MarkLogic Server

Application Developer's Guide

MarkLogic 6
September, 2012

Last Revised: 6.0-3, April, 2013

Copyright © 2013 MarkLogic Corporation. All rights reserved.

Table of Contents

Application Developer's Guide

1.0 Developing Applications in MarkLogic Server 10
1.1 Overview of MarkLogic Server Application Development 10
1.2 Skills Needed to Develop MarkLogic Server Applications 10
1.3 Where to Find Specific Information 11

2.0 Loading Schemas 12
2.1 Configuring Your Database 12
2.2 Loading Your Schema 13
2.3 Referencing Your Schema 14
2.4 Working With Your Schema 14
2.5 Validating XML Against a Schema 15

3.0 Understanding Transactions in MarkLogic Server 17
3.1 Terms and Definitions 18
3.2 Overview of MarkLogic Server Transactions 21
3.2.1 Transaction Attributes: Mode, Type, and Number of Statements 21
3.2.2 Single-Statement Transaction Concept Summary 22
3.2.3 Multi-Statement Transaction Concept Summary 23
3.3 Transaction Type 24
3.3.1 Transaction Type Overview 24
3.3.2 Query Transactions: Point-in-Time Evaluation 25
3.3.3 Update Transactions: Readers/Writers Locks 26
3.3.4 Example: Query and Update Transaction Interaction 28
3.4 Single vs. Multi-statement Transactions 29
3.4.1 Single-Statement, Automatically Committed Transactions 29
3.4.2 Multi-Statement, Explicitly Committed Transactions 30
3.4.3 Semi-Colon as a Statement Separator 35
3.5 Transaction Mode 36
3.5.1 Transaction Mode Overview 37
3.5.2 Controlling the Transaction Mode 37
3.5.3 Auto Transaction Mode 39
3.5.4 Query Transaction Mode 39
3.5.5 Update Transaction Mode 40
3.6 Interactions with xdmp:eval/invoke 41
3.6.1 Isolation Option to xdmp:eval/invoke 41
3.6.2 Preventing Undetectable Deadlocks 42
3.6.3 Seeing Updates From eval/invoke Later in the Transaction 44
3.6.4 Running Multi-Statement Transactions under xdmp:eval/invoke 45

3.7 Functions With Non-Transactional Side Effects 46
3.8 Reducing Blocking with Multi-Version Concurrency Control 47
3.9 Administering Transactions 47
3.10 Transaction Examples 48
3.10.1 Example: Incrementing the System Timestamp 49
3.10.2 Example: Multi-statement Transactions and Same-statement Isolation ...49
3.10.3 Example: Multi-Statement Transactions and Different-transaction Isolation 51
3.10.4 Example: Generating a Transaction Report With xdmp:host-status 52

4.0 Working With Binary Documents 54
4.1 Terminology 54
4.2 Loading Binary Documents 55
4.3 Configuring MarkLogic Server for Binary Content 55
4.3.1 Setting the Large Size Threshold 55
4.3.2 Sizing and Scalability of Binary Content 56
4.3.3 Selecting a Location For Binary Content 57
4.3.4 Monitoring the Total Size of Large Binary Data in a Forest 58
4.3.5 Detecting and Removing Orphaned Binaries 59
4.4 Developing Applications That Use Binary Documents 60
4.4.1 Adding Metadata to Binary Documents Using Properties 60
4.4.2 Downloading Binary Content With HTTP Range Requests 61
4.4.3 Creating Binary Email Attachments 63
4.5 Useful Built-ins for Manipulating Binary Documents 64

5.0 Importing XQuery Modules, XSLT Stylesheets, and Resolving Paths 66
5.1 XQuery Library Modules and Main Modules 66
5.1.1 Main Modules 66
5.1.2 Library Modules 67
5.2 Rules for Resolving Import, Invoke, and Spawn Paths 67
5.3 Module Caching Notes 69
5.4 Example Import Module Scenario 70

6.0 Library Services Applications 71
6.1 Understanding Library Services 71
6.2 Building Applications with Library Services 73
6.3 Required Range Element Indexes 73
6.4 Library Services API 74
6.4.1 Library Services API Categories 75
6.4.2 Managed Document Update Wrapper Functions 75
6.5 Security Considerations of Library Services Applications 76
6.5.1 dls-admin Role 76
6.5.2 dls-user Role 76
6.5.3 dls-internal Role 77
6.6 Transactions and Library Services 77

6.7 Putting Documents Under Managed Version Control 77
6.8 Checking Out Managed Documents 78
6.8.1 Displaying the Checkout Status of Managed Documents 78
6.8.2 Breaking the Checkout of Managed Documents 78
6.9 Checking In Managed Documents 79
6.10 Updating Managed Documents 79
6.11 Defining a Retention Policy 80
6.11.1 Purging Versions of Managed Document 80
6.11.2 About Retention Rules 81
6.11.3 Creating Retention Rules 81
6.11.4 Retaining Specific Versions of Documents 83
6.11.5 Multiple Retention Rules 84
6.11.6 Deleting Retention Rules 86
6.12 Managing Modular Documents in Library Services 87
6.12.1 Creating Managed Modular Documents 87
6.12.2 Expanding Managed Modular Documents 89
6.12.3 Managing Versions of Modular Documents 90

7.0 Transforming XML Structures With a Recursive typeswitch Expression ..93
7.1 XML Transformations 93
7.1.1 XQuery vs. XSLT 93
7.1.2 Transforming to XHTML or XSL-FO 93
7.1.3 The typeswitch Expression 94
7.2 Sample XQuery Transformation Code 94
7.2.1 Simple Example 95
7.2.2 Simple Example With cts:highlight 96
7.2.3 Sample Transformation to XHTML 97
7.2.4 Extending the typeswitch Design Pattern 99

8.0 Document and Directory Locks 100
8.1 Overview of Locks 100
8.1.1 Write Locks 100
8.1.2 Persistent 100
8.1.3 Searchable 101
8.1.4 Exclusive or Shared 101
8.1.5 Hierarchical 101
8.1.6 Locks and WebDAV 101
8.1.7 Other Uses for Locks 101
8.2 Lock APIs 101
8.3 Example: Finding the URI of Documents With Locks 102
8.4 Example: Setting a Lock on a Document 103
8.5 Example: Releasing a Lock on a Document 103
8.6 Example: Finding the User to Whom a Lock Belongs 104

9.0 Properties Documents and Directories 105

9.1 Properties Documents 105
9.1.1 Properties Document Namespace and Schema 105
9.1.2 APIs on Properties Documents 107
9.1.3 XPath property Axis 108
9.1.4 Protected Properties 109
9.1.5 Creating Element Indexes on a Properties Document Element 109
9.1.6 Sample Properties Documents 109
9.1.7 Standalone Properties Documents 109
9.2 Using Properties for Document Processing 110
9.2.1 Using the property Axis to Determine Document State 110
9.2.2 Document Processing Problem 111
9.2.3 Solution for Document Processing 112
9.2.4 Basic Commands for Running Modules 113
9.3 Directories 113
9.3.1 Properties and Directories 114
9.3.2 Directories and WebDAV Servers 114
9.3.3 Directories Versus Collections 115
9.4 Permissions On Properties and Directories 115
9.5 Example: Directory and Document Browser 115
9.5.1 Directory Browser Code 116
9.5.2 Setting Up the Directory Browser 117

10.0 Point-In-Time Queries 119
10.1 Understanding Point-In-Time Queries 119
10.1.1 Fragments Stored in Log-Structured Database 119
10.1.2 System Timestamps and Merge Timestamps 120
10.1.3 How the Fragments for Point-In-Time Queries are Stored 120
10.1.4 Only Available on Query Statements, Not on Update Statements 121
10.1.5 All Auxiliary Databases Use Latest Version 121
10.1.6 Database Configuration Changes Do Not Apply to Point-In-Time Fragments 122
10.2 Using Timestamps in Queries 122
10.2.1 Enabling Point-In-Time Queries in the Admin Interface 122
10.2.2 The xdmp:request-timestamp Function 124
10.2.3 Requires the xdmp:timestamp Execute Privilege 124
10.2.4 The Timestamp Parameter to xdmp:eval, xdmp:invoke, xdmp:spawn ..124
10.2.5 Timestamps on Requests in XCC 125
10.2.6 Scoring Considerations 125
10.3 Specifying Point-In-Time Queries in xdmp:eval, xdmp:invoke, xdmp:spawn, and XCC 126
10.3.1 Example: Query Old Versions of Documents Using XCC 126
10.3.2 Example: Querying Deleted Documents 126
10.4 Keeping Track of System Timestamps 127
10.5 Rolling Back a Forest to a Particular Timestamp 129
10.5.1 Tradeoffs and Scenarios to Consider For Rolling Back Forests 129
10.5.2 Setting the Merge Timestamp 130

10.5.3 Notes About Performing an xdmp:forest-rollback Operation 130
10.5.4 General Steps for Rolling Back One or More Forests 131

11.0 System Plugin Framework 132
11.1 How MarkLogic Server Plugins Work 132
11.1.1 Overview of System Plugins 132
11.1.2 System Plugins versus Application Plugins 133
11.1.3 The plugin API 133
11.2 Writing System Plugin Modules 134
11.3 Information Studio Plugins 134
11.4 Password Plugin Sample 134
11.4.1 Understanding the Password Plugin 135
11.4.2 Modifying the Password Plugin 136

12.0 Using the map Functions to Create Name-Value Maps 138
12.1 Maps: In-Memory Structures to Manipulate in XQuery 138
12.2 map:map XQuery Primitive Type 138
12.3 Serializing a Map to an XML Node 139
12.4 Map API 139
12.5 Examples 139
12.5.1 Creating a Simple Map 140
12.5.2 Returning the Values in a Map 140
12.5.3 Constructing a Serialized Map 140
12.5.4 Add a Value that is a Sequence 141

13.0 Function Values 142
13.1 Overview of Function Values 142
13.2 xdmp:function XQuery Primitive Type 142
13.3 XQuery APIs for Function Values 143
13.4 When the Applied Function is an Update from a Query Statement 143
13.5 Example of Using Function Values 143

14.0 Reusing Content With Modular Document Applications 146
14.1 Modular Documents 146
14.2 XInclude and XPointer 147
14.2.1 Example: Simple id 148
14.2.2 Example: xpath() Scheme 148
14.2.3 Example: element() Scheme 148
14.2.4 Example: xmlns() and xpath() Scheme 149
14.3 CPF XInclude Application and API 149
14.3.1 XInclude Code and CPF Pipeline 149
14.3.2 Required Security Privileges—xinclude Role 150
14.4 Creating XML for Use in a Modular Document Application 150
14.4.1 <xi:include> Elements 151

14.4.2 <xi:fallback> Elements 151
14.4.3 Simple Examples 151
14.5 Setting Up a Modular Document Application 153

15.0 Controlling App Server Access, Output, and Errors 155
15.1 Creating Custom HTTP Server Error Pages 155
15.1.1 Overview of Custom HTTP Error Pages 155
15.1.2 Error XML Format 156
15.1.3 Configuring Custom Error Pages 156
15.1.4 Execute Permissions Are Needed On Error Handler Document for Modules Databases 157
15.1.5 Example of Custom Error Pages 157
15.2 Setting Up URL Rewriting for an HTTP App Server 158
15.2.1 Overview of URL Rewriting 158
15.2.2 Loading the Shakespeare XML Content 159
15.2.3 A Simple URL Rewriter 161
15.2.4 Creating URL Rewrite Modules 163
15.2.5 Prohibiting Access to Internal URLs 164
15.2.6 URL Rewriting and Page-Relative URLs 164
15.2.7 Using the URL Rewrite Trace Event 165
15.3 Outputting SGML Entities 167
15.3.1 Understanding the Different SGML Mapping Settings 167
15.3.2 Configuring SGML Mapping in the App Server Configuration 168
15.3.3 Specifying SGML Mapping in an XQuery Program 169
15.4 Specifying the Output Encoding 169
15.4.1 Configuring App Server Output Encoding Setting 169
15.4.2 XQuery Built-In For Specifying the Output Encoding 170
15.5 Specifying Output Options at the App Server Level 171

16.0 Using the REST Library to Develop RESTful Web Services 172
16.1 Terms Used in this Chapter 172
16.2 Overview of the REST Library 173
16.3 A Simple Rewriter and Endpoint 174
16.4 The options Node 177
16.5 Validating options Node Elements 179
16.6 Extracting Multiple Components from a URL 180
16.7 Handling Errors 182
16.8 Handling Redirects 182
16.9 Handling HTTP Verbs 184
16.9.1 Handling OPTIONS Requests 185
16.9.2 Handling POST Requests 187
16.10 Defining Parameters 188
16.10.1Parameter Types 189
16.10.2Supporting Parameters Specified in a URL 189
16.10.3Required Parameters 190

16.10.4Default Parameter Value 190
16.10.5Specifying a List of Values 191
16.10.6Repeatable Parameters 191
16.10.7Parameter Key Alias 191
16.10.8Matching Regular Expressions in Parameters 192
16.11 Adding Conditions 193
16.11.1Authentication Condition 194
16.11.2Accept Headers Condition 194
16.11.3User Agent Condition 194
16.11.4Function Condition 195
16.11.5And Condition 195
16.11.6Or Condition 196

17.0 Working With JSON 197
17.1 JSON, XML, and MarkLogic 197
17.2 Converting JSON to XML and XML to JSON 197
17.2.1 Loading JSON Documents Using the REST or Java Client APIs 198
17.2.2 Using the XQuery Library to Convert between JSON and XML 198
17.3 Low-Level JSON APIs and Primitive Types 199
17.4 Examples 201
17.4.1 Conversion To and From JSON and XML Examples 201
17.4.2 Low-Level API Examples 205
17.4.3 Loading JSON Using the REST or Java Client API 206

