` function. The affected parts of the content are highlighted in the table.

Stage	XML Content

Original Document	`<personal>`
	`  <name>Little Bopeep</name>`
	`  <summary>Seeking lost sheep. Please call` **`123-456-7890`**`.</summary>`
	`  ` **`<id>123456</id>`**
	`</personal>`

Redacted Result	`<personal>`
	`  <name>Little Bopeep</name>`
	`  <summary>Seeking lost sheep. Please call` **`###-###-7890`**`.</summary>`
	`</personal>`

The following table shows the result of redacting the JSON sample document. Notice that the telephone number in the `summary` property has been partially redacted by the `redact-us-phone` function. Also, the `id` property has been completely hidden by the `conceal` function.The affected parts of the content are highlighted in the table.

Stage	JSON Content
Original Document	{"personal": { "name": "Jack Sprat", "summary": "Free nutrition advice! Call **(234)567-8901** now!", **"id": 234567** }}
Redacted Result	{"personal": { "name": "Jack Sprat", "summary": "Free nutrition advice! Call **(###)###-8901** now!" }}

24.4 Security Considerations

Redaction is a kind of read transformation, intended for use when exporting documents from the database. Redaction does not secure your content within the database. For example, users with sufficient document permissions can still search, read, and update documents containing the information you wish to redact. Use security features such as Element Level Security, document permissions, and URI privileges for real-time security. For more details, see the *Security Guide*.

Rule documents and rule collections are potentially sensitive information. You should carefully consider the access controls and security requirements applicable to your redaction rules and rule collections.

For example, you should implement security controls that limit exposures such as the following:

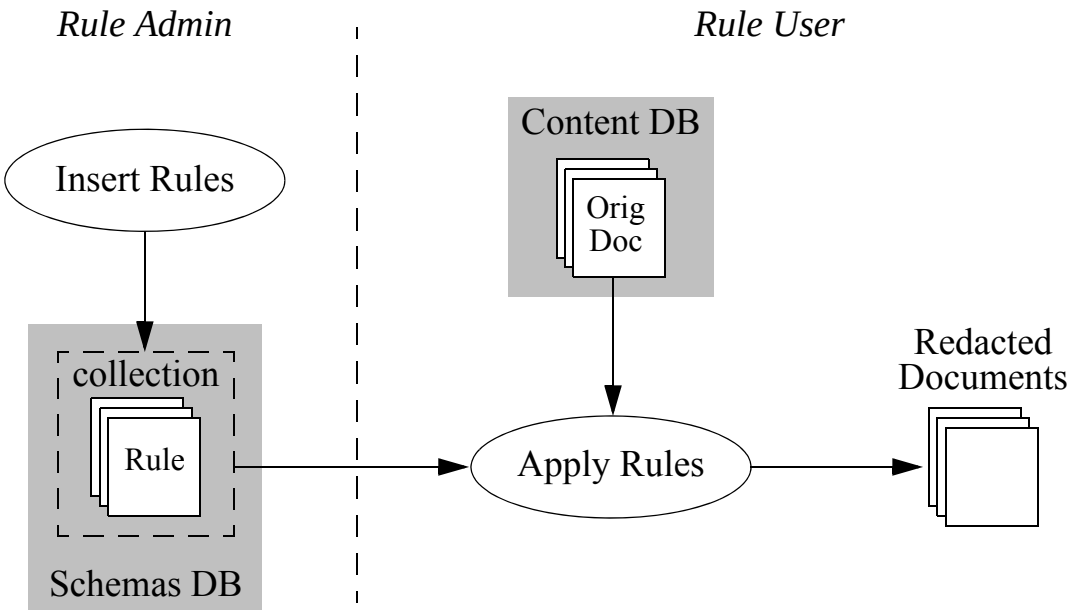
- An attacker who can read a rule has access to potentially sensitive business logic. Even if the attacker lacks read access to your content, read access to rule logic can reveal the structure of your content.
- An attacker who can modify a rule or change which rules are in a rule collection can affect the outcome of a redaction operation, exposing data that would otherwise be redacted.

Consider the following actors when designing your security architecture:

- Rule Administrators: Users who should be able to create, modify, and delete rules; manage rule collections; and create and modify redaction dictionaries. You might have multiple such users, with rights to administer different rule collections.

- Rule Users: Users who should be able to apply rules but not create, modify, or delete rules or manage rule collections. Different rule users might have access to different rules or rule collections.
- Other Users: Other users typically should not have access to or control over rule documents, rule collections, or redaction dictionaries.

The following diagram illustrates high level redaction flow and the separation of responsibilities between the rule administrator and the rule user:



Redaction Workflow

The following table lists some common tasks around administering and using redaction rules, the actor who usually performs this task, and the relevant security features available in MarkLogic. The security features are discussed in more detail below the table.

Task	Actor	Supporting Security Feature
Create or modify rule documents	Rule administrator	Document Permissions
Control which rule documents are in a rule collection	Rule administrator	Protected Collections
Create or modify redaction dictionaries	Rule administrator	Document Permissions
Use rule collections to redact documents	Rule user	Document Permissions The `redaction-user` security role

Document permissions enable you to control who can read, create, or update rule documents and redaction dictionaries. A rule administrator should usually have read and update permissions on such documents. Rule users should usually only have read permissions on rule documents and redaction dictionaries. To learn more about document permissions, see Protecting Documents in the *Security Guide*.

Placing rule documents in a protected collection enables you to control who can add documents to or remove documents from the collection. Rule administrators should usually have update permissions on a protected rule collection. Rule users should not have any special permissions on a protected rule collection. A protected collection must be explicitly created before you can add documents to it. To learn more about protected collections, see Collections and Security in the *Search Developer's Guide*.

> **Note:** A protected collection cannot be used to control who can read or modify the contents of documents in the collection; you must rely on document permissions for this control. Protected collections also cannot be used to control who can see which documents are in the collection.

MarkLogic predefines a `redaction-user` role. This role (or equivalent privileges) is required to validate rules and redact documents. That is, you must have this role to use the XQuery functions `rdt:redact` and `rdt:rule-validate`, the JavaScript functions `rdt.redact` and `rdt.ruleValidate`, or the `-redaction` option of mlcp.

To learn more about security features in MarkLogic, see the *Security Guide*.

24.5 Defining Redaction Rules

This section covers details related to authoring redaction rules. The following topics are covered:

- Rule Definition Basics
- Choosing a Redaction Strategy
- Choosing a Redaction Function
- Defining XML Namespace Prefix Bindings
- Limitations on XPath Expressions in Redaction Rules
- Defining Rules Usable on Multiple Document Formats
- XML Rule Syntax Reference
- JSON Rule Syntax Reference

24.5.1 Rule Definition Basics

You can define redaction rules in XML or JSON. The format of a rule (XML or JSON) has no effect on the type of document to which it can be applied.

A rule definition must include at least the following:

- An XPath expression defining the document components to which the rule applies. Some restrictions apply; for details, see "Limitations on XPath Expressions in Redaction Rules" on page 449.
- A descriptor specifying either a built-in or user-defined redaction function. The function performs the redaction on the node(s) selected by the path expression.

A rule definition can include additional data, such as a description or options. For details, see "XML Rule Syntax Reference" on page 452 or "JSON Rule Syntax Reference" on page 454.

Designing a rule should include the following tasks:

- Choose a redaction strategy. For example, decide whether to mask or conceal redacted values. For details, see "Choosing a Redaction Strategy" on page 447.
- Determine whether to use a built-in or user-defined redaction function. For details, see "Choosing a Redaction Function" on page 448.

The following example rule specifies that the built-in redaction function redact-us-ssn should be applied to nodes matching the XPath expression `//ssn`. The `redact-us-ssn` function accepts a `level` parameter that specifies how much of the SSN to mask (full or partial). Use the `options` section of the rule definition to specify the level.

Format	Example Rule

XML	`<rdt:rule xml:lang="zxx"`
	`  xmlns:rdt="http://marklogic.com/xdmp/redaction">`
	`  <rdt:description>Mask SSNs</rdt:description>`
	`  <rdt:path>//ssn</rdt:path>`
	`  <rdt:method>`
	`  <rdt:function>redact-us-ssn</rdt:function>`
	`  </rdt:method>`
	`  <rdt:options>`
	`  <rdt:level>partial</rdt:level>`
	`  </rdt:options>`
	`</rdt:rule>`

JSON	`{"rule": {`
	`  "description": "Mask SSNs",`
	`  "path": "//ssn",`
	`  "method": { "function": "redact-us-ssn" },`
	`  "options": { "level": "partial" }`
	`}}`

If you apply these rules to example documents from "Preparing to Run the Examples" on page 526, you should see the ssn XML element and JSON property values such as the following:

```
###-##-7890
###-##-9012
###-##-6789
###-##-8901
```

You can also create your own XQuery or Server-Side JavaScript redaction functions and define rules that apply them. A user-defined function is identified in the `method` XML element or JSON property by function name, URI of the implementing module, and the module namespace URI (if your function is implemented in XQuery). For details, see "User-Defined Redaction Functions" on page 498.

The following example specifies that the user-defined redaction function "redact-name" should be applied to nodes matching the XPath expression `//name`. For more details and examples, see "User-Defined Redaction Functions" on page 498.

Format	Example Rule

XML	`<rdt:rule xml:lang="zxx"`
	`  xmlns:rdt="http://marklogic.com/xdmp/redaction">`
	`  <rdt:description>Mask names</rdt:description>`
	`  <rdt:path>//name</rdt:path>`
	`  <rdt:method>`
	`  <rdt:function>redact</rdt:function>`
	`  <rdt:module>/example/redact-name.xqy</rdt:module>`
	`  <rdt:module-namespace>`
	`  http://marklogic.com/example/redaction`
	`  </rdt:module-namespace>`
	`  </rdt:method>`
	`</rdt:rule>`

JSON	`{"rule": {`
	`  "description": "Mask names",`
	`  "path": "//name",`
	`  "method": {`
	`  "function": "redact",`
	`  "module": "/example/redact-name.sjs"`
	`  }`
	`}}`

24.5.2 Choosing a Redaction Strategy

Redaction usually changes content in one of the following ways:

- Partial masking: Replace only a portion of the redacted value. For example, replace all but the last 4 digits in a credit card number with the character "#".
- Full masking: Replace the entire redacted value with a new value. For example, replace all characters in an account number with a random string of characters.
- Concealment: Completely eliminate the redacted value or node.

When using masking, you should also consider the following points:

- Should the replacement value always be the same for a given input (deterministic), or should it be randomized?

  Deterministic masking can preserve relationships between values and facilitate searches, which can be either beneficial or undesirable, depending on the application.

- Should the replacement value be drawn from a known list of values (a dictionary)?

  When you do not use a dictionary, the replacement value is either a randomly generated or repeating set of characters, depending on whether you choose random or deterministic masking. A redaction dictionary enables you to source replacement values from a pre-defined set of values instead.

- Is it important to preserve or obscure the "shape" of the input data?

  For example, when you redact "John Smith", should the resulting value be two words or one? Should the word length of the original input be preserved, or should it be normalized to something such as "FIRSTNAME LASTNAME"?

Once you determine the privacy requirements of your application, you can select an appropriate built-in redaction function or create one of your own.

24.5.3 Choosing a Redaction Function

A redaction function implements the logic of a given redaction rule, such as determining whether or not a node needs to be modified, generating a replacement value, or hiding a value or node. You can use one of the built-in redaction functions or create a user-defined redaction function.

The following built-in redaction functions are installed with MarkLogic. These functions should meet the needs of most applications. These functions are discussed in detail in "Built-in Redaction Function Reference" on page 462. Examples are included with each function description.

- `mask-deterministic`
- `mask-random`
- `conceal`
- `redact-number`
- `redact-regex`
- `redact-us-ssn`
- `redact-us-phone`
- `redact-email`
- `redact-ipv4`

- `redact-datetime`

If the built-in functions do not meet the needs of your application, you can create your own redaction function using XQuery or Server-Side JavaScript. For example, you might need a user-defined function to implement conditional redaction such as "redact the name if the customer is a minor". For more details, see "User-Defined Redaction Functions" on page 498.

24.5.4 Defining XML Namespace Prefix Bindings

If you need to use namespace prefixes in the `path` XPath expression, define the namespace prefix binding by adding a `namespaces` component to your rule. For example, the following rule snippet uses an "emp" namespace prefix in its path value, and then defines a binding between the "emp" prefix and the namespace URI "http://my/employees".

Rule Format	Rule Snippet

XML	`<rdt:rule ...>`
	`  <rdt:path>//emp:ssn</rdt:path>`
	`  <rdt:namespaces>`
	`  <rdt:namespace>`
	`  <rdt:prefix>emp</rdt:prefix>`
	`  <rdt:namespace-uri>http://my/employees</rdt:namespace-uri>`
	`  </rdt:namespace>`
	`  <rdt:namespace>...</rdt:namespace>`
	`  </rdt:namespaces>`
	`  <rdt:method>...</rdt:method>`
	`</rdt:rule>`

JSON	`{"rule": {`
	`  "path": "//emp:ssn",`
	`  "namespaces": [`
	`  {"namespace": {`
	`  "prefix": "emp"`
	`  "namespace-uri": "http://my/employees"`
	`  }}, ...`
	`  ],`
	`  "method": { ... }`
	`}}`

24.5.5 Limitations on XPath Expressions in Redaction Rules

Redaction rules applied to XML documents are restricted to the subset of XPath supported by XSLT. For example, you cannot use backward axes such as `parent::*`. The supported subset is defined in https://www.w3.org/TR/xslt#patterns.

Redaction rules applied to JSON documents have no such restrictions. However, if you apply rules to a mix of XML and JSON documents, you should limit your rules to the supported XPath subset.

Rule validation does not check the rule path for conformance to this limitation because it cannot know if the rule will ever be applied to an XML document. If you apply a rule to an XML document with an invalid path, the exception `RDT-INVALIDRULEPATH` is raised.

24.5.6 Defining Rules Usable on Multiple Document Formats

This section discusses important considerations when defining rules you expect to apply to both XML and JSON documents.

The XPath expression in the `path` XML element or JSON property of a rule is restricted to the subset of XPath supported by XSLT when the rule is applied to XML documents. Therefore, you must restrict your rule paths when redacting a mixture of XML and JSON context. For more details, see "Limitations on XPath Expressions in Redaction Rules" on page 449.

You must understand the interactions between XPath and the document model to ensure proper selection of nodes by a redaction rule. The XML and JSON document models differ in ways that can be surprising if you are not familiar with the models. For example, a simple path expression such as "//id" might match an element in an XML document, but all the items in an array value in JSON.

The built-in redaction functions compensate for differences in the JSON and XML document models in most cases, so they behave in a consistent way regardless of document type. If you write your own redaction functions, you might need to make similar adjustments.

You can write a single XPath expression that selects nodes in both XML and JSON documents, but if you do not understand the document models thoroughly, it might not select the nodes you expect. Keep the following tips in mind:

- XML and JSON contain different node types. Only XML documents contain element and attribute nodes; only JSON documents contain object, text, number, boolean, and null nodes. Thus, an expression such as "//@color" will never match nodes in a JSON document, even if the document contains a "color" property.
- There is no "JSON property node". A JSON document such as `{"a": 42}` is modeled as an unnamed root object node with a single number node child. The number node is named "a" and has the value 42. You can change the value of the number node, but you can only conceal the property by manipulating the parent object node.
- Each item in a JSON array is a node with same name. For example, given `{"a": [1,2]}`, the path expression "//a" selects two number nodes, not the containing array node. Selecting the array node requires a JSON specific path expression such as `//array-node('a')`. Thus, concealing an array-valued property requires a different strategy than concealing, say, a string-valued property.
- A JSON property node whose name is not a valid XML element local name, such as one that contains whitespace, can only be selected using a node test operator such as `node(`*`name`*`)`. For example, given a document such as `{"aa bb": "value"}`, use the path expression `/node('aa bb')` to select the property named "aa bb".

- The `fn:data()` function aggregates text children of XML elements, but does not do so for JSON properties. See the example in the table below.

For more details, see "Working With JSON" on page 355.

Any redaction function that can receive input from both XML and JSON must be prepared to handle multiple node types. For example, the same XPath expression might select an element node in XML, but an object node in JSON.

The rest of this section demonstrates some of the XML and JSON document model differences you should be aware of. For a more detailed discussion of XPath over JSON, see "Traversing JSON Documents Using XPath" on page 357.

Suppose you are redacting the following example documents:

XML	JSON

<person>
  <name>
  <first>John</first>
  <last>Smith</last>
  </name>
  <id>1234</id>
  <alias>Johnboy</alias>
  <alias>Smitty</alias>
</person>
	{ "person": {
	  "name": {
	  "first": "John",
	  "last": "Smith"
	  },
	  "id": 1234,
	  "alias": ["Johnboy", "Smitty"],
	  "home phone": "123-4567"
	}}

Then the following table summarizes the nodes selected by several XPath expressions.

