MarkLogic Server

REST Application Developer's Guide

MarkLogic 9
May, 2017

Last Revised: 9.0-8, December 2018

Copyright © 2019 MarkLogic Corporation. All rights reserved.

Table of Contents

REST Application Developer's Guide

1.0 Introduction to the MarkLogic REST API 13
- 1.1 Capabilities of the REST Client API 13
- 1.2 Getting Started with the MarkLogic REST API 14
  - 1.2.1 Preparation 14
  - 1.2.2 Choose a REST API Instance 15
  - 1.2.3 Load Documents Into the Database 15
  - 1.2.4 Search the Database 16
  - 1.2.5 Tear Down the REST API Instance 18
- 1.3 REST Client API Service Summary 18
- 1.4 Security Requirements 21
  - 1.4.1 Basic Security Requirements 21
  - 1.4.2 Controlling Access to Documents and Other Artifacts 22
  - 1.4.3 Evaluating Requests Against a Different Database 23
  - 1.4.4 Evaluating or Invoking Server-Side Code 24
- 1.5 Terms and Definitions 24
- 1.6 Understanding REST Resources 25
  - 1.6.1 Addressing a Resource 25
  - 1.6.2 Specifying the REST API Version 26
  - 1.6.3 Specifying Parameters in a Resource Address 26
- 1.7 Understanding the Example Commands 27
  - 1.7.1 Introduction to the curl Tool 27
  - 1.7.2 Modifying the Example Commands for Windows 28
- 1.8 Overriding the Content Database 29
- 1.9 Performing Point-in-Time Operations 29
- 1.10 Controlling Input and Output Content Type 31
  - 1.10.1 General Content Type Guidelines 31
  - 1.10.2 Details on Content Type Determination 32
  - 1.10.3 Example: Inserting and Reading a Document 34
  - 1.10.4 Example: Inserting and Reading Metadata 34
  - 1.10.5 Example: Documents With No or Unknown URI Extension 35
  - 1.10.6 Example: Mixing Document and Non-Document Data 36
- 1.11 Error Reporting 37

2.0 Administering REST Client API Instances 39
- 2.1 What Is an Instance? 39
- 2.2 Creating an Instance 40
- 2.3 Example: Creating an Instance 42
- 2.4 Removing an Instance 43

2.5 Retrieving Configuration Information 44
2.5.1 Retrieving Configuration for All Instances 44
2.5.2 Retrieving Instance Configuration by Content Database 45
2.5.3 Retrieving Instance Configuration by Instance Name 46
2.6 Configuring Instance Properties 47
2.6.1 Instance Configuration Properties 48
2.6.2 Listing Instance Property Settings 49
2.6.3 Setting Instance Properties 50
2.6.4 Resetting Instance Properties 52

3.0 Manipulating Documents 53
3.1 Summary of Document Management Services 53
3.1.1 Summary of the /documents Service 54
3.1.2 Summary of the /graphs Service 55
3.2 Loading Content into the Database 55
3.2.1 Loading Content 56
3.2.2 Adding Metadata 57
3.2.2.1 Inserting or Updating Metadata 57
3.2.2.2 Example: Replacing One Metadata Category Using XML 58
3.2.2.3 Example: Replacing Multiple Metadata Categories Using XML 59
3.2.2.4 Example: Replacing Multiple Metadata Categories Using JSON 59
3.2.3 Loading Content and Adding Metadata in the Same Request 60
3.2.3.1 Loading Content and Metadata Using Request Parameters 60
3.2.3.2 Loading Content and Metadata Using a Multipart Message 61
3.2.4 Automatically Generating Document URIs 62
3.2.5 Loading Triples 63
3.2.6 Controlling Access to Documents Created with the REST API 64
3.2.7 Transforming Content During Ingestion 64
3.3 Retrieving Documents from the Database 65
3.3.1 Retrieving the Contents of a Document 65
3.3.2 Retrieving Metadata About a Document 65
3.3.3 Retrieving Content and Metadata in a Single Request 66
3.3.4 Transforming Content During Retrieval 67
3.4 Partially Updating Document Content or Metadata 68
3.4.1 Introduction to Content and Metadata Patching 68
3.4.2 Basic Steps for Patching Documents 70
3.4.3 XML Patch Reference 70
3.4.3.1 patch 71
3.4.3.2 insert 72
3.4.3.3 replace 73
3.4.3.4 replace-insert 75
3.4.3.5 delete 78
3.4.3.6 replace-library 79
3.4.4 Managing XML Namespaces in a Patch 80

3.4.5 XML Examples of Partial Updates 81
3.4.5.1 Example cURL Commands 82
3.4.5.2 Example: Inserting an Element 82
3.4.5.3 Example: Inserting an Attribute 83
3.4.5.4 Example: Inserting a Text Node 83
3.4.5.5 Example: Inserting a Comment or Processing Instruction 84
3.4.5.6 Example: Multiple Inserts in One Patch 84
3.4.5.7 Example: Replacing an Element or Element Contents 85
3.4.5.8 Example: Replacing an Attribute Value 86
3.4.5.9 Example: Replacing or Inserting an Element in One Operation ... 86
3.4.5.10 Example: Replacing or Inserting an Attribute in One Operation .. 87
3.4.5.11 Example: Deleting an Element or Attribute 88
3.4.6 JSON Patch Reference 88
3.4.6.1 pathlang 89
3.4.6.2 patch 89
3.4.6.3 insert 90
3.4.6.4 replace 91
3.4.6.5 replace-insert 93
3.4.6.6 delete 96
3.4.6.7 replace-library 97
3.4.7 JSON Examples of Partial Update 98
3.4.7.1 Example cURL Commands 99
3.4.7.2 Example: Insert 99
3.4.7.3 Example: Replace 100
3.4.7.4 Example: Replace-Insert 104
3.4.7.5 Example: Delete 108
3.4.8 Patching Metadata 109
3.4.9 Constructing Replacement Data on the Server 112
3.4.9.1 Using a Replacement Constructor Function 112
3.4.9.2 Using Builtin Replacement Constructors 115
3.4.9.3 Writing an XQuery User-Defined Replacement Constructor .116
3.4.9.4 Writing a JavaScript User-Defined Replacement Constructor 118
3.4.9.5 Installing or Updating a User-Defined Replace Library 121
3.4.10 How Position Affects the Insertion Point 122
3.4.10.1 Specifying Position in XML 122
3.4.10.2 Specifying Position in JSON 124
3.4.11 Path Expressions Usable in Patch Operations 126
3.4.12 Limitations of JSON Path Expressions 126
3.4.13 Introduction to JSONPath 126
3.5 Performing a Lightweight Document Check 127
3.6 Removing Documents from the Database 128
3.6.1 Removing a Document or Metadata 129
3.6.2 Removing Multiple Documents 129
3.6.3 Removing All Documents 130

3.6.4 Removing a Semantic Graph 130
3.7 Using Optimistic Locking to Update Documents 131
3.7.1 Understanding Optimistic Locking 131
3.7.2 Enabling Optimistic Locking 132
3.7.3 Obtaining a Version Id 133
3.7.4 Applying a Conditional Update 134
3.7.5 Example: Updating a Document Using Optimistic Locking 135
3.8 Client-Side Cache Management Using Content Versioning 137
3.8.1 Enabling Content Versioning 137
3.8.2 Using Content Versioning for Cache Refreshing 138
3.8.3 Example: Refreshing a Cached Document 138
3.9 Working with Binary Documents 140
3.9.1 Types of Binary Documents 140
3.9.2 Streaming Binary Content 140
3.10 Working with Temporal Documents 141
3.11 Working with Metadata 142
3.11.1 Metadata Categories 142
3.11.2 XML Metadata Format 143
3.11.3 JSON Metadata Format 144
3.11.4 Working With Document Properties Using JSON 145
3.11.5 Disabling Metadata Merging 146
3.11.5.1 When to Consider Disabling Metadata Merging 147
3.11.5.2 Understanding Metadata Merging 147
3.11.5.3 How to Disable Metadata Merging 147

4.0 Using and Configuring Query Features 149
4.1 Query Feature Overview 149
4.2 Querying Documents and Metadata 151
4.2.1 Constraining a Query by Collection or Directory 151
4.2.2 Searching With String Queries 152
4.2.3 Searching With Structured Queries 153
4.2.4 Searching With cts:query 154
4.2.5 Debugging /search Queries With Logging 155
4.3 Querying Lexicons and Range Indexes 157
4.3.1 Querying the Values in a Lexicon or Range Index 158
4.3.2 Finding Value Co-Occurrences in Lexicons 161
4.3.3 Using a Query to Constrain Results 163
4.3.4 Identifying Lexicon and Range Index Values in Query Options 164
4.3.4.1 Defining Queryable Lexicon or Range Index Values 164
4.3.4.2 Defining Queryable Lexicon or Range Index Co-Occurrences 166
4.3.5 Creating Indexes on JSON Properties 168
4.3.6 Limiting the Number of Results 168
4.4 Using Query By Example to Prototype a Query 169
4.4.1 What is QBE 169
4.4.2 Searching Documents With QBE 170

4.4.3 Validating a QBE 172
4.4.4 Generating a Combined Query from a QBE 172
4.5 Analyzing Lexicons and Range Indexes With Aggregate Functions 174
4.5.1 Aggregate Function Overview 174
4.5.2 Using Query Options to Apply Aggregate Functions 174
4.5.3 Using Request Parameters to Apply Aggregate Functions 176
4.5.4 Example: Applying a Builtin Aggregate Function 178
4.5.5 Example: Applying an Aggregate UDF 178
4.6 Specifying Dynamic Query Options with Combined Query 179
4.6.1 Syntax and Semantics 180
4.6.2 Interaction with Queries in Request Parameters 181
4.6.3 Interaction with Persistent Query Options 182
4.6.4 Performance Considerations 183
4.6.5 Combined Query Examples 184
4.6.5.1 Example: Overriding Persistent Constraints 184
4.6.5.2 Example: Modifying the Search Response 185
4.6.5.3 Example: Including a cts Query in a Combined Query 186
4.6.5.4 Example: Including a QBE in a Combined Query 187
4.7 Querying Triples 188
4.8 Invoking an Optic API Query Plan 189
4.8.1 Generating a Plan 190
4.8.2 Invoking a Plan 190
4.8.3 Controlling the Inclusion of Type Information in a Row Set 191
4.8.4 Generating a Row Set 192
4.8.4.1 Example Input Plan 192
4.8.4.2 Single JSON Object 193
4.8.4.3 Single JSON Array 194
4.8.4.4 Single XML Element 196
4.8.4.5 Line Delimited JSON Objects 197
4.8.4.6 Line Delimited JSON Arrays 198
4.8.4.7 Comma-Separated Text (CSV) 199
4.8.4.8 Comma-Separated Arrays 199
4.8.4.9 Multipart With Rows as JSON Objects 200
4.8.4.10 Multipart With Rows as JSON Arrays 201
4.8.4.11 Multipart With Rows as XML Elements 203
4.8.5 Passing Parameters into a Plan 205
4.8.6 Handling Complex Column Values 205
4.8.7 Generating an Execution Plan 207
4.9 Searching Values Metadata Fields 208
4.10 Configuring Query Options 208
4.10.1 Controlling Queries With Options 208
4.10.2 Adding Query Options to a Request 209
4.10.3 Creating or Modifying Query Options 210
4.10.4 Creating or Modifying One Option 211
4.10.5 Checking Index Availability 212
4.10.6 Retrieving Options 214

4.10.7 Retrieving a List of Installed Query Options 215
4.10.8 Removing Query Options 216
4.10.8.1 Removing Query Options 216
4.10.8.2 Removing a Single Option 217
4.10.8.3 Removing All Named Query Options 217
4.11 Using Namespace Bindings 217
4.11.1 When Do You Need a Namespace Binding 218
4.11.2 Creating or Updating a Namespace Binding 218
4.11.3 Creating or Updating Multiple Namespace Bindings 220
4.11.4 Listing Available Namespace Bindings 222
4.11.5 Deleting Namespace Bindings 224
4.12 Generating Search Facets 224
4.13 Paginating Results 226
4.14 Customizing Search Results 227
4.14.1 Customizing Search Snippets 228
4.14.1.1 Customizing the Default Search Snippets 228
4.14.1.2 Creating Your Own Snippet Extension 233
4.14.1.3 Generating Custom JSON Snippets 238
4.14.2 Transforming the Search Response 239
4.14.2.1 Basic Search Transform Usage 239
4.14.2.2 What to Expect as Input Content 239
4.14.2.3 Expected Output 241
4.15 Generating Search Term Completion Suggestions 241
4.15.1 Basic Steps 242
4.15.2 Example: Generating Search Suggestions Using GET 242
4.15.2.1 Initialize the Database 243
4.15.2.2 Install Query Options 244
4.15.2.3 Retrieve Unconstrained Search Suggestions 244
4.15.2.4 Retrieve Constrained Search Suggestions 245
4.15.3 Example: Generating Search Suggestions Using POST 246
4.15.4 Where to Find More Information 247

5.0 Reading and Writing Multiple Documents 249
5.1 Terms and Definitions 249
5.2 Writing Multiple Documents 249
5.2.1 Example: Loading Multiple Documents 250
5.2.2 Request Body Overview 253
5.2.3 Response Overview 254
5.2.4 Constructing a Content Part 256
5.2.4.1 Controlling Document Type 257
5.2.4.2 Specifying an Explicit Document URI 257
5.2.4.3 Automatically Generating a Document URI 258
5.2.4.4 Adding Content Options 258
5.2.4.5 Example Content Part Headers 258
5.2.5 Understanding Metadata Scoping 259
5.2.6 Understanding When Metadata is Preserved or Replaced 262

5.2.7 Constructing a Metadata Part 263
5.2.7.1 Constructing a Request Default Metadata Part 263
5.2.7.2 Constructing a Document-Specific Metadata Part 264
5.2.7.3 Disabling Request Default Metadata 268
5.2.8 Applying a Write Transformation 268
5.2.9 Example: Controlling Metadata Through Defaults 269
5.2.9.1 Payload Description 269
5.2.9.2 Generating the POST Body 270
5.2.9.3 Executing the Request 272
5.2.9.4 Verifying the Results 273
5.2.10 Example: Reverting to System Default Metadata 274
5.2.10.1 Payload Description 274
5.2.10.2 Generating the POST Body 276
5.2.10.3 Executing the Request 277
5.2.10.4 Verifying the Results 278
5.2.11 Example: Adding Documents to a Collection 279
5.2.11.1 Payload Description 280
5.2.11.2 Generating the POST Body 280
5.2.11.3 Executing the Request 282
5.2.11.4 Verifying the Results 283
5.2.11.5 Extending the Example 285
5.2.12 Generating Example Payloads with XQuery 285
5.3 Reading Multiple Documents by URI 288
5.4 Reading Multiple Documents Matching a Query 289
5.4.1 Overview of Bulk Read By Query 289
5.4.2 Example: Using Query By Example (QBE) 291
5.4.3 Example: Using a String, Structured, or Combined Query 293
5.4.4 Extracting a Portion of Each Matching Document 294
5.4.5 Including Search Results in the Response 295
5.4.6 Paginating Results 296
5.5 Bulk Read Response Overview 297

6.0 Managing Transactions 301
6.1 Service Summary 301
6.2 Overview of RESTful Transactions 302
6.3 Creating a Transaction 303
6.4 Associating a Transaction with a Request 304
6.5 Committing or Rolling Back a Transaction 304
6.6 Checking Transaction Status 305
6.7 Managing Transactions When Using a Load Balancer 306

7.0 Alerting 311
7.1 Summary of /alert Services 311
7.2 Alerting Pre-Requisites 311
7.3 Alerting Concepts 312

7.4 Defining an Alerting Rule 313
7.5 Installing an Alerting Rule 314
7.6 Testing for Matches to Alerting Rules 315
7.6.1 Basic Steps 315
7.6.2 Identifying Input Documents Using a Query 317
7.6.3 Identifying Input Documents Using URIs 318
7.6.4 Matching Against a Transient Document 318
7.6.5 Filtering Match Results 319
7.6.6 Transforming Match Results 319
7.6.6.1 Writing a Match Result Transform 319
7.6.6.2 Using a Match Result Transform 320
7.7 Retrieving Rule Definitions 321
7.8 Testing for the Existence of Rule 321
7.9 Deleting a Rule 322

8.0 Working With Content Transformations 323
8.1 Writing Transformations 323
8.1.1 Guidelines for Writing Transforms 323
8.1.2 Writing JavaScript Transformations 324
8.1.3 Writing XQuery Transformations 325
8.1.4 Writing XSLT Transformations 327
8.1.5 Expected Input and Output 328
8.1.6 Reporting Errors 328
8.1.7 Context Map Keys 330
8.1.8 Controlling Transaction Mode 330
8.2 Installing Transformations 331
8.2.1 Basic Installation 331
8.2.2 Including Transform Metadata 332
8.3 Applying Transformations 333
8.4 Discovering Transformations 333
8.5 Retrieving the Implementation of a Transformation 334
8.6 JavaScript Example: Adding a JSON Property During Ingestion 335
8.7 XQuery Example: Adding an Attribute During Ingestion 338
8.8 XSLT Example: Adding an Attribute During Ingestion 340
8.9 XQuery Example: Modifying Document Type During Ingestion 342

9.0 Extending the REST API 345
9.1 Available REST API Extension Points 345
9.2 Understanding Resource Service Extensions 346
9.3 Creating a JavaScript Resource Service Extension 346
9.3.1 Guidelines for Writing JavaScript Resource Service Extensions 347
9.3.2 The JavaScript Resource Extension Interface 347
9.3.3 Reporting Errors 349
9.3.4 Setting the Response Status Code 350
9.3.5 Setting Additional Response Headers 351

9.3.6 Context Object Properties 351
9.3.7 Controlling Transaction Mode 353
9.4 Creating an XQuery Resource Service Extension 353
9.4.1 Guidelines for Writing XQuery Resource Service Extensions 354
9.4.2 The Resource Extension Interface 354
9.4.3 Reporting Errors 356
9.4.4 Setting the Response Status Code 357
9.4.5 Setting Additional Response Headers 358
9.4.6 Context Map Keys 359
9.4.7 Controlling Transaction Mode 359
9.5 Installing a Resource Service Extension 360
9.6 Using a Resource Service Extension 362
9.7 Example: JavaScript Resource Service Extension 362
9.7.1 JavaScript Extension Implementation 362
9.7.2 Installing the Example Extension 366
9.7.3 Using the Example Extension 367
9.8 Example: XQuery Resource Service Extension 370
9.8.1 XQuery Extension Implementation 370
9.8.2 Installing the Example Extension 372
9.8.3 Using the Example Extension 374
9.9 Controlling Access to a Resource Service Extension 374
9.10 Discovering Resource Service Extensions 374
9.11 Retrieving the Implementation of a Resource Service Extension 375
9.12 Managing Dependent Libraries and Other Assets 376
9.12.1 Maintenance of Dependent Libraries and Other Assets 376
9.12.2 Installing or Updating an Asset 377
9.12.3 Referencing an Asset 378
9.12.4 Removing an Asset 378
9.12.5 Retrieving an Asset List 378
9.12.6 Retrieving an Asset 379
9.13 Evaluating an Ad-Hoc Query 380
9.13.1 Required Privileges 380
9.13.2 Evaluating an Ad-Hoc JavaScript Query 381
9.13.3 Evaluating an Ad-Hoc XQuery Query 383
9.13.4 Specifying External Variable Values 385
9.13.5 Interpreting the Results of an Ad-Hoc Query 386
9.13.6 Managing Session State 388
9.14 Evaluating a Module Installed on MarkLogic Server 388

10.0 Migrating REST Applications 393
10.1 Before You Begin 393
10.2 Migrating the REST API Instance and Database Configuration 393
10.2.1 Export the Configuration from the Source Cluster 394
10.2.2 Import the Configuration to the Destination Cluster 394
10.3 Migrating the Contents of the Database 395

11.0 Technical Support 397

12.0 Copyright 399
12.0 COPYRIGHT 399

1.0 Introduction to the MarkLogic REST API

The REST Client API provides a set of RESTful services for creating, updating, retrieving, deleting and query documents and metadata. This section provides a brief overview of the features of the API.

- Capabilities of the REST Client API
- Getting Started with the MarkLogic REST API
- REST Client API Service Summary
- Security Requirements
- Terms and Definitions
- Understanding REST Resources
- Understanding the Example Commands
- Overriding the Content Database
- Performing Point-in-Time Operations
- Controlling Input and Output Content Type
- Error Reporting

1.1 Capabilities of the REST Client API

The REST Client API is a RESTful interface for building client applications. The capabilities of the API include the following:

- Create, retrieve, update, and delete documents, metadata, and semantic triples in a MarkLogic database.
- Search documents and semantic graphs and query lexicon values, using several query formats, including string query, structured query, combined query, Query by Example (QBE), and SPARQL.
- Customize your queries by configuring dynamic and/or persistent query options.
- Apply transformations to document contents and search results.
- Extend the API to expose custom capabilities you author in XQuery and install on MarkLogic Server.

For a complete list of services, see “REST Client API Service Summary” on page 18.

You can use the REST Client API to work with XML, JSON, text, and binary documents. In most cases, your application can use either XML or JSON to exchange non-document data such as queries and search results with MarkLogic Server.

REST API client applications interact with MarkLogic Server through a REST API instance, a specially configured HTTP App Server. Each REST API instance is intended to service a single content database and client application. You can create and configure a REST API instance via a REST request or interactively. For details, see “Administering REST Client API Instances” on page 39.

You can configure whether errors are returned to your application as XML or JSON. For details, see “Error Reporting” on page 37.

1.2 Getting Started with the MarkLogic REST API

This section leads you through a simple example that uses the REST Client API to insert documents into a database and perform a search. We will follow these steps:

1. Preparation
2. Choose a REST API Instance
3. Load Documents Into the Database
4. Search the Database
5. Tear Down the REST API Instance

1.2.1 Preparation

Before beginning this walkthrough, you should have the following software installed:

- MarkLogic Server, version 6.0-1 or later
- `curl`, a command line tool for issuing HTTP requests, or an equivalent tool.

Though the examples rely on `curl`, you can use any tool or library capable of sending HTTP requests. If you are not familiar with `curl` or do not have `curl` on your system, see “Introduction to the curl Tool” on page 27.

To create the input documents used by the walkthrough:

1. Create a text file named `one.xml` with the following contents:

```
<one>
  <child>The noble Brutus has told Caesar was ambitious</child>
</one>
```

2. Create a text file named `two.json` with the following contents:

```
{
  "two": {
    "child": "I come to bury Caesar, not to praise him."
  }
}
```

1.2.2 Choose a REST API Instance

You must have a *REST API instance* to use the REST Client API. A REST API instance is an HTTP App Server specially configured to service HTTP requests against the API. For details, see “What Is an Instance?” on page 39.

> **Note:** Each REST API instance can only host a single application. You cannot share the modules database across multiple REST API instances.

When you install MarkLogic Server 8 or later, the App Server on port 8000 can be used as a REST API instance. This instance is attached to the Documents database. The examples in this walkthrough and the remainder of this guide use the REST API instance on port 8000.

You can also create a REST API instance on a different port, attached to a different database. For details, see “Creating an Instance” on page 40.

1.2.3 Load Documents Into the Database

This procedure loads sample content into the database associated with your REST API instance using the `/documents` service. The `/documents` service allows you to create, read, update and delete documents in the database.

To load the sample documents into the database:

1. Navigate to the directory containing the sample documents you created in “Preparation” on page 14.

2. Execute the following command to load `one.xml` into the database with the URI `/xml/one.xml`:

   ```
   $ curl --anyauth --user user:password -X PUT -d@'./one.xml' \
       -H "Content-type: application/xml" \
       'http://localhost:8000/LATEST/documents?uri=/xml/one.xml'
   ```

   The URL tells the `/documents` service to create an XML document with database URI `/xml/one.xml` (`uri=...`) from the contents in the request body. If the request succeeds, the service returns status 201 (Document Created).

3. Execute the following command to load `two.json` into the database with the URI `/json/two.json`:

   ```
   $ curl --anyauth --user user:password -X PUT -d@'./two.json' \
     -H "Content-type: application/json" \
     'http://localhost:8000/LATEST/documents?uri=/json/two.json'
   ```

4. Optionally, use Query Console to explore the database. The database should contain 2 documents, `/xml/one.xml` and `/json/two.json`.

To learn more about the document manipulation features of the `documents` service, see "Manipulating Documents" on page 53.

1.2.4 Search the Database

The REST Client API provides several query services. This procedure uses the `search` service to perform a simple string query search of the database, finding documents containing "caesar". For details, see "Using and Configuring Query Features" on page 149.

To search the database:

1. Execute the following command to send a search request to the instance, requesting matches to the search string "caesar". Results are returned as XML by default.

```
$ curl --anyauth --user user:password \
    'http://localhost:8000/LATEST/search?q=caesar'
```

2. Examine the XML search results returned in the response body. Notice that there are two matches, one in each document.

```
<search:response snippet-format="snippet" total="2" start="1" ...>
  <search:result index="1" uri="/xml/one.xml" ...>
    <search:snippet>
      <search:match
path="fn:doc(&quot;/xml/one.xml&quot;)/one/child">The noble Brutus has
told <search:highlight>Caesar</search:highlight> was
ambitious</search:match>
    </search:snippet>
  </search:result>
  <search:result index="2" uri="/json/two.json"
path="fn:doc(&quot;/json/two.json&quot;)" score="2048"
confidence="0.283107" fitness="0.235702">
    <search:snippet>
      <search:match
path="fn:doc(&quot;/json/two.json&quot;)/*:json/*:two/*:child">I come
to bury <search:highlight>Caesar</search:highlight>, not to praise
him.</search:match>
    </search:snippet>
  </search:result>
  <search:qtext>caesar</search:qtext>
  <search:metrics>...</search:metrics>
</search:response>
```

3. Run the search command again, generating JSON output by an Accept header:

```
$ curl --anyauth --user user:password \
    -H "Accept: application/json" \
    'http://localhost:8000/LATEST/search?q=caesar'

{
  "snippet-format": "snippet",
```

```
  "total": 2,
  "start": 1,
  "page-length": 10,
  "results": [
    {
      "index": 1,
      "uri": "\/xml\/one.xml",
      "path": "fn:doc(\"\/xml\/one.xml\")",
      "score": 2048,
      "confidence": 0.283107,
      "fitness": 0.235702,
      "matches": [
        {
          "path": "fn:doc(\"\/xml\/one.xml\")\/one\/child",
          "match-text": [
            "The noble Brutus has told ",
            {
              "highlight": "Caesar"
            },
            " was ambitious"
          ]
        }
      ]
    },
    {
      "index": 2,
      "uri": "\/json\/two.json",
      "path": "fn:doc(\"\/json\/two.json\")",
      "score": 2048,
      "confidence": 0.283107,
      "fitness": 0.235702,
      "matches": [
        {
          "path":
"fn:doc(\"\/json\/two.json\")\/*:json\/*:two\/*:child",
          "match-text": [
            "I come to bury ",
            {
              "highlight": "Caesar"
            },
            ", not to praise him."
          ]
        }
      ]
    }
  ],
  "qtext": "caesar",
  "metrics": { ... }
}
```

Additional query features allow you to search using structured queries or Query By Example (QBE), to search by JSON property and value or XML element and element attribute values, and to search and analyze lexicons and range indexes. You can also define search options to tailor your search and results. For details, see "Using and Configuring Query Features" on page 149.

1.2.5 Tear Down the REST API Instance

If you are using the pre-configured REST API instance on port 8000, skip this step. If you are using a REST API instance on another port that you created to walk through the examples, then you can tear it down following these instructions.

This procedure uses the `rest-apis` service on port 8002 to remove a REST Client API instance. By default, removing the instance leaves the content and modules databases associated with the instance intact, but in this example we remove them by using the `include` request parameter.

> **Note:** Tearing down a REST API instance causes a server restart.

Follow this procedure to remove your instance and associated content and modules databases:

1. Run the following shell command to remove the instance and database. Change the *instance-name* to the name you used when you created your instance.

```
$ curl --anyauth --user user:password -X DELETE \
  'http://localhost:8002/LATEST/rest-apis/instance-name?include=conten
t&include=modules'
```

2. Navigate to the Admin Interface in your browser to confirm removal of the databases and App Server:

```
http://localhost:8001
```

1.3 REST Client API Service Summary

The following table gives a brief overview of the services provided by the REST Client API and where to find out more information. Additional, finer grained services are available in many cases. For example, in addition to `/config/query` (manage query options), there is a `/config/query/{name}` service (manage a specific option). For details, refer to the *MarkLogic REST API Reference*.

These services are made available through an HTTP App Server when you create an instance of the REST Client API. For details, see "Creating an Instance" on page 40.

Service	Description	More Information
`/rest-apis`	REST Client API instance administration, including creating and tearing down instances.	"Administering REST Client API Instances" on page 39
`/documents`	Document manipulation, including creating, updating and deleting documents and meta data.	"Manipulating Documents" on page 53
`/search`	Search content and metadata using string and structured queries.	"Using and Configuring Query Features" on page 149
`/qbe`	Search content using Query By Example, a query syntax that closely resembles the structure of your documents.	"Using Query By Example to Prototype a Query" on page 169
`/values`	Retrieve lexicon and range index values and value co-occurrences. Apply builtin and user-defined aggregate functions to lexicon and range index values and value co-occurrences.	"Using and Configuring Query Features" on page 149
`/suggest`	Retrieve text completion suggestions based on query text entered by the user.	"Generating Search Term Completion Suggestions" on page 241
`/graphs`	Store and manage graphs containing semantic triples data.	"Loading Triples" on page 63
`/graphs/sparql`	Perform semantic queries using SPARQL.	"Querying Triples" on page 188
`/graphs/things`	Retrieve a list of all graph nodes (triples) in the database.	See Exploring Triples with the REST Client API in the *Semantics Developer's Guide*.
`/eval`	Evaluate ad-hoc JavaScript or XQuery code on MarkLogic Server.	"Evaluating an Ad-Hoc Query" on page 380
`/invoke`	Evaluate a JavaScript or XQuery module installed on MarkLogic Server.	"Evaluating a Module Installed on MarkLogic Server" on page 388

Service	Description	More Information
`/alert`	Support for creating alerting applications.	“Alerting” on page 311
`/transactions`	Support for evaluating REST requests in multi-statement transactions. Create, commit, rollback, and monitor transactions.	“Managing Transactions” on page 301
`/config/query`	Create, modify, delete, and read configuration options used to control queries made services such as /search, /qbe, and /values.	“Configuring Query Options” on page 208
`/config/indexes`	Compare query options against the database configuration to determine whether all required indexes are configured in the database.	“Checking Index Availability” on page 212
`/config/properties`	Configure instance-wide properties, such as enabling debug output and setting the content type of error messages.	“Configuring Instance Properties” on page 47
`/config/transforms`	Create, update, delete, and read user-defined content transformations. Transformations can be used to modify content when it is inserted into or retrieved from the database using the `/documents` service, or to transform search results.	“Working With Content Transformations” on page 323
`/config/namespaces`	Create and manage instance-wide namespace prefix bindings. Such bindings allow you to use namespace prefixes in queries that do support other means of defining prefixes.	“Using Namespace Bindings” on page 217
`/config/resources`	Manage resource service extensions.	“Extending the REST API” on page 345

Service	Description	More Information
`/resources`	Access to user-defined resource service extensions.	"Extending the REST API" on page 345
`/ext`	Manage assets in the modules database associated with a REST API instance, such as dependent XQuery library modules used by transformations and resource service extensions.	"Managing Dependent Libraries and Other Assets" on page 376

1.4 Security Requirements

This describes the basic security model used by the REST Client API, and some common situations in which you might need to change or extend it. The following topics are covered:

- Basic Security Requirements
- Controlling Access to Documents and Other Artifacts
- Evaluating Requests Against a Different Database
- Evaluating or Invoking Server-Side Code

1.4.1 Basic Security Requirements

The user with which you make a REST Client API request must have appropriate privileges for the content accessed by the request, such as permission to read or update documents in the target database.

In addition, the user must use one or more of the pre-defined roles listed below, or the equivalent privleges. The role/privilege requirement for each REST Client API operation is listed in the *MarkLogic REST API Reference*. The capabilities of each role in the table is subsumed in the roles below it.

Role	Description
`rest-extension-user`	Enables access to resource service extension methods. This role is implicit in the other pre-defined REST API roles, but you may need to explicitly include it when defining custom roles. For details, see "Controlling Access to Documents and Other Artifacts" on page 22.
`rest-reader`	Enables read operations through the REST Client API, such as retrieving documents and metadata. This role does not grant any other privileges, so the user might still require additional privileges to read content.
`rest-writer`	Enables write operations through the REST Client API, such as creating documents, metadata, or configuration information. This role does not grant any other privileges, so the user might still require additional privileges to write content.
`rest-admin`	Enables administrative operations through the REST Client API, such as creating an instance and managing instance configuration. This role does not grant any other privileges, so the user might still require additional privileges.

To restrict access on a per-user basis, you should use custom roles, rather than assigning users to the pre-defined `rest-reader` and `rest-writer` roles. For details, see "Controlling Access to Documents and Other Artifacts" on page 22.

Some operations require additional privileges, such as using a database other than the default database associated with the REST API and using the `/eval` and `/invoke` services. These requirements are detailed elsewhere in "Security Requirements" on page 21.

1.4.2 Controlling Access to Documents and Other Artifacts

By default, documents you create with the REST Client API have a read permission for the `rest-reader` role and an update permission for the `rest-writer` role. A user with the `rest-reader` role can read all documents created with the REST API, and a user with the `rest-writer` role can update all documents created with the REST API.

To enable users to create and update documents using the REST API yet restrict access, use custom roles with the `rest-reader` and `rest-writer` execute privileges and suitable default permissions, rather than relying on the pre-defined `rest-reader` and `rest-writer` roles.

The `rest-reader` and `rest-writer` privileges grant users permission to execute REST API code for reading and writing documents, while the default permissions controls access to a document whether it is through the REST API or through other code running on MarkLogic Server. For details, see the *Security Guide*.

The rest-extension-user role enables users to access resource service extension methods. This role is implicit in the other pre-defined roles, but you need to explicitly include it if you're defining custom roles for users that should also be able to use extensions.

For example, suppose you have two groups of users, A and B. Both can create documents using the REST API, but Group A users should not be able to read documents created by Group B, and vice versa. You can implement these restrictions in the following way:

1. Create a GroupA security role.

2. Assign the `rest-reader` and `rest-writer` execute privileges to the GroupA role. Use the privileges, not the base roles. That is, assign these privileges to the role:

   ```
   http://marklogic.com/xdmp/privileges/rest-reader
   http://marklogic.com/xdmp/privileges/rest-writer
   ```

3. If you also want to enable the execution of REST resource extensions, assign the `rest-extension-user` role to the GroupA role. Note that the `rest-extension-user` role provides a base role, not a privilege.

4. Give the GroupA role suitable default permissions. For example, set the default permissions of the role to `update` and `read`.

5. Assign the GroupA role to the appropriate users.

6. Repeat Steps 1-3 for a new GroupB role and assign GroupB to the appropriate users.

Now, users with the GroupA role can create documents with the REST API and read or update them, but users with the GroupB role have no access to documents created by GroupA. Similarly, users with the GroupB role can create documents and read or update them, but users with the GroupA role have no access to documents created by GroupB users. A user with the default `rest-reader` role, however, can read documents created by both GroupA and GroupB users.

Other security configurations are possible. For more details, see the *Security Guide*.

1.4.3 Evaluating Requests Against a Different Database

Most methods support a `database` request parameter that enables the request to be evaluated against a content database other than the default database associated with the REST API instances. Only users with the `http://marklogic.com/xdmp/privileges/xdmp-eval-in` (`xdmp:eval-in`) or equivalent privilege can use this feature.

If you want to enable this capability, you must create a role that enables `xdmp:eval-in`, in addition to appropriate mix of `rest-*` roles.

For details about roles and privileges, see the *Security Guide*.

1.4.4 Evaluating or Invoking Server-Side Code

You can evaluate ad-hoc queries and pre-installed modules on MarkLogic Server using the `/eval` and `/invoke` services, respectively. These services require special privileges, such as `xdmp-eval`, instead of the normal REST API roles like `rest-reader` and `rest-writer`.

For details, see the following:

- "Evaluating an Ad-Hoc Query" on page 380 or `POST:/v1/eval`
- "Evaluating a Module Installed on MarkLogic Server" on page 388 or `POST:/v1/invoke`.

1.5 Terms and Definitions

The following terms and definitions are used in this guide:

Term	Definition
REST	REpresentational State Transfer, an architecture style that, in the context of the REST Client API, describes the use of HTTP to make calls between a client application and MarkLogic Server to create, update, delete and query content and metadata in the database.
resource	An abstraction of a REST Client API service, as presented by the REST architecture.
resource address	A URL that identifies a MarkLogic Server resource. Resource addresses are described in "Understanding REST Resources" on page 25.
rewriter	An XQuery module that interprets the URL of an incoming HTTP request and rewrites it to an internal URL that services the request.
REST API instance	An instantiation of the REST Client API against which applications can make RESTful HTTP requests. An instance consists of an HTTP App Server, a URL rewriter, a content database, a modules database, and the modules that implement the API. For details, see "Administering REST Client API Instances" on page 39.
extension	An user-defined XQuery module that implements additional resource services that are made available through the REST Client API. For details, see "Extending the REST API" on page 345.

Term	Definition
string query	A simple search string constructed using either the default MarkLogic Server search grammar, or a user-defined grammar. For example, “cat” and “cat OR dog” are string queries. For details, see “Querying Documents and Metadata” on page 151.
structured query	The pre-parsed representation of a query, expressed as XML or JSON. Structured queries allow you to express complex queries very efficiently. For details, see “Querying Documents and Metadata” on page 151 and Searching Using Structured Queries in the *Search Developer’s Guide*.
lexicon	A list of unique words or values, either throughout an entire database or within named elements, attributes, or fields. You can also define lexicons that allow quick access to the document and collection URIs in the database. Lexicons are usually backed by a range index. For details, see “Querying Lexicons and Range Indexes” on page 157 and Browsing With Lexicons in the *Search Developer’s Guide*.
endpoint	An XQuery module on MarkLogic Server that is invoked by and responds to an HTTP request for monitoring information.

1.6 Understanding REST Resources

This section covers the basic structure of a REST Client API URL. If you are already familiar with REST resource addressing, you can skip this section. The following topics are covered:

- Addressing a Resource
- Specifying the REST API Version
- Specifying Parameters in a Resource Address

1.6.1 Addressing a Resource

A resource address takes the form of a URL that includes a host name and a port number:

```
http://host:port/version/resource/
```

The host and port must reference a host running MarkLogic Server with a REST Client API instance running on that port. A REST Client API instance is served by an HTTP App Server. For details, see “Creating an Instance” on page 40.

A resource address always includes the API version in URL. For details, see “Specifying the REST API Version” on page 26.

You can optionally include parameters in a resource address as follows:

```
http://host:port/version/resource?param=value&param=value
```

For details, see “Specifying Parameters in a Resource Address” on page 26.

1.6.2 Specifying the REST API Version

To guarantee stable behavior of the REST Client API as new versions are released, each resource address in the REST Client API includes a version number. The examples in this chapter show the version as `LATEST` or simply *version*. The version has the format:

```
v#
```

Where # is the version number. For example, in the initial version of the API, the current version number is 1, so you can access the /documents service using the following URL:

```
http://localhost:8000/v1/documents
```

You can use LATEST to reference the current version, without regard to the actual version number. For example:

```
http://localhost:8000/LATEST/documents
```

The current version number is `v1`.

Note: The version number is only updated when resource addresses and/or parameters have changed. It is not updated when resource addresses and/or parameters are added or removed.

1.6.3 Specifying Parameters in a Resource Address

Resource services accept request parameters to tailor behavior or control input and output format.

To specify multiple parameters, use the ‘?’ sign before the first parameter and the ‘&’ sign before any additional parameters:

```
http://host:port/version/resource?param1=value&param2=value....
```

Some resources only accept parameter values as URL-encoded form data in the request body. Such requests require an input content MIME type of `x-www-form-urlencoded`. You can use the curl option --data-url-encode to set such parameters to a properly encoded value. For details, see “Introduction to the curl Tool” on page 27.

See the *MarkLogic REST API Reference* for a list of parameters available with each resource.

1.7 Understanding the Example Commands

The examples in this guide use the curl command line tool to send HTTP requests that exercise the REST Client API. The examples also use Unix command line syntax. Review this section if you are not familiar with curl or Unix command line syntax. If you are not familiar with RESTful URL conventions, see “Understanding REST Resources” on page 25.

- Introduction to the curl Tool
- Modifying the Example Commands for Windows

1.7.1 Introduction to the curl Tool

`curl` is a command line tool for sending HTTP requests. You are not required to use curl with the REST Client API. You can use any tool or library capable of sending HTTP requests. However, since all the examples in this guide use `curl`, this section introduces you to the most relevant options.

If you do not have `curl`, you can download a copy from http://curl.haxx.se/download.html, or use an equivalent tool. This section provides a brief overview of the `curl` command line options used in this guide. For details, see the `curl` man page or the online documentation at http://curl.haxx.se/docs/.

The `curl` command line is of the form:

```
curl options URL
```

The options most often used in the examples in this guide are summarized in the table below.

Option	Description
`--anyauth`	Have `curl` figure out the authentication method. The method depends on your REST API instance App Server configuration. Alternatively, you can specify an explicit method using options such as `--digest` or `--basic`.
`--user` *`username:password`*	Username and password with which to authenticate the request. Use a MarkLogic Server user that has sufficient privileges to carry out the requested operation. For details, see “Security Requirements” on page 21.
`-X` *`http_method`*	The type of HTTP request (GET, PUT, POST, DELETE) that `curl` should send. If `-X` is not given, GET is assumed.

Option	Description
`-d` *data*	Data to include in the request body. Data may be placed directly on the command line as an argument to `-d`, or read from a file by using `@`*filename*. The examples in this guide usually read from file to simplify the command line. For example, `curl -X POST -d @./my-body.xml ...` reads the post body contents from the file ./my-body.xml. If you need to preserve line breaks in the data in the body, use `--data-binary` instead.
`--data-binary` *data*	Similar to -d, but the input is interpreted as binary by curl. This option prevents curl from applying any transformations to the input. For example, curl removes newlines from non-binary data, so if you pass data from a file containing JavaScript or SPARQL code that uses single line comments (// your comment), you need to use `--data-binary` rather than `-d`. Otherwise, the JavaScript or SPARQL payload will be invalid.
`--data-urlencode` *data*	Similar to `-d`, but curl will URL encode the data. Use this option with methods that expect `x-www-form-urlencoded` input, such as `POST /LATEST/eval`.
`-H` *headers*	HTTP headers to include in the request. This is most often used by the examples to specify `Accept` and `Content-type headers`.
`-i`	Specifies that the curl output should include the HTTP response headers in the output. By default, curl doesn't display the response header, which can make it difficult to see if your request succeeded.

For example, the following command sends a POST request with the contents of the file `my-body.json` in the request body, and specifies the Content-type as `application/json`:

```
$ curl --anyauth --user me:mypassword -X POST -d @my-body.json \
    -H "Content-type: application/json" \
    http://localhost:8000/LATEST/config/query/my-options
```

Note: When reading data for a POST or PUT request body with curl, you must use `--data-binary` rather than `-d` if you need to preserve newlines in the data. For example, use --data-binary when uploading SPARQL or a JavaScript module that uses line-oriented comments (`// a comment`).

1.7.2 Modifying the Example Commands for Windows

The command line examples in this guide use Unix command line syntax, usable from either a Unix or Cygwin command line. If you are the Windows command interpreter, `Cmd.exe`, use the following guidelines to modify the example commands for your environment:

- Omit the “$” character at the beginning of the command. This is the Unix default prompt, equivalent to “>” in Windows.
- For aesthetic reasons, long example command lines are broken into multiple lines using the Unix line continuation character '\'. Remove the line continuation characters and place the entire command on one line, or replace the line continuation characters with the Windows equivalent, '^'.
- Replace arguments enclosed in single quotes (') with double quotes ("). If the single-quoted string contains embedded double quotes, escape the inner quotes.
- Escape any unescaped characters that have special meaning to the Windows command interpreter.

1.8 Overriding the Content Database

Each REST API instance has a default content database associated with. You specify this database when you create the instance, and it cannot be changed subsequently. However, many REST Client API methods support a `database` parameter with which you can select a different content database on a per request basis. Evaluating requests against an alternative database requires additional security privileges; for details, see “Evaluating Requests Against a Different Database” on page 23.

For example, a request of the following form implicitly searches the default content database associated with the instance on port 8000 of localhost:

```
GET http://localhost:8000/LATEST/search?q=dog
```

You can add a database parameter to search a different database:

```
GET http://localhost:8000/LATEST/search?q=dog&database=my-other-db
```

Note that if you’re using multi-statement transactions, you must create, use, and commit (or rollback) on the transaction using the same database. You cannot create a transaction on one database and then attempt to perform an operation such as read, write, or search using the transaction id and a different database.

Not all requests support a `database` parameter. Requests that operate on configuration data, extensions, transforms, and other data stored in the modules database do not support a `database` parameter. For details, see the *MarkLogic REST API Reference*.

You cannot override the modules database associated with the REST instance.

1.9 Performing Point-in-Time Operations

If you need to perform read-only operations spanning multiple requests that must all return results based on a consistent snapshot of the database, you can use the “point-in-time query” feature of the REST Client API. In this context, “query” means a read-only operation, such as a search or document read.

Most read-only request will return an `ML-Effective-Timestamp` header that contains a system timestamp. You can pass the value from this header to subsequent read-only requests via a `timestamp` request parameter to ensure these requests see the same snapshot of the database.

Note that this timestamp must be a timestamp generated by MarkLogic, not an arbitrary value you create. To learn more about point-in-time queries (reads) and timestamps, see Point-In-Time Queries in the *Application Developer's Guide*.

For example, suppose you are incrementally fetching search results in a context in which the database is changing and consistency of results is important. You can capture the `ML-Effective-Timestamp` value from the first request, and pass it to all the subsequent requests via a `timestamp` parameter.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET -i \
    -H "Accept: application/xml" \
    'http://localhost:8000/LATEST/search?q=dog'

HTTP/1.1 200 OK
Content-type: application/xml; charset=utf-8
ML-Effective-Timestamp: 14913561007926020
Server: MarkLogic
Content-Length: 366
Connection: Keep-Alive
Keep-Alive: timeout=5

<search:response snippet-format="snippet"
    total="100" start="1" page-length="10"
    xmlns:search="http://marklogic.com/appservices/search">
  ...
</search:response>

$ curl --anyauth --user user:password -X GET -i \
    -H "Accept: application/xml" \
    'http://localhost:8000/LATEST/search?q=dog&timestamp=14913561007926020&st
art=11'

<search:response snippet-format="snippet"
    total="100" start="11" page-length="10"
    xmlns:search="http://marklogic.com/appservices/search">
  ...
</search:response>
```

Another example use case is reading a large number of documents from the database by URI (or search query) in batches. If you need a consistent snapshot of the documents, use the point-in-time feature.

You can use this feature across different kinds of operations. For example you might get the initial timestamp from a request to `/v1/search`, and then use it to perform a SPARQL query at the same point-in-time via `/v1/graphs/sparql`.

This capability is supported on any operation that accepts a timestamp parameter, including document read (`/documents`), document search (`/search`, `/qbe`, `/values/{name}`), semantic search (`/graphs`, `/graphs/sparql`), and row search (`/rows`). For more details, see the *MarkLogic REST API Reference*.

1.10 Controlling Input and Output Content Type

Input and output to the REST API comes in two forms: Document content and non-document data. Document content can be XML, JSON, text, or binary. Non-document data is anything that is not document content, such as document metadata, queries, query options, search results, and configuration data. Non-document data can usually be either XML or JSON, and you can choose which format you want to work with.

This section includes the following topics that explain how the REST API determines input and output content type, based on URI extension, HTTP headers, and the `format` request parameter.

- General Content Type Guidelines
- Details on Content Type Determination
- Example: Inserting and Reading a Document
- Example: Inserting and Reading Metadata
- Example: Documents With No or Unknown URI Extension
- Example: Mixing Document and Non-Document Data

1.10.1 General Content Type Guidelines

The following guidelines apply to specifying input and output content type for most requests:

- Document content: Rely on the MarkLogic Server MIME type mapping defined for the URI extension.
- Non-document data: Set the request Content-type and/or Accept headers. In most cases, this means setting the header(s) to `application/xml` or `application/json`.

The installation-wide MarkLogic Server MIME type mappings define associations between MIME type, URI extensions, and document format. For example, the default mappings associate the MIME type `application/pdf` and the “pdf” URI extensionwith the binary document format. You can view, change, and extend the mappings in the “Mimetypes” section of the Admin Interface or using the XQuery functions `admin:mimetypes-get` and `admin:mimetypes-add`.

As long as your documents have URI extensions with MIME type mappings and you set the HTTP Content-type and/or Accept headers consistent with your data, these guidelines are all you need. For situations that do not fit this model, see “Details on Content Type Determination” on page 32.

1.10.2 Details on Content Type Determination

This section provides a detailed description of how content type is determined. This information is useful for requests that do not conform to the guidelines in “General Content Type Guidelines” on page 31. For example, you might need this deeper understanding in the following situations:

- Reading or writing documents that have no URI extension or an unrecognized URI extension.
- Reading or writing document content and non-document data in the same request, such as reading a document and its metadata in a single request.
- Creating requests that have both input and output, such as a `POST /LATEST/search` request that has a query in the POST body and search results in the response.
- Requesting non-document data through a browser. Browsers often do not give you full control over the HTTP headers.

The table below summarizes how input and output content type is determined, depending on type of data and the request context (input or output). The content type sources in the third column are listed from highest to lowest precedence. For example, for input document content, the URI extension mapping is used if possible; the Content-type header is only used if there is no mapping available.

Data Type	Context	Precedence of Content Type Sources
Document	Input	Primary: URI extension MIME type mapping, as long as the request does not specify a transform function. Fallback: Content-type header MIME type mapping. For multipart input, the request Content-type header must be `multipart/mixed`, so the Content-type header for each part specifies the MIME type of the content for that part.
	Output	Primary: URI extension MIME type mapping. Fallback: • For text, XML, and JSON documents, the document type (the type of root node on the document). • For binary documents, the Accept header MIME type mapping, except for requests with multipart output. • For multipart output, binary documents with no extension or an unknown extension: `application/x-unknown-content-type` by default.
Non-Document	Input	Primary: The Content-type header MIME type mapping. For multipart input, the request Content-type header must be `multipart/mixed`, so the Content-type header for each part specifies the MIME type of the content for that part. Fallback: The `format` request parameter.
	Output	Primary: The `format` request parameter. Fallback: The Accept header MIME type mapping, except for requests with multipart output.

The `format` request parameter is supported by most REST API methods that accept or produce non-document data. You can set it to one of a limited set of values, usually `xml` or `json`; see the API documentation for individual methods for allowed values.

Requests which accept or produce multipart data behave asymmetrically because the Content-type header (multipart input) or Accept header (multipart output) must be `multipart/mixed`. On input, you can use the part Content-type header to indicate the non-document data format in a given part, but on output you can only use the `format` parameter to request a specific output format. A multi-document write using `POST /LATEST/documents` is an example of an operation with multipart input. Reading a document and its metadata in a single request is an example of an operation with multipart output. On such a read, you can use the `format` parameter to specify the metadata format.

1.10.3 Example: Inserting and Reading a Document

This example demonstrates how the general content type guidelines apply to document content. The example relies on the pre-defined MIME type mapping between the `json` URI extension and the MIME type `application/json`.

The following command inserts a JSON document into the database with URI “example.json”. Because of the MIME type mapping, a JSON document is created, whether or not you specify `application/json` in the request Content-type header.