18.0 Using Triggers to Spawn Actions 209
18.1 Overview of Triggers 209
18.1.1 Trigger Components 209
18.1.2 Databases Used By Triggers 210
18.2 Triggers and the Content Processing Framework 211
18.3 Pre-Commit Versus Post-Commit Triggers 212
18.3.1 Pre-Commit Triggers 212
18.3.2 Post-Commit Triggers 212
18.4 Trigger Events 213
18.4.1 Database Events 213
18.4.2 Data Events 213
18.5 Trigger Scope 214
18.6 Modules Invoked or Spawned by Triggers 215
18.6.1 Difference in Module Behavior for Pre- and Post-Commit Triggers 215
18.6.2 Module External Variables trgr:uri and trgr:trigger 216
18.7 Creating and Managing Triggers With triggers.xqy 216
18.8 Simple Trigger Example 217
18.9 Avoiding Infinite Trigger Loops (Trigger Storms) 219

19.0 User-Defined Functions 222
19.1 What Are Aggregate User-Defined Functions 222

19.2 In-Database MapReduce Concepts 222
19.2.1 What is MapReduce? 223
19.2.2 How In-Database MapReduce Works 223
19.3 Implementing an Aggregate User-Defined Function 224
19.3.1 Creating and Deploying an Aggregate UDF 224
19.3.2 Implementing AggregateUDF::map 225
19.3.3 Implementing AggregateUDF::reduce 227
19.3.4 Implementing AggregateUDF::finish 228
19.3.5 Registering an Aggregate UDF 229
19.3.6 Aggregate UDF Memory Management 230
19.3.7 Implementing AggregateUDF::encode and AggregateUDF::decode 232
19.3.8 Aggregate UDF Error Handling and Logging 233
19.3.9 Aggregate UDF Argument Handling 234
19.3.10Type Conversions in Aggregate UDFs 235
19.4 Implementing Native Plugin Libraries 237
19.4.1 How MarkLogic Server Manages Native Plugins 238
19.4.2 Building a Native Plugin Library 238
19.4.3 Packaging a Native Plugin 239
19.4.4 Installing a Native Plugin 240
19.4.5 Uninstalling a Native Plugin 241
19.4.6 Registering a Native Plugin at Runtime 241
19.4.7 Versioning a Native Plugin 242
19.4.8 Checking the Status of Loaded Plugins 243
19.4.9 The Plugin Manifest 244
19.4.10Native Plugin Security Considerations 245

20.0 Technical Support 246

Combined Product Notices 247
COPYRIGHT 247
TRADEMARK NOTICE 259

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 XQuery 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 content application. Developers build applications using XQuery both to search the content and as a programming language in which to develop the applications. The applications can integrate with other environments via web services or via an XCC interface from Java or .NET. But it is possible to create entire applications using only MarkLogic Server, and programmed entirely in XQuery.

This *Application Developer's Guide* focuses primarily on techniques, design patterns, and concepts needed to use XQuery to build content and search applications in MarkLogic Server. If you are using the Java and 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 API, see the *Java Application Developer's Guide*. For information about developing application with the REST API, see *REST 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*.
- Understanding of search engines and full-text queries.
- Java or .Net experience, if you are using XCC to develop applications.
- 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 online 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 *Understanding and Using 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 on using XCC to access content in MarkLogic Server from Java or .Net, 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 17 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:

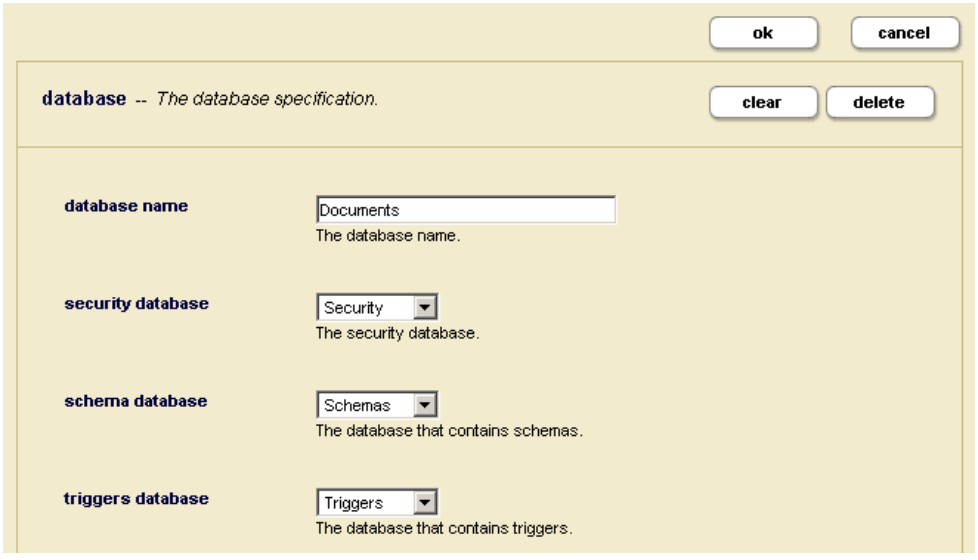
- Configuring Your Database
- Loading Your Schema
- Referencing Your Schema
- Working With Your Schema
- Validating XML Against a Schema

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 below) 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 Against a Schema

You can 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() }
```

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
- 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. 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 may 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 mode* and the kind of statements in the transaction. A transaction is either a *single-statement transaction* or a *multi-statement transaction*, depending on the transaction mode at the time it is created.

Term	Definition
transaction mode	Controls the type and the commit semantics of newly created transactions. Mode is one of `auto`, `query`, or `update`. In the default mode, `auto`, all transactions are single-statement transactions. In `query` or `update` mode, all transactions are multi-statement transactions. To learn more about transaction mode, see "Transaction Mode" on page 36.
single-statement transaction	Any transaction created in `auto` transaction 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 `query` or `update` transaction 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`.
query transaction	A transaction which cannot perform any updates; a read-only transaction. A transaction consisting of a single *query statement* in `auto` transaction mode, or any transaction created in `query` transaction mode. 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 may perform updates (make changes to the database). A transaction consisting of a single update statement in `auto` transaction mode, or any transaction created under `update` transaction mode. 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.

Term	Definition
rollback	Immediately terminate a transaction and discard all updates made by the transaction. All transactions are automatically rolled back on error. Multi-statement transactions may 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`.
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 26.
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 mode. The precise nature of a session depends on the context of the conversation. For details, see "Sessions" on page 34.
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.

- Transaction Attributes: Mode, Type, and Number of Statements
- Single-Statement Transaction Concept Summary
- Multi-Statement Transaction Concept Summary

The remainder of the chapter covers these concepts in detail.

3.2.1 Transaction Attributes: Mode, Type, and Number of Statements

The MarkLogic Server API and XQuery extensions support two transaction programming models. The model affects the number of statements in the transaction, the visibility of updates, and the commit semantics. The two models are:

- Single-statement transactions created in `auto` transaction mode, which are automatically committed (default).
- Multi-statement transactions which must be explicitly committed. Updates made by a statement in a multi-statement transaction are visible to subsequent statements in the same transaction.

A program may use either or both models. The default model is suitable for most applications. Multi-statement transactions are powerful, but introduce more complexity to your application. Focus on the concepts which match your chosen transactional programming model.

In addition to being single or multi-statement, transactions are typed as either update or query. The type determines what operations are permitted and if, when, and how locks are acquired.

The transactional model (single or multi-statement) and the transaction type (query or update) are derived from the transaction mode in effect at the time the transaction is created. The transaction mode may be `auto`, `update`, or `query`. The default mode, `auto`, creates transactions that use the single-statement model. The `query` and `update` modes create multi-statement transactions. Changing the transaction mode in the middle of a transaction does not affect existing transactions.

The following table summarizes how these transaction attributes interlock:

Mode	Model	Type	Commit	Uses Locks
`auto` (default)	single-statement	query or update, depending upon static analysis	automatic	only for an update transaction
`query`	multi-statement	query	explicit	no
`update`	multi-statement	update	explicit	yes

A statement is either a query statement or an update statement, based on static analysis of the statement prior to evaluation. The kind of statement determines the transaction type in `auto` transaction mode and affects lock handling in multi-statement update transactions.

3.2.2 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
Transaction type is determined by static analysis of the statement.	Transaction Type
Updates made by a statement are not visible until the statement (transaction) completes.	Update Transactions: Readers/Writers Locks
Semi-colon may be used as a statement/transaction separator to include multiple statements in a main module.	Single-Statement, Automatically Committed Transactions Semi-Colon as a Statement Separator

3.2.3 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`.	Multi-Statement, Explicitly Committed Transactions Committing Multi-Statement Transactions
Rollback may be implicit or explicit (`xdmp:rollback`).	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
Transactions run in a session.	Sessions
Sessions have a transaction mode property which may be `auto`, `query` or `update`. The mode affects: • transaction type • commit semantics • how many statements a transaction may contain	Transaction Mode
The default transaction mode, `auto`, always creates single-statement, auto-commit transactions, with the type determined by static analysis of the statement.	Single vs. Multi-statement Transactions
`update` and `query` transaction mode always create multi-statement, explicit commit transactions, with the type determined by the transaction mode.	Transaction Mode 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
Semi-colon may be used as a statement separator to include multiple statements in a transaction.	Semi-Colon as a Statement Separator
Updates made by a statement are visible to subsequent statements in the same transaction.	Multi-Statement, Explicitly Committed Transactions

3.3 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
- Query Transactions: Point-in-Time Evaluation
- Update Transactions: Readers/Writers Locks
- Example: Query and Update Transaction Interaction

3.3.1 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 may perform both query and update operations. Query transactions and statements are read-only and may not attempt update operations. A query transaction may 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 39.

MarkLogic Server determines transaction type in two ways, depending upon the transaction programming model in effect when a transaction is created:

- In the default model (single-statement, auto-commit), the transaction type is determined through static analysis of the statement in the transaction.
- In the multi-statement model, transaction type is derived from the transaction mode explicitly set by the prolog option xdmp:transaction-mode or the XQuery built-in function `xdmp:set-transaction-mode`.

The model, in turn, is determined by the transaction mode, as discussed in “Transaction Mode” on page 36.

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 25 and "Update Transactions: Readers/Writers Locks" on page 26.

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.

3.3.2 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

3.3.2.1 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 the "Understanding and Controlling Database Merges" chapter of 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 119. For an example of how system timestamps change as updates occur to the system, see "Example: Incrementing the System Timestamp" on page 49.

3.3.2.2 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.

3.3.2.3 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.

3.3.3 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

3.3.3.1 Identifying Update Transactions

In a single-statement transaction, if the potential for updates is found in the statement 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 single-statement transaction that is determined (during static analysis) to be an update transaction 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 mode when the transaction is created. If the transaction mode is `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.

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.

3.3.3.2 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 :)
fn:node-replace("/docs/test.xml"/node(), <a>hello</al>);
  (: readers/writers lock released :)
  (: query statement, no locks needed :)
fn:doc("/docs/test.xml");
```

If the same example is rewritten as a multi-statement transactions, 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 :)
fn:node-replace("/docs/test.xml"/node(), <a>hello</al>);
  (: readers/writers lock still held :)
fn:doc("/docs/test.xml");

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

3.3.3.3 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 mode affects the visibility of updates, indirectly, because it affects when commit occurs.

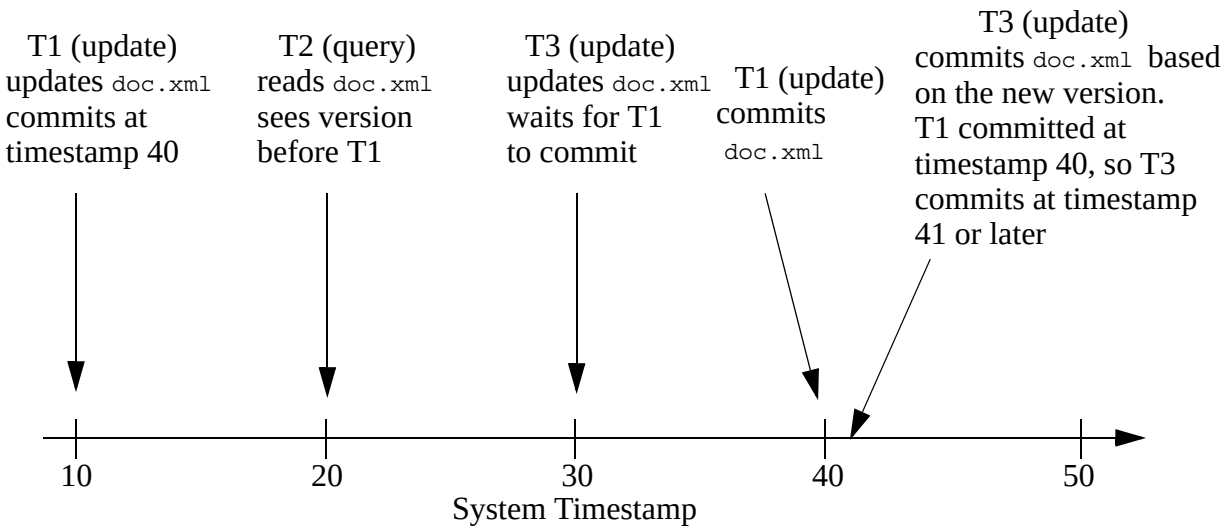
In the default single-statement transaction model, the commit occurs automatically when the statement completes. To use a newly updated document, you must separate the update and the access into two single-statement transactions or use multi-statement transactions.

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.

3.3.4 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.

3.4 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

3.4.1 Single-Statement, Automatically Committed Transactions

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

Updates made by the 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 28.

Use the semi-colon separator extension to include multiple single-statement transactions in your program. For details, see "Semi-Colon as a Statement Separator" on page 35.

3.4.2 Multi-Statement, Explicitly Committed Transactions

By explicitly setting the transaction mode to `query` or `update`, you may use multi-statement transactions. This section covers:

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

For details on setting the transaction mode, see "Controlling the Transaction Mode" on page 37.