XPath	Selected XML Nodes	Selected JSON Nodes
`//id`	an element node: `<id>1234</id>`	a number node equivalent to the constructor expression: `number-node {"id":1234}`

`//alias`	two element nodes
	`<alias>Johnboy</alias>`
	`<alias>Smitty</alias>`
		two text nodes, equivalent to the constructor expression:
		
		`text {"Johnboy"}`
		`text {"Smitty"}`

`//node("alias")`	two element nodes
	`<alias>Johnboy</alias>`
	`<alias>Smitty</alias>`
		An array node and two text nodes, equivalent to the constructor expressions:
		
		`array-node {"Johnboy", "Smitty"}`
		`text {"Johnboy"}`
		`text {"Smitty"}`

`//array-node("alias")`	no match	An array node, equivalent to the constructor expression: `array-node {"Johnboy", "Smitty"}`
`//alias/text()`	two text nodes	no match

`//name/data()`	a string:
	"JohnSmith"
		an object node:
		`{`
		`"first": "John",`
		`"last": "Smith"`
		`}`

`//node("home phone")`	N/A - invalid XML localname	a text node, equivalent to the constructor expression: `text {"123-4567"}`

24.5.7 XML Rule Syntax Reference

A redaction rule expressed in XML has the following form. All rule elements must be in the default namespace `http://marklogic.com/xdmp/redaction` and should not use namespace prefixes. For JSON syntax, see "JSON Rule Syntax Reference" on page 454.

```
<rule xml:lang="zxx" xmlns="http://marklogic.com/xdmp/redaction">
  <description>any text</description>
  <path>XPath expression</path>
  <namespaces>
    <namespace>
```

```
        <prefix>namespace prefix</prefix>
        <namespace-uri>uri</namespace-uri>
      </namespace>
    </namespaces>
    <method>
      <function>redaction function name</function>
      <module>user-defined module URI</module>
      <module-namespace>user-defined module namespace</module-namespace>
    </method>
    <options>params as elements</options>
  </rule>
```

Note the presence of `rule/@xml:lang`. The `@lang` value "zxx" is not a valid language. Rather, "zxx" is a special value that tells MarkLogic not to tokenize, stem, and index this element. Though you are not required to include this setting in your rules, it is strongly recommended that you do so because rules are configuration information and not meant to be searchable.

The following table provides more detail on the `rule` child elements.

Element	Description
`description`	Optional. A description of this rule.
`path`	Required. An XPath expression identifying the content to redact. The expression must be an absolute path (begin with "/") that selects an XML and/or JSON node, such as a element, attribute, object, array, text, boolean, number, or null node. It must not select a document node. Additional restrictions may apply; for details, see "Limitations on XPath Expressions in Redaction Rules" on page 449.
`namespaces`	Optional. If the XPath expression in `path` uses namespace prefixes, define the prefix-namespace URI bindings here. For details, see "Defining XML Namespace Prefix Bindings" on page 449.

Element	Description

`method`	Required. The specification of the redaction function to apply to content matching `path`. The `function` child element is required. The `module` and `module-namespace` child elements only used to specify a user-defined redaction function, as shown below.
	
	Use this form to apply a built-in redaction function. For details, see "Built-in Redaction Function Reference" on page 462.
	
	`<method>`
	`  <function>`*builtInFuncName*`</function>`
	`</method>`
	
	Use this form to apply a user-defined function implemented in JavaScript:
	
	`<method>`
	`  <function>`*userDefinedFuncName*`</function>`
	`  <module>`*javascriptModuleURI*`</module>`
	`</method>`
	
	Use this form to apply a user-defined function implemented in XQuery:
	
	`<method>`
	`  <function>`*userDefinedFuncLocalName*`</function>`
	`  <module>`*xqueryModuleURI*`</module>`
	`  <module-namespace>`*moduleNSURI*`</module-namespace>`
	`</method>`
	
	For details, see "User-Defined Redaction Functions" on page 498.

`options`	Optional. Specify data to pass to the redaction function. Each child element becomes a map entry (XQuery) or object property (JavaScript) in the `options` parameter passed to the redaction function. The element name is the map key or property name.

24.5.8 JSON Rule Syntax Reference

A redaction rule expressed in JSON has the following form. For XML syntax, see "XML Rule Syntax Reference" on page 452.

```
{"rule": {
  "description": "any text",
  "path": "XPath expression",
  "method": {
    "function": "redaction function name",
    "module": "user-defined module URI",
    "moduleNamespace": "user-defined module namespace URI",
  },
```

```
    "namespaces": [
      {"namespace": {
        "prefix": "namespace prefix",
        "namespaceUri": "uri"
      }, ...
    ],
    "options": {
      "anyPropName": anyValue
    }
  } }
```

The following table provides more detail on each element.

Element	Description
`description`	Optional. A description of this rule.
`path`	Required. An XPath expression identifying the content to redact. The expression must be an absolute path (begin with "/") that selects an XML and/or JSON node, such as a element, attribute, object, array, text, boolean, number, or null node. The path must not select a document node. Additional restrictions may apply; for details, see "Limitations on XPath Expressions in Redaction Rules" on page 449.
`namespaces`	Optional. If the XPath expression in `path` uses namespace prefixes, define the prefix-namespace URI bindings here. For details, see "Defining XML Namespace Prefix Bindings" on page 449.

Element	Description

`method`	Required. The specification of the redaction function to apply to content matching `path`. This element must have one of the forms shown below.
	
	Use this form to apply a built-in redaction function. For details, see "Built-in Redaction Function Reference" on page 462.
	
	`"method": { "function": "builtInFuncName" }`
	
	Use this form to apply a user-defined function implemented in JavaScript:
	
	`"method": {`
	`  "function": "userDefinedFuncName",`
	`  "module": "javascriptModuleURI"`
	`}`
	
	Use this form to apply a user-defined function implemented in XQuery:
	
	`"method": {`
	`  "function": "userDefinedFuncName",`
	`  "module": "xqueryModuleURI",`
	`  "moduleNamespace": "xqueryModuleNSURI"`
	`}`
	
	For details, see "User-Defined Redaction Functions" on page 498.

`options`	Optional. Specify data to pass to the redaction function. This becomes the value of the `options` input parameter of the redaction function. For a redaction function implemented in XQuery, the options are passed to the function as a `map:map`, using the property names as map keys.

24.6 Installing Redaction Rules

Before you can use a redaction rule, it must be installed as a document in the schema database associated with the database containing the documents to be redacted.

A rule document can only contain one rule and should not contain any non-rule data. A rule collection can contain multiple rule documents, but should not contain any non-rule documents. Every rule document must be associated with at least one collection because rules are specified by collection to redaction operations.

Use any MarkLogic document insertion APIs to insert rules into the schema database, such as the `xdmp:document-insert` XQuery function, the `xdmp.documentInsert` Server-Side JavaScript function, or the document creation features of the Node.js, Java, or REST Client APIs. You can assign rules to a collection at insertion time or as a separate operation.

If you run one of the following examples in Query Console using your schema database as the context database, a rule document is inserted into the database and assigned to two collections, "pii-rules" and "security-rules".

Language	Example

XQuery	`xquery version "1.0-ml";`
	`xdmp:document-insert("/redactionRules/ssn.xml",`
	`  <rule xml:lang="zxx"`
	`  xmlns="http://marklogic.com/xdmp/redaction">`
	`  <description>hide SSNs</description>`
	`  <path>//ssn</path>`
	`  <method>`
	`  <function>redact-us-ssn</function>`
	`  </method>`
	`  <options>`
	`  <pattern>partial</pattern>`
	`  </options>`
	`  </rule>,`
	`  <options xmlns="xdmp:document-insert">`
	`  <permissions>{xdmp:default-permissions()}</permissions>`
	`  <collections>`
	`  <collection>security-rules</collection>`
	`  <collection>pii-rules</collection>`
	`  </collections>`
	`  </options>`
	`)`

Server-Side JavaScript	`declareUpdate();`
	
	`xdmp.documentInsert(`
	`  '/redactionRules/ssn.json',`
	`  { rule: {`
	`  description: 'hide SSNs',`
	`  path: '//ssn',`
	`  method: { function: 'redact-us-ssn' },`
	`  options: { pattern: 'partial' }`
	`  }},`
	`  { permissions: xdmp.defaultPermissions(),`
	`  collections: ['security-rules','pii-rules']});`

You should set permissions on your rule documents to constrain who can access or modify the rules. For more details, see "Security Considerations" on page 443.

24.7 Applying Redaction Rules

This section discusses applying redaction rules once rule collections have been installed on MarkLogic. The following topics are covered:

- Overview

- Applying Rules Using mlcp
- Applying Rules Using XQuery
- Applying Rules Using JavaScript
- No Guaranteed Ordering of Rules

The mlcp command line tool is the recommended interface because it can efficiently apply redaction to large numbers of documents when you export them from the database or copy them between databases. To learn more about mlcp, see the *mlcp User Guide*.

The `rdt:redact` and `rdt.redact` functions are suitable for debugging redaction rules or redacting small sets of documents.

24.7.1 Overview

Once you install one or more rule documents in the Schemas database and assign them to a collection, you can redact documents in the following ways:

- Exporting documents from a database using the mlcp command line tool.
- Copying documents between databases using the mlcp command line tool.
- Calling the XQuery function `rdt:redact` function.
- Calling the Server-Side JavaScript function `rdt.redact`.

The mlcp command line tool will provide the highest throughput, but you may find `rdt:redact` or `rdt.redact` convenient when developing and debugging rules.

Regardless of the redaction method you use, you select a set of documents to be redacted and one or more rule collections to apply to those documents.

You should be aware of the following restrictions and guidelines when using redaction:

- You can redact both XML and JSON documents in the same operation.
- You can apply rules defined in XML to JSON documents and vice versa.
- You can only apply redaction rules to XML and JSON documents.
- You cannot redact document metadata such as document properties.
- You cannot rely on the order in which rules are applied. For details, see “No Guaranteed Ordering of Rules” on page 461.
- You must have read permissions for both the documents to be redacted and the redaction rules.
- If you apply a rule that uses a user-defined redaction function, you must have execute permissions for the module that contains the implementation. For details, see “Security Considerations” on page 443.

Your redaction operation will fail if any of the rule collections contain an invalid rule or no rules. You can use the `rdt:rule-validate` XQuery function or the `rdt.ruleValidate` JavaScript function to verify your rule collections before applying them. For details, see "Validating Redaction Rules" on page 461.

24.7.2 Applying Rules Using mlcp

You can apply redaction rules when using the mlcp `export` and `copy` commands. Use the `-redaction` option to specify one or more rule collections to apply to the documents as they are read from the source database. The redaction is performed by MarkLogic on the source host.

The following example command applies the rules in the collections with URIs "pii-rules" and "hipaa-rules" to documents in the database directory "/employees/" on export.

```
# Windows users, see Modifying the Example Commands for Windows
$ mlcp.sh export -host localhost -port 8000 -username user \
    -password password -mode local -output_file_path \
    /example/exported/files -directory_filter /employees/ \
    -redaction "pii-rules,hipaa-rules"
```

The following example applies the same rules during an mlcp copy operation:

```
$ mlcp.sh copy -mode local -input_host srchost -input_port 8000 \
    -input_username user1 -input_password password1 \
    -output_host desthost -output_port 8000 -output_username user2 \
    -output_password password2 -directory_filter /employees/ \
    -redaction "pii-rules,hipaa-rules"
```

For more details, see Redacting Content During Export or Copy Operations in the *mlcp User Guide*.

24.7.3 Applying Rules Using XQuery

Use the `rdt:redact` XQuery library function to create redacted in-memory copies of documents on MarkLogic Server. This function is best suited for testing and debugging your rules or for redacting a small number of documents. To extract large sets of redacted documents from MarkLogic, use the mlcp command line tool instead.

The following example applies the redaction rules in the collections with URIs "pii-rules" and "hipaa-rules" to the documents in the collection "personnel":

```
xquery version "1.0-ml";
import module namespace rdt = "http://marklogic.com/xdmp/redaction"
    at "/MarkLogic/redaction.xqy";
rdt:redact(fn:collection("people"), ("pii-rules","hipaa-rules"))
```

The output is a sequence of document nodes, where each document is the result of applying the rules in the rule collections. The results includes both documents modified by the redaction rules and unmodified documents that did not match any rules or were not changed by the redaction functions.

If any of the rule collections passed to `rdt:redact` is empty, an `RDT-NORULE` exception is thrown. This protects you from accidentally failing to apply any rules, leading to unredacted content.

An exception is also thrown if any of the rule collections contain non-rule documents, if any of the rules are invalid, or if the path expression for a rule selects something other than a node. You can use `rdt:rule-validate` to test the validity of your rules before calling `rdt:redact`.

24.7.4 Applying Rules Using JavaScript

Use the `rdt.redact` JavaScript function to create redacted in-memory copies of documents on MarkLogic Server. This function is best suited for testing and debugging your rules or for redacting a small number of documents. To extract large sets of redacted documents from MarkLogic, use the mlcp command line tool instead.

You must use a `require` statement to bring the redaction functions into scope in your application. These functions are implemented by the XQuery library module `/MarkLogic/redaction.xqy`. For example:

```
const rdt = require('/MarkLogic/redaction');
```

The following example applies the redaction rules in the collections with URIs “pii-rules” and “hipaa-rules” to the documents in the collection “personnel”:

```
const rdt = require('/MarkLogic/redaction');
rdt.redact(fn.collection('people'), ['pii-rules','hipaa-rules'])
```

The output is a `Sequence` of document nodes, where each document is the result of applying the rules in the rule collections. A `Sequence` is an `Iterable`. For example, you can process your results with a for-of loop similar to the following:

```
const rdt = require('/MarkLogic/redaction');
const redacted =
  rdt.redact(fn.collection('people'), ['my-rules']);
for (let doc of redacted) {
  // do something with the redacted document
}
```

The results includes both documents modified by the redaction rules and unmodified documents that did not match any rules or were not changed by the redaction functions.

If any of the rule collections passed to `rdt.redact` is empty, an `RDT-NORULE` exception is thrown. This protects you from accidentally failing to apply any rules, leading to unredacted content. An exception is also thrown if any of the rule collections contain non-rule documents, if any of the rules are invalid, or if the path expression for a rule selects something other than a node.

You can use `rdt.ruleValidate` to test the validity of your rules before calling `rdt.redact`. For details, see “Validating Redaction Rules” on page 461.

24.7.5 No Guaranteed Ordering of Rules

The order in which rules are applied is undefined. You cannot rely on the order in which rules within a rule collection are run, nor on the ordering of rules across multiple rule collections used in the same redaction operation.

In addition, the final redacted result for a given reflects the result of at most one rule. If you have multiple rules that select the same node, they will all run, but the final document produced by redaction reflects the result of at most one of these rules.

Therefore, you should not have multiple rules in the same redaction operation that redact or examine the same nodes.

For example, suppose you have two rule collections, A and B, with the following characteristics:

```
Collection A contains:
  ruleA1 using path //id
  ruleA2 using path //id
Collection B contains:
  ruleB1 using path //id
```

If you apply both rule collections to a set of documents, you cannot know or rely on the order in which `ruleA1`, `ruleA2`, and `ruleB1` are applied to any selected id node. In addition, the output only reflect the changes to `//id` made by one of `ruleA1`, `ruleA2`, and `ruleB1`.

24.8 Validating Redaction Rules

You can use the `rdt:rule-validate` XQuery function or the `rdt.ruleValidate` Server-Side JavaScript function to test your rule collections for validity before using them. You should validate your rules before deploying them to production because an invalid rule or an empty rule collection will cause a redaction operation to fail.

Validation confirms that your rule(s) and rule collection(s) conforms to the expected structure and does not rely on any non-existent code, such as an undefined redaction function.

Note that a successfully validated rule can still cause runtime errors. For example, rule validation does not include dictionary validation if your rule uses dictionary-based masking. Similarly, validation does not verify that the XPath expression in a rule conforms to the limitations described in “Limitations on XPath Expressions in Redaction Rules” on page 449.

If all the rules in the input rule collections are valid, the validation function returns the URIs of all validated rules. Otherwise, an exception is thrown when the first validation error is encountered.

The following example validates the rules in two rule collections with URIs “pii-rules” and “hipaa-rules”.

Language	Example

XQuery	`xquery version "1.0-ml";`
	`import module namespace rdt =`
	`  "http://marklogic.com/xdmp/redaction"`
	`  at "/MarkLogic/redaction.xqy";`
	`rdt:rule-validate(("pii-rules", "hipaa-rules"))`

JavaScript	`const rdt = require('/MarkLogic/redaction.xqy');` `rdt.ruleValidate(["pii-rules", "hipaa-rules"])`

24.9 Built-in Redaction Function Reference

MarkLogic provides several built-in redaction functions for use in your redaction rules. To use one of these functions, create a rule with a `method` child XML element or JSON property of the following form.

XML	JSON

`<method>`
`  <function>`*`builtInName`*`</function>`
`</method>`
	`"method": {`
	`  "function": "`*`builtInFuncName`*`"`
	`}`

If the built-in accepts configuration parameters, specify them in the `options` child XML element or JSON property of the rule. For syntax, see “Defining Redaction Rules” on page 445. For parameter specifics and examples, see the reference section for each built-in.