```
$ curl --anyauth --user user:password -X PUT -d '{"key":"value"}' \
  -i -H "Content-type: anything" \
  http://host:port/LATEST/documents?uri=example.json
```

The following command reads the document just inserted. Whether or not you set the Accept header to `application/json`, MarkLogic Server sets the response Content-type header to `application/json` because the URI extension is `json`.

```
$ curl --anyauth --user user:password -X GET -i \
  http://host:port/LATEST/documents?uri=example.json
...
HTTP/1.1 200 OK
vnd.marklogic.document-format: json
Content-type: application/json; charset=utf-8
...
{"key":"value"}
```

If the URI has no extension or there is no MIME type mapping defined for the extension, MarkLogic Server falls back on sources such as the HTTP Content-type header for input and the document type or Accept header for output. For details, see “Details on Content Type Determination” on page 32.

1.10.4 Example: Inserting and Reading Metadata

This example illustrates how the general content type guidelines apply to non-document data. The example inserts and reads document metadata.

The following command inserts metadata for a document. Assume the file `./mymatadata` contains a JSON representation of document metadata. The request Content-type header tells MarkLogic Server to interpret the metadata in the request body as JSON.

```
$ curl --anyauth --user user:password -X PUT -d @./mymetadata \
  -H "Content-type: application/json" \
  'http://host:port/LATEST/documents?uri=anything&category=metadata'
```

For a complete example, `PUT:/v1/documents` or see “Adding Metadata” on page 57.

The following command reads the metadata for a document. The Accept header tells MarkLogic Server to return the metadata as XML.

```
$ curl --anyauth --user user:password -X GET \
  -H "Accept: application/xml" \
  'http://host:port/LATEST/documents?uri=anything&category=metadata'
```

For a complete example, see `GET:/v1/documents` or “Retrieving Metadata About a Document” on page 65.

If you cannot control the Content-type header for input or the Accept header for output, you can use the `format` request parameter. For details, see “Details on Content Type Determination” on page 32.

```
http://host:port/LATEST/documents?uri=anything&category=metadata&format=json
```

1.10.5 Example: Documents With No or Unknown URI Extension

This example illustrates how the output content type is determined when reading a document with no URI extension or a URI extension that has no MIME type mapping.

The following command inserts a text document into the database at a URI that has no extension. Since there is no extension, MarkLogic Server uses the MIME type mapping defined for `text/plain` in the request Content-type header to determine the document type.

```
curl --anyauth --user user:password -X PUT -i \
  -d '{ "key" : "value" }' -H "Content-type: text/plain" \
  http://host:port/LATEST/documents?uri=no-extension
```

If you leave off the Content-type header or set it to value for which there is no MIME type mapping, a binary document is created because binary is the default document type when there is no extension or MIME type mapping.

The following command reads the document inserted above. The response Content-type header is `text/plain` because the root node of the document is a text node.

```
curl --anyauth --user user:password -X GET -i \
  http://host:port/LATEST/documents?uri=no-extension
```

For binary documents, the root document node type is too generic for most applications. You can use the Accept header to coerce the response Content-type header to a specific MIME type. The content is unaffected. For example, if you read a binary document with no extension that you know actually contains PDF, then the following command returns the document with a Content-type header of `application/pdf`.

```
curl --anyauth --user user:password -X GET -i \
  -H "Accept: application/pdf" \
  http://host:port/LATEST/documents?uri=no-extension
```

If you cannot control the Accept header, then the response Content-type for a binary document is `application/x-unknown-content-type` by default.

1.10.6 Example: Mixing Document and Non-Document Data

This example describes how content type is determined for requests that include both document and non-document data as input or as output. In this example, the non-document data is metadata for a document.

The following example command inserts an XML document and its metadata into the database. Assume the file "multipart-body" contains a `multipart/mixed` POST body with a part for the metadata and a part for the content. For a complete example, see "Loading Content and Metadata Using a Multipart Message" on page 61.

```
curl --anyauth --user user:password -X PUT -d @./multipart-body \
  -i -H "Content-type: multipart/mixed; boundary=BOUNDARY" \
  http://host:port/LATEST/documents?uri=example.xml
```

The metadata format is derived from the Content-type header on the metadata part in the POST body. The document content type is derived from the URI extension of `.xml`. If the document URI did not have an extension, the document content type would be derived from the Content-type header on the document part in the POST body.

The following example command reads a JSON document and its metadata. The response is multipart data, with one part containing the metadata and one part containing the document. Since the Accept header must be multipart/mixed in this case, the `format` parameter is used to request the metadata as JSON.

```
curl --anyauth --user user:password -X GET -i \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://host:port/LATEST/documents?uri=example.json?format=json'
```

In the response, the Content-type header for the metadata part is set to `application/json` because of the `format` parameter value. The Content-type header for the document part is set to `application/json` because the document URI is `.json`. If the document URI had no extension, the Content-type header for the document part would still be `application/json` as long as root node of the document indicates a JSON document.

For a complete example of reading a document and its metadata, see “Retrieving Content and Metadata in a Single Request” on page 66.

1.11 Error Reporting

This section covers the error reporting conventions followed by the REST Client API.

If a request to a REST Client API instance fails, an error status code is returned and additional error detail is provided in the response body. The error response content type can be either XML or JSON. The format is derived from the following sources, in order of highest to lowest precedence:

- The MIME type in the X-Error-Accept header.
- The MIME type in the Accept header, if it signifies XML or JSON.
- The default error format configured into the REST API instance. For details, see “Creating an Instance” on page 40.

If you do not set `error-format` when creating a REST instance, it defaults to JSON.

Use X-Error-Accept to avoid undesired interaction with the Accept header. For example, if you set the Accept header on a read request to XML in order to read an XML document, then any error response for that request will be XML. If your application expects JSON errors, then you can use X-Error-Accept to request JSON errors without affecting the response content type for the success case. For example:

```
curl --anyauth --user user:password -X GET i \
  -H "Accept: application/xml" -H "X-Error-Accept: application/json" \
  http://localhost:8000/v1/documents?uri=nonexistent.xml
```

The following example shows the XML error output for a request specifying unsupported parameters. The return status code is 400 (Bad Request) and the details of the error, identifying the failure as a `REST-UNSUPPORTEDPARAM` exception, are contained in the response body.

```
HTTP/1.1 400 Bad Request
Content-type: application/xml
Server: MarkLogic
Content-Length: 333
Connection: close

<error-response xmlns="http://marklogic.com/xdmp/error">
  <status-code>400</status-code>
  <status>Bad Request</status>
  <message-code>REST-UNSUPPORTEDPARAM</message-code>
  <message>REST-UNSUPPORTEDPARAM: (rest:UNSUPPORTEDPARAM) Endpoint
does not support query parameter: unknown</message>
</error-response>
```

The following example is the same error, with the error detail returned as JSON:

```
HTTP/1.1 400 Bad Request
Content-type: application/json
Server: MarkLogic
Content-Length: 206
Connection: close

{
  "errorResponse": {
    "status-code": "400",
    "status": "Bad Request",
    "message-code": "REST-UNSUPPORTEDPARAM",
    "message": "REST-UNSUPPORTEDPARAM: (rest:UNSUPPORTEDPARAM)
Endpoint does not support query parameter: unknown"
  }
}
```

Errors that can be corrected by the client application, such as an invalid parameter or an unsupported HTTP method, are usually reported as a 4XX error with a `REST-` or `RESTAPI-` message code.

Errors that cannot be addressed by the client application are usually reported as a 500 Internal Server Error. A 500 error does not necessarily mean that the problem cannot be corrected or that MarkLogic Server got an internal error. A 500 error usually indicates a problem that requires correction on the server host rather than in the client application.

An example of a 500 error that is correctable on the server side is failing to create an element range index required to support an operation. If a client application uses the `/search` service with a search constraint that requires a non-existent index, a 500 error is returned. To correct the error, an administrator would create the required index in MarkLogic Server.

Content transformations and resource service extensions should report errors using `RESTAPI-SRVEXERR`, as described in “Reporting Errors” on page 328.

2.0 Administering REST Client API Instances

To use the REST Client API, you must create an *instance* of the API. The instance includes an HTTP App Server that exposes the API services, a URL rewriter, and the XQuery modules that implement the API.

- What Is an Instance?
- Creating an Instance
- Example: Creating an Instance
- Removing an Instance
- Retrieving Configuration Information
- Configuring Instance Properties

2.1 What Is an Instance?

Before you can use the REST Client API, you must have access to an *instance* that consists of an HTTP App Server specially configured to handle REST Client API requests, a content database, and a modules database.

> **Note:** Each REST API instance can host a single application. If you have multiple REST API applications, you must create an instance for each one, and each one must have its own modules database.

When you install MarkLogic Server, a pre-configured REST API instance is available on port 8000. This instance uses the Documents database as the content database and the Modules database as the modules database.

You can also use the REST Client API in conjunction with the REST Management API on port 8002.

The instance on port 8000 is convenient for getting started, but you will usually create a dedicated instance for production purposes. This chapter covers creating and managing your own instance.

The default content database associated with a REST API instance can be created for you when the instance is created, or you can create it separately before making the instance. You can associate any content database with an instance. Administer your content database as usual, using the Admin Interface, XQuery or JavaScript Admin API, or REST Management API.

Many REST API methods support a database request parameter with which you can override the default content database on a per request basis. For example, if you want to search a database other than the content database associated with the instance, you can you can use a request such as `GET /LATEST/search?database=other-db&q=dog`. Using another database requires extra privileges; for details, see “Evaluating Requests Against a Different Database” on page 23.

The modules database can be created for you during instance creation, or you can create it separately before making the instance. If you choose to pre-create the modules database, it must not be shared across instances. Special code is inserted into the modules database during instance creation. The modules database also holds any persistent query options, extensions, and content transformations you create for your application.

Aside from the instance properties described in this chapter, you cannot pre-configure the App Server associated with an instance. However, once the instance is created, you can further customize properties such as request timeouts using the Admin Interface, XQuery or JavaScript Admin API, or REST Management API.

2.2 Creating an Instance

This section describes how to create your own REST API instance. You can specify the instance name, the port, and the content and modules databases when you create your own instance.

To create a new REST instance, send a POST request to the `/rest-apis` service on port 8002 with a URL of the form:

```
http://host:8002/version/rest-apis
```

You can use either the keyword LATEST or the current version for *version*. For details, see “Specifying the REST API Version” on page 26.

For a complete example, see “Example: Creating an Instance” on page 42.

The POST body should contain instance configuration information in XML or JSON. Set the HTTP `Content-type` header to `application/xml` or `application/json` to indicate the content format. MarkLogic Server responds with HTTP status 201 if the instance is successfully created.

The configuration data must specify at least a name for the instance. Optionally, you can specify additional instance properties such as a group name, database name, modules database name, and port.

The following example configuration creates an instance named RESTstop on port 8020, attached to the Documents database, using the default modules database, RESTstop-modules.

XML	JSON

<rest-api xmlns="http://marklogic.com/rest-api">
  <name>RESTstop</name>
  <database>Documents</database>
  <port>8020</port>
</rest-api>
	{ "rest-api": {
	  "name": "RESTstop",
	  "database": "Documents",
	  "port": "8020"
	} }

The following example creates an instance using the preceding XML configuration information, saved to a file named “config.xml”. To use the JSON example configuration data, change the Content-type header value to `application/json`.

```
$ cat config.xml
<rest-api xmlns="http://marklogic.com/rest-api">
  <name>RESTstop</name>
  <database>Documents</database>
  <port>8020</port>
</rest-api>

# Windows users, see Modifying the Example Commands for Windows
$ curl -X POST --anyauth --user user:password -d @"./config.xml" \
    -H "Content-type: application/xml" \
    http://localhost:8002/LATEST/rest-apis
```

Once the instance is created, you can customize and administer the App Server portion using the Admin UI, just as you can with any MarkLogic Server App Server. You can also set selected instance properties using the /config/properties service; for details, see “Configuring Instance Properties” on page 47.

The following table provides more details on the configuration components.

Element/key name	Default	Description
`name`	none	Required. The name of the instance.
`group`	Default	Optional. The group in which to create the App Server.
`database`	*instance_name*	Optional. The name of the content database to associate with the instance. If the database does not exist, MarkLogic Server creates a new database with default settings and three forests. Use `forests-per-host` to change the number of forests.
`error-format`	`json`	Optional. Specify the format in which errors will be returned to clients when no suitable MIME type can be derived from the request headers. Allowed values: `json` or `xml`. Default: `json`.
`forests-per-host`	`3`	Optional. The number of forests to create per host for the content database.

Element/key name	Default	Description
`modules-database`	*instance_name*-modules	Optional. The name of the modules database to associate with the instance. If the database does not exist, MarkLogic Server creates a new database with default settings and one forest.
`port`	The next available port number, starting with 8003.	Optional. The port number to associate with the App Server.
`xdbc-enabled`	true	Optional. Whether or not to enable XDBC services in the instance. XDBC services must be enabled to use the `/eval` and `/invoke` services.

Warning Changing the configuration of the Application Server for a REST instance is not advised without expert guidance. REST Server behavior is indeterminate after changes to the Application Server. Do not change the root, port, modules, database, error format, error handler, or rewriter, as these are known to cause the REST instance to fail.

2.3 Example: Creating an Instance

Follow this procedure to create a “rest-example” database with 3 forests and a “rest-example” HTTP App Server attached to the database. The example uses a minimal JSON configuration as input. The equivalent XML configuration is shown after the procedure.

1. Create the instance and database by sending a POST request to `/rest-apis` on port 8002. MarkLogic Server responds with status code 201 (Created). For example:

```
$ curl --anyauth --user user:password -i -X POST \
    -d'{"rest-api":{"name":"rest-example"}}' \
    -H "Content-type: application/json" \
    http://localhost:8002/LATEST/rest-apis
```

   By specifying only the instance name, we request MarkLogic Server to create a database with the same name, including one forest, and to use the next available port above 8002.

2. Navigate to the Admin Interface in your browser:

```
http://localhost:8001
```

   Observe the creation of a database named “rest-example” with forests named “rest-example-1”, “rest-example-2”, and “rest-example3”; and an HTTP App Server

called “rest-example” attached to this database. A modules database is also created for the instance.

3. Test the instance by navigating to the following URL in your browser. (Your port number may differ if port 8003 was already in use). When prompted for a username and password, authenticate with a user that has the `rest-reader` role.

```
http://localhost:8003
```

If the instance is properly configured, you will see a short HTML summary of capabilities provided by the instance.

If you prefer to configure your instance using XML, use the following for the POST body and change the HTTP Content-type header to `application/xml` in Step 1, above.

```
<rest-api xmlns="http://marklogic.com/rest-api">
  <name>rest-example</name>
</rest-api>
```

2.4 Removing an Instance

To remove an instance of the REST Client API, send a DELETE request to the `/rest-apis` service on port 8002.

Warning You usually should not apply this procedure to the pre-configured REST API instance on port 8000. Doing so can disable other services on that port, including XDBC, Query Console, and the REST Management API.

To remove an instance but leave the content database intact, send the DELETE request with a URL of the form:

```
http://host:8002/version/rest-apis/instance-name
```

Where *`instance-name`* is the name chosen when the instance is created. MarkLogic Server responds with 204 on success and 400 (“Bad Request”) if the instance is not found or more than one occurrence of the instance name is found.

Note: Deleting an instance causes MarkLogic Server to restart.

To also remove the content and/or modules database and forests associated with the instance, use the `include` request parameter.

Note: All content in the removed databases is lost.

The following example removes the “RESTstop” instance, including the content and modules databases, and the forests attached to the databases:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl -X DELETE --anyauth --user user:password \
  'http://localhost:8002/LATEST/rest-apis/RESTstop?include=content&inc
lude=modules'
```

If the instance name is not unique across groups, use the `group` request parameter to differentiate the target instance. The following URL requests removal of the instance of "RESTstop" in the "stage" group:

```
$ curl -X DELETE --anyauth --user user:password \
  http://localhost:8002/LATEST/rest-apis/RESTstop?group=stage
```

2.5 Retrieving Configuration Information

You can retrieve instance configuration information, as XML or JSON, by sending a GET request to the `/rest-apis` service on port 8002.

- Retrieving Configuration for All Instances
- Retrieving Instance Configuration by Content Database
- Retrieving Instance Configuration by Instance Name

2.5.1 Retrieving Configuration for All Instances

To retrieve configuration details about an instance, send a GET request to the `/rest-apis` service on port 8002 with a URL of the form:

```
http://host:8002/version/rest-apis
```

MarkLogic Server responds with configuration information about all instances, in XML or JSON. Use the Accept header or format request parameter to select the output content type. For details, see "Controlling Input and Output Content Type" on page 31.

The following example requests information in JSON about all instances.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl -X GET --anyauth --user user:password \
    -H "Accept: application/json" \
    http://localhost:8002/LATEST/rest-apis
```

The response includes configuration details about 2 instances, the pre-installed App-Services instance on port 8000, and another instance named "samplestack" on port 8006.

```
{ "rest-apis": [
    {
      "name": "App-Services",
      "group": "Default",
      "database": "Documents",
      "modules-database": "Modules",
      "port": "8000"
```

```
    },
    {
      "name": "samplestack",
      "group": "Default",
      "database": "samplestack",
      "modules-database": "samplestack-modules",
      "port": "8006"
    }
]}
```

Requesting the same data as XML results in the following:

```
<rapi:rest-apis xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:rest-api>
    <rapi:name>App-Services</rapi:name>
    <rapi:group>Default</rapi:group>
    <rapi:database>Documents</rapi:database>
    <rapi:modules-database>Modules</rapi:modules-database>
    <rapi:port>8000</rapi:port>
  </rapi:rest-api>
  <rapi:rest-api>
    <rapi:name>samplestack</rapi:name>
    <rapi:group>Default</rapi:group>
    <rapi:database>samplestack</rapi:database>
    <rapi:modules-database>samplestack-modules</rapi:modules-database>
    <rapi:port>8006</rapi:port>
  </rapi:rest-api>
</rapi:rest-apis>
```

You can also constrain the results to a particular content database or a specific instance. See “Retrieving Instance Configuration by Content Database” on page 45 and “Retrieving Instance Configuration by Instance Name” on page 46.

2.5.2 Retrieving Instance Configuration by Content Database

To retrieve configuration details about instances associated with a specific database, send a GET request to the `/rest-apis` service on port 8002 with a URL of the form:

```
http://host:8002/version/rest-apis?database=database_name
```

MarkLogic Server responds with configuration information about all instances serving the named database, in XML or JSON. Use the Accept header or `format` request parameter to select the output type. For details, see “Controlling Input and Output Content Type” on page 31.

The following example requests information in JSON about instances associated with the database named “Documents”:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl -X GET --anyauth --user user:password \
    -H "Accept: application/json" \
    'http://localhost:8002/LATEST/rest-apis?database=Documents'
```

The response contains data similar to the following, depending on the requested response format.

Format	Response Data

XML	`<rapi:rest-apis xmlns:rapi="http://marklogic.com/rest-api">`
	`  <rapi:rest-api>`
	`  <rapi:name>App-Services</rapi:name>`
	`  <rapi:group>Default</rapi:group>`
	`  <rapi:database>Documents</rapi:database>`
	`  <rapi:modules-database>Modules</rapi:modules-database>`
	`  <rapi:port>8000</rapi:port>`
	`  </rapi:rest-api>`
	`</rapi:rest-apis>`

JSON	`{ "rest-apis": [`
	`  {`
	`  "name": "App-Services",`
	`  "group": "Default",`
	`  "database": "Documents",`
	`  "modules-database": "Modules",`
	`  "port": "8000"`
	`  }`
	`] }`

2.5.3 Retrieving Instance Configuration by Instance Name

To retrieve configuration details about a specific instance, send a GET request to the `/rest-apis` service on port 8002 with a URL of the form:

```
http://host:8002/version/rest-apis/instance_name
```

MarkLogic Server responds with configuration information about the named instance, in XML or JSON. Use the Accept header or `format` request parameter to select the output type. For details, see “Controlling Input and Output Content Type” on page 31.

The following example requests information in JSON about the “App-Services” instance, which is the pre-configured instance on port 8000.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl -X GET --anyauth --user user:password \
    -H "Accept: application/json" \
    http://localhost:8002/LATEST/rest-apis/App-Services
```

The response contains data similar to the following, depending on the requested response format.

Format	Response Data

XML	`<rapi:rest-api xmlns:rapi="http://marklogic.com/rest-api">`
	`  <rapi:name>App-Services</rapi:name>`
	`  <rapi:group>Default</rapi:group>`
	`  <rapi:database>Documents</rapi:database>`
	`  <rapi:modules-database>Modules</rapi:modules-database>`
	`  <rapi:port>8000</rapi:port>`
	`</rapi:rest-api>`

JSON	`{`
	`  "name": "App-Services",`
	`  "group": "Default",`
	`  "database": "Documents",`
	`  "modules-database": "Modules",`
	`  "port": "8000"`
	`}`

Use the `group` parameter to disambiguate the instance if there is more than one instance or App Server with the same *instance_name* in your MarkLogic Server cluster.

2.6 Configuring Instance Properties

You can use the `/config/properties` and `/config/properties/{name}` services to manage properties that control the behavior of a REST Client API instance. The following topics are covered:

- Instance Configuration Properties
- Listing Instance Property Settings
- Setting Instance Properties
- Resetting Instance Properties

2.6.1 Instance Configuration Properties

You can read and set the following instance properties to control global behaviors of your REST Client API instance:

Name	Description
`debug`	Turn debug logging on and off. Allowed values: `true`, `false`. Default: `false`.
`document-transform-all`	Whether or not to apply a default read transform to every document read, regardless of user. When false, a default transform does not modify documents read by a user with `rest-writer` or `rest-admin role` or equivalent privileges. Use `document-transform-out` to specify a default read transformation. Default: `true`.
`document-transform-out`	The name of a default content transformation to apply when retrieving documents from the database using `/documents`. If set, this must be the name of a transform installed using `/config/transforms/{name}`. The default transform is applied before any transform specified with the `transform` request parameter. Default: apply no transformations. For details, see “Working With Content Transformations” on page 323.
`update-policy`	This property controls the availability and behavior of conditional GET, PUT, POST, and DELETE on the `/documents` service and metadata merging policy. Allowed values: `version-required`, `version-optional`, `merge-metadata`, `overwrite-metadata`. Default: `merge-metadata`. For details, see “Using Optimistic Locking to Update Documents” on page 131 and “Client-Side Cache Management Using Content Versioning” on page 137.
`validate-options`	Whether or not to validate query options when they are created or updated using `/config/query*`. When option validation is enabled, improperly structured query options are rejected. Allowed values: `true`, `false`. Default: `true`.

Name	Description
`validate-queries`	Whether or not validate the a query before applying it to a search. This is a development debugging aid. For example, by default, a syntactically incorrect query is treated as an empty and-query that matches all documents. Enabling this property causes you to get an error report instead. You should not enable this property in a production deployment. Allowed values: `true`, `false`. Default: `false`.
`config-directory`	Controls the directory where configuration is stored within the modules database. This property controls all operations on persisted alerting rules and query options as well as persistence of the REST server properties. Allowed values: `group-appserver`, `global`. This value affects the leading part of the path for a configuration resource as follows: • `group-appserver`: `/GROUP_NAME/APPSERVER_NAME/` (default) • `global`: `/marklogic.rest.config.global/` **Note:** We strongly recommend that you set the property once during bootstrapping of the REST API server and never change this property.
`content-versions` (deprecated)	Deprecated as of MarkLogic Server v7.0-3. Use the `update-policy` configuration property instead.

2.6.2 Listing Instance Property Settings

To list all available properties and their settings, send a GET request of the following form to the `/config/properties` service:

```
http://host:port/version/config/properties
```

To retrieve the setting of a single property, send a GET request of the following form, where *`property-name`* is one of the property names listed in “Instance Configuration Properties” on page 48.

```
http://host:port/version/config/properties/property-name
```

You can retrieve property settings as XML or JSON. Use the Accept header or `format` request parameter to select the output type. For details, see “Controlling Input and Output Content Type” on page 31. The default content type is XML.

The following example command retrieves all property settings as XML:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password \
    http://localhost:8000/LATEST/config/properties

<rapi:properties xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:content-versions>none</rapi:content-versions>
  <rapi:debug>false</rapi:debug>
  <rapi:document-transform-all>true</rapi:document-transform-all>
  <rapi:document-transform-out/>
  <rapi:update-policy>merge-metadata</rapi:update-policy>
  <rapi:validate-options>true</rapi:validate-options>
  <rapi:validate-queries>false</rapi:validate-queries>
</rapi:properties>
```

The following example retrieves the same information as JSON:

```
$ curl --anyauth --user user:password \
    -H "Accept: application/json" \
    http://localhost:8000/LATEST/config/properties

{
  "content-version":"none",
  "debug":false,
  "document-transform-all":true,
  "document-transform-out":"",
  "update-policy":"merge-metadata",
  "validate-options":true,
  "validate-queries":false
}
```

The following example retrieves the setting for just the “debug” property as JSON:

```
$ curl --anyauth --user user:password \
    -H "Accept: application/json" \
    http://localhost:8000/LATEST/config/properties/debug

{ "debug" : false }
```

The equivalent XML output is shown below:

```
<rapi:debug xmlns:rapi="http://marklogic.com/rest-api">
  false
</rapi:debug>
```

2.6.3 Setting Instance Properties

You can update instance property settings by sending a PUT request to either `/config/properties` or `/config/properties/{name}`. Using `/config/properties` allows you to set multiple properties in a single call.

To set one or more properties, send a PUT request to `/config/properties` with a URL of the form:

```
http://host:port/version/config/properties
```

The body of the request should contain an XML or JSON properties structure containing only the properties you wish to change. When using XML, the `<properties/>` structure must be in the namespace "http://marklogic.com/rest-api". See the examples below.

To set a specific property, send a PUT request of the following form to `/config/properties/{name}`, where property-name is the property you wish to change. For a list of property names, see "Instance Configuration Properties" on page 48.

```
http://host:port/version/config/properties/property-name
```

Property settings can be specified in either XML or JSON. Use the Content-type header or `format` request parameter to select the output content type. For details, see "Controlling Input and Output Content Type" on page 31.

The following example sets the "debug" and "validate-options" properties using XML input to `/config/properties`:

```
$ cat props.xml
<properties xmlns="http://marklogic.com/rest-api">
  <debug>true</debug>
  <validate-options>false</validate-options>
</properties>

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -d@"./props.xml" \
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/config/properties
```

The equivalent JSON input is shown below:

```
{
  "debug" : "true",
  "validate-options" : "true"
}
```

The following example sets the "debug" property to true using XML input to `/config/properties/{name}`:

```
$ cat debug-prop.xml
<debug xmlns="http://marklogic.com/rest-api">
  true
</debug>

$ curl --anyauth --user user:password -X PUT -d@"./debug-prop.xml" \
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/config/properties/debug
```

The equivalent JSON input is:

```
{"debug" : "true"}
```

2.6.4 Resetting Instance Properties

To reset all properties to their default values, send a DELETE request of the following form to /config/properties:

```
http://host:port/version/config/properties
```

The following example command resets all instance properties:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X DELETE \
    http://localhost:8000/LATEST/config/properties
```

To reset just one property to its default value, send a DELETE request of the following form to /config/properties/{name}:

```
http://host:port/version/config/properties/property-name
```

Where property-name is one of the properties listed in “Instance Configuration Properties” on page 48.

The following example command resets just the “debug” property:

```
$ curl --anyauth --user user:password -X DELETE \
    http://localhost:8000/LATEST/config/properties/debug
```

3.0 Manipulating Documents

This chapter discusses the following topics related to using the REST Client API to create, read, update and delete documents and metadata:

- Summary of Document Management Services
- Loading Content into the Database
- Retrieving Documents from the Database
- Partially Updating Document Content or Metadata
- Performing a Lightweight Document Check
- Removing Documents from the Database
- Using Optimistic Locking to Update Documents
- Client-Side Cache Management Using Content Versioning
- Working with Binary Documents
- Working with Temporal Documents
- Working with Metadata

3.1 Summary of Document Management Services

This section gives a brief summary of the REST Client API services available for creating, reading, updating, and delete documents. This section covers the following topics:

- Summary of the /documents Service
- Summary of the /graphs Service

3.1.1 Summary of the /documents Service

Use the `/documents` service to create, read, update, and delete document content and metadata for XML, JSON, text, and binary documents, including XML documents containing embedded semantic RDF triple data. To load documents containing only RDF triples, use the `/graph` service; for details, see "Summary of the /graphs Service" on page 55.

The following table summarizes the supported operations:

Operation	Method	Description
Create/Update	PUT	Create or update the content or metadata of a document.
Create	POST	Create a document with a URI automatically generated by MarkLogic Server.
Partial Update	POST	Update a portion of the content of an XML or JSON document, or update metadata of any document.
Multi-Document Write	POST	Create or update multiple the content and/or metadata of multiple documents in a single request.
Retrieve	GET	Retrieve content and/or metadata for one or more documents.
Delete	DELETE	Remove a document, or remove or reset document metadata.
Test	HEAD	Test for the existence of a document or determine the size.

The service supports the following additional document features through request parameters:

- Transaction control
- Transformation of content during ingestion
- Content repair during ingestion
- Transformation of results during document insertion or retrieval
- Conditional document insertion using optimistic locking
- Conditional reads based on content versioning
- Multi-request reads at fixed point-in-time

Note: XML, JSON and text documents must use UTF-8 encoding.

Note: The service does not support content conversions except through custom transformations. For example, you cannot retrieve an XML document as JSON.

You can also use `POST:/v1/documents/protection` to protect temporal documents from various levels of update. For details, see "Working with Temporal Documents" on page 141 and the *MarkLogic REST API Reference*.

This chapter only covers single-document operations. For multi-document operations, see "Reading and Writing Multiple Documents" on page 249.

3.1.2 Summary of the /graphs Service

Use the `/graphs` service to create, read, update, and delete documents containing RDF triples. For triples embedded in XML documents, use the `/documents` service.

The following table summarizes the supported operations:

Operation	Method	Description
Create/Update	PUT	Create or replace triples in a named graph or the default graph.
Update	POST	Merge triples into a named graph or the default graph.
Retrieve	GET	Retrieve a named graph or the default graph.
Delete	DELETE	Remove triples in a graph.
Test	HEAD	Test for the existence of a graph in the database or retrieve the headers that would be returned by a GET request.

Though a graph can include both triples embedded in XML documents and triples from a pure triples document (one with a `<sem:triples>` root element), the `/graphs` service does not operate on embedded triples. For example, if you make a DELETE request on a graph that includes embedded triples, the embedded triples and their containing document are unaffected, and therefore, the graph continues to exist.

For details, see Loading Triples Using the REST API in *Semantics Developer's Guide*.

3.2 Loading Content into the Database

This section focuses on ingesting whole documents and their metadata into the database. To learn about modifying just a portion of a document, see "Partially Updating Document Content or Metadata" on page 68.

To insert content or metadata into the database, make a PUT or POST request to the `/documents` service. This section covers the following topics:

- Loading Content
- Adding Metadata

- Loading Content and Adding Metadata in the Same Request
- Automatically Generating Document URIs
- Loading Triples
- Controlling Access to Documents Created with the REST API
- Transforming Content During Ingestion

3.2.1 Loading Content

This section describes how to insert or update an entire document at a user-defined database URI. If you want to have MarkLogic Server generate document URIs automatically, see "Automatically Generating Document URIs" on page 62. You can also modify only a portion of a document; for details, see "Partially Updating Document Content or Metadata" on page 68. For loading multiple documents in a single request, see "Reading and Writing Multiple Documents" on page 249.

To insert or update an XML, JSON, text, or binary document, make a PUT request to a URL of the form:

```
http://host:port/version/documents?uri=document_uri
```

When constructing the request:

1. Set the `uri` parameter to the URI of the destination document in the database.

2. Place the content in the request body.

3. Specify the MIME type of the content in the `Content-type` HTTP header. The Content-type header does not necessarily determine the document type; for details, see "Controlling Input and Output Content Type" on page 31.

   Note: XML, JSON and text documents must use UTF-8 encoding.

Documents you create with the REST Client API have a read permission for the `rest-reader` role and an update permission for the `rest-writer` role. To restrict access, use custom roles. For details, see "Controlling Access to Documents and Other Artifacts" on page 22.

The following example command sends a request to insert the contents of the file `./my.xml` into the database as an XML document with URI `/xml/example.xml`:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -T ./my.xml \
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/documents?uri=/xml/example.xml
```

If the MIME type is not set in the HTTP Content-type header, MarkLogic Server uses the file extension on the document URI to determine the content format, based on the MIME type mappings defined for your installation. For details, see “Controlling Input and Output Content Type” on page 31.

You can also set metadata such as collections, permissions, and named properties when loading content. See “Loading Content and Adding Metadata in the Same Request” on page 60.

You can have MarkLogic Server generate document URIs for you. For details, see “Automatically Generating Document URIs” on page 62.

3.2.2 Adding Metadata

This section describes inserting or updating metadata using a PUT request to `/documents`. The following topics are covered:

- Inserting or Updating Metadata
- Example: Replacing One Metadata Category Using XML
- Example: Replacing Multiple Metadata Categories Using XML
- Example: Replacing Multiple Metadata Categories Using JSON

3.2.2.1 Inserting or Updating Metadata

This section describes how to insert or update metadata independent of the document contents. To bundle content and metadata together, see “Loading Content and Adding Metadata in the Same Request” on page 60.

To insert or update only metadata for a document, make a PUT request to a URL of the form:

```
http://host:port/version/documents?uri=document_uri&category=metadata_
category
```

Where `category` can appear multiple times, with the values described in “Metadata Categories” on page 142.

> **Note:** You cannot supply metadata via request parameters when there is no document content. In this case, you must place the XML or JSON metadata in the request body.

When constructing the request:

1. Set the `category` parameter to the type of metadata to insert or replace. Specify `category` multiple times to include more than one type of metadata.

2. Place the metadata in the request body. You can supply metadata using XML or JSON. For metadata format details, see “Working with Metadata” on page 142.

3. Specify the metadata format in the HTTP `Content-type` header or the `format` parameter. The `Content-type` header generally take precedence; see below.

If the `Content-type` header MIME type is a MIME type that signifies XML or JSON, such as `application/xml` or `application/json`, then the Content-type header determines how MarkLogic interprets the request body. If the `Content-type` header is absent or you cannot set it to a compatible MIME type, use the `format` request parameter to specify the metadata format. For details, see "Controlling Input and Output Content Type" on page 31.

Metadata category merging is not available.

A PUT request for metadata completely replaces each category of metadata specified in the request. For example, a PUT request for collections replaces all existing collections.

When the category is `metadata`, all metadata is replaced or reset to default values.

Note: When setting permissions, at least one update permission must be included.

Note: Any explicitly specified permissions are combined with the default permissions for the role of the current user.

Metadata for categories other than those named by the `category` parameter(s) are ignored. For example, if the request body contains metadata for both collections and properties, but only `category=collections` is given in the URL, then only the collections are updated.

3.2.2.2 Example: Replacing One Metadata Category Using XML

The following example places the document with URI `/xml/example.xml` into the "interesting" collection by specifying `category=collections`. The document is removed from any other collections. The metadata XML in the request body defines the name of the collection(s).

```
$ cat metadata.xml
<rapi:metadata xmlns:rapi="http://marklogic.com/rest-api"
               xmlns:prop="http://marklogic.com/xdmp/property">
  <rapi:collections>
    <rapi:collection>interesting</rapi:collection>
  </rapi:collections>
</rapi:metadata>

# Windows users, see Modifying the Example Commands for Windows
$ curl -X PUT -T ./metadata.xml -H "Content-type: application/xml" \
    --anyauth --user user:password \
    'http://localhost:8000/LATEST/documents?uri=/xml/example.xml&categ
ory=collections'
```

3.2.2.3 Example: Replacing Multiple Metadata Categories Using XML

This example replaces multiple types of metadata on the document with URI /xml/example.xml by specifying multiple category parameters. The metadata in the request body defines a collection name ("interesting") and a property ("my-property"). The request URL includes `category=collections` and `category=properties`. Any collections or properties previously set for the document /xml/example.xml are replaced with the new values.

```
$ cat > metadata.xml
<rapi:metadata xmlns:rapi="http://marklogic.com/rest-api"
               xmlns:prop="http://marklogic.com/xdmp/property">
  <rapi:collections>
    <rapi:collection>interesting</rapi:collection>
  </rapi:collections>
  <prop:properties>
    <my-property>value</my-property>
  </prop:properties>
</rapi:metadata>

# Windows users, see Modifying the Example Commands for Windows
$ curl -X PUT -T ./metadata.xml -H "Content-type: application/xml" \
    --anyauth --user user:password \
    'http://localhost:8000/LATEST/documents?uri=/xml/example.xml&categ
ory=collections&category=properties'
```

3.2.2.4 Example: Replacing Multiple Metadata Categories Using JSON

This example replaces multiple types of metadata on the document with URI /xml/example.xml by specifying multiple category parameters. The JSON metadata in the request body defines a collection name ("interesting") and a property ("my-property"). The request URL includes `category=collections` and `category=properties`. Any collections or properties previously set for the document /xml/example.xml are replaced with the new values.

```
$ cat > metadata.json
{
  "collections":["interesting"],
  "properties": {
    "my-property":"name"
  }
}

# Windows users, see Modifying the Example Commands for Windows
$ curl -X PUT -T ./metadata.json --anyauth --user user:password \
    -H "Content-type: application/json" \
    'http://localhost:8000/LATEST/documents?uri=/xml/example.xml&categ
ory=collections&category=properties'
```

The example uses the Content-type header to communicate the metadata content type to MarkLogic Server. The content type can also be specified using the `format` request parameter; for details, see "Controlling Input and Output Content Type" on page 31.

3.2.3 Loading Content and Adding Metadata in the Same Request

You can update content and metadata for a document in a single request using the following methods. You must choose one or the other; they cannot be combined.

- Loading Content and Metadata Using Request Parameters
- Loading Content and Metadata Using a Multipart Message

For loading multiple content and/or metadata for multiple documents in a single request, see “Reading and Writing Multiple Documents” on page 249.

3.2.3.1 Loading Content and Metadata Using Request Parameters

Use this method when you want to specify metadata using request parameters. To load content and include metadata in the request parameters, send a PUT request of the following form to the/documents service:

```
http://host:port/version/documents?uri=doc_uri&metadata_param=value
```

Where *metadata_param* is one of `collection`, `perm:`*role*, `prop:`*name*, `quality`, or `value:`*key*. For example, to set a document property named “color” to red, include `prop:color=red` in the URL.

When constructing the request:

1. Set the `uri` parameter to the URI of the destination document in the database.
2. Place the content in the request body.
3. Specify the MIME type of the content in the `Content-type` HTTP header.
4. Specify the value of one or more metadata categories through request parameters, such as `collection` or `prop`.

If the MIME type is not set in the Content-type header, MarkLogic Server uses the file extension on the document URI to determine the content format, based on the MIME type mapping defined for the database. MIME type mappings for file suffixes are defined in the Admin Interface. For details, see “Controlling Input and Output Content Type” on page 31.

The following example inserts a binary document with the URI `/images/critter.jpg` into the database, adds it to the “animals” collection, and sets a “species” property:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl -X PUT -T ./critter.jpg --anyauth --user user:password \
    -H "Content-type: image/jpeg" \
    'http://localhost:8000/LATEST/documents?uri=/images/critter.jpg&co
llection=animals&prop:species="canus lupus"'
```

Alternatively, you can pass JSON or XML metadata in the request body with the content. See “Loading Content and Metadata Using a Multipart Message” on page 61. You cannot combine the two methods.

3.2.3.2 Loading Content and Metadata Using a Multipart Message

Use this method when you want to insert or update both content and metadata for a document in a single request, and you want to specify the metadata as JSON or XML in the request body. You can also specify metadata using request parameters; for details, see “Loading Content and Metadata Using Request Parameters” on page 60. You cannot combine the two methods.

Construct a PUT request with a `multipart/mixed` message body where the metadata is in the first part and the document content is in the second part of the request body. The request URL is of the form:

```
http://host:port/version/documents?uri=doc_uri&category=content&category=
metadata_category
```

Where `category` can appear multiple times, with the values described in “Metadata Categories” on page 142.

When constructing the request:

1. Set the `uri` parameter to the URI of the destination document in the database.
2. Set `category=content` in the request URL to indicate content is included in the body.
3. Set additional `category` parameters to indicate the type(s) of metadata to add or update.
4. Specify `multipart/mixed` in the HTTP Content-type header for the request.
5. Set the part boundary string in the HTTP Content-type header to a string of your choosing.
6. Set the `Content-type` of the first part to either `application/xml` or `application/json` and place the XML or JSON metadata in the part body.
7. Set the `Content-type` of the second part to the MIME type of the content and place the content in the part body.

For details on metadata formats, see “Working with Metadata” on page 142.

> **Note:** Metadata must always be the first part of the multipart body.

The following example inserts an XML document with the URI `/xml/box.xml` into the database and adds it to the “shapes” and “squares” collection. The collection metadata is provided in XML format in the first part of the body, and the content is provided as XML in the second part.

```
$ cat ./the-body
--BOUNDARY
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>
<metadata xmlns="http://marklogic.com/rest-api">
  <collections>
    <collection>shapes</collection>
    <collection>squares</collection>
  </collections>
</metadata>
--BOUNDARY
Content-Type: text/xml

<?xml version="1.0" encoding="UTF-8"?>
<data>
  <mybox>
    This is my box. There are many like it, but this one is mine.
  </mybox>
</data>
--BOUNDARY--

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    --data-binary @./the-body \
    -H "Content-type: multipart/mixed; boundary=BOUNDARY" \
    'http://localhost:8000/LATEST/documents?uri=/xml/box.xml&category=
collections&category=content'
```

3.2.4 Automatically Generating Document URIs

MarkLogic Server can automatically generate databse URIs for documents inserted with the REST API. You can only use this feature to create new documents. To update an existing document, you must know the URI; for details, see “Loading Content” on page 56.

To use this feature, send a POST request to `/documents` that includes the `extension` request parameter and does not include the `uri` request parameter. That is, a POST request with a URL of the following form:

```
http://host:port/version/documents?extension=file-extension
```

Where extension specifies the document URI file prefix, such as `xml` or `txt`. Optionally, you can also include a database directory prefix with the `directory` request parameter.

Note: The directory prefix should end in a forward slash (`/`).

MarkLogic Server returns the auto-generated URI in the Location header of the response.

The following example inserts a document into the database with a URI of the form `/my/directory/auto-generated.xml`:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X POST -d@'./my-content' -i \
  -H "Content-type: application/xml" \
  'http://localhost:8000/LATEST/documents?extension=xml&directory=/my/
directory/'

HTTP/1.1 201 Document Created
Location: /my/directory/6041706572796142832.xml
...
```

You can use the same features with generated URIs as when loading documents with user-defined URIs. For example, you can insert both documents and metadata and apply content transformations. For details, see “Loading Content into the Database” on page 55 and the *MarkLogic REST API Reference*.

3.2.5 Loading Triples

You can use the `/graphs` service to load semantic triples into the database in several formats. The service implements the W3C Graph Store Protocol, described by the following specification:

http://www.w3.org/TR/2013/REC-sparql11-http-rdf-update-20130321/

Note: The collection lexicon must be enabled on your database when using the semantics REST services or use the GRAPH '?g' construct in a SPARQL query. For details, see Configuring the Database to Work with Triples in the *Semantics Developer’s Guide*.

To load triples, send a PUT or POST request to the /graphs service with one of the following forms of URL, depending upon whether you wish to address a specific named graph or the default graph:

```
http://host:port/version/graphs?graph=graph-uri

http://host:port/version/documents?default
```

For details, see Loading Triples Using the REST API and Supported RDF Triple Formats in *Semantics Developer’s Guide*.

You can also create, update, and delete triples and graphs with SPARQL Update by sending a POST request to the `/graphs/sparql` service. For example, the following SPARQL update request inserts a triple into the default graph:

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
INSERT DATA
{ <http://example/book0> dc:title "A default book"  }
```

If you place the above SPARQL Update request into a file named update.sparql, then the following request to the `/graphs/sparql` performs the requested update:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X POST -i
  --data-binary @./update.sparql \
  -H "Content-type:application/sparql-update" \
  -H "Accept:application/sparql-results+xml" \
  'http://localhost:8000/LATEST/graphs/sparql'
```

When you put your SPARQL Update request in the HTTP request body, the request Content-type must be `application/sparql-update`. You can also specify your update using URL encoded form data. For more details and examples, see SPARQL Update with the REST Client API in the *Semantics Developer's Guide*.

3.2.6 Controlling Access to Documents Created with the REST API

Documents you create with the REST Client API have a read permission for the `rest-reader` role and an update permission for the `rest-writer` role by default. To restrict access to particular users, create custom roles rather than assigning users to the default `rest-*` roles. For example, you can use a custom role to restrict users in one group from seeing documents created by another.

For details, see "Controlling Access to Documents and Other Artifacts" on page 22.

3.2.7 Transforming Content During Ingestion

You can transform content during ingestion by applying custom transform. A transform is an XQuery module or XSLT stylesheet you write and install using `/config/transforms/{name}`. For details, see "Working With Content Transformations" on page 323.

To apply a transform when creating or updating a document, add the `transform` parameter to your request. If the transform expects parameters, specify them using `trans:`*`paramName`* parameters. That is, your request should be of the form:

```
http://host:port/version/documents?...&transform=name&trans:arg=value
```

The following example applies a transform installed under the name "example" that expects a parameter named "reviewer":

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    -d@./the-body -H "Content-type: application/xml" \
    'http://localhost:8000/LATEST/documents?uri=/doc/theDoc.xml&transf
orm=example&trans:reviewer=me'
```

For a complete example, see "XQuery Example: Adding an Attribute During Ingestion" on page 338 or "XSLT Example: Adding an Attribute During Ingestion" on page 340.

3.3 Retrieving Documents from the Database

To a retrieve document from the database, make a GET request to the `/documents` service. You can retrieve just the contents, just the metadata, or both contents and metadata. This section covers the following topics:

- Retrieving the Contents of a Document
- Retrieving Metadata About a Document
- Retrieving Content and Metadata in a Single Request
- Transforming Content During Retrieval

To retrieve multiple documents in a single request, see “Reading and Writing Multiple Documents” on page 249.

3.3.1 Retrieving the Contents of a Document

To retrieve a document from the database, construct a GET request of the following form:

```
http://host:port/version/documents?uri=doc_uri
```

HTTP content type negotiation is not supported. If the HTTP Accept header is not set, MarkLogic Server uses the file extension on the document URI to determine the response content type, based on the server-wide MIME type mapping definitions. See “Mimetypes” in the Admin Interface.

Though content negotiation is not supported, you can use the transform feature to apply server-side transformations to the content before the response is constructed. For details, see “Working With Content Transformations” on page 323.

To retrieve multiple documents in a single request, see “Reading and Writing Multiple Documents” on page 249.

3.3.2 Retrieving Metadata About a Document

To retrieve metadata about a document without retrieving the contents, construct a GET request of the following form:

```
http://host:port/version/documents?uri=doc_uri&category=metadata_category
```

Where `category` can appear multiple times, with the values described in “Metadata Categories” on page 142.

When constructing the request:

1. Set the `category` parameter to the type of metadata to retrieve. Specify `category` multiple times to request more than one type of metadata.

2. Specify the metadata content type (XML or JSON) in the HTTP `Accept` header or the `format` parameter.

For details on metadata categories and formats, see "Working with Metadata" on page 142.

Use the `format` parameter or the `Accept` header to specify the format of the metadata. If both `format` and `Accept` are set, `format` takes precedence. If neither `format` nor `Accept` is specified, XML is assumed. For details, see "Controlling Input and Output Content Type" on page 31.

To retrieve metadata as XML, set `format` to "xml" or set the `Accept` header to `application/xml`. To retrieve metadata as JSON, set `format` to "json" or set the `Accept` header to `application/json`.

3.3.3 Retrieving Content and Metadata in a Single Request

To retrieve content and metadata for a document in a single request, construct GET request to a URL of the form:

```
http://host:port/version/documents?uri=doc_uri&category=content&catego
ry=metadata_category
```

Where `category` can appear multiple times, with the values described in "Metadata Categories" on page 142.

The request response is a `multipart/mixed` message, with the metadata in the first body part and content in the second body part. The Content-Type headers for the parts are determined as follows:

- The MIME type of the metadata part is determined by the `format` parameter, which you can set to either `xml` or `json`; the default is xml. For details on metadata format, see "Working with Metadata" on page 142.
- The MIME type of the content part is determined by the server-wide MIME type mapping for the document URI extension. See "Mimetypes" in the Admin Interface.

The following example command retrieves a document and its metadata in a single request:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    -H "Accept: multipart/mixed;boundary=BOUNDARY" \
    'http://localhost:8000/LATEST/documents?uri=/xml/box.xml&category=
metadata&category=content&format=xml'

--BOUNDARY
Content-Type: application/xml
Content-Length: 518

<?xml version="1.0" encoding="UTF-8"?>
<rapi:metadata uri="/xml/box.xml"
    xsi:schemaLocation="http://marklogic.com/rest-api/database
dbmeta.xsd"
```

```
    xmlns:rapi="http://marklogic.com/rest-api"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <rapi:collections>
    <rapi:collection>shapes</rapi:collection>
    <rapi:collection>squares</rapi:collection>
  </rapi:collections>
  <rapi:permissions/>
  <prop:properties xmlns:prop="http://marklogic.com/xdmp/property"/>
  <rapi:quality>0</rapi:quality>
  <rapi:metadata-values/>
</rapi:metadata>
--BOUNDARY
Content-Type: text/xml
Content-Length: 128

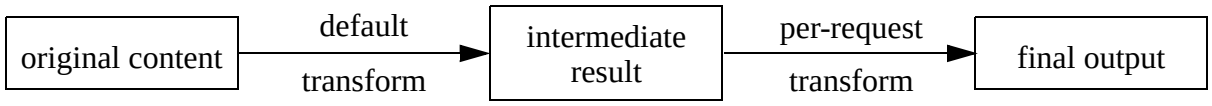
<?xml version="1.0" encoding="UTF-8"?>
<data><mybox>This is my box. There are many like it, but this one is
mine.</mybox></data>
--BOUNDARY--
```

HTTP content negotiation is not supported, but custom server-side content transformations can be applied using the `transform` parameter. For details, see “Retrieving the Contents of a Document” on page 65 and “Working With Content Transformations” on page 323.