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

3.4.2.1 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:

- Semi-colon acts as a separator between statements in the same transaction.
- 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` to commit the transaction.
- You may use `xdmp:rollback` to abort the transaction.

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`, when it is explicitly rolled back using `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 may 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. 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 34.

```
xquery version "1.0-ml";

declare option xdmp:transaction-mode "update";

(: 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 :)

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

(: 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 26, multi-statement update transactions use locks. A multi-statement update transaction may 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 25. The system timestamp is determined when the query transaction is created, so all statements in the transaction see the same version of accessed documents.

3.4.2.2 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:transaction-mode "update";

(: 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:transaction-mode "update";

(: transaction created :)
xdmp:document-insert(“not-lost.xml”, <data/>)
, xdmp:commit();
fn: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:transaction-mode "update";

(: transaction created :)
xdmp:document-insert("lost.xml", <data/>)
, xdmp:commit()
, fn: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 may 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:transaction-mode "update";

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()
```

3.4.2.3 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`). 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:transaction-mode "update";
                                              (: begin transaction :)
xdmp:document-insert("/docs/mst.xml", <data/>);
xdmp:commit()
, "this expr is evaluated and committed";
                                              (: end transaction :)
                                              (:begin transaction :)
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 using `xdmp:rollback`. 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 may span multiple HTTP requests. For details, see “Sessions” on page 34.

3.4.2.4 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 mode. When using multi-statement transactions, you should understand when evaluation may occur under 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` 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 mode may apply to queries run in response to multiple HTTP requests. Uncommitted transactions remain open until the HTTP session times out, which may 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.

3.4.3 Semi-Colon as a Statement Separator

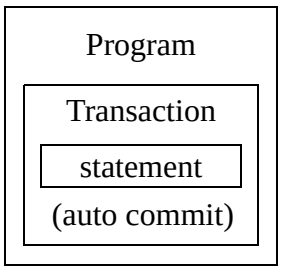
MarkLogic Server extends the XQuery language to include the semi-colon (;) in the XQuery body as a separator between statements. Statements are evaluated in the order in which they appear. Each semi-colon separated statement in a transaction is fully evaluated before the next statement begins.

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.

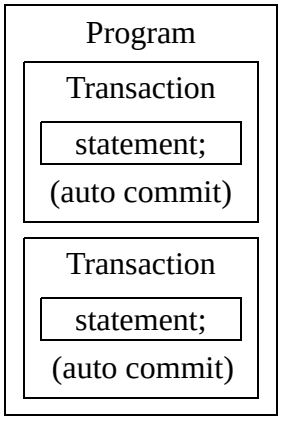
Semi-colon separated statements in `auto` transaction mode (the default) are not multi-statement transactions. Each statement is a single-statement transaction. If one update statement commits and the next one throws a runtime error, the first transaction is not rolled back. If you have logic that requires 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 209.

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.

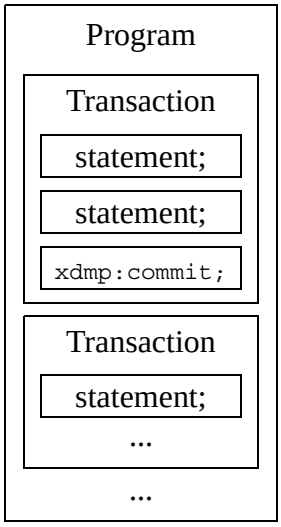
The following diagram contrasts the relationship between statements and transactions in single and multi-statement transactions:



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.

3.5 Transaction Mode

This section covers the following topics related to transaction mode:

- Transaction Mode Overview
- Controlling the Transaction Mode
- Auto Transaction Mode
- Query Transaction Mode
- Update Transaction Mode

3.5.1 Transaction Mode Overview

The transaction mode in effect when a transaction is created determines the transaction type. Thus, the mode determines what kinds of statement can run, when and how transactions are committed, and whether or not updates are visible across statement boundaries.

Most applications will use the default transaction mode, `auto`. In this mode, all transactions are single-statement transactions, as discussed in “Single-Statement, Automatically Committed Transactions” on page 29.

To use multi-statement transactions, you must explicitly set the transaction mode to `query` or `update`. Use `query` mode for read-only transactions. Use `update` mode for transactions that may perform updates. Selecting the appropriate mode allows the server to properly optimize your queries. For more information, see “Multi-Statement, Explicitly Committed Transactions” on page 30.

The transaction mode is only considered during transaction creation. Changing the mode has no effect on the current transaction. For details, see “Controlling the Transaction Mode” on page 37.

Explictly setting the transaction mode affects only the current session. Queries run under `xdmp:eval` 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 41.

3.5.2 Controlling the Transaction Mode

You need not set the transaction mode if your application uses single-statement transactions since the default mode is `auto`. If you need to use multi-statement transactions, then you must explicitly set the mode.

To explicitly set the transaction mode:

- declare the `xdmp:transaction-mode` option in the prolog of your program, or
- call `xdmp:set-transaction-mode` prior to creating transactions that should run in that mode.
- set the `transaction-mode` option in the options node passed to `xdmp:eval`, `xdmp:invoke`, or `xdmp:spawn`.

Use the xdmp:transaction-mode prolog option when you need to set the transaction mode before the first transaction is created.

Changing the transaction mode in the middle of a transaction does not affect the current transaction.

The following example demonstrates that `xdmp:set-transaction-mode` has no effect on the current transaction by using `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:transaction-mode "update";

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

```
xquery version "1.0-ml";

declare option xdmp:transaction-mode "update";

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

declare option xdmp:transaction-mode "query";

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

3.5.3 Auto Transaction Mode

The default transaction mode is `auto`. In this mode, all transactions are single-statement transactions. See "Single-Statement, Automatically Committed Transactions" on page 29.

Most 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.

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.

You may 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.

3.5.4 Query Transaction Mode

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 30.

In query transaction mode:

- Transactions may span multiple statements.
- The transaction is assumed to be read-only, so no locks are acquired. All statements in the transaction are executed 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. An error is raised 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 may 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();
```

3.5.5 Update Transaction Mode

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 30.

In update transaction mode:

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

Update transactions may 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 26.

3.6 Interactions with xdmp:eval/invoke

The `xdmp:eval` and `xdmp:invoke` functions allow you to start one transaction from the context of another. The `xdmp:eval` function submits a string to be evaluated and the `xdmp:invoke` function evaluates a stored module. You can control the semantics of `xdmp:eval` and `xdmp: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 Undetectable Deadlocks
- Seeing Updates From eval/invoke Later in the Transaction
- Running Multi-Statement Transactions under xdmp:eval/invoke

3.6.1 Isolation Option to xdmp:eval/invoke

The `xdmp:eval` and `xdmp:invoke` functions take an options node as the 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 `xdmp:eval` or `xdmp: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 `xdmp:eval` or `xdmp: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 when you call `xdmp:eval` or `xdmp: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 an undetectable deadlock situation (see "Preventing Undetectable Deadlocks" on page 42).

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.

3.6.2 Preventing Undetectable 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 detect. 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 an undetectable deadlock condition. These undetectable deadlocks can only occur in update statements; query statements will never cause a deadlock.

An undetectable 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 undetectable deadlock conditions can only be resolved by cancelling the query or letting the query time out.

To be completely safe, you can prevent these undetectable 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 an undetectable 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 an undetectable deadlock and you will have to restart MarkLogic Server 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 only way to clear the deadlock it to restart MarkLogic Server. 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.

3.6.3 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 an undetectable deadlock condition, as described in “Preventing Undetectable Deadlocks” on page 42), 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()
```

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

When you run a query using `xdmp:eval` or `xdmp:invoke` with `different-transaction` isolation, or via `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 `xdmp:eval/invoke` with `different-transaction` isolation or under `xdmp:spawn`,

- Set the transaction mode explicitly inside the eval/invoke'd query or set `transaction-mode` in the options node if the transaction should run as a multi-statement transaction.
- Always call `xdmp:commit` inside an eval/invoke'd multi-statement query if updates should be preserved.

Setting the transaction mode explicitly in the prolog of the eval/invoke'd query is equivalent to setting it by passing an options node to `xdmp:eval/invoke` with `transaction-mode` set. Setting the mode through the options mode enables you to set the transaction 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 51.

The same considerations apply to multi-statement queries evaluated using `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 49.

3.7 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.

There are several functions that evaluate asynchronously as soon as they are called, whether called from an update transaction or a query transaction. Examples of these functions are `xdmp:spawn`, `xdmp:http-get`, and `xdmp:log`. These functions have side effects outside the scope of the calling statement or the containing transaction (*non-transactional* side effects).

When evaluating an XQuery 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` 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` 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 `xdmp:spawn` call evaluates asyncronously as soon as it is called. Similarly, if you are calling a web service with `xdmp:http-get` 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 209 and the *Content Processing Framework Guide* manual.

3.8 Reducing Blocking with Multi-Version Concurrency Control

You may 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 25.

Nonblocking mode may be useful for your application if:

- Low query latency is more important than update latency.
- Your application participates in XA transactions. XA transactions may involve multiple participants and non-MarkLogic Server resources, so they may 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 may 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 may not see the most up to date results.

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

3.9 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 may run on remote hosts. If a remote transaction does not terminate normally, it may 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 may 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 52. For details on the built-ins, see the *XQuery & XSLT API Reference*.

3.10 Transaction Examples

This section includes the following examples:

- Example: Incrementing the System Timestamp
- Example: Multi-statement Transactions and Same-statement Isolation
- Example: Multi-Statement Transactions and Different-transaction Isolation

3.10.1 Example: Incrementing the System Timestamp

The following example shows how updates to the system increment the system timestamp. For background on system timestamps, see “System Timestamps and Fragment Versioning” on page 25.

```
<results>{
  <start-of-query>{xdmp:request-timestamp()}</start-of-query>,
for $x in (1 to 10)
return
(xdmp:eval("xdmp:document-insert('test.xml', <test/>)"),
 xdmp:eval("xdmp:document-delete('test.xml')" ) ),
<end-of-query>{xdmp:eval('xdmp:request-timestamp()')}</end-of-query>,
  <number-of-transactions>{xdmp:eval('xdmp:request-timestamp()') -
  xdmp:request-timestamp()}</number-of-transactions>,
<elapsed-time>{xdmp:query-meters()/*:elapsed-time/text()}
  </elapsed-time>
}
</results>
```

This XQuery program executes as a query statement, which runs at a timestamp. Each of the `xdmp:eval` statements is evaluated as a separate transaction, and they are each update statements. The query starts off by printing out the system timestamp, then it starts a FLWOR expression which evaluates 10 times. Each iteration of the FLWOR creates a document called `test.xml` in a separate transaction, then deletes the document in another transaction. Finally, it calculates the number of transactions that have occurred since the query started by starting a new transaction to get the current system timestamp and subtracting that from the timestamp in which the query runs. Therefore, this query produces the following results, which show that 20 transactions have occurred.

```
<results>
  <start-of-query>382638</start-of-query>
  <end-of-query>382658</end-of-query>
  <number-of-transactions>20</number-of-transactions>
  <elapsed-time>PT0.102211S</elapsed-time>
</results>
```

Note: This example executes 20 update statements with `xdmp:eval`, each of which starts its own transaction, and each of which increments the system timestamp by 1. If there are other transactions happening concurrently on the system while these transactions are executing, those transactions might also increment the system timestamp, and it is possible that the last `xdmp:eval` statement which executes the `xdmp:request-timestamp` function might return a number more that 20 greater than the first call to `xdmp:request-timestamp`.

3.10.2 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 41.

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 219.

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.

3.10.3 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 36 and “Interactions with xdmp:eval/invoke” on page 41.

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 25.

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 26.

3.10.4 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 an undetectable 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 may 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 61.
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 61.

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 57, 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*.

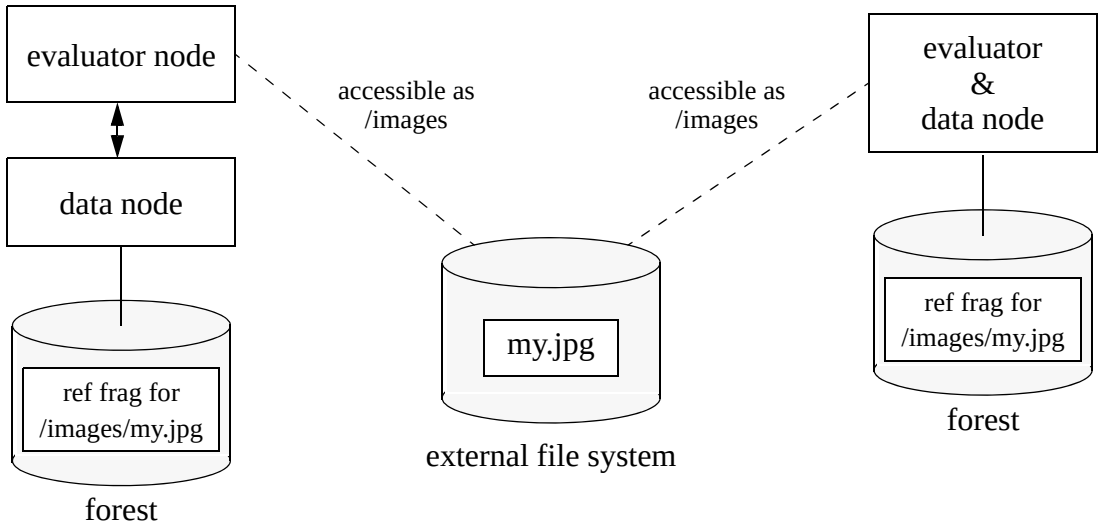
4.3.3 Selecting a Location For Binary Content

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 105.

MarkLogic Server server offers two features to assist with adding metadata to binary documents: The “Filter Documents” transformer in Information Studio, and the XQuery built-in `xdmp:document-filter`.

The “Filter Documents” transformer automatically extracts text and metadata for certain types of documents and attaches them as document properties. For details, see Extracting Metadata from Binary Content With the Filter Documents Transform in the *Information Studio Developer’s Guide*.

The built-in `xdmp:document-filter` extracts 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 with `xdmp:document-filter` 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.

For example, if the App Server receives a GET request from an HTTP client which includes the following and /images/really_big.jpg is stored in the App Server modules database:

```
GET /images/really_big.jpg HTTP/1.1
Range: bytes=0-499
```

Then the App Server 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($doc)-1
                else xs:integer($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 modules from other XQuery modules in MarkLogic Server. 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. It includes the following sections:

- 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 save this module to a file named `c:/code/helloworld.xqy`, and you have an App Server with filesystem root `c:/code`, you can import this in either a main module or a library module and call the function as follows:

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

hw:helloworld()
```

This XQuery program will import the library module with the `helloworld` namespace and then evaluate its `helloworld()` function.