The following table summarizes the built-in redaction functions and expected input parameters. Refer to the section on each function for more details and examples.

Function Name	Description
mask-deterministic	Replace values with masking text that is deterministic. That is, a given input generates the same mask value every time it is applied. You can control features such as the length and type of the generated value.
mask-random	Replace values with random text. The masking value can vary across repeated application to the same input value. You can control the length of the generated value and type of replacement text (numbers or letters).
conceal	Remove the value to be masked.
redact-number	Replace values with random numbers. You can control the data type, range, and format of the masking values.
redact-us-ssn	Redact data that matches the pattern of a US Social Security Number (SSN). You can control whether or not to preserve the last 4 digits and what character to use as a masking character.
redact-us-phone	Redact data that matches the pattern of a US telephone number. You can control whether or not to preserve the last 4 digits and what character to use as a masking character.
redact-email	Redact data that matches the pattern of an email address. You can control whether to mask the entire address, only the username, or only the domain name.
redact-ipv4	Redact data that matches the pattern of an IPv4 address. You can control what character to use as a masking character.
redact-datetime	Redact data that matches the pattern of a dateTime value. You can control the expected input format and the masking dateTime format.
redact-regex	Redact data that matches a given regular expression. You must specify the regular expression and the masking text.

For a complete example of using all the built-in functions, see “Example: Using the Built-In Redaction Functions” on page 487.

24.9.1 mask-deterministic

Use this built-in to mask a value with a consistent masked value. That is, with deterministic masking, a given input always produces the same output. The original value is not derivable from the masked value.

Deterministic masking can be useful for preserving relationships across records. For example, you could mask the names in a social network, yet still be able to trace relationships between people (X knows Y, and Z knows Y).

Use the following parameters to configure the behavior of this function. Set parameters in the `options` section of a rule.

- `length`: The length, in characters, of the output value to generate. Optional. Default: 64. You cannot use this option with the `dictionary` option.
- `character`: The class of character(s) to use when constructing the masked value. Allowed values: `any` (default), `alphanumeric`, `numeric`, `alphabetic`. You cannot use this option with the `dictionary` option.
- `dictionary`: The URI of a redaction dictionary. Use the dictionary as the source of replacement values. You cannot use this option with any other options.
- `salt`: A salt to apply when generating masking values. MarkLogic applies the salt even when drawing replacement values from a dictionary. The default behavior is no salt.
- `extend-salt`: Whether/how to extend the salt with runtime information. You can extend the salt with the rule set collection name or the cluster id. Allowed values: `none`, `collection`, `cluster-id` (default).

When you use dictionary-based masking, a given input will always map to the same redaction dictionary entry. If you modify the dictionary, then the dictionary mapping will also change.

The `salt` and `extend-salt` options options introduce rule and/or cluster-specific randomness to the generated masking values. Each masking value is still deterministic when salted: The same input produces the same output. However, the same input with different salts produces different output. For details, see “Salting Masking Values for Added Security” on page 523.

The following example rule applies deterministic masking to nodes selected by the XPath expression "//name". The replacement value will be 10 characters long because of the `length` option.

XML	JSON

<rule xml:lang="zxx"
xmlns="http://marklogic.com/xdmp/redaction">
<path>//name</path>
<method>
<function>
mask-deterministic
</function>
</method>
<options>
<length>10</length>
</options>
</rule>
	{"rule": {
	"path": "//name",
	"method": {
	"function": "mask-deterministic",
	},
	"options": {
	"length": 10
	}
	} }

The following table illustrates the effect of applying `mask-deterministic` to several different types of nodes. For an end-to-end example, see "Example: Using the Built-In Redaction Functions" on page 487.

Path Expr	Fmt	Original Document	Redacted Result
//name Simple atomic value	XML	<person> <name>Little Bopeep</name> </person>	<person> <name>8d1f713a30</name> </person>

	JSON	{
		"name": "Georgie Porgie"
		}
			{
			"name": "34fe55c66a"
			}

//alias Multiple items (array in JSON)	XML	<person> <alias>Peepers</alias> <alias>Bo</alias> </person>	<person> <alias>7a4fabd518</alias> <alias>850517542f</alias> </person>
	JSON	{ "alias": ["George", "GP"] }	{ "alias": ["ef36ccc0c8", "fa6f1defad"] }

Path Expr	Fmt	Original Document	Redacted Result

//address
 	
Complex value
	XML	<person>
		  <address>
		  <street>
		  100 Nursery Lane
		  </street>
		  <city>Hometown</city>
		  <country>
		  Neverland
		  </country>
		  </address>
		</person>
			<person>
			  <address>
			  8d1f713a30
			  </address>
			</person>

	JSON	{"address": {
		  "street": "300 Nursery Lane",
		  "city": "Hometown",
		  "country": "Neverland"
		}}
			{ "address": "fc1f5fcb6d"}

In most cases, the entire value of the node is replaced by the redacted value, even if the original contents are complex, such as the `//address` example, above.

However, notice the `//alias` example above, which selects individual `alias` array items in the JSON example, rather than the entire array. If you want to redact the entire array value, you need a rule with a JSON-specific path selector. For example, a rule path such as `//array-node('alias')` selects the entire array in the JSON documents, resulting in a value such as the following for the "alias" property:

```
"alias": "6b162c290e"
```

For more details, see "Defining Rules Usable on Multiple Document Formats" on page 450.

To illustrate the effects of the various `character` option settings, assume a `length` option of 10 and the following input targeted for redaction:

```
<pii>
  <priv>redact me</priv>
  <priv>redact me</priv>
  <priv>redact me too</priv>
</pii>
```

Then the following table shows the result of applying each possible value of the `character` option.

`character` Setting	Redacted Value

`any` (default)	`<pii>`
	`  <priv>3ba1a188e6</priv>`
	`  <priv>3ba1a188e6</priv>`
	`  <priv>a62597fd0c</priv>`
	`</pii>`

`alphanumeric`	`<pii>`
	`  <priv>F1Fp64Cnox</priv>`
	`  <priv>F1Fp64Cnox</priv>`
	`  <priv>LiN5mrmG0g</priv>`
	`</pii>`

`numeric`	`<pii>`
	`  <priv>1838664450</priv>`
	`  <priv>1838664450</priv>`
	`  <priv>5771438029</priv>`
	`</pii>`

`alphabetic`	`<pii>`
	`  <priv>PQXWBHfASy</priv>`
	`  <priv>PQXWBHfASy</priv>`
	`  <priv>ZroFQNkNqi</priv>`
	`</pii>`

24.9.2 mask-random

Use this built-in to replace a value with a random masking value. A given input produces different output each time it is applied. The original value is not derivable from the masked value. Random masking can be useful for obscuring relationships across records.

Use the following parameters to configure the behavior of this function. Set parameters in the `options` section of a rule.

- `length`: The length, in characters, of the output value to generate. Optional. Default: 64. You cannot use this `option` with the dictionary option.
- `character`: The type of character(s) to use when constructing the masked value. Allowed values: `any` (default), `alphanumeric`, `numeric`, `alphabetic`. You cannot use this option with the `dictionary` option.
- `dictionary`: The URI of a redaction dictionary. Use the dictionary as the source of replacement values. You cannot use this option with any other options.

The following example rule applies random masking to nodes selected by the XPath expression "//name". The replacement value will be 10 characters long because of the `length` option.

XML	JSON

<rule xml:lang="zxx"
xmlns="http://marklogic.com/xdmp/redaction">
<path>//name</path>
<method>
<function>
mask-random
</function>
</method>
<options>
<length>10</length>
</options>
</rule>
	{"rule": {
	"path": "//name",
	"method": {
	"function": "mask-random",
	},
	"options": {
	"length": 10
	}
	} }

The following table illustrates the effect of applying `mask-random` to several different types of nodes. For an end-to-end example, see "Example: Using the Built-In Redaction Functions" on page 487.

Path Expr	Fmt	Original Document	Redacted Result
//name Simple atomic value	XML	<person> <name>Little Bopeep</name> </person>	<person> <name>8d1f713a30</name> </person>

	JSON	{
		"name": "Georgie Porgie"
		}
			{
			"name": "34fe55c66a"
			}

//alias Multiple items (array in JSON)	XML	<person> <alias>Peepers</alias> <alias>Bo</alias> </person>	<person> <alias>7a4fabd518</alias> <alias>850517542f</alias> </person>
	JSON	{ "alias": ["George", "GP"] }	{ "alias": ["ef36ccc0c8", "fa6f1defad"] }

Path Expr	Fmt	Original Document	Redacted Result

//address
 	
Complex value
	XML	<person>
		  <address>
		  <street>
		  100 Nursery Lane
		  </street>
		  <city>Hometown</city>
		  <country>
		  Neverland
		  </country>
		  </address>
		</person>
			<person>
			  <address>
			  8d1f713a30
			  </address>
			</person>

	JSON	{"address": {
		  "street": "300 Nursery Lane",
		  "city": "Hometown",
		  "country": "Neverland"
		}}
			{ "address": "fc1f5fcb6d"}

In most cases, the entire value of the node is replaced by the redacted value, even if the original contents are complex, such as the `//address` example, above.

However, notice the `//alias` example above, which selects individual `alias` array items in the JSON example, rather than the entire array. If you want to redact the entire array value, you need a rule with a JSON-specific path selector. For example, a rule path such as `//array-node('alias')` selects the entire array in the JSON documents, resulting in a value such as the following for the "alias" property:

```
"alias": "6b162c290e"
```

For more details, see "Defining Rules Usable on Multiple Document Formats" on page 450.

To illustrate the effects of the various `character` option settings, assume a `length` option of 10 and the following input targeted for redaction:

```
<pii>
  <priv>redact me</priv>
  <priv>redact me</priv>
  <priv>redact me too</priv>
</pii>
```

Then the following table shows the result of applying each possible value of the `character` option.

`character` Setting	Redacted Value

`any` (default)	`<pii>`
	`  <priv>2457f4f294</priv>`
	`  <priv>f18e883ba9</priv>`
	`  <priv>e5b253aea9</priv>`
	`</pii>`

`alphanumeric`	`<pii>`
	`  <priv>qIEsmeJua6</priv>`
	`  <priv>WfVLAAckzu</priv>`
	`  <priv>P8BGgCdt5s</priv>`
	`</pii>`

`numeric`	`<pii>`
	`  <priv>7902282158</priv>`
	`  <priv>8313199931</priv>`
	`  <priv>2026296703</priv>`
	`</pii>`

`alphabetic`	`<pii>`
	`  <priv>rZimfgZwSG</priv>`
	`  <priv>knqbTrKTdl</priv>`
	`  <priv>wKYeTkVjLC</priv>`
	`</pii>`

24.9.3 conceal

Use this built-in to entirely remove a selected value.

The following example rule applies concealment to values selected by the path expression `//name`.

XML	JSON

`<rule xml:lang="zxx"`
`  xmlns="http://marklogic.com/xdmp/redaction">`
`  <path>//name</path>`
`  <method>`
`  <function>conceal</function>`
`  </method>`
`</rule>`
	`{"rule": {`
	`  "path": "//name",`
	`  "method": {`
	`  "function": "conceal",`
	`  }`
	`} }`

The following table illustrates the effect of applying `conceal` to several different types of nodes. For an end-to-end example, see "Example: Using the Built-In Redaction Functions" on page 487.

Path Expr	Fmt	Original Document	Redacted Result

`//name`
 	
Simple atomic value
	XML	`<person>`
		`  <name>`
		`  Little Bopeep`
		`  </name>`
		`  <id>12-3456789</id>`
		`</person>`
			`<person>`
			`  <id>12-3456789</id>`
			`</person>`

	JSON	`{ "name":"Jack Sprat",`
		`  "id": "45-6789123"`
		`}`
			`{ "id": "45-6789123" }`

`//alias`
 	
Multiple items (array in JSON)
	XML	`<person>`
		`  <alias>Peepers</alias>`
		`  <alias>Bo</alias>`
		`  <id>12-3456789</id>`
		`</person>`
			`<person>`
			`  <id>12-3456789</id>`
			`</person>`

	JSON	`{ "alias": [`
		`  "George",`
		`  "G.P."`
		`  ],`
		`  "id": "45-6789123"`
		`}`
			`{ "alias": [],`
			`  "id": "45-6789123"`
			`}`

`//address`
 	
Complex value
	XML	`<person>`
		`  <address>`
		`  <street>`
		`  100 Nursery Lane`
		`  </street>`
		`  <city>Hometown</city>`
		`  <country>`
		`  Neverland`
		`  </country>`
		`  </address>`
		`  <id>12-3456789</id>`
		`</person>`
			`<person>`
			`  <id>12-3456789</id>`
			`</person>`

	JSON	`{"address": {`
		`  "street": "300 Nursery Lane",`
		`  "city": "Hometown",`
		`  "country": "Neverland"`
		`  },`
		`  "id": "45-6789123"`
		`}`
			`{ "id": "45-6789123" }`

In most cases, the entire selected node is concealed, even if the original contents are complex, such as the `//address` example, above.

However, note that a path such as `//alias`, above, conceals each array item in the JSON sample, rather than concealing the entire array. This is because the `alias` path step matches each array item individually; for details, see "Defining Rules Usable on Multiple Document Formats" on page 450 and "Traversing JSON Documents Using XPath" on page 357.

If you want to redact the entire array value, you need a rule with a JSON-specific path selector, such as `//array-node('alias')`. For more details, see "Defining Rules Usable on Multiple Document Formats" on page 450.

24.9.4 redact-number

Use this built-in to mask values with a random number that conforms to a configurable range and format.

This function differs from the `mask-random` function in that it provides finer control over the masking value. Also, `mask-random` always generates a text node, while `redact-number` generates either a number node or a text node, depending on the configuration.

The `redact-number` function enables you to control the following aspects of the masking value:

- Constrain the value to a range by specifying a min and/or max value.
- Constrain the value to a specific numeric type (integer, decimal, or double).
- Specify a format for the value using a "picture string". For example, limit the number of digits after the decimal point or include a currency symbol such as a dollar sign.

Use the following options to configure the behavior of this function:

- `min`: The minimum acceptable masking value, inclusive. This function will not generate a masking value less than the `min` value. Optional. Default: 0.
- `max`: The maximum acceptable masking value, inclusive. This function will not generate a masking value greater than the `max` value. Optional. Default: 18446744073709551615.
- `format`: Special formatting to apply to the replacement value. Optional. Default: No special formatting. The format string must conform to the syntax for an XSLT "picture string", as described in the function reference for `fn:format-number` (XQuery) or `fn.formatNumber` (JavaScript) and in https://www.w3.org/TR/xslt20/#function-format-number. If you specify a format, the replacement value is a text node in JSON documents instead of a number node. Note: If you specify a format, then the values in the range defined by `min` and `max` must be convertible to decimal.
- `type`: The data type of the replacement value. Optional. Allowed values: `integer`, `decimal`, `double`. Default: `integer`. The values specified in the `min` and `max` options are subject to the specified type restriction.

The following example rule applies `redact-number` to values selected by the XPath expression `//balance`. The matched values will be replaced by decimal values in the range 0.0 to 100000.00, with two digits after the decimal point. The rule generates replacement values such as 3.55, 19.79, 82.96.

XML	JSON

`<rdt:rule xml:lang="zxx"`
`  xmlns:rdt="http://marklogic.com/xdmp/redaction">`
`  <rdt:path>//balance</rdt:path>`
`  <rdt:method>`
`  <rdt:function>redact-number</rdt:function>`
`  </rdt:method>`
`  <rdt:options>`
`  <min>1</min>`
`  <max>100000</max>`
`  <format>0.00</format>`
`  <type>decimal</type>`
`  </rdt:options>`
`</rdt:rule>`
	`{"rule": {`
	`  "path": "//balance",`
	`  "method": {`
	`  "function": "redact-number",`
	`  },`
	`  "options": {`
	`  "min": 1,`
	`  "max": 100000,`
	`  "format": "0.00",`
	`  "type": "decimal`
	`  }`
	`} }`

When applied to a JSON document, the node replaced by redaction can be either a text node or a number node, depending on whether or not you use the `format` option. With no explicit formatting, redaction produces a number node for JSON. With explicit formatting, redaction produces a text node. For example, `redact-number` might affect the value of a JSON property named “key” as follows:

```
no format option
"key": 61.4121623617221

format option value "0.00"
"key": "61.41"
```

The value range defined by a `redact-number` rule must be valid for the data type. For example, the following set of options is invalid because the specified range does not express a meaningful integer range from which to generate values:

```
min: 0.1
max: 0.9
type: integer
```

The values of `min` and `max` must be castable to the specified type.

The following table illustrates the effect of applying `redact-number` with various option combinations. For an end-to-end example, see "Example: Using the Built-In Redaction Functions" on page 487.