3.3.4 Transforming Content During Retrieval

You can apply custom transforms to a document before returning it to the requestor. A transform is a JavaScript module, XQuery module, or XSLT stylesheet you write and install using `/config/transforms/{name}`. For details, see “Working With Content Transformations” on page 323.

You can configure a default transform that is automatically applied whenever a document is retrieved. You can also specify a per-request transform using the `transform` request parameter. If there is both a default transform and a per-request transform, the transforms are chained together, with the default transform running first. Thus, the output of the default transform is the input to the per-request transform:



To configure a default transformation, set the `document-transform-out` configuration parameter for the REST Client API instance. Instance-wide parameters are set using `/config/properties`. For details, see “Configuring Instance Properties” on page 47.

To specify a per-request transform, add the `transform` parameter to your request. If the transform expects parameters, specify them using `trans:`*`paramName`* parameters. That is, your request should be of the form:

```
http://host:port/version/documents?...&transform=name&trans:arg=value
```

The following example applies a transform installed under the name “example” that expects a parameter named “reviewer”:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
    'http://localhost:8000/LATEST/documents?uri=/doc/theDoc.xml&transf
orm=example&trans:reviewer=me'
```

3.4 Partially Updating Document Content or Metadata

This section focuses on modifying a portion of an existing XML or JSON document or its metadata. To learn about inserting or updating an entire document or its metadata, see “Loading Content into the Database” on page 55.

This section covers the following topics:

- Introduction to Content and Metadata Patching
- Basic Steps for Patching Documents
- XML Patch Reference
- Managing XML Namespaces in a Patch
- XML Examples of Partial Updates
- JSON Patch Reference
- JSON Examples of Partial Update
- Patching Metadata
- Constructing Replacement Data on the Server
- How Position Affects the Insertion Point
- Path Expressions Usable in Patch Operations
- Limitations of JSON Path Expressions
- Introduction to JSONPath

3.4.1 Introduction to Content and Metadata Patching

A *partial update* is an update you apply to a portion of a document or metadata. For example, inserting an XML element or attribute or changing the value associated with a JSON property. You can only apply partial content updates to XML and JSON documents. You can apply partial metadata updates to any document type.

A *patch* is a partial update descriptor, expressed in XML or JSON, that tells MarkLogic Server where to apply an update and what update to apply. A patch is a wrapper XML element or JSON object that encapsulates one or more update operations.

Use a partial update to do the following operations:

- Add, replace, or delete an XML element, XML attribute, or JSON sub-object of an existing document.
- Add, replace, or delete a subset of the metadata of an existing document. For example, modify a permission or insert a document property.
- Dynamically generate replacement content or metadata on MarkLogic Server using builtin or user-defined functions. For details, see "Constructing Replacement Data on the Server" on page 112.

To perform a partial update, send a POST or PATCH request to the `/documents` service with a patch in the request body. For example, the following patch descriptor inserts a new `<child/>` element as the last child of the element located through the XPath expression `/data`:

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:insert context="/data" position="last-child">
    <child>the last one</child>
  </rapi:insert>
</rapi:patch>
```

Assuming the above patch is saved to the file `patch.xml`, the following command uses a POST request to apply the patch to the document with the URI `/doc/example.xml`.

```
$ curl --anyauth --user user:password -X POST -d @./patch.xml -i \
    -H "Content-type: application/xml" \
    -H "X-HTTP-Method-Override: PATCH" \
    http://localhost:8000/LATEST/documents?uri=/doc/example.xml
```

The following example is an equivalent request using PATCH instead of POST:

```
$ curl --anyauth --user user:password -X PATCH -d @./patch.xml -i \
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/documents?uri=/doc/example.xml
```

For details, see "Basic Steps for Patching Documents" on page 70.

You can apply multiple updates in a single request. You can patch both content and metadata in a single request.

Content negotiation is not supported: A content patch must match the content type of the target document. That is, you cannot submit a JSON patch for an XML document or an XML patch for a JSON document.

If a patch contains multiple operations, they are applied independently to the target document. That is, within the same patch, one operation does not affect the `context` or `select` results or the content changes of another. Each operation in a patch is applied independently to every matched node. If any operation in a patch fails with an error, the entire patch fails.

Content transformations are not directly supported in a partial update. However, you can implement a custom replacement content generation function to achieve the same effect. For details, see "Constructing Replacement Data on the Server" on page 112.

3.4.2 Basic Steps for Patching Documents

To perform a partial update of an XML or JSON document, or of document metadata, send a POST or PATCH request to the `/documents` service with a URL of the following form:

```
http://host:port/version/documents?uri=doc_uri
```

The body of the request must contain a content patch, as described in "XML Patch Reference" on page 70 and "JSON Patch Reference" on page 88. You can patch metadata and content in the same request by using the category request parameter; for details, see "Patching Metadata" on page 109.

When constructing your request, follow these guidelines:

1. Set the `uri` parameter to the URI of the target document in the database.

2. Set the `category` request parameter to `content`, `metadata`, and/or one of the metadata subcategories. The default is `content`.

3. Set the `Content-type` to either `application/xml` or `application/json`. When patching content, you must use the content type that matches the target document; only XML and JSON are supported.

4. Place the XML or JSON patch in the request body.

5. If you POST, set the request header `X-HTTP-Method-Override` to `PATCH` to tell MarkLogic Server this is a partial update request.

New content that is added or replaced can be specified directly in the patch, or generated on MarkLogic Server by a replacement content generator function. For details, see "Constructing Replacement Data on the Server" on page 112.

3.4.3 XML Patch Reference

This section summarizes the structure of the XML patch structure used to describe partial updates to XML documents and document metatdata. Each section includes a syntax summary, a description of the components, and an example.

The following elements, all in the namespace http://marklogic.com/rest-api, are covered. Start wit the `<patch/>` wrapper.

- patch
- insert
- replace
- replace-insert
- delete
- replace-library

3.4.3.1 patch

The top level wrapper element around a partial update descriptor. A `<patch/>` has the following structure:

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:insert />
  <rapi:replace />
  <rapi:replace-insert />
  <rapi:delete />
  <rapi:replace-library />
</rapi:patch>
```

All elements are optional. The operations can occur multiple times. The `<replace-library/>` can occur at most once and is only required if you use user-defined functions to generate replacement content server-side; for details, see “Constructing Replacement Data on the Server” on page 112.

Note: A patch can only replace a single node.

For example, the following patch includes an insertion, deletion, and replacement:

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:insert context="/inventory/history" position="last-child">
    <modified>2012-11-5</modified>
  </rapi:insert>
  <rapi:delete select="saleExpirationDate"/>
  <rapi:replace select="price" apply="ml.multiply">1.1</rapi:replace>
</rapi:patch>
```

Where a patch operation includes new XML content, such as an element insertion, the new content must use namespaces appropriate to the target document. Namespaces declared in the root `<patch/>` node are in scope for this content. For details, see “Managing XML Namespaces in a Patch” on page 80.

For more details, see the individual operations.

3.4.3.2 insert

Insert a new element, attribute, text node, comment, or processing instruction in an XML document or in document metadata. An `<insert/>` element has the following structure:

```
<insert context=xpath-expr position=pos-selector
        cardinality=occurrence>
  content-to-insert
</insert>
```

For example, the following patch adds a `<modified/>` element as the last child of every `<history/>` node that matches the XPath expression `/inventory/history`:

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:insert context="/inventory/history" position="last-child">
    <modified>2012-11-5</modified>
  </rapi:insert>
</rapi:patch>
```

For additional examples, see “XML Examples of Partial Updates” on page 81.

The following table summarizes the parts of an `<insert/>` element.

Component	Req’d	Description
`@context=`*`xpath-expr`*	Y	An XPath expression that selects an existing node(s) or attribute on which to operate. Namespaces declared on the root `<patch/>` node are in scope for this path. When inserting an attribute, `@context` can identify either the containing element(s) or another attribute of the containing element(s). In all other cases, `@context` should identify one or more nodes. If no matches are found for the `@context` expression, the operation is silently ignored. The path expression is restricted to the subset of XPath for which `cts:valid-document-patch-path` (XQuery) or `cts.validDocumentPatchPath` (JavaScript) returns true. For details, see “Path Expressions Usable in Patch Operations” on page 126.
`@position=`*`pos-selector`*	Y	Where to insert the content, relative to the node(s) selected by `@context`. The attribute value must be one of `before`, `after`, or `last-child`.

Component	Req'd	Description
`@cardinality=`*`occurrence`*	N	The required occurrence of matches to `@position`. If the number of matches does not meet the expectation, the operation fails and an error is returned. Allowed values: • Zero or one matches required: `?` (question mark) • Exactly one match required: `.` (period) • Zero or more matches required: `*` (asterisk) • One or more matches required: `+` (plus) Default: * (The occurrence requirement is always met).
`content-to-insert`	Y	When inserting elements, text, comments, and processing directives, specify the new nodes as they should appear in the target document. When inserting attributes, specify the attribute(s) on a `<rapi:attribute-list/>` element. The following example specifies 2 attributes to insert: `<rapi:attribute-list a1="v1" a2="v2"/>`

3.4.3.3 replace

Replace an existing element or attribute. If no matching element or attribute exists, the operation is silently ignored. A `<replace/>` element has the following structure:

```
<replace select=xpath-expr cardinality=occurrence apply=func-name>
  replacement-content
</replace>
```

For example, the following patch replaces the first `<child/>` element of `<parent/>`:

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:replace select="/parent/child[1]">
    <child>REPLACED</child>
  </rapi:replace>
</rapi:patch>
```

For additional examples, see "XML Examples of Partial Updates" on page 81.

You can also supply a content generation builtin or user-defined function using `@apply`, using the format below. For example, you can use the builtin `ml.multiply` function to multiply the current value of an element that contains numeric data, without needing to know the current value.

```
<rapi:replace select="my-elem" apply=ml.multiply>5</rapi:replace>
```

For details, see “Constructing Replacement Data on the Server” on page 112.

The following table summarizes the parts of a `<replace/>` element.

Component	Req’d	Description
`@select=`*`xpath-expr`*	Y	An XPath expression that selects an existing node(s) or attribute to replace. Namespaces declared on the root `<patch/>` node are in scope for this path. The path expression is restricted to the subset of XPath for which `cts:valid-document-patch-path` (XQuery) or `cts.validDocumentPatchPath` (JavaScript) returns true. For details, see “Path Expressions Usable in Patch Operations” on page 126. The selected node cannot be the target of any other operation in the patch. The ancestor of the selected node may not be modified by a `delete`, `replace`, or `replace-insert` operation in the same patch.
`@cardinality=`*`occurrence`*	N	The required occurrence of matches to `@select`. If the number of matches does not meet the expectation, the operation fails and an error is returned. Allowed values: • Zero or one matches required: `?` (question mark) • Exactly one match required: `.` (period) • Zero or more matches required: `*` (asterisk) • One or more matches required: `+` (plus) Default: * (The occurrence requirement is always met).

Component	Req'd	Description

replacement-content	N	The content with which to replace the selected node or attribute. If present, this must be a single node. If there is no *replacement-content*, you must specify a content generation function using `@apply`.
		
		To replace an element, specify either a replacement element or text. When you use text, the text replaces just the content of the target element.
		
		To replace an element with a text node, wrap the text in a `<text/>` element in the namespace `http://marklogic.com/rest-api`. For example:
		
		`<rapi:text>replacement text</rapi:text>`
		
		To replace an attribute, use either a `<text/>` or `<attribute-list/>` wrapper in the namespace `http://marklogic.com/rest-api`. For example:
		
		`<rapi:attribute-list my-attr="`*new-value*`" />`
		`<rapi:text>`*new-value*`</rapi:text>`

`@apply=`*func-name*	N	The local name of a replacement content generation function. If you do not specify a function, the operation must include *replacement-content*. If you name a user-defined function, the `<patch/>` must also contain a `<replace-library/>` element. For details, see "Constructing Replacement Data on the Server" on page 112.

3.4.3.4 replace-insert

Replace an element or attribute; if there are no existing matching elements or attributes, perform an insertion operation instead. A `<replace-insert/>` element has the following structure:

```
<replace-insert select=xpath-expr context=xpath-expr position=pos
    cardinality=occurrence apply=func-name>
  replacement-content
</replace-insert>
```

For example, the following patch replaces the first `<grandchild/>` element of a `<child/>` of `<parent/>` elements. If no `<child/>` element has a `<grandchild/>` element, one is inserted in every `<child/>` of a `<parent/>` element.

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:replace-insert context="/parent/child" select="grandchild[1]"
      position="last-child">
    <grandchild>CHANGED</grandchild>
  </rapi:replace-insert>
</rapi:patch>
```

For additional examples, see “XML Examples of Partial Updates” on page 81.

You can also use `@apply` to specify a content generation builtin or user-defined function for generating dynamic content. For details, see “Constructing Replacement Data on the Server” on page 112.

The following table summarizes the parts of a `<replace-insert/>` element.

Component	Req’d	Description
`@select=`*`xpath-expr`*	Y	An XPath expression that selects an existing node(s) or attribute to replace. Namespaces declared on the root `<patch/>` node are in scope for this path. The path may be absolute or relative to the `@context` path. The path expression is restricted to the subset of XPath for which `cts:valid-document-patch-path` (XQuery) or `cts.validDocumentPatchPath` (JavaScript) returns true. For details, see “Path Expressions Usable in Patch Operations” on page 126. If no matches are found for the `@select` expression, `@context` and `@position` are used to attempt an insert. If no match is then found for `@context`, the behavior depends on the value of `@cardinality`. The selected node cannot be the target of any other operation in the patch. The ancestor of the selected node may not be modified by a delete, replace, or replace-insert operation in the same patch.

Component	Req'd	Description

replacement-content	N	The content with which to replace the selected node or attribute. If there is no *replacement-content*, you must specify a content generation function using `@apply`.
		
		To replace an element, specify either a replacement element or text. When you use text, the text replaces just the content of the target element.
		
		To replace an element with a text node, wrap the text in a `<text/>` element in the namespace `http://marklogic.com/rest-api`. For example:
		
		`<rapi:text>replacement text</rapi:text>`
		
		To replace an attribute, use either a `<text/>` or `<attribute-list/>` wrapper in the namespace `http://marklogic.com/rest-api`. For example:
		
		`<rapi:attribute-list my-attr="`*new-value*`" />`
		`<rapi:text>`*new-value*`</rapi:text>`

`@context=`*xpath-expr*	Y	An XPath expression that selects an existing node(s) or attribute on which to operate if the `@select` target does not exist. `@context` is also used to resolve `@select` if the select expression is a relative path. Namespaces declared on the root `<patch/>` node are in scope for this path. The path expression is restricted to the subset of XPath for which `cts:valid-document-patch-path` (XQuery) or `cts.validDocumentPatchPath` (JavaScript) returns true. For details, see "Path Expressions Usable in Patch Operations" on page 126. When inserting an attribute, `@context` can identify either the containing element(s) or another attribute of the containing element(s). In all other cases, `@context` should identify one or more nodes. If no matches are found for either the `@context` or `@select` expressions, the operation is silently ignored. The ancestor of the selected node may not be modified by a delete, replace, or replace-insert operation in the same patch.

Component	Req'd	Description
`@position=`*`pos-selector`*	Y	If `@select` does not match anything, where to insert the content, relative to the node(s) selected by `@context`. The attribute value must be one of `before`, `after`, or `last-child`. Ignored for attributes.
`@cardinality=`*`occurrence`*	N	The required occurrence of matches to `@position`. If the number of matches does not meet the expectation, the operation fails and an error is returned. Allowed values: • Zero or one matches required: `?` (question mark) • Exactly one match required: `.` (period) • Zero or more matches required: `*` (asterisk) • One or more matches required: `+` (plus) Default: * (The occurrence requirement is always met).
`@apply=`*`func-name`*	N	The local name of a replacement content generation function. If you do not specify a function, the operation must include *`replacement-content`*. If you name a user-defined function, the `<patch/>` must also contain a `<replace-library/>` element. For details, see "Constructing Replacement Data on the Server" on page 112.

3.4.3.5 delete

Remove an element or attribute from an existing node. A `<delete/>` element has the following structure:

```
<delete select=xpath-expr cardinality=occurrence />
```

For example, the following patch deletes the first `<child/>` element of a `<parent/>` and removes the `my-attr` attribute from the `<parent/>` element.

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:delete select="/parent/child[1]" />
  <rapi:delete select="/parent/@my-attr" />
</rapi:patch>
```

For additional examples, see "XML Examples of Partial Updates" on page 81.

The following table summarizes the parts of a `<delete/>` element.

Component	Req'd	Description
`@select=`*`xpath-expr`*	Y	An XPath expression that selects the node(s) or attribute(s) to delete. Namespaces declared on the root `<patch/>` node are in scope for this path. The path expression is restricted to the subset of XPath for which `cts:valid-document-patch-path` (XQuery) or `cts.validDocumentPatchPath` (JavaScript) returns true. For details, see "Path Expressions Usable in Patch Operations" on page 126. The selected node cannot be the target of any other operation in the patch. The ancestor of the selected node may not be modified by a delete, replace, or replace-insert operation in the same patch.
`@cardinality=`*`occurrence`*	N	The required occurrence of matches to `@select`. If the number of matches does not meet the expectation, the operation fails and an error is returned. Allowed values: • Zero or one matches required: `?` (question mark) • Exactly one match required: `.` (period) • Zero or more matches required: `*` (asterisk) • One or more matches required: `+` (plus) Default: * (The occurrence requirement is always met).

3.4.3.6 replace-library

Specify an XQuery library module that contains user-defined replacement content generation functions. These functions can be used in `@apply` on `<replace/>` and `<replace-insert/>` operations. For details, see "Constructing Replacement Data on the Server" on page 112.

For example, the following patch uses the function `my-ns:my-func` in the library module in the with the URI `/my.domain/my-module.xqy`.

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:replace-library
    at="/my.domain/my-module.xqy" ns="my-ns" />
  <rapi:replace select="/parent/child[1]" apply="my-func" />
</rapi:patch>
```

For additional examples, see “Constructing Replacement Data on the Server” on page 112.

The following table summarizes the parts of a `<replace-library/>` element.

Component	Req’d	Description
`@at=`*`lib-module-path`*	N	The path to the XQuery library module containing user-defined replacement content generation functions. The module must be installed in the modules database associated with the REST API instance.
`@ns=`*`func-namespace`*	N	The module namespace alias defined by the `@at` module.

3.4.4 Managing XML Namespaces in a Patch

The patch descriptor must be in the namespace `http://marklogic.com/rest-api`. To distinguish the `<patch/>` elements from elements in your document, you should include an alias for the patch descriptor namespace. For example:

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
    ...
</rapi:patch>
```

Declare additional namespace aliases for the content described by the patch. Namespaces declared on the `<patch/>` root element are in scope in `@context` and `@select` path expressions and in the content portion of `<insert/>`, `<replace/>` and `<replace-insert/>` elements.

For example, if you patch a document with the following contents:

```
<parent xmlns="http://my/namespace">
  <child>first</child>
</parent>
```

Then your `<patch/>` should either define `http://my/namespace` as the default namespace or define a namespace alias for it. For example, the following patch inserts a `<child/>` element when you define a default namespace:

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api"
    xmlns="http://my/namespace">
  <rapi:insert context="/parent" position="last-child">
    <child>last</child>
  </rapi:insert>
</rapi:patch>
```

The following is the same patch, using an explicit namespace alias of `my-ns`:

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api"
    xmlns:my-ns="http://my/namespace">
  <rapi:insert context="/my-ns:parent" position="last-child">
    <my-ns:child>last</my-ns:child>
  </rapi:insert>
</rapi:patch>
```

Failure to use namespace declarations properly in a patch can lead to surprising results. For example, if you did not include a namespace declaration for `http://my/namespace`, the patch would not work because the context XPath expression would not match any elements in the target document. Similarly, if you include a namespace alias but do not use it in the definition of the new child, the new child element would be inserted in no namespace: `<child xmlns="">...</child>`.

3.4.5 XML Examples of Partial Updates

This section includes the following examples of applying a partial update to an XML document or metadata:

- Example cURL Commands
- Example: Inserting an Element
- Example: Inserting an Attribute
- Example: Inserting a Text Node
- Example: Inserting a Comment or Processing Instruction
- Example: Multiple Inserts in One Patch
- Example: Replacing an Element or Element Contents
- Example: Replacing an Attribute Value
- Example: Replacing or Inserting an Element in One Operation
- Example: Replacing or Inserting an Attribute in One Operation
- Example: Deleting an Element or Attribute

Note: Many of the examples include extra whitespace to improve readability. You should not do likewise unless you want extra whitespace in your target document. Whitespace in the body of an `<insert/>`, `<replace/>`, or `<replace-insert/>` operation is usually significant.

3.4.5.1 Example cURL Commands

The examples in this section can be applied using commands similar to the following, assuming you copy the sample input document to a file called `example.xml` and the patch to a file called `patch.xml`.

```
# Windows users, see Modifying the Example Commands for Windows
# Create the example document
$ curl --anyauth --user user:password -X PUT -d @./example.xml \
    -i -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.xml

# Apply the patch
$ curl --anyauth --user user:password -X POST -d @./patch.xml \
    -i -H "Content-type: application/xml" \
    -H "X-HTTP-Method-Override: PATCH" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.xml

# Retrieve the results
curl --anyauth --user user:password -X GET \
    -i -H "Accept: application/xml" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.xml
```

3.4.5.2 Example: Inserting an Element

The following patch inserts an `<INSERTED/>` element before each `<child/>` element matched by the context expression.

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:insert context="/parent/child" position="before">
    <INSERTED/>
  </rapi:insert>
</rapi:patch>
```

The table below shows how applying the patch changes the target document.

Before Update	After Update

<parent>
  <child>one</child>
  <child>two</child>
  <child>three</child>
</parent>
	<parent>
	  <INSERTED/>
	  <child>one</child>
	  <INSERTED/>
	  <child>two</child>
	  <INSERTED/>
	  <child>three</child>
	</parent>

3.4.5.3 Example: Inserting an Attribute

The following patch adds two attributes, `my-attr1` and `my-attr2`, on the second `<child/>` element.

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:insert context="/parent/child[2]">
    <rapi:attribute-list my-attr1="val1" my-attr2="val2" />
  </rapi:insert>
</rapi:patch>
```

The table below shows how applying the patch changes the target document.

Before Update	After Update
<parent> <child>one</child> <child>two</child> <child>three</child> </parent>	<parent> <child>one</child> <child **my-attr1="val1" my-attr2="val2"**> two </child> <child>three</child> </parent>

3.4.5.4 Example: Inserting a Text Node

The following patch inserts a text node before the second `<child/>` element matched by the context expression.

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:insert context="/parent/child[2]"
     position="before">INSERTED</rapi:insert>
</rapi:patch>
```

The table below shows how applying the patch changes the target document.

Before Update	After Update
<parent> <child>one</child> <child>two</child> <child>three</child> </parent>	<parent> <child>one</child> **INSERTED** <child>two</child> <child>three</child> </parent>

3.4.5.5 Example: Inserting a Comment or Processing Instruction

The following patch inserts a comment as the first child of every `<parent/>` element matched by the context expression.

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:insert context="/parent/*[1]" position="before">
    <!-- INSERTED -->
  </rapi:insert>
</rapi:patch>
```

The table below shows how applying the patch changes the target document.

Before Update	After Update
<parent> <child>one</child> <child>two</child> <child>three</child> </parent>	<parent> **<!-- INSERTED -->** <child>one</child> <child>two</child> <child>three</child> </parent>

3.4.5.6 Example: Multiple Inserts in One Patch

The following patch performs several insert operations on the target document:

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:insert context="/parent/child" position="before">
    <INSERTED/>
  </rapi:insert>
  <rapi:insert context="/parent/child[2]"
      position="before">INSERTED</rapi:insert>
  <rapi:insert context="/parent/child[2]">
    <rapi:attribute-list my-attr="val" />
  </rapi:insert>
  <rapi:insert context="/parent/*[1]" position="before">
    <!-- INSERTED -->
  </rapi:insert>
</rapi:patch>
```

The table below shows how applying the patch changes the target document. Notice that the insert operations act independently on the target document, rather than interacting with each other. For example, the comment insertion operation, which uses the context expression `/parent/*[1]`, inserts the comment before the first original `<child/>` element, not before the `<INSERTED/>` element added by one of the other operations in the patch.

Before Update	After Update

<parent>
  <child>one</child>
  <child>two</child>
  <child>three</child>
</parent>
	<parent>
	  <INSERTED/>
	  <!-- INSERTED -->
	  <child>one</child>
	  <INSERTED/>
	  INSERTED
	  <child **my-attr="val"**>two</child>
	  <INSERTED/>
	  <child>three</child>
	</parent>

3.4.5.7 Example: Replacing an Element or Element Contents

The following patch replaces the first `<child/>` node with a new element, replaces the contents of the second `<child/>`, and replaces the third `<child/>` with a text node.

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <!-- replace an entire element -->
  <rapi:replace select="/parent/child[1]">
    <REPLACED />
  </rapi:replace>

  <!-- replace the element contents -->
  <rapi:replace select="/parent/child[2]">
    REPLACED
  </rapi:replace>

  <!-- replace an element with a text node -->
  <rapi:replace select="/parent/child[3]">
    <rapi:text>REPLACED</rapi:text>
  </rapi:replace>
</rapi:patch>
```

The table below shows how applying the patch changes the target document.

Before Update	After Update

<parent>
  <child>one</child>
  <child>two</child>
  <child>three</child>
</parent>
	<parent>
	  <REPLACED />
	  <child>**REPLACED**</child>
	  REPLACED
	</parent>

3.4.5.8 Example: Replacing an Attribute Value

The following patch demonstrates two methods of replacing the value of an attribute. The first `<replace/>` uses a `<text/>` wrapper and the second uses an `<attribute-list>` container element.

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:replace select="/parent/@attr1">
    <rapi:text>REPLACE-1</rapi:text>
  </rapi:replace>
  <rapi:replace select="/parent/child/@attr2">
    <rapi:attribute-list attr2="REPLACE-2"/>
  </rapi:replace>
</rapi:patch>
```

The table below shows how applying the patch changes the target document.

Before Update	After Update

`<parent attr1="val1">`
`  <child attr2="val2">one</child>`
`  <child attr2="val2">two</child>`
`</parent>`
	`<parent attr1="`**`REPLACE-1`**`">`
	`  <child attr2="`**`REPLACE-2`**`">one</child>`
	`  <child attr2="`**`REPLACE-2`**`">two</child>`
	`</parent>`

3.4.5.9 Example: Replacing or Inserting an Element in One Operation

The following patch replaces the first `<grandchild/>` element of a `<child/>` of `<parent/>` elements. If no `<child/>` element has a `<grandchild/>` element, one is inserted in every `<child/>` of a `<parent/>` element.

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:replace-insert context="/parent/child" select="grandchild[1]"
    position="last-child">
    <grandchild>CHANGED</grandchild>
  </rapi:replace-insert>
</rapi:patch>
```

The following table illustrates applying this patch to two documents, one for which there is a match to the replacement target, and one for which there is no match. In the first case, the node matching the `@select` expression (relative to `@context`) is replaced with the patch contents. In the second case, there are no matching nodes, so the content in the patch is inserted into all nodes that match the `@context` expression.

Before Update	After Update

Document contains matching elements.
 	
`<parent>`
`  <child/>`
`  <child>`
`  <grandchild/>`
`  <grandchild>`
`  unchanged`
`  </grandchild>`
`  </child>`
`</parent>`
	Matching elements replaced.
	
	`<parent>`
	`  <child/>`
	`  <child>`
	`  **<grandchild>CHANGED</grandchild>**`
	`  <grandchild>`
	`  unchanged`
	`  </grandchild>`
	`  </child>`
	`</parent>`

Document contains no matching elements.
 	
`<parent>`
`  <child/>`
`  <child/>`
`</parent>`
	New elements inserted using `@context`.
	
	`<parent>`
	`  <child>`
	`  **<grandchild>CHANGED</grandchild>**`
	`  </child>`
	`  <child>`
	`  **<grandchild>CHANGED</grandchild>**`
	`  </child>`
	`</parent>`

The following patch is equivalent, except that the `<grandchild/>` elements are replaced with a text node:

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:replace-insert context="/parent/child" select="grandchild[1]"
      position="last-child">
    <rapi:text>CHANGED</rapi:text>
  </rapi:replace-insert>
</rapi:patch>
```

3.4.5.10 Example: Replacing or Inserting an Attribute in One Operation

The following patch replaces `/parent/child/@my-attr` and inserts `/parent/child/@new-attr`.

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:replace-insert select="@my-attr" context="/parent/child">
    REPLACED
```

```
    </rapi:replace-insert>
    <rapi:replace-insert select="@new-attr" context="/parent/child">
      <rapi:attribute-list new-attr="INSERTED" />
    </rapi:replace-insert>
  </rapi:patch>
```

The following table illustrates the results of applying this patch:

Before Update	After Update
<parent> <child /> <child my-attr="val2" /> <child /> </parent>	<parent> <child **new-attr="INSERTED"**/> <child **my-attr="REPLACED"** **new-attr="INSERTED"**/> <child **new-attr="INSERTED"**/> </parent>

3.4.5.11 Example: Deleting an Element or Attribute

The following patch deletes all `<grandchild/>` elements and deletes an attribute from the `<parent/>` element.

```
  <rapi:patch xmlns:rapi="http://marklogic.com/rest-api">
    <rapi:delete select="/parent/child/grandchild"/>
    <rapi:delete select="/parent/@my-attr"/>
  </rapi:patch>
```

The table below shows how applying the patch changes the target document.

Before Update	After Update
<parent my-attr="val"> <child/> <child> <grandchild/> <grandchild/> </child> </parent>	<parent> <child/> <child></child> </parent>

3.4.6 JSON Patch Reference

This section summarizes the structure of the JSON patch structured used to describe partial updates to JSON documents. Each section includes a syntax summary, a description of component JSON properties, and an example.

This topic covers the following properties of a patch:

- pathlang
- patch
- insert
- replace
- replace-insert
- delete
- replace-library

The `insert`, `replace`, `replace-insert`, and `delete` patch operations all require constructing one or more path expressions that identify the content to be changed. To accurately address JSON with XPath, you need to understand the MarkLogic JSON document model. For details, see Traversing JSON Documents Using XPath in the *Application Developer's Guide*.

3.4.6.1 pathlang

Specify the path language (JSONPath or XPath) with which the patch specifies the document components targeted by an patch operation, such as the value of the `select` and `context` properties of a `replace-insert` operation.

The default value is `"xpath"`, so you only need to include this property in your patch when you want to use JSONPath.

> **Note:** The use of JSONPath is deprecated.

The following example sets the path language to JSONPath:

```
{ "pathlang": "jsonpath",
  "patch": [
    { "insert": {
        "context": "$.parent.child",
        "position": "last-child",
        "content": { "new-key": "new-value" }
  }}
]}
```

To learn more about addressing JSON with XPath, see "Limitations of JSON Path Expressions" on page 126 and Traversing JSON Documents Using XPath in the *Application Developer's Guide*.

3.4.6.2 patch

A patch is the top level wrapper object for applying partial updates to JSON documents. A JSON patch has the following structure. A patch object can contain multiple operations.

```
{"patch": [
    insert,
    replace,
    replace-insert,
    delete,
    replace-library
] }
```

All subobjects are optional. The operations (`insert`, `replace`, `replace-insert`) can occur multiple times. You can only have one `replace-library`, and it is only required if you use user-defined functions to generate replacement content.

For example, the following patch describes an insert and a replace operation. For more examples, see “JSON Examples of Partial Update” on page 98.

```
{"patch": [
  { "insert": {
      "context": "/parent/child",
      "position": "last-child",
      "content": { "new-key": "new-value" }
  }},
  { "replace": {
      "select": "/parent/child[0]",
      "content": "new-value"
    }}
]}
```

3.4.6.3 insert

Insert new JSON properties or array elements. An `insert` operation has the following structure:

```
{
  "insert": {
    "context": path-expr,
    "position": pos-selector,
    "content": new-content,
    "cardinality": occurrence
  }
}
```

For example, if `/parent/child` is the path to an array valued JSON property, then the following patch inserts the value "INSERTED" after each element of the array:

```
{"insert": {
    "context": "/parent/child",
    "position": "after",
    "content": "INSERTED"
}}
```

The following table summarizes the components of an `insert` operation.

JSON Property	Req'd	Description
`"context": path-expr`	Y	An XPath or JSONPath expression that selects an existing JSON property or array element on which to operate. The expression can select multiple items. If no matches are found for the `context` expression, the operation is silently ignored. The path expression is restricted to the subset of XPath (or its JSON Path equivalent) that can be used to define an index. For details, see "Path Expressions Usable in Patch Operations" on page 126.
`"position" : pos-selector`	Y	Where to insert the content, relative to the JSON property or value selected by `context`. The *pos-selector* must be one of `"before"`, `"after"`, or `"last-child"`. For details, see "Specifying Position in JSON" on page 124.
`"content": new-content`	Y	The new content to be inserted.
`cardinality: occurrence`	N	The required occurrence of matches to `position`. If the number of matches does not meet the expectation, the operation fails and an error is returned. Allowed values: • Zero or one matches required: `?` (question mark) • Exactly one match required: `.` (period) • Zero or more matches required: `*` (asterisk) • One or more matches required: `+` (plus) Default: * (The occurrence requirement is always met).

3.4.6.4 replace

Replace an existing JSON property or array element. If no matching JSON property or array element exists, the operation is silently ignored. A replace operation has the following structure:

```
{ "replace": {
    "select" : path-expr,
    "content" : new-value,
    "cardinality": occurrence,
    "apply" : func-name
  }
}
```

For example, the following patch replaces the value of the first array element in the JSON property named "child" that is a subobject of the JSON property named "parent":

```
{"patch": [
  {"replace": {
      "select": "/parent/child[1]",
      "content": "REPLACED"
  }}
]}
```

For additional examples, see “JSON Examples of Partial Update” on page 98.

You can use `apply` to specify a content generation builtin or user-defined function for generating dynamic content. For details, see “Constructing Replacement Data on the Server” on page 112.

The following table summarizes the parts of a `replace` operation.

Component	Req’d	Description
`"select":` *path-expr*	Y	An XPath or JSONPath expression that selects the JSON property or array element to replace. If no matches are found for the `select` expression, the operation is silently ignored. The path expression is restricted to the subset of XPath that can be used to define an index (or its JSONPath equivalent). For details, see “Path Expressions Usable in Patch Operations” on page 126. The selected item(s) cannot be the target of any other operation in the patch. The ancestor of the selected item may not be modified by a `delete`, `replace`, or `replace-insert` operation in the same patch.

Component	Req'd	Description
`"content" : new-value`	N	The replacement value. If there is no *new-value*, you must specify a content generation function using `apply`. If `select` targets a property and *new-value* is an object, the operation replaces the entire target property. Otherwise, the operation replaces just the value.
`cardinality: occurrence`	N	The required occurrence of matches to `select`. If the number of matches does not meet the expectation, the operation fails and an error is returned. Allowed values: • Zero or one matches required: `?` (question mark) • Exactly one match required: `.` (period) • Zero or more matches required: `*` (asterisk) • One or more matches required: `+` (plus) Default: * (The occurrence requirement is always met).
`"apply": func-name`	N	The local name of a replacement content generation function. If you do not specify a function, the operation must include a `content` property. If you name a user-defined function, the `patch` must include a `replace-library` property that describes the XQuery library module containing the function implementation. For details, see "Constructing Replacement Data on the Server" on page 112.

3.4.6.5 replace-insert

Replace a property or array item; if there are no existing matching properties, perform an insertion operation instead. A `replace-insert` operation has the following structure:

```
{ "replace-insert": {
    "select" : path-expr,
    "context" : path-expr,
    "position": pos-selector,
    "cardinality": occurrence,
    "content" : replacement-content,
    "apply" : func-name
  }
}
```

Optionally, you can supply a builtin or user-defined content generation function name using `apply`. If the function does not expect parameters, you can omit content when using `apply`. For details, see “Constructing Replacement Data on the Server” on page 112.

For example, the following patch replaces the value of `"target"` with `"new-value"` in the array value of `/parent/child`, if it exists. If no such value is found, then `"new-value"` is inserted as the last element in the array addressable as `/parent/array-node('child')`.

```
{ "patch": [
    { "replace-insert" : {
        "select": "/parent/child[. = 'target']",
        "context": "/parent/array-node('child')",
        "position": "last-child",
        "content": "new-value"
    }}
]}
```

For additional examples, see “JSON Examples of Partial Update” on page 98.

The following table summarizes the parts of a `replace-insert` operation.

Component	Req’d	Description
`"select"` : *path-expr*	Y	An XPath or JSONPath expression that selects the property or array element to replace. If no matches are found for the `select` expression, `context` and `position` are used to attempt an insert. If no match is found for `context`, the operation does nothing. The path expression is restricted to the subset of XPath that can be used to define an index, or its JSONPath equivalent. For details, see “Path Expressions Usable in Patch Operations” on page 126. The selected item(s) cannot be the target of any other operation in the patch. The ancestor of the selected item may not be modified by a `delete`, `replace`, or `replace-insert` operation in the same patch.

Component	Req'd	Description
replacement-content	N	The content with which to replace the selected value. If there is no *`replacement-content`*, you must specify a content generation function using `apply`. If `select` targets a property and *`replacement-content`* is an object, the operation replaces the entire target property. Otherwise, the operation replaces just the value.
`"context" :` *path-expr*	Y	An XPath or JSONPath expression that selects an existing property or array element on which to operate. The expression can select multiple items. If no matches are found for the either the `select` or `context` expression, the operation is silently ignored. The path expression is restricted to the subset of XPath that can be used to define an index, or its JSONPath equivalent. For details, see "Path Expressions Usable in Patch Operations" on page 126. The ancestor of the selected node may not be modified by a `delete`, `replace`, or `replace-insert` operation in the same patch.
`"position" :` *pos-selector*	N	If `select` does not match anything, where to insert the content, relative to the property or value selected by `context`. The *pos-selector* must be one of `"before"`, `"after"`, or `"last-child"`. For details, see "Specifying Position in JSON" on page 124. Default: `last-child`.

Component	Req'd	Description
`cardinality: occurrence`	N	The required occurrence of matches to `position`. If the number of matches does not meet the expectation, the operation fails and an error is returned. Allowed values: • Zero or one matches required: `?` (question mark) • Exactly one match required: `.` (period) • Zero or more matches required: `*` (asterisk) • One or more matches required: `+` (plus) Default: * (The occurrence requirement is always met).
`"apply" :` *func-name*	N	The local name of a replacement content generation function. If you do not specify a function, the operation must include a `content` property. If you name a user-defined function, the `patch` must include a `replace-library` property that describes the XQuery library module containing the function implementation. For details, see "Constructing Replacement Data on the Server" on page 112.

3.4.6.6 delete

Remove a property or array element. A `delete` operation has the following structure:

```
{ "delete": {
    "select" : path-expr,
    "cardinality": occurrence
} }
```

For example, the following patch deletes any JSON property named `child` that is contained in the property named `parent`.

```
{"patch": [
  { "delete" : { "select": "/parent//child" } }
]}
```

For additional examples, see "JSON Examples of Partial Update" on page 98.

The following table summarizes the parts of a `delete` operation.

Component	Req'd	Description
`select : path-expr`	Y	An XPath or JSONPath expression that selects the JSON property or array element to remove. If no matches are found for the `select` expression, the operation is silently ignored. The path expression is restricted to the subset of XPath that can be used to define an index, or its JSONPath equivalent. For details, see "Path Expressions Usable in Patch Operations" on page 126. The selected item(s) cannot be the target of any other operation in the patch. The ancestor of the selected item may not be modified by a `delete`, `replace`, or `replace-insert` operation in the same patch.
`cardinality: occurrence`	N	The required occurrence of matches to `select`. If the number of matches does not meet the expectation, the operation fails and an error is returned. Allowed values: • Zero or one matches required: `?` (question mark) • Exactly one match required: `.` (period) • Zero or more matches required: `*` (asterisk) • One or more matches required: `+` (plus) Default: * (The occurrence requirement is always met).

3.4.6.7 replace-library

Specify an XQuery library module that contains user-defined replacement content generation functions. These functions can be used in the `apply` property of a `replace` or `replace-insert` operation. A patch can contain at most one `replace-library`. For details, see "Constructing Replacement Data on the Server" on page 112.

For example, the following patch uses the function `my-ns:my-func` in the library module in the with the URI `/my.domain/my-module.xqy`.

```
{ "patch": [
   {"replace-library": {
     "at": "/my.domain/my-module.xqy",
     "ns": "my-ns"
```

```
    }},
    {"replace": {
      "select": "/inventory[name eq 'orange']/price",
      "apply": "my-func"
    }}
]}
```

For additional examples, see “Constructing Replacement Data on the Server” on page 112.

The following table summarizes the parts of a `<replace-library/>` element.

Component	Req’d	Description
`"at":` *`lib-module-path`*	N	The path to the XQuery library module containing user-defined replacement content generation functions. The module must be installed in the modules database associated with the REST API instance.
`"ns":` *`func-namespace`*	N	The module namespace alias defined by the `at` module.

3.4.7 JSON Examples of Partial Update

This section includes the examples of applying a partial update with a JSON patch. The following topics are covered:

- Example cURL Commands
- Example: Insert
- Example: Replace
- Example: Replace-Insert
- Example: Delete

To understand these examples, it is important that you understand how to use XPath to address JSON document components. For details, see Working With JSON in the *Application Developer’s Guide*.

3.4.7.1 Example cURL Commands

The examples in this section can be applied using commands similar to the following, assuming you copy the sample input document to a file called `example.json` and the patch to a file called `patch.json`.

```
# Windows users, see Modifying the Example Commands for Windows

# Create the example document
$ curl --anyauth --user user:password -X PUT -d @./example.json \
    -i -H "Content-type: application/json" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.json

# Apply the patch
$ curl --anyauth --user user:password -X POST -d @./patch.json \
    -i -H "Content-type: application/json" \
    -H "X-HTTP-Method-Override: PATCH" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.json

# Retrieve the results
curl --anyauth --user user:password -X GET \
    -i -H "Accept: application/json" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.json
```

3.4.7.2 Example: Insert

The following patch inserts a new property in the root object in the first child position, inserts a new array element value in the `child3` subject object, inserts a new property in the root object in last child position.

```
{ "patch": [
    { "insert": {
          "context": "/parent/child1",
          "position": "before",
          "content": { "INSERT1": "INSERTED1" }
    }},
    { "insert": {
          "context": "/parent/child3[1]",
          "position": "after",
          "content": "INSERTED2"
    }},
    { "insert": {
          "context": "/node()",
          "position": "last-child",
          "content": { "INSERT3": "INSERTED3" }
    }}
  ] }
```

The following table shows how applying the patch changes the target document.

Before Update	After Update

{ "parent": {
"child1": {
"grandchild": "value"
},
"child2": "simple",
"child3": ["av1", "av2"],
"child4": [
{ "a1" : "v1"},
{ "a2": "v2" }
]
} }
	{ "parent": {
	"INSERT1": "INSERTED1",
	"child1": { "grandchild": "value" },
	"child2": "simple",
	"child3": ["av1", **"INSERTED2"**, "av2"],
	"child4": [
	{ "a1": "v1" },
	{ "a2": "v2" }
	]
	},
	"INSERT3": "INSERTED3"
	}

To insert a property if and only if it does not already exist, use a predicate in the context path expression to test for existence. For example, the following patch will insert a property named TARGET only if the object does not already contain such a property:

```
{ "patch": [
  { "insert": {
          "context": "/parent[fn:empty(TARGET)]",
          "position": "last-child",
          "content": { "TARGET": "INSERTED" }
  }}
]}
```

This patch must use the `last-child` position because the context selects the node that will contain the new property.

For more details on the JSON document model and traversing JSON documents with XPath, see Working With JSON in the *Application Developer's Guide*.

3.4.7.3 Example: Replace

The patch in this example demonstrates the following types of replacement operations:

- Replace the value of a property with an object value (REPLACE1)
- Replace the value of a property with a simple value (REPLACED2)
- Replace the value of one item in an array (REPLACE3)
- Replace the value of an array (REPLACED4)
- Replace the value of all items in an array with the same value (REPLACED5)
- Replace the value of a nested array item (REPLACED6)

- Replace the value of one item in a nested array (REPLACED7)

The following patch replaces one property with another, replaces the simple value of a property, and replaces the array value of a property.

```
{ "patch": [
    { "replace": {
        "select": "/parent/child1",
        "content": { "REPLACE1": "REPLACED1" }
    }},
    { "replace": {
        "select": "/parent/child2",
        "content": "REPLACED2"
    }},
    { "replace": {
        "select": "/parent/child3[1]",
        "content": "REPLACED3"
    }},
    { "replace": {
        "select": "/parent/array-node('child4')",
        "content": [ "REPLACED4a", "REPLACED4b" ]
    }},
    { "replace": {
        "select": "/parent/child5",
        "content": "REPLACED5"
    }},
    { "replace": {
        "select": "/parent/array-node('child6')/node()[2]",
        "content": [ "REPLACED6a", "REPLACED6b" ]
    }},
    { "replace": {
        "select": "/parent/child7[2]",
        "content": "REPLACED7"
    }}
]}
```

The following table shows how applying the patch changes the target document. Further explanation follows the table.

Before Update	After Update

{ "parent": {
"child1": {
"grandchild": "value"
},
"child2": "simple",
"child3": ["av1", "av2"],
"child4": ["av1", "av2"],
"child5": ["av1", "av2"],
"child6": [
"av1",
["nav1", "nav2"],
"av2"
],
"child7": [
"av1",
["nav1", "nav2"],
"av2"]
} }
	{ "parent": {
	"child1": {
	"REPLACE1": "REPLACED1"
	},
	"child2": **"REPLACED2"**,
	"child3": [**"REPLACED3"**, "av2"],
	"child4": **["REPLACED4a", "REPLACED4b"]**,
	"child5": [**"REPLACED5"**, **"REPLACED5"**],
	"child6": [
	"av1",
	["REPLACED6a", "REPLACED6b"],
	"av2"
	],
	"child7": [
	"av1",
	[**"REPLACED7"**, "nav2"],
	"av2"
	]
	} }

The following table breaks the patch down into sections and describes how each operation works.

Patch Operation	Explanation

{"replace": {
"select": "/parent/child1",
"content": {
"REPLACE1": "REPLACED1" }
}}
	Replaces value of the “child1” property with a new value. The select path expression addresses the object node named “child1”. The replacement content is a new object node.

{"replace": { "select": "/parent/child2", "content": "REPLACED2" }}	Replaces the value of the “child2” property with a new atomic value. The select path expression addresses the text node named “child2”. The replacement content is a new text node.

Patch Operation	Explanation
{"replace": { "select": "/parent/child3[1]", "content": "REPLACED3" }}	Replaces the value of the first item in the array node named “child3” with a new value. All items in the array share the name “child3”, so the select path expression “/parent/child3” address every item in the array. The “[1]” index then selects the first of these. The replacement content is a new text value.
{"replace": { "select": "/parent/array-node('child4')", "content": ["REPLACED4a", "REPLACED4b"] }}	Replaces the value of the “child4” property with a new array value. The select expression addresses the array node named “child4”, rather than the individual array items as in the “child3” example. The replacement content is a new array node.
{"replace": { "select": "/parent/child5", "content": "REPLACED5" }}	Replaces all array items with the same value. All items in an array share the same name, so the select expression addresses all text nodes named “child5”. The replacement content for each item is a new text value.

Patch Operation	Explanation

`{"replace": {`
`  "select":`
`  "/parent/array-node('child6')/node()[2]",`
`  "content": [`
`  "REPLACED6a", "REPLACED6b"`
`  ]`
`}}`
	Replaces a nested array with a new array. Since all items in an array share the same name, recursively, an XPath expression such as “/parent/child6” addresses not only the “top level” items “av1” and “av2”, but also the items in the nested array, “nav1” and “nav2”. Thus, an index expression such as “/parent/child6[2]” is equivalent to selecting the second item in an array of the form [“av1”, “nav1”, “nav2”, “av2”]. To manipulate the nested array as an array, you must use explicit node accessors. The select expression “/parent/array-node('child6')/node()” addresses the 3 top level nodes, the second of which is the nested array. The replacement content is a new array. Contrast this example with next one, which replaces one item in the nested array.

`{"replace": {`
`  "select": "/parent/child7[2]",`
`  "content": "REPLACED7"`
`}}`
	Replaces the second item in an array with a new value. Since all items in an array share the same name, recursively, an XPath expression such as “/parent/child6” addresses not only the “top level” items “av1” and “av2”, but also the items in the nested array, “nav1” and “nav2”. Thus, an index expression such as “/parent/child6[2]” is equivalent to selecting the second item in an array of the form [“av1”, “nav1”, “nav2”, “av2”]. The replacement value is a new text value.

For more details on the JSON document model and traversing JSON documents with XPath, see Working With JSON in the *Application Developer's Guide*.

3.4.7.4 Example: Replace-Insert

The following patch demonstrates `replace-insert` on array values. The patch includes two operations for each child of parent, one that results in a replace and one that results in an insert.

```
{ "patch": [
    { "replace-insert": {
```

```
            "select": "/parent/child1[1]",
            "context": "/parent/array-node('child1')",
            "position": "last-child",
            "content": "REPLACED1"
        }},
        { "replace-insert": {
            "select": "/parent/child1[3]",
            "context": "/parent/child1[2]",
            "position": "after",
            "content": "INSERTED1"
        }},
        { "replace-insert": {
            "select": "/parent/child2[. = 'c2_v1']",
            "context": "/parent/node('child2')",
            "position": "last-child",
            "content": "REPLACED2"
        }},
        { "replace-insert": {
            "select": "/parent/child2[. = 'INSERTED2']",
            "context": "/parent/node('child2')",
            "position": "last-child",
            "content": "INSERTED2"
        }},
        { "replace-insert": {
            "select": "/parent/child3[c3_a]",
            "context": "/parent/node('child3')",
            "position": "last-child",
            "content": { "REPLACED3" : "REPLACED_V" }
        }},
        { "replace-insert": {
            "select": "/parent/child3[INSERTED3]",
            "context": "/parent/node('child3')",
            "position": "last-child",
            "content": { "INSERTED3" : "INSERTED_V" }
        }}
    ]}
```

The following table shows how applying the patch changes the target document.