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>` instruction.

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).

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, 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

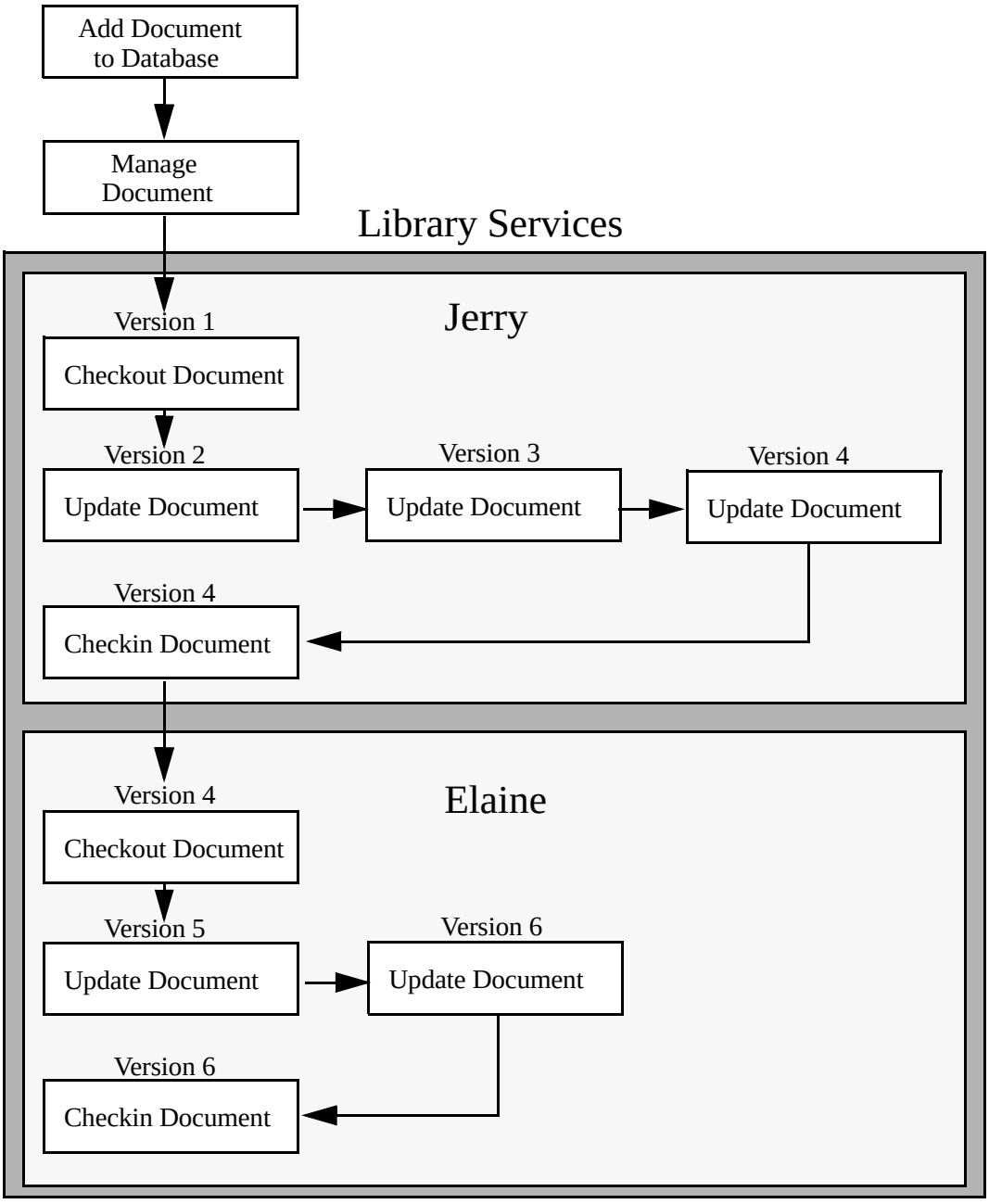
The Library Services enable you to create and maintain versions of managed documents in MarkLogic Server. Access to managed documents is controlled using a check-out/check-in model. You must first check out a managed document before you can perform any update operations on the document. A checked out 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 80.

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 87.

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.



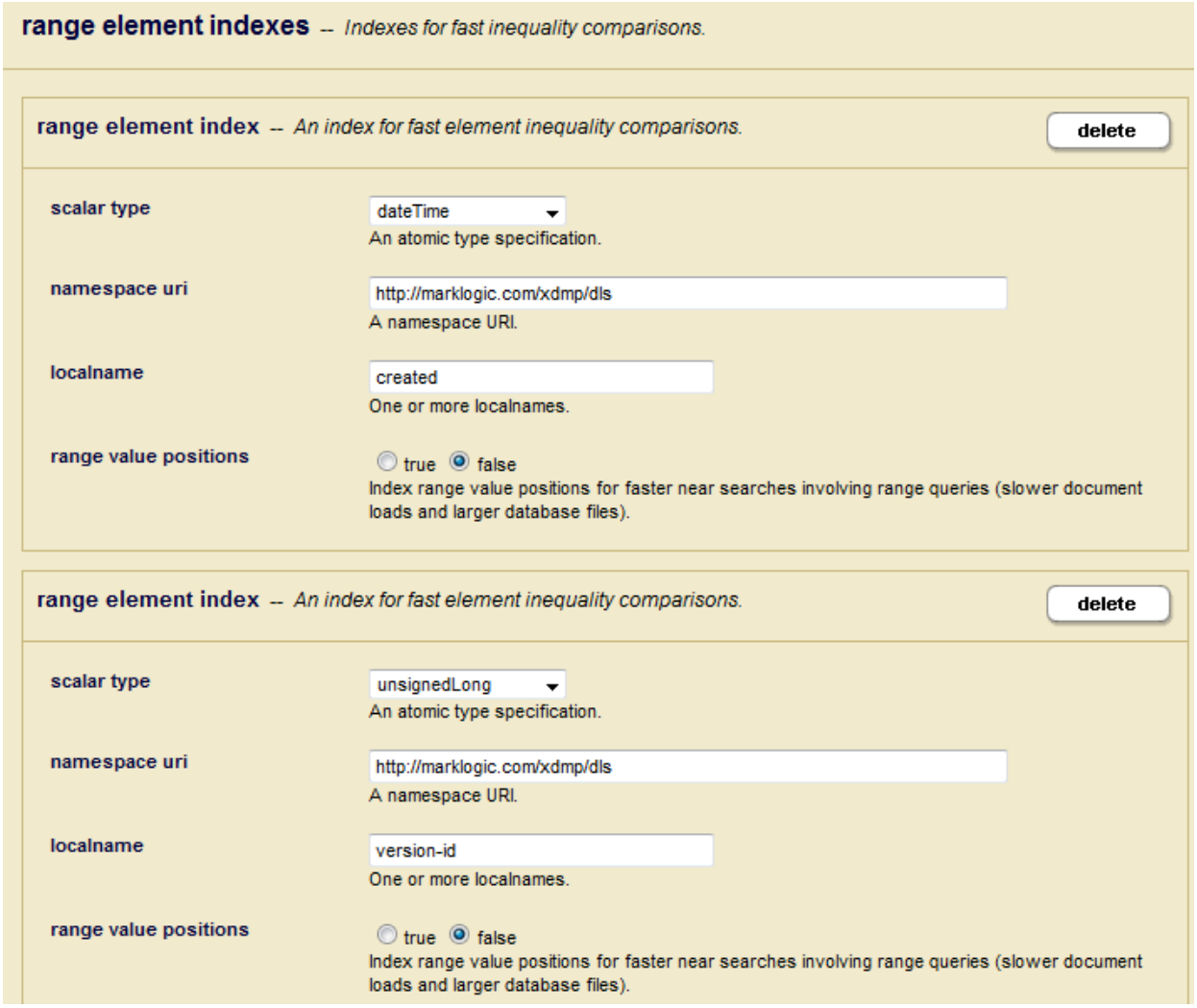
6.2 Building Applications with Library Services

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	Localname	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 77, "Checking Out Managed Documents" on page 78 and "Checking In Managed Documents" on page 79.
- Document update functions for updating the content of documents and their properties. For usage information, see "Updating Managed Documents" on page 79 and "Managed Document Update Wrapper Functions" on page 75.
- Retention policy functions for managing when particular document versions are purged. For usage information, see "Defining a Retention Policy" on page 80.
- XInclude functions for creating and managing linked documents. For usage information, see "Managing Modular Documents in Library Services" on page 87.
- `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 80.

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 80.

> **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 93.

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 146, 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 146, 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 87, 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>
```

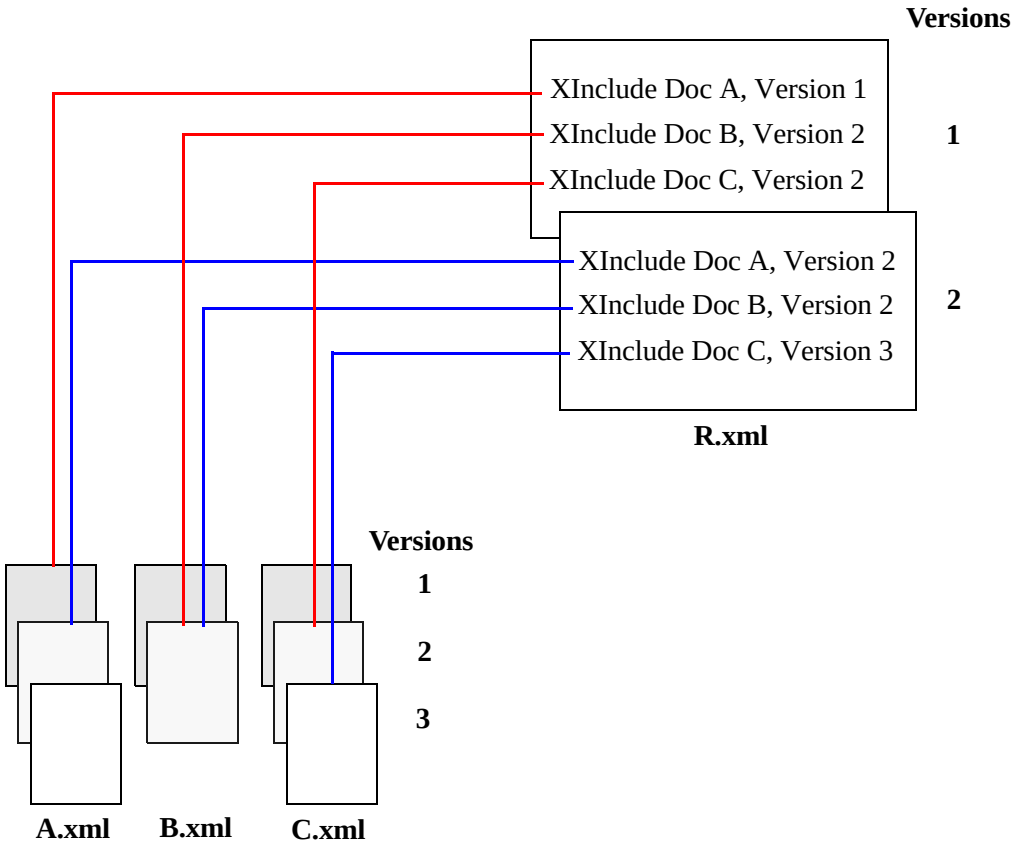
Note: When using the `dls:node-expand` function to expand modular documents that contain XInclude links to specific versioned documents, specify the `$restriction` parameter as an empty sequence.

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 node back into the original function.