Option Configuration	Fmt	Example Redaction Result
default (no options)	XML	<balance>8137497966986464072</balance> <balance>2363247638359197582</balance>
	JSON	"balance": 8137497966986464072 "balance": 2363247638359197582
min: 100 max: 10000	XML	<balance>3842</balance> <balance>6622</balance>
	JSON	"balance": 3842 "balance": 6622
min: 100 max: 10000 type: decimal	XML	<balance>100.82</balance> <balance>269.419736229</balance>
	JSON	"balance": 100.82 "balance": 269.419736229
min: 100 max: 10000 type: decimal format: 0.00	XML	<balance>102.77</balance> <balance>9596.90</balance>
	JSON	"balance": "102.77" "balance": "9596.90" Note that masking values are text nodes due to the use of the format option.

24.9.5 redact-us-ssn

Use this built-in to mask values that conform to one of the following patterns. These patterns correspond to typical representations for US Social Security Numbers (SSNs). The character *N* in these patterns represents a single digit in the range 0 - 9.

- *NNN-NN-NNNN* (dash separator)
- *NNN.NN.NNNN* (dot separator)
- *NNN NN NNNN* (space separator)
- *NNNNNNNNN*

When a pattern match is found, every redacted digit is replaced with the same character. For example, a value such as "123-45-6789" might become "XXX-XX-XXXX", depending on the rule configuration.

You can use the following parameters to configure the behavior of this function. Set parameters in the `options` section of a rule.

- `level`: How much to redact. Optional. This option can have the following values:
  - `full`: Default. Replace all digits with the character specified by the `character` option.
  - `partial`: Retain the last 4 digits; replace all other digits with the `character` specified by the character option.
  - `full-random`: Replace all digits with random digits. The `character` option is ignored. You will get a different value each time you redact a given value.
- `character`: The character with which to replace each redacted digit when `level` is `full` or `partial`. Optional. Default: "#".

The following example redacts SSNs selected by the path expression `//id`. The parameters specify that last 4 digits of the SSN are preserved and the remaining digits are replaced with the character "X".

XML	JSON

`<rule xml:lang="zxx"`
`  xmlns="http://marklogic.com/xdmp/redaction">`
`  <path>//id</path>`
`  <method>`
`  <function>redact-us-ssn</function>`
`  </method>`
`  <options>`
`  <level>partial</level>`
`  <character>X</character>`
`  </options>`
`</rule>`
	`{"rule": {`
	`  "path": "//id",`
	`  "method": {`
	`  "function": "redact-us-ssn",`
	`  },`
	`  "options": {`
	`  "level": "partial",`
	`  "character": "X"`
	`  }`
	`} }`

The following table illustrates the effect of applying `redact-us-ssn` with various input values and configuration parameters. For a complete example, see "Example: Using the Built-In Redaction Functions" on page 487.

Configuration	Fmt	Original Document	Redacted Result
Path: //ssn Level: full Char: # (default)	XML	<pii> <ssn>123-45-6789</ssn> <ssn>123.45.6789</ssn> <ssn>123456789</ssn> </pii>	<pii> <ssn>###-##-####</ssn> <ssn>###.##.####</ssn> <ssn>#########</ssn> </pii>
	JSON	{"pii": { ssn: ["123-45-6789", "123.45.6789", "123456789"] } }	{"pii": { ssn: ["###-##-####", "###.##.####", "#########"] } }
Path: //ssn Level: partial	XML	<pii> <ssn>123-45-6789</ssn> <ssn>123.45.6789</ssn> <ssn>123456789</ssn> </pii>	<pii> <ssn>###-##-6789</ssn> <ssn>###.##.6789</ssn> <ssn>#####6789</ssn> </pii>
	JSON	{"pii": { ssn: ["123-45-6789", "123.45.6789", "123456789"] } }	{"pii": { ssn: ["###-##-6789", "###.##.6789", "#####6789"] } }
Path: //ssn Level: full-random	XML	<pii> <ssn>123-45-6789</ssn> <ssn>123.45.6789</ssn> <ssn>123456789</ssn> </pii>	<pii> <ssn>492-54-3352</ssn> <ssn>441.65.4885</ssn> <ssn>501965954</ssn> </pii>
	JSON	{"pii": { ssn: ["123-45-6789", "123.45.6789", "123456789"] } }	{"pii": { ssn: ["492-54-3352", "441.65.4885", "501965954"] } }

Configuration	Fmt	Original Document	Redacted Result
Path: //ssn Level: full Character: X	XML	<pii> <ssn>123-45-6789</ssn> <ssn>123.45.6789</ssn> <ssn>123456789</ssn> </pii>	<pii> <ssn>XXX-XX-XXXX</ssn> <ssn>XXX.XX.XXXX</ssn> <ssn>XXXXXXXXX</ssn> </pii>

	JSON	{"pii": {
		  ssn: [
		  "123-45-6789",
		  "123.45.6789",
		  "123456789"
		  ]
		} }
			{"pii": {
			  ssn: [
			  "XXX-XX-XXXX",
			  "XXX.XX.XXXX",
			  "XXX.XX.XXXX"
			  ]
			} }

24.9.6 redact-us-phone

Use this built-in to mask values that conform to one of the following patterns. These patterns correspond to typical representations for US telephone numbers. The character *N* in these patterns represents a single digit in the range 0 - 9.

- *NNN-NNN-NNNN* ("-" separator)
- *NNN.NNN.NNNN* (". " separator)
- (*NNN*)*NNN-NNNN* (no whitespace allowed)
- *NNNNNNNNNN*

When a pattern match is found, every redacted digit is replaced with the same character. For example, a value such as "123-456-7890" might become "XXX-XXX-XXXX", depending on the configuration of the rule.

You can use the following parameters to configure the behavior of this function. Set parameters in the `options` section of a rule.

- `level`: How much to redact. Optional. This option can have the following values:
  - `full`: Default. Replace all digits with the character specified by the `character` option.
  - `partial`: Retain the last 4 digits; replace all other digits with the `character` specified by the character option.
  - `full-random`: Replace all digits with random digits. The `character` option is ignored. You will get a different random value each time you redact a given input.
- `character`: The character with which to replace each redacted digit when `level` is `full` or `partial`. Optional. Default: "#".

The following example masks telephone numbers selected by the path expression `//ph`. The parameters specify that last 4 digits of the telephone number are preserved and the remaining digits are replaced with the character "X".

XML	JSON

<rule xml:lang="zxx"
xmlns="http://marklogic.com/xdmp/redaction">
<path>//ph</path>
<method>
<function>redact-us-phone</function>
</method>
<options>
<level>partial</level>
<character>X</character>
</options>
</rule>
	{"rule": {
	"path": "//ph",
	"method": {
	"function": "redact-us-phone",
	},
	"options": {
	"level": "partial",
	"character": "X"
	}
	} }

The following table illustrates the effect of applying `redact-us-phone` with various input values and configuration parameters. For a complete example, see "Example: Using the Built-In Redaction Functions" on page 487.

Configuration	Fmt	Original Document	Redacted Result
Path: //ph Level: full Char: # (default)	XML	<pii> <ph>123-456-7890</ph> <ph>123.456.7890</ph> <ph>(123)456-7890</ph> <ph>1234567890</ph> </pii>	<pii> <ph>###-###-####</ph> <ph>###.###.####</ph> <ph>(###)###-####</ph> <ph>##########</ph> </pii>
	JSON	{"pii": { "ph": ["**123-456-789**0", "123.456.7890", "(123)456-7890", "1234567890"] } }	{"pii": { "ph": ["###-###-####", "###.###.####", "(###)###-####", "##########"] } }

Configuration	Fmt	Original Document	Redacted Result
Path: //ph Level: partial Char: #	XML	<pii> <ph>123-456-7890</ph> <ph>123.456.7890</ph> <ph>(123)456-7890</ph> <ph>1234567890</ph> </pii>	<pii> <ph>###-###-7890</ph> <ph>###.###.7890</ph> <ph>(###)###-7890</ph> <ph>######7890</ph> </pii>
	JSON	{"pii": { "ph": ["123-456-7890", "123.456.7890", "(123)456-7890", "1234567890"] } }	{"pii": { "ph": ["###-###-7890", "###.###.7890", "(###)###-7890", "######7890"] } }
Path: //ph Level: full-random Char: #	XML	<pii> <ph>123-456-7890</ph> <ph>123.456.7890</ph> <ph>(123)456-7890</ph> <ph>1234567890</ph> </pii>	<pii> <ph>291-826-5242</ph> <ph>121.350.3951</ph> <ph>(804)380-8192</ph> <ph>9644991161</ph> </pii>
	JSON	{"pii": { "ph": ["123-456-7890", "123.456.7890", "(123)456-7890", "1234567890"] } }	{"pii": { "ph": ["291-826-5242", "121.350.3951", "(804)380-8192", "9644991161"] } }
Path: //ph Level: full Character: X	XML	<pii> <ph>123-456-7890</ph> <ph>123.456.7890</ph> <ph>(123)456-7890</ph> <ph>1234567890</ph> </pii>	<pii> <ph>XXX-XXX-XXXX</ph> <ph>XXX.XXX.XXXX</ph> <ph>(XXX)XXX-XXXX</ph> <ph>XXXXXXXXXX</ph> </pii>
	JSON	{"pii": { "ph": ["**123-456-789**0", "123.456.7890", "(123)456-7890", "1234567890"] } }	{"pii": { "ph": ["XXX-XXX-XXXX", "XXX.XXX.XXXX", "(XXX)XXX-XXXX", "XXXXXXXXXX"] } }

24.9.7 redact-email

Use this built-in to mask values that conform to the pattern of an email address. The function assumes an email has the form `name@domain`.

Use the following parameters to configure the behavior of this function. Set parameters in the `options` section of a rule.

- `level`: How much of each email address to redact. Allowed values: `full`, `name`, `domain`. Optional. Default: `full`.

Redacting the username portion of an email address replaces the username with "NAME". Redacting the domain portion of an email address replaces the domain name with "DOMAIN". Thus, full redaction on the email address "jsmith@example.com" produces the replacement value "NAME@DOMAIN".

The following example rule fully redacts email addresses selected by the path expression "//email".

XML	JSON

<rule xml:lang="zxx"
xmlns="http://marklogic.com/xdmp/redaction">
<path>//email</path>
<method>
<function>redact-email</function>
</method>
<options>
<level>full</level>
</options>
</rule>
	{"rule": {
	"path": "//email",
	"method": {
	"function": "redact-email",
	},
	"options": {
	"level": "full"
	}
	} }

The following table illustrates the effect of applying `redact-email` with various levels of redaction. For a complete example, see "Example: Using the Built-In Redaction Functions" on page 487.

Configuration	Fmt	Original Document	Redacted Result

`//email`
 	
`level: full (default)`
	XML	`<person>`
		`  <email>`
		`  bopeep@mothergoose.com`
		`  </email>`
		`</person>`
			`<person>`
			`  <email>`
			`  NAME@DOMAIN`
			`  </email>`
			`</person>`

	JSON	`{"email":`
		`  "gp@mothergoose.com"`
		`}`
			`{"email": "NAME@DOMAIN" }`

`//email`
 	
`level: name`
	XML	`<person>`
		`  <email>`
		`  bopeep@mothergoose.com`
		`  </email>`
		`</person>`
			`<person>`
			`  <email>`
			`  NAME@mothergoose.com`
			`  </email>`
			`</person>`

	JSON	`{"email":`
		`  "gp@mothergoose.com"`
		`}`
			`{"email":`
			`  "NAME@mothergoose.com"`
			`}`

`//email`
 	
`level: domain`
	XML	`<person>`
		`  <email>`
		`  bopeep@mothergoose.com`
		`  </email>`
		`</person>`
			`<person>`
			`  <email>`
			`  bopeep@DOMAIN`
			`  </email>`
			`</person>`

	JSON	`{"email":`
		`  "gp@mothergoose.com"`
		`} }`
			`{"email": "gp@DOMAIN"}`

24.9.8 redact-ipv4

Use this built-in to mask values that conform to the pattern of an IP address. This function only redacts IPv4 addresses. That is, a value is redacted if it conforms to the following pattern, where *N* represents a decimal digit (0-9).

- Four blocks of 1-3 decimal digits, separated by period ("."). The value of each block of digits must less than or equal to 255. For example: `123.201.098.112`, `123.45.678.0`.

The redacted IP address is normalized to contain characters for the maximum number of digits. That is, an IP address such as 123.4.56.7 is masked as "###.###.###.###".

Use the following options to configure the behavior of this function. Set parameters in the `options` section of a rule.

- `character`: The character with which to replace each redacted digit. Optional. Default: "#".

The following example rule redacts IP addresses selected by the path expression `//ip`. The `character` parameter specifies the digits of the redacted IP address are replaced with "X".

XML	JSON

`<rule xml:lang="zxx"`
`  xmlns="http://marklogic.com/xdmp/redaction">`
`  <path>//ip</path>`
`  <method>`
`  <function>redact-ipv4</function>`
`  </method>`
`  <options>`
`  <character>X</character>`
`  </options>`
`</rule>`
	`{"rule": {`
	`  "path": "//ip",`
	`  "method": {`
	`  "function": "redact-ipv4",`
	`  },`
	`  "options": {`
	`  "character": "X"`
	`  }`
	`} }`

The following table illustrates the effect of applying `redact-ipv4` with various configuration options. For a complete example, see "Example: Using the Built-In Redaction Functions" on page 487.

Configuration	Fmt	Original Document	Redacted Result

`//ip`
 	
`default`
	`XML`	`<person>`
		`  <ip>123.45.6.78</ip>`
		`</person>`
			`<person>`
			`  <ip>###.###.###.###</ip>`
			`</person>`

	`JSON`	`{"ip": "123.45.6.78"}`	`{"ip": "###.###.###.###"}`

`//ip`
 	
`character: X`
	`XML`	`<person>`
		`  <ip>123.45.6.78</ip>`
		`  <ip>123.145.167.189</ip>`
		`</person>`
			`<person>`
			`  <ip>`**`XXX.XXX.XXX.XXX`**`</ip>`
			`  <ip>`**`XXX.XXX.XXX.XXX`**`</ip>`
			`</person>`

	`JSON`	`{"ip": [`
		`  "123.45.6.78",`
		`  "123.145.167.189"`
		`]}`
			`{"ip": [`
			`  "`**`XXX.XXX.XXX.XXX`**`",`
			`  "`**`XXX.XXX.XXX.XXX`**`"`
			`]}`

24.9.9 redact-datetime

Use this built-in to mask values that represent a dateTime value. You can use this function to mask dateTime value in one of two ways:

- Parse the input dateTime value and replace it with a masking value derived from applying a dateTime picture string to the input dateTime components. For example, redact the value "2012-05-23" by obscuring the month and date, producing a masking value such as "2012-MM-DD". You can only use this type of dateTime redaction to redact values that can be parsed by `fn:parse-dateTime` or `fn.parseDateTime`.
- Replace any value with a random dateTime value, formatted according to a specified picture string. You can restrict the value to a particular year range.

You can use the following parameters to configure the behavior of this function. Set parameters in the `options` section of a rule.

- `level`: The type of dateTime redaction. Required. Allowed values: `parsed`, `random`.
- `format`: A dateTime picture string describing how to format the masking value. Required.
- `picture`: A dateTime picture string describing the required input value format. This option is required when `level` is `parsed` and ignored otherwise. Any input value that does not conform to the expected format is not redacted.
- `range`: A comma separated pair of years, used to constrain the masking value range when `level` is `random`. Optional. This option is ignored if `level` is not `random`. For example, a `range` value of "1900,1999" will only generate masking values for the years 1900 through 1999, inclusive.

  Note: When you apply `redact-datetime` with a `picture` option, the content selected by your rule path must serialize to text whose leading characters conform to the picture string. If there are other leading characters in the serialized content, redaction fails with an error.

The following example rule redacts dateTime values using the parsed method. The `picture` option specifies that only input values of the form YYYY-MM-DD are redacted. The `format` option specifies that the masking value is of the form MM-DD-YYYY, with the day portion replaced by the literal value "NN".

XML	JSON

<rule xml:lang="zxx"
xmlns="http://marklogic.com/xdmp/redaction">
<path>//deathdate</path>
<method>
<function>redact-datetime</function>
</method>
<options>
<level>parsed</level>
<picture>[Y0001]-[M01]-[D01]</picture>
<format>NN-NN-[Y0001]</format>
</options>
</rule>
	{"rule": {
	"path": "//deathdate",
	"method": {
	"function": "redact-datetime",
	},
	"options": {
	"level": "parsed"
	"picture":"[Y0001]-[M01]-[D01]",
	"format": "NN-NN-[Y0001]"
	}
	} }

If you apply the above rules to a value such as "2012-11-09", the redacted value becomes "NN-NN-2012".

The following example rule redacts values using the random method. The format option specifies that the masking value be of the form YYYY-MM-DD, and that the masking values be in the year range 1900 to 1999, inclusive. The format of the value to be redacted does not matter.

XML	JSON

<rule xml:lang="zxx"
xmlns="http://marklogic.com/xdmp/redaction">
<path>//deathdate</path>
<method>
<function>redact-datetime</function>
</method>
<options>
<level>random</level>
<format>[Y0001]-[M01]-[D01]</format>
<range>1900,1999</format>
</options>
</rule>
	{"rule": {
	"path": "//deathdate",
	"method": {
	"function": "redact-datetime",
	},
	"options": {
	"level": "random"
	"format":"[Y0001]-[M01]-[D01]",
	"range": "1900,1999"
	}
	} }

For a complete example, see "Example: Using the Built-In Redaction Functions" on page 487.