Before Update	After Update

{"parent": {
  "child1": ["c1_v1", "c1_v2"],
  "child2": ["c2_v1", "c2_v2"],
  "child3": [
  { "c3_a": "c3_v1" },
  { "c3_b": "c3_v2" }
  ]
} }
	{ "parent": {
	  "child1": [
	  "REPLACED1",
	  "c1_v2",
	  "INSERTED1"
	  ],
	  "child2": [
	  "REPLACED2",
	  "c2_v2",
	  "INSERTED2"
	  ],
	  "child3": [
	  { "REPLACED3": "REPLACED_V" },
	  { "c3_b": "c3_v2" },
	  { "INSERTED3": "INSERTED_V" }
	  ]
	} }

Recall that the `select` path identifies the content to replace. When working with an array item, an absolute path is usually required. For example, consider the following patch operation:

```
{ "replace-insert": {
    "select": "/parent/child1[1]",
    "context": "/parent/array-node('child1')",
    "position": "last-child",
    "content": "REPLACED1"
}}
```

The goal is to replace the value of the first item in the array value of `/parent/child1` if it exists. If the array is empty, insert the new value. That is, one of these two transformations takes place:

```
{"parent"{"child1": ["c1_v1", "c1_v2"], ... }
    ==> {"parent"{"child1": ["REPLACED1", "c1_v2"], ... }

{"parent"{"child1": [], ... }
    ==> {"parent"{"child1": ["REPLACED1"], ... }
```

The `select` expression, `/parent/child1[1]`, must target an array item value, while the context expression must target the containing array node by referencing `/parent/array-node("child1")`. You cannot make the select expression relative to the context expression in this case.

Note that while you can target an entire array item value with `replace-insert`, you cannot target just the value of a property. For example, consider the following array:

```
"child3": [
  { "c3_a": "c3_v1" },
  { "c3_b": "c3_v2" }
]
```

You can use `replace-insert` on the entire object-valued item `{ "c3a": "c3v1" }`, as is done in the example. However, you cannot construct an operation that targets just the value of the property in the object (`"c3_v1"`). The replacement of the property value is fundamentally different from inserting a new item in the array. A property (as opposed to the containing object) can only be replaced by deleting it and then inserting a new item.

You cannot use a `replace-insert` operation to conditionally insert or replace a property because the insertion content and the replacement content requirements differ. However, you can use separate insert and replace operations within the same patch to achieve the same effect.

For example, the following patch inserts a new property named TARGET if it does not already exists, and replaces its value if it does already exist:

```
{ "patch": [
  { "insert": {
          "context": "/parent[fn:empty(TARGET)]",
          "position": "last-child",
          "content": { "TARGET": "INSERTED" }
  }},
  { "replace": {
        "select": "/parent/TARGET",
        "content": "REPLACED"
  }}
]}
```

The following table illustrates the effect of applying the patch:

Before Update	After Update

{"parent": {
  "child": "some_value"
}}
	{"parent":{
	  "child": "some_value",
	  "TARGET": "INSERTED"
	}}

{"parent": {
  "child": "some_value",
  "TARGET": "INSERTED"
}}
	{"parent": {
	  "child": "some_value",
	  "TARGET": **"REPLACED"**
	}}

For more details on the JSON document model and traversing JSON documents with XPath, see Working With JSON in the *Application Developer's Guide*.

3.4.7.5 Example: Delete

The following patch removes properties and array elements at various levels of a document. See the table below for how the patch affects the example content.

```
{{ "patch": [
    { "delete" : {
        "select": "/props/node('anyType')"
    }},
    { "delete" : {
        "select": "/props/objOrLiteral"
    }},
    { "delete" : {
        "select": "/props/array-node('arrayVal')"
    }},
    { "delete" : {
        "select": "/arrayItems/all"
    }},
    { "delete" : {
        "select": "/arrayItems/byPos[1]"
    }},
    { "delete" : {
        "select": "/arrayItems/byVal[. = 'DELETE']"
    }},
    { "delete" : {
        "select": "/arrayItems/byName[DELETE]"
    }}
] }
```

The following table shows how applying the patch changes the target document.

Before Update	After Update

`{ "props": {`
`  "anyType": [1, 2],`
`  "objOrLiteral": "anything",`
`  "arrayVal": [3, 4]`
`},`
`"arrayItems": {`
`  "byPos": ["DELETE", "PRESERVE"],`
`  "byVal": ["DELETE", "PRESERVE"],`
`  "byName": [`
`  {"DELETE": 5},`
`  {"PRESERVE": 6}`
`  ],`
`  "all": ["DELETE1", "DELETE2"]`
`}`
`}`
	`{ "props": { },`
	`"arrayItems": {`
	`  "byPos": [ "PRESERVE" ],`
	`  "byVal": [ "PRESERVE" ],`
	`  "byPropName": [`
	`  { "PRESERVE": 6 }`
	`  ],`
	`  "all": [ ]`
	`}`
	`}`

Note that when removing properties, you must either use a named node step to identify the target property or be aware of the value type. Consider these 3 `select` path expressions from the example program:

```
/props/node('anyType')
/props/objOrLiteral'
/props/array-node('arrayVal')
```

The first path is type agnostic: `/props/node("anyType")`. This expression selects any nodes named `anyType`, regardless of the type of value. The second path, `/props/objOrLiteral`, deletes the entire name-value pair only if the value is an object or literal (string, number, boolean). If the value of the property is an array, it deletes the items in the array. That is, this operation applied to the original document will delete the contents of the array, not the `arrayVal` property:

```
pb.remove('/props/arrayVal')
==> arrayVal: [ ]
```

The third form, `/props/array-node('arrayVal')`, deletes the `arrayVal` property, but it will only work on properties with array type. Therefore, if you need to delete a property by name without regard to its type, use path of the form `/path/to/parent/node('propName')`.

For more details on the JSON document model and traversing JSON documents with XPath, see Working With JSON in the *Application Developer's Guide*.

3.4.8 Patching Metadata

You can partially update metadata for any document type, including XML, JSON, text, and binary. To update a portion of the metadata for a document, send a POST request to the `/documents` service with a URI of the following form.

```
http://host:port/version/documents?uri=document-uri&category=category
```

The category parameter should either be `metadata` or one of the metadata sub-categories of `collections`, `properties`, `permissions`, `metadata-values`, and `quality`. The category affects what pieces of metadata MarkLogic Server considers for updating. For example, if you use `category=permissions`, but your patch only contains an operation applied to properties, the patch has no effect.

Category does not affect how you construct path expressions in a patch. That is, `context` and `select` paths within the patch are not relative to the specified metadata category.

The request body must contain an XML or JSON patch descriptor. Construct your metadata patch to apply to the structure described in "Working with Metadata" on page 142, using the same syntax and semantics as for a content patch. For details, see in "XML Patch Reference" on page 70 or "JSON Patch Reference" on page 88.

> **Note:** Document quality (the `<quality/>` XML element or `quality` JSON property) can only be the target of a replace operation.

You can update content and metadata in the same request by including operations on both in your patch descriptor and setting the category request parameter appropriately. For example:

```
http://localhost:8000/LATEST/documents?uri=/my.xml&category=content&category=metadata
```

When patching content and metadata in the same request, the usual content type restrictions apply: You can only apply content patches to XML and JSON documents, and you cannot use a JSON patch descriptor on XML content, or vice versa.

Since JSON does not use qualified names, you can construct a `context` or `select` path that addresses both metadata and content items. If your content contains JSON properties with the same name as the metadata properties (`properties`, `collections`, `permissions`, `metadataValues`, and `quality`) and you cannot construct an unambiguous path, do not combine content and metadata partial updates in the same request.

Paths for `context` and `select` can be relative or absolute with respect to the metadata structure. For example, both of the following operations insert a document into a new collection, using different `context` paths.

XML	JSON
<rapi:insert **context="/rapi:metadata/rapi:collections"** position="last-child"> <rapi:collection>INSERTED</rapi:collection> </rapi:insert>	{"insert": { **"context":"$.collections"**, "position":"last-child", "content":"INSERTED" }}
<rapi:insert **context="rapi:collections"** position="last-child"> <rapi:collection>INSERTED</rapi:collection> </rapi:insert>	{"insert": { **"context":"collections"**, "position":"last-child", "content":"INSERTED" }}

The following example is a patch that performs multiple metadata updates: Add the target document to a new collection, add or replace a property, and replace the document quality:

Format	Patch
XML	<rapi:patch xmlns:rapi="http://marklogic.com/rest-api" xmlns:prop="http://marklogic.com/xdmp/property"> <rapi:insert context="/rapi:metadata/rapi:collections" position="last-child"> <rapi:collection>INSERTED</rapi:collection> </rapi:insert> <rapi:replace-insert select="my-property" context="/rapi:metadata/prop:properties" position="last-child"> <my-property>REPLACE_OR_INSERT</my-property> </rapi:replace-insert> <rapi:replace select="/rapi:metadata/rapi:quality">2</rapi:replace> </rapi:patch>

JSON	{ "patch" : [
	{ "insert": {
	"context": "$.collections",
	"position": "last-child",
	"content" : "INSERTED"
	}},
	{ "replace-insert": {
	"select": "my-property",
	"context": "$.properties",
	"position": "last-child",
	"content" : { "my-property": "REPLACE_OR_INSERT" }
	}},
	{ "replace": {
	"select": "$.quality",
	"content" : 2
	}}
	],
	"pathlang": "jsonpath"
	}

Note: When patching properties using XML, you must explicitly declare the namespace `http://marklogic.com/xdmp/property` on the root `<patch/>` element because the `<properties/>` metadata element is in a different namespace from the other metadata elements.

Note: Values metadata (`category=metadata-values`) is represented differently in XML and JSON. The equivalent of the XML `metadata-values` element is a JSON property named `metadataValues`, and where XML has `metadata-value` child

elements to represent each key-value pair, JSON has *key*:*value* child properties. For details, see "Working with Metadata" on page 142.

3.4.9 Constructing Replacement Data on the Server

You can use builtin or user-defined replacement functions to generate the content for a partial update operation dynamically on MarkLogic Server. The builtin functions support simple arithmetic and string manipulation. For example, you can use a builtin function to increment the current value of numeric data or concatenate strings. For more complex operations, create and install a user-defined function.

The following topics are covered:

- Using a Replacement Constructor Function
- Using Builtin Replacement Constructors
- Writing an XQuery User-Defined Replacement Constructor
- Writing a JavaScript User-Defined Replacement Constructor
- Installing or Updating a User-Defined Replace Library

3.4.9.1 Using a Replacement Constructor Function

You can only use replacement generator functions with the `replace` and `replace-insert` operations. The REST Client API ships with a set of builtin arithmetic and string manipulation functions. If your patch includes a `replace-library` specification, you can also create user-defined functions.

The following example increases every `price` in an `inventory` element or object by 10% by using the builtin `ml.multiply` function to multiply the current value by 1.1.

XML	JSON

`<rapi:patch`
`  xmlns:rapi="http://marklogic.com/rest-api">`
`  <rapi:replace`
`  select="/inventory/price"`
`  **apply="ml.multiply"**>**1.1**</rapi:replace>`
`</rapi:patch>`
	`{"patch": [`
	`  {"replace": {`
	`  "select": "/inventory/price",`
	`  **"apply": "ml.multiply"**,`
	`  "content": **1.1**`
	`  } }`
	`]}`

The following example uses a user-defined function to double the `price` in every `inventory` element or object. The `dbl` function is implemented by the XQuery module `/my.domain/my-lib.xqy`, which uses the module namespace `http://my/ns`. The `dbl` function does not expect any argument values, so there is no content included in the `replace` operation.

XML	JSON
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api"> **<rapi:replace-library** **at="/my.domain/my-lib.xqy"** **ns="http://my/ns" />** <rapi:replace select="/inventory/price" **apply="dbl"/**> </rapi:patch>	{"patch": [{**"replace-library": {** **"at": "/my.domain/my-lib.xqy",** **"ns": "http://my/ns"** } }, {"replace": { "select": "/inventory/price", **"apply": "dbl"** } }] }

Use the following guidelines to specify a replacement generator function in a patch descriptor:

1. Set the value of the `apply` XML attribute or JSON property to the function local name.

2. If the function is a user-defined function, include a `replace-library` XML element or JSON property in the `patch` to specify the module containing the function implementation. The implementation module must be installed in the modules database of the REST API instance.

3. If the function expects arguments, specify them as child nodes of the operation element in XML or in the value of the `content` JSON property. See the example below.

If a function expects a single argument, you can place the value directly in the content or in a `<rapi:value/>` XML element or `$value` JSON property. The previous example places the value (1.1) directly in the content section of the patch.

If a function expects multiple arguments, enclose each value in a `<rapi:value/>` XML element or `$value` JSON property. For example, the following patch uses the builtin function `ml.concat-between` which expects two string parameters.

XML	JSON
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api"> <rapi:replace select="/my/data" apply="ml.concat-between"> **<rapi:value>my prefix</rapi:value>** **<rapi:value>my suffix</rapi:value>** </rapi:replace> </rapi:patch>	{"patch": [{"replace": { "select": "/my/data", "apply": "ml.concat-between", **"content": [** **{ "$value" : "my prefix" },** **{ "$value" : "my suffix" }**] } }] }

In XML, the implicit datatype of arguments is xs:`untypedAtomic`, though a generator function can cast the value to the expected parameter type. For example, the builtin arithmetic functions cast to `xs:double` when possible. Use `@xsi:type` to explicitly specify a type on a value.

The following example specifies an explicit `xs:date` datatype. Note that you must declare the xs and `xsi` namespace aliases on the root of your `patch`:

```
<rapi:patch
    xmlns:rapi="http://marklogic.com/rest-api"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <rapi:replace select="/elem" apply="my-func">
    <rapi:value xsi:type="xs:date">2013-03-15</rapi:value>
  </rapi:replace>
</rapi:patch>
```

In JSON, use `$datatype` to explicitly specify an `xsi:type` for a `$value`. Omit the `xs:` namespace prefix on the type name. The following example explicitly specifies `xs:date` as the datatype for the replace function input parameter:

```
{"patch": [
  {"replace": {
    "select": "/my/data",
    "apply": "ml.concat-between",
    "content": [
      { "$value" : "2013-03-15", $datatype: "date"}
    ]
  } }
] }
```

3.4.9.2 Using Builtin Replacement Constructors

The REST Client API includes several builtin server-side functions you can use to dynamically generate the content for a `replace` or `replace-insert` operation. For example, you can use a builtin function to increment the current value of a data item.

All the builtin function names have a "`ml.`" prefix. You cannot use this prefix on user-defined replacement generator functions.

The builtin arithmetic functions are equivalent to the XQuery `+`, `-`, `*`, and `div` operators, and accept values castable to the same datatypes. That is, numeric, date, dateTime, duration, and Gregorian (`xs:gMonth`, `xs:gYearMonth`, etc.) values. The operand type combinations are as supported by XQuery; for details, see http://www.w3.org/TR/xquery/#mapping. All other functions expect values castable to string.

The table below lists the builtin replacement generator functions provided by the REST Client API. In the table, *`$current`* represents the current value of the target of the replace operation; *`$arg`* and *`$argN`* represent argument values passed in by the patch.

Function Name	Num Args	Effect
`ml.add`	1	`$current + $arg`
`ml.subtract`	1	`$current - $arg`
`ml.multiply`	1	`$current * $arg`
`ml.divide`	1	`$current div $arg`
`ml.concat-before`	1	`fn:concat($arg, $current)`
`ml.concat-after`	1	`fn:concat($current, $arg)`
`ml.concat-between`	2	`fn:concat($arg1, $current, $arg2)`
`ml.substring-before`	1	`fn:substring-before($current, $arg)`
`ml.substring-after`	1	`fn:substring-after($current, $arg)`
`ml.replace-regex`	2 or 3	`fn:replace($current, $arg1, $arg2, $arg3)`

3.4.9.3 Writing an XQuery User-Defined Replacement Constructor

This section describes how to implement a custom replacement content constructor in XQuery. You can use such a function to generate content for the `replace` and `replace-insert` operations. You can also implement a constructor in Server-Side JavaScript; for details, see "Writing a JavaScript User-Defined Replacement Constructor" on page 118.

You must install your implementation in the modules database associated with your REST API instance before you can use it. For details, see "Installing or Updating a User-Defined Replace Library" on page 121.

A user-defined replacement constructor function has the following interface:

```
declare function module-ns:func-name(
  $current as node()?,
  $args as item()*
) as node()*
```

The current node (`$current`) is empty only when the function is invoked as an insert on behalf of a `replace-insert`.

The argument list supplied by the operation is passed through `args`. You are responsible for validating the argument values. If the content supplied by the patch operation is JSON array or a sequence of XML `<rapi:value/>` elements, then `$args` is the result of calling the `fn:data` function on each value. If an explicit datatype is specified by the patch operation, the cast is applied before invoking your function.

Your function should report errors using `fn:error` and `RESTAPI-SRVEXERR`. For details, see "Reporting Errors" on page 328.

The following example XQuery library module implements two replacement content constructors, `mylib:dbl` to double the value in a target node, and `my-lib:min` to replace the value of a node with the minimum value of the current node and a sequence of values passed in by the patch. For simplicity, this example skips most of the input data validation that a production implementation should include.

```
xquery version "1.0-ml";

module namespace my-lib = "http://marklogic.com/example/my-lib";

(: Double the value of a node :)
declare function my-lib:dbl(
  $current as node()?,
  $args as item()*
) as node()*
{
  if ($current/data() castable as xs:decimal)
  then
    let $new-value := xs:decimal($current) * 2
```

```
    return
      typeswitch($current)
      case number-node()       (: JSON :)
        return number-node {$new-value}
      case element()           (: XML :)
        return element {fn:node-name($current)} {$new-value}
      default return fn:error((), "RESTAPI-SRVEXERR",
         ("400", "Bad Request",
          fn:concat("Not an element or number node: ",
                    xdmp:path($current))
          ))
  else fn:error((), "RESTAPI-SRVEXERR",
    ("400", "Bad Request", fn:concat("Non-decimal data: ", $current)))
};

(: Find the minimum value in a sequence of value composed of :)
(: the current node and a set of input values. :)
declare function my-lib:min(
  $current as node()?,
  $args as item()*
) as node()*
{
  if ($current/data() castable as xs:decimal)
  then
    let $new-value := fn:min(($current, $args))
    return
      typeswitch($current)
      case element()           (: XML :)
        return element {fn:node-name($current)} {$new-value}
      case number-node()       (: JSON :)
        return number-node {$new-value}
      default return fn:error((), "RESTAPI-SRVEXERR",
        ("400", "Bad Request",
           fn:concat("Not an element or number node: ",
                     xdmp:path($current))))
  else fn:error((), "RESTAPI-SRVEXERR", ("400", "Bad Request",
         fn:concat("Non-decimal data: ", $current)))
};
```

The following patch snippet uses the above module, assuming the module is installed in the modules database with the URI `/ext/replace/my-lib.xqy`:

Format	Patch

XML	`<rapi:patch`
	`  xmlns:rapi="http://marklogic.com/rest-api">`
	`  **<rapi:replace-library at="/ext/replace/my-lib.xqy"**`
	`  **ns="http://marklogic.com/example/my-lib" />**`
	`  <rapi:replace select="/parent/child" apply="dbl"/>`
	`</rapi:patch>`

JSON	`{"patch": [`
	`  {**"replace-library": {**`
	`  **"at": "/ext/replace/my-lib.xqy",**`
	`  **"ns": "http://marklogic.com/example/my-lib"**`
	`  } },`
	`  {"replace": {`
	`  "select": "/parent/child",`
	`  "apply": "dbl"`
	`  } }`
	`] }`

3.4.9.4 Writing a JavaScript User-Defined Replacement Constructor

This section describes how to implement a custom replacement content constructor in Server-Side JavaScript. You can use such a function to generate content for the `replace` and `replace-insert` operations. You can also implement a constructor in XQuery; for details, see "Writing an XQuery User-Defined Replacement Constructor" on page 116.

You must install your implementation in the modules database associated with your REST API instance before you can use it. For details, see "Installing or Updating a User-Defined Replace Library" on page 121.

A user-defined replacement constructor function has the following interface:

```
function funcname(current, args)
```

Where the parameters have the following semantics:

- The *current* parameter holds the node targeted for replacement. It can be null if the function is invoked on behalf of an insertion, such as a `replace-insert` operation that is inserting new content.
- The *args* parameters contains any user-defined data defined in the patch operation via the `rapi:values` XML element or `$value` JSON property. It can be null, a single item or a `Sequence`, depending on whether the operation defines zero, one, or multiple values.

You are responsible for validating the *args* values. The value of each item in *args* is the result of calling the `fn:data` function on each value defined by the operation. If an explicit datatype is specified by the patch operation, the cast is applied before invoking your function.

Your function must return zero, one, or more nodes containing the generated content. Use a Sequence to return multiple nodes.

Your function should report errors using `fn:error` and `RESTAPI-SRVEXERR`. For details, see "Reporting Errors" on page 328.

The following example implements two replacement content constructors, `dbl` to double the value in a target node, and `min` to replace the value of a node with the minimum value of the current node and a sequence of values passed in by the patch. For simplicity, this example skips most of the input data validation that a production implementation should include. The example functions operate on either an XML element or a JSON number node. In production, you would likely have different functions for different content types.

```
'use strict';

function dbl(current,args) {
  switch(xdmp.nodeKind(current)) {
    case "number": {
      return new NodeBuilder()
          .addNumber(fn.data(current).valueOf() * 2)
          .toNode();
    }
    case "element": {
      const currentValue = fn.data(current).valueOf();
      if (xdmp.castableAs(
           "http://www.w3.org/2001/XMLSchema",
           "decimal", currentValue)) {
        return new NodeBuilder()
          .addElement(fn.nodeName(current).toString(),
                      xs.string(currentValue * 2))
          .toNode();
      } else {
        fn.error(null, 'RESTAPI-SRVEXERR', Sequence.from([
          '400', 'Bad Request',
          'Non-decimal data: ' + xdmp.path(current)]));
      }
    }
    default:
      fn.error(null, 'RESTAPI-SRVEXERR', Sequence.from([
        '400', 'Bad Request',
        'Not an element or number node: ' + xdmp.path(current)]));
  }
};

function min(current, args) {
  switch(xdmp.nodeKind(current)) {
    case "number": {
```

```
        return new NodeBuilder()
          .addNumber(Math.min(
            ...args.toArray().concat(fn.data(current).valueOf())))
          .toNode();
      }
      case "element": {
        const currentValue = fn.data(current).valueOf();
        if (xdmp.castableAs(
            "http://www.w3.org/2001/XMLSchema",
            "decimal", currentValue)) {
          return new NodeBuilder()
            .addElement(fn.nodeName(current).toString(),
               xs.string(Math.min(...args.toArray().concat(currentValue))))
            .toNode();
        } else {
          fn.error(null, 'RESTAPI-SRVEXERR', Sequence.from([
            '400', 'Bad Request',
            'Non-decimal data: ' + xdmp.path(current)]));
        }
      }
      default:
        fn.error(null, 'RESTAPI-SRVEXERR', Sequence.from([
          '400', 'Bad Request',
          'Not an element or number node: ' + xdmp.path(current)]));
    }
};


exports.dbl = dbl;
exports.min = min;
```

The following patch snippet uses the `dbl` function of the above module, assuming the module is installed in the modules database with the URI `/ext/replace/my-lib.sjs`:

Format	Patch

XML	`<rapi:patch`
	`  xmlns:rapi="http://marklogic.com/rest-api">`
	`  **<rapi:replace-library at="/ext/replace/my-lib.sjs" />**`
	`  <rapi:replace select="/parent/child" **apply="dbl"**/>`
	`</rapi:patch>`

JSON	`{"patch": [`
	`  {**"replace-library": {**`
	`  **"at": "/ext/replace/my-lib.sjs"**`
	`  } },`
	`  {"replace": {`
	`  "select": "/parent/child",`
	`  "apply": "dbl"`
	`  } }`
	`] }`

The following patch snippet illustrates how to pass arguments to the min function of the same module. Two numeric values are passed into the min function, in the form of a `Sequence`.

Format	`Patch`

XML	`<rapi:patch`
	`  xmlns:rapi="http://marklogic.com/rest-api">`
	`  **<rapi:replace-library at="/ext/replace/my-lib.sjs" />**`
	`  <rapi:replace select="/parent/child" **apply="min"**/>`
	`  <rapi:value>20</rapi:value>`
	`  <rapi:value>2</rapi:value>`
	`  </rapi:replace>`
	`</rapi:patch>`

JSON	`{"patch": [`
	`  {**"replace-library": {**`
	`  **"at": "/ext/replace/my-lib.sjs"**`
	`  } },`
	`  {"replace": {`
	`  "select": "/parent/child",`
	`  "apply": "min",`
	`  "content" : [{"$value": 20}, {"$value": 2}]`
	`  } }`
	`] }`

3.4.9.5 Installing or Updating a User-Defined Replace Library

To install user-defined replacement constructor functions, place your function(s) into an XQuery or Server-Side JavaScript library module and install the module in the modules database associated with your REST API instance.

Your implementation, including any dependent libraries, must be installed in the modules database associated with your REST API instance. The simplest way to achieve this is to use the `/ext` service that is part of the REST Client API. For details, see “Managing Dependent Libraries and Other Assets” on page 376.

To install your library using the `/ext` service, send a PUT request to a URL of one of the following forms, with your library module implementation in the request body:

`http://`*`host`*`:`*`port`*`/`*`version`*`/ext/`*`your/path/your-lib`*`.xqy?perm:`*`perm`*

`http://`*`host`*`:`*`port`*`/`*`version`*`/ext/`*`your/path/your-lib`*`.sjs?perm:`*`perm`*

If you do not specify any permissions, the module will only be executable by users with the `rest-admin` role.

When you use `/ext` to install your module, MarkLogic Server installs the module in the request body into the modules database associated with your REST API instance, at a URI derived from the above URL, beginning with `/ext`. For example, the following command installs the library module with the URI `/ext/replace/my-lib.xqy`:

```
$ curl --anyauth --user user:password -X PUT -d @./my-lib.xqy \
    -i -H "Content-type: application/xquery" \
  http://localhost:8000/LATEST/ext/replace/my-lib.xqy?perm:my-user:execute
```

The following patch snippet uses the module URI in the value of the at XML attribute or JSON property in a `replace-lib` directive:

```
<rapi:replace-library at="/ext/replace/my-lib.xqy"
   ns="http://marklogic.com/example/my-lib" />
```

If your library module requires dependent libraries, you can install them in the same database directory (`/ext/replace/`, in the above example). The `/ext` service allows you to manage at both the directory and file level. For details, see “Managing Dependent Libraries and Other Assets” on page 376.

Use the same procedure to update your implementation after initial installation.

3.4.10 How Position Affects the Insertion Point

The `insert` and `replace-insert` patch operations include a position member that defines the point of insertion when coupled with a context expression. This section describes the details of how position affects the insertion point. The following topics are covered:

- Specifying Position in XML
- Specifying Position in JSON

3.4.10.1 Specifying Position in XML

The `@position` of an `<insert/>` or `<replace-insert/>` operation specifies where to insert new content, relative to the `context` node or attribute. Position can be one of the following values:

- `before`: Insert before the element or attribute identified by `@context`. The `@context` must not target the root element.
- `after`: Insert after the element or attribute identified by `@context`. The `@context` must not target the root element.
- `last-child`: Insert as the last child of the element identified by `@context`. The `@context` must target a node, not an attribute.

The `last-child` is not meaningful for operations on attributes.

The following table shows example combinations of `@context` and `@position` and the resulting insertion point for new content. The insertion point is indicated by `***`.

Context	Position	Example Insertion Point

`/parent/child[1]`	`before`	`<parent>`
		`  ***`
		`  <child>`
		`  <grandchild/>`
		`  </child>`
		`  <child a1="val" a2="val"/>`
		`</parent>`

`/parent/child`	`after`	`<parent>`
		`  <child>`
		`  <grandchild/>`
		`  </child>`
		`  ***`
		`  <child a1="val" a2="val"/>`
		`  ***`
		`</parent>`

`/parent/child[1]`	`last-child`	`<parent>`
		`  <child>`
		`  <grandchild/>`
		`  ***`
		`  </child>`
		`  <child a1="val" a2="val"/>`
		`</parent>`

`/parent/child/@a1`	`before`	`<parent>`
		`  <child>`
		`  <grandchild/>`
		`  </child>`
		`  <child *** a1="val" a2="val"/>`
		`</parent>`

`/parent/child/@a1`	`after`	`<parent>`
		`  <child>`
		`  <grandchild/>`
		`  </child>`
		`  <child a1="val" *** a2="val"/>`
		`</parent>`

Context	Position	Example Insertion Point
/parent/child	last-child	<parent> <child ***> <grandchild/> </child> <child a1="val" a2="val" *** /> </parent>
/parent/child/@a1	last-child	Error. You cannot insert into an attribute.

3.4.10.2 Specifying Position in JSON

The `position` property in a patch specifies where to insert new content relative to the `context` item. Position can be one of the following values:

- `before`: Insert before the property or array element value selected by `context`.
- `after`: Insert after the property or array element value selected by `context`.
- `last-child`: Insert into the value of the target property or array, in the position of last child. The value selected by `context` must be an object or array.

The same usage applies whether inserting on behalf of an `insert` operation or a `replace-insert` operation.

You cannot use `last-child` to insert a property as an immediate child of the root node of a document. Use before or after instead. For details, see “Limitations of JSON Path Expressions” on page 126.

The following table shows example combinations of `context` and `position` and the resulting insertion point for new content. The insertion point is indicated by `***`.

Context	Position	Example Insertion Point

`/theTop/node("child1")`
 	
`a property`
	`after`	`{"theTop" : {`
		`  "child1" : [ "val1", "val2" ],`
		`  ***`
		`  "child2" : [`
		`  {"one": "val1"},`
		`  {"two": "val2"}`
		`  ]`
		`} }`

`/theTop/child1[1]`
 	
`an array item`
	`before`	`{"theTop" : {`
		`  "child1" : [ *** "val1", "val2" ],`
		`  "child2" : [`
		`  {"one": "val1"},`
		`  {"two": "val2"}`
		`  ]`
		`} }`

`/theTop/array-node("child1")`
 	
an array
	`last-child`	`{"theTop" : {`
		`  "child1" : [ "val1", "val2" ***],`
		`  "child2" : [`
		`  {"one": "val1"},`
		`  {"two": "val2"}`
		`  ]`
		`} }`

`/node("theTop")`
 	
an object
	`last-child`	`{"theTop" : {`
		`  "child1" : [ "val1", "val2"],`
		`  "child2" : [`
		`  {"one": "val1"},`
		`  {"two": "val2"}`
		`  ]`
		`  ***`
		`} }`

`/theTop/child1`
 	
a value
	`last-child`	Error because the selected value is not an object or array. For example, an error results if the target document has the following contents:
		
		`{"theTop" : {`
		`  "child1" : "val1",`
		`  "child2" : [...]`
		`} }`

3.4.11 Path Expressions Usable in Patch Operations

A patch operation uses one or more path expressions to identify the target of the operation.

For performance and security reasons, the path expression used in the `context` and `select` specifications of a patch operation is limited to a subset of XPath. JSONPath expressions are constrained by equivalent limitations.

For details, see Patch Feature of the Client APIs in the *XQuery and XSLT Reference Guide*.

3.4.12 Limitations of JSON Path Expressions

You can replace just the value of a property, but you cannot replace an entire property with a new one in a single operation.

For example, given content of the following form, you can use a replace operation to change the value property “a” from 1 to 2:

```
{ "a": 1 }
```

However, you cannot use a replace operation to change the content to the following:

```
{ "b": "anyvalue" }
```

In this example, the path expression “/a” addresses a number node with name “a” and value 1. The select expression in the replace operation enables you to replace the value of the selected name, but not the name.

To achieve the desired effect, you must first delete the existing property, and then insert a new one.

This limitation also means you cannot use a `replace-insert` operation to perform an operation such as “insert this property if it doesn’t exist; otherwise, replace it”. However, you can use separate `insert` and `replace` operations in a single patch to perform an operation such as “insert this property if it doesn’t exist; otherwise, replace its value”. For an example, see “JSON Examples of Partial Update” on page 98.

3.4.13 Introduction to JSONPath

When applying partial updates to JSON documents, you can specify the target of a patch operation using JSONPath expressions in the `context` and `select` properties of a patch operation. This section gives a brief overview of JSONPath syntax. For a complete description of JSONPath, see http://goessner.net/articles/JsonPath/.

> **Note:** The default path expression language is XPath. Using JSONPath is deprecated. To use JSONPath, you must set the `pathlang` property to `jsonpath`. For details, see “pathlang” on page 89.

- The anonymous root of a JSON object is identified by a dollar sign ($).
- Delimit path steps by either periods (.) or square brackets ([]).
- Use square brackets to enclose predicate expressions and array element selectors.
- The first element in an array has position 0, which is different from XPath.
- Predicates must be enclosed in ?(), as in `[?(@.value < 10)]`.

The set of JSONPath expressions usable for document operations like partial update is limited to JSONPath equivalent to XPath expressions that can be used to define a path range index. For details, see “Path Expressions Usable in Patch Operations” on page 126.

The following table contains some simple JSONPath examples that satisfy the restrictions. The bold text indicates what is selected by each expression.

JSONPath	Selection

`$.parent`
 	
`$['parent']`
	{ **"parent": [**
	{"child": ["one", "two"] },
	{"child": {"grandchild": "three" }}
	] }

`$.parent[0]`
 	
`$['parent'][0]`
	{ "parent": [
	{"child": ["one", "two"] },
	{"child": {"grandchild": "three" }}
	] }

`$.parent['child']`	{ "parent": [**{"child": ["one", "two"] },** **{"child": {"grandchild": "three" }**}] }
`$.parent.child[*]['one']`	{ "parent": [{"child": [**"one"**, "two"] }, {"child": {"grandchild": "three" }}] }

3.5 Performing a Lightweight Document Check

Use this method to:

- Test for the existence of a document in the database.
- Retrieve a document identifier without fetching content or metadata when content versioning is enabled.
- Determine the total length of a document for setting the end boundary when iterating over content ranges in binary documents.

To perform a document check, construct a HEAD request to a URL of the form:

```
http://host:port/version/documents?uri=document_uri
```

The following example sends a HEAD request for an XML document:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X HEAD \
    -H "Accept: application/xml" \
    http://localhost:8000/LATEST/documents?uri=/xml/box.xml
...
Content-type: application/xml
Server: MarkLogic
Connection: close

HTTP/1.1 200 Document Retrieved
vnd.marklogic.document-format: xml
Content-type: application/xml
Server: MarkLogic
Connection: close
```

Issuing the same request on a non-existent document returns status 404:

```
$ curl --anyauth --user user:password -X HEAD \
    -H "Accept: application/xml" \
    http://localhost:8000/LATEST/documents?uri=/xml/dne.xml
...
Content-type: application/xml
Server: MarkLogic
Connection: close

HTTP/1.1 404 Not Found
Content-type: application/xml
Server: MarkLogic
Connection: close
```

3.6 Removing Documents from the Database

This section covers using the `/documents` and `/search` services to remove documents from the database. The following topics are covered:

- Removing a Document or Metadata
- Removing Multiple Documents
- Removing All Documents
- Removing a Semantic Graph

3.6.1 Removing a Document or Metadata

To remove a document and its metadata from the database, construct a DELETE request with a URL of the form:

```
http://host:port/version/documents?uri=document_uri
```

When you delete a document, its metadata is also deleted.

You can remove multiple documents (or reset their metadata) by specifying multiple URIs. Optimistic locking is not supported when you specify multiple URIs.

To remove or reset just metadata for a document, construct a DELETE request with a URL of the form:

```
http://host:port/version/documents?uri=document_uri&category=metadata_
category
```

The `category` parameter can appear multiple times, with the values described in “Metadata Categories” on page 142. Resetting permissions resets the document permissions to the default permissions for the current user. Resetting quality resets the document quality to the default (0).

Deleting a binary document with extracted metadata stored in a separate XHTML document also deletes the XHTML metadata document. For more information, see “Working with Binary Documents” on page 140.

3.6.2 Removing Multiple Documents

In addition to removing multiple documents by URI as described in “Removing a Document or Metadata” on page 129, you can remove all documents in a collection or in a database directory.

To remove all documents in a collection, send a DELETE request with a URL of the following form:

```
http://host:port/version/search?collection=collection_name
```

Similarly, to remove all documents in a directory, send a DELETE request with a URL of the following form:

```
http://host:port/version/search?directory=directory_name
```

Where *directory_name* is the name of a directory in the database. The directory name must include a trailing “/”.

You can specify only one collection or one directory in a single request.

> **Note:** Failing to specify either a directory or a collection removes all documents in the database.

Removing a document also removes its metadata. Deleting a binary document with extracted metadata stored in a separate XHTML document also deletes the XHTML metadata document. For more information, see “Working with Binary Documents” on page 140.

The following example remove all documents in the “/plays” directory:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -i -X DELETE \
    http://localhost:8000/LATEST/search?directory=/plays/
...
HTTP/1.1 204 Updated
Server: MarkLogic
Content-Length: 0
Connection: close
```

3.6.3 Removing All Documents

To remove all documents in the database, send a DELETE request with a URL of the following form:

```
http://host:port/version/search
```

Clearing the database requires the `rest-admin` role or equivalent.

There is no confirmation or other safety net when you clear the database in this way. Creating a backup is advised.

The following example removes all documents and metadata from the content database:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -i -X DELETE \
    http://localhost:8000/LATEST/search
```

3.6.4 Removing a Semantic Graph

To remove the triples in a graph, send a DELETE to the `/graphs` service with a URL of one of the following forms:

```
http://host:port/version/graphs?graph=graph-uri

http://host:port/version/graphs?default
```

The `/graphs` service only affects triples stored in a triple document (a document with a `<sem:triples>` root element). Therefore, if the graph includes triples embedded in normal XML documents, the embedded triples are unaffected and the graph will continue to exist.

You can also delete a graph by using an empty graph in a PUT request to the `/graphs` service.

For details, see *Semantics Developer’s Guide*.

3.7 Using Optimistic Locking to Update Documents

An application using optimistic locking creates a document only when the document does not exist and updates or deletes a document only when the document has not changed since this application last changed it. However, optimistic locking does not actually involve placing a lock on document.

Optimistic locking is useful in environments where integrity is important, but contention is rare enough that it is useful to minimize server load by avoiding unnecessary multi-statement transactions.

This section covers the following topics:

- Understanding Optimistic Locking
- Enabling Optimistic Locking
- Obtaining a Version Id
- Applying a Conditional Update
- Example: Updating a Document Using Optimistic Locking

3.7.1 Understanding Optimistic Locking

Consider an application that reads a document, makes modifications, and then updates the document in the database with the changes. The traditional approach to ensuring document integrity is to perform the read, modification, and update in a multi-statement transaction. This holds a lock on the document from the point when the document is read until the update is committed. However, this pessimistic locking blocks access to the document and incurs more overhead on the App Server.

With *optimistic locking*, the application does not hold a lock on a document between read and update. Instead, the application saves the document state on read, and then checks for changes at the time of update. The update fails if the document has changed between read and update. This is a *conditional update*.

Optimistic locking is useful in environments where integrity is important, but contention is rare enough that it is useful to minimize server load by avoiding unnecessary multi-statement transactions. The REST Client API also supports multi-statement transactions. For details, see “Managing Transactions” on page 301.

The REST Client API uses content versioning to implement optimistic locking. When content versioning is enabled, MarkLogic Server associates an opaque version id with a document. The version id changes each time you update the document. The version id is returned when you read a document, and you can pass it back in an update or delete operation to test for changes prior to commit.

Note: Content versioning in the REST Client API does *not* implement document versioning. When content versioning is enabled, MarkLogic Server does not keep multiple versions of a document or track what changes occur. The version id can only be used to detect that a change occurred.

Using optimistic locking in your application requires the following steps:

1. Enabling Optimistic Locking by setting the `update-policy` REST instance configuration property.

2. Obtaining a Version Id for documents you wish to conditionally update.

3. Applying a Conditional Update by supplying the version id in your update operations.

3.7.2 Enabling Optimistic Locking

Enable optimistic locking using the `update-policy` instance configuration property, as described in "Configuring Instance Properties" on page 47. For example, the following command sets `update-policy` to `version-optional`:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    -d'{"update-policy":"version-optional"}' \
    -H "Content-type: application/json" \
    http://localhost:8000/LATEST/config/properties
```

The `update-policy` property can be set to `merge-metadata` (the default), `version-required`, or `version-optional`. Set the property to `version-required` if you want every document update or delete operation to use optimistic locking. Set the property to `version-optional` to allow selective use of optimistic locking.

Note: The `update-policy` property replaces the older `content-versions` policy; `content-versions` is deprecated. Setting `update-policy` to `version-optional` is equivalent to setting `content-versions` to `optional`. Setting `update-policy` to `version-required` is equivalent to setting `content-versions` to `required`.

The table below describes how each setting for this property affects document operations.

Setting	Effect
`merge-metadata`	This is the default setting. If you insert, update, or delete a document that does not exist, the operation succeeds. If a version id is present, it is ignored.
`version-optional`	If you insert, update or delete a document that does not exist, the operation succeeds. If a version id is present, the operation fails if the document exists and the current version id does not match the supplied version id.
`version-required`	If you update or delete a document without supplying a version id and the document does not exist, then the operation succeeds; if the document exists, the operation fails. If a version id is present, the operation fails if the document exists and the current version id does not match the version in the header.
`overwrite-metadata`	The behavior is the same as `merge-metadata`, except that metadata in the request overwrites any pre-existing metadata, rather than being merged into it. This setting disables optimistic locking.

3.7.3 Obtaining a Version Id

When optimistic locking is enabled, a version id is included in the response when you read documents. You can only obtain a version id if optimistic locking is enabled by setting `update-policy`; for details, see “Enabling Optimistic Locking” on page 132.

You can obtain a version id for use in conditional updates in the following ways:

- Perform a single document GET or HEAD request to `/documents`. The version id is returned in the ETag header of the response.
- Perform a multi-document read request, such as GET `/documents` with multiple URIs. The version id is returned in the Content-Disposition header of each content part in the multipart response.

You can mix and match these methods for obtaining a version id.

The following example command returns the version id for a single document:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -i -X HEAD \
    -H "Accept: application/xml" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.xml
```

```
...
HTTP/1.1 200 Document Retrieved
Content-type: application/xml
ETag: "13473172834878540"
Server: MarkLogic
Connection: close
```

The following example command returns the version id for multiple documents in a single request:

```
curl --anyauth --user user:password -X GET -i \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  http://localhost:8000/LATEST/documents?uri=doc1.xml\&uri=doc2.json
...
--BOUNDARY
Content-Type: application/xml
Content-Disposition: attachment; filename="doc1.xml";
category=content; format=xml; versionId=14075140367230760
Content-Length: 87

...document contents...
--BOUNDARY
Content-Type: application/json
Content-Disposition: attachment; filename="doc2.json";
category=content; format=json; versionId=14075140367230760
Content-Length: 15

...document contents...
```

For details on multi-document requests, see “Reading and Writing Multiple Documents” on page 249.

3.7.4 Applying a Conditional Update

To apply a conditional update, supply a version id on your update operation. The version id is ignored if optimistic locking is not enabled; for details, see “Enabling Optimistic Locking” on page 132.

When a document update includes a version id, MarkLogic Server checks for changes to the version id before committing the update (or delete) operation. If the version id has changed, the update fails. In a multi-document update, the entire batch of updates is rejected if any conditional update fails.

You can supply a version id on an update in the following ways:

- Pass the version id in the If-Match header of a single document PUT, POST, PATCH, or DELETE request to `/documents`.
- Pass the version id in the Content-Disposition header of a content part in a multi-document POST request to `/documents`.

The following example performs a conditional update of a single document by passing a version id in the If-Match request header:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -i -X PUT -d"<modified-data/>"  \
    -H "Content-type: application/xml" \
    -H "If-Match: 13473769780393030" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.xml
```

The following POST body excerpt for a multi-document update unconditionally updates the first document (`doc1.xml`) and conditionally updates the second document (`doc2.xml`). Note that if `update-policy` is set to `version-required`, the request will fail if `doc1.xml` already exists because no version id is supplied for it.

```
--BOUNDARY
Content-Type: application/xml
Content-Disposition: attachment; filename="doc1.xml"
Content-Length: 87

...document contents...
--BOUNDARY
Content-Type: application/json
Content-Disposition: attachment; filename="doc2.json";
versionId=14075140367230760
Content-Length: 15

...document contents...
--BOUNDARY--
```

The following command applies the above update, if the complete POST body is in the file `/example/optimistic-locking`.

```
$ curl --anyauth --user user:password -X POST -i \
  --data-binary @/example/optimistic-locking \
  -H "Content-type: multipart/mixed; boundary=BOUNDARY" \
  -H "Accept: application/xml" \
  http://localhost:8000/LATEST/documents
```

For details on multi-document requests, see “Reading and Writing Multiple Documents” on page 249.

3.7.5 Example: Updating a Document Using Optimistic Locking

The following example demonstrates using optimistic locking to conditionally update a single document. Both a successful and a failed update are shown.

This example uses a PUT request to demonstrate a conditional update. For a POST, PATCH, or DELETE operation, follow a similar procedure, passing the version id in the If-Match header of your update or delete request. For an equivalent multi-document update, pass the version id in the Content-Disposition header of each content part.

1. If content versioning is not already enabled, enable it. The following command sets `update-policy` to `version-optional`:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    -d'{"update-policy":"version-optional"}' \
    -H "Content-type: application/json" \
    http://localhost:8000/LATEST/config/properties
```

2. Insert an example document into the database to initialize the example:

```
$ curl --anyauth --user user:password -i -X PUT -d"<data/>" \
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.xml
```

3. Unconditionally retrieve a local copy of the document. Note the version id is returned in the ETag header:

```
$ curl --anyauth --user user:password -i -X GET \
    -H "Accept: application/xml" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.xml
...
HTTP/1.1 200 Document Retrieved
vnd.marklogic.document-format: xml
Content-type: application/xml
ETag: "13473769780393030"
Server: MarkLogic
Content-Length: 47
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<data/>
```

4. Conditionally update the document by sending new contents with the version id from Step 3 in the If-Match header. Since the document has not changed since Step 3, the update succeeds. The document version id is changed by this operation.

```
$ curl --anyauth --user user:password -i -X PUT -d"<modified-data/>"  \
    -H "Content-type: application/xml" \
    -H "If-Match: 13473769780393030" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.xml
...
HTTP/1.1 204 Content Updated
Server: MarkLogic
Content-Length: 0
Connection: close
```

5. To illustrate update failure when the version ids do not match, attempt to update the document again, using the version id from Step 3, which is no longer current. The update fails.

```
$ curl --anyauth --user user:password -i -X PUT -d"<data/>"  \
    -H "Content-type: application/xml" \
    -H "If-Match: 13473769780393030" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.xml
...
HTTP/1.1 412 Precondition Failed
Content-type: application/xml
Server: MarkLogic
Content-Length: 370
Connection: close

<?xml version="1.0"?>
<rapi:error xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:status-code>412</rapi:status-code>
  <rapi:status>Precondition Failed</rapi:status>
  <rapi:message-code>RESTAPI-CONTENTWRONGVERSION</rapi:message-code>
  <rapi:message>RESTAPI-CONTENTWRONGVERSION: (err:FOER0000) Content
version mismatch:  uri: /docs/example.xml version:
13473788748796580</rapi:message>
</rapi:error>
```

3.8 Client-Side Cache Management Using Content Versioning

You can use content versioning to refresh a copy of a document stored on the client only if the document in the database has been modified. This section covers the following topics:

- Enabling Content Versioning
- Using Content Versioning for Cache Refreshing
- Example: Refreshing a Cached Document

3.8.1 Enabling Content Versioning

Enable content versioning using the `update-policy` instance configuration property, as described in “Configuring Instance Properties” on page 47.

The default update policy is `merge-metadata`. If you set `update-policy` to `version-required` or `version-optional`, content versioning is enabled, and a GET or HEAD request to `/documents` returns a version id in the ETag response header.

Note: Enabling content versioning can affects document insertion, update, and deletion. For details, see “Enabling Optimistic Locking” on page 132.

Note: Content versioning in the REST Client API does *not* implement document versioning. When content versioning is enabled, MarkLogic Server does not keep multiple versions of a document or track what changes occur. The version id can only be used to detect that a change occurred.

3.8.2 Using Content Versioning for Cache Refreshing

When content versioning is enabled, sending a GET request to `/documents` with a version id in the If-None-Match HTTP header only retrieves a new copy of the document if it has changed relative to the version id in the header.

> **Note:** The `update-policy` property replaces the older `content-versions` property; `content-versions` is deprecated. Setting `update-policy` to `version-optional` is equivalent to setting `content-versions` to `optional`. Setting `update-policy` to `version-required` is equivalent to setting `content-versions` to `required`.

Follow the procedure below to use content version for cache refreshing. For a complete example, see "Example: Refreshing a Cached Document" on page 138.

1. If content versioning is not already enabled, set the `update-policy` instance configuration property to `version-optional` or `version-required`; see "Enabling Content Versioning" on page 137.

2. Read a document by making a GET request to `/documents`. The response includes the version id in the ETag header.

3. Save the version id returned in the ETag response header.

4. When you want to refresh the cache, send another GET request to `/documents` with the version id from Step 2 in the If-None-Match HTTP header.

   If the current version id matches the one in the If-None-Match header, no document is retrieved and MarkLogic Server returns status 304. If the current version id differs from the one in the If-None-Match header, the document is returned, along with the new version id in the ETag response header.

3.8.3 Example: Refreshing a Cached Document

The following example demonstrates using content versioning to refresh a client-side document cache. The example includes includes a case where the document is unchanged in the database, as well as the case where the local cache is out of date.

1. If content versioning is not already enabled, enable it by setting `update-policy` to `version-optional`:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    -d'{"update-policy":"version-optional"}' \
    -H "Content-type: application/json" \
    http://localhost:8000/LATEST/config/properties
```

2. Insert a sample document into the database to initialize the example:

```
$ curl --anyauth --user user:password -i -X PUT -d"<data/>" \
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.xml
```

3. Unconditionally retrieve a local copy of the document, as if to cache it. Note the version id is returned in the ETag header:

```
$ curl --anyauth --user user:password -i -X GET \
    -H "Accept: application/xml" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.xml
...
HTTP/1.1 200 Document Retrieved
vnd.marklogic.document-format: xml
Content-type: application/xml
ETag: "13473769780393030"
Server: MarkLogic
Content-Length: 47
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<data/>
```

4. Conditionally retrieve the document, as if to refresh the cache. Supply the version id from Step 3 in the If-None-Match header. Since the document has not changed, no content is retrieved.

```
$ curl --anyauth --user user:password -i -X GET \
    -H "Accept: application/xml" \
    -H "If-None-Match: 13473769780393030" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.xml
...
HTTP/1.1 304 Content Version Not Modified
ETag: "13473769780393030"
Server: MarkLogic
Content-Length: 0
Connection: close
```

5. Update the document in the database, which changes the version id.

```
$ curl --anyauth --user user:password -i -X PUT -d"<modified-data/>"  \
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.xml
...
HTTP/1.1 204 Content Updated
Server: MarkLogic
Content-Length: 0
Connection: close
```

6. Conditionally retrieve the document again, as if to refresh the cache. Supply the version id from Step 3 in the If-None-Match header. Since the document has changed, the content is retrieved. The new version id is also returned via the ETag header.

```
$ curl --anyauth --user user:password -i -X GET \
    -H "Accept: application/xml" \
    -H "If-None-Match: 13473769780393030" \
    http://localhost:8000/LATEST/documents?uri=/docs/example.xml
...
HTTP/1.1 200 Document Retrieved
vnd.marklogic.document-format: xml
Content-type: application/xml
ETag: "13473770707201670"
Server: MarkLogic
Content-Length: 56
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<modified-data/>
```

3.9 Working with Binary Documents

This section covers the following topics:

- Types of Binary Documents
- Streaming Binary Content

3.9.1 Types of Binary Documents

This section provides a brief summary of binary document types. For details, see Working With Binary Documents in the *Application Developer's Guide*.