```
xquery version "1.0-ml";
(: This function takes the children of the node and passes them
   back into the typeswitch function.  :)
declare function local:passthru($x as node()) as node()*
{
for $z in $x/node() return local:dispatch($z)
};

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

let $x :=
<foo>foo
  <bar>bar</bar>
  <baz>baz
    <buzz>buzz</buzz>
  </baz>
  foo
</foo>
return
local:dispatch($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 function takes the children of the node and passes them
   back into the typeswitch function.  :)
declare function local:passthru($x as node()) as node()*
{
for $z in $x/node() return local:dispatch($z)
};

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

let $x :=
<foo>foo
  <bar>bar</bar>
  <baz>baz
    <buzz>buzz</buzz>
  </baz>
  foo
</foo>
return
cts:highlight(local:dispatch($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 function takes the children of the node and passes them
   back into the typeswitch function.  :)
declare function local:passthru($x as node()) as node()*
{
for $z in $x/node() return local:dispatch($z)
};

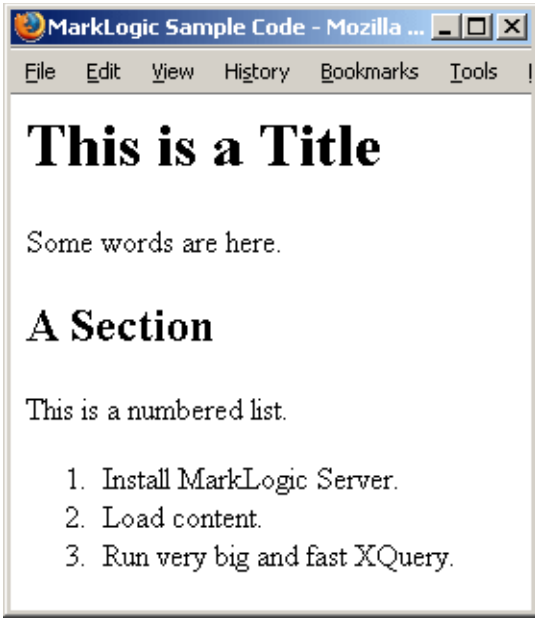
(: This is the recursive typeswitch function :)
declare function local:dispatch($x as node()) as node()*
{
typeswitch ($x)
  case text() return $x
  case element (a) return local:passthru($x)
  case element (title) return <h1>{local:passthru($x)}</h1>
  case element (para) return <p>{local:passthru($x)}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

let $x :=
<a>
 <title>This is a Title</title>
 <para>Some words are here.</para>
 <sectionTitle>A Section</sectionTitle>
 <para>This is a numbered list.</para>
 <numbered>
   <number>Install MarkLogic Server.</number>
   <number>Load content.</number>
   <number>Run very big and fast XQuery.</number>
 </numbered>
</a>
return
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>MarkLogic Sample Code</title></head>
<body>{local:dispatch($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 `dispatch` 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 dispatch($x as node(), $excluded as element()*)
  as node()*
{
(: Test whether $x is an excluded element, if so return empty,
   otherwise run the typeswitch expression.
:)
if ( some $node in $excluded satisfies $x )
then ( )
else ( typeswitch ($x) ..... )
};
```

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

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 102.

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 104.

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 110.

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 105. 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 108).

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 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 *Understanding and Using 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 17.

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.

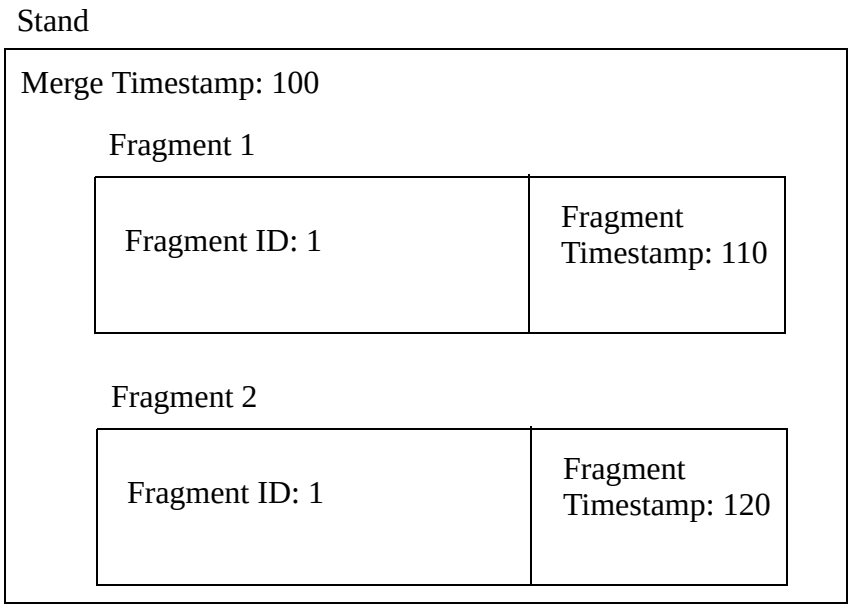
There is a control at the database level called the `merge timestamp`, set via the Admin Interface. By default, the `merge timestamp` is set to 0, which sets the timestamp of a merge to the timestamp corresponding to when the merge completes. To use point-in-time queries, you can set the `merge timestamp` 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 122.

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 17.

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 *not* reflected in the query.

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

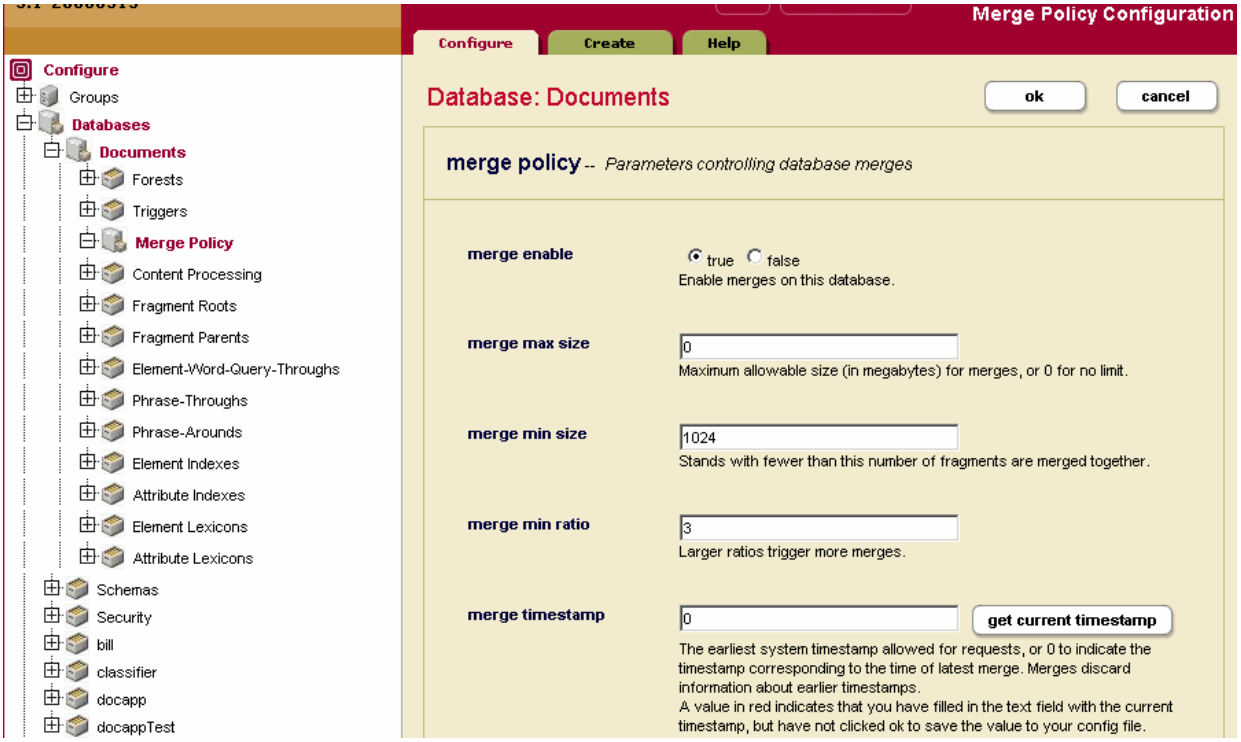
By default, query statements are run at the system timestamp in effect when the statement initiates. To run a query statement at a different system timestamp, you must set up your system to store older versions of documents and then specify the timestamp when you issue a point-in-time query statement. 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 127.

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 122) 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 *Understanding and Using 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 or .NET level.

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 126.

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 17.

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 127.
- 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 120.

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 120.

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
- Information Studio Plugins
- 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 Server 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 an Enterprise Edition 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 132, 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. For example, application plugins are used in Information Studio, and these application plugins enable developers to extend the functionality of Information Studio with their own customized functionality. 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. For details on using plugins with Information Studio to create custom collectors and transformers, see Creating Custom Collectors and Transforms in the *Information Studio Developer's Guide*.

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. Information Studio uses this mechanism in its user interface to offer collectors and transformers. 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 Information Studio Plugins

Information Studio uses application plugins for its collectors and transformers, but does not put them in the `<marklogic-dir>/Plugins` directory. Instead, it puts them in the `<marklogic-dir>/Assets/plugins/marklogic/appservices` directory. In the directory that Information Studio uses, it is possible to create your own plugins to extend Information Studio or for use outside of Information Studio. You need to initialize those plugins with the `plugin:initialize-scope` function, which specifies the path within the `Assets/plugins/marklogic/appservices` directory to find the XQuery modules. For more details, see the Plugin Module API documentation in the *MarkLogic XQuery and XSLT Function Reference* and the Creating Custom Collectors and Transforms *Information Studio Developer's Guide*.

For information about the differect between application plugins and system plugins,see "System Plugins versus Application Plugins" on page 133.

11.4 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

11.4.1 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.

11.4.2 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 on 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 an Enterprise Edition 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
- 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 to 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 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

12.5.1 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”.

12.5.2 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 :)
```

12.5.3 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.

12.5.4 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>
```

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 ad 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 17.

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 148.

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).

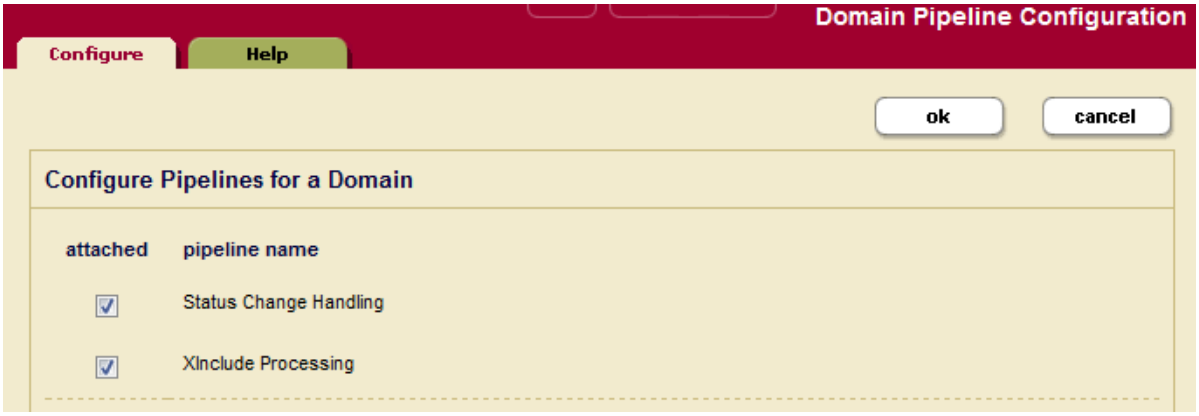
2. Click the domains link from the left tree menu. Either create a new domain or modify an existing domain to encompass the scope of the documents you want processed with the XInclude processing. For details on domains, see the *Content Processing Framework Guide* guide.

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 149.

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
- 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 Pages
- Error XML Format
- Configuring Custom Error Pages
- Execute Permissions Are Needed On Error Handler Document for Modules Databases
- Example of Custom Error Pages

15.1.1 Overview of Custom HTTP Error Pages

A custom HTTP Server error page is a way to redirect application exceptions to an XQuery program. When any 400 or 500 HTTP exception is thrown (except for a 503 error), an XQuery module 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 an XQuery module, you can make it perform arbitrary work.

The XQuery module can get the HTTP error code and the contents of the HTTP response using the `xdmp:get-response-code` API. The XQuery module for the error handler also has access to the XQuery stack trace, if there is one; the XQuery stack trace is passed to the module as an external variable with the name `$error:errors` in the XQuery `1.0-ml` dialect and as `$err:errors` in the XQuery `0.9-ml` dialect (they are both bound to the same namespace, but the `err` prefix is predefined in `0.9-ml` and `error` prefix is predefined in `1.0-ml`).

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 page itself throws an exception, that exception is passed to the client with the error code from the error page. It will also include a stack trace that includes the original error code and exception.

15.1.2 Error XML Format

Error messages are thrown with an XML error stack trace that uses the `error.xsd` schema. Stack trace includes any exceptions thrown, line numbers, and XQuery Version. Stack trace is accessible from custom error pages through the `$error:errors` external variable. The following is a sample error XML output for an XQuery module with a syntax error:

```
<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>
```

15.1.3 Configuring Custom Error Pages

To configure a custom error page for an HTTP App Server, enter the name of the XQuery 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 66.

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 XQuery code 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 of Custom Error Pages

The following XQuery module is an extremely simple XQuery error handler.

```
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
```

This simply returns all of the information from the page that throws the exception. In a typical error page, you would use some or all of the information and make a user-friendly representation of it to display to the users. Because you can write arbitrary XQuery in the error page, you can do a wide variety of things, including sending an email to the application administrator, redirecting it to a different page, and so on.

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 “Using the REST Library to Develop RESTful Web Services” on page 172.

This section includes the following topics:

- Overview of URL Rewriting
- Loading the Shakespeare XML Content
- A Simple URL Rewriter
- 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, only you write an XQuery 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 164.

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 157.

15.2.2 Loading the Shakespeare XML Content

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 Content Source to `Documents`.
- Copy the query below into a Query Console window.
- Click `Run` to run the query.

The following query loads the current database with the XML files obtained 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.

15.2.3 A Simple URL Rewriter

The simplest way to rewrite a URL is to create a URL rewrite script that reads the external URL given to the server by the browser and converts it to the raw URL recognized by the server.

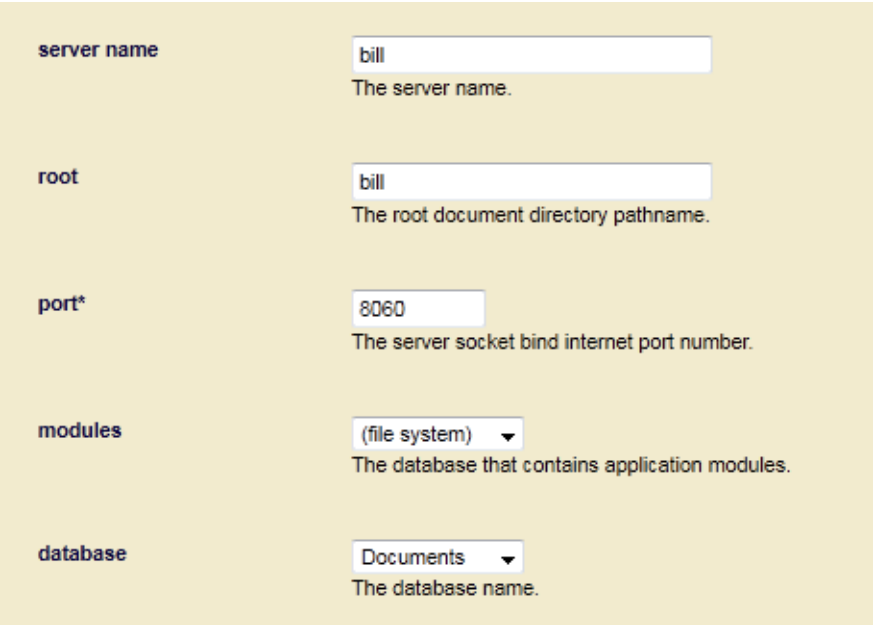
The following procedure describes how to set up your MarkLogic Server for the simple URL rewriter example described in this section.

1. In the Admin Interface, click the Groups icon in the left frame.

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`.

6. Create a simple module, named `mac.xqy`, that uses the `fn:doc` function to call the `macbeth.xml` file in the database:

```
xdmp:set-response-content-type("text/html")
fn:doc("macbeth.xml")
```

7. Save `mac.xqy` in the `/<MarkLogic_Root>/bill` directory.

8. Open a browser and enter the following URL to view `macbeth.xml` (in raw XML format):

```
http://localhost:8060/mac.xqy
```

A "cleaner," more descriptive URL would be something like:

```
http://localhost:8060/macbeth
```

To accomplish this URL rewrite, do the following:

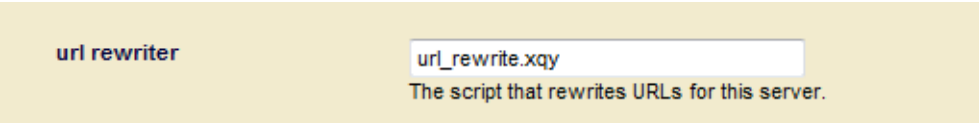
1. Create script named `url_rewrite.xqy` that uses the `xdmp:get-request-url` function to read the URL given by the user and the `fn:replace` function to convert the `/macbeth` portion of the URL to `/mac.xqy`:

```
xquery version "1.0-ml";

let $url := xdmp:get-request-url()
return fn:replace($url, "^/macbeth$", "/mac.xqy")
```

2. Save the `url_rewrite.xqy` script in the `/<MarkLogic_Root>/bill` directory.

3. In the Admin Interface, open the `bill` App Server and specify `url_rewrite.xqy` in the `url rewriter` field:



4. Enter the following URL in your browser:

```
http://localhost:8060/macbeth
```

   Note: Though the URL is converted by the `fn:replace` function to `/mac.xqy`, `/Macbeth` is displayed in the browser's URL field after the page is opened.

The `xdmp:get-request-url` function returns the portion of the URL following the scheme and network location (domain name or *host_name*:*port_number*). In the above example, `xdmp:get-request-url` returns `/Macbeth`. Unlike, `xdmp:get-request-path`, which returns only the request path (without any parameters), the `xdmp:get-request-url` function returns the request path and any query parameters (request fields) in the URL, all of which can be modified by your URL rewrite script.

You can create more elaborate URL rewrite modules, as described in "Creating URL Rewrite Modules" on page 163 and "Using the REST Library to Develop RESTful Web Services" on page 172.

15.2.4 Creating URL Rewrite Modules

This section describes how to create simple URL rewrite modules. For more robust URL rewriting solutions, see "Using the REST Library to Develop RESTful Web Services" on page 172.

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 `/mac.xqy`:

```
xquery version "1.0-ml";

let $url := xdmp:get-request-url()
return fn:replace($url,"^/$", "/mac.xqy")
```

In this example, you hide the `.xqy` extension from the browser's address bar and convert a static URL into a dynamic URL (containing a `?` character), you could do something like:

```
let $url := xdmp:get-request-url()

return fn:replace($url,
      "^/product-([0-9]+)\.html$",
      "/product.xqy?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:

```
let $url := xdmp:get-request-url()

return fn:replace($url,
      "^/product/([a-zA-Z0-9_-]+)/([0-9]+)\.html$",
      "/product.xqy?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:

```
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
```

15.2.5 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:

```
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")
```

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` or some other scheme that is guaranteed to generate non-existent URLs.

15.2.6 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:

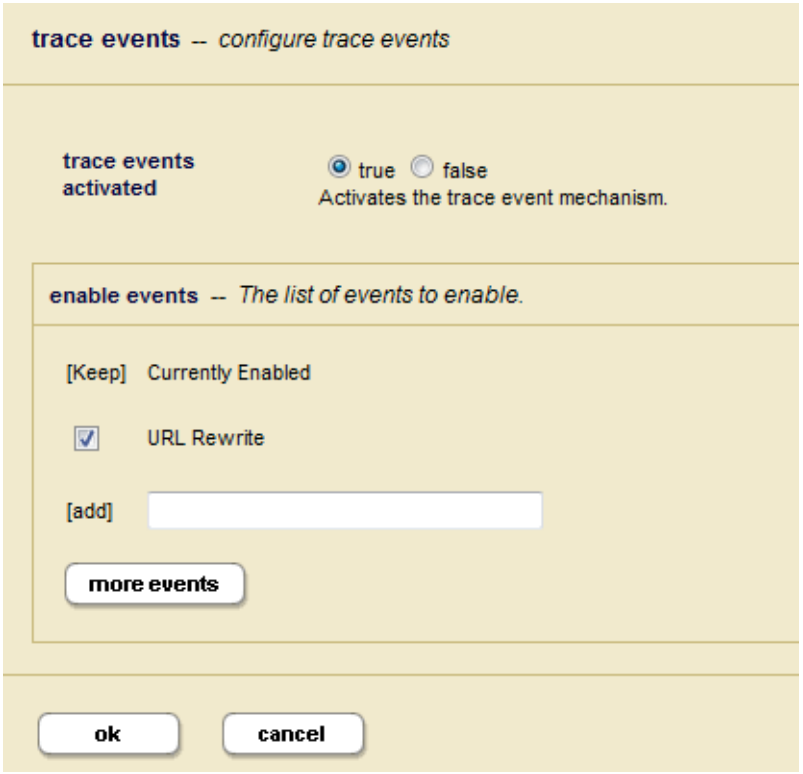
```
<frame src="http://127.0.0.1:8000/myapp/top.html" name="headerFrame">
```

15.2.7 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 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

15.3.1 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.

15.3.2 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.

15.3.3 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.

15.4 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

15.4.1 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.

15.4.2 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`

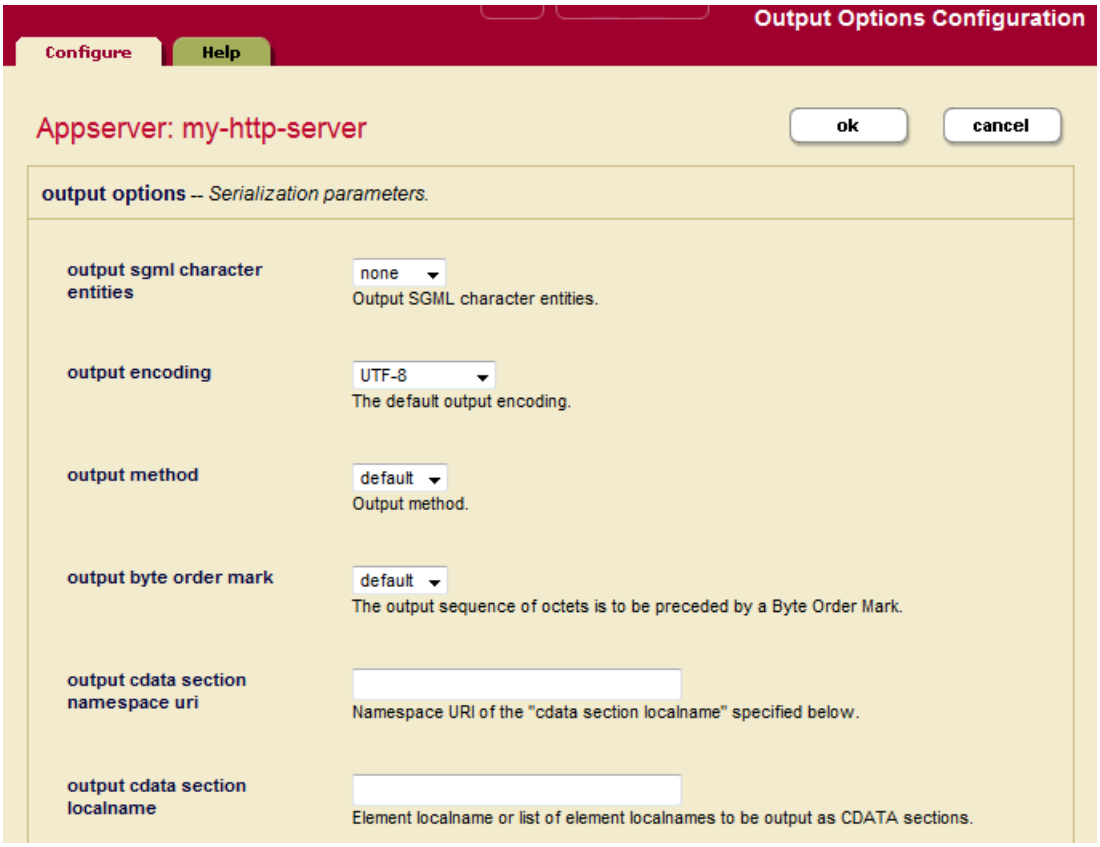
Additionally, you can specify the output encoding for XML output in an XQuery program using the `<output-encoding>` option to the following XML-serializing APIs:

- `xdmp:quote`
- `xdmp:save`

For details, see the *MarkLogic XQuery and XSLT Function Reference* for these functions.

15.5 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 Using the REST Library to Develop RESTful 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 have loaded the Shakespeare XML content, as described in "Loading the Shakespeare XML Content" on page 159, and configured the "bill" App Server and directory, as described in "A Simple URL Rewriter" on page 161.

The topics in this section are:

- Terms Used in this Chapter
- Overview of the REST Library
- A Simple Rewriter and Endpoint
- 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* is an XQuery module that interprets the URL of the incoming request and rewrites it to an internal URL that services the request.
- 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.

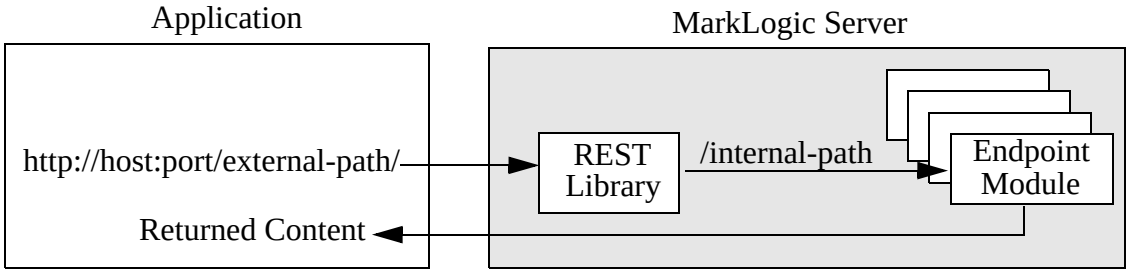
The REST library does the following:

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 REST Library is documented in the *XQuery and XSLT Reference Guide*. For additional information on URL rewriting, see “Setting Up URL Rewriting for an HTTP App Server” on page 158.

16.3 A Simple 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 “A Simple URL Rewriter” on page 161:

```
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 180, 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 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.

Element	Attributes	Child Elements	Number of Children	For More Information

`options`	`user-params = ignore`
	`| allow`
	`| forbid`
		`request`	`0..n`	"A Simple Rewriter and Endpoint" on page 174

`request`	`uri=`*`string`*
	
	`endpoint=`*`string`*
	
	`user-params = ignore`
	`| allow`
	`| 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 Rewriter and Endpoint" on page 174
				
				"Extracting Multiple Components from a URL" on page 180
				
				"Adding Conditions" on page 193

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 180 "Defining Parameters" on page 188

`param`	`name=`*`string`* `(required)`
	
	`as=`*`string`*
	
	`values=`*`string`*
	
	`match=`*`string`*
	
	`default=`*`string`*
	
	`required = true`
	`| false`
	
	`repeatable = true`
	`| false`
				"Defining Parameters" on page 188
				
				"Supporting Parameters Specified in a URL" on page 189.

`http`	`method = `*`string`*
	`(required)`
	
	`user-params = ignore`
	`| allow`
	`| 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 184
				
				"Adding Conditions" on page 193.

`auth`		`privilege`
		
		`kind`
			`0..n`
			
			`0..n`
				"Authentication Condition" on page 194

`function`	`ns=`*`string`* `(required)`
	
	`apply=`*`string`*
	`(required)`
	
	`at=`*`string`* `(required)`
				"Function Condition" on page 195

16.5 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 Rewriter and Endpoint" on page 174, 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 Rewriter and Endpoint" on page 174, 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.

16.6 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 Rewriter and Endpoint" on page 174 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 189.

```
<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 Rewriter and Endpoint" on page 174, 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]
```

16.7 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.

16.8 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 180.

```
<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.

16.9 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

16.9.1 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.

16.9.2 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.

16.10 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

16.10.1 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 180, 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/

16.10.2 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.

16.10.3 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>
```

16.10.4 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>
```

16.10.5 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"/>
```

16.10.6 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.