24.9.10 redact-regex

Use this built-in to mask values that match a regular expression. The regular expression and the replacement text are configurable.

Use the following options to configure the behavior of this function:

- `pattern`: A regular expression identifying the values to be redacted. Required. Use the regular expression language syntax defined for XQuery and XPath. For details, see http://www.w3.org/TR/xpath-functions/%23regex-syntax.
- `replacement`: The text with which to replace values matching `pattern`.

The pattern and replacement text are applied to the input values as if by calling the `fn:replace` XQuery function or the `fn.replace` Server-Side JavaScript function.

Note that the replacement pattern can contain back references to portions of the matched text. A back reference enables you to "capture" portions of the matched text and re-use them in the replacement value. See the example at the end of this section.

Regular expression patterns can contain characters that require escaping in your rule definitions. The following contains a few examples of problem characters. This is not an exhaustive list.

- Curly braces ("{ }") in pattern in an XML rule installed with XQuery must be escaped as "{{" and "}}" to prevent the XQuery interpreter from treating them as code block delimiters.
- A left angle bracket ("<") in an XML rule must be replaced by the entity reference "<".
- Backslashes ("\") in a JSON rule definition must be escaped as "\\" because "\" is a special character in JSON strings.

The following example redacts text which has one of the following forms, where *N* represents a single digit in the range 0-9.

- *NN-NNNNNNN* (dash separator)
- *NN.NNNNNNN* (dot separator)
- *NN NNNNNNN* (space separator)
- *NNNNNNN*

The following regular expression matches the supported forms:

```
\d{2}[-.\s]\d{7}
```

The following rule specifies that values in an `id` XML element or JSON property that match the pattern should be replaced with the text "NN-NNNNNNN". Notice the escaped characters in the pattern.

XML	JSON

<rule xml:lang="zxx"
xmlns="http://marklogic.com/xdmp/redaction">
<path>//id</path>
<method>
<function>redact-regex</function>
</method>
<options>
<pattern>
\d{{2}}[-.\s]\d{{7}}
</pattern>
<replacement>NN-NNNNNNN</replacement>
</options>
</rule>
	{"rule": {
	"path": "//id",
	"method": {
	"function": "redact-regex",
	},
	"options": {
	"pattern":
	"\\d{2}[-.\\s]\\d{7}",
	"replacement": "NN-NNNNNNN"
	}
	} }

The table below illustrates the result of applying the rule to documents matching the rule.

Format	Original Document	Redacted Result
XML	<person> <id>**12-3456789**</id> </person>	<person> <id>NN-NNNNNNN</id> </person>
JSON	{"id": "12-3456789"}	{"id": "NN-NNNNNNN" } }

The following rule uses a back reference in the pattern to leave the first 2 digits of the id intact. The pattern in the previous example has been modified to have parentheses around the sub-expression for the first block of digits ("(\d{2})". The parentheses "capture" that block of text in a variable that is referenced in the replacement string as "$1".

XML

```
<rule xml:lang="zxx"
  xmlns="http://marklogic.com/xdmp/redaction">
  <path>//id</path>
  <method>
  <function>redact-regex</function>
  </method>
  <options>
    <pattern>
      (\d{{2}})[-.\s]\d{{7}}
    </pattern>
    <replacement>$1-NNNNNNN</replacement>
  </options>
</rule>
```

JSON

```
{"rule": {
  "path": "//id",
  "method": {
   "function": "redact-regex",
  },
  "options": {
    "pattern":
      "(\\d{2})[-.\\s]\\d{7}",
    "replacement": "$1-NNNNNNN"
  }
} }
```

Applying this rule to the same documents as before results in the following redaction:

```
12-NNNNNNN
```

For more details, see the `fn:replace` XQuery function or the `fn.replace` Server-Side JavaScript function.

For a complete example, see "Example: Using the Built-In Redaction Functions" on page 487.

24.10 Example: Using the Built-In Redaction Functions

This example exercises all the built-in redaction functions using the sample documents from "Preparing to Run the Examples" on page 526. You can choose to work with either an XML rule set or a JSON rule set. The rules are equivalent in both rule sets.

This example has the following parts:

- Example Rule Summary
- Install the rules. Choose XML or JSON. Install one set or the other, not both.
  - Install the XML Rules
  - Install the JSON Rules
- Apply the Rules
- Review the Results

24.10.1 Example Rule Summary

Each rule in this example exercises a different built-in redaction function. Each rule also operates on a different XML element or JSON property value of the sample documents to prevent overlap among the rules. You should never apply collection of rules that act on the same document components.

The rules are inserted with a URI of the following form, where *name* is the XML element local name or JSON property name of the node selected by the rule. (The URI suffix depends on the rule format you install.)

```
/rules/redact-name.{xml|json}
```

For example, `/rules/redact-alias.xml` targets the `alias` XML element or JSON property of the sample documents.

Every rule is inserted into two collections, an "all" collection and a collection that identifies the built-in used by the rule. For example, `/rules/redact-alias.json`, which uses the `mask-random` built-in, is inserted in the collections "all" and "random". This enables you to apply the rules together or selectively.

The table below summarizes the rules installed by this example:

Rule URI Basename	Built-in Function Used	Path Selector	Collections
`redact-name`	`mask-deterministic`	`//name`	`all, deterministic`
`redact-alias`	`mask-random`	`//alias`	`all, random`
`redact-address`	`conceal`	`//address`	`all, conceal`
`redact-balance`	`redact-number`	`//balance`	`all, balance`
`redate-datetime`	`redact-datetime`	`//anniversary`	`all, datetime`
`redact-ssn`	`redact-us-ssn`	`//ssn`	`all, ssn`
`redact-phone`	`redact-us-phone`	`//phone`	`all, phone`
`redact-email`	`redact-email`	`//email`	`all, email`
`redact-ip`	`redact-ipv4`	`//ip`	`all, ip`
`redact-id`	`redact-regex`	`//id`	`all, regex`

24.10.2 Install the XML Rules

To install the XML rules, copy the following script into Query Console and run it against the Schemas database. For a detailed example of installing rules with Query Console, see "Example: Getting Started With Redaction" on page 437.

Follow these steps to install the example rules in XML format using XQuery. If you prefer to use JSON rules, see "Install the JSON Rules" on page 492. For a detailed example of installing rules with Query Console, see "Example: Getting Started With Redaction" on page 437.

1. Copy the script below into Query Console.

2. Set the Query Type to XQuery.

3. Set the Database to Schemas.

4. Click Run. The rules are installed in the Schemas database.

5. Optionally, use the Query Console database explorer to review the rules.

Use the following script to install the rules. For a summary of what these rules do, see "Example Rule Summary" on page 488.

```
xquery version "1.0-ml";
import module namespace rdt = "http://marklogic.com/xdmp/redaction"
    at "/MarkLogic/redaction.xqy";

let $rules := (
<rules>
  <rule>
    <name>redact-name</name>
    <collection>deterministic</collection>
    <rdt:rule xml:lang="zxx"
        xmlns:rdt="http://marklogic.com/xdmp/redaction">
      <rdt:path>//name</rdt:path>
      <rdt:method>
        <rdt:function>mask-deterministic</rdt:function>
      </rdt:method>
      <rdt:options>
        <length>10</length>
      </rdt:options>
    </rdt:rule>
  </rule>
  <rule>
    <name>redact-alias</name>
    <collection>random</collection>
    <rdt:rule xml:lang="zxx"
        xmlns:rdt="http://marklogic.com/xdmp/redaction">
      <rdt:path>//alias</rdt:path>
      <rdt:method>
        <rdt:function>mask-random</rdt:function>
```

```
    </rdt:method>
    <rdt:options>
      <length>10</length>
    </rdt:options>
  </rdt:rule>
</rule>
<rule>
  <name>redact-address</name>
  <collection>conceal</collection>
  <rdt:rule xml:lang="zxx"
      xmlns:rdt="http://marklogic.com/xdmp/redaction">
    <rdt:path>//address</rdt:path>
    <rdt:method>
      <rdt:function>conceal</rdt:function>
    </rdt:method>
  </rdt:rule>
</rule>
<rule>
  <name>redact-balance</name>
  <collection>balance</collection>
  <rdt:rule xml:lang="zxx"
      xmlns:rdt="http://marklogic.com/xdmp/redaction">
    <rdt:path>//balance</rdt:path>
    <rdt:method>
      <rdt:function>redact-number</rdt:function>
    </rdt:method>
    <rdt:options>
      <min>0</min>
      <max>100000</max>
      <format>0.00</format>
      <type>decimal</type>
    </rdt:options>
  </rdt:rule>
</rule>
<rule>
  <name>redact-anniversary</name>
  <collection>datetime</collection>
  <rdt:rule xml:lang="zxx"
      xmlns:rdt="http://marklogic.com/xdmp/redaction">
    <rdt:path>//anniversary</rdt:path>
    <rdt:method>
      <rdt:function>redact-datetime</rdt:function>
    </rdt:method>
    <rdt:options>
      <level>random</level>
      <format>[Y0001]-[M01]-[D01]</format>
      <range>1900,1999</range>
    </rdt:options>
  </rdt:rule>
</rule>
<rule>
  <name>redact-ssn</name>
  <collection>ssn</collection>
  <rdt:rule xml:lang="zxx"
```

```
      xmlns:rdt="http://marklogic.com/xdmp/redaction">
    <rdt:path>//ssn</rdt:path>
    <rdt:method>
      <rdt:function>redact-us-ssn</rdt:function>
    </rdt:method>
    <rdt:options>
      <level>partial</level>
    </rdt:options>
  </rdt:rule>
</rule>
<rule>
  <name>redact-phone</name>
  <collection>phone</collection>
  <rdt:rule xml:lang="zxx"
      xmlns:rdt="http://marklogic.com/xdmp/redaction">
    <rdt:path>//phone</rdt:path>
    <rdt:method>
      <rdt:function>redact-us-phone</rdt:function>
    </rdt:method>
    <rdt:options>
      <level>full</level>
    </rdt:options>
  </rdt:rule>
</rule>
<rule>
  <name>redact-email</name>
  <collection>email</collection>
  <rdt:rule xml:lang="zxx"
      xmlns:rdt="http://marklogic.com/xdmp/redaction">
    <rdt:path>//email</rdt:path>
    <rdt:method>
      <rdt:function>redact-email</rdt:function>
    </rdt:method>
    <rdt:options>
      <level>name</level>
    </rdt:options>
  </rdt:rule>
</rule>
<rule>
  <name>redact-ip</name>
  <collection>ip</collection>
  <rdt:rule xml:lang="zxx"
      xmlns:rdt="http://marklogic.com/xdmp/redaction">
    <rdt:path>//ip</rdt:path>
    <rdt:method>
      <rdt:function>redact-ipv4</rdt:function>
    </rdt:method>
    <rdt:options>
      <character>X</character>
    </rdt:options>
  </rdt:rule>
</rule>
<rule>
  <name>redact-id</name>
```

```
    <collection>regex</collection>
    <rdt:rule xml:lang="zxx"
        xmlns:rdt="http://marklogic.com/xdmp/redaction">
      <rdt:path>//id</rdt:path>
      <rdt:method>
        <rdt:function>redact-regex</rdt:function>
      </rdt:method>
      <rdt:options>
        <pattern>\d{{2}}[-.\s]\d{{7}}</pattern>
        <replacement>NN-NNNNNNN</replacement>
      </rdt:options>
    </rdt:rule>
  </rule>
</rules>
)
return
for $r in $rules/rule return
  let $collections := (<collection>all</collection>, $r/collection)
  let $options :=
    <options xmlns="xdmp:document-insert">
      <permissions>{xdmp:default-permissions()}</permissions>
      <collections>
        <collection>all</collection>
        <collection>{$r/*:collection/data()}</collection>
      </collections>
    </options>
  return xdmp:document-insert(
    fn:concat("/rules/", $r/name, ".xml"),
    $r/rdt:rule, $options
  )
```

24.10.3 Install the JSON Rules

Follow these steps to install the example rules in JSON format using Server-Side JavaScript. If you prefer to use XML rules, see “Install the XML Rules” on page 489. For a detailed example of installing rules with Query Console, see “Example: Getting Started With Redaction” on page 437.

1. Copy the script below into Query Console.

2. Set the Query Type to JavaScript.

3. Set the Database to Schemas.

4. Click Run. The rules are installed in the Schemas database.

5. Optionally, use the Query Console database explorer to review the rules.

Use the following script to install the rules. For a summary of what these rules do, see “Example Rule Summary” on page 488.

```
declareUpdate();
const rules = [
  { name: 'redact-name',
    content:
      {rule: {
       path: '//name',
       method: {function: 'mask-deterministic'},
       options: {length: 10}
      }},
    collection: 'deterministic'
  },
  { name: 'redact-alias',
    content:
      {rule: {
       path: '//alias',
       method: {function: 'mask-random'},
       options: {length: 10}
      }},
    collection: 'random'
  },
  { name: 'redact-address',
    content:
      {rule: {
       path: '//address',
       method: {function: 'conceal'},
      }},
    collection: 'conceal'
  },
  { name: 'redact-balance',
    content:
      {rule: {
       path: '//balance',
       method: {function: 'redact-number'},
       options: {min: 0, max: 100000, type: 'decimal', format: '0.00'}
      }},
    collection: 'balance'
  },
  { name: 'redact-anniversary',
    content:
      {rule: {
       path: '//anniversary',
       method: {function: 'redact-datetime'},
       options: {
         level: 'random',
         format: '[Y0001]-[M01]-[D01]',
         range: '1900,1999'
       }
      }},
    collection: 'datetime'
  },
```

```
  { name: 'redact-ssn',
    content:
      {rule: {
       path: '//ssn',
       method: {function: 'redact-us-ssn'},
       options: {level: 'partial'}
      }},
    collection: 'ssn'
  },
  { name: 'redact-phone',
    content:
      {rule: {
       path: '//phone',
       method: {function: 'redact-us-phone'},
       options: {level: 'full'}
      }},
    collection: 'phone'
  },
  { name: 'redact-email',
    content:
      {rule: {
       path: '//email',
       method: {function: 'redact-email'},
       options: {level: 'name'}
      }},
    collection: 'email'
  },
  { name: 'redact-ip',
    content:
      {rule: {
       path: '//ip',
       method: {function: 'redact-ipv4'},
       options: {character: 'X'}
      }},
    collection: 'ip'
  },
  { name: 'redact-id',
    content:
      {rule: {
       path: '//id',
       method: {function: 'redact-regex'},
       options: {
         pattern: '\\d{2}[-.\\s]\\d{7}',
         replacement: 'NN-NNNNNNN'
       }
      }},
    collection: 'regex'
  }
];
rules.forEach(function (rule, i, a) {
  xdmp.documentInsert(
  '/rules/' + rule.name + '.json',
  rule.content,
  { permissions: xdmp.defaultPermissions(),
```

```
      collections: ['all', rule.collection] }
  );
})
```

24.10.4 Apply the Rules

Follow these steps to apply the complete set of example rules:

If you have not already done so, install the sample documents from “Preparing to Run the Examples” on page 526. This example assumes they are installed in the Documents database.

Choose one of the following methods to apply the rules:

- Redact Using XQuery
- Redact Using JavaScript
- Redact Using mlcp

24.10.4.1 Redact Using XQuery

Follow these steps to apply the example rules using XQuery and Query Console. All the rules will be applied to the sample documents.

1. Copy the following script into Query Console:

```
xquery version "1.0-ml";
import module namespace rdt = "http://marklogic.com/xdmp/redaction"
    at "/MarkLogic/redaction.xqy";
rdt:redact(fn:collection("people"), "all")
```

2. Set the Query Type to XQuery.

3. Set the Database to Documents.

4. Click Run.

The redacted documents should be displayed in query console. For a discussion of the expected results, see “Review the Results” on page 496.

24.10.4.2 Redact Using JavaScript

Follow these steps to apply the example rules using Server-Side JavaScript and Query Console. All the rules will be applied to the sample documents.

1. Copy the following script into Query Console:

```
const rdt = require('/MarkLogic/redaction.xqy');
rdt.redact(fn.collection('people'), 'all');
```

2. Set the Query Type to JavaScript.

3. Set the Database to Documents.

4. Click Run.

The redacted documents should be displayed in Query Console. For a discussion of the expected results, see "Review the Results" on page 496.

24.10.4.3Redact Using mlcp

Use a command line similar to the following to export the redacted documents from the Documents database. All the rules will be applied to the sample documents.

Change the example command line as needed to match your environment. The output directory (`./results`) must not already exist.

```
# Windows users, see Modifying the Example Commands for Windows
$ mlcp.sh export -host localhost -port 8000 -username user \
    -password password -mode local -output_file_path \
    ./results -collection_filter people \
    -redaction "all"
```

The redacted documents should be exported to `./results`. For a discussion of the expected results, see "Review the Results" on page 496.