MarkLogic Server can store binary documents in three representations:

- Small binary documents are stored entirely in the database.
- Large binary documents are stored on disk with a small reference fragment in the database. The on-disk content is managed by MarkLogic Server.
- External binary documents are stored on disk with a small reference fragment in the database. However, the on-disk content is not managed by MarkLogic Server.

Small and large binary documents are created automatically for you, depending on the document size. External binary documents cannot be created using the REST Client API.

Large binary documents can be streamed out of the database using Range requests. For details, see "Streaming Binary Content" on page 140.

3.9.2 Streaming Binary Content

Streaming binary content out of the database avoids loading the entire document into memory. You can stream binary documents by sending GET requests to `/documents` that include range requests under following conditions:

- The size of the binary content returned is over the large binary size threshold. For details, see Working With Binary Documents in the *Application Developer's Guide*.
- The request is for content only. That is, no metadata is requested.
- The MIME type of the content is determinable from the Accept header or the document URI file extension.

The following example requests the first 500K of the binary document with URI `/binaries/large.jpg`:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -i -o piece.jpg -X GET \
    -H "Accept: application/jpg" -r "0-511999" \
    http://localhost:8000/LATEST/documents?uri=/binaries/large.jpg
...
HTTP/1.1 206 Binary Part Retrieved
Content-type: application/jpeg
Content-Range: bytes 0-511999/533817
Server: MarkLogic
Content-Length: 511999
Connection: close
```

3.10 Working with Temporal Documents

Most document write operations, such as PUT, POST, and PATCH on the `/documents` service enable you to work with temporal documents by exposing the following request parameters:

- `temporal-collection`: The URI of the temporal collection into which the new document should be inserted, or the name of the temporal collection that contains the document being updated.
- `temporal-document`: The "logical" URI of the document in the temporal collection; the temporal collection document URI. This is equivalent to the first parameter of the `temporal:statement-set-document-version-uri` XQuery function or of the `temporal.statementSetDocumentVersionUri` Server-Side JavaScript function.
- `source-document`: The temporal collection document URI of the document being operated on. This parameter is only applicable when updating existing documents. This parameter facilitates working with documents with user-maintained version URIs.
- `system-time`: The system start time for an update or insert.

To create temporal documents, you must use an endpoint that accepts a caller-specified URI. You cannot use `POST /v1/documents?extension={ext}` to create a temporal document because MarkLogic generates the document URI when you use this method.

During an update operation, if you do not specify `source-document` or `temporal-document`, then the `uri` request parameter indicates the source document. If you specify `temporal-document`, but do not specify `source-document`, the temporal-document URI identifies the source document.

The `uri` request parameter always refers to the output document URI. When the MarkLogic manages the version URIs, the document URI and temporal document collection URI have the same value. When the user manages version URIs, they can be different.

Use `POST:/v1/documents/protection` to protect a temporal document from operations such as update, delete, and wipe for a specified period of time. This method is equivalent to calling the `temporal:document-protect` XQuery function or the `temporal.documentProtect` Server-Side JavaScript function.

Use `POST:/v1/temporal/collections/{name}` to advance LSQT on a temporal collection. This method is equivalent to calling the `temporal:advance-lsqt` XQuery function or the `temporal.advanceLsqt` Server-Side JavaScript function.

For more details, see the *Temporal Developer's Guide* and specific methods in the *MarkLogic REST API Reference*.

3.11 Working with Metadata

The `/documents` service supports inserting, updating, and retrieving document metadata. Metadata is the properties, collections, permissions, quality and key-value metadata of content.

Metadata manipulation is most often exposed by the REST Client API through a `category` URL parameter. Metadata can be passed around as either XML or JSON, usually controlled through either an HTTP header or a `format` URL parameter. For specifics of a particular method, see the *MarkLogic REST API Reference*.

This section covers the following topics related to metadata storage and retrieval:

- Metadata Categories
- XML Metadata Format
- JSON Metadata Format
- Working With Document Properties Using JSON
- Disabling Metadata Merging

3.11.1 Metadata Categories

Where the REST Client API accepts specification of metadata categories, the following categories are recognized:

- `collections`
- `permissions`
- `properties`
- `quality`
- `metadata-values`

- `metadata`

The `metadata` category is shorthand for all the other categories. That is, `metadata` includes collections, permissions, properties, key-value metadata and quality.

Some requests also support a `content` category as a convenience for requesting or updating both metadata and document content in a single request.

The `metadata-values` category represents a "metadata field". This type of metadata is expressed as simple key-value pairs and you must configure a database field on a key before you can search it. While similar in some ways to document properties, this type of metadata is stored separately from the associated document and can be operated on like any other field. For more details, see Metadata Fields in the *Administrator's Guide*.

3.11.2 XML Metadata Format

Metadata contains information about document collections, permissions, properties, quality, and key-value metadata. The format is fully described by the schema file:

```
MARKLOGIC_INSTALL_DIR/Config/restapi.xsd
```

The following is a summary of the structure of the metadata. All elements are in the namespace `http://marklogic.com/rest-api`. You can have 0 or more `<collection/>`, `<permission/>`, `<metdata-values>`, `<metadata-value/>` or property child elements. There can be only one `<quality/>` element. The element name and contents of each document property element depends on the property.

```
<metadata xmlns="http://marklogic.com/rest-api">
  <collections>
    <collection>collection-name</collection>
  </collections>
  <permissions>
    <permission>
      <role-name>name</role-name>
      <capability>capability</capability>
    </permission>
  </permissions>
  <properties>
    <property-element/>
  </properties>
  <quality>integer</quality>
  <metadata-values>
    <metadata-value key="keyName">value</metadata-value>
  </metdata-values>
</metadata>
```

The following example shows a document in two collections, with one permission, two properties, and one key-value metadata item.

```
<rapi:metadata xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:collections>
```

```
    <rapi:collection>shapes</rapi:collection>
    <rapi:collection>squares</rapi:collection>
  </rapi:collections>
  <rapi:permissions>
    <rapi:permission>
      <rapi:role-name>hadoop-user-read</rapi:role-name>
      <rapi:capability>read</rapi:capability>
    </rapi:permission>
  </rapi:permissions>
  <prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
    <myprop>this is my prop</myprop>
    <myotherprop>this is my other prop</myotherprop>
  </prop:properties>
  <rapi:quality>0</rapi:quality>
  <rapi:metadata-values>
    <rapi:metadata-value key="level">high</rapi:metadata-value>
  </rapi:metadata-values>
</rapi:metadata>
```

3.11.3 JSON Metadata Format

Metadata contains information about document collections, permissions, properties, quality, and key-value metadata. A block of metadata can contain multiple collections, permissions and properties, but only 1 quality. The structure of each child of the `properties` JSON property depends on the property contents.

```
{
  "collections" : [ string ],
  "permissions" : [
    {
      "role-name" : string,
      "capabilities" : [ string ]
    }
  ],
  "properties" : {
    property-name : property-value
  },
  "quality" : integer
  "metadataValues": {
    key: value
  }
}
```

The following example shows a document in two collections, with one permission, two document properties, and two key-value metadata items.

```
{
  "collections": [
    "shapes",
    "squares"
  ],
  "permissions": [
```

```
      {
        "role-name": "hadoop-user-read",
        "capabilities": [
          "read"
        ]
      }
    ],
    "properties": {
      "myprop": "this is my prop",
      "myotherprop": "this is my other prop"
    },
    "quality": 0,
    "metadataValues": {
      "level": "high",
      "rating": 5
    }
  }
```

3.11.4 Working With Document Properties Using JSON

In most cases, your application should work with properties in a consistent format. That is, if you insert or update properties as XML, then you should retrieve them in XML. If you insert or update properties as JSON, you should retrieve them in JSON.

You should only insert document property data as JSON when you can specify the content type, such as through the request or part header of `POST:/v1/documents`. Do not use the `prop` request parameter of `PUT:/v1/documents` to insert document properties with JSON values as such input is untyped.

Internally, document properties are always stored as XML. When you insert a user-defined property using JSON, the XML representation is an XML element in the namespace `http://marklogic.com/xdmp/json/basic` with a local name that matches the property name in JSON. When you retrieve the property data as JSON, it is converted from XML back to JSON.

As long as you handle the data consistently, the conversion to and from XML is largely transparent to your application. However, you need to be aware of the XML representation when searching properties. Also, the implementation of transforms and extensions only sees the XML representation.

If you configure an index based on a user-defined property inserted using JSON, treat them as XML elements and use the `http://marklogic.com/xdmp/json/basic` in your configuration.

Protected system properties such as `last-updated` cannot be modified by your application. When retrieved as JSON, such protected properties are wrapped in an object with the JSON property name `$ml.prop`. For example:

```
{ "properties": {
    "$ml.prop": {
      "last-updated": "2013-11-06T10:01:11-08:00"
```

```
    }
} }
```

The following example code adds a document property expressed as JSON to `/my/doc.json`:

```
$ curl --anyauth --user user:password -X PUT -i \
  -H "Content-type: application/json" \
  -d '{"properties": {"pname":"pvalue"}}' \
  'http://localhost:8000/LATEST/documents?uri=/doc/my.json&category=pr
operties'
```

If you retrieve the properties of `/doc/my.json` as JSON, you get back what was inserted. The internal representation as XML is hidden from your application. For example:

```
$ curl --anyauth --user user:password -X GET \
  -H "Accept: application/json" \
  'http://localhost:8000/LATEST/documents?uri=/doc/my.json&category=pr
operties'

{"properties":{"pname":"pvalue"}}
```

However, if you retrieve the properties as XML or examine them using the Explore feature of Query Console, you will see the XML representation. For example:

```
$ curl --anyauth --user user:password -X GET \
  -H "Accept: application/xml" \
  'http://localhost:8000/LATEST/documents?uri=/doc/my.json&category=pr
operties'
<rapi:metadata uri="/doc/my.json" ...>
  <prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
    <pname type="string" xmlns="http://marklogic.com/xdmp/json/basic">
      pvalue
    </pname>
  </prop:properties>
</rapi:metadata>
```

3.11.5 Disabling Metadata Merging

If you use the REST Client API to ingest a large number of documents at one time and you find the performance unacceptable, you might see a small performance improvement by disabling metadata merging. This topic explains the tradeoff and how to disable metadata merging.

- When to Consider Disabling Metadata Merging
- Understanding Metadata Merging
- How to Disable Metadata Merging

3.11.5.1 When to Consider Disabling Metadata Merging

The performance gain from disabling metadata merging is modest, so you are unlikely to see significant performance improvement from it unless you ingest a large number of documents. You might see a performance gain under one of the following circumstances:

- Ingesting a large number of documents, one at a time.
- Updating a large number of documents that share the same metadata or that use the default metadata.

You cannot disable metadata merging in conjunction with update policies `version-optional` or `version-required`.

Metadata merging is disabled by default for multi-document write requests, as long as the request includes content for a given document. For details, see “Understanding When Metadata is Preserved or Replaced” on page 262.

3.11.5.2 Understanding Metadata Merging

Disabling metadata merging effectively eliminates the distinction between inserting a new document and updating an existing document, with respect to metadata handling.

Metadata merging is enabled by default. When you update metadata, the metadata in your update is merged with existing metadata. You can extend or update metadata without re-specifying unmodified document properties, collections, permissions, document quality, or key-value metadata. Similarly, when you update document content without including any metadata updates, the existing metadata is preserved.

When metadata merging is disabled, updating any metadata on a document overwrites all metadata for the document with the metadata provided in the request (plus default values for unspecified metadata categories). A content update that does not include any metadata resets the document metadata to the default values.

For example, if you set permissions on a document when you initially insert it into the database, and then later add the document to a collection while metadata merging is disabled, the permissions will be reset to default document permissions. In order to preserve the pre-existing permissions, you must specify them explicitly in your update.

3.11.5.3 How to Disable Metadata Merging

Metadata merging is controlled by the `update-policy` instance configuration property. The default value is `merge-metadata`.

To disable metadata merging, set `update-policy` to `overwrite-metadata` using the procedure described in “Configuring Instance Properties” on page 47. For example:

```
$ cat props.xml
<properties xmlns="http://marklogic.com/rest-api">
```

```
  <update-policy>overwrite-metadata</update-policy>
</properties>

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -d@"./props.xml" \
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/config/properties
```

The equivalent JSON input is shown below:

```
{
  "update-policy" : "overwrite-metadata"
}
```

4.0 Using and Configuring Query Features

This chapter covers the following topics:

- Query Feature Overview
- Querying Documents and Metadata
- Querying Lexicons and Range Indexes
- Using Query By Example to Prototype a Query
- Analyzing Lexicons and Range Indexes With Aggregate Functions
- Specifying Dynamic Query Options with Combined Query
- Querying Triples
- Invoking an Optic API Query Plan
- Searching Values Metadata Fields
- Configuring Query Options
- Using Namespace Bindings
- Generating Search Facets
- Paginating Results
- Customizing Search Results
- Generating Search Term Completion Suggestions

4.1 Query Feature Overview

The REST Client API includes several services for querying content in a MarkLogic Server database. The usage model for the query features is:

1. Optionally, use the `/config/query` service to install a set of named query options to apply to future queries. The name is used to apply the options to subsequent queries. For details, see “Configuring Query Options” on page 208. You can also specify options at query time for requests that accept a combined query; for details see “Specifying Dynamic Query Options with Combined Query” on page 179.

2. Optionally, use the `/config/namespaces` service to install namespace bindings for namespace aliases you will use in queries. For details, see “Using Namespace Bindings” on page 217.

3. Use `/search`, `/qbe`, or `/values` to perform a query.

    a. Depending on the service, the query might be expressed using a request parameter or in the request body. For details, see the reference documentation for each method.

b. Apply options by specifying the name of previously installed query options in the `options` request parameter and/or providing dynamic query options in the request body using a combined query. For details, see “Adding Query Options to a Request” on page 209.

c. Specify the content type (XML or JSON) of the query results using the format request parameter or Accept headers.

4. Query results of the requested content type are returned in the response body.

By default, a search operation returns search response data that summarizes the matches and can contain snippets of matching content. This chapter focuses on such operations. However, you can also use `/search` and `/qbe` to retrieve the matching documents or metadata instead of a search response; for details, see “Reading and Writing Multiple Documents” on page 249.

This chapter also covers lexicon and range-index query and analysis, and the configuration of persistent query options. The `/search` and `/values` services use query options to control and configure queries and search results. For details, see Appendix: Query Options Reference in the *Search Developer’s Guide*.

The following table gives a brief summary of each query related service.

Service	Description
`/search`	Use string, structured, cts and combined queries to search documents and metadata. Queries may be expressed in XML or JSON. For details, see “Querying Documents and Metadata” on page 151.
`/qbe`	Use Query By Example (QBE) for rapid prototyping of XML and JSON document searches. Queries using QBE syntax may be expressed in XML or JSON. For details, see “Using Query By Example to Prototype a Query” on page 169.

Service	Description
`/values`	Query lexicon and range index values and value co-occurrences; analyze lexicon and range index values and value-co-occurrences with builtin and user-defined aggregate functions. For details, see "Querying Lexicons and Range Indexes" on page 157.
`/rows`	Extract relational and semantic data as rows, using an Optic API plan. For details, see "Invoking an Optic API Query Plan" on page 189.
`/config/query`	Use XML or JSON to configure persistent query options for use with services such as `/search`, `/qbe`, and `/values`. For details, see "Configuring Query Options" on page 208.
`/config/namespaces`	NOTE: This service is deprecated. Use the REST Management API instead. Configure bindings between namespace prefixes and namespace URIs so you can use QNames in query contexts where it is not possible to dynamically specify a namespace. For details, see "Using Namespace Bindings" on page 217.

4.2 Querying Documents and Metadata

This section describes how to use the `/search` service to search documents and metadata.

- Constraining a Query by Collection or Directory
- Searching With String Queries
- Searching With Structured Queries
- Searching With cts:query
- Debugging /search Queries With Logging

You can use `/search` to retrieve a search response, matching documents and metadata, or both. The examples in this section only return a search response. To retrieve whole documents and/or their metadata, see "Reading Multiple Documents Matching a Query" on page 289.

If you need to retrieve search results across multiple requests that reflect the state of the database at a fixed point in time, see "Performing Point-in-Time Operations" on page 29.

4.2.1 Constraining a Query by Collection or Directory

Use the `collection` and `directory` request parameters to the `/search` service to limit search results to matches in specific collections or database directories.

You can specify multiple collections or directories. For example, the following URL finds documents containing “julius” in the “tragedy” and “comedy” collections:

```
http://localhost:8000/LATEST/search?q=julius&collection=tragedy&collection=comedy
```

You can use collection and directory constraints together. For example, by adding a directory to the above search, you can further limit matches to documents in the “/shakespeare/plays” directory:

```
http://localhost:8000/LATEST/search?q=julius&collection=tragedy&collection=comedy&directory=/shakespeare/plays
```

For details about collections and directories, see Collections in the *Search Developer’s Guide* and Directories in the *Application Developer’s Guide*.

4.2.2 Searching With String Queries

The MarkLogic Server Search API default search grammar allows you to quickly construct simple searches such as “cat”, “cat AND dog”, or “cat NEAR dog”. You can also customize the search grammar. For details, see Search Grammar in the *Search Developer’s Guide*.

To search for matches to a simple string query, send a GET or POST request to the `/search` service with a URL of the form:

```
http://host:port/version/search?q=query_string
```

Where *`query_string`* is a string conforming to the search grammar.

On a GET request, you can include both a string query and a structured query or cts:query in the same request using the `structuredQuery` request parameter.

On a POST request, put the structured or cts:query in the POST body, either standalone or as part of a combined query. The queries are AND’d together. For more information, see “Searching With Structured Queries” on page 153, “Searching With cts:query” on page 154, and “Specifying Dynamic Query Options with Combined Query” on page 179.

You can request search results in XML or JSON. Use the Accept header or `format` request parameter to select the output content type. For details, see “Controlling Input and Output Content Type” on page 31.

For details on the structure of search results, see Appendix: Query Options Reference in the *Search Developer’s Guide*.

For a complete list of parameters available with the `/search` service, see `GET:/v1/search` in the *REST Resources API*.

4.2.3 Searching With Structured Queries

Structured queries enable you to create complex queries, represented in XML or JSON. For details, see Searching Using Structured Queries in the *Search Developer's Guide*.

To search using a structured query, send a GET or POST request to the `/search` service. To pass the structured query as a request parameter, send a GET request with a URL of the form:

```
http://host:port/version/search?structuredQuery=query
```

Where *query* is an XML or JSON representation of a structured query.

To pass a structured query in the request body, send a POST request of the following form and place a structured or combined query in the POST body. Set the Content-type header appropriately:

```
http://host:port/version/search
```

If the request also includes a string query, the string query and structured query are AND'd together. For details, see "Searching With String Queries" on page 152 and "Specifying Dynamic Query Options with Combined Query" on page 179.

You can request search results in XML or JSON. Use the Accept header or `format` request parameter to select the output content type. For details, see "Controlling Input and Output Content Type" on page 31.

For a complete list of parameters available with the `/search` service, see `GET:/v1/search` or `POST:/v1/search` in the *REST Resources API*.

The following example combines a structured query equivalent to "Yorick NEAR Horatio" and a string query for "knew", requesting search results in JSON. The effect is equivalent to searching for "(Yorick NEAR Horatio) AND knew".

```
$ cat sq.xml
<search:query xmlns:search="http://marklogic.com/appservices/search">
  <search:near-query>
    <search:term-query>
      <search:text>Yorick</search:text>
    </search:term-query>
    <search:term-query>
      <search:text>Horatio</search:text>
    </search:term-query>
  </search:near-query>
</search:query>

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X POST -d@"./sq.xml" \
  -H 'Content-type: application/xml' \
  -H 'Accept: application/json' \
  'http://localhost:8000/LATEST/search?q=knew'
```

4.2.4 Searching With cts:query

In addition to querying the database with higher level search abstractions such as string query and structured query, you can also use a lower level cts:query, represented in XML or JSON. To learn more about cts:query, see Composing cts:query Expressions in the *Search Developer's Guide*.

To search using a cts:query, send a GET or POST request to the `/search` service. To pass the cts:query as a request parameter, send a GET request with a URL of the form:

```
http://host:port/version/search?structuredQuery=ctsquery
```

Where *ctsquery* is an XML or JSON representation of a cts:query.

To pass a cts:query in the request body, send a POST request of the following form and place the JSON or XML serialization of a cts:query in the POST body. Set the Content-type header appropriately.

```
http://host:port/version/search
```

For information on creating a serialized cts:query string in XML and JSON, see Serializations of cts:query Constructors in the *Search Developer's Guide*. Note that the JSON representation of a cts:query must have a `ctsquery` root JSON property:

```
{ "ctsquery": { serializedCtsQuery } }
```

If the request also includes a string query, the string query and cts:query are AND'd together. For details, see "Searching With String Queries" on page 152.

You can request search results in XML or JSON. Use the Accept header or `format` request parameter to select the output content type. For details, see "Controlling Input and Output Content Type" on page 31.

For a complete list of parameters available with the `/search` service, see `GET:/v1/search` or `POST:/v1/search` in the *REST Resources API*.

The following example combines a cts:query equivalent to "Yorick NEAR Horatio" and a string query for "knew", requesting search results in JSON. The effect is equivalent to searching for "(Yorick NEAR Horatio) AND knew".

```
$ cat cts_near.xml
<cts:near-query distance="10" xmlns:cts="http://marklogic.com/cts">
  <cts:word-query>
    <cts:text xml:lang="en">Yorick</cts:text>
  </cts:word-query>
  <cts:word-query>
    <cts:text xml:lang="en">Horatio</cts:text>
  </cts:word-query>
</cts:near-query>
```

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X POST -d@"./cts_near.xml" \
  -H 'Content-type: application/xml' \
  -H 'Accept: application/json' \
  'http://localhost:8000/LATEST/search?q=knew'
```

The following query is the equivalent `cts:near-query`, expressed in JSON.

```
{"ctsquery":{
  "nearQuery":{
    "queries":[
      {"wordQuery":{"text":["Yorick"], "options":["lang=en"]}},
      {"wordQuery":{"text":["Horatio"], "options":["lang=en"]}}
    ],
    "distance":10
  }}
}
```

4.2.5 Debugging /search Queries With Logging

When you enable the `debug` REST API instance property and make queries using `/search`, query details are sent to the MarkLogic Server error log. Enable this logging by setting the debug property to true, as described in “Configuring Instance Properties” on page 47.

When you enable debug logging, requests to `/search` log the following information helpful in debugging queries:

- EFFECTIVE OPTIONS. These are the query options in effect for your query. If the request does not include an `options` request parameter or a combined query, these are the default options.
- CTS-QUERY. Your query, expressed as a `cts:query` object. A `cts:query` object is the low level XML representation of any query expression. For details, see Understanding cts:query in the *Search Developer’s Guide*.
- SEARCH-QUERY. Your query, expressed as a `search:query` object. This information is only included if your request includes a structured query. A `search:query` is the XML representation of a structured query. For details, see Searching Using Structured Queries in the *Search Developer’s Guide*.

Examining the logging output can help you understand how MarkLogic Server sees your query. In the following example, notice that a string query for “Welles” is really a `cts:word-query` for the text string “Welles”. With the addition of a structured query in the request body to limit the query to occurrences of “Welles” to documents that also include “John” when it occurs near “Huston”, the logging output includes a `cts:query` and a `search:query` that reflect the result of combining the string query and structured query.

1. Enable debug logging for your REST API instance.

```
$ cat debug-on.xml
<properties xmlns="http://marklogic.com/rest-api">
  <debug>true</debug>
</properties>

$ curl --anyauth  --user user:password -X PUT \
    -d @./debug-on.xml -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/config/properties
```

2. Issue a query to `/search`.

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
    http://localhost:8000/LATEST/search?q="Welles"
...search matches returned
```

3. View the debug output in the MarkLogic Server error log, located in *MARKLOGIC_DIR*`/Logs/ErrorLog.txt`. There is no `search:query` in the log because this is a simple string query. (The timestamp and Info: header on each log line have been elided).

```
$ tail -15 /var/opt/MarkLogic/Logs/ErrorLog.txt
... Request environment:
... GET /v1/search?q=Douglas
... Rewritten to:
    /MarkLogic/rest-api/endpoints/search-list-query.xqy?q=Douglas
... ACCEPT */*
... PARAMS:
...   q: (Welles)
...
... Endpoint Details:
... EFFECTIVE OPTIONS:
... <options xmlns="http://marklogic.com/appservices/search">
...   <search-option>unfiltered</search-option>
...   <quality-weight>0</quality-weight>
... </options>
... CTS-QUERY:
... <cts:word-query qtextref="cts:text"
            xmlns:cts="http://marklogic.com/cts">
      <cts:text>Welles</cts:text>
    </cts:word-query>
```

4. Add a structured query to the search to constrain matches to documents that also contain “John” occurring near “Huston”.

```
$ cat ./structq.json
{"query":
  {"near-query":
    {"queries":[
      {"term-query":{"text":"John"}},
      {"term-query":{"text":"Huston"}}
    ]}
```

```
  }
}

$ curl --anyauth --user user:password -X POST -d@./structq.json \
    -H "Content-type: application/json" \
    http://localhost:8000/LATEST/search?q=Welles
...search results returned
```

5. View the logging output again and notice the inclusion of a search:query section in the log. Also, notice that the `cts:query` and the `search:query` represent the combined string and structured queries.

```
... Request environment:
... POST /v1/search?q=Welles
... Rewritten to:
    /MarkLogic/rest-api/endpoints/search-list-query.xqy?q=Welles
... ACCEPT */*
... PARAMS:
...   q: (Welles)
...
... Endpoint Details:
... EFFECTIVE OPTIONS:
... <options xmlns="http://marklogic.com/appservices/search">
...   <search-option>unfiltered</search-option>
...   <quality-weight>0</quality-weight>
... </options>
... CTS-QUERY:
... cts:and-query(
      (
        cts:word-query("Welles", ("lang=en"), 1),
        cts:near-query((
          cts:word-query("John", ("lang=en"), 1),
          cts:word-query("Huston", ("lang=en"), 1)),
        10, (), 1)
       ), ()
     )
... SEARCH-QUERY:
... <search:query
          xmlns:search="http://marklogic.com/appservices/search">
...   <search:qtext>Welles</search:qtext>
...   <search:near-query>
...     <search:term-query>
...       <search:text>John</search:text>
...     </search:term-query>
...     <search:term-query>
...       <search:text>Huston</search:text>
...     </search:term-query>
...   </search:near-query>
... </search:query>
```

4.3 Querying Lexicons and Range Indexes

The `/values` service supports the following operations:

- Query the values in a single lexicon or range index, or
- Find co-occurrences of values in multiple range indexes
- Analyze range index values or value co-occurrences using builtin or user-defined aggregate functions. For details, see "Analyzing Lexicons and Range Indexes With Aggregate Functions" on page 174.

For related search concepts, see Browsing With Lexicons in the *Search Developer's Guide*.

This section covers the following topics:

- Querying the Values in a Lexicon or Range Index
- Finding Value Co-Occurrences in Lexicons
- Using a Query to Constrain Results
- Identifying Lexicon and Range Index Values in Query Options
- Creating Indexes on JSON Properties
- Limiting the Number of Results

If you need to retrieve lexicon and range index data across multiple requests that reflect the state of the database at a fixed point in time, see "Performing Point-in-Time Operations" on page 29.

4.3.1 Querying the Values in a Lexicon or Range Index

Use the `/values/{name}` service to query the values in a lexicon or range index. Such queries must be supported by query options that include a `<values/>` element identifying the target lexicon or index; for details, see "Defining Queryable Lexicon or Range Index Values" on page 164.

To query the values in a lexicon or range index, use the `/values/{name}` service as follows:

1. Install query options or define dynamic query options that include a `values` option naming the target lexicon or index. For details, see "Adding Query Options to a Request" on page 209.

2. Send a GET or POST request to the `/values/{name}` service, where *`name`* is the name of a `values` definition in the query options from Step 1 or in the default query options. If you use persistent query options, include the `options` parameter and replace *`options_name`* with the name under which the query options are installed.

```
http://host:port/version/values/name?options=options_name
```

When constructing your request:

1. Substitute a named `values` specification for *`name`* in the URL. This must be a `values` range specification in the query options named by the `options` parameter, in the `options` portion of a combined query in the POST body, or in the default query options.

2. To use custom query options, specify them using the `options` parameter and/or the `options` portion of a combined query in the POST body. For details, see "Identifying Lexicon and Range Index Values in Query Options" on page 164 and "Adding Query Options to a Request" on page 209.

3. To use the default query options, omit the `options` parameter and `options` portion of a combined query. The default query options should include a range specification matching *`name`*.

4. To constrain the analysis to values in certain fragments, specify a query using the `q` and/or `structuredQuery` parameters, or a structured or combined query in the POST body. For details, see "Using a Query to Constrain Results" on page 163.

5. Specify the input (POST only) and output content type (XML or JSON) using the `format` parameter or the HTTP Content-type and Accept headers. For details, see "Controlling Input and Output Content Type" on page 31.

Additional request parameters are available. For details, see the *MarkLogic REST API Reference*.

The following example assumes the query options shown in the table below are installed under the name `index-options`:

Format	Query Options

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <values name="speaker">`
	`  <range type="xs:string">`
	`  <element ns="" name="SPEAKER"/>`
	`  </range>`
	`  </values>`
	`</options>`

JSON	`{`
	`  "options": {`
	`  "values": [`
	`  {`
	`  "name": "speaker",`
	`  "range": {`
	`  "type": "xs:string",`
	`  "element": { "ns": "", "name": "SPEAKER" }`
	`  }`
	`  }`
	`  ]`
	`  }`
	`}`

Then the example command below queries `/values/speaker` to retrieve the values of all SPEAKER elements:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    http://localhost:8000/LATEST/values/speaker?options=index-options
...
<values-response name="speaker" type="xs:string" \
    xmlns="http://marklogic.com/appservices/search" \
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <distinct-value frequency="1">[GOWER]</distinct-value>
  <distinct-value frequency="1">[PROSPERO]</distinct-value>
  ...
</values-response>
```

If you use the `format` parameter to request JSON output from the same request, the results are similar to the following:

```
{
  "values-response": {
    "name": "speaker",
    "type": "xs:string",
    "distinct-value": [
      {
        "frequency": 1,
        "_value": "[GOWER]"
      },
      {
        "frequency": 1,
        "_value": "[PROSPERO]"
      },
      ...
    ],
    "metrics": {
      "values-resolution-time": "PT0.016665S",
      "aggregate-resolution-time": "PT0.00001S",
      "total-time": "PT0.018102S"
    }
  }
}
```

If you add a string query and/or a structured or cts query to the request, you can limit the results to index values in matching fragments. For example, the following requests returns only the SPEAKER values in fragments containing “HAMLET”:

```
$ curl --anyauth --user user:password -X GET \
    'http://localhost:8000/LATEST/values/speaker?options=index-options
&q="HAMLET"'
```

4.3.2 Finding Value Co-Occurrences in Lexicons

A co-occurrence is a set of index or lexicon values occurring in the same document fragment. The REST Client API enables you to query for *n*-way co-occurrences. That is, tuples of values from multiple lexicons or indexes, occurring in the same fragment.

Use this procedure to query for co-occurrences of values in lexicons or range indexes with the `/values/{name}` service:

1. Specify query options that include a `tuples` specification for the target lexicons or indexes. For details, see "Defining Queryable Lexicon or Range Index Co-Occurrences" on page 166 and "Adding Query Options to a Request" on page 209.

2. Send a GET or POST request of the following form to the `/values/{name}` service, where *`name`* is the name of a `tuples` definition in the query options from Step 1. If you use persistent query options, include the `options` parameter and replace *`options_name`* with the name under which the query options are installed.

   ```
   http://host:port/version/values/name?options=options_name
   ```

When constructing your request:

1. Substitute a named `tuples` specification for *`name`* in the URL. This must be a `tuples` range specification in the query options named by the `options` parameter, in the `options` portion of a combined query in the POST body, or in the default query options if `options` is omitted.

2. To use custom query options, specify them using the `options` parameter and/or the `options` portion of a combined query in the POST body. For details, see "Identifying Lexicon and Range Index Values in Query Options" on page 164 and "Adding Query Options to a Request" on page 209.

3. To use the default query options, omit the `options` parameter and `options` portion of a combined query. The default query options should include a range specification matching *`name`*.

4. To constrain the analysis to values in certain fragments, specify a query using the `q` and/or `structuredQuery` parameters, or a structured, cts, or combined query in the POST body. For details, see "Using a Query to Constrain Results" on page 163.

5. Specify the input (POST only) and output content type (XML or JSON) using the `format` parameter or the HTTP Content-type and Accept headers. For details, see "Controlling Input and Output Content Type" on page 31.

Additional request parameters are available. For details, see the *MarkLogic REST API Reference*.

For more information about co-occurrences, see Value Co-Occurrences Lexicons in the *Search Developer's Guide*.

The following example assumes the query options shown in the table below are installed under the name `index-options`. Note that the options include a `<tuples/>` definition named "speaker-scene".

Format	Query Options

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <tuples name="speaker-scene">`
	`  <range type="xs:string">`
	`  <element ns="" name="SPEAKER"/>`
	`  </range>`
	`  <range type="xs:string">`
	`  <path-index>`
	`  /PLAY/ACT/SCENE/TITLE`
	`  </path-index>`
	`  </range>`
	`  </tuples>`
	`</options>`

JSON	`{`
	`  "options": {`
	`  "tuples": [`
	`  {`
	`  "name": "speaker-scene",`
	`  "range": [`
	`  {`
	`  "type": "xs:string",`
	`  "element": {`
	`  "ns": "",`
	`  "name": "SPEAKER"`
	`  }`
	`  },`
	`  {`
	`  "type": "xs:string",`
	`  "path-index": {`
	`  "text": "\/PLAY\/ACT\/SCENE\/TITLE"`
	`  }`
	`  }`
	`  ]`
	`  }`
	`  ]`
	`  }`
	`}`

Given these query options, this example queries `/values/speaker-scene` to retrieve co-occurrences of `SPEAKER` and scene titles:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
   http://localhost:8000/LATEST/values/speaker-scene?options=index-opt
ions
...
<values-response name="speaker-scene"
    xmlns="http://marklogic.com/appservices/search"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <tuple frequency="1">
    <distinct-value xsi:type="xs:string" ...>A Lord</distinct-value>
    <distinct-value ...>SCENE II.  The forest.</distinct-value>
  </tuple>
  ...
</values-response>
```

If you add a string query and/or a structured or cts query to the request, you can limit the results to co-occurrences in matching fragments. For example, the following requests returns only the `SPEAKER` values in fragments containing “HAMLET”:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    'http://localhost:8000/LATEST/values/speaker-scene?options=index-o
ptions&q="HAMLET"'
```

4.3.3 Using a Query to Constrain Results

You can use a string query, structured query, cts:query, or combined query with `/values/{name}` to limit results to fragments that match the query. The values must occur in fragments matching the query. The fragments are selected in the same manner as an “unfiltered” `cts:search`; for details, see Understanding Unfiltered Searches in the *Query Performance and Tuning Guide*.

If you use a string query, the query is treated as a word query; for details, see `cts:word-query`. Supply a string query using one of the following:

- The `q` request parameter of a POST or GET request
- The `qtext` XML element or JSON property of a combined query supplied in the body of a POST request.

Specify a structured query or cts:query in either XMl or JSON using one of the following:

- The `structuredQuery` request parameter on a GET request.
- In a combined query supplied in the body of a POST request.
- In the body of a POST request that does not use a combined query.

The following example limits the results to just those fragments containing the word “moon”:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
```

```
        'http://localhost:8000/LATEST/values/speaker?options=index-options
    &q=moon'
```

For the full example without a query constraint, see "Querying the Values in a Lexicon or Range Index" on page 158.

For details on query syntax, see the following sources:

- Searching Using String Queries in the *Search Developer's Guide*
- Searching Using Structured Queries in the *Search Developer's Guide*
- Composing cts:query Expressions in the *Search Developer's Guide*
- "Specifying Dynamic Query Options with Combined Query" on page 179

4.3.4 Identifying Lexicon and Range Index Values in Query Options

When you use the `/values/{name}` service, the in-scope query options must include a `values` or `tuples` option specification with the given name. Use the `values` option to make the values in a single lexicon or index available. Use the `tuples` option to make co-occurrences of values in multiple lexicons or indexes available.

This section covers the following topics:

- Defining Queryable Lexicon or Range Index Values
- Defining Queryable Lexicon or Range Index Co-Occurrences

4.3.4.1 Defining Queryable Lexicon or Range Index Values

Use this procedure to make the values in a lexicon or range index available through the `/values/{name}` service:

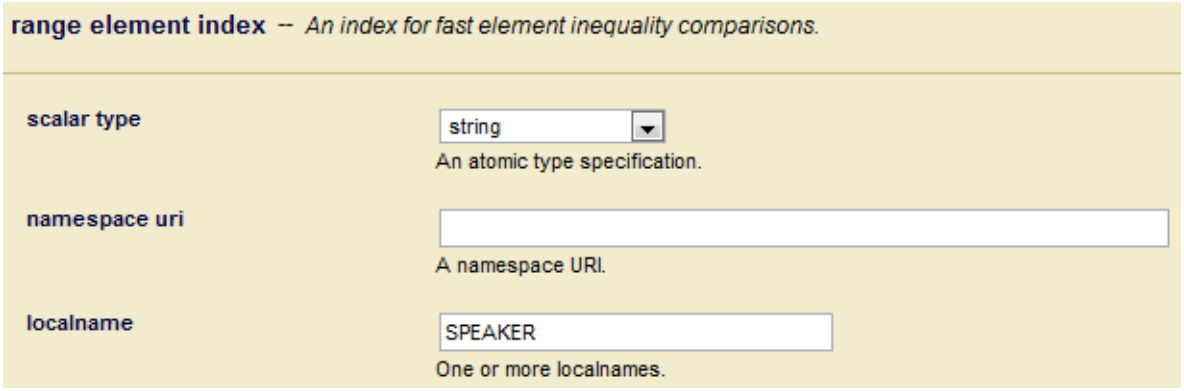
1. Create a lexicon or range index on the database, as described in Range Indexes and Lexicons in the *Administrator's Guide* and

2. Associate a name with the index or lexicon by defining a `<values/>` element (or JSON property) in query options.

3. Supply the query options from Step 2 in your request, as described in "Adding Query Options to a Request" on page 209.

For more information on lexicons and range indexes, see Browsing With Lexicons in the *Search Developer's Guide* and "Creating Indexes on JSON Properties" on page 168.

For example, if the database configuration includes an element range index on SPEAKER, such as the one shown below:



Then the following query options enable the `SPEAKER` element values to be referenced using as the resource `/values/speaker`:

Format	Query Options

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <values name="speaker">`
	`  <range type="xs:string">`
	`  <element ns="" name="SPEAKER"/>`
	`  </range>`
	`  </values>`
	`</options>`

JSON	`{`
	`  "options": {`
	`  "values": [`
	`  {`
	`  "name": "speaker",`
	`  "range": {`
	`  "type": "xs:string",`
	`  "element": { "ns": "", "name": "SPEAKER" }`
	`  }`
	`  }`
	`  ]`
	`  }`
	`}`

You can pre-install the options using the `/config/query/{name}` service and specify them in the name in the `options` request parameter, or you can supply the options as part of a combined query in the POST body.

The following example command installs the XML query options with the name “index-options”:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    -d@"./index-options.xml" -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/config/query/index-options
```

Now, you can query the `SPEAKER` index using the `/values/speaker` resource with the query options named “index-options”. For example:

```
$ curl --anyauth --user user:password -i -X GET \
   -H "Accept: application/xml" \
   'http://localhost:8000/LATEST/values/speaker?options=index-options'
```

Alternatively, use a POST request and specify the options in the request body using a combined query. For example:

```
$ cat > combo.xml
<search xmlns="http://marklogic.com/appservices/search">
  <options>
    <values name="speaker">
      <range type="xs:string">
        <element ns="" name="SPEAKER"/>
      </range>
    </values>
  </options>
</search>
^D

$ curl --anyauth --user user:password -i -X POST \
   -H "Content-type: application/xml" -H "Accept: application/xml" \
   -d @./combo.xml \
   'http://localhost:8000/LATEST/values/speaker'
```

4.3.4.2 Defining Queryable Lexicon or Range Index Co-Occurrences

A `<tuples/>` element in query options specifies the indexes to use in constructing *n*-way value co-occurrences. For more information about co-occurrences, see Value Co-Occurrences Lexicons in the *Search Developer’s Guide*.

Use this procedure to make co-occurrences of values in multiple lexicons or range indexes available through the `/values/{name}` service:

1. Create the lexicons or range indexes on the database, as described in Range Indexes and Lexicons in the *Administrator’s Guide*.

2. Associate a name with the lexicon/index tuple by defining a `<tuples>` element in query options (or a “tuples” object in JSON).

3. Supply the query options from Step 2 in your request, as described in “Adding Query Options to a Request” on page 209.

For example, suppose the database configuration contains a string-valued element range index on `<SPEAKER>` and a path range index on the XPath expression `/PLAY/ACT/SCENE/TITLE`. The following query options enable querying co-occurrrences of these two indexes under the name “speaker-scene”:

Format	Query Options

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <tuples name="speaker-scene">`
	`  <range type="xs:string">`
	`  <element ns="" name="SPEAKER"/>`
	`  </range>`
	`  <range type="xs:string">`
	`  <path-index>/PLAY/ACT/SCENE/TITLE</path-index>`
	`  </range>`
	`  </tuples>`
	`</options>`

JSON	`{`
	`  "options": {`
	`  "tuples": {`
	`  "name": "speaker-scene",`
	`  "range": {`
	`  "type": "xs:string",`
	`  "path-index": {`
	`  "text": "/PLAY/ACT/SCENE/TITLE",`
	`  }`
	`  }`
	`  }`
	`  }`
	`}`

You can pre-install the options using the `/config/query/{name}` service and specify them by name in the `options` request parameter, or you can supply the options as part of a combined query in the POST body.

The following example command installs the XML query options with the name “index-options”:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    -d@"./index-options.xml" -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/config/query/index-options
```

Now, you can query the SPEAKER index using the `/values/speaker-scene` resource with the query options named “index-options”. For example:

```
$ curl --anyauth --user user:password -i -X GET \
   -H "Accept: application/xml" \
   'http://localhost:8000/LATEST/values/speaker-scene?options=index-op
tions'
```

Alternatively, use a POST request and specify the options in the request body using a combined query. For example:

```
$ cat > combo.xml
<search xmlns="http://marklogic.com/appservices/search">
  <options>
    <tuples name="speaker-scene">
      <range type="xs:string">
        <element ns="" name="SPEAKER"/>
      </range>
      <range type="xs:string">
        <path-index>/PLAY/ACT/SCENE/TITLE</path-index>
      </range>
    </tuples>
  </options>
</search>
^D

$ curl --anyauth --user user:password -i -X POST \
   -H "Content-type: application/xml" -H "Accept: application/xml" \
   -d @./combo.xml \
   'http://localhost:8000/LATEST/values/speaker-scene'
```

4.3.5 Creating Indexes on JSON Properties

To efficiently search using JSON properties, you should define indexes on the properties. For example,a `json-property` structured query performs best when you define a range index on the property you’re querying. In addition, range queries require a backing index.

To create an index on a JSON property, treat the JSON property as an XML element for purposes of index creation. That is, use the interfaces for creating element index, such as an element range index.

For details, see Creating Indexes and Lexicons Over JSON Documents in the *Application Developer’s Guide*.

4.3.6 Limiting the Number of Results

You can use the `limit`, `start`, and `pageLength` request parameters to limit the number of values or co-occurrences returned by `GET:/v1/values` or `POST:/v1/values`.

The `limit` parameter specifies the maxiumum number of value to retrieve from a lexicon. Use `start` and `pageLength` to return results one page at a time, similar to the way you can page through results from the `/search` service. If `limit` is present, then `start` and `pageLength` are applied to the subset of values selected by `limit`, so the values on a page never extend beyond the values selected by `limit`.

For example, in the following request, at most 2 values or tuples are returned because `start` + `pageLength` would extend beyond the 5 values selected by `limit`.

```
GET /LATEST/values?limit=5&start=4&pageLength=3
```

If you specify a `start` value, you must also specify a `pageLength`. For a detailed discussion of how these parameters affect the results returned by values requests, see Returning Lexicon Values With search:values in the *Search Developer's Guide*.

4.4 Using Query By Example to Prototype a Query

This section describes how to search XML and JSON documents using a Query By Example (QBE). You cannot use QBE to search other document types or to search metadata.

This section covers the following topics:

- What is QBE
- Searching Documents With QBE
- Validating a QBE
- Generating a Combined Query from a QBE

4.4.1 What is QBE

A Query By Example (QBE) enables rapid prototyping of queries for "documents that look like this" using search criteria that resemble the structure of documents in your database.

For example, if your documents include an `author` element or key, you can use the following QBE to find documents with an `author` value of "Mark Twain".

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <author>Mark Twain</author>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{`
	`  "$query": { "author": "Mark Twain" }`
	`}`

You can only use QBE to search XML and JSON documents. Metadata search is not supported. You can search by element, element attribute, and JSON property; fields are not supported. For structural details, see Searching Using Query By Example in *Search Developer's Guide*.

When you're satisfied with your prototype or ready to use more powerful Search API features, you can use the API to convert a QBE into a combined query for use with the `/search` service.

The REST Client API includes the following support for QBE through the `/qbe` service:

- Search XML and JSON documents using a QBE.
- Validate the correctness of a QBE.
- Convert a QBE to a combined query for improved performance and full expressiveness.

4.4.2 Searching Documents With QBE

To search using QBE, send a GET or POST request to the `/qbe` service. To pass the QBE as a request parameter, send a GET request with a URL of the form:

```
http://host:port/version/qbe?query=your-qbe
```

Where *your-`qbe`* is an XML or JSON representation of a QBE.

To pass a QBE in the request body, send a POST request of the following form and place a QBE in the POST body. Set the Content-type header appropriately.

```
http://host:port/version/qbe
```

You can also create a multipart POST request that contains a QBE and query options in the request body. A request of this form enables you to specify dynamic query options with a QBE, similar to using a combined query with `POST:/v1/search` and enables you to specify the QBE and query options in different formats. When you use a multipart request body, the QBE must be the first part and the query options must be the second part. For details, see `POST:/v1/qbe`.

You can request search results in XML or JSON. Use the Accept header or `format` request parameter to select the output content type. For details, see "Controlling Input and Output Content Type" on page 31.

You can validate the correctness of your input QBE as part of your search or as a standalone operation. For details, see "Validating a QBE" on page 172.

For a complete list of parameters available with the `/qbe` service, see `GET:/v1/qbe` or `POST:/v1/qbe` in the *REST Resources API*.

The following example matches XML documents that have an author element value of "Mark Twain". Results are returned as XML.

```
$ cat qbe.xml
<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">
  <q:query>
    <author>Mark Twain</author>
  </q:query>
</q:qbe>

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X POST -d@"./qbe.xml" \
  -H 'Content-type: application/xml' \
  'http://localhost:8000/LATEST/qbe'
```

The following example shows an equivalent search using JSON.

```
$ cat qbe.sjon
{"$query": {
    "author": "Mark Twain"
} }

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X POST -d@"./qbe.json" \
  -H 'Content-type: application/json' \
  -H 'Accept: application/json' \
  'http://localhost:8000/LATEST/qbe
```

The `/qbe` service supports most of the same features as the `/search` service, such as using pre-installed persistent query options, result pagination, and search result transformations. For details, see `GET:/v1/qbe` or `POST:/v1/qbe` in the *REST Resources API*.

You can also use `/qbe` to retrieve matching documents and metadata. The examples in this section only return a search response. To retrieve whole documents and/or their metadata, see "Reading Multiple Documents Matching a Query" on page 289.

4.4.3 Validating a QBE

When you perform a search, MarkLogic Server does not verify the correctnesss of your QBE. If your QBE is syntactically or semantically incorrect, you might get errors or surprising results. To avoid such issues, you can validate your QBE prior to or as part of a search.

To validate your query as a standalone operation, add the request parameter `view=validate` to a GET or POST request to the `/qbe` service. Rather than performing a search, MarkLogic Server checks your QBE for correctness and returns an indication of validity returned.

For example, the following command validates a JSON QBE:

```
$ curl --anyauth --user user:password -i -X POST \
    -H "Content-type: application/json" -d @./qbe.json \
    'http://localhost:8000/LATEST/qbe?view=validate'
```

If your query is valid, MarkLogic Server responds with status 200 (OK) and the response body contains a `valid-query` element, similar to the following:

```
<q:valid-query
  xmlns:q="http://marklogic.com/appservices/querybyexample"/>
```

If your query is invalid, MarkLogic Server can respond with either a status 200 (OK) or status 400 (Bad Request), depending on the nature of the error. When the status code is 400, the response body contains an error. When the status code is 200, the response body contains an `invalid-query` element that encapsulates the reason validation failed. For example:

```
<q:invalid-query
    xmlns:q="http://marklogic.com/appservices/querybyexample">
  <q:report id="QBE-QUERY">Query can only contain a filtered flag,
score configuration,  composers, word queries, and criteria</q:report>
</q:invalid-query>
```

To validate a query as part of your search, use the request parameters `view=validate` in conjunction with `view=results`. If your query is valid, the search proceeds as usual. If you query is not valid, an error report is returned.