16.10.7 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"/
```

16.10.8 Matching Regular Expressions in Parameters

As shown in “Supporting Parameters Specified in a URL” on page 189, 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. This section goes into more detail on the use of the `match` attribute.

The `match` attribute in the `param` element defines a subexpression, 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 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

16.11.1 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 187 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`.

16.11.2 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>
```

16.11.3 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>
```

16.11.4 Function Condition

The `function` condition gives you the ability to test for arbitrary conditions. By specifying the namespace, localname, 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.

16.11.5 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.

16.11.6 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.

17.0 Working With JSON

This chapter describes how to work with JSON in MarkLogic Server, and includes the following sections:

- JSON, XML, and MarkLogic
- Converting JSON to XML and XML to JSON
- Low-Level JSON APIs and Primitive Types
- Examples

17.1 JSON, XML, and MarkLogic

JSON (JavaScript Object Notation) is a data-interchange format which is designed to pass data to and from JavaScript. It is very useful in a web application to pass data back and forth between a server (such as MarkLogic Server) and JavaScript, and JSON is a popular mechanism for such data interchange. JSON, like XML, is designed so it is both machine- and human-readable. For more details about JSON, see json.org.

MarkLogic Server includes advanced support for JSON. You can use JSON to store key/value documents or just 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:

- The REST APIs all have options to return data as JSON, making it easy for client developers to interact with data from MarkLogic.
- It is easy to 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 REST and Java client APIs provide full support for loading and querying JSON documents, where the JSON documents are automatically stored as XML and retrieved as JSON. This allows for fine-grained access to the JSON documents, as well as the ability to search and facet on JSON content.
- There are low-level XQuery APIs and XQuery types to support being able to store and model JSON arrays as XML, translate back and forth between XQuery and JSON types, and so on. The low-level APIs are built-in to MarkLogic Server, allowing for extremely fast transformations.
- There is a higher-level XQuery library, allowing for both simple and complex transformations to and from JSON and XML. The XQuery library makes use of the lower-level APIs where appropriate, abstracting their use from the high-level API.

17.2 Converting JSON to XML and XML to JSON

The core of the JSON support in MarkLogic is being able 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:

- Loading JSON Documents Using the REST or Java Client APIs
- Using the XQuery Library to Convert between JSON and XML

17.2.1 Loading JSON Documents Using the REST or Java Client APIs

If you are using the REST or Java Client APIs, there is no need to write any XQuery (and no need to understand any of the mechanics of how the conversion occurs). The REST and Java APIs both allow you to specify JSON as the document type during a load, and the API takes care of any needed conversion to or from JSON or XML. For details on the REST Client API, see REST Application Developer's Guide. For details on the Java Client API, see Java Application Developer's Guide.

Note: Element names in the XML representation of a JSON document inserted using the REST or Java APIs are in the namespace "http://marklogic.com/xdmp/json/basic". You must use this namespace when defining indexes on keys in JSON documents.

17.2.2 Using the XQuery Library to Convert between JSON and XML

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 APIs and Primitive Types" on page 199. This section describes how to use the XQuery library and includes the following parts:

- Conversion Philosophy
- Converting From JSON to XML
- Converting From XML to JSON
- Understanding the Configuration Strategies For Custom Transformations

17.2.2.1 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.

17.2.2.2 Converting From JSON to XML

The main function to convert from JSON to XML is:

- `json:transform-from-json`

For examples, see “Conversion To and From JSON and XML Examples” on page 201.

17.2.2.3 Converting From XML to JSON

The main function to convert from XML to JSON is:

- `json:transform-to-json`

For examples, see “Conversion To and From JSON and XML Examples” on page 201.

17.2.2.4 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`
- `json:check-config`

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*.

17.3 Low-Level JSON 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 uses these lower-level APIs, and if you use the JSON library module, you will likely not need to use the low-level APIs.

There are two APIs devoted to serialization of the key/value pairs: 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 197.

For the signatures and description of each function, see the *MarkLogic XQuery and XSLT Function Reference*.

17.4 Examples

This section includes example code that use the JSON functions and includes the following examples:

- Conversion To and From JSON and XML Examples
- Low-Level API Examples
- Loading JSON Using the REST or Java Client API

17.4.1 Conversion To and From JSON and XML Examples

The following examples show different techniques for converting JSON to XML and XML to JSON:

- From JSON and Back to XML—Basic Strategy
- To JSON and Back to XML—Full Strategy
- To JSON and Back to XML—Custom Strategy

17.4.1.1 From JSON and Back to XML—Basic Strategy

The following uses the `basic` (which is the default) strategy for transforming a JSON string to XML and then back to JSON.

```
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":[]}
```

17.4.1.2 To JSON and Back to XML—Full Strategy

The following uses the `full` strategy for transforming a XML to a JSON string. The full strategy outputs a JSON string with keys named in a consistent way.

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

declare variable $doc := document {
<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> } ;

let $c := json:config("full") ,
    $x := map:put($c,"whitespace" , "ignore" ),
    $j := json:transform-to-json(  $doc ,$c ),
    $xj := json:transform-from-json($j,$c)
return ($j, $xj)

{"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>
```

17.4.1.3 To JSON and Back to XML—Custom Strategy

The following uses the custom strategy to carefully control both directions of the conversion. The REST Client API uses a similar approach to transform options nodes back and forth between XML and JSON.

```
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") ,
    $cx := map:put( $c, "whitespace"     , "ignore" ),
    $cx := map:put( $c, "array-element-names" ,
                    xs:QName("search:bucket") ),
    $cx := map:put( $c, "attribute-names",
                     ("facet","type","ge","lt","name","ns" ) ),
    $cx := map:put( $c, "text-value", "label" ),
    $cx := map:put( $c , "camel-case", fn:true() ),
    $j := json:transform-to-json(  $doc ,$c ) ,
    $x := json:transform-from-json($j,$c)
return ($j, $x)

=>
{"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>
  <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>
```

17.4.2 Low-Level API Examples

The following examples use the low-level JSON APIs:

- Serializing to a JSON String
- Parsing a JSON String into a List of Items

17.4.2.1 Serializing to a JSON String

The following code returns a JSON string that includes a map, a string, and an integer.

```
let $map := map:map()
let $put := map:put($map, "some-key", 45683)
let $string := "this is a string"
let $int := 123
return
xdmp:to-json(($map, $string, $int))

(:
returns:
[{"some-key":45683}, "this is a string", 123]
:)
```

For details on maps, see “Using the map Functions to Create Name-Value Maps” on page 138.

17.4.2.2 Parsing a JSON String into a List of Items

Consider the following, which is the inverse of the previous example:

```
let $json :=
  '[{"some-key":45683}, "this is a string", 123]'
return
xdmp:from-json($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-key">
    <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 XQuery item. You can also use some or all of the items in the XQuery 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 :=
  '[{"some-key":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-key">
    <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-key":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 138.

17.4.3 Loading JSON Using the REST or Java Client API

With the REST Client API and the Java Client API, you can load JSON documents and they are automatically converted into XML for storage in MarkLogic Server. The implementation of the REST Client API uses the JSON library described in this chapter, and the Java Client API is built on top of the REST API, so it also indirectly uses the JSON library.

The following example shows how to use the REST Client API to load a JSON document in MarkLogic; you can similarly load JSON documents using the Java Client API:

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
```

Within the MarkLogic Server database, the endpoint automatically converts the JSON to XML and saves the following document with the URI `/test/keys.json`:

```
<?xml version="1.0" encoding="UTF-8"?>
<json type="object" xmlns="http://marklogic.com/xdmp/json/basic">
  <key1 type="string">value1</key1>
  <key2 type="object">
    <a type="string">value2a</a>
    <b type="string">value2b</b>
  </key2>
</json>
```

You can then query 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. For details on the Java Client API, see Java Application Developer's Guide.

18.0 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)

18.1 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 211.

18.1.1 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.

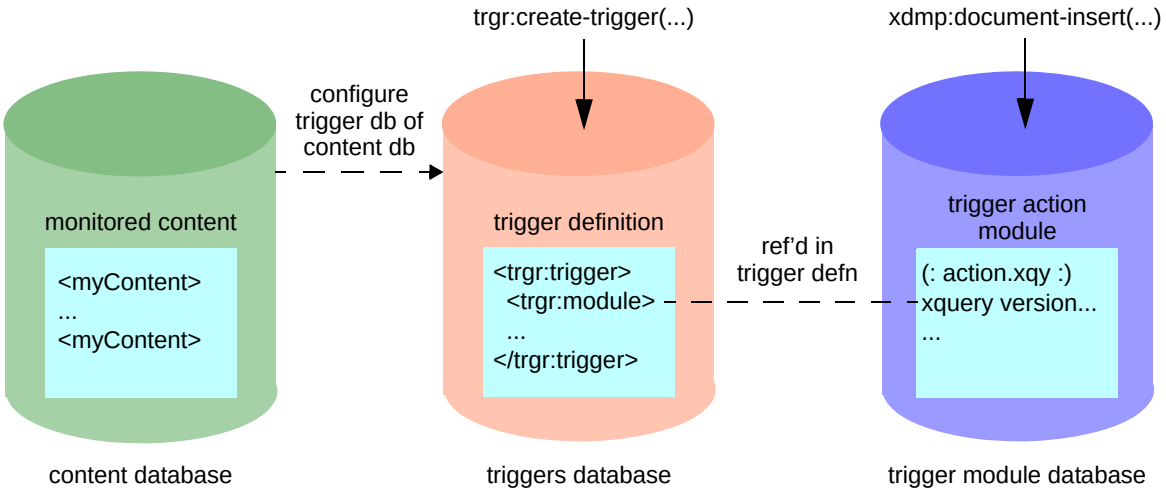
A trigger definition is created and installed by calling `trgr:create-trigger`. To learn more about trigger event definitions, see “Trigger Events” on page 213.

18.1.2 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.

18.2 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.

18.3 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

18.3.1 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.

18.3.2 Post-Commit Triggers

The module spawned as the result of a post-commit trigger is evaluated as a separate transaction from the module that produced the triggering event. It executes asynchronously, and if the trigger module fails, it does not roll back the calling transaction. Furthermore, there is no guarantee that the trigger module will complete if it is called.

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.

18.4 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

18.4.1 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.

18.4.2 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 214.
- 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 212.

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 212.

18.5 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.

18.6 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

18.6.1 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.

18.6.2 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;
```

18.7 Creating and Managing Triggers With triggers.xqy

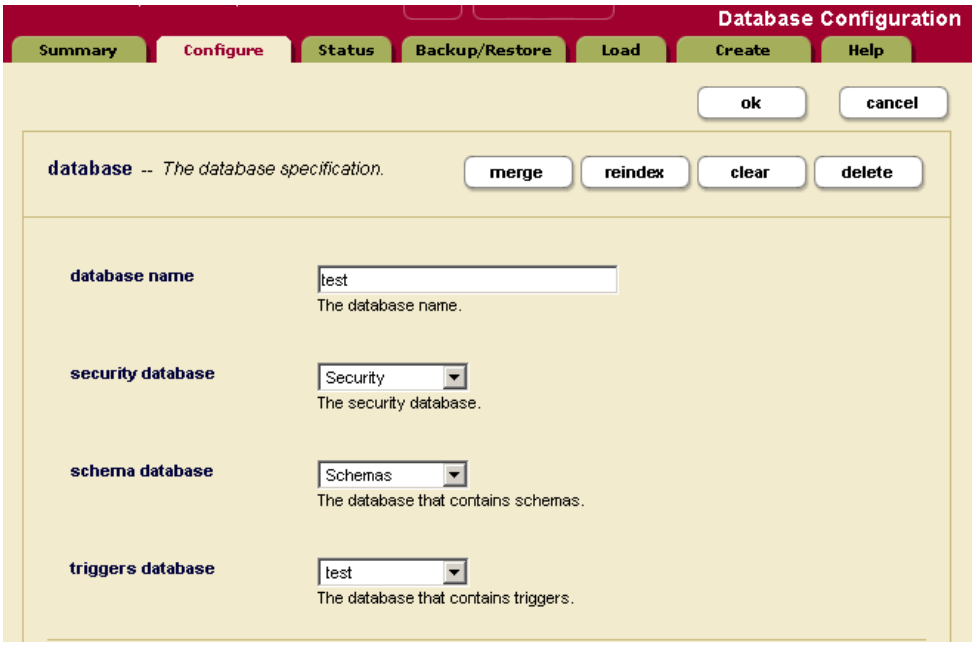
The `<install_dir>/Modules/MarkLogic/triggers.xqy` XQuery module file contains functions to create, delete, and manage triggers. If you are using the Status Change Handling pipeline, the pipeline takes care of all of the trigger details; you do not need to create or manage any triggers. For details on 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 217. The functions in this module are used to create the needed triggers when you use the Admin Interface to create a domain.

18.8 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 test used as both the database for the content and the triggers.



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("test"), "/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 `test` (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 `test` 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 (test 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 `test` 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"`.

18.9 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.

19.0 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
- Implementing Native Plugin Libraries

19.1 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.

The 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 223.

This chapter covers implementing, building, packaging, and installing an aggregate UDF. For information on using aggregate UDFs, see Using Aggregate User-Defined Functions in the *Search Developer’s Guide*.

19.2 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*.

19.2.1 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 223.
- 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*.

19.2.2 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.

19.3 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*. To learn more about native plugins, see "Implementing Native Plugin Libraries" on page 237.

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

19.3.1 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 223.

> **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 "Implementing Native Plugin Libraries" on page 237.

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 241.

3. Package your implementation into a native plugin. See "Packaging a Native Plugin" on page 239.

4. Install the plugin by calling the XQuery function `plugin:install-from-zip`. See "Installing a Native Plugin" on page 240.

A complete example is available in *marklogic_dir*`/Samples/NativePlugins`. You should use the sample Makefile as the basis for building your plugin. For details, see "Building a Native Plugin Library" on page 238.

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.

19.3.2 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 233.

This section covers the following topics:

- Iterating Over Index Values with TupleIterator
- Controlling the Ordering of Map Input Tuples

19.3.2.1 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 235.

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...
        }
    }
```

19.3.2.2 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;
}
```

19.3.3 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.

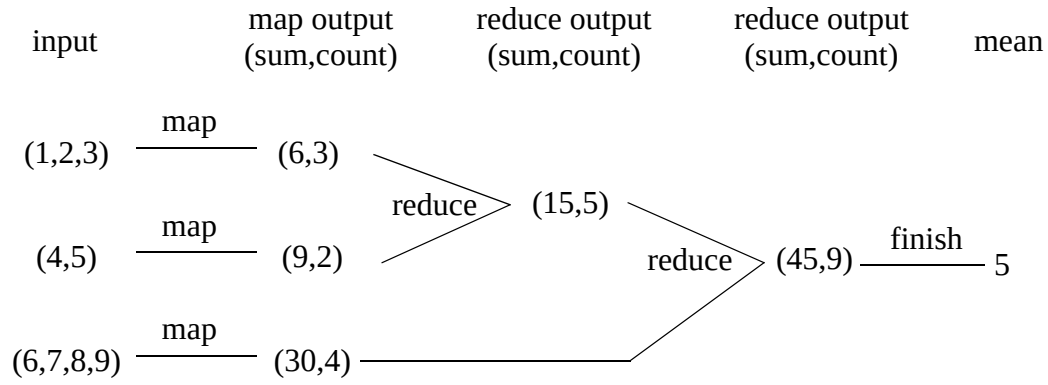
The `reduce` method has the following signature. Fold the data from the input `AggregateUDF` into the object on which reduce is called. Use the `Reporter` to report errors and log messages; see "Aggregate UDF Error Handling and Logging" on page 233.