For more details on using mlcp with Redaction, see Redacting Content During Export or Copy Operations in the *mlcp User Guide*.

24.10.5 Review the Results

Applying all the example rules redacts most XML elements and JSON properties of the sample documents. Recall that the following rules are applied to each element or property:

Path Selector	Built-in Function
`//name`	`mask-deterministic`
`//alias`	`mask-random`
`//address`	`conceal`
`//balance`	`mask-number`
`//anniversary`	`redact-datetime`
`//ssn`	`redact-us-ssn`
`//phone`	`redact-us-phone`

Path Selector	Built-in Function
//email	redact-email
//ip	redact-ipv4
//id	redact-regex
//balance	redact-number

The following table illustrates the effect on the sample documents `/redact-ex/person1.xml`. The redacted values you observe will differ from those shown if the rule generates a value, rather than masking an existing value.

Original Document	Redacted Document
<person> <name>Little Bopeep</name> <alias>Peepers</alias> <alias>Bo</alias> <address> <street>100 Nursery Lane</street> <city>Hometown</city> <country>Neverland</country> </address> <ssn>123-45-6789</ssn> <phone>123-456-7890</phone> <email>bopeep@mothergoose.com</email> <ip>111.222.33.4</ip> <id>12-3456789</id> <birthdate>2015-01-15</birthdate> <anniversary>2017-04-18</anniversary> <balance>12.34</balance> </person>	<person> <name>63a63aa762</name> <alias>47c1fc8b29</alias> <alias>7a314dcf2d</alias> <ssn>###-##-6789</ssn> <phone>###-###-####</phone> <email>NAME@mothergoose.com</email> <ip>XXX.XXX.XXX.XXX</ip> <id>NN-NNNNNNN</id> <birthdate>2015-01-15</birthdate> <anniversary>1930-05-13</anniversary> <balance>0.67</balance> </person>

The following table illustrates the effect on the sample document `/redact-ex/person3.json`.

Original Document	Redacted Document

<pre>{ "name": "Georgie Porgie",
  "alias": ["George", "G.P."],
  "address": {
  "street": "300 Nursery Lane",
  "city": "Hometown",
  "country": "Neverland"
  },
  "ssn": "345678901",
  "phone": "(345)678-9012",
  "email": "gp@mothergoose.com",
  "ip": "33.44.5.66",
  "id": "34-5678912",
  "birthdate": "2012-07-12",
  "anniversary": "2014-10-15",
  "balance": "12345.67"
}</pre>
	<pre>{ "name":"34fe55c66a",
	  "alias":["27a76af34e", "8b87c3e8c6"],
	  "ssn":"#####8901",
	  "phone":"(###)###-####",
	  "email":"NAME@mothergoose.com",
	  "ip":"XXX.XXX.XXX.XXX",
	  "id":"NN-NNNNNNN",
	  "birthdate":"2012-07-12",
	  "anniversary": "1926-05-19"
	  "balance": "5.28"
	}</pre>

You should observe similar changes to `/redact-ex/person2.xml` and `/redact-ex/person3.json`.

> **Note:** The results in Query Console will not necessarily be in the order person1, person2, person3, etc.

24.11 User-Defined Redaction Functions

If the built-in redaction functions do not address the needs of your application, you can implement a user-defined redaction function in XQuery or Server-Side JavaScript. Follow these steps to deploy and apply a user-defined function:

1. Implement the function. For details, see “Implementing a User-Defined Redaction Function” on page 499.

2. Install the function in the Modules database associated with your App Server. For details, see “Installing a User-Defined Redaction Function” on page 500.

3. Define a rule that specifies your function. For syntax, see “Defining Redaction Rules” on page 445.

4. Install and apply the rule.

This section covers the following topics:

- Implementing a User-Defined Redaction Function
- Installing a User-Defined Redaction Function

For a complete example, see “Example: Using Custom Redaction Rules” on page 503.

24.11.1 Implementing a User-Defined Redaction Function

A user-defined function can be implemented in XQuery or Server-Side JavaScript. Your implementation must conform to one of the following interfaces:

Language	Interface

XQuery	`declare function yourNS:yourFunc(`
	`  $node as node(),`
	`  $options as map:map`
	`) as node()?`

Server-Side JavaScript	`function yourFunc(node, options)`
	`// where:`
	`//   node is a Node`
	`//   options is an Object with paramName:value properties`
	`//   return 1 Node or nothing`

The input `node` parameter is the node selected by the XPath expression in a rule using your function. The `options` parameter can be used to pass user-defined data from the rule into your function. Your function should return a node (redacted or not) or nothing.

Your function should be defined in an XQuery or JavaScript library module. The module should be installed in the modules database associated with the App Server through which redaction will be applied. For details, see “Installing a User-Defined Redaction Function” on page 500.

The following table contains module templates suitable for defining your own conforming module. For a complete example, see “Example: Custom Redaction Using JavaScript” on page 503 or “Example: Custom Redaction Using XQuery” on page 509.

Language	Interface

XQuery	`xquery version "1.0-ml";`
	`module namespace yourNS = "/your/module/namespace";`
	
	`declare function yourNS:redact(`
	`  $node as node(),`
	`  $options as map:map`
	`) as node()?`
	`{`
	`  (: your implementation here :)`
	`};`

Server-Side JavaScript	`function yourFunc(node, options)`
	`{`
	`    // your implementation here`
	`}`
	
	`exports.redact = yourFunc`

24.11.2 Installing a User-Defined Redaction Function

Install your implementation in the modules database associated with your App Server using normal document insertion methods, such as the `xdmp:document-insert` XQuery function, the `xdmp.documentInsert` Server-Side JavaScript function, or any of the document insertion features of the Node.js, Java, or REST Client APIs.

For more details, see one of the following topics:

- Installing a Redaction Module Using XQuery
- Installing a Redaction Module Using JavaScript
- Installing a Redaction Module Using the Client APIs

24.11.2.1Installing a Redaction Module Using XQuery

The procedure in this section demonstrates how to use Query Console and XQuery to install a module in the modules database. You can also use Server-Side JavaScript and the Java, Node.js, and REST Client APIs for this task.

The procedure outlined here makes the following assumptions. You will need to modify the procedure and example code to match your environment and application requirements.

- MarkLogic is installed on localhost.
- The modules database associated with your App Server is Modules.
- Your implementation is saved to a file on the file system with the path `/your/module/path/impl.xqy`.
- The default document permissions are suitable for the module permissions.

Use a procedure similar to the following to install your XQuery module in the Modules database.

1. Navigate to Query Console in your browser. For example, go to `http://localhost:8000/qconsole`.

2. Paste the following script into Query Console. Modify the module URI and the path in the `xdmp:document-get` line to match your environment.

```
(: MODIFY THE FILE SYSTEM PATH AND URI TO MATCH YOUR ENV :)
xquery version "1.0-ml";
xdmp:document-insert(
  "/your/module/uri",
  xdmp:document-get("/your/module/path/impl.xqy"),
  <options xmlns="xdmp:document-insert">
    <permissions>{xdmp:default-permissions()}</permissions>
  </options>
)
```

3. Select Modules in the Database dropdown.

4. Select XQuery in the Query Type dropdown.

5. Click the Run button. The module is installed in the Modules database.

You can use the Explore feature of Query Console to browse the Modules database and confirm the installation.

24.11.2.2Installing a Redaction Module Using JavaScript

The procedure in this section demonstrates how to use Query Console and Server-Side JavaScript to install a module in the modules database. You can also use XQuery or the Java, Node.js, and REST Client APIs for this task.

The procedure outlined here makes the following assumptions. You will need to modify the procedure and example code to match your environment and application requirements.

- MarkLogic is installed on localhost.

- The modules database associated with your App Server is Modules.
- Your implementation is saved to a file on the file system with the path `/your/module/path/impl.sjs`.
- The default document permissions are suitable for the module permissions.

Use a procedure similar to the following to install your XQuery module in the Modules database.

1. Navigate to Query Console in your browser. For example, go to `http://localhost:8000/qconsole`.

2. Paste the following script into Query Console. Modify the module URI and the path in the `xdmp.documentGet` line to match your environment.

```
// MODIFY THE FILE SYSTEM PATH and URI TO MATCH YOUR ENV
declareUpdate();
xdmp.documentInsert(
  '/your/module/uri',
  xdmp.documentGet('/your/module/path/impl.sjs'));
```

3. Select Modules in the Database dropdown.

4. Select JavaScript in the Query Type dropdown.

5. Click the Run button. The module is installed in the Modules database.

You can use the Explore feature of Query Console to browse the Modules database and confirm the installation.

24.11.2.3Installing a Redaction Module Using the Client APIs

The Java Client API, Node.js Client API, and Node.js Client API include the capability to install modules in the modules database. See one of the following topics for details on how to install a module using one of the Client APIs.

- Java: Managing Dependent Libraries and Other Assets in the *Java Application Developer's Guide*
- Node.js: Managing Assets in the Modules Database in the *Node.js Application Developer's Guide*
- REST: Managing Dependent Libraries and Other Assets in the *REST Application Developer's Guide*

24.12 Example: Using Custom Redaction Rules

This example walks you through installing and applying a custom redaction function. Two versions of the example are available, one that it JSON/JavaScript centric and another that is XML/XQuery centric. This artificial split is made to keep the example simple. You can mix XML and JSON freely with both XQuery and Server-Side JavaScript.

Choose one of the following examples to explore using custom redaction rules.

- Example: Custom Redaction Using JavaScript
- Example: Custom Redaction Using XQuery

24.12.1 Example: Custom Redaction Using JavaScript

This example operates on JSON documents that include personal profile data such as name, address, and date of birth. A custom Server-Side JavaScript redaction function is used to redact the name if the person is less than 18 years old. A rule-specific option value controls the replacement text.

For simplicity, this example only uses JavaScript and JSON. You can also write a custom a function to handle both XML and JSON. For a similar XQuery/XML example, see “Example: Custom Redaction Using JavaScript” on page 503.

Before running the example, install the sample documents from “Preparing to Run the Examples” on page 526.

The example has the following parts:

- Input Data
- Installing the Redaction Function
- Installing the Redaction Rule
- Applying the Rule Using JavaScript
- Applying the Rule Using mlcp

24.12.1.1Input Data

The input documents have the following structure. The `birthdate` property is used to determine whether or not to redact the `name` property.

```
{ "name": "any text",
  ...
  "birthdate": "YYYY-MM-DD"
}
```

To install the sample documents, see “Preparing to Run the Examples” on page 526.

24.12.1.2Installing the Redaction Function

Use the following procedure to install the custom function into the Modules database with the URI `/redaction/redact-xml-name.sjs`. These instructions use Server-Side JavaScript and Query Console, but you can use any document insertion interface. Discussion of the function follows the procedure.

1. Save the following custom redaction function implementation to a file named "redact-json-name.sjs". Choose a location readable by MarkLogic.

```
function redactName(node, options) {
  const parent = fn.head(node.xpath('./parent::node()'));

  // only redact if containing obj has the expected 'shape'
  if (parent.nodeKind == 'object' &&
      parent.hasOwnProperty('birthdate')) {
    const birthday =
      xdmp.parseDateTime('[Y0001]-[M01]-[D01]', parent.birthdate);
    const age = Math.floor(fn.daysFromDuration(
                fn.currentDateTime().subtract(birthday)) / 365);
    if (age < 18) {
      // underage, so redact
      const builder = new NodeBuilder();
      builder.addText(options.newName);
      return builder.toNode();
    }
  }
  // not expected input, or not underage - do nothing
  return node;
};

exports.redact = redactName;
```

2. Navigate to Query Console in your browser. For example, go to `http://localhost:8000/qconsole`.

3. Paste the following script into Query Console. Modify the path in the `xdmp.documentGet` line to match the file location from Step 1.

```
// MODIFY THE FILE SYSTEM PATH TO MATCH YOUR ENV
declareUpdate();
xdmp.documentLoad(
  '/your/path/redact-json-name.sjs',
  {uri: '/redaction/redact-json-name.sjs'});
```

4. Select Modules in the Database dropdown.

5. Select JavaScript in the Query Type dropdown.

6. Click the Run button. The module is installed in the Modules databasew ith the URI "/redaction/redact-xml-name.sjs".

You can use Query Console to explore the Modules database and confirm the installation.

The custom function expects to receive a JSON node corresponding to the node that is a candidate for redaction. This node must be a child of an object that also has a `birthdate` property. This code snippet implements this check:

```
const parent = fn.head(node.xpath('./parent::node()'));

// only redact if containing obj has the expected 'shape'
if (parent.nodeKind == 'object' &&
  parent.hasOwnProperty('birthdate')) {

  ...
```

Note that you could theoretically write the function to expect the parent object as input and have the redaction rule use an XPath expression such as `/name/parent::node()`. However, such a rule path is invalid if the rule is ever applied to an XML document, so we traverse up to the parent node inside the redaction function instead of in the rule. For more details, see "Limitations on XPath Expressions in Redaction Rules" on page 449.

The redaction function uses the `birthdate` element to compute the age. If the age is less than 18, then the text in the `name` element is redacted. The value of the "newName" property in the options object is used as the replacement text.

```
const birthday =
  xdmp.parseDateTime('[Y0001]-[M01]-[D01]', parent.birthdate);
const age = Math.floor(fn.daysFromDuration(
                fn.currentDateTime().subtract(birthday)) / 365);
if (age < 18) {
  // underage, so redact
  const builder = new NodeBuilder();
  builder.addText(options.newName);
  return builder.toNode();
}
```

Redaction functions must return a node, not a simple value. In this case, we need to return a JSON text node that will replace the original input node. You cannot construct a text node from a native JavaScript object, so the function uses a `NodeBuilder` to construct the return node.

These requirements are not specific to working with the root object node. Any time you have a node as input and want to modify it as a native JavaScript type, you need to use `toObject`. Similarly, you must always return a node, not a native JavaScript value.

24.12.1.3Installing the Redaction Rule

Use the following procedure to install the rule in the schemas database associated with your content database. Some discussion of the rule follows the procedure.

These instructions assume you will use the pre-installed App Server on localhost:8000 and the Documents database, which is configured to use the Schemas database. This example uses Server-Side JavaScript and Query Console to install the rule, but you can use any document insertion interface.

1. Navigate to Query Console in your browser. For example, go to `http://localhost:8000/qconsole`.

2. Paste the following script into a new query tab in Query Console.

```
declareUpdate();
xdmp.documentInsert('/rules/redact-name.json',
  { rule: {
    path: '/name',
    method: {
      function: 'redact',
      module: '/redaction/redact-json-name.sjs'
    },
    options: { newName: 'Jane Doe' }
  }},
  { permissions: xdmp.defaultPermissions(),
    collections: ['custom-rules'] }
);
```

3. Select Schemas in the Database dropdown.

4. Select JavaScript in the Query Type dropdown.

5. Click the Run button. The rule document is installed with the URI "/rules/redact-name.json" and added to the "custom-rules" collection.

The path expression in the rule selects the `name` property for redaction. Since the custom function uses the `birthdate` sibling property of `name` to control the redaction, it would be more natural in some ways to apply the rule to the parent object. However, the parent object is anonymous, so it cannot be addressed by name in an XPath expression.

An XPath expression such as `/name/parent::node()` would select the anonymous parent object, but it will cause an error if the rule is ever applied to an XML document. Since we have a mixed XML and JSON document set, we choose write the rule and the custom function to use the `name` property as the redaction target.

The custom function is identified in the rule by exported function name and the URI of the implementation installed in the modules database:

```
method: {
  function: 'redact',
  module: '/redaction/redact-json-name.sjs'
}
```

The `options` property contains a single child, `newName`. This value is used as the replacement value for any redacted `name` elements:

```
options: { newName: 'Jane Doe' }
```

For a similar XQuery/XML example of defining and installing a rule that uses a custom function, see "Example: Custom Redaction Using XQuery" on page 509.

24.12.1.4Applying the Rule Using JavaScript

Follow this procedure to apply the example custom redaction function using Query Console and `rdt.redact`. You should already have installed the custom redaction module, rule, and sample documents.

1. Navigate to Query Console in your browser. For example, go to `http://localhost:8000/qconsole`.

2. Paste the following script into a new query tab in Query Console:

```
const jsearch = require('/MarkLogic/jsearch');
const rdt = require('/MarkLogic/redaction');

jsearch.collections('people').documents()
    .map(function (match) {
        match.document = fn.head(
          rdt.redact(fn.root(match.document), 'custom-rules')
        .root;
        return match;
    }).result();
```

3. Select Documents in the Databases dropdown.

4. Select JavaScript in the Query Type dropdown.

5. Click the Run button. The rules in the "custom-rules" collection are applied to the documents in the "people" collection.

If you use the sample documents from "Preparing to Run the Examples" on page 526, running the script should have the following effect on the search result matches:

- /redact-ex/person1.xml: Unredacted because it doesn't match the rule path
- /redact-ex/person2.xml: Unredacted because it doesn't match the rule path
- /redact-ex/person3.json: Name changed to "Jane Doe"
- /redact-ex/person4.json: Unredacted because not under the age limit

(Note, if you installed both the XQuery/XML and JavaScript/JSON custom redaction examples, `/people/person1.xml` will also be redacted to display"John Doe".)