```
$ curl --anyauth --user user:password -i -X POST \
    -H "Content-type: application/json" -d @./qbe.json \
    'http://localhost:8000/LATEST/qbe?view=validate&view=results'
```

4.4.4 Generating a Combined Query from a QBE

Generating a combined query from a QBE has the following potential benefits:

- Improve search performance.
- Access a wider array of search features.
- Debug your QBE by examining the lower level Search API constructs it generates.

To generate a combined query from a QBE, add the `view=structured` request parameter to a GET or POST request to the `/qbe` service. Rather than performing a search, the request returns a combined query in the response. You can use the resulting query with the `/search` service.

You cannot combine `view=structured` with other `view` settings, such as `validate` or `results`.

The following command generates a combined query from a QBE:

```
$ cat qbe.xml
<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">
  <q:query>
    <author>Mark Twain</author>
  </q:query>
</q:qbe>

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X POST -d@"./qbe.xml" \
  -H 'Content-type: application/xml' \
  'http://localhost:8000/LATEST/qbe?view=structured'

HTTP/1.1 200 OK
...
<search:search xmlns:search="http://marklogic.com/appservices/search">
  <search:query>
    <search:value-query>
      <search:element ns="" name="author"/>
      <search:text>Mark Twain</search:text>
      <search:term-option>exact</search:term-option>
    </search:value-query>
  </search:query>
  <search:options>
    <search:search-option>unfiltered</search:search-option>
    <search:quality-weight>0</search:quality-weight>
    <search:result-decorator apply="href-decorator"
      ns="http://marklogic.com/rest-api/lib/href-decorator"
      at="/MarkLogic/rest-api/lib/rest-result-decorator.xqy"/>
  </search:options>
</search:search>
```

For more details, see “Searching Using Structured Queries” on page 83 in *Search Developer’s Guide* and “Specifying Dynamic Query Options with Combined Query” on page 179.

4.5 Analyzing Lexicons and Range Indexes With Aggregate Functions

This section covers the following topics:

- Aggregate Function Overview
- Using Query Options to Apply Aggregate Functions
- Using Request Parameters to Apply Aggregate Functions
- Example: Applying a Builtin Aggregate Function
- Example: Applying an Aggregate UDF

4.5.1 Aggregate Function Overview

An aggregate function performs an operation over values or value co-occurrences in lexicons and range indexes. For example, you can use an aggregate function to compute the sum of values in a range index.

There are two kinds of aggregate functions, builtin and user-defined. MarkLogic Server provides builtin aggregate functions for several common analytical functions; see the list in Using Builtin Aggregate Functions in the *Search Developer's Guide*.

In addition, you can also implement aggregate user-defined functions (UDFs) in C++ and deploy them as native plugins. Aggregate UDFs must be installed before you can use them. For details, see Implementing an Aggregate User-Defined Function in the *Application Developer's Guide*.

You can use the REST Client API to apply aggregate functions using `/values/{name}` in two ways:

- Include one or more `<aggregate/>` elements (XML) or sub-objects (JSON) in a `<values/>` or `<tuples/>` range specification in the query options.
- Include one or more `aggregate` request parameters.

If aggregate functions are specified through both query options and request parameters, the request parameter(s) overrides the aggregates specified in the query options.

Note: You can only specify multiple aggregate UDFs from more than one plugin using query options.

Note: You cannot use the REST Client API to apply aggregate UDFs that require additional parameters.

4.5.2 Using Query Options to Apply Aggregate Functions

To specify an aggregate function in query options, include an `<aggregate/>` element in a `<values/>` or `<tuples/>` range specification. If you include multiple `<aggregate/>` specifications, MarkLogic Server applies all the functions.

For a builtin aggregate, specify the function name in the "apply" attribute of an `<aggregate/>` element. For example, the query options below specify the builtin aggregate "count", which is equivalent to the XQuery builtin `cts:count-aggregate`.

Format	Query Options

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <values name="speaker">`
	`  <range type="xs:string">`
	`  <element ns="" name="SPEAKER"/>`
	`  </range>`
	`  **<aggregate apply="count"/>**`
	`  </values>`
	`</options>`

JSON	`{`
	`  "options": {`
	`  "values": [`
	`  {`
	`  "name": "speaker",`
	`  "range": {`
	`  "type": "xs:string",`
	`  "element": { "ns": "", "name": "SPEAKER" }`
	`  },`
	`  **"aggregate": { "apply":"count" }**`
	`  }`
	`  ]`
	`  }`
	`}`

An aggregate UDF is identified by the function name and a relative path to the plugin that implements the aggregate, as described in Using Aggregate User-Defined Functions in the *Search Developer's Guide*. Specify the function name with the "apply" attribute and the plugin path with the "udf" attribute in an `<aggregate/>` element or object. For example, the following query options specify a native UDF called "count" provided by a plugin installed under "native/sampleplugin":

Format	Query Options
XML	<options xmlns="http://marklogic.com/appservices/search"> <values name="speaker"> <range type="xs:string"> <element ns="" name="SPEAKER"/> </range> **<aggregate apply="count" udf="native/sampleplugin" />** </values> </options>

JSON	{
	  "options": {
	  "values": [
	  {
	  "name": "speaker",
	  "range": {
	  "type": "xs:string",
	  "element": { "ns": "", "name": "SPEAKER" }
	  },
	  "aggregate": {
	  "apply":"count",
	  "udf":"native/sampleplugin"
	  }
	  }
	  ]
	  }
	}

To use query options to apply an aggregate function:

1. Define query options that include one or more aggregate definitions, as shown above.

2. Supply the query options from Step 1 in your request, as described in"Adding Query Options to a Request" on page 209.

3. Apply the aggregate by sending a GET or POST request to `/values/{name}` and including the options from Step 2. For example:
   `GET /LATEST/values/speaker?options=index-options`.

For details on using query options, see "Configuring Query Options" on page 208.

4.5.3 Using Request Parameters to Apply Aggregate Functions

To analyze lexicon or index values or co-occurrences with builtin aggregate function, make a GET or POST request to the `/values/{name}` service with a URL of the form:

```
http://host:port/version/values/name?aggregate=aggr_name&options=optio
ns_name
```

To analyze lexicon or index values or co-occurrences with a previously installed aggregate UDF, make a GET request to the `/values/{name}` service with a URL of the form:

```
http://host:port/version/values/name?aggregate=aggr_name&aggregatePath
=aggr_path&options=options_name
```

When constructing the request:

1. Substitute a named range specification for *`name`* in the URL. This must be a range specification (`<values/>` or `<tuples/>`) in the query options supplied in the request, or in the default query options if options are omitted.

2. To use custom query options, specify them using the `options` parameter and/or the `options` portion of a combined query in the POST body. For details, see "Adding Query Options to a Request" on page 209.

3. To use the default query options, omit the `options` parameter and `options` portion of a combined query. The default query options should include a range specification matching *`name`*.

4. Specify the aggregate function name using the `aggregate` parameter.

   The name must be one of the builtin aggregate functions listed in Using Builtin Aggregate Functions the *Search Developer's Guide*, or a function implemented by the plugin identified by `aggregatePath`.

5. If you're applying an aggregate UDF, specify the relative path to the plugin implementing the aggregate function using the `aggregatePath` parameter.

6. To constrain the analysis to values in certain fragments, specify a query using the `q` and/or `structuredQuery` parameters, or a structured or combined query in the POST body. For details, see "Using a Query to Constrain Results" on page 163.

7. Specify the result content type (XML or JSON) using the `format` parameter or the HTTP Accept headers. The default content type is XML. For details, see "Controlling Input and Output Content Type" on page 31.

8. If you only want the aggregate value in the results, set the `view` parameter to `aggregate`. By default, MarkLogic Server returns both the lexicon or index values and the aggregate result.

Additional request parameters are available. For details see the *MarkLogic REST API Reference*.

When applying an aggregate UDF, the output is dependent on the UDF. Aggregate UDFs return a sequence of items, which can be atomic values or key-value maps. For details, see Aggregate User-Defined Functions in the *Application Developer's Guide* and the XQuery builtin function `cts:aggregate`.

4.5.4 Example: Applying a Builtin Aggregate Function

The following example counts the number of values in the index identified as "speaker" in "index-options" (`options=index-options`). The counted values include only those in fragments containing "HAMLET" (`q=HAMLET`). The output should contain only the aggregate result (`view=aggregate`).

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    'http://localhost:8000/LATEST/values/speaker?options=index-options
&aggregate=count&view=aggregate&q=HAMLET'
...
<values-response name="speaker" type="xs:string"
    xmlns="http://marklogic.com/appservices/search"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <aggregate-result name="count">92</aggregate-result>
  <metrics>
    <aggregate-resolution-time>PT0.001108S</aggregate-resolution-time>
    <total-time>PT0.003719S</total-time>
  </metrics>
</values-response>
```

4.5.5 Example: Applying an Aggregate UDF

This example demonstrates using an aggregate user-defined function to count the number of values in an element range index using `/values/{name}`. The aggregate UDF is specified via request parameters, as described in "Using Request Parameters to Apply Aggregate Functions" on page 176.

This example assumes the following pre-requisites are already met:

- A native plugin is installed with the path "native/sampleplugin".
- The plugin implements a "count" function that counts the values in a range index. That is, a function equivalent to the "count" builtin aggregate function.
- Query options are installed with the name "index-options".
- The query options include a range specification named "speaker" for the SPEAKER element range index.

The following command uses the "count" aggregate UDF to count the number of values in the SPEAKER index in fragments containing "HAMLET". The output only contains the aggregate result (`view=aggregate`).

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    'http://localhost:8000/LATEST/values/speaker?options=index-options
&aggregate=count&aggregatePath=native/sampleplugin&view=aggregate&q=HA
MLET'
```

The use of `view=aggregate` limits the output to only the aggregate results, as shown below. XML is the default output format. Use the `format` parameter or the HTTP Accept headers to request JSON output.

Format	Example Output

XML	`<values-response name="speaker" type="xs:string"`
	`  xmlns="http://marklogic.com/appservices/search"`
	`  xmlns:xs="http://www.w3.org/2001/XMLSchema">`
	`  **<aggregate-result name="count">92</aggregate-result>**`
	`  <metrics>`
	
	`<aggregate-resolution-time>PT0.001514S</aggregate-resolution-time>`
	`  <total-time>PT0.004049S</total-time>`
	`  </metrics>`
	`</values-response>`

JSON	`{`
	`  "values-response": {`
	`  "name": "speaker",`
	`  "type": "xs:string",`
	`  **"aggregate-result": [**`
	`  **{**`
	`  **"name": "count",**`
	`  **"_value": "92"**`
	`  **}**`
	`  ],`
	`  "metrics": {`
	`  "aggregate-resolution-time": "PT0.001336S",`
	`  "total-time": "PT0.004104S"`
	`  }`
	`  }`
	`}`

4.6 Specifying Dynamic Query Options with Combined Query

A *combined query* is an XML or JSON wrapper around a string and/or a structured query, cts query or QBE, and query options. Use a combined query to specify query options at runtime without first persisting the options as named options using the `/config/query` service. Combined queries are useful for rapid prototying during development, and for applications that need to modify query options on a per query basis.

To use a combined query, send a POST request to `/v1/search` or `/v1/values/{name}` with the combined query in the request body. See the following topics for more details:

- Syntax and Semantics
- Interaction with Queries in Request Parameters
- Interaction with Persistent Query Options
- Performance Considerations
- Combined Query Examples

4.6.1 Syntax and Semantics

A combined query can contain a string query, a structured query, a QBE, a cts query, query options, or a combination of these. For example, you can create a combined query that contains only query options, only a structured query, or a structured query, string query, and query options.

The following table shows the structure of a combined query. See the usage notes after the table for more details.

Format	Combined Query Template

XML	`<search xmlns="http://marklogic.com/appservices/search">`
	`  <!-- any serialized cts:query -->`
	`  <query>`
	`  <!-- structured query, same syntax as standalone -->`
	`  </query>`
	`  <qbe:query xmlns:qbe="http:marklogic.com/appservices/querybyexample">`
	`  <!-- the query portion of a Query By Example -->`
	`  </qbe:query>`
	`  <qtext>`*your string query*`</qtext>`
	`  <sparql>`*your SPARQL query*`</sparql>`
	`  <options>`
	`  <!-- same syntax as standalone query options -->`
	`  </options>`
	`</search>`

JSON	`{"search": {`
	`  "ctsquery": "`*any serialized cts query*`",`
	`  "query": { `*structured query, same syntax as standalone*`},`
	`  "$query": {`*query portion of a QBE, same syntax as standalone*`},`
	`  "ctsquery": { `*cts.query serialized as JSON*` }`
	`  "qtext": "`*your string query here*`",`
	`  "sparql": "`*your SPARQL query here*`",`
	`  "options": { `*same syntax as standalone query options*` },`
	`} }`

You should be aware of the following usage notes:

- Within the combined query wrapper, the queries and options use the same syntax as when they occur standalone.
- You can include at most one of a structured query, a cts query, or a QBE. You can combine this query with a string query (`qtext` element or JSON property), in which case the two queries are AND'd together.
- You can include a SPARQL query in a combined query if and only if you are using the combined query with `POST:/v1/graphs/sparql` to perform a semantic query. In this case, the other portions of the combined query are used used to perform a search that further constrains the result of the SPARQL query. For details, see `POST:/v1/graphs/sparql`.
- Not all query options are applicable to all query types. For example, if you define a range constraint in the options, it is only usable as part of a string or structured query.
- When you use a QBE, you include only the `qbe:query` XML element or `$query` JSON property. This means the `response` and `format` portions of a QBE are not available. You can use query options to express the equivalent of the `response` customizations, but there is no equivalent to `format`. If you need to use `format`, use a standalone QBE or express your query using a cts:query or structured query.

For examples, see "Combined Query Examples" on page 184.

For details on sub-query and query options syntax, see the following sections of the *Search Developer's Guide*:

- Searching Using String Queries
- Searching Using Structured Queries
- Searching Using Query By Example
- Composing cts:query Expressions
- Search Customization Using Query Options

4.6.2 Interaction with Queries in Request Parameters

When making a `POST:/v1/search` or `POST:/v1/values/{name}` request, you can use a combined query in conjunction with the `q` request parameter. The string query in the request parameter value is AND'd with the sub-query(s) in the combined query.

The following table summarizes the interaction between the `q` request parameter and the sub-queries of a combined query. Note that the "queries" in the table are an abstraction rather than actual example queries.

Request Parameter	Combined Query	Final Query
q=*query-rp*	<search> <query>*query-cq*</query> </search>	*query-rp* AND *query-cq*
q=*query-rp*	<search> <qtext>*query-cq*</qtext> </search>	*query-rp* AND *query-cq*
q=*query-rp*	<search> <query>*query-cq1*</query> <qtext>*query-cq2*</qtext> </search>	*query-rp* AND *query-cq1* AND *query-cq2*
none	<search> <query>*query-cq1*</query> <qtext>*query-cq2*</qtext> </search>	*query-cq1* AND *query-cq2*

4.6.3 Interaction with Persistent Query Options

Dynamic query options supplied in a combined query are merged with persistent and default options that are in effect for the search. If the same non-constraint option is specified in both the combined query and persistent options, the setting in the combined query takes precedence.

When a combined search supplies query options, persisted options are only merged with request options if the `options` parameter specifies the options by name. Default persisted options are not merged.

Constraints are overridden by name. That is, if the dynamic and persistent options contain a `<constraint/>` element with the same `@name`, the definition in the dynamic query options is the one that applies to the query. Two constraints with different name are both merged into the final options.

```
<options xmlns="http://marklogic.com/appservices/search">
  <fragment-scope>properties</fragment-scope>
  <return-metrics>false</return-metrics>
  <constraint name="same">
    <collection prefix="http://server.com/persistent/"/>
  </constraint>
  <constraint name="not-same">
    <element-query name="title" ns="http://my/namespace" />
  </constraint>
</options>
```

Further, suppose you submit a `POST:/v1/search` request that uses `my-options` and includes the following query options in a combined query in the request body:

```
$ cat body.xml
<search xmlns="http://marklogic.com/appservices/search">
  <options>
    <return-metrics>true</return-metrics>
    <debug>true</debug>
  <constraint name="same">
    <collection prefix="http://server.com/dynamic/"/>
  </constraint>
    <constraint name="different">
      <element-query name="scene" ns="http://my/namespace" />
    </constraint>
  </options>
</search>

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X POST \
    -d@./body.xml -H "content-type: application/xml" \
    'http://localhost:8000/LATEST/search?q=TRAGEDY&options=my-options'
```

The query is evaluated with the following merged options. The persistent options contribute the `fragment-scope` option and the constraint named `not-same`. The dynamic options in the combined query contribute the `return-metrics` and `debug` options and the constraints named `same` and `different`. The `return-metrics` setting and the constraint named `same` from `my-options` are discarded.

```
<options xmlns="http://marklogic.com/appservices/search">
  <fragment-scope>properties</fragment-scope>
  <return-metrics>true</return-metrics>
  <debug>true</debug>
  <constraint name="same">
    <collection prefix="http://server.com/dynamic/"/>
  </constraint>
    <constraint name="different">
      <element-query name="scene" ns="http://my/namespace" />
    </constraint>
  </options>
  <constraint name="not-same">
    <element-query name="title" ns="http://my/namespace" />
  </constraint>
</options>
```

4.6.4 Performance Considerations

Using persistent query options usually performs better than using a combined query. In most cases, the difference between the two approaches is slight.

When MarkLogic Server processes a combined query, the per request query options must be parsed and merged with named and default options on every search. When you only use persistent named or default query options, you reduce this overhead.

If your application does not require dynamic per-request query options, you should use the `/config/query/{name}` service to persist your options under a name and use the `options` request parameter to associate the options with a simple string, structured, or values query. For details, see "Configuring Query Options" on page 208.

4.6.5 Combined Query Examples

This section includes the following examples:

- Example: Overriding Persistent Constraints
- Example: Modifying the Search Response
- Example: Including a cts Query in a Combined Query
- Example: Including a QBE in a Combined Query

4.6.5.1 Example: Overriding Persistent Constraints

The following example uses a bucketed constraint, backed by an element range index, to group items into facets by price. Assume following options that define price range buckets are stored as persistent options using the `/config/query/{name}` service, as described in "Creating or Modifying Query Options" on page 210.

```
<options xmlns="http://marklogic.com/appservices/search">
 <constraint name="price" facet="true">
    <range type="xs:int">
    <element ns="" name="price"/>
    <bucket name="under50" ge="0" lt="50">under $50</bucket>
    <bucket name="under100" ge="50" lt="101">$50-$100</bucket>
    <bucket name="over100" ge="101">over $100</bucket>
  </range>
 </constraint>
</options>
```

The application can use these persistent options to generate a faceted navigation page that allows users to browse by price. You can use dynamic query options to render a page that includes a custom facet from a price range entered by the user. The resulting combined query might look like the following if the user defined a price range of $100-150:

```
<search xmlns="http://marklogic.com/appservices/search">
  <options>
    <constraint name="price" facet="true">
      <range type="xs:int">
        <element ns="" name="price"/>
        <bucket name="under50" ge="0" lt="50">under $50</bucket>
        <bucket name="under100" ge="50" lt="101">$50-$100</bucket>
```

```
        <bucket name="over100" ge="101">over $100</bucket>
        <bucket name="custom" ge="100" lt="151">$100 to $150</bucket>
      </range>
    </constraint>
  </options>
  <query>
    <range-constraint-query>
      <constraint-name>price</constraint-name>
      <value>custom</value>
    </range-constraint-query>
  <query>
</search>
```

4.6.5.2 Example: Modifying the Search Response

The following example uses a combined query that contains only query options to enable the `return-query` option on the fly to explore how a string query is represented as a `cts:query` after parsing. The `return-query` setting in the dynamic options overrides any `return-query` setting in the default options. All other settings in the default options are unchanged and apply to the query evaluation.

```
$ cat combo-query.xml
<search xmlns="http://marklogic.com/appservices/search">
  <options>
    <return-query>true</return-query>
  </options>
</search>

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X POST \
    -d@./combo-query.xml -H "content-type: application/xml" \
    'http://localhost:8000/LATEST/search?q=Horatio NEAR Yorick'

<search:response snippet-format="snippet" total="0" start="1" ...>
  <search:qtext>Horation NEAR Yorick</search:qtext>
  <search:query>
    <cts:near-query qtextjoin="NEAR" strength="30" ...>
      <cts:word-query qtextref="cts:text">
        <cts:text>Horatio</cts:text>
      </cts:word-query>
      <cts:word-query qtextref="cts:text">
        <cts:text>Yorick</cts:text>
      </cts:word-query>
    </cts:near-query>
  </search:query>
  ...
</search:response>
```

For details, see “Interaction with Persistent Query Options” on page 182.

4.6.5.3 Example: Including a cts Query in a Combined Query

The following example illustrates how to use a serialized cts query in a combined query. The query is a word query on the term “henry” where it appears in a TITLE XML element or JSON property.

Format	Example

XML	`<search:search xmlns:search="http://marklogic.com/appservices/search">`
	`  <cts:element-word-query xmlns:cts="http://marklogic.com/cts">`
	`  <cts:element>TITLE</cts:element>`
	`  <cts:text xml:lang="en">henry</cts:text>`
	`  </cts:element-word-query>`
	`  <search:options>`
	`  <search:extract-document-data>`
	`  <search:extract-path>/PLAY/TITLE</search:extract-path>`
	`  </search:extract-document-data>`
	`  <search:transform-results apply="empty-snippet"/>`
	`  <search:search-option>filtered</search:search-option>`
	`  </search:options>`
	`</search:search>`

JSON	`{"search" : {`
	`  "ctsquery": {`
	`  "elementWordQuery":{`
	`  "element":["TITLE"],`
	`  "text":["henry"],`
	`  "options":["lang=en"]`
	`  }},`
	`  "options": {`
	`  "extract-document-data": {`
	`  "extract-path": "/PLAY/TITLE"`
	`  },`
	`  "transform-results": {`
	`  "apply": "empty-snippet"`
	`  },`
	`  "search-option": ["filtered"]`
	`  }`
	`} }`

The options in the combined query do the following:

- Suppress the generation of snippets (`transform-results`)
- Extract just the title from the matched documents (`extract-document-data`)
- Force a filtered search

For more information about serializing cts queries, see Serializations of cts:query Constructors in the *Search Developer’s Guide*.

4.6.5.4 Example: Including a QBE in a Combined Query

The following example illustrated using a QBE in a combined query. The XML QBE is a word query on the term “henry” when it appears in a TITLE element. The JSON QBE is a word query on the term “henry” when it appears in a JSON property named “TITLE”.

Format	Example

XML	`<search:search xmlns:search="http://marklogic.com/appservices/search">`
	`  <qbe:query  xmlns:qbe="http://marklogic.com/appservices/querybyexample">`
	`    <TITLE><qbe:word>henry</qbe:word></TITLE>`
	`  </qbe:query>`
	`  <search:options>`
	`    <search:extract-document-data>`
	`      <search:extract-path>/PLAY/TITLE</search:extract-path>`
	`    </search:extract-document-data>`
	`    <search:transform-results apply="empty-snippet"/>`
	`    <search:search-option>filtered</search:search-option>`
	`  </search:options>`
	`</search:search>`

JSON	`{"search" : {`
	`  "$query": {`
	`    "TITLE" : { "$word": "henry" }`
	`  },`
	`  "options": {`
	`    "extract-document-data": {`
	`      "extract-path": "/PLAY/TITLE"`
	`    },`
	`    "transform-results": {`
	`      "apply": "empty-snippet"`
	`    },`
	`    "search-option": ["filtered"]`
	`  }`
	`} }`

The options in the combined query do the following:

- Suppress the generation of snippets (`transform-results`)
- Extract just the title from the matched documents (`extract-document-data`)
- Force a filtered search

Note that options such as `extract-document-data` and `transform-results` take the place of response customizations available to a standalone QBE in the `qbe:response` XML element or `$response` JSON property. Since only the query portion of a QBE can be included in a combined query, you must use query options to achieve equivalent results.

Recall that an XML QBE matches only XML documents and a JSON QBE matches only JSON documents by default. With a standalone QBE, you can override this behavior using the `qbe:format` XML element or `$format` JSON property, but this is not available when using QBE in a combined query. If you need to use this feature, use a standalone QBE.

You can include a string query in the combined query along with your QBE. The two queries are AND'd together in this case. The following example demonstrates a combined query that includes both a string query and a QBE:

Format	Example

XML	`<search:search xmlns:search="http://marklogic.com/appservices/search">`
	`  <qbe:query  xmlns:qbe="http://marklogic.com/appservices/querybyexample">`
	`  <TITLE><qbe:word>henry</qbe:word></TITLE>`
	`  </qbe:query>`
	`  <search:qtext>fourth</search:qtext>`
	`  <search:options>`
	`  <search:extract-document-data>`
	`  <search:extract-path>/PLAY/TITLE</search:extract-path>`
	`  </search:extract-document-data>`
	`  <search:transform-results apply="empty-snippet"/>`
	`  <search:search-option>filtered</search:search-option>`
	`  </search:options>`
	`</search:search>`

JSON	`{"search" : {`
	`  "$query": {`
	`  "TITLE" : { "$word": "henry" }`
	`  },`
	`  "qtext": "fourth",`
	`  "options": {`
	`  "extract-document-data": {`
	`  "extract-path": "/PLAY/TITLE"`
	`  },`
	`  "transform-results": {`
	`  "apply": "empty-snippet"`
	`  },`
	`  "search-option": ["filtered"]`
	`  }`
	`} }`

4.7 Querying Triples

You can query semantic data in the database by sending a GET request to the `/graphs/sparql` service with a URL of the following form:

```
http://host:port/version/graphs/sparql?query=sparql-query
```

Optionally, you can define the RDF Dataset over which to query by specifying one or more graph URIs using the `named-graph-uri` and/or `default-graph-uri` request parameters:

```
http://host:port/version/graphs/sparql?query=sparql-query&named-graph-
uri=graph-uri&default-graph-uri=graph-uri
```

You can also specify the dataset within the query. If you specify a dataset in both the request parameters and the query, the dataset defined with `named-graph-uri` and `default-graph-uri` takes precedence. If no dataset is defined in the request parameters or in the query, the dataset includes all triples, regardless of graph.

The SPARQL query in the query request parameter must be URL-encoded.

You can also put the query in the body of a POST request to `/graphs/sparql`. As with the GET request, define the RDF Dataset using `named-graph-uri` and/or `default-graph-uri`. For example, make a POST request with a URL of the following form:

```
http://host:port/version/graphs/sparql?named-graph-uri=graph-uri&defau
lt-graph-uri=graph-uri
```

Note: The collection lexicon must be enabled on your database before you can use the semantics REST services or use the `GRAPH '?g'` construct in a SPARQL query.

When you use POST, the request body can contain either a SPARQL query or a combined query that includes a SPARQL query. For details, see `POST:/v1/graphs`.

If you need to read graphs or query results across multiple requests that reflect the state of the database at a fixed point in time, see “Performing Point-in-Time Operations” on page 29.

For more details on working with semantic data, see Configuring the Database to Work with Triples and Semantic Queries in the *Semantics Developer's Guide*.

4.8 Invoking an Optic API Query Plan

The MarkLogic Optic API enables you to perform relational operations on indexed values and documents and view the results as row data. The `/rows` service of the REST Client API enables you to invoke an Optic API query plan and retrieve the results.

This section covers the following topics:

- Generating a Plan
- Invoking a Plan
- Controlling the Inclusion of Type Information in a Row Set
- Generating a Row Set
- Passing Parameters into a Plan

- Handling Complex Column Values
- Generating an Execution Plan

4.8.1 Generating a Plan

Use the export capability of the XQuery or Server-Side JavaScript Optic API, or `PlanBuilder.ExportablePlan.export` in the Java Client API to generate an Optic API query plan.

For more details, see the following topics:

- Exporting and Importing a Serialized Optic Query in the *Application Developer's Guide*
- `com.marklogic.client.expression.PlanBuilder.ExportablePlan` in the *Java Client API Documentation*

4.8.2 Invoking a Plan

To invoke a previously exported Optic API query plan send a GET or POST request of the following form to the `/rows` service:

```
http://host:port/version/rows
```

For a GET request, specify a URI-encoded exported Optic plan as the value of the `plan` request parameter. The plan must be expressed as JSON. For example:

```
http://localhost:8000/LATEST/rows?plan=...
```

For a POST request, put the serialized Optic plan in the request body. The plan must be expressed as JSON.

If your plan uses placeholder parameters, use the `bind` request parameters to specify values for placeholders. You must specify a binding for every placeholder parameter. For details, see "Passing Parameters into a Plan" on page 205.

The `/rows` service can produce two categories of response data: A row set resulting from execution of a plan, or an execution plan produced by the Optic "explain" feature. The default response is a row set. To generate an execution plan, use `output=explain`. For more details, see "Generating an Execution Plan" on page 207.

When generating a row set, you can use the following request parameters plus the Accept header MIME type to tailor the structure of the row set.

- `output` - Specify whether to return a row set or an execution plan, whether to a row set in the form of a JSON array or JSON object (when returning JSON), and what form of input to pass to a mapper or reducer specified in the plan.
- `column-types` - Controls whether value datatype information is embedded in each row or provided only once, in the column header.

- `row-format` - Controls whether the row parts should be formatted as JSON or XML when generating a multi-part response.
- `node-columns` - Controls the handling of non-atomic column values when generating a multi-part response. This information can be included inline or by reference. For details, see "Handling Complex Column Values" on page 205.

For more details and examples of layout variations, see the following:

- "Generating a Row Set" on page 192
- `GET:/v1/rows` in the *MarkLogic REST API Reference*
- `POST:/v1/rows` in the *MarkLogic REST API Reference*

4.8.3 Controlling the Inclusion of Type Information in a Row Set

By default, most row format layouts embed column value type information in each row. You can use the `column-types` request parameter to provide type information only in the column header data instead of each row.

```
// embed type info in each row (default behavior)
http://host:port/LATEST/rows?column-types=rows

// embed type info in the column header info
http://host:port/LATEST/rows?column-types=header
```

Only use `column-types=header` if your column value types are consistent across rows or the type information is not important to your application.

For example, if you generate a row set in the form of a single JSON object, each column value includes a "type" property, as shown in the row below. For a complete example of this row set, see "Single JSON Object" on page 193.

```
{ "columns": [
    {"name":"main.employees.EmployeeID"},
    {"name":"main.employees.FirstName"},
    {"name":"main.employees.LastName"}
  ],
  "rows":[
    { "main.employees.EmployeeID":{"type":"xs:integer","value":1},
      "main.employees.FirstName":{"type":"xs:string","value":"John"},
      "main.employees.LastName":{"type":"xs:string","value":"Widget"}
    },
...
```

If you use `column-types=header`, the type information is moved to the "columns" property, as shown here:

```
{ "columns": [
    { "name":"main.employees.EmployeeID",
```

```
      "type":"xs:integer"
    },
    {"name":"main.employees.FirstName",
      "type":"xs:string"
    },
    {"name":"main.employees.LastName",
      "type":"xs:string"
    }
  ],
  "rows":[
    { "main.employees.EmployeeID":1,
      "main.employees.FirstName":"John",
      "main.employees.LastName":"Widget"
    },...
```

The examples in “Generating a Row Set” on page 192 demonstrate how `column-types=rows` and `column-types=header` affects the output for each row set layout.

4.8.4 Generating a Row Set

The default output from GET/POST `/v1/rows` is a row set. You can use the Accept headers and request parameters to tailor the layout of the row set to meet the needs of your application. This section provides guidelines and examples for these variations.

See the following topics for settings and examples of generating each type of row set:

- Example Input Plan
- Single JSON Object
- Single JSON Array
- Single XML Element
- Line Delimited JSON Objects
- Line Delimited JSON Arrays
- Comma-Separated Text (CSV)
- Comma-Separated Arrays
- Multipart With Rows as JSON Objects
- Multipart With Rows as XML Elements

You can also use the `/rows` service to generate an execution plan rather than a row set. For details, see “Generating an Execution Plan” on page 207.

4.8.4.1 Example Input Plan

The examples in this section use the data, templates, and a plan from SQL on MarkLogic Server Quick Start in the *SQL Data Modeling Guide*. If you want to run the examples, you should use the quick start to configure a database, load the data, and create templates.

The curl commands in this section use the following exported plan as input to `POST:/v1/rows`. This plan was generated using the Optic API "explain" method and is applicable to the quick start configuration and data.

```
{"$optic":{
  "ns":"op",
  "fn":"operators",
  "args":[
    {"ns":"op", "fn":"from-view", "args":["main", "employees", null, null]},
    {"ns":"op",
     "fn":"select",
     "args":[[
         {"ns":"op", "fn":"col", "args":["EmployeeID"]},
         {"ns":"op", "fn":"col", "args":["FirstName"]},
         {"ns":"op", "fn":"col", "args":["LastName"]}
       ], null ]
    },
    {"ns":"op",
     "fn":"order-by",
     "args":[[{"ns":"op", "fn":"col", "args":["EmployeeID"]}]]
    }
  ]
}}
```

For an example of how to export a plan, see the `AccessPlan.prototype.export` JavaScript function or the `op:export` XQuery function.

You can use the same plan with `GET:/v1/rows`, but it must be URI encoded when passing it as the value of the `plan` request parameter.

4.8.4.2 Single JSON Object

The following example generates a response payload that contains a single JSON object with "columns" and "rows" properties. To get this output:

- Set the Accept header to application/json
- Set the `output` request parameter to `object` or leave it unset

The value of the "rows" property is an array containing one item per row. Each row is represented as a JSON object whose property names correspond to the column names. The structure of the row property values depends on the `column-types` request parameter.

For example, the following request produces the output shown after the curl command when run against the data from SQL on MarkLogic Server Quick Start in the *SQL Data Modeling Guide* and the plan from "Example Input Plan" on page 192.

```
curl --anyauth --user username:password -i -X POST -d @./plan.json \
  -H "Content-type: application/json" -H "Accept: application/json" \
  'http://localhost:8000/LATEST/rows?database=SQLdata&output=object'
```

```
{ "columns": [
    {"name":"main.employees.EmployeeID"},
    {"name":"main.employees.FirstName"},
    {"name":"main.employees.LastName"}
  ],
  "rows":[
    { "main.employees.EmployeeID":{"type":"xs:integer","value":1},
      "main.employees.FirstName":{"type":"xs:string","value":"John"},
      "main.employees.LastName":{"type":"xs:string","value":"Widget"}
    },
    { "main.employees.EmployeeID":{"type":"xs:integer","value":2},
      "main.employees.FirstName":{"type":"xs:string","value":"Jane"},
      "main.employees.LastName":{"type":"xs:string","value":"Lead"}
    },
    { "main.employees.EmployeeID":{"type":"xs:integer","value":3},
      "main.employees.FirstName":{"type":"xs:string","value":"Steve"},
      "main.employees.LastName":{"type":"xs:string","value":"Manager"}
    }, ...
  ]
}
```

You can use the `column-types` request parameter to extract just the value of each column, without type information. For example, if you set `column-types` to `header`, you see output similar to the following. The type information is included in the “columns” property, instead of with each row.

```
{ "columns": [
    { "name":"main.employees.EmployeeID",
      "type":"xs:integer"
    },
    {"name":"main.employees.FirstName",
      "type":"xs:string"
    },
    {"name":"main.employees.LastName",
      "type":"xs:string"
    }
  ],
  "rows":[
    { "main.employees.EmployeeID":1,
      "main.employees.FirstName":"John",
      "main.employees.LastName":"Widget"
    },
    ...
  ]
}
```

4.8.4.3 Single JSON Array

The following example generates a response payload that contains a single JSON array. To get this output:

- Set the Accept header to application/json
- Set the `output` request parameter to `array`

Each item of the top level array is an array. The first item is an array containing the column names. Each subsequent item is an array representing one row, with one item per column value. The order of the columns is consistent for the header and each row in the row set. The structure of the column values depends on the `column-types` request parameter.

For example, the following request produces the output shown after the curl command when run against the data from SQL on MarkLogic Server Quick Start in the *SQL Data Modeling Guide* and the plan from "Example Input Plan" on page 192.

```
curl --anyauth --user username:password -i -X POST -d @./plan.json \
  -H "Content-type: application/json" -H "Accept: application/json" \
  'http://localhost:8000/LATEST/rows?database=SQLdata&output=array'

[
  [ {"name":"main.employees.EmployeeID"},
    {"name":"main.employees.FirstName"},
    {"name":"main.employees.LastName"}
  ],[
    {"type":"xs:integer","value":1},
    {"type":"xs:string","value":"John"},
    {"type":"xs:string","value":"Widget"}
  ],[
    {"type":"xs:integer","value":2},
    {"type":"xs:string","value":"Jane"},
    {"type":"xs:string","value":"Lead"}
  ],[
    {"type":"xs:integer","value":3},
    {"type":"xs:string","value":"Steve"},
    {"type":"xs:string","value":"Manager"}
  ], ...
]
```

You can use the `column-types` request parameter to extract just the value of each column, without type information. For example, if you set `column-types` to `header`, you see output similar to the following. The type information is included in the column header array, instead of in each row.

```
[
  [ {"name":"main.employees.EmployeeID","type":"xs:integer"},
    {"name":"main.employees.FirstName","type":"xs:string"},
    {"name":"main.employees.LastName","type":"xs:string"}
  ],
  [1,"John","Widget"],
  [2,"Jane","Lead"],
  [3,"Steve","Manager"],
  ...
]
```

4.8.4.4 Single XML Element

The following example generates a response payload that contains a single XML element that represents a table. To get this output:

- Set the Accept header to application/xml

You might also choose to set the `output` request parameter as it affects the form of input to any mapper or reducer used by the plan, but the response payload is not affected by this parameter when generating XML.

The response is rooted at a single `<table/>` element. The table contains one `<columns/>` element containing column header data and one `<rows/>` element containing the row data. The order of the columns is consistent for the header and each row in the row set. The structure of the data also depends on the `column-types` request parameter.

For example, the following request produces the output shown after the curl command when run against the data from SQL on MarkLogic Server Quick Start in the *SQL Data Modeling Guide* and the plan from “Example Input Plan” on page 192.

```
curl --anyauth --user username:password -i -X POST -d @./plan.json \
  -H "Content-type: application/json" -H "Accept: application/xml" \
  'http://localhost:8000/LATEST/rows?database=SQLdata'

<t:table xmlns:t="http://marklogic.com/table">
  <t:columns>
    <t:column name="main.employees.EmployeeID"/>
    <t:column name="main.employees.FirstName"/>
    <t:column name="main.employees.LastName"/>
  </t:columns>
  <t:rows>
    <t:row>
      <t:cell name="main.employees.EmployeeID" type="xs:integer">1</t:cell>
      <t:cell name="main.employees.FirstName" type="xs:string">John</t:cell>
      <t:cell name="main.employees.LastName" type="xs:string">Widget</t:cell>
    </t:row>
    <t:row>
      <t:cell name="main.employees.EmployeeID" type="xs:integer">2</t:cell>
      <t:cell name="main.employees.FirstName" type="xs:string">Jane</t:cell>
      <t:cell name="main.employees.LastName" type="xs:string">Lead</t:cell>
    </t:row>
    <t:row>
      <t:cell name="main.employees.EmployeeID" type="xs:integer">3</t:cell>
      <t:cell name="main.employees.FirstName" type="xs:string">Steve</t:cell>
      <t:cell name="main.employees.LastName" type="xs:string">Manager</t:cell>
    </t:row>
    ...
  </t:rows>
</t:table>
```

You can use the `column-types` request parameter to extract just the value of each column, without type information. For example, if you set `column-types` to `header`, you see output similar to the following. The type information is included in the column header element, rather than in each row element.

```
<t:table xmlns:t="http://marklogic.com/table">
  <t:columns>
    <t:column name="main.employees.EmployeeID" type="xs:integer"/>
    <t:column name="main.employees.FirstName" type="xs:string"/>
    <t:column name="main.employees.LastName" type="xs:string"/>
  </t:columns>
  <t:rows>
    <t:row>
      <t:cell name="main.employees.EmployeeID">1</t:cell>
      <t:cell name="main.employees.FirstName">John</t:cell>
      <t:cell name="main.employees.LastName">Widget</t:cell>
    </t:row>
    ...
  </t:rows>
</t:table>
```

4.8.4.5 Line Delimited JSON Objects

The following example generates a response payload that contains line-delimited JSON objects. To get this output:

- Set the Accept header to application/json-seq
- Set the `output` request parameter to `object` or leave it unset

The application/json-seq MIME type is based on the following RFC: https://tools.ietf.org/html/rfc7464.

The first line in the response is an object containing the column names. The following lines each represent a row, expressed as a JSON object. The property names of each row object correspond to the column names. The structure of the row property values depends on the `column-types` request parameter.

For example, the following request produces the output shown after the curl command when run against the data from SQL on MarkLogic Server Quick Start in the *SQL Data Modeling Guide* and the plan from “Example Input Plan” on page 192.

```
curl --anyauth --user username:password -i -X POST -d @./plan.json \
  -H "Content-type: application/json" -H "Accept: application/json-seq" \
  'http://localhost:8000/LATEST/rows?database=SQLdata&output=object'

{"columns":[{"name":"main.employees.EmployeeID"},{"name":"main.employe
es.FirstName"},{"name":"main.employees.LastName"}]}
{"main.employees.EmployeeID":{"type":"xs:integer","value":1},"main.emp
loyees.FirstName":{"type":"xs:string","value":"John"},"main.employees.
LastName":{"type":"xs:string","value":"Widget"}}
```

```
{"main.employees.EmployeeID":{"type":"xs:integer","value":2},"main.emp
loyees.FirstName":{"type":"xs:string","value":"Jane"},"main.employees.
LastName":{"type":"xs:string","value":"Lead"}}
{"main.employees.EmployeeID":{"type":"xs:integer","value":3},"main.emp
loyees.FirstName":{"type":"xs:string","value":"Steve"},"main.employees
.LastName":{"type":"xs:string","value":"Manager"}}
...
```

You can use the `column-types` request parameter to extract just the value of each column, without type information. For example, if you set `column-types` to `header`, you see output similar to the following. The type information is included in the column header row, rather than in each row.

```
{"columns":[{"name":"main.employees.EmployeeID","type":"xs:integer"},{
"name":"main.employees.FirstName","type":"xs:string"},{"name":"main.em
ployees.LastName","type":"xs:string"}]}
{"main.employees.EmployeeID":1,"main.employees.FirstName":"John","main
.employees.LastName":"Widget"}
{"main.employees.EmployeeID":2,"main.employees.FirstName":"Jane","main
.employees.LastName":"Lead"}
{"main.employees.EmployeeID":3,"main.employees.FirstName":"Steve","mai
n.employees.LastName":"Manager"}
...
```

4.8.4.6 Line Delimited JSON Arrays

The following example generates a response payload that contains line-delimited JSON arrays. To get this output:

- Set the Accept header to application/json-seq
- Set the `output` request parameter to `array`

The application/json-seq MIME type is based on the following RFC: https://tools.ietf.org/html/rfc7464.

The first line in the response is an array containing the column names (as JSON objects). The following lines each represent a row, expressed as a JSON array of objects, with each object representing a column value. The property names of each row object correspond to the column names. The structure of the row property values depends on the `column-types` request parameter.

For example, the following request produces the output shown after the curl command when run against the data from SQL on MarkLogic Server Quick Start in the *SQL Data Modeling Guide* and the plan from “Example Input Plan” on page 192.

```
curl --anyauth --user username:password -i -X POST -d @./plan.json \
  -H "Content-type: application/json" -H "Accept: application/json-seq" \
  'http://localhost:8000/LATEST/rows?database=SQLdata&output=array'

[{"name":"main.employees.EmployeeID"},{"name":"main.employees.FirstNam
e"},{"name":"main.employees.LastName"}]
[{"type":"xs:integer","value":1},{"type":"xs:string","value":"John"},{
```

```
"type":"xs:string","value":"Widget"}]
[{"type":"xs:integer","value":2},{"type":"xs:string","value":"Jane"},{
"type":"xs:string","value":"Lead"}]
[{"type":"xs:integer","value":3},{"type":"xs:string","value":"Steve"},
{"type":"xs:string","value":"Manager"}]
...
```

You can use the `column-types` request parameter to extract just the value of each column, without type information. For example, if you set `column-types` to `header`, you see output similar to the following. The type information is included in the column header line, rather than each row.

```
[{"name":"main.employees.EmployeeID","type":"xs:integer"},{"name":"mai
n.employees.FirstName","type":"xs:string"},{"name":"main.employees.Las
tName","type":"xs:string"}]
[1,"John","Widget"]
[2,"Jane","Lead"]
[3,"Steve","Manager"]
...
```

4.8.4.7 Comma-Separated Text (CSV)

The following example generates a response payload that contains a row set as CSV data. To get this output:

- Set the Accept header to text/csv
- Set the `output` request parameter to `object` or leave it unset

The first line in the response is a comma-separated list of column names. The following lines each represent a row, with comma-separated column values.

For example, the following request produces the output shown after the curl command when run against the data from SQL on MarkLogic Server Quick Start in the *SQL Data Modeling Guide* and the plan from “Example Input Plan” on page 192.

```
curl --anyauth --user username:password -i -X POST -d @./plan.json \
  -H "Content-type: application/json" -H "Accept: text/csv" \
  'http://localhost:8000/LATEST/rows?database=SQLdata&output=object'

main.employees.EmployeeID,main.employees.FirstName,main.employees.LastName
1,John,Widget
2,Jane,Lead
3,Steve,Manager
...
```

4.8.4.8 Comma-Separated Arrays

The following example generates a response payload that contains a row set as CSV data. To get this output:

- Set the Accept header to text/csv

- Set the `output` request parameter to `array`

The first line in the response is an array containing the column names. The following lines each represent a row, expressed as an array. Each array item is a column value.

For example, the following request produces the output shown after the curl command when run against the data from SQL on MarkLogic Server Quick Start in the *SQL Data Modeling Guide* and the plan from "Example Input Plan" on page 192.

```
curl --anyauth --user username:password -i -X POST -d @./plan.json \
  -H "Content-type: application/json" -H "Accept: text/csv" \
  'http://localhost:8000/LATEST/rows?database=SQLdata&output=array'

["main.employees.EmployeeID", "main.employees.FirstName",
"main.employees.LastName"]
[1, "John", "Widget"]
[2, "Jane", "Lead"]
[3, "Steve", "Manager"]
...
```

4.8.4.9 Multipart With Rows as JSON Objects

The following example generates a multipart response payload that contains a part for the column names, and a part for each row, with the row data expressed as a JSON object. To get this output:

- Set the Accept header to multipart/mixed
- Set the `output` request parameter to `object`, or leave it unset
- Set the `row-format` request parameter to `json`, or leave it unset

The first part contains the column names, expressed as a JSON object. The Content-Disposition part header includes a `kind=columns` specifier. By default, each subsequent part contains the contents of one row, expressed as a JSON object whose property names correspond to the column names. The Content-Disposition part header for a row includes a `kind=row` specifier.

The structure of the values also depends on the `column-types` and `node-columns` request parameters. See the examples below and "Handling Complex Column Values" on page 205.

For example, the following request produces the output shown after the curl command when run against the data from SQL on MarkLogic Server Quick Start in the *SQL Data Modeling Guide* and the plan from "Example Input Plan" on page 192.

```
curl --anyauth --user username:password -i -X POST -d @./plan.json \
  -H "Content-type: application/json" \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/rows?database=SQLdata&output=object'

--BOUNDARY
Content-Type: application/json; charset=utf-8
Content-Disposition: inline; kind=columns
```

```
{"columns":[{"name":"main.employees.EmployeeID"},{"name":"main.employe
es.FirstName"},{"name":"main.employees.LastName"}]}
--BOUNDARY
Content-Type: application/json; charset=utf-8
Content-Disposition: inline; kind=row

{"main.employees.EmployeeID":{"type":"xs:integer","value":1},"main.emp
loyees.FirstName":{"type":"xs:string","value":"John"},"main.employees.
LastName":{"type":"xs:string","value":"Widget"}}
--BOUNDARY
Content-Type: application/json; charset=utf-8
Content-Disposition: inline; kind=row

{"main.employees.EmployeeID":{"type":"xs:integer","value":2},"main.emp
loyees.FirstName":{"type":"xs:string","value":"Jane"},"main.employees.
LastName":{"type":"xs:string","value":"Lead"}}
--BOUNDARY
Content-Type: application/json; charset=utf-8
Content-Disposition: inline; kind=row

{"main.employees.EmployeeID":{"type":"xs:integer","value":3},"main.emp
loyees.FirstName":{"type":"xs:string","value":"Steve"},"main.employees
.LastName":{"type":"xs:string","value":"Manager"}}
--BOUNDARY--
...
```

You can use the `column-types` request parameter to extract just the value of each column, without type information. For example, if you set `column-types` to `header`, you see output similar to the following. The type information appears in the column part instead of in each row part.

```
--BOUNDARY
Content-Type: application/json; charset=utf-8
Content-Disposition: inline; kind=columns

{"columns":[{"name":"main.employees.EmployeeID","type":"xs:integer"},{
"name":"main.employees.FirstName","type":"xs:string"},{"name":"main.em
ployees.LastName","type":"xs:string"}]}
--BOUNDARY
Content-Type: application/json; charset=utf-8
Content-Disposition: inline; kind=row

{"main.employees.EmployeeID":1,"main.employees.FirstName":"John","main
.employees.LastName":"Widget"}
```

4.8.4.10 Multipart With Rows as JSON Arrays

The following example generates a multipart response payload that contains a part for the column names, and a part for each row, with the row data expressed as JSON arrays. To get this output:

- Set the Accept header to multipart/mixed
- Set the `output` request parameter to `array`

- Set the `row-format` request parameter to `json`, or leave it unset

The first part contains the column names, expressed as a JSON array. The Content-Disposition part header includes a `kind=columns` specifier.

By default, each subsequent part contains the contents of one row, expressed as a JSON array. Each array item represents one column value. The Content-Disposition part header for a row includes a `kind=row` specifier.

The structure of the column values also depends on the `column-types` and `node-columns` request parameters. See the examples below and “Handling Complex Column Values” on page 205.

For example, the following request produces the output shown after the curl command when run against the data from SQL on MarkLogic Server Quick Start in the *SQL Data Modeling Guide* and the plan from “Example Input Plan” on page 192.