```
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`.

19.3.4 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 233.

```
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 235.

19.3.5 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 241.

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 230.

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);
}
```

19.3.6 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

19.3.6.1 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.

19.3.6.2 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 229.

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

19.3.7 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;
};
```

19.3.8 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 in 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.");
  }
}
```

19.3.9 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

19.3.9.1 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 234.

For a more complete example, see "Example: Passing Arguments to an Aggregate UDF" on page 235.

19.3.9.2 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 235.

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.

19.3.9.3 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.");
    }
  }
}
```

19.3.10 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

19.3.10.1Where 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

19.3.10.2Type 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.

19.3.10.3C++ 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

19.4 Implementing Native Plugin Libraries

A native plugin allows you to extend the functionality of MarkLogic Server through a C++ shared library that implements a MarkLogic-defined interface such as `marklogic::AggregateUDF`. This section covers the following topics about native plugins:

- 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

19.4.1 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; see Using Aggregate User-Defined Functions in the *Search Developer's Guide*.

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, queries 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.

19.4.2 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 4.1.2
Solaris	gcc 4.1.2
Windows	Microsoft Visual Studio 9 SP1
MacOS	gcc 4.2.1

19.4.3 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 244.
- 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>
```

19.4.4 Installing a Native Plugin

After packaging your native plugin as described in “Packaging a Native Plugin” on page 239, install or update your plugin using the XQuery function `plug:install-from-zip`. For example:

```
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/udf/sampleplugin.zip")/node())
```

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. 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 238.

19.4.5 Uninstalling a Native Plugin

To uninstall a native plugin, call the XQuery 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:

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

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 238. There can be a slight delay

19.4.6 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");
}
```

19.4.7 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.

19.4.8 Checking the Status of Loaded Plugins

Using the Admin Interface or `xdmp:host-status`, 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 monitor loaded plugins using `xdmp:host-status`, open Query Console and run a query similar to the following:

```
xquery version "1.0-ml";
(: List native plugins loaded on this host :)
xdmp:host-status(xdmp:host())//*:native-plugins
```

You should see output similar to the following if there are plugins loaded:

```
<?xml version="1.0" encoding="UTF-8"?>
<native-plugins xmlns="http://marklogic.com/xdmp/status/host">
  <native-plugin>
    <path>native/sampleplugin-id/libsampleplugin.so</path>
    <version>1520437518</version>
    <capabilities>
      <aggregate>max_dateTime</aggregate>
      <aggregate>max_string</aggregate>
      <aggregate>variance</aggregate>
      <aggregate>min_point</aggregate>
      <aggregate>max</aggregate>
      <aggregate>median-test</aggregate>
      <aggregate>min</aggregate>
    </capabilities>
```

```
  </native-plugin>
</native-plugins>
```

19.4.9 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>
```

19.4.10 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 it's 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.

20.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement. For evaluation licenses, MarkLogic may provide support on an “as possible” basis.

For customers with a support contract, we invite you to visit our support website at http://support.marklogic.com to access information on known and fixed issues.

For complete product documentation, the latest product release downloads, and other useful information for developers, visit our developer site at http://developer.marklogic.com.

If you have questions or comments, you may contact MarkLogic Technical Support at the following email address:

support@marklogic.com

If reporting a query evaluation problem, please be sure to include the sample XQuery code.

Combined Product Notices

MarkLogic Server 6.0 and supporting products.
Last updated: August 19, 2013

COPYRIGHT

Copyright © 2013 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.

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.

1. Contains SAP BusinessObjects Text Analysis XI from SAP AG. Copyright © 1996-2011. All rights reserved.

2. Highslide Software from Highslide Software Torstein Honsi. All Highslide Software is protected by local and international copyright laws. All rights reserved.

3. Icons developed by Yusuke Kamiyamane. Copyright © 2011 Yusuke Kamiyamane. All rights reserved. Icons are licensed subject to http://creativecommons.org/licenses/by/3.0/legalcode.

4. Antenna House OfficeHTML Copyright © 2000-2008 Antenna House, Inc. All rights reserved.

5. Argus Copyright ©1999-2008 Iceni Technology Ltd. All rights reserved.

6. Rosette Linguistics Platform 6.5.2/6.5.3 from Basis Technology Corporation, Copyright © 2004-2008 Basis Technology Corporation. All rights reserved.

7. ISYS Search. Copyright © 2011 ISYS™ Search Software, Inc. All rights reserved.

8. Software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/) Copyright © 1995-1998 Eric Young (eay@cryptsoft.com). All rights reserved. Copyright © 1998-2011 The OpenSSL Project. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. All advertising materials mentioning features or use of this software must display the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)" 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org. 5. Products derived from this

software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project. 6. Redistributions of any form whatsoever must retain the following acknowledgment: "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/)" THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com). Additional terms may apply to the foregoing software as further set forth at http://www.openssl.org/source/license.html

9. ICU v. 4.2.1 available from http://site.icu-project.org// Copyright © 1995-2011 International Business Machines Corporation and others. All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE BE LIABLE FOR ANY CLAIM, OR ANY SPECIAL INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE. Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

10. Tidy available at http://www.w3.org/People/Raggett/tidy/. Copyright © 1998-2008 World Wide Web Consortium (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All Rights Reserved.

11. 'zlib' general purpose compression library v. 1.2.3. Copyright © 1995-2010 Jean-loup Gailly and Mark Adler. http://www.zlib.net/zlib_license.html

12. RSA Data Security, Inc. MD5 Message-Digest Algorithm, copyright © 1991-2, RSA Data Security, Inc. Created 1991. All rights reserved. License to copy and use this software is granted provided that it is identified as the "RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing this software or this function. License is also granted to make and use derivative works provided that such works are identified as "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing the derived work. RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind. These notices must be retained in any copies of any part of this documentation and/or software.

13. TRE software, available at http://laurikari.net/tre/. Copyright © 2001-2009 Ville Laurikari <vl@iki.fi>. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: (1) Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. (2) Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

14. Eigen Library, version 2.0.10, available at http://eigen.tuxfamily.org/index.php?title=Main_Page. Eigen Library is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. Eigen Library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details. You should have received a copy of the GNU Lesser General Public License along with Eigen Library. If not, see: http://www.gnu.org/copyleft/lesser.html

15. FunctX XQuery Library v.1.0 available at http://www.xqueryfunctions.com/xq/, Copyright © Datypic (Priscilla Walmsley). You can redistribute such library and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. This library is distributed in the

hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License as available at http://www.gnu.org/licenses/lgpl-2.1.html#SEC4 for more details.

16. DeployJava.js v. 1.8, Copyright © 2006, 2011, Oracle and/or its affiliates. All rights reserved. ORACLE PROPRIETARY/CONFIDENTIAL. Use of the foregoing library is subject to the following license terms. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of Oracle nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

17. YUI libraries v.2.8 available at http://developer.yahoo.com/yui/. Copyright © 2011 Yahoo! Inc. All rights reserved. Use of the foregoing libraries is subject to the following license terms. Redistribution and use of this software in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of Yahoo! Inc. nor the names of YUI's contributors may be used to endorse or promote products derived from this software without specific prior written permission of Yahoo! Inc. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING

NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Additional notices related to components included in the foregoing library are also available at the following URL: http://yuilibrary.com/license/

18. XSLTForms libraries available at http://sourceforge.net/projects/xsltforms/ . You can redistribute such libraries and/or modify them under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. These libraries are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License as available at http://www.gnu.org/licenses/lgpl-2.1.html#SEC4 for more details.

19. jQuery v.1.7 libraries available at http://jquery.com/ and jQuery UI v.1.8.2 libraries available at http://jqueryui.com/. Copyright © 2011 John Resig. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OFMERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE ANDNONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BELIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTIONOF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTIONWITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

20. JSON v. 2 libraries available at http://www.json.org/json2.js. Copyright © 2002 JSON.org. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. The Software shall be used for Good, not Evil. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

21. CodeMirror libraries available at http://codemirror.net/. Copyright © 2011 by Marijn Haverbeke marijnh@gmail.com. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

22. History.js available https://github.com/balupton/History.js/ Copyright © 2011, Benjamin Arthur Lupton. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of Benjamin Arthur Lupton nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

23. Sample content provided for use with the software is copyrighted by the respective authors as contributed by each of them to Wikipedia pursuant to the Creative Commons Attribution-ShareAlike License found at http://creativecommons.org/licenses/by-sa/3.0/.

24. Visual Studio Tools for Office 3.0 (Microsoft VSTO 3.0) licensed under Microsoft Public License Microsoft Public License (Ms-PL) found at http://msdn.microsoft.com/en-us/library/ff647676.aspx.

25. The MarkLogic Connector for Hadoop contains Apache Jakarta Commons Modeler. Copyright 2001-2007 The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"). You may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

26. jemalloc http://www.canonware.com/jemalloc/index.html, released under the terms of the following BSD-derived licenses: Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice(s), this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice(s), this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER(S) ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER(S) BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Portions of the software are copyright (C) 2002-2013 Jason Evans, and portions copyright (C) 2009-2013 Facebook, Inc. and copyright (C) 2007-2012 Mozilla Foundation. Neither the name of Facebook, Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. All rights reserved.

27. Gziphelper; version: 1.0, Feb, 2003, authored by Gao Dasheng, is distributed subject to the following terms: This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software. Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions: (1.) the origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required; (2) altered source versions must be plainly marked as such, and must not be misrepresented as being the original software; (3) this notice may not be removed or altered from any source distribution.

28. Some features of MarkLogic allow you to develop and use Google Maps. Review Google Map's privacy policy at http://www.google.com/policies/privacy/, legal notices at http://maps.google.com/help/legalnotices_maps.html, and terms of use at http://maps.google.com/help/terms_maps.html.

29. commons-codec, HttpComponents, commons-logging, and Google Maps Utility are used under the Licensed under the Apache License, Version 2.0 (the "License"). You may not use these files except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

30. Mapstraction. Copyright (c) 2012, Mapstraction. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

31. logback-classic and logback-core are provided in object code form only under the Eclipse Public License - v 1.0 available at http://www.eclipse.org/legal/epl-v10.html. Copyright (C) 1999-2012, QOS.ch. All rights reserved. Source code for the Program is available http://logback.qos.ch/.

32. Simple Logging Facade for Java (SLF4J). Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

33. cURL v. 7.22.0 available at http://curl.haxx.se/. Copyright © 1996 - 2011, Daniel Stenberg, <daniel@haxx.se>. All rights reserved. Permission to use, copy, modify, and distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies. THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE. Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

34. heatmap.js distributed subject to the "THE BEER-WARE LICENSE" (Revision 42): <phk@FreeBSD.ORG> wrote this file. As long as you retain this notice you can do whatever you want with this stuff. If we meet some day, and you think this stuff is worth it, you can buy me a beer in return Poul-Henning Kamp.

35. Except for components that are otherwise subject to other open source licenses, the MarkLogic ODBC Driver is released under the GNU Lesser General Public License as published by the Free Software Foundation, version 2.1 of the License.

36. Solr-commons-csv-3.5.0 Copyright 2005-2006 The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

37. Avro v. 1.7.1. Copyright © The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

38. Guava v. 11.0.2. Copyright © Google. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

39. Hadoop-auth v. 2.0.0. Copyright © The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

40. hadoop-hdfs v. 2.0. Copyright © The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

41. commons-lang v. 2.4, 2.5. Copyright © The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

42. commons-configuration v. 2.0. Copyright © The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

43. jackson-core-asl v. 1.8.8. Copyright © FasterXML. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

44. jackson-mapper-asl v. 1.8.8. Copyright © FasterXML. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

45. log4j v. 1.2.16. Copyright © The Apache Software Foundation. Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0. Unless required by applicable law or agreed to in writing, this file is distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

46. protobuf-java v. 2.4.0a. Copyright © Google. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of Google nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

47. xpp3 v. 1.1.3.3. Copyright © 2002 Extreme! Lab, Indiana University. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Indiana University Extreme! Lab(http://www.extreme.indiana.edu/)." Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear. The names "Indiana Univeristy" and "Indiana Univeristy Extreme! Lab" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact http://www.extreme.indiana.edu/. Products derived from this software may not use "Indiana Univeristy" name nor may "Indiana Univeristy" appear in their name, without prior written permission of the Indiana University. THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS, COPYRIGHT HOLDERS OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IFADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

48. xstream v. 1.4.2. Copyright © 2003-2006, Joe Walnes. Copyright © 2006-2009, 2011 XStream Committers. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of XStream nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR

BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

TRADEMARK NOTICE

The MarkLogic name and logo are registered trademarks of MarkLogic Corporation in the United States and/or other countries. Excel and PowerPoint are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Any other names or logos included in the Software, this notice or the documentation are property of the respective trademark owners.