Note that the node passed to rdt.redact is obtained by applying `fn.root` to `match.document`.

```
rdt.redact(fn.root(match.document), 'custom-rules')
```

The `rdt.redact` function expects a document node as input, whereas `match.document` is the root node under the document node, such as a JSON object-node or XML element node. In the context of `DocumentsSearch.map`, the node in `match.document` is an in-database node, not an in-memory construct, so we can access the enclosing document node using `fn.root`, as shown above.

A similar technique is used, in reverse, to save the redaction result back into the search results:

```
match.document = fn.head(rdt.redact(...)).root;
```

This is necessary because `rdt.redact` function returns a `Sequence` of in-memory document nodes. To save the redacted content in the expected form, we access the first node in the `Sequence` with `fn.head`, and then "dereference" it using the ".root" property so that `match.document` again contains the root node under the document node.

24.12.1.5Applying the Rule Using mlcp

You can apply the example custom redaction rule with mlcp by running a command similar to the one below. The command exports the redacted documents to `./mlcp-output`. This directory must not already exist.

Modify the command line as needed to match your environment.

```
# Windows users, see Modifying the Example Commands for Windows
$ mlcp.sh export -host localhost -port 8000 -username user \
    -password password -mode local -output_file_path \
    ./mlcp-output -collection_filter people \
    -redaction "custom-rules"
```

For more details, see Redacting Content During Export or Copy Operations in the *mlcp User Guide*.

If you use the sample documents from "Preparing to Run the Examples" on page 526, running the script should create 4 files in the directory `./mlcp-output`.

These files should reflect the following effects relative to the input documents:

- /redact-ex/person1.xml: Unredacted because it doesn't match the rule path
- /redact-ex/person2.xml: Unredacted because it doesn't match the rule path
- /redact-ex/person3.json: Name changed to "Jane Doe"
- /redact-ex/person4.json: Unredacted because not under the age limit

(Note, if you installed both the XQuery/XML and JavaScript/JSON custom redaction examples, `/people/person1.xml` will also be redacted to display"John Doe".)

24.12.2 Example: Custom Redaction Using XQuery

This example operates on XML documents that include personal profile data such as name, address, and date of birth. A custom XQuery redaction function is used to redact the name if the person is less than 18 years old. A rule-specific option value controls the replacement text.

This example only uses XQuery and XML. You can write a custom a function to handle both XML and JSON, but you might find it more convenient to use XQuery for XML and Server-Side JavaScript for JSON. For an equivalent JavaScript/JSON example, see "Example: Custom Redaction Using JavaScript" on page 503.

Before running this example, you must install the sample documents from "Preparing to Run the Examples" on page 526.

The example has the following parts:

- Input Data
- Installing the Redaction Function
- Installing the Redaction Rule
- Applying the Rule Using XQuery
- Applying the Rule Using mlcp

24.12.2.1 Input Data

The input documents have the following structure. The `birthdate` element is used to determine whether or not to redact the `name` element.

```
<person>
  <name>any text</name>
  ...
  <birthdate>YYYY-MM-DD</birthdate>
</person>
```

To install the sample documents, see "Preparing to Run the Examples" on page 526.

24.12.2.2 Installing the Redaction Function

Use the following procedure to install the custom function into the Modules database with the URI `/redaction/redact-xml-name.xqy`. These instructions use XQuery and Query Console, but you can use any document insertion interface.

1. Save the following custom redaction function implementation to a file named "redact-xml-name.xqy". Choose a location readable by MarkLogic.

```
xquery version "1.0-ml";
module namespace my = "http://marklogic.com/example/redaction";
```

```
declare function my:redact(
  $node as node(),
  $options as map:map
) as node()?
{
  if (xdmp:node-kind($node) = "element" and
      fn:local-name-from-QName(fn:node-name($node)) = "person")
  then
    let $birthdate :=
      xdmp:parse-dateTime('[Y0001]-[M01]-[D01]', $node//birthdate)
    let $age := math:floor(fn:days-from-duration(
        fn:current-dateTime() - $birthdate)) div 365
    return
      if ($age < 18)
      then element { fn:node-name($node) } {
        $node/@*,
        for $n in ($node/node()) return
          if (fn:local-name-from-QName(fn:node-name($n)) = "name")
          then element {fn:node-name($n)} {
            $n/@*, text {map:get($options, "new-name")}
          }
          else $n
      }
      else $node
  else $node
};
```

2. Navigate to Query Console in your browser. For example, go to `http://localhost:8000/qconsole`.

3. Paste the following script into Query Console. Modify the path in the `xdmp:document-get` line to match the file location from Step 1.

```
(: MODIFY THE FILE SYSTEM PATH TO MATCH YOUR ENV :)
xquery version "1.0-ml";
xdmp:document-load(
  "/your/path/redact-xml-name.xqy",
  <options xmlns="xdmp:document-load">
    <uri>/redaction/redact-xml-name.xqy</uri>
  </options>
)
```

4. Select Modules in the Database dropdown.

5. Select XQuery in the Query Type dropdown.

6. Click the Run button. The module is installed in the Modules database with the URI "/redaction/redact-xml-name.xqy".

You can use Query Console to explore the Modules database and confirm the installation.

The custom function expects to receive a `<person/>` node as input and options that include a "new-name" key specifying the replacement name value.

The function uses the `birthdate` element to compute the age. If the age is less than 18, then the text in the `name` element is redacted.

If the input does not have the expected "shape" or the age is 18 or older, the input node is returned, unchanged.

For a similar JavaScript-based solution, see "Example: Custom Redaction Using JavaScript" on page 503.

24.12.2.3Installing the Redaction Rule

Use the following procedure to install the rule in the schemas database associated with your content database. Some discussion of the rule follows the procedure.

These instructions assume you will use the pre-installed App Server on localhost:8000 and the Documents database, which is configured to use the Schemas database. This example uses XQuery and Query Console to install the rule, but you can use any document insertion interface.

1. Navigate to Query Console in your browser. For example, go to `http://localhost:8000/qconsole`.

2. Paste the following script into a new query tab in Query Console.

```
xquery version "1.0-ml";
xdmp:document-insert("/rules/redact-name.xml",
  <rdt:rule xml:lang="zxx"
      xmlns:rdt="http://marklogic.com/xdmp/redaction">
    <rdt:path>/person</rdt:path>
    <rdt:method>
      <rdt:function>redact</rdt:function>
      <rdt:module>/redaction/redact-xml-name.xqy</rdt:module>
      <rdt:module-namespace>http://marklogic.com/example/redaction</rd
t:module-namespace>
    </rdt:method>
    <rdt:options>
      <new-name>John Doe</new-name>
    </rdt:options>
  </rdt:rule>
, <options xmlns="xdmp:document-insert">
    <permissions>{xdmp:default-permissions()}</permissions>
    <collections>
      <collection>custom-rules</collection>
    </collections>
  </options>)
```

3. Select Schemas in the Database dropdown.

4. Select XQuery in the Query Type dropdown.

5. Click the Run button. The rule document is installed with URI "/rules/redact-name.xml" and added to the "custom-rules" collection.

Recall that the sample documents are rooted at a `<person/>` element, so the rule selects the entire contents by using "/person" as the path value. This enables the redaction function to easily examine `/person/birthdate`, as well as modify `/person/name`.

The custom function is identified in the rule by function name, module URI, and module namespace:

```
<rdt:method>
  <rdt:function>redact</rdt:function>
  <rdt:module>/redaction/redact-xml-name.xqy</rdt:module>
    <rdt:module-namespace>
      http://marklogic.com/example/redaction
    </rdt:module-namespace>
</rdt:method>
```

The `options` element contains a single element, `new-name`, that is used as the replacement value for any redacted `name` elements:

```
<rdt:options>
  <new-name>John Doe</new-name>
</rdt:options>
```

For a similar JavaScript/JSON example of defining and installing a rule that uses a custom function, see "Example: Custom Redaction Using JavaScript" on page 503.

24.12.2.4Applying the Rule Using XQuery

Follow this procedure to apply the example custom redaction function using Query Console and `rdt:redact`. You should already have installed the custom redaction module, rule, and sample documents.

1. Navigate to Query Console in your browser. For example, go to `http://localhost:8000/qconsole`.

2. Paste the following script into a new query tab in Query Console:

```
xquery version "1.0-ml";
import module namespace rdt = "http://marklogic.com/xdmp/redaction"
    at "/MarkLogic/redaction.xqy";
rdt:redact(
  cts:search(fn:doc(), cts:collection-query("people")),
  "custom-rules")
```

3. Select Documents in the Databases dropdown.

4. Select XQuery in the Query Type dropdown.

5. Click the Run button. The rules in the "custom-rules" collection are applied to the documents in the "people" collection.

If you use the sample documents from "Preparing to Run the Examples" on page 526, running the script should return the following:

- /redact-ex/person1.xml: Name redacted by changing it to John Doe
- /redact-ex/person2.xml: Unredacted due to age > 18
- /redact-ex/person3.json: Unredacted because it doesn't match the rule path
- /redact-ex/person4.json: Unredacted because it doesn't match the rule path

(Note, if you installed both the XQuery/XML and JavaScript/JSON custom redaction examples, `/people/person3.json` will also be redacted to display "Jane Doe".)

24.12.2.5Applying the Rule Using mlcp

You can apply the example custom redaction rule with mlcp by running a command similar to the following. The command exports the redacted documents to `./mlcp-output`. This directory must not already exist.

Modify the command line as needed to match your environment.

```
# Windows users, see Modifying the Example Commands for Windows
$ mlcp.sh export -host localhost -port 8000 -username user \
    -password password -mode local -output_file_path \
    ./mlcp-output -collection_filter people \
    -redaction "custom-rules"
```

For more details, see in Redacting Content During Export or Copy Operations the *mlcp User Guide*.

If you use the sample documents from "Preparing to Run the Examples" on page 526, running the script should create 4 files in the directory `./mlcp-output`. These files should reflect the following effects relative to the input documents:

- /redact-ex/person1.xml: Name redacted by changing it to John Doe
- /redact-ex/person2.xml: Unredacted due to age > 18
- /redact-ex/person3.json: Unredacted because it doesn't match the rule path
- /redact-ex/person4.json: Unredacted because it doesn't match the rule path

(Note, if you installed both the XQuery/XML and JavaScript/JSON custom redaction examples, `person3.json` will also be redacted to display"Jane Doe".)

24.13 Using Dictionary-Based Masking

Some pre-defined redaction functions that mask content can extract the masking value from a redaction dictionary. This section covers the following topics related to using a dictionary for a masking source:

- Defining a Redaction Dictionary
- Installing a Redaction Dictionary
- Using a Redaction Dictionary

24.13.1 Defining a Redaction Dictionary

A redaction dictionary is an XML or JSON document with the form specified below.

Format	Syntax

XML	`<dictionary xmlns="http://marklogic.com/xdmp/redaction">`
	`  <entry>`*value*`</entry>`
	`  ...`
	`</dictionary>`

JSON	`{ "dictionary": {`
	`  "entry":[`
	`  `*value*`,`
	`  ...`
	`  ]`
	`}}`

The following requirements apply. If these requirements are not met, you will get an `RDT-INVALIDDICTIONARY` error when you use the dictionary.

- A dictionary must contain at least one entry.
- The value in an entry cannot be empty or null.
- The value must be atomic. That is:
  - In XML, the entry value can be any text (word, phrase, date, decimal, etc.).
  - In JSON, the value can be a string, number, or boolean value.

The following example is a trivial dictionary containing four entries of various types. For a complete example, see "Example: Dictionary-Based Masking" on page 516.

Format	Syntax

XML	`<dictionary xmlns="http://marklogic.com/xdmp/redaction">`
	`  <entry>a phrase</entry>`
	`  <entry>a_term</entry>`
	`  <entry>1234</entry>`
	`  <entry>true</entry>`
	`</dictionary>`

JSON	`{ "dictionary": {`
	`  "entry":[`
	`  "a phrase",`
	`  "a_term",`
	`  1234,`
	`  true`
	`  ]`
	`}}`

24.13.2 Installing a Redaction Dictionary

Before you can use a redaction dictionary, you must install it in the schemas database associated with the database that contains the content to be redacted. This should be the same database in which you install your redaction rules.

Install the using the same techniques discussed in "Installing Redaction Rules" on page 456.

For security purposes, you should use document permissions to carefully control who can read or modify your dictionary. For more details, see "Security Considerations" on page 443.

24.13.3 Using a Redaction Dictionary

The pre-defined redaction functions that support dictionary-based masking do so through a `dictionary` option that accepts a dictionary URI as its value.

For example, the `mask-deterministic` and `mask-random` built-in redaction functions support a `dictionary` option, so you can draw values from a dictionary with a rule similar to the following:

```
<rule xml:lang="zxx" xmlns="http://marklogic.com/xdmp/redaction">
  <path>//country</path>
  <method>
  <function>mask-random</function>
  </method>
  <options>
    <dictionary>/rules/dict/countries.xml</dictionary>
  </options>
</rule>
```

For more details, see “Built-in Redaction Function Reference” on page 462. For a complete example, see “Example: Dictionary-Based Masking” on page 516.

24.14 Example: Dictionary-Based Masking

This section contain an example that demonstrates how to install a redaction dictionary and use it with built-in redaction functions. The examples rules perform the following redactions:

- The `mask-deterministic` function and a JSON dictionary is applied to the `country` XML element or JSON property of the sample data.
- The `mask-random` function and an XML dictionary is applied to the street XML element or JSON property of the sample data.

Before running this example, you must install the sample documents from “Preparing to Run the Examples” on page 526.

Use the following steps to exercise the example:

- Install the Dictionaries
- Install the Rules
- Apply the Rules

24.14.1 Install the Dictionaries

Use either of the following procedures to install example dictionaries. The procedure installs two dictionaries: A dictionary of country names, defined in XML, and a dictionary of street addresses, defined in JSON.

- Install Dictionaries Using XQuery
- Install Dictionaries Using JavaScript

24.14.1.1Install Dictionaries Using XQuery

The following procedure installs the two example dictionaries:

1. Copy the script below into a new query in Query Console.
2. Set the Query Type to XQuery.
3. Set the Database to Schemas.
4. Click Run. The dictionaries are installed in the Schemas database with the URIs `/rules/dict/countries.xml` and `/rules/dict/streets.json`.
5. Optionally, use the Query Console database explorer to review the dictionaries.

Use the following script in Step 1, above.

```
(: NOTE: RUN AGAINST YOUR SCHEMAS DB :)

(: Install example XML dictionary :)
xquery version "1.0-ml";
let $dictURI := '/rules/dict/countries.xml'
let $dict :=
  <dictionary xmlns="http://marklogic.com/xdmp/redaction">
    <entry>Brazil</entry>
    <entry>China</entry>
    <entry>France</entry>
    <entry>Germany</entry>
    <entry>United States</entry>
    <entry>United Kingdom</entry>
  </dictionary>
return
xdmp:document-insert($dictURI, $dict,
  <options xmlns="xdmp:document-insert">
    <permissions>{xdmp:default-permissions()}</permissions>
  </options>);

(: Install example JSON dictionary :)
xquery version "1.0-ml";
let $dictURI := '/rules/dict/steets.json'
let $dict := xdmp:unquote(
  '{ "dictionary": {
    "entry": [
      "10 Oak Ln",
      "2451 Elm St",
      "892 Veterans Blvd",
      "P.O. Box 1234",
      "250 Park Ln",
      "16 Highway 82, Suite 301"
    ]
  } }')
return
xdmp:document-insert(
  $dictURI, $dict,
  <options xmlns="xdmp:document-insert">
    <permissions>{xdmp:default-permissions()}</permissions>
  </options>);
```

24.14.1.2Install Dictionaries Using JavaScript

The following procedure installs the two example dictionaries:

1. Copy the script below into a new query in Query Console.
2. Set the Query Type to JavaScript.
3. Set the Database to Schemas.

4. Click Run. The dictionaries are installed in the Schemas database with the URIs `/rules/dict/countries.xml` and `/rules/dict/streets.json`.

5. Optionally, use the Query Console database explorer to review the dictionaries.

Use the following script in Step 1, above.

```
// NOTE: RUN AGAINST YOUR SCHEMAS DB
declareUpdate();

// Install example XML dictionary
const countryDict = fn.head(xdmp.unquote(
  '<dictionary xmlns="http://marklogic.com/xdmp/redaction">' +
    '<entry>Brazil</entry>' +
    '<entry>China</entry>' +
    '<entry>France</entry>' +
    '<entry>Germany</entry>' +
    '<entry>United States</entry>' +
    '<entry>United Kingdom</entry>' +
  '</dictionary>'));
xdmp.documentInsert(
  '/rules/dict/countries.xml', countryDict,
  { permissions: xdmp.defaultPermissions() }
);

// Install example JSON dictionary
const streetDict =
  { dictionary: {
    entry: [
      '10 Oak Ln',
      '2451 Elm St',
      '892 Veterans Blvd',
      'P.O. Box 1234',
      '250 Park Ln',
      '16 Highway 82, Suite 301'
    ]
  } };
xdmp.documentInsert(
  '/rules/dict/streets.json', streetDict,
  { permissions: xdmp.defaultPermissions() }
);
```

24.14.2 Install the Rules

Use either of the following procedures to install rules that exercise the dictionaries. One rule is defined using XML, and the other rule is defined using JSON.