```
curl --anyauth --user username:password -i -X POST -d @./plan.json \
  -H "Content-type: application/json" \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/rows?database=SQLdata&output=array'

--BOUNDARY
Content-Type: application/json; charset=utf-8
Content-Disposition: inline; kind=columns

[{"name":"main.employees.EmployeeID"},{"name":"main.employees.FirstNam
e"},{"name":"main.employees.LastName"}]
--BOUNDARY
Content-Type: application/json; charset=utf-8
Content-Disposition: inline; kind=row

[{"type":"xs:integer","value":1},{"type":"xs:string","value":"John"},{
"type":"xs:string","value":"Widget"}]
--BOUNDARY
Content-Type: application/json; charset=utf-8
Content-Disposition: inline; kind=row

[{"type":"xs:integer","value":2},{"type":"xs:string","value":"Jane"},{
"type":"xs:string","value":"Lead"}]
--BOUNDARY
Content-Type: application/json; charset=utf-8
Content-Disposition: inline; kind=row

[{"type":"xs:integer","value":3},{"type":"xs:string","value":"Steve"},
{"type":"xs:string","value":"Manager"}]
--BOUNDARY
...
```

You can use the `column-types` request parameter to extract just the value of each column, without type information. For example, if you set `column-types` to `header`, you see output similar to the following. The type information appears in the column part instead of in each row part.

```
--BOUNDARY
Content-Type: application/json; charset=utf-8
Content-Disposition: inline; kind=columns

[{"name":"main.employees.EmployeeID","type":"xs:integer"},{"name":"mai
n.employees.FirstName","type":"xs:string"},{"name":"main.employees.Las
tName","type":"xs:string"}]
--BOUNDARY
Content-Type: application/json; charset=utf-8
Content-Disposition: inline; kind=row

[1,"John","Widget"]
```

4.8.4.11 Multipart With Rows as XML Elements

The following example generates a multipart response payload that contains a part for the column names, and a part for each row, with the data expressed XML elements. To get this output:

- Set the Accept header to multipart/mixed
- Set the `row-format` request parameter to `xml`

You might also choose to set the `output` request parameter as it affects the form of input to any mapper or reducer used by the plan, but the response payload is not affected by this parameter when generating XML.

The first part contains the column names, expressed as `columns` XML element. The Content-Disposition part header includes a `kind=columns` specifier. Each subsequent part contains the contents of one row, expressed as a `row` XML element. The Content-Disposition part header for a row includes a `kind=row` specifier.

The structure of the column values also depends on the `column-types` and `node-columns` request parameters. See the examples below and “Handling Complex Column Values” on page 205.

For example, the following request produces the output shown after the curl command when run against the data from SQL on MarkLogic Server Quick Start in the *SQL Data Modeling Guide* and the plan from “Example Input Plan” on page 192.

```
curl --anyauth --user username:password -i -X POST -d @./plan.json \
  -H "Content-type: application/json" \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/rows?database=SQLdata&row-format=xml'

--BOUNDARY
Content-Type: application/xml; charset=utf-8
Content-Disposition: inline; kind=columns
```

```
<t:columns xmlns:t="http://marklogic.com/table">
<t:column name="main.employees.EmployeeID"/>
<t:column name="main.employees.FirstName"/>
<t:column name="main.employees.LastName"/>
</t:columns>

--BOUNDARY
Content-Type: application/xml; charset=utf-8
Content-Disposition: inline; kind=row

<t:row xmlns:t="http://marklogic.com/table"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<t:cell name="main.employees.EmployeeID" type="xs:integer">1</t:cell>
<t:cell name="main.employees.FirstName" type="xs:string">John</t:cell>
<t:cell name="main.employees.LastName"
type="xs:string">Widget</t:cell>
</t:row>

--BOUNDARY
Content-Type: application/xml; charset=utf-8
Content-Disposition: inline; kind=row

<t:row xmlns:t="http://marklogic.com/table"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<t:cell name="main.employees.EmployeeID" type="xs:integer">2</t:cell>
<t:cell name="main.employees.FirstName" type="xs:string">Jane</t:cell>
<t:cell name="main.employees.LastName" type="xs:string">Lead</t:cell>
</t:row>

--BOUNDARY
Content-Type: application/xml; charset=utf-8
Content-Disposition: inline; kind=row

<t:row xmlns:t="http://marklogic.com/table"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<t:cell name="main.employees.EmployeeID" type="xs:integer">3</t:cell>
<t:cell name="main.employees.FirstName"
type="xs:string">Steve</t:cell>
<t:cell name="main.employees.LastName"
type="xs:string">Manager</t:cell>
</t:row>

--BOUNDARY

...
```

You can use the `column-types` request parameter to extract just the value of each column, without type information. For example, if you set `column-types` to `header`, you see output similar to the following. Type information is included in the column part instead of each row part.

```
--BOUNDARY
Content-Type: application/xml; charset=utf-8
```

```
Content-Disposition: inline; kind=columns

<t:columns xmlns:t="http://marklogic.com/table">
<t:column name="main.employees.EmployeeID" type="xs:integer"/>
<t:column name="main.employees.FirstName" type="xs:string"/>
<t:column name="main.employees.LastName" type="xs:string"/>
</t:columns>

--BOUNDARY
Content-Type: application/xml; charset=utf-8
Content-Disposition: inline; kind=row

<t:row xmlns:t="http://marklogic.com/table"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<t:cell name="main.employees.EmployeeID">1</t:cell>
<t:cell name="main.employees.FirstName">John</t:cell>
<t:cell name="main.employees.LastName">Widget</t:cell>
</t:row>
```

4.8.5 Passing Parameters into a Plan

If your plan uses placeholder parameters, use the `bind` request parameter to pass values for the placeholders into the plan.

You can specify just a value for the named parameter, or a value and a type, or a value and a language code. If you do not specify a type, the value is interpreted as a string. For more details, see Parameterizing a Plan in the *Application Developer's Guide*.

For example, if you defined a placeholder variable named "start" in your plan definition, then you can specify a value for the parameter in the following ways:

```
http://localhost:8000/LATEST/rows?bind:start=apple

http://localhost:8000/LATEST/rows?bind:start:string=apple

http://localhost:8000/LATEST/rows?bind:start@en=apple
```

4.8.6 Handling Complex Column Values

If a row contains column values with non-atomic type, such as XML element, JSON array, JSON object, binary, or text nodes, MarkLogic serializes them inline by default. If the non-atomic type is not native to the serialization format, such as an XML element column value in a row serialized as JSON, you can optionally extract as a separate part and refer to it by reference in the serialized row by using the `node-columns=reference` request parameter.

For example, suppose you extract rows in which one column contains an XML element value and another column contains a JSON object value. If you serialize the row as JSON, then the JSON object column values can be represented natively, but the XML elements become just a string:

```
{ "row":1,
  "elem":"<alpha><a>true</a></alpha>",
  "obj":{"alpha":10}
}
```

If you generate a multipart/mixed response, then you can use the `node-columns` request parameter to generate rows containing a reference to the non-native complex values instead of inlining them. The referenced value is provided in a separate part. For example:

```
curl --anyauth --user username:password -i -X POST -d @./complex.json \
  -H "Content-type: application/json" \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/rows?database=SQLdata&node-columns=ref
erence'
...
--BOUNDARY
Content-Type: application/json; charset=utf-8
Content-Disposition: inline; kind=columns

{"columns":[{"name":"row"},{"name":"elem"},{"name":"obj"}]}
--BOUNDARY
Content-Type: application/json; charset=utf-8
Content-Disposition: inline; kind=row

{"row":{"type":"xs:integer","value":1},"elem":{"type":"cid","value":"c
id:elem[0]"},"obj":{"type":"object","value":{"alpha":10}}}
--BOUNDARY
Content-Type: application/xml; charset=utf-8
Content-ID: <elem[0]>
Content-Disposition: inline; kind=row-attachment

<alpha><a>true</a></alpha>
...
```

The row parts are identifiable by the “kind=row” in the Content-disposition header. The complex value parts are identifiable by “kind=row-attachment” in the Content-disposition header. The row attachment parts have a Content-Type part header that accurately reflects the MIME type of the complex column value.

The value part reference uses the id from the Content-id part header on the referenced value part. The content id is based on the column name and row number. The content id uses the standard Content-id/CID format described in the following RFC: https://tools.ietf.org/html/rfc2392.

For example, if you have the following value part:

```
Content-Type: application/xml; charset=utf-8
Content-ID: <elem[0]>
```

```
Content-Disposition: inline; kind=row-attachment

<alpha><a>true</a></alpha>
```

Then a row referencing this part has the following form if you do not use any row formatting parameters. Notice the use of "cid:elem[0]" to reference the value part.

```
{"row":{"type":"xs:integer","value":1},"elem":{"type":"cid","value":"c
id:elem[0]"},"obj":{"type":"object","value":{"alpha":10}}}
```

The layout of the column value varies, depending on your use of formatting parameters such as `output`, `column-types`, and `row-format`. The following examples illustrate a few variants:

```
// using output=array
[{"type":"xs:integer","value":1},
 {"type":"cid","value":"cid:elem[0]"},
 {"type":"object","value":{"alpha":10}}]

// using output=object and column-types=header
{"row":1,"elem":"cid:elem[0]","obj":{"alpha":10}}

// using row-format=xml
<t:row xmlns:t="http://marklogic.com/table"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <t:cell name="row" type="xs:integer">1</t:cell>
  <t:cell name="elem" type="cid">cid:elem[0]</t:cell>
  <t:cell name="obj" type="object">{"alpha":10}</t:cell>
</t:row>
```

For more complete examples of output formatting variations, see "Generating a Row Set" on page 192.

4.8.7 Generating an Execution Plan

An Optic API execution plan expresses the logical dataflow of a plan as a sequence of atomic operations. For more details, see Optic Execution Plan in the *Application Developer's Guide*.

You can generate an execution plan with the /rows service by setting the output parameter value to "explain". Use the Accept header to specify either a JSON or XML response.

For example, the following command generates a JSON execution plan.

```
curl --anyauth --user username:password -i -X POST -d @./plan.json \
  -H "Content-type: application/json" \
  -H "Accept: application/json" \
  'http://localhost:8000/LATEST/rows?database=SQLdata&output=explain'
```

4.9 Searching Values Metadata Fields

Values metadata, sometimes called key-value metadata, can only be searched if you define a metadata field on the keys you want to search. Once you define a field on a metadata key, use the normal field search capabilities to include a metadata field in your search. For example, you can use a `cts:field-word-query` or a structured query `word-query` on a metadata field, or define a constraint on the field and use the constraint in a string query.

For more details, see Metadata Fields in the *Administrator's Guide*. For some examples, see Example: Structured Search on Key-Value Metadata Fields or Searching Key-Value Metadata Fields in the *Search Developer's Guide*.

4.10 Configuring Query Options

Use the `/config/query` resources to install, list, and manage sets of query options. Use the `options` request parameter to apply installed query options to requests to `/search`, `/qbe`, `/values`, and `/suggest`. This section covers the following topics:

- Controlling Queries With Options
- Adding Query Options to a Request
- Creating or Modifying Query Options
- Checking Index Availability
- Retrieving Options
- Retrieving a List of Installed Query Options
- Removing Query Options
- Removing All Named Query Options

4.10.1 Controlling Queries With Options

You can use persistent or dynamic query options to customize your queries. MarkLogic Server comes configured with default query options. You can extend and modify the default options using `/config/query/default`.

Use the `options` request parameter to the `/search`, `/qbe`, `/values`, and `/suggest` services to customize your queries. Query options provide capabilities such as:

- define word, value and element constraints
- define lexicon and range index specifications
- control search characteristics such as case sensitivity and ordering
- extend the search grammar
- customize query results including pagination, snippeting, and filtering

For details on these and other options, see Appendix: Query Options Reference in the *Search Developer's Guide*.

You can also create custom persistent or dynamic query options. Persistent query options are named, pre-defined query options that you install using the `/config/query/{name}`. Dynamic query options are per-request, transient options defined in a combined query. For details, see "Creating or Modifying Query Options" on page 210 and "Specifying Dynamic Query Options with Combined Query" on page 179

Once you install query options under a name, you can apply them to a `/search`, `/qbe`, `/values`, or `/suggest` request using the `options` request parameter. The following example searches for the word "julius", using the query options named "my-options":

```
http://localhost:8000/LATEST/search?q=julius&options=my-options
```

4.10.2 Adding Query Options to a Request

You can customize a query with query options in the following ways:

- Use the `options` request parameter of a GET request to `/search`, `/qbe`, `/values/{name}`, or `/suggest` to supply the name of pre-installed persistent options.
- Use the `options` parameter of a POST request to `/search`, `/qbe`, `/values/{name}`, or `/suggest` to supply the name of pre-installed persistent options.
- Use the `options` element (XML) or sub-object (JSON) of a combined query passed in the body of a POST request to `/search` or `/values/{name}` to supply dynamic options.

Pre-installed, persistent query options usually provide better performance. Using dynamic query options introduces option parsing and merging overhead to every query.

Persistent and dynamic query options can be specified in either XML or JSON. Persistent options must be installed before you can use them; for details, see "Creating or Modifying Query Options" on page 210.

Dynamic options are only usable with services that support POSTing a combined query in the request body. Methods that support combined query allow you to specify persistent and dynamic options in the same request. Where both are present, they are merged; in case of a conflict, a dynamic option setting overrides a persistent option setting. For details, see "Specifying Dynamic Query Options with Combined Query" on page 179.

4.10.3 Creating or Modifying Query Options

To install or modify named persistent query options, send a PUT or POST request to the `/config/query` service with a URL of the form:

```
http://host:port/version/config/query/name
```

When constructing the request:

1. Set the *name* portion of the URL to a unique name for these options, or to `default`. Use the name to identify the query options in subsequent request, as described in “Controlling Queries With Options” on page 208.

2. Place the XML or JSON option data in the request body.

   For syntax details, see Appendix: Query Options Reference in the *Search Developer's Guide*.

3. Specify the MIME type of the body content in the `format` parameter or the HTTP `Content-type` header, as described in “Controlling Input and Output Content Type” on page 31. You may only send XML or JSON. The default format is XML.

When choosing the HTTP verb, consider the following:

- To install new query options, use either PUT or POST.
- To replace the named query options, use PUT.
- To add options to the named query options, use POST. Any options not included in the payload remain unchanged. New options are added.

If query option validation is enabled, the request will fail if it results in invalid query options. If the request fails, the options are unchanged. Option validation is enabled by default. You can disable option validation using the `validate-options` instance configuration property. For details, see “Configuring Instance Properties” on page 47.

The following example installs query options named “title-only” that define a search constraint named “title”. The constraint limits queries to terms appearing in a `TITLE` element. The query options are then used to find occurrences of “julius” in `TITLE` elements:

```
$ cat bill-options.txt
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="title">
    <word>
      <element ns="" name="TITLE" />
    </word>
  </constraint>
</options>

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -T './bill-options.txt' \
```

```
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/config/query/title-only
...
HTTP/1.1 204 Content Updated

$ curl --anyauth --user user:password -X GET \
  'http://localhost:8000/LATEST/search?q=title:julius&options=title-on
ly'
...
<search:response total="1" start="1" page-length="10"
   xmlns="" xmlns:search="http://marklogic.com/appservices/search">
    ...
</search:response>
```

To add case-sensitivity to the query options installed above, this example sends a POST request to `/config/query`. The body content type, JSON, is given via the `Content-type` header.

```
$ cat add-cs.json
{
  "options":
    { "term":
      { "term-option":"case-sensitive" }
    }
}

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X POST -d@'add-cs.json' \
    -H "Content-type: application/json" \
    http://localhost:8000/LATEST/config/query/title-only
...
HTTP/1.1 201 Content Created
```

To confirm the change, the modified option is fetched using a GET request:

```
$ curl --anyauth --user user:password -X GET \
    http://localhost:8000/LATEST/config/query/title-only
...
<options xmlns="http://marklogic.com/appservices/search">

  <constraint name="title">
    <word>
      <element ns="" name="TITLE"/>
    </word>
  </constraint>
  <term>
    <term-option>case-sensitive</term-option>
  </term>
</options>
```

4.10.4 Creating or Modifying One Option

To add or modify just one setting in a set of query options, send a PUT or POST request to the `/config/query` service with a URL of the form:

```
http://host:port/version/config/query/name/option_name
```

When constructing the request:

1. Set the *name* portion of the URL to the name of the enclosing query options, or to `default`.
2. Set the *option_name* portion of the URL to a query option name, such as `constraint` or `term`.
3. Place the XML or JSON option data in the request body. The data should be an `options` node (XML) or map (JSON) that includes the option named in the URL.
4. Specify the content type of the body in the `format` parameter or the HTTP `Content-type` header, as described in "Controlling Input and Output Content Type" on page 31. You may only send XML or JSON. The default format is XML.

The *option_name* portion of the URL must be the name of an option that can appear as an immediate child of a query options XML node or JSON object. Finer grained access is not supported. For details on query options names and structure, see Appendix: Query Options Reference in the *Search Developer's Guide*.

When choosing the HTTP verb, consider the following:

- To add a new option to the set, use either PUT or POST.
- To replace all existing options of the same name, use PUT.
- To add a new occurrence of an existing option that can appear multiple times, use POST. For example, query options can contain multiple `<constraint/>` elements.

If query option validation is enabled, the request will fail if it results in invalid query options. If the request fails, the options are unchanged. Option validation is enabled by default. You can disable option validation using the validate-options instance configuration property. For details, see "Configuring Instance Properties" on page 47.

4.10.5 Checking Index Availability

Some query options require the database configuration to include supporting indexes. For example, if your query options contain a range constraint, then you can only use those options on a database whose configuration includes a corresponding range index.

You can use the /config/indexes service to compare query options to the database configuration and get a report on whether or not all required indexes are present. For missing indexes, the report includes information to help create the missing index. You can either check all query options configurations, or a particular one (by name).

To check all query options configurations, send a GET request to the `/config/indexes` service of the form:

```
http://host:port/version/config/indexes
```

To check a specific query options configuration, send a GET request to the `/config/indexes/{name}` service of the form:

```
http://host:port/version/config/indexes/name
```

Where *name* is the name under which the options were installed using the `/config/query/{name}` service, as described in “Creating or Modifying Query Options” on page 210.

You can request an index report in XML, JSON, or HTML, using either the `format` request parameter or the HTTP Accept headers. The HTML report is a user-friendly report that contains more details.

For example, suppose the following query options are installed under the name “tuples”. These options require 2 range indexes: An element range index on `<SPEAKER/>` and a path range index on the XPath expression `/PLAY/ACT/SCENE/TITLE`:

```
<options xmlns="http://marklogic.com/appservices/search">
  <tuples name="speaker-title">
    <range type="xs:string">
      <element ns="" name="SPEAKER"/>
    </range>
    <range type="xs:string">
      <path-index>/PLAY/ACT/SCENE/TITLE</path-index>
    </range>
  </tuples>
</options>
```

The following command requests an index check check report in XML for the “tuples” options. Use the `format` parameter or the Accept headers to request a different report format.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
    http://localhost:8000/LATEST/config/indexes/tuples
```

The table below shows the generated report. Notice that the report indicates that the index configuration is not complete, and shows which query option is not complete. You can use the `path-index` value to create the required path range index.

Format	Example Output

XML	<rapi:index-summaries xmlns:rapi="http://marklogic.com/rest-api">
	  <rapi:index-count>1</rapi:index-count>
	  <rapi:complete>false</rapi:complete>
	  <rapi:index-summary>
	  <rapi:name>**/v1/config/query/tuples**</rapi:name>
	  <rapi:complete>**false**</rapi:complete>
	  <range type="xs:string"
	  xmlns="http://marklogic.com/appservices/search">
	  <path-index>/PLAY/ACT/SCENE/TITLE</path-index>
	  </range>
	  </rapi:index-summary>
	</rapi:index-summaries>

JSON	{
	  "index-summaries": {
	  "index-summary": [
	  {
	  "name": "\/v1\/config\/query\/tuples",
	  "complete": "false",
	  "range": {
	  "type": "xs:string",
	  "path-index": "\/PLAY\/ACT\/SCENE\/TITLE"
	  }
	  }
	  ],
	  "index-count": "1",
	  "complete": "false"
	  }
	}

4.10.6 Retrieving Options

To retrieve previously installed persistent query options, send a GET request to the `/config/query` service with a URL of the form:

```
http://host:port/version/config/query/name
```

Where name is the *name* of the query options.

MarkLogic Server responds with the contents of the named query options, as XML or JSON. XML is the default format. Use the Accept header or `format` request parameter to select the output content type. For details, see “Controlling Input and Output Content Type” on page 31.

You can also retrieve the settings for a specific option within the query options by sending a GET request with a URL of the form:

```
http://host:port/version/config/query/name/option_name
```

Where *option_name* is an option name. For details, see Appendix: Query Options Reference in the *Search Developer's Guide*.

As when retrieving a whole set, results can be requested as XML or JSON. Use the Accept header or `format` request parameter to select the output content type. For details, see "Controlling Input and Output Content Type" on page 31. No version information is returned when examining only one option.

The following example retrieves the contents of the query options called "title-only":

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    http://localhost:8000/LATEST/config/query/title-only
...
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="title">
    <word>
      <element ns="" name="TITLE"/>
    </word>
  </constraint>
  <term>
    <term-option>case-sensitive</term-option>
  </term>
</options>
```

The following example retrieves only the term option of the "title-only" query options:

```
$ curl --anyauth --user user:password -X GET \
    http://localhost:8000/LATEST/config/query/title-only/term
...
<options xmlns="http://marklogic.com/appservices/search">
  <term>
    <term-option>case-sensitive</term-option>
  </term>
</options>
```

4.10.7 Retrieving a List of Installed Query Options

To retrieve a list of the names of all installed query options, send a GET request to `/config/query` with a URL of the form:

```
http://host:port/version/config/query
```

MarkLogic Server responds with a list of names in XML or JSON. XML is the default format. Use the Accept header or `format` request parameter to select the output content type. For details, see "Controlling Input and Output Content Type" on page 31.

If there are no custom query options installed, MarkLogic Server responds with an empty XML `<options>` node or JSON array.

The following example retrieves the list of named query options as XML. The results show 2 sets of query options, named “title-only” and “play-type”.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    http://localhost:8000/LATEST/config/query
...
<rapi:query-options xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:options>
    <rapi:name>title-only</rapi:name>
    <rapi:uri>/v1/config/query/title-only</rapi:uri>
  </rapi:options>
  <rapi:options>
    <rapi:name>play-type</rapi:name>
    <rapi:uri>/v1/config/query/play-type</rapi:uri>
  </rapi:options>
</rapi:query-options>
```

The following example requests the same information as JSON, using the `format` request parameter:

```
$ curl --anyauth --user user:password -X GET \
    http://localhost:8000/LATEST/config/query
...
[
  {"name":"title-only","uri":"/v1/config/query/title-only"}
  {"name":"play-type","uri":"/v1/config/query/play-type"}
]
```

4.10.8 Removing Query Options

You can remove a single setting, one set of query options, or all query options, as described in the following topics:

- Removing Query Options
- Removing a Single Option
- Removing All Named Query Options

4.10.8.1 Removing Query Options

To remove the query options installed under a particular name, send a DELETE request to the `/config/query` service with a URL of the form:

```
http://host:port/version/config/query/name
```

Where *name* is the name of the query options to remove. MarkLogic Server responds with 204 if the option set is successfully deleted.

4.10.8.2 Removing a Single Option

To remove a specific setting in a set of query options, send a DELETE request to the `/config/query` service with a URL of the form:

```
http://host:port/version/config/query/name/option_name
```

Where *option_name* is the name of the option to remove and *name* is the name of the containing query options. MarkLogic Server responds with 204 if the option is successfully deleted.

The *option_name* portion of the URL must be the name of an option that can appear as an immediate child of a query options XML node or JSON object. Finer grained access is not supported. All options with this name are removed. For example, if the query options named "my-options" contain multiple `<constraint/>` options, then the following request removes all of them:

```
DELETE http://localhost:8000/LATEST/config/query/my-options/constraint
```

For details on query options names and structure, see Appendix: Query Options Reference in the *Search Developer's Guide*.

If query option validation is enabled, the request will fail if it results in invalid query options. If the request fails, the options are unchanged. Option validation is enabled by default. You can disable option validation using the validate-options instance configuration property. For details, see "Configuring Instance Properties" on page 47.

For details on query option names, see Appendix: Query Options Reference in the *Search Developer's Guide*.

4.10.8.3 Removing All Named Query Options

To remove all named query options from a REST Client API instance, send a DELETE request to the `/config/query` service with a URL of the form:

```
http://host:port/version/config/query
```

MarkLogic Server responds with 204 if the options successfully deleted.

4.11 Using Namespace Bindings

> **Note:** The `/config/namespaces` service is deprecated. You should use the Management REST API to manage namespace bindings instead.

This sections covers the following topics:

- When Do You Need a Namespace Binding
- Creating or Updating a Namespace Binding
- Creating or Updating Multiple Namespace Bindings
- Listing Available Namespace Bindings
- Deleting Namespace Bindings

4.11.1 When Do You Need a Namespace Binding

Note: The `/config/namespaces` service is deprecated. Use the REST Management API to manage namespace bindings instead. See the `namespaces` property of `PUT:/manage/LATEST/servers/[id-or-name]/properties` and `GET:/manage/LATEST/servers/[id-or-name]/properties`.

Use the `/config/namespaces` service to pre-define namespace prefixes for contexts in which you cannot define a namespace binding in the request.

4.11.2 Creating or Updating a Namespace Binding

Note: The `/config/namespaces` service is deprecated. Use the REST Management API to manage namespace bindings instead. See the `namespaces` property of `PUT:/manage/LATEST/servers/[id-or-name]/properties`.

This section describes how to create a single namespace binding. You can also define multiple bindings in a single request; for details, see “Creating or Updating Multiple Namespace Bindings” on page 220.

To define a namespace binding, send a PUT request to the `/config/namespaces/{name}` service with a URL of the form:

```
http://host:port/version/config/namespaces/name
```

Where *name* is the namespace prefix you want to define. If a binding already exists for this prefix, it is replaced.

When constructing your request:

1. Set the *name* portion of the URL to the desired namespace prefix.

2. Place the XML or JSON binding definition in the request body. See the table below.

3. Specify the content type of the body in the `format` parameter or the HTTP `Content-type` header, as described in “Controlling Input and Output Content Type” on page 31. You may only send XML or JSON. The default format is XML.

The request body must have the following form:

Format	Body

XML	`<namespace xmlns="http://marklogic.com/rest-api">`
	`  <prefix>`*the_prefix*`</prefix>`
	`  <uri>`*the_uri*`</uri>`
	`</namespace>`

JSON	`{`
	`  "prefix" : "`*the_prefix*`",`
	`  "uri" : "`*the_uri*`"`
	`}`

The following example binds the prefix “bill” to the namespace URI “http://marklogic.com/examples/shakespeare”:

```
$ cat ns-binding.xml

<namespace xmlns="http://marklogic.com/rest-api">
  <prefix>bill</prefix>
  <uri>http://marklogic.com/examples/shakespeare</uri>
</namespace>

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -d@./ns-binding.xml \
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/config/namespaces/bill
```

You can examine the binding by sending a GET request to `/config/namespaces/{name}` or to `/config/namespaces`. The latter lists all bindings. For example:

```
$  curl --anyauth --user user:password -X GET \
    http://localhost:8000/LATEST/config/namespaces/bill

<rapi:namespace xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:prefix>bill</rapi:prefix>
  <rapi:uri>http://marklogic.com/examples/shakespeare</rapi:uri>
</rapi:namespace>
```

To get the equivalent output as JSON, use the `format` parameter or specify `application/json` in the HTTP Accept header. For example:

```
$  curl --anyauth --user user:password -X GET \
    -H "Accept: application/json" \
    http://localhost:8000/LATEST/config/namespaces/bill
```

```
{
  "prefix":"bill",
  "uri":"http://marklogic.com/examples/shakespeare"
}
```

4.11.3 Creating or Updating Multiple Namespace Bindings

Note: The `/config/namespaces` service is deprecated. You should use the REST Management API to manage namespace bindings instead. For details, see `PUT:/manage/v2/servers/[id-or-name]/properties`.

This section describes how to define multiple namespace bindings with a single request. You can also define a specific single binding; for details, see “Creating or Updating a Namespace Binding” on page 218.

To define multiple namespace bindings, send a PUT or POST request to the `/config/namespaces` service with a URL of the form:

```
http://host:port/version/config/namespaces
```

When constructing your request:

1. Choose PUT to replace all bindings with those the request body. Choose POST to append to existing bindings.

2. Place the XML or JSON binding definitions in the request body. See the table below.

3. Specify the content type of the body in the `format` parameter or the HTTP `Content-type` header, as described in “Controlling Input and Output Content Type” on page 31. You may only send XML or JSON. The default format is XML.

If you use POST and a binding already exists for one defined in the request body, MarkLogic Server returns status 400 (Bad Request).

The request body must have the following form:

Format	Body
XML	<namespace-bindings xmlns="http://marklogic.com/rest-api"> <namespace> <prefix>*a_prefix*</prefix> <uri>*a_namespace_uri*</uri> </namespace> <namespace> <prefix>*another_prefix*</prefix> <uri>*another_namespace_uri*</uri> </namespace> </namespace-bindings>

JSON	{
	  "namespace-bindings": [
	  {
	  "prefix": "*a_prefix*",
	  "uri": "*a_namespace_uri*"
	  },
	  {
	  "prefix": "*another_prefix*",
	  "uri": "*another_namespace_uri*"
	  }
	  ]
	}

The following example binds the prefix “one” to the namespace URI “http://marklogic.com/examples/one” and the prefix “two” to the namespace URI “http://marklogic.com/examples/two”:

```
$ cat ns-bindings.xml
<namespace-bindings xmlns="http://marklogic.com/rest-api">
  <namespace>
    <prefix>one</prefix>
    <uri>http://marklogic.com/examples/one</uri>
  </namespace>
  <namespace>
    <prefix>two</prefix>
    <uri>http://marklogic.com/examples/two</uri>
  </namespace>
</namespace-bindings>

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -d@./ns-bindings.xml \
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/config/namespaces
```

The following example creates equivalent bindings using JSON input:

```
$ cat ns-bindings.json
{
  "namespace-bindings": [
    {
      "prefix": "one",
      "uri": "http:\/\/marklogic.com\/examples\/one"
    },
    {
      "prefix": "two",
      "uri": "http:\/\/marklogic.com\/examples\/two"
    }
  ]
}

$ curl --anyauth --user user:password -X PUT -d@./ns-bindings.json \
    -H "Content-type: application/json" \
    http://localhost:8000/LATEST/config/namespaces
```

You can examine the binding by sending a GET request to `/config/namespaces/{name}` or to `/config/namespaces`.

4.11.4 Listing Available Namespace Bindings

Note: The `/config/namespaces` service is deprecated. You should use the REST Management API to manage namespace bindings instead. For details, see `GET:/manage/v2/servers/[id-or-name]/properties`.

To list all available namespace bindings, send a GET request to /config/namespaces with a URL of the form:

```
http://host:port/version/config/namespaces
```

To retrieve the binding for a single namespace prefix, send a GET request to /config/namespaces/{name} with a URL of the form:

```
http://host:port/version/config/namespaces/name
```

Where *name* is a bound namespace prefix.

You can request output as either XML or JSON. Use the Accept header or `format` request parameter to select the output content type. For details, see “Controlling Input and Output Content Type” on page 31.

The following example command requests all namespace bindings, as XML:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
    http://localhost:8000/LATEST/config/namespaces
```

The output from the command is shown in the table below, assuming two namespace bindings are installed, for the prefixes “one” and “two”.

Format	Example Output
XML	<rapi:namespace-bindings xmlns:rapi="http://marklogic.com/rest-api"> <rapi:namespace> <rapi:prefix>one</rapi:prefix> <rapi:uri>http://marklogic.com/examples/one</rapi:uri> </rapi:namespace> <rapi:namespace> <rapi:prefix>two</rapi:prefix> <rapi:uri>http://marklogic.com/examples/two</rapi:uri> </rapi:namespace> </rapi:namespace-bindings>

JSON	{
	"namespace-bindings": [
	{
	"prefix": "one",
	"uri": "http://marklogic.com/examples/one"
	},
	{
	"prefix": "two",
	"uri": "http://marklogic.com/examples/two"
	}
	]
	}

When you use `GET /config/namespaces/{name}`, the output is similar, but without the namespace-bindings wrapper. For example to retrieve the binding for the “one” prefix:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
    http://localhost:8000/LATEST/config/namespaces/one

<rapi:namespace xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:prefix>one</rapi:prefix>
  <rapi:uri>http://marklogic.com/examples/one</rapi:uri>
</rapi:namespace>
```

The following command retrieves the same information as JSON:

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/json" \
    http://localhost:8000/LATEST/config/namespaces/one
```

```
{"prefix":"one","uri":"http://marklogic.com/examples/one"}
```

4.11.5 Deleting Namespace Bindings

Note: The `/config/namespaces` service is deprecated. You should use the REST Management API to manage namespace bindings instead. For details, see `PUT:/manage/v2/servers/[id-or-name]/properties`.

To remove all namespace bindings, send a DELETE request to /config/namespaces with a URL of the form:

```
http://host:port/version/config/namespaces
```

To remove a specific binding, send a DELETE request to /config/namespaces/{name} with a URL of the form:

```
http://host:port/version/config/namespaces/name
```

Where *name* is a bound namespace prefix. If no binding exists for name, MarkLogic Server returns status 404 (Not Found).

The following example deletes just the binding for the prefix “one”:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X DELETE \
    http://localhost:8000/LATEST/config/namespaces/one
```

4.12 Generating Search Facets

The MarkLogic Server Search API enables you to expose search facets in your application. Facets enable users to filter search results by narrowing down the search criteria. To learn more about facets, see Constrained Searches and Faceted Navigation in the *Search Developer’s Guide*.

To generate facet information in search results, use query options that include constraints that support facets, such as collection and element constraints. You can also define custom constraints; see Creating a Custom Constraint in the *Search Developer’s Guide*.

The following example returns facet information about play types and enables searching by the facet “type”, assuming the documents in the database have been added to the collections `/play-type/Comedy`, `/play-type/Tragedy`, and `/play-type/History`. For more examples, see Constraint Options in the *Search Developer’s Guide*.

Note: This example uses a collection constraint, which requires the collection lexicon to be enabled on the database.

To enable the play type facet, create query options that define a collection constraint:

Format	Query Options
XML	<options xmlns="http://marklogic.com/appservices/search"> <constraint name="type"> <collection prefix="/play-type/"/> </constraint> <return-facets>true</return-facets> </options>

JSON	{
	  "options": {
	  "constraint": [
	  {
	  "name": "type",
	  "collection": { "prefix": "\/play-type\/" }
	  }],
	  "return-facets": true
	  }
	}

Install the options with the name “facet-options”:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    -d@"./facet-options.xml" -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/config/query/facet-options
```

If you query `/search` or `/qbe` using “index-options” as the query options, facet results are included in the search results. For example, the following query finds all occurrences of “castle” and the search response includes a count of the matches for each play type:

```
$ curl --anyauth --user user:password -X GET \
  'http://localhost:8000/LATEST/search?options=facet-options&q=castle'
...
<search:response snippet-format="snippet" total="88" ...>
  <search:result />
  <search:facet name="type" type="collection">
    <search:facet-value name="Comedy" count="5">
      Comedy
    </search:facet-value>
    <search:facet-value name="History" count="35">
      History
    </search:facet-value>
    <search:facet-value name="Tragedy" count="47">
      Tragedy
    </search:facet-value>
  </search:facet>
...
</search:response>
```

To get the equivalent results as JSON, use `format=json` or include `application/json` in the HTTP Accept header. For example:

```
$ curl --anyauth --user user:password -X GET \
  -H "Accept: application/json" \
  'http://localhost:8000/LATEST/search?options=facet-options&q=castle'
...
{
  "snippet-format": "snippet",
  "total": 88,
  "start": 1,
  "page-length": 10,
  "results": [...],
  "facets": {
    "type": {
      "type": "collection",
      "facetValues": [
        {
          "name": "Comedy",
          "count": 5
        },
        {
          "name": "History",
          "count": 35
        },
        {
          "name": "Tragedy",
          "count": 47
        }
      ]
    }
  },
  "qtext": "castle",
  ...
}
```

You can query by facet by including a `facet-name:facet-value` search term. The following command matches occurrences of “castle” in plays of type Comedy:

```
$ curl --anyauth --user user:password -X GET \
    'http://localhost:8000/LATEST/search?options=facet-options&q=castl
e type:Comedy'
```

4.13 Paginating Results

When you query the database, you can paginate the query results using the `start` and `pageLength` request parameters. Use `start` to specify the index of the first result to return, and `pageLength` to control the number results to return.

By default, queries return the first 10 results. That is, the default start position is 1 and the default page length is 10. You can fetch successive non-overlapping pages of results by incrementing the start position by the page length in each call.

For more information, see the Search Developer's Guide and Fast Pagination and Unfiltered Searches in the *Scalability, Availability, and Failover Guide*.

The following example command fetches the first 5 results matching "castle". Notice that the search response includes the total number of matches and each `search:result` includes an index.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
   'http://localhost:8000/LATEST/search?q=castle&start=1&pageLength=5'
...
<search:response snippet-format="snippet" total="88"
    start="1" page-length="5" ...>
  <search:result index="1" uri="/shakespeare/plays/hen_vi_2.xml" .../>
  <search:result index="2" uri="/shakespeare/plays/rich_ii.xml" .../>
  <search:result index="3" uri="/shakespeare/plays/macbeth.xml" .../>
  <search:result index="4" uri="/shakespeare/plays/hen_vi_3.xml" .../>
  <search:result index="5" uri="/shakespeare/plays/othello.xml" .../>
...
</search:response>
```

To fetch the next 5 results, increment start by 5:

```
$ curl --anyauth --user user:password -X GET \
   'http://localhost:8000/LATEST/search?q=castle&start=6&pageLength=5'
...
<search:response snippet-format="snippet" total="88"
    start="1" page-length="5" ...>
  <search:result index="6" uri="/shakespeare/plays/lear.xml" .../>
  <search:result index="7" uri="/shakespeare/plays/hamlet.xml" .../>
  <search:result index="8" uri="/shakespeare/plays/rich_iii.xml" .../>
  <search:result index="9" uri="/shakespeare/plays/hen_v.xml" .../>
  <search:result index="10" uri="/shakespeare/plays/m_wives.xml" .../>
...
</search:response>
```

4.14 Customizing Search Results

This section covers several features that the REST Client API provides for search result customization.

- The `transform-results` query option enables you to fine tune the default snippets. For example, you can control how many matches to return.
- Custom snippet extensions identified in query options enable you to modify the contents of snippets returned in the `search:match` portion of a `search:response`.
- Transform functions enable you to completely change the structure returned by a search or values query. Your transform takes a `search:response` or a matched document as input and produces a result document.

- The `extract-document-data` query option enables you to return a portion of each matching document. For details, see Extracting a Portion of Matching Documents in the *Search Developer's Guide*.

The following topics cover these features:

- Customizing Search Snippets
- Transforming the Search Response

4.14.1 Customizing Search Snippets

Search results usually include portions of matching documents with the search matches highlighted, perhaps with some text showing the context of the search matches. These search result pieces are known as *snippets*. MarkLogic Server has a default search snippet format, but you can customize the snippet format by either modifying the configuration of the default snippet function or creating a custom snippet extension.

This section covers the following topics:

- Customizing the Default Search Snippets
- Creating Your Own Snippet Extension
- Generating Custom JSON Snippets

4.14.1.1 Customizing the Default Search Snippets

MarkLogic Server creates snippets by applying a default transformation to search results. You can modify the results of the default transformation using the `transform-results` query option, as described in Modifying Your Snippet Results in the *Search Developer's Guide*. To use this feature with the REST API:

1. Create query options that contain a `transform-results` XML element or JSON object.

2. Install these query options under a name or as the default options using the `/config/query` service, or include them directly in a combined query. For details, see "Configuring Query Options" on page 208.

3. If you installed the configuration as named query options, apply the options to your query by supplying the name in the `options` request parameter. For details, see "Controlling Queries With Options" on page 208.

4. If you installed the configuration as default options, they will be automatically applied to search results returned by any query that does not use named query options.

The following example searches for the term "hamlet" in a collection of Shakespeare plays using the default snippet options. The search results include four snippets in each match.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
    http://localhost:8000/LATEST/search?q=hamlet

<search:response snippet-format="snippet" total="20" ...>
  <search:result index="1" uri="/shakespeare/plays/hamlet.xml" ...>
    <search:snippet>
      <search:match
        path="fn:doc(&quot;/shakespeare/plays/hamlet.xml&quot;)/PLAY/T
ITLE">The Tragedy of <search:highlight>Hamlet</search:highlight>,
Prince of Denmark
      </search:match>
      <search:match
        path="fn:doc(&quot;/shakespeare/plays/hamlet.xml&quot;)/PLAY/P
ERSONAE/PERSONA[2]"><search:highlight>HAMLET</search:highlight>, son
to the late, and nephew to the present king.
      </search:match>
      <search:match
        path="fn:doc(&quot;/shakespeare/plays/hamlet.xml&quot;)/PLAY/P
ERSONAE/PERSONA[4]">HORATIO, friend to
<search:highlight>Hamlet</search:highlight>.
      </search:match>
      <search:match
        path="fn:doc(&quot;/shakespeare/plays/hamlet.xml&quot;)/PLAY/P
ERSONAE/PERSONA[16]">GERTRUDE, queen of Denmark, and mother to
<search:highlight>Hamlet</search:highlight>.
      </search:match>
    </search:snippet>
  </search:result>
...
</search:response>
```

If you request results as JSON, you get snippets of the following form instead of the XML shown above.

```
...
"matches": [
  {
    "path":
      "fn:doc(\"\/shakespeare\/plays\/hamlet.xml\")\/PLAY\/TITLE",
    "match-text": [
      "The Tragedy of ",
      { "highlight": "Hamlet" },
      ", Prince of Denmark"
    ]
  },
]
...
```

Notice that the resulting snippets are from the `<TITLE/>` and `<PERSONAE/>` elements. If you apply the options below to the search instead, at most three snippets are returned and only matches in `<LINE/>` elements are included.

Format	Query Options

XML	`<options xmlns="http://marklogic.com/appservices/search">`
	`  <transform-results apply="snippet">`
	`  <max-matches>3</max-matches>`
	`  <preferred-matches>`
	`  <element ns="" name="LINE" />`
	`  </preferred-matches>`
	`  </transform-results>`
	`  <search-option>filtered</search-option>`
	`</options>`

JSON	`{`
	`  "options": {`
	`  "transform-results": {`
	`  "apply": "snippet",`
	`  "max-matches": 3,`
	`  "preferred-matches": {`
	`  "element": [{`
	`  "ns": "",`
	`  "name": "LINE"`
	`  }]`
	`  }`
	`  },`
	`  "search-option": ["filtered"]`
	`  }`
	`}`

This example installs the above XML options and applies them to the same query:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    -d @./transform-results.xml \
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/config/query/snippet-lines

$ curl --anyauth --user user:password -X GET \
   -H "Accept: application/xml" \
   http://localhost:8000/LATEST/search?q=hamlet&options=snippet-lines

<search:response snippet-format="snippet" total="20" ...>
  <search:result index="1" uri="/shakespeare/plays/hamlet.xml" ...>
    <search:snippet>
      <search:match
        path="fn:doc(&quot;/shakespeare/plays/hamlet.xml&quot;)/PLAY/A
```

```
CT[1]/SCENE[1]/SPEECH[48]/LINE[6]">Dared to the combat; in which our
valiant <search:highlight>Hamlet</search:highlight>--
      </search:match>
      <search:match
        path="fn:doc(&quot;/shakespeare/plays/hamlet.xml&quot;)/PLAY/A
CT[1]/SCENE[1]/SPEECH[48]/LINE[17]">His fell to
<search:highlight>Hamlet</search:highlight>. Now, sir, young
Fortinbras,
      </search:match>
      <search:match
        path="fn:doc(&quot;/shakespeare/plays/hamlet.xml&quot;)/PLAY/A
CT[1]/SCENE[1]/SPEECH[59]/LINE[6]">Unto young
<search:highlight>Hamlet</search:highlight>; for, upon my life,
      </search:match>
    </search:snippet>
  </search:result>
...
</search:response>
```

To install the equivalent JSON options, set the Content-type header to `application/json`. For example:

```
$ curl --anyauth --user user:password -X PUT \
    -d @./transform-results.json \
    -H "Content-type: application/json" \
    http://localhost:8000/LATEST/config/query/snippet-lines
```

To receive JSON search results, set the Accept header to `application/json` or use the `format` request parameter. For example:

```
$ curl --anyauth --user user:password -X GET \
   -H "Accept: application/json" \
   'http://localhost:8000/LATEST/search?q=hamlet&options=snippet-lines'

{
  "snippet-format": "snippet",
  "total": 20,
  "start": 1,
  "page-length": 10,
  "results": [
    {
      "index": 1,
      "uri": "\/shakespeare\/plays\/hamlet.xml",
      "path": "fn:doc(\"\/shakespeare\/plays\/hamlet.xml\")",
      "score": 158720,
      "confidence": 0.80079,
      "fitness": 1,
      "matches": [
        {
          "path":
"fn:doc(\"\/shakespeare\/plays\/hamlet.xml\")\/PLAY\/ACT[1]\/SCENE[1]\
/SPEECH[48]\/LINE[6]",
          "match-text": [
```

```
            "Dared to the combat; in which our valiant ",
            {
              "highlight": "Hamlet"
            },
            "--"
          ]
        },
        {
          "path":
"fn:doc(\"\/shakespeare\/plays\/hamlet.xml\")\/PLAY\/ACT[1]\/SCENE[1]\
/SPEECH[48]\/LINE[17]",
          "match-text": [
            "His fell to ",
            {
              "highlight": "Hamlet"
            },
            ". Now, sir, young Fortinbras,"
          ]
        },
        {
          "path":
"fn:doc(\"\/shakespeare\/plays\/hamlet.xml\")\/PLAY\/ACT[1]\/SCENE[1]\
/SPEECH[59]\/LINE[6]",
          "match-text": [
            "Unto young ",
            {
              "highlight": "Hamlet"
            },
            "; for, upon my life,"
          ]
        }
      ]
    }
  ],
  ...
}
```

Alternatively, you can use a combined query that contains your options, instead of installing persistent options. For example, you can pass the following query in the body of a `POST:/v1/search` request. For details, see “Generating a Combined Query from a QBE” on page 172.

```
<search xmlns="http://marklogic.com/appservices/search">
  <qtext>hamlet</qtext>
  <options>
    <transform-results apply="snippet">
      <max-matches>3</max-matches>
      <preferred-elements>
        <element ns="" name="LINE" />
      </preferred-elements>
    </transform-results>
    <search-option>filtered</search-option>
```

```
  </options>
</search>
```

4.14.1.2 Creating Your Own Snippet Extension

If the `transform-results` query options with the default snippet format does not meet the needs of your application, you can create a custom snippet transformation function, as described in Specifying Your Own Code in transform-results in the *Search Developer's Guide*.

Install custom snippet transformations using the `/ext` service to load them into the Modules database associated with your REST API instance. Use the `apply`, `at`, and `ns` attributes of the `transform-results` query option to specify your custom module.

To create and use a custom snippeting function:

1. Create an XQuery library module that implements your custom snippet function, as described in Specifying Your Own Code in transform-results in the *Search Developer's Guide*.

2. Install your module in the modules database of your REST API instance using the `/ext` service, similar to installing a dependent library for a resource service extension. For details, see "Installing or Updating an Asset" on page 377.

3. Create and install query options that include a `transform-results` option that uses your custom snippeting function. For details, see Search Customization Via Options and Extensions in the *Search Developer's Guide*.

4. Apply the options to your query by supplying the name of the options with the `options` request parameter. For details, see "Controlling Queries With Options" on page 208.

If your application requires non-trivial JSON snippet customization, your snippet function must generate the XML representation of the desired JSON. For details, see "Generating Custom JSON Snippets" on page 238.

The following example builds on the example from "Customizing the Default Search Snippets" on page 228. The example searches for the term "hamlet" in a collection of Shakespeare plays using the default snippet format with the custom options that extract the first three matches in lines of dialog. These options are installed under the name "snippet-lines". With the default snippet transformation function, the search produces the following results:

```
# Install the options
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    -d @./transform-results.xml \
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/config/query/snippet-lines

# Use the options in a query
$ curl --anyauth --user user:password -X GET \
```

```
   -H "Accept: application/xml" \
   'http://localhost:8000/LATEST/search?q=hamlet&options=snippet-lines'

<search:response snippet-format="snippet" total="20" ...>
  <search:result index="1" uri="/shakespeare/plays/hamlet.xml" ...>
    <search:snippet>
      <search:match
        path="fn:doc(&quot;/shakespeare/plays/hamlet.xml&quot;)/PLAY/A
CT[1]/SCENE[1]/SPEECH[48]/LINE[6]">Dared to the combat; in which our
valiant <search:highlight>Hamlet</search:highlight>--
      </search:match>
      <search:match
        path="fn:doc(&quot;/shakespeare/plays/hamlet.xml&quot;)/PLAY/A
CT[1]/SCENE[1]/SPEECH[48]/LINE[17]">His fell to
<search:highlight>Hamlet</search:highlight>. Now, sir, young
Fortinbras,
      </search:match>
      <search:match
        path="fn:doc(&quot;/shakespeare/plays/hamlet.xml&quot;)/PLAY/A
CT[1]/SCENE[1]/SPEECH[59]/LINE[6]">Unto young
<search:highlight>Hamlet</search:highlight>; for, upon my life,
      </search:match>
    </search:snippet>
  </search:result>
...
</search:response>
```

The custom snippeting function below returns the act, scene, speech and line number in each match, instead of the text surrounding the search term.

```
xquery version "1.0-ml";

module namespace example = "http://marklogic.com/example";

import module namespace search =
  "http://marklogic.com/appservices/search"
  at "/MarkLogic/appservices/search/search.xqy";


declare function example:snippet(
   $result as node(),
   $ctsquery as schema-element(cts:query),
   $options as element(search:transform-results)?
) as element(search:snippet)
{
  let $default-snippet := search:snippet($result, $ctsquery, $options)
  return element
    { fn:QName(fn:namespace-uri($default-snippet),
               fn:name($default-snippet)) }
    { $default-snippet/@*,
      for $child in $default-snippet/node()
      return
        if ($child instance of element(search:match))
```

```
            then element {fn:QName(fn:namespace-uri($child),
                          fn:name($child)) } {
              $child/@*,
              let $parts :=
                fn:tokenize(
                  fn:substring-after($child/@path, "/PLAY/ACT["), "\[")
              let $location := fn:concat(
                "Act ", fn:substring-before($parts[1], "]"),
                ", Scene ", fn:substring-before($parts[2], "]"),
                ", Speech ", fn:substring-before($parts[3], "]"),
                ", Line ", fn:substring-before($parts[4], "]"))
              return text {$location}
            }
            else $child
        }
    };
```

If the above module is installed in the instance modules database with the URI `/ext/my.domain/my-snippets.xqy`, then we can use the following options to generate custom snippets by supplying the local name of the custom function (`snippet`) in `apply`, the namespace of the module (`http://marklogic.com/example`) in `ns`, and the URI of the module in the instance modules database (`/my.domain/my-snippets.xqy`) in `at`.