- Install Rules Using XQuery
- Install Rules Using JavaScript

24.14.2.1Install Rules Using XQuery

The following procedure installs two rules, each of which uses one of the dictionaries installed in "Install the Dictionaries" on page 516:

1. Copy the script below into a new query in Query Console.
2. Set the Query Type to XQuery.
3. Set the Database to Schemas.
4. Click Run. The rules are installed in the Schemas database with the URIs `/rules/randomize-country.xml` and `/rules/redact-street.json`.
5. Optionally, use the Query Console database explorer to review the rules.

Use the following script in Step 1, above.

```
(: NOTE: RUN AGAINST YOUR SCHEMAS DB :)

(: Install rule using mask-random with a dictionary :)
xquery version "1.0-ml";
let $ruleURI := '/rules/randomize-country.xml'
let $rule :=
<rule xml:lang="zxx"
  xmlns="http://marklogic.com/xdmp/redaction">
  <path>//country</path>
  <method>
  <function>mask-random</function>
  </method>
  <options>
    <dictionary>/rules/dict/countries.xml</dictionary>
  </options>
</rule>
return xdmp:document-insert(
  $ruleURI, $rule,
  <options xmlns="xdmp:document-insert">
    <permissions>{xdmp:default-permissions()}</permissions>
    <collections>
      <collection>dict</collection>
      <collection>dict-random</collection>
    </collection>
  </options>);

(: Install rule using mask-deterministic with a dictionary :)
xquery version "1.0-ml";
let $ruleURI := '/rules/redact-street.json'
let $rule := xdmp:unquote(
  '{"rule": {
     "path": "//street",
     "method": {"function": "mask-deterministic"},
     "options": {"dictionary": "/rules/dict/streets.json"}
```

```
  }}'
)
return xdmp:document-insert(
  $ruleURI, $rule,
  <options xmlns="xdmp:document-insert">
    <permissions>{xdmp:default-permissions()}</permissions>
    <collections>
      <collection>dict</collection>
      <collection>dict-deter</collection>
    </collections>
  </options>
);
```

24.14.2.2Install Rules Using JavaScript

The following procedure installs two rules, each of which uses one of the dictionaries installed in “Install the Dictionaries” on page 516:

1. Copy the script below into a new query in Query Console.
2. Set the Query Type to JavaScript.
3. Set the Database to Schemas.
4. Click Run. The rules are installed in the Schemas database with the URIs `/rules/randomize-country.xml` and `/rules/redact-street.json`.
5. Optionally, use the Query Console database explorer to review the rules.

Use the following script in Step 1, above.

```
// NOTE: RUN AGAINST YOUR SCHEMAS DB

declareUpdate();

// Install rule using mask-random with dictionary
xdmp.documentInsert(
  '/rules/randomize-country.xml',
  fn.head(xdmp.unquote(
    '<rule xml:lang="zxx" xmlns="http://marklogic.com/xdmp/redaction">' +
      '<path>//country</path>' +
      '<method>' +
        '<function>mask-random</function>' +
      '</method>' +
      '<options>' +
        '<dictionary>/rules/dict/countries.xml</dictionary>' +
      '</options>' +
    '</rule>')),
    { permissions: xdmp.defaultPermissions(),
      collections: ['dict','dict-random'] }
);
```

```
// Install rule using mask-deterministic with dictionary
xdmp.documentInsert(
  '/rules/redact-street.json',
  {rule: {
     path: '//street',
     method: {function: 'mask-deterministic'},
     options: {dictionary: '/rules/dict/streets.json'}
  }},
  { permissions: xdmp.defaultPermissions(),
    collections: ['dict','dict-deter'] }
);
```

24.14.3 Apply the Rules

Choose one of the following methods for exercising the rules that use dictionary-based masking:

- Apply the Rules Using XQuery
- Apply the Rules Using JavaScript
- Apply the Rules Using mlcp

24.14.3.1Apply the Rules Using XQuery

Follow these steps to apply the example rules using XQuery and Query Console. All the rules will be applied to the sample documents.

1. Copy the following script into Query Console:

```
xquery version "1.0-ml";
import module namespace rdt = "http://marklogic.com/xdmp/redaction"
    at "/MarkLogic/redaction.xqy";
let $results := rdt:redact(fn:collection("people"), "dict")
return (
  "*** REDACTED STREETS ***",
  $results//street/data(),
  "*** REDACTED COUNTRIES ****",
  $results//country/data()
)
```

2. Set the Query Type to XQuery.

3. Set the Database to Documents.

4. Click Run. The redacted street and country names from each document are displayed.

You should see output similar to the following, though the values may vary.

```
*** REDACTED STREETS ***
P.O. Box 1234
2451 Elm St
```

```
892 Veterans Blvd
250 Park Ln
*** REDACTED COUNTRIES ****
United States
Brazil
Germany
France
```

If you run the script again, the values for the street names will not change because they are redacted using `mask-deterministic`. The values for the countries will change with each run since they are redacted using `mask-random`.

24.14.3.2Apply the Rules Using JavaScript

Follow these steps to apply the example rules using XQuery and Query Console. All the rules will be applied to the sample documents.

1. Copy the following script into Query Console:

```
const rdt = require('/MarkLogic/redaction.xqy');
const results = rdt.redact(fn.collection('people'), 'dict');

// Extract the redacted streed and country data for display purposes
const displayAccumulator = ['*** STREETS ***'];
for (let doc of results) {
  displayAccumulator.push(doc.xpath('//street/data()'));
}
displayAccumulator.push('*** COUNTRIES ***');
for (let doc of results) {
  displayAccumulator.push(doc.xpath('//country/data()'));
}

// Dump the redacted street and country values
displayAccumulator
```

2. Set the Query Type to JavaScript.

3. Set the Database to Documents.

4. Click Run. The redacted street and country names from each document are displayed.

You should see output similar to the following, though the values may vary.

```
*** REDACTED STREETS ***
P.O. Box 1234
2451 Elm St
892 Veterans Blvd
250 Park Ln
*** REDACTED COUNTRIES ****
United States
Brazil
```

```
Germany
France
```

If you run the script again, the values for the street names will not change because they are redacted using `mask-deterministic`. The values for the countries will change with each run since they are redacted using `mask-random`.

24.15 Salting Masking Values for Added Security

When you use the `mask-deterministic` built-in redaction function without a salt, two rules with equivalent options always produce the same output for the same input. You can use a "salt" to introduce masking value variance across rules, rule sets, or clusters. When you use a salt, each masking value is still deterministic in that the same input produces the same output. However, the same input with different salts produces different output.

The `mask-deterministic` function supports applying a salt to masking value generation via the following options. You can use them individually or together.

- `salt`: A user-defined salt value. This option has no value by default.
- `extend-salt`: Include the cluster id or rule set collection name in the salt. This option defaults to `cluster-id`.

To completely disable the salt, set `salt` to an empty string (or leave it unspecified) and set `extend-salt` to `none`.

For example, consider the following rules that apply equivalent redaction logic to two different paths, using no salt:

```
<rule xml:lang="zxx" xmlns="http://marklogic.com/xdmp/redaction">
    <path>/data/pii1</path>
    <method>
      <function>mask-deterministic</function>
    </method>
    <options>
      <length>20</length>
      <salt/>
      <extend-salt>none</extend-salt>
    </options>
</rule>

<rule xml:lang="zxx" xmlns="http://marklogic.com/xdmp/redaction">
    <path>/data/pii2</path>
    <method>
      <function>mask-deterministic</function>
    </method>
    <options>
      <length>20</length>
      <salt/>
      <extend-salt>none</extend-salt>
```

```
        </options>
    </rule>
```

If you apply these rules to the following documents, both produce the same masking value by default for the input "John Smith":

Unredacted Data	Redacted Data
<data> <pii1>John Smith</pii1> </data>	<data> <pii1>**6c50dad68163a7a079db**</pii1> </data>
<data> <pii2>John Smith</pii2> </data>	<data> <pii2>**6c50dad68163a7a079db**</pii2> </data>

An attacker could use a similar "salt-less" rule to generate a lookup table that indicates "John Smith" redacts to "6c50dad68163a7a079db". That knowledge can be used to reverse engineer redacted output.

However, if you modify the "/data/pii1" rule to include a `salt` option:

```
<rule xml:lang="zxx" xmlns="http://marklogic.com/xdmp/redaction">
    <path>/data/pii1</path>
    <method>
      <function>mask-deterministic</function>
    </method>
    <options>
      <length>20</length>
      <salt>anyoldthing</salt>
    </options>
</rule>
```

Then the masking values generated by the two rules differ as shown below. An attacker cannot deduce the relationship between the redacted value ("89d7499b154a8b81c17f") and the input value ("John Smith") without also knowing the salt.

Unredacted Data	Redacted Data
<data> <pii1>John Smith</pii1> </data>	<data> <pii1>**89d7499b154a8b81c17f**</pii1> </data>
<data> <pii2>John Smith</pii2> </data>	<data> <pii2>**6c50dad68163a7a079db**</pii2> </data>

By default, `extend-salt` option is set to `cluster-id` and the `salt` option is empty. This means that equivalent rules applied on the same cluster will generate the same output, but the same values would not be generated on a different cluster.

Similarly, setting `extend-salt` to `collection` means that an attacker who has access to one rule set cannot generate a lookup table that can be used to reverse engineer redacted values generated by a different rule set.

The following table outlines the impact of various `salt` and `extend-salt` option combinations, assuming all other options are the same.

salt	extend-salt	Effect
empty (default)	none	For a given input, all rules with no salt value produce the same output.
any value	none	For a given input, all rules with the same salt value produce the same output for the same input.
empty	cluster-id	For a given input, a rule applied in cluster C produces the same output as other rules with no salt applied in cluster C. Any rule specifying a non-empty salt applied in cluster C produces different output, as does any rule applied in a different cluster.
any value	cluster-id	For a given input, a rule applied in Cluster C only produces the same output as other rules with the same salt applied in cluster C. Any rule with a different or no salt applied in Cluster C produces different output, as does any rule applied in a different cluster.
empty	collection	For a given input, any rule in rule collection R produces the same output as other rules in R that do not specify a salt. Rules in another rule collection produce different output.
any value	collection	For a given input, a rule in rule collection R only produces the same output as other rules in R with the same salt. Rules in another rule collection produce different output, even with the same salt.

24.15.0.1Apply the Rules Using mlcp

Use a command line similar to the following to export the redacted documents from the Documents database. Both dictionary-based rules will be applied to the sample documents.

Change the example command line as needed to match your environment. The output directory (`./dict-results`) must not already exist.

```
# Windows users, see Modifying the Example Commands for Windows
$ mlcp.sh export -host localhost -port 8000 -username user \
    -password password -mode local -output_file_path \
    ./dict-results -collection_filter people \
    -redaction "dict"
```

The redacted documents should be exported to `./dict-results`. The `//street` and `//country` values should reflect values from the street and country dictionaries, respectively.

The redacted streets values should be the same each time you export the documents because they are redacted using `mask-deterministic`. The redacted country values should change each time you export the documents because they are redacted using `mask-random`.

For more details on using mlcp with Redaction, see Redacting Content During Export or Copy Operations in the *mlcp User Guide*.

24.16 Preparing to Run the Examples

Unless otherwise noted, the examples in this chapter are based on the same set of source documents. The source document set consists of two XML documents and two JSON documents with similar structure. They include some complex element and property values, such as child XML elements or JSON objects, and JSON arrays.

The documents are inserted into collections so they can easily be selected for redaction. The "people" collection contains all the samples. The "xml-people" collection includes only the XML samples. The "json-people" collection includes only the JSON samples.

When you complete the steps in this section, your Documents database will contain the following documents. The collection names are shown in parentheses after the URI in the following list.

- `/redact-ex/person1.xml` (`people`, `xml-people`)
- `/redact-ex/person2.xml` (`people`, `xml-people`)
- `/redact-ex/person3.json` (`people`, `json-people`)
- `/redact-ex/person4.json` (`people`, `json-people`)

Follow these steps to install the sample documents:

1. Navigate to Query Console in your browser. For example, go to `http://localhost:8000/qconsole`.

2. Paste the following script into a new query tab in Query Console:

```
xquery version "1.0-ml";
xdmp:document-insert("/redact-ex/person1.xml",
```

```
  <person>
    <name>Little Bopeep</name>
    <alias>Peepers</alias>
    <alias>Bo</alias>
    <address>
      <street>100 Nursery Lane</street>
      <city>Hometown</city>
      <country>Neverland</country>
    </address>
    <ssn>123-45-6789</ssn>
    <phone>123-456-7890</phone>
    <email>bopeep@mothergoose.com</email>
    <ip>111.222.33.4</ip>
    <id>12-3456789</id>
    <birthdate>2015-01-15</birthdate>
    <anniversary>2017-04-18</anniversary>
    <balance>12.34</balance>
  </person>,
  <options xmlns="xdmp:document-insert">
    <permissions>{xdmp:default-permissions()}</permissions>
    <collections>
      <collection>people</collection>
      <collection>xml-people</collection>
    </collections>
  </options>
);

xquery version "1.0-ml";
xdmp:document-insert("/redact-ex/person2.xml",
  <person>
    <name>Humpty Dumpty</name>
    <alias>Dumpy</alias>
    <address>
      <street>200 Nursery Lane</street>
      <city>Hometown</city>
      <country>Neverland</country>
    </address>
    <ssn>234.56.7890</ssn>
    <phone>234.567.8901</phone>
    <email>hdumpty@mothergoose.com</email>
    <ip>222.3.44.5</ip>
    <id>23-4567891</id>
    <birthdate>1965-06-12</birthdate>
    <anniversary>2012-11-09</anniversary>
    <balance>567.89</balance>
  </person>,
  <options xmlns="xdmp:document-insert">
    <permissions>{xdmp:default-permissions()}</permissions>
    <collections>
      <collection>people</collection>
      <collection>xml-people</collection>
    </collections>
  </options>
);
```

```
xquery version "1.0-ml";
xdmp:document-insert("/redact-ex/person3.json", xdmp:unquote('
  { "name": "Georgie Porgie",
    "alias": ["George", "G.P."],
    "address": {
      "street": "300 Nursery Lane",
      "city": "Hometown",
      "country": "Neverland"
    },
    "ssn": "345678901",
    "phone": "(345)678-9012",
    "email": "gp@mothergoose.com",
    "ip": "33.44.5.66",
    "id": "34-5678912",
    "birthdate": "2012-07-12",
    "anniversary": "2014-10-15",
    "balance": 12345.67
  }'),
  <options xmlns="xdmp:document-insert">
    <permissions>{xdmp:default-permissions()}</permissions>
    <collections>
      <collection>people</collection>
      <collection>json-people</collection>
    </collections>
  </options>
);

xquery version "1.0-ml";
xdmp:document-insert("/redact-ex/person4.json", xdmp:unquote('
  { "name": "Jack Sprat",
    "alias": ["Jacko","Beanpole"],
    "address": {
      "street": "400 Nursery Lane",
      "city": "Hometown",
      "country": "Neverland"
    },
    "ssn": "456-78-9012",
    "phone": "4567890123",
    "email": "jack.sprat@mothergoose.com",
    "ip": "4.55.6.77",
    "id": "45-6789123",
    "birthdate": "1995-10-04",
    "anniversary": "2010-05-23",
    "balance": "90.12"
  }'),
  <options xmlns="xdmp:document-insert">
    <permissions>{xdmp:default-permissions()}</permissions>
    <collections>
      <collection>people</collection>
      <collection>json-people</collection>
    </collections>
  </options>
);
```

3. Select Documents in the Database dropdown.

4. Select XQuery in the Query Type dropdown.

5. Click the Run button. The sample documents are installed in the Documents database.

6. Optionally, click Explore next to the Database dropdown to explore the database and confirm insertion of the sample documents.

25.0 Copyright

MarkLogic Server 9.0 and supporting products.
Last updated: November 18, 2019

COPYRIGHT

© 2019 MarkLogic Corporation. All rights reserved.
This technology is protected by U.S. Patent No. 7,127,469B2, U.S. Patent No. 7,171,404B2, U.S. Patent No. 7,756,858 B2, and U.S. Patent No 7,962,474 B2, US 8,892,599, and US 8,935,267.

The MarkLogic software is protected by United States and international copyright laws, and incorporates certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers set forth below.

For all copyright notices, including third-party copyright notices, see the Combined Product Notices for your version of MarkLogic.

26.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at http://help.marklogic.com to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the Support Handbook for instructions on registering support contacts and on working with the MarkLogic Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at http://developer.marklogic.com. For technical questions, we encourage you to ask your question on Stack Overflow.