Format	Query Options

XML	<options xmlns="http://marklogic.com/appservices/search">
	  <transform-results **apply="snippet"**
	  ns="http://marklogic.com/example"
	  at="/ext/my.domain/my-snippets.xqy">
	  <max-matches>3</max-matches>
	  <preferred-elements>
	  <element ns="" name="LINE" />
	  </preferred-elements>
	  </transform-results>
	  <search-option>filtered</search-option>
	</options>

JSON	{
	  "options": {
	  "transform-results": {
	  "apply": "snippet",
	  "ns": "http://marklogic.com/example",
	  "at": "/ext/my.domain/my-snippets.xqy",
	  "max-matches": 3,
	  "preferred-matches": {
	  "element": [{
	  "ns": "",
	  "name": "LINE"
	  }]
	  }
	  },
	  "search-option": ["filtered"]
	  }
	}

This example installs the above XML options and applies them to the query for “hamlet” to produce snippets that display the act, scene, speech and dialog number of each match:

```
# Windows users, see Modifying the Example Commands for Windows

# Install the custom snippeting module
$ curl --anyauth --user user:password -X PUT \
    -d @./my-snippets.xqy \
    -H "Content-type: application/xquery" \
    http://localhost:8000/LATEST/ext/my.domain/my-snippets.xqy
```

```
# Install the options
$ curl --anyauth --user user:password -X PUT \
    -d @./transform-results.xml \
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/config/query/custom-snippet

# Use the options in a query
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
    "http://localhost:8000/LATEST/search?q=hamlet&options=custom-snipp
et"

<search:response snippet-format="custom" total="20" start="1" ...>
  <search:result index="1" uri="/shakespeare/plays/hamlet.xml"
      path="fn:doc(&quot;/shakespeare/plays/hamlet.xml&quot;)" ...>
    <search:snippet>
      <search:match
        path="fn:doc(&quot;/shakespeare/plays/hamlet.xml&quot;)/PLAY/A
CT[1]/SCENE[1]/SPEECH[48]/LINE[6]">Act 1, Scene 1, Speech 48, Line 6
      </search:match>
      <search:match
        path="fn:doc(&quot;/shakespeare/plays/hamlet.xml&quot;)/PLAY/A
CT[1]/SCENE[1]/SPEECH[48]/LINE[17]">Act 1, Scene 1, Speech 48, Line 17
      </search:match>
      <search:match
        path="fn:doc(&quot;/shakespeare/plays/hamlet.xml&quot;)/PLAY/A
CT[1]/SCENE[1]/SPEECH[59]/LINE[6]">Act 1, Scene 1, Speech 59, Line 6
      </search:match>
    </search:snippet>
  </search:result>
  ...
</search:response>
```

To generate the equivalent results in JSON, change the Accept header to `application/json`. For example:

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/json" \
    "http://localhost:8000/LATEST/search?q=hamlet&options=custom-snipp
et"

{
  "snippet-format": "custom",
  "total": 20,
  "start": 1,
  "page-length": 10,
  "results": [
    {
      "index": 1,
      "uri": "/shakespeare/plays/hamlet.xml",
      "path": "fn:doc(\"/shakespeare/plays/hamlet.xml\")",
      "score": 158720,
      "confidence": 0.80079,
```

```
      "fitness": 1,
      "matches": [
        {
          "path":
"fn:doc(\"\/shakespeare/plays/hamlet.xml\")/PLAY/ACT[1]/SCENE[1]/SPEEC
H[48]/LINE[6]",
          "match-text": [
            "Act 1, Scene 1, Speech 48, Line 6"
          ]
        },
        {
          "path":
"fn:doc(\"\/shakespeare\/plays\/hamlet.xml\")/PLAY/ACT[1]/SCENE[1]/SPE
ECH[48]/LINE[17]",
          "match-text": [
            "Act 1, Scene 1, Speech 48, Line 17"
          ]
        },
        {
          "path":
"fn:doc(\"/shakespeare/plays/hamlet.xml\")/PLAY/ACT[1]/SCENE[1]/SPEECH
[59]/LINE[6]",
          "match-text": [
            "Act 1, Scene 1, Speech 59, Line 6"
          ]
        }
      ]
    }
  ],
  ...
}
```

4.14.1.3 Generating Custom JSON Snippets

If you need to create custom JSON snippets that cannot be expressed as XML, set `@format` to `"json"` on the `search:snippet` you return, and populate the `search:snippet` with serialized JSON. For example:

```
declare function my:snippeter(
   $result as node(),
   $ctsquery as schema-element(cts:query),
   $options as element(search:transform-results)?
) as element(search:snippet) {
    element search:snippet {
        attribute format { "json" },
        text {'{"MY":"CUSTOM SNIPPET"}'}
    }
};
```

You cannot change just a portion of the snippet to serialized JSON. For example, you cannot just change the `search:match` text.

4.14.2 Transforming the Search Response

You can make arbitrary changes to the response from a search or values query by applying a transformation function to the response.

Search response transforms use the same interface and framework as content transformations applied during document ingestion. For details on the interface and on installing a transform, see “Working With Content Transformations” on page 323.

This following topics cover additional details specific to working with search transforms:

- Basic Search Transform Usage
- What to Expect as Input Content
- Expected Output

4.14.2.1 Basic Search Transform Usage

To use a transform function:

1. Create a transform function according to the interface described in “Writing Transformations” on page 323. For additional details, see “What to Expect as Input Content” on page 239.

2. Install your transform function on the REST API instance following the instructions in “Installing Transformations” on page 331.

3. Apply your transform to a request to the `/search` or `/values` service by specifying the name in the `transform` request parameter, as described in “Applying Transformations” on page 333.

For example, assuming you installed a transform function under the name “example”, the following command applies the function to the results of a search:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
   'http://localhost:8000/LATEST/search?q=dog&transform=example'
```

4.14.2.2 What to Expect as Input Content

Search response transforms use the interface and framework described in “Working With Content Transformations” on page 323. A search transform can be invoked on either a database document or a search results summary (“search response”), depending on the search request context. Any customizations made by the `transform-results` query option or result decorators are applied before calling your transform function.

Your transform should be prepared to receive multiple documents types in the `content` parameter:

- When the input to a transform is a document matched by the search, the `content` parameter is the matched document, so it may be any supported document type (XML, JSON, Text, Binary).
- When the input is a search response, the type of the document in the `content` parameter can be either XML or JSON, depending on the result format requested by the client.

  When the search response is expressed as XML, the input document node has a `<search:response/>` root element, in the `content` parameter.

For example, suppose you send a query to `/search` that finds matches in 1 XML document and 1 JSON document. The following table summarizes the number of times your transforms is invoked and with what content, in several request contexts. (The order of invocation is not guaranteed.)

Search Request Context	Requested Result Format	Transform Invocations
Simple search, returning only a search response	XML	`transform(`*`searchResponseAsXML`*`)`
	JSON	`transform(`*`searchResponseAsJSON`*`)`
Multi-document read with no search response	XML or JSON	`transform(`*`matchedXMLDocument`*`)` `transform(`*`matchedJSONDocument`*`)`

Multi-document read with search response	XML	`transform(`*`searchResponseAsXML`*`)`
		`transform(`*`matchedXMLDocument`*`)`
		`transform(`*`matchedJSONDocument`*`)`

	JSON	`transform(`*`searchResponseAsJSON`*`)`
		`transform(`*`matchedXMLDocument`*`)`
		`transform(`*`matchedJSONDocument`*`)`

You can probe the document type to test whether the input to your transform receives JSON or XML input. For example, in server-side JavaScript, you can test the `documentFormat` property of a document node:

```
function myTransform(context, params, content) {
  if (content.documentFormat == "JSON") {
    // handle as JSON or a JavaScript object
  } else {
    // handle as XML
}

  ...
}
```

In XQuery and XSLT, you can test the node kind of the root of the document, which will be `element` for XML and `object` for JSON.

```
declare function dumper:transform(
  $context as map:map,
  $params as map:map,
  $content as document-node()
) as document-node()
{
  if (xdmp:node-kind($content/node() eq "element")
  then(: process as XML :)
  else (: process as JSON :)
```

As with read and write transforms, the content object is immutable in JavaScript, so you must call toObject to create a mutable copy:

```
var output = content.toObject();
...modify output...
return output;
```

4.14.2.3 Expected Output

A search transform function is expected to return a document node of the same type as the document node passed in via the `content` parameter. If your transform returns a document node that is not the same type, set the new output type on the `context` parameter.

The type of document returned must be consistent with the `output-type` context value in an XQuery or XML transform, or the `outputType` context property in a JavaScript transform.

4.15 Generating Search Term Completion Suggestions

Use the `/suggest` service to generate search term completion suggestions that match a wildcard terminated string. For example, if the user enters the text “doc” into a search box, you can query `/suggest` with “doc” to retrieve a list of terms matching “doc*”, and then display them to user. This service is analogous to calling the XQuery function `search:suggest`.

The following topics are covered:

- Basic Steps
- Example: Generating Search Suggestions Using GET
- Example: Generating Search Suggestions Using POST
- Where to Find More Information

4.15.1 Basic Steps

To retrieve a list of search suggestions using the REST Client API, use the following procedure. For a detailed example, see "Example: Generating Search Suggestions Using GET" on page 242.

1. Configure at least one index on the XML element, XML attribute, or JSON property you want to include in the search for suggestions. For performance reasons, a range or collection index is recommended over a word lexicon; for details, see `search:suggest`.

2. Define query options that use your index as a suggestion source by including it in the definition of a `default-suggestion-source` or `suggestion-source option`. For details, see Search Term Completion Using search:suggest in the *Search Developer's Guide*.

3. Send a GET or POST request to the `/suggest` service with a URL of the following form, where `partial-q` is the text for which to retrieve suggestions.

```
http://host:port/version/suggest?partial-q=text_to_match&options=your_
options
```

If you use a GET request to generate suggestions, you must pre-install the query options. For details, see "Configuring Query Options" on page 208.

If you use a POST request to generate suggestions, the POST body must contains a combined query. You can use persistent query options and/or include dynamic query options in the combined query. For details, see "Specifying Dynamic Query Options with Combined Query" on page 179.

If you define your suggestion source in the default query options, you can omit the `options` request parameter or dynamic query options.

You can further constrain the suggestions by including one or more additional queries using the `q` request parameter or in the combined query of a POST request body. The request returns only suggestions in documents that also match the additional queries. Multiple additional queries are AND'd together. The following example URL returns only those suggestions found in fragments that match the query `prefix:xdmp`.

```
http://localhost:8000/LATEST/suggest/partial-q=doc&q=prefix:xdmp
```

Additional request parameters allow you to limit the number of suggestions returned and specify a substring of `partial-q` to match against. For details, see `GET:/v1/suggest` or `POST:/v1/suggest` in *MarkLogic REST API Reference*.

4.15.2 Example: Generating Search Suggestions Using GET

This example walks you through configuring your database and REST instance to try retrieving search suggestions. The Documents database is assumed in this example, but you can use any database.

1. If you do not already have a REST API instance, create one. This example assumes your instance is on port 8000.

2. Initialize the Database.

3. Install Query Options

4. Retrieve Unconstrained Search Suggestions

5. Retrieve Constrained Search Suggestions

4.15.2.1 Initialize the Database

Run the following query in Query Console to load the sample data into your database, or use PUT or POST requests to `/documents` to create equivalent documents. The example will retrieve suggestions for the `<name/>` element, with and without a constraint based on the `<prefix/>` element.

```
xdmp:document-insert("/suggest/load.xml",
  <function xmlns="http://marklogic.com/example">
    <prefix>xdmp</prefix>
    <name>document-load</name>
  </function>
  );
xdmp:document-insert("/suggest/insert.xml",
  <function xmlns="http://marklogic.com/example">
    <prefix>xdmp</prefix>
    <name>document-insert</name>
  </function>
  );
xdmp:document-insert("/suggest/query.xml",
  <function xmlns="http://marklogic.com/example">
    <prefix>cts</prefix>
    <name>document-query</name>
  </function>
  );
xdmp:document-insert("/suggest/search.xml",
  <function xmlns="http://marklogic.com/example">
    <prefix>cts</prefix>
    <name>search</name>
  </function>
  );
```

To create the range index used by the example, run the following query in Query Console, or use the Admin Interface to create an equivalent index on the `name` element. The following query assumes you are using the Documents database; modify as needed.

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

```
  admin:database-add-range-element-index(
    admin:get-configuration(),
    xdmp:database("Documents"),
    admin:database-range-element-index(
    "string", "http://marklogic.com/example",
    "name", "http://marklogic.com/collation/", fn:false())
  )
);
```

4.15.2.2 Install Query Options

This step installs persistent query options that define `name` as the default suggestion source, plus an additional value constraint on `prefix`. Copy the following query options into a file named `suggest-options.xml`:

```
<options xmlns="http://marklogic.com/appservices/search">
 <default-suggestion-source>
   <range type="xs:string" facet="true">
      <element ns="http://marklogic.com/example" name="name"/>
   </range>
 </default-suggestion-source>
 <constraint name="prefix">
   <value>
      <element ns="http://marklogic.com/example" name="prefix"/>
   </value>
 </constraint>
</options>
```

Install the options using the `/config/query` service, as described in “Creating or Modifying Query Options” on page 210. Use a command similar to the following:

```
$ curl --anyauth --user user:password -X PUT \
    -d @./suggest-options.xml -i -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/config/query/opt-suggest
```

The database and REST API instance are now configured to retrieve the example search suggestions.

4.15.2.3 Retrieve Unconstrained Search Suggestions

The following request returns all values of `<name/>` that match the wildcard string “doc*”:

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
  'http://localhost:8000/LATEST/suggest?partial-q=doc&options=opt-sugg
est'
...
<search:suggestions xmlns:search=...>
  <search:suggestion>document-insert</search:suggestion>
  <search:suggestion>document-load</search:suggestion>
  <search:suggestion>document-query</search:suggestion>
</search:suggestions>
```

If you perform the same request with an Accept header of `application/json`, the response data looks like the following:

```
{
  "suggestions": [
    "document-insert",
    "document-load",
    "document-query"
  ]
}
```

4.15.2.4 Retrieve Constrained Search Suggestions

Recall that the query options include a value constraint on the `prefix` element:

```
<options xmlns="http://marklogic.com/appservices/search">
 <default-suggestion-source>
   <range type="xs:string" facet="true">
      <element ns="http://marklogic.com/example" name="name"/>
   </range>
 </default-suggestion-source>
 <constraint name="prefix">
   <value>
      <element ns="http://marklogic.com/example" name="prefix"/>
   </value>
 </constraint>
</options>
```

The following command uses this constraint in the additional query `prefix:xdmp` to limit results to suggestions in fragments with the `prefix` value `xdmp`. This eliminates the function `document-query` from the results because it has the prefix `cts`.

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
  'http://localhost:8000/LATEST/suggest?partial-q=doc&options=opt-sugg
est&q=prefix:xdmp'
...
<search:suggestions xmlns:search=...>
  <search:suggestion>document-insert</search:suggestion>
  <search:suggestion>document-load</search:suggestion>
</search:suggestions>
```

If you perform the same request with an Accept header of `application/json`, the response data looks like the following:

```
{
  "suggestions": [
    "document-insert",
    "document-load"
  ]
}
```

4.15.3 Example: Generating Search Suggestions Using POST

This example uses a POST request with the following combined query to limit results to suggestions in fragments with the `prefix` value `xdmp`. The example assumes your database contains the documents loaded in “Initialize the Database” on page 243.

```
<search xmlns="http://marklogic.com/appservices/search" >
  <query>
    <value-query>
      <element ns="http://marklogic.com/example" name="prefix"/>
      <text>xdmp</text>
    </value-query>
  </query>
  <options xmlns="http://marklogic.com/appservices/search">
    <default-suggestion-source>
      <range type="xs:string" facet="true">
        <element ns="" name="name"/>
      </range>
    </default-suggestion-source>
  </options>
</search>
```

Note that you do not need to pre-install persistent query options in this case because the options are included in the query.

The following command uses this query to retrieve suggestions for the input string "doc".

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
  'http://localhost:8000/LATEST/suggest?partial-q=doc
...
<search:suggestions xmlns:search=...>
  <search:suggestion>document-insert</search:suggestion>
  <search:suggestion>document-load</search:suggestion>
</search:suggestions>
```

The following query is the equivalent combined query, expressed as JSON:

```
{ "search": {
    "qtext": [ "document" ],
    "query": {
      "value-query": {
        "element": {
          "ns": "http://marklogic.com/example",
          "name": "prefix"
        },
        "text": [ "xdmp" ]
      }
    },
    "options": {
      "default-suggestion-source": {
        "range": {
```

```
            "type": "xs:string",
            "facet": true,
            "element": {
              "ns": "",
              "name": "name"
            }
          }
        }
      }
} }
```

If you perform the same request with an Accept header of `application/json`, the response data looks like the following:

```
{
  "suggestions": [
    "document-insert",
    "document-load"
  ]
}
```

4.15.4 Where to Find More Information

For more details on using search suggestions, including performance recommendations and additional examples, see the following:

- `search:suggest` (XQuery function)
- Search Term Completion Using search:suggest in *Search Developer's Guide*.
- default-suggestion-source in the *Search Developer's Guide*
- suggestion-source in the *Search Developer's Guide*
- "Specifying Dynamic Query Options with Combined Query" on page 179

5.0 Reading and Writing Multiple Documents

The REST Client API includes interfaces that enable you to read or write multiple documents in a single request. You can select documents for a bulk read using a list of URIs or a query.

- Terms and Definitions
- Writing Multiple Documents
- Reading Multiple Documents by URI
- Reading Multiple Documents Matching a Query
- Bulk Read Response Overview

5.1 Terms and Definitions

This chapter uses the following terms and definitions:

Term	Definition
content part	A part of a `multipart/mixed` request body or response that contains document content. Content must be XML, JSON, Text, or Binary.
metadata part	A part of a `multipart/mixed` request body or response that contains document metadata such as properties, collections, quality, permissions, and key-value metadata. The content in a metadata part must XML or JSON and use the syntax described in “Working with Metadata” on page 142.
system default metadata	The default metadata values configured into MarkLogic Server. There are no defaults for collections or properties.
request default metadata	During bulk write, request-specific metadata that applies to documents without document-specific metadata. Request default metadata supercedes system default metadata. For details, see “Constructing a Metadata Part” on page 263.
document-specific metadata	During bulk write, a metadata part that applies to a specific document. Document-specific metadata supercedes request default metadata and system default metadata. For details, see “Constructing a Metadata Part” on page 263.

5.2 Writing Multiple Documents

To write multiple documents in a single request, send a POST request to the `/documents` service with a URL of the following form and set the Content-Type header to `multipart/mixed`.

```
http://host:port/version/documents
```

The request must not include a `uri` or `extension` request parameter and the Content-Type header value must be `multipart/mixed`. Each part in the request body contains either content or metadata, which is indicated by the Content-Disposition part header.

MarkLogic Server creates or updates the documents and metadata in a bulk write request in a single transaction. If a single insertion or update fails, the entire batch of document operations fails.

When you use bulk write, pre-existing document properties are preserved, but other categories of metadata are completely replaced. If you want to preserve pre-existing metadata, use a single document write, such as a PUT request to `/documents`. For details, see "Manipulating Documents" on page 53.

> **Note:** When running the examples in this section, do not cut and paste the example `multipart/mixed` POST bodies. A `multipart/mixed` request body contains control characters that are not preserved when you copy the text. For details, see "Generating Example Payloads with XQuery" on page 285.

For details, see the following topics.

- Example: Loading Multiple Documents
- Request Body Overview
- Response Overview
- Constructing a Content Part
- Understanding Metadata Scoping
- Understanding When Metadata is Preserved or Replaced
- Constructing a Metadata Part
- Applying a Write Transformation
- Example: Controlling Metadata Through Defaults
- Example: Reverting to System Default Metadata
- Example: Adding Documents to a Collection
- Generating Example Payloads with XQuery

5.2.1 Example: Loading Multiple Documents

This example provides a quick introduction to using the bulk write feature. It creates an XML document and a JSON document.

The XML document uses the system default metadata since the document part is not preceded by any document-specific or request default metadata. The JSON document uses the document-specific metadata included in the request.

The POST body contains three parts: An XML content part for a document with URI `doc1.xml`, a JSON metadata part for a document with URI `doc2.json`, and a content part for a JSON document with URI `doc2.json`.

The part Content-Type header indicates the MIME type of the part contents. The part Content-Disposition header indicates whether a part contains content or metadata. For details, see "Request Body Overview" on page 253.

The following graphic shows the breakdown of the parts.

Example Multipart POST Body for Bulk Write

XML Content

Document-Specific Metadata

JSON Content

```
--BOUNDARY
Content-Type: application/xml
Content-Disposition: attachment; filename="doc1.xml"

<?xml version="1.0" encoding="UTF-8"?>
<root><a>Some data in an XML document</a></root>
--BOUNDARY
Content-Type: application/json
Content-Disposition: attachment; filename="doc2.json";
category=metadata

{"quality" : 2 }
--BOUNDARY
Content-Type: application/json
Content-Disposition: attachment; filename="doc2.json"

{"key":"value"}
--BOUNDARY--
```

You cannot create a working multipart POST body by cutting and pasting from the examples in this guide. If your development environment does not include other tools or libraries for constructing a multipart HTTP request body, you can use the following XQuery to generate the payload using the technique described in "Generating Example Payloads with XQuery" on page 285.

```
xquery version "1.0-ml";

declare variable $OUTPUT-FILENAME := "/example/simple-body";
declare variable $BOUNDARY := "BOUNDARY";

let $xml-doc:= document{ <root><a>Some data in an XML document</a></root> }
let $json-metadata := text { '{"quality" : 2 }'}
let $json-doc :=text{'{"key":"value"}'}
let $manifest :=
   <manifest>
     <!-- doc1.xml content -->
     <part>
       <headers>
          <Content-Type>application/xml</Content-Type>
          <Content-Disposition>attachment;
```

```
filename="doc1.xml"</Content-Disposition>
       </headers>
     </part>
     <!-- doc2.json metadata -->
     <part>
       <headers>
          <Content-Type>application/json</Content-Type>
          <Content-Disposition>attachment; filename="doc2.json";
category=metadata</Content-Disposition>
       </headers>
     </part>
     <!-- doc2.json content -->
     <part>
       <headers>
         <Content-Type>application/json</Content-Type>
         <Content-Disposition>attachment;
filename="doc2.json"</Content-Disposition>
       </headers>
     </part>
   </manifest>
return
xdmp:save($OUTPUT-FILENAME,
  xdmp:multipart-encode(
    $BOUNDARY, $manifest, ($xml-doc,$json-metadata,$json-doc)
  )
)
```

The following command writes the documents and receives a JSON response. Use the Accept header to control whether the request returns an XML or JSON response. The boundary in the header must match the boundary that separates the parts in the request body.

```
$ curl --anyauth --user user:password -X POST \
  --data-binary @/example/simple-body -i \
  -H "Accept: application/json" \
  -H "Content-type: multipart/mixed; boundary=BOUNDARY" \
  http://localhost:8000/LATEST/documents
```

MarkLogic Server responds with the following data. The response includes a `document` entry for each created or updated document. Each `document` entry provides enough information to retrieve the associated document and/or its metadata. For details, see “Response Overview” on page 254.

```
HTTP/1.1 200 OK
Server: MarkLogic
Content-Type: text/xml; charset=UTF-8
Content-Length: 449
Connection: Keep-Alive
Keep-Alive: timeout=5

<rapi:documents xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:document>
    <rapi:uri>doc1.xml</rapi:uri>
    <rapi:category>metadata</rapi:category>
    <rapi:category>content</rapi:category>
    <rapi:mime-type>application/xml</rapi:mime-type>
  </rapi:document>
```

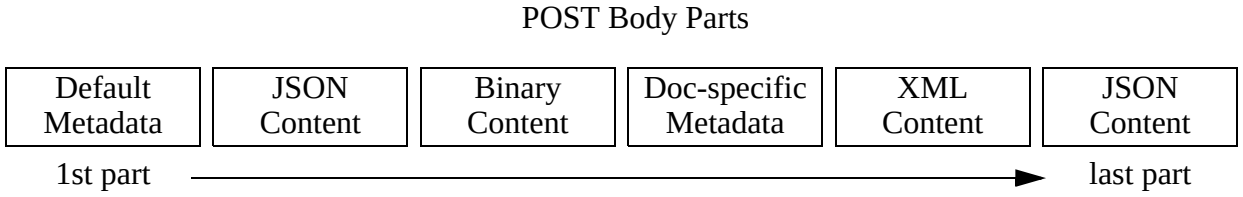
```
    <rapi:document>
      <rapi:uri>doc2.json</rapi:uri>
      <rapi:category>metadata</rapi:category>
      <rapi:category>content</rapi:category>
      <rapi:mime-type>application/json</rapi:mime-type>
    </rapi:document>
  </rapi:documents>
```

5.2.2 Request Body Overview

The multipart body of a bulk write request represents a "stream" of document metadata and/or content. Each part contains either content or metadata. The request body has the following key features:

- Document content can be heterogeneous. You can create any combination of XML, JSON, Text, and Binary documents in a single request.
- Metadata can be expressed in XML or JSON.
- You can create documents with the system default metadata, request default metadata, or document-specific metadata. You can mix these metadata sources in the same request.

For example, a request with the following parts in the POST body creates 2 JSON documents, a binary document, and an XML document. The XML document has document-specific metadata, while the other documents use the request default metadata from the first part.



For a complete example, see "Example: Loading Multiple Documents" on page 250.

The part headers describe the kind of part (content or metadata), the MIME type of the part body, and additional part-specific options. The table below outlines the role of the headers for content and metadata parts.

Part Type	Header	Description
Content	Content-Type	Determines the document type (JSON, XML, Text, or Binary).
	Content-Disposition	Identifies the part as a content part. Contains additional parameters that specify the document URI and options such as XML repair.
Metadata	Content-Type	Specifies the metadata MIME type. Metadata must be JSON or XML.
	Content-Disposition	Identifies the part as a metadata part. Determines whether the metadata is document-specific metadata or request default metadata.

For details, see the following topics:

- “Constructing a Content Part” on page 256
- “Constructing a Metadata Part” on page 263

5.2.3 Response Overview

If a bulk write request is successful, the response body contains a list of document descriptors, one per document created, in the order of creation. The response can be XML or JSON, depending on the Accept header.

If there is an error creating one of the documents, the entire batch is rejected and the database is unchanged. In the event of an error, MarkLogic Server responds with error status code and the response body includes details about the failing document.

For example, the following response data reflects successful creation of 3 documents. Both metadata and content are available for `doc1.xml` and `doc2.json`, but only metadata is availble for `doc3.xml`.

Format	Response Data
XML	<rapi:documents xmlns:rapi="http://marklogic.com/rest-api"> <rapi:document> <rapi:uri>**doc1.xml**</rapi:uri> <rapi:category>metadata</rapi:category> <rapi:category>content</rapi:category> <rapi:mime-type>application/xml</rapi:mime-type> </rapi:document> <rapi:document> <rapi:uri>**doc2.json**</rapi:uri> <rapi:category>metadata</rapi:category> <rapi:category>content</rapi:category> <rapi:mime-type>application/json</rapi:mime-type> </rapi:document> <rapi:document> <rapi:uri>**doc3.xml**</rapi:uri> <rapi:category>metadata</rapi:category> <rapi:mime-type>application/xml</rapi:mime-type> </rapi:document> </rapi:documents>

JSON	{ "documents": [
	  { "uri": "doc1.xml",
	  "mime-type": "application/xml",
	  "category": [
	  "metadata",
	  "content"
	  ]
	  },
	  { "uri": "doc2.json",
	  "mime-type": "application/json",
	  "category": [
	  "metadata",
	  "content"
	  ]
	  },
	  { "uri": "doc3.xml",
	  "mime-type": "application/xml",
	  "category": [
	  "metadata",
	  "content"
	  ]
	  }
	] }

You can use the data in the response to retrieve the inserted documents. For example, you can retrieve `doc2.json` using the following GET request, where the MIME type in the Accept header comes from the response `mime-type` element/key value and the `uri` and `category` request parameter values come from the corresponding response element/key values.

```
$ curl --anyauth --user user:password -X GET -i \
  -H "Accept: application/json" \
  'http://localhost:8000/LATEST/documents?uri=doc2.json&category=content'
```

The following table provides more details on the information returned about each document:

Element or Key Name	Description
`uri`	The database URI of the document. This is either the URI supplied in the POST body or a server-generated URI. You can use this data to build the `uri` parameter of a document retrieval request such as `GET:/v1/documents`.
`category`	The categories of data updated (`content` or `metadata`). You can use this data to build the `category` parameter of a document retrieval request, such as `GET:/v1/documents`. If the request includes a content update for a document, both `content` and `metadata` are returned. If only metadata is updated, then only `metadata` is returned.
`mime-type`	A MIME type suitable for retrieving the contents of the document. For example, you can use this data in the Accept header of a document retrieval request such as `GET:/v1/documents`.

5.2.4 Constructing a Content Part

When constructing a content part, you must include a Content-Type header that controls the document type and a Content-Disposition header that controls the document URI. The Content-Disposition header can also include additional options.

A content part is distinguished from a metadata part by the absence of `category=metadata` in the Content-Disposition header.

For details, see the following topics:

- Controlling Document Type
- Specifying an Explicit Document URI

- Automatically Generating a Document URI
- Adding Content Options
- Example Content Part Headers

5.2.4.1 Controlling Document Type

You can create JSON, XML, Binary, and Text documents in a bulk write. The MIME type of the part Content-Type header determines the database document type.

> **Note:** You are responsible for ensuring the MIME type and the part contents match. MarkLogic Server does not sniff the contents or perform implicit content conversions.

The MarkLogic Server MIME type mapping determines the document type for XML, Text, and Binary documents. For example, the following header creates an XML document because the MIME type `application/xml` is mapped to XML in the default server MIME type mapping.

```
Content-Type: application/xml
```

To review or change the MIME type mapping, see the “Mimetypes” subsection of the Groups section of the Admin Interface.

For JSON documents, content is assumed to be JSON if the MIME type begins with `application/`, optionally followed by `some-format+`, and ends with `json`. For example, all the following headers indicate a part that contains JSON:

```
Content-Type: application/json
Content-Type: application/rdf+json
```

5.2.4.2 Specifying an Explicit Document URI

To include an explicit URI for a document, set the content disposition of the metadata and/or content part to `attachment` and put the URI in the `filename` parameter. That is, use the following header template:

```
Content-Disposition: attachment;filename=/your/uri
```

A metadata part of the above form signifies document-specific metadata. If the request includes content for the same document, the corresponding content part must immediately follow the document-specific metadata part. For details, see “Constructing a Metadata Part” on page 263.

For examples, see “Example Content Part Headers” on page 258.

5.2.4.3 Automatically Generating a Document URI

To have MarkLogic Server generate the URI for a document, set the content disposition to `inline` and include an `extension` parameter that specifies the URI extension. You can also include an optional `directory` parameter that specifies a destination database directory. That is, use one of the following header templates:

```
Content-Disposition: inline;extension=suffix
Content-Disposition: inline;extension=suffix;directory=/your/dir/
```

Note: Do not include a separator in the extension value. MarkLogic Server will prefix your suffix with a period (.).

Note: A directory path must end with a slash (/).

Best practice is to use an extension suffix for which there is a MIME type mapping in MarkLogic Server. The part Content-type should match the MIME type corresponding to the extension. To review or change the MIME type mapping, see the "Mimetypes" subsection of the Groups section of the Admin Interface.

You can only use server-generated URIs on content parts. Metadata for documents with server-assigned URIs can only come from request default metadata or system default metadata. For details, see "Constructing a Metadata Part" on page 263.

For example, the following headers signify a JSON content part with a server-generated URI that uses the directory `/my/dir` and ends with `.json`, such as `/my/dir/1234567890.json`.

```
Content-type: application/json
Content-Disposition: inline;extension=json;directory=/my/dir/
```

For more examples, see "Example Content Part Headers" on page 258.

5.2.4.4 Adding Content Options

Use the Content-Disposition header to include document-specific options such as `lang`, `repair`, or `extract`. For example, the use of `extract=properties` in the following header tells MarkLogic Server to extract metadata from a binary document and save it as a document property.

```
Content-type: image/jpeg
Content-Disposition: attachment;filename=my.jpg;extract=properties
```

For option details, refer to the API reference for `POST:/v1/documents`.

5.2.4.5 Example Content Part Headers

The table below shows example headers for several kinds of content part.

Content Part Description	Example Headers
XML document with an explicit URI	`Content-type: application/xml` `Content-Disposition: attachment;filename=/my/uri.xml`
XML document with a repair option	`Content-type: application/xml` `Content-Disposition: attachment;filename=my.xml;repair=none`
JSON document with an explicit URI	`Content-type: application/json` `Content-Disposition: attachment;filename=/my/uri.json`
JSON document with a server assigned URI and a client-specified database directory	`Content-type: application/json` `Content-Disposition: inline;extension=json;directory=/my/dir/`
Binary document with a server assigned URI and a client specified database directory	`Content-type: image/jpeg` `Content-Disposition: inline;extension=jpg;directory=/images/`

5.2.5 Understanding Metadata Scoping

This topic describes how metadata is selected for documents created or updated with a bulk write. For details on the structure of a metadata part, see “Constructing a Metadata Part” on page 263.

Note: For performance reasons, pre-existing metadata other than properties is completely replaced during a bulk write operation, either with values supplied in the request or with system defaults.

Metadata in a bulk write can be drawn from 3 possible sources, as shown in the table below. The table lists the metadata sources from highest to lowest precedence, so a source supercedes those below it if both are present.

Metadata Type	Description
document-specific metadata	A metadata part that includes a document URI in its headers applies to a single document. If the request includes content for the document, the content part must occur immediately after the metadata part.
request default metadata	A metadata part that does not include a document URI in its headers is default metadata that applies to all subsequent content parts that do not have document-specific metadata. The default applies until the occurrence of the next request default metadata part or the end of the request body.
system default metadata	Default metadata configured into MarkLogic server. This metadata applies any time no document-specific or request default metadata is in scope.

The following rules determine what metadata applies during document creation.

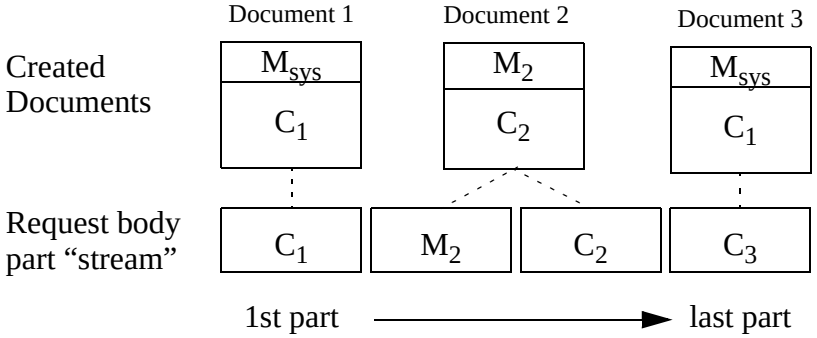
- If a content part is immediately preceded by a corresponding document-specific metadata part, use it.
- If a content part is not immediately preceded by document-specific metadata, then use the most recent preceding request default metadata part.
- If there is no request default metadata and no document-specific metadata, then use the system default metadata.
- Any request metadata part entirely replaces previous in-scope request metadata. For performance reasons, no merging occurs.
- A metadata category not included in the part body is either set to the system default metadata value or left unchanged, depending upon whether or not the request includes a content update. For details, see “Understanding When Metadata is Preserved or Replaced” on page 262.

  Note: If a request body includes both content and document-specific metadata for the same document, the metadata part must immediately precede the content part.

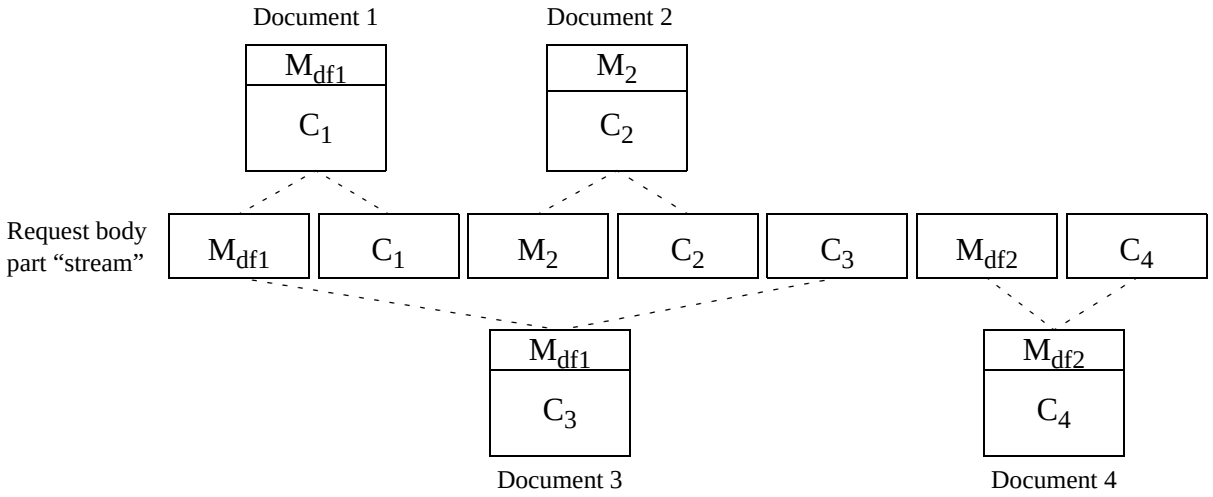
Each occurrence of request default metadata completely replaces any preceding request default metadata. Similarly, request default metadata and document-specific metadata are not merged. For example, if a document-specific metadata part contains only a collections setting, it inherits quality, permissions and properties from the system default metadata, not from any preceding request default metadata.

The following examples illustrate application of these rules. In these examples, C_n represents a content part for the Nth document, M_n represents document-specific metadata for the Nth document, M_{dfn} represents the Nth occurrence of request default metadata, and M_{sys} is the system default metadata.

The following input creates 3 documents. Documents 1 and Document 3 use system default metadata. Document 2 uses document-specific metadata.



The following input creates four documents, using a combination of request default metadata and document-specific metadata. Document 1, Document 3, and Document 4 use request default metadata. Document 2 uses document-specific metadata. Document 1 and Document 3 use the first block of request default metadata, M_{df1}. After Document 3 is created, M_{df2} replaces M_{df1} as the request default metadata, so Document 4 uses the metadata in M_{df2}.



5.2.6 Understanding When Metadata is Preserved or Replaced

This topic discusses when a bulk write preserves or replaces pre-existing metadata. You can skip this section if your bulk write operations only create new documents or you do not need to preserve pre-existing metadata such as permissions, document quality, collections, and properties.

When there is no request default metadata and no document-specific metadata, all metadata categories other than properties are set to the system default values. Properties are unchanged.

In all other cases, either request default metadata or document-specific metadata is used when creating a document, as described in “Understanding Metadata Scoping” on page 259.

When you update both content and metadata for a document in the same bulk write request, the following rules apply, whether applying request default metadata or document-specific metadata:

- The metadata in scope is determined as described in “Understanding Metadata Scoping” on page 259.
- Any metadata category that has a value in the in-scope metadata completely replaces that category.
- Any metadata category other than properties that is missing or empty in the in-scope metadata is completely replaced by the system default value.
- If the in-scope metadata does not include properties, then existing properties are preserved.
- If the in-scope metadata does not include collections, then collections are reset to the default. There is no system default for collections, so this results in a document being removed from all collections if no default collections are specified for the user role performing the update.

When you include a document-specific metadata part, but no content part, you update only the metadata for a document. In this case, the following rules apply:

- Any metadata category that has a value in the document-specific metadata completely replaces that category.
- Any metadata category that is missing or empty in the document-specific metadata is preserved.

The table below shows how pre-existing metadata changes if a bulk write updates just the content, just the collections metadata (via document-specific metadata), or both content and collections metadata (via request default metadata or document-specific metadata).

Metadata Category	Update Content Only	Update Metadata Only	Update Content & Metadata
collections	reset	modified to new value	modified to new value
quality	reset	preserved	reset
permissions	reset	preserved	reset
properties	preserved	preserved	preserved
metadata-values	reset, except for system-managed keys	preserved	reset, except for system-managed keys

The results are similar if the metadata update modifies other metadata categories.

5.2.7 Constructing a Metadata Part

This section describes how metadata values are applied during a bulk write and how to construct a request default or document-specific metadata part. Metadata must expressed as XML or JSON and must be specified using the syntax described in “Working with Metadata” on page 142.

The following topics are covered:

- Constructing a Request Default Metadata Part
- Constructing a Document-Specific Metadata Part
- Disabling Request Default Metadata

5.2.7.1 Constructing a Request Default Metadata Part

A request default metadata part applies to all documents created from the occurrence of the metadata part until the next request default metadata part or the end of the POST body, except for documents that have document-specific metadata.

A request default metadata part must use headers of the following form, where *`metadata-MIME-type`* is a MIME type that maps to XML or JSON metadata content:

```
Content-Type: metadata-MIME-type
Content-Disposition: inline; category=metadata
```

For example, use following headers on a default metadata part that contains JSON metadata:

```
Content-Type: application/json
Content-Disposition: inline; category=metadata
```

The part contents must be document metadata of the form described in “Working with Metadata” on page 142.

For a complete example, see “Example: Controlling Metadata Through Defaults” on page 269.

5.2.7.2 Constructing a Document-Specific Metadata Part

Document-specific metadata applies to a single document. The part must use headers of the following form, where *`metadata-MIME-type`* is a MIME type that maps to XML or JSON metadata content and *`document-uri`* is the database URI of the target document.

```
Content-Type: metadata-MIME-type
Content-Disposition: attachment; filename=document-uri; category=metadata
```

If both content and metadata for the document are included in the POST request, then the document-specific metadata part must occur immediately before the content part.

If the input includes document specific metadata without an accompanying content part, the document must already exist. Only the metadata of the document changes.

Any metadata sub-category other than properties that is not included in the part body is set to the request default value, if one is in scope. Otherwise, it is set to the system default value. However, there is no system default for collections or properties. Existing properties are unchanged if not included. For collections, if the user performing the update has a role that includes default collections, then that default is used. If there are no default collections for the role, then the document is removed from all collections.

Note: When updating permissions, you do not need to include the default `rest-reader` and `rest-writer` roles.

Note: Any explicitly specified permissions are combined with the default permissions for the role of the current user.

For example, suppose a pre-exisitng document has the following customized metadata:

- document quality of 2
- membership in a collection named “April 2014”
- read permissions for a custom role
- a custom property named “custom-prop”.

That is, if you retrieve the metadata for this document using `GET /version/documents`, then the response contains metadata similar to the following. The metadata values that differ from the system default are shown in bold.

Format	Metadata

XML	<rapi:metadata xmlns:rapi="http://marklogic.com/rest-api"...>
	  <rapi:collections>
	  <rapi:collection>April 2014</rapi:collection>
	  </rapi:collections>
	  <rapi:permissions>
	  <rapi:permission>
	  <rapi:role-name>rest-writer</rapi:role-name>
	  <rapi:capability>update</rapi:capability>
	  </rapi:permission>
	  <rapi:permission>
	  <rapi:role-name>rest-reader</rapi:role-name>
	  <rapi:capability>read</rapi:capability>
	  </rapi:permission>
	  **<rapi:permission>
	  <rapi:role-name>some-custom-role</rapi:role-name>
	  <rapi:capability>read</rapi:capability>
	  </rapi:permission>**
	  </rapi:permissions>
	  <prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
	  <custom-prop>some property value</custom-prop>
	  <prop:last-modified>2014-04-10T16:11:56-07:00</prop:last-modified>
	  </prop:properties>
	  <rapi:quality>2</rapi:quality>
	</rapi:metadata>

Format	Metadata

JSON	<pre>{
	  "collections": ["April 2014"],
	  "permissions": [
	  {
	  "role-name": "rest-writer",
	  "capabilities": ["update"]
	  },
	  {
	  "role-name": "rest-reader",
	  "capabilities": ["read"]
	  },
	  {
	  "role-name": "custom-role",
	  "capabilities": ["read"]
	  }
	  ],
	  "properties": {
	  "custom-prop": "some property value",
	  "$ml.prop": { "last-modified": "2014-04-10T16:21:15-07:00" }
	  },
	  "quality": 2
	}</pre>

If you use a bulk write to update the metadata for this document, specifying only a new document quality of 1, then the permissions and collections are reset to the system default, but the custom property is preserved. Assuming out-of-the box MarkLogic Server system defaults, the updated document is no longer be in any collections, and the read permission for “custom-role” is removed, resulting in the following metadata:

Format	Metadata

XML	<rapi:metadata xmlns:rapi="http://marklogic.com/rest-api"...>
	  <rapi:collections/>
	  <rapi:permissions>
	  <rapi:permission>
	  <rapi:role-name>rest-writer</rapi:role-name>
	  <rapi:capability>update</rapi:capability>
	  </rapi:permission>
	  <rapi:permission>
	  <rapi:role-name>rest-reader</rapi:role-name>
	  <rapi:capability>read</rapi:capability>
	  </rapi:permission>
	  </rapi:permissions>
	  <prop:properties xmlns:prop="http://marklogic.com/xdmp/property">
	  <custom-prop>some property value</custom-prop>
	  <prop:last-modified>2014-04-10T16:11:56-07:00</prop:last-modified>
	  </prop:properties>
	  <rapi:quality>1</rapi:quality>
	</rapi:metadata>

JSON	{
	  "collections": [],
	  "permissions": [
	  {
	  "role-name": "rest-writer",
	  "capabilities": ["update"]
	  },
	  {
	  "role-name": "rest-reader",
	  "capabilities": ["read"]
	  }
	  ],
	  "properties": {
	  "custom-prop": "some property value",
	  "$ml.prop": { "last-modified": "2014-04-10T16:21:15-07:00" }
	  },
	  "quality": 1
	}

5.2.7.3 Disabling Request Default Metadata

A request default metadata part applies until the next default request metadata part or the end of the POST body. To "turn off" request default metadata and revert to the system default metadata, create a request default metadata part that contains no metadata values.

In JSON, use `"{}"` to express empty metadata content. In XML, use a `<rapi:metadata>` element with no child elements.

For example, the following part erases any previous request default metadata. In the absence of additional metadata parts, documents created on behalf of subsequent parts use system default metadata.

```
Content-Type: application/json
Content-Disposition: inline; category=metadata
Content-Length: 3

{ }
```

The following is an equivalent empty XML metadata part:

```
Content-Type: application/xml
Content-Disposition: inline; category=metadata
Content-Length: 98

<?xml version="1.0" encoding="UTF-8"?>
<rapi:metadata xmlns:rapi="http://marklogic.com/rest-api"/>
```

For a complete example, see "Example: Reverting to System Default Metadata" on page 274.

5.2.8 Applying a Write Transformation

A content write transformation is a user-defined XQuery function or XSLT style sheet used to modify documents during insertion, as described in "Working With Content Transformations" on page 323.

You can apply a content transformation to all documents created by a bulk write request by making a POST request with a URL of the following form:

```
http://host:port/version/documents?transform=name&trans:param=value
```

For example, the following URL specifies a transform function named `example` with a value of "me" for the function parameter named `reviewer`.

```
http://myhost:8000/LATEST/documents?transform=example&trans:reviewer=me
```

Transform parameters are optional, depending on the transform interface. The same transform parameter values are applied to every document document in the request.

MarkLogic Server applies the transformation prior to inserting each document into the database. Your transform can choose not to make any modifications, but it will be invoked for every document.

You cannot specify more than one transform per request. However, your transform function can import and call other transformation functions to get the same effect.

For details, see “Working With Content Transformations” on page 323.

5.2.9 Example: Controlling Metadata Through Defaults

This example creates documents that use the system default, request default, and document specific metadata, as described in “Constructing a Metadata Part” on page 263.

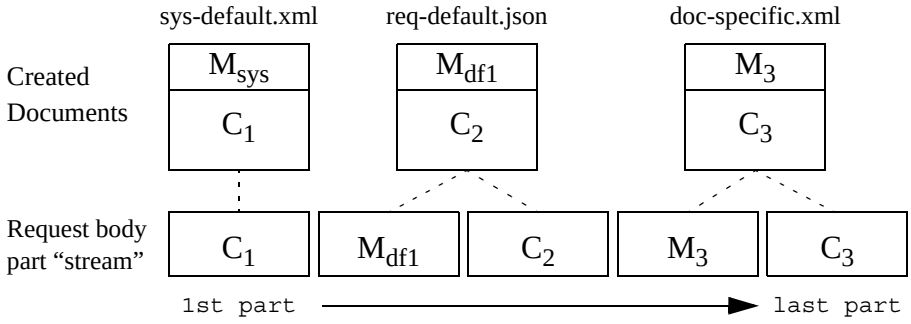
- Payload Description
- Generating the POST Body
- Executing the Request
- Verifying the Results

5.2.9.1 Payload Description

This example uses document quality to illustrate how default metadata affects the documents you create. The document quality setting used in this example result in creation of the following documents:

- `sys-default.xml` with document quality 0, from the system default metadata
- `req-default.json` with document quality 2, from M_{df1}
- `doc-specific.xml` with document quality 1, from M_3

The following picture represents the parts in the request body and the documents created from them. In the picture, M_n represents metadata, C_n represents content. Note that the metadata is not literally embedded in the created documents; content and metadata are merely grouped here for illustrative purposes.



The following multipart body implements the pictured bulk write input. The annotations on the right indicate the start of each part in the graphic above. These annotations are not present in the actual POST body.

```
--BOUNDARY
Content-Type: application/xml                                        (C1)
Content-Disposition: attachment; filename="sys-default.xml"
Content-Length: 86

<?xml version="1.0" encoding="UTF-8"?>
<root>a doc with system default metadata</root>
--BOUNDARY
Content-Type: application/json                                      (Mdfl)
Content-Disposition: inline; category=metadata
Content-Length: 16

{"quality" : 2 }
--BOUNDARY
Content-Type: application/json                                       (C2)
Content-Disposition: attachment; filename="req-default.json"
Content-Length: 45

{"key":"a doc with request default metadata"}
--BOUNDARY
Content-Type: application/xml                                        (M3)
Content-Disposition: attachment; filename="doc-specific.xml";
category=metadata
Content-Length: 147

<?xml version="1.0" encoding="UTF-8"?>
<rapi:metadata xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:quality>1</rapi:quality>
</rapi:metadata>
--BOUNDARY
Content-Type: application/xml                                        (C3)
Content-Disposition: attachment; filename="doc-specific.xml"
Content-Length: 89

<?xml version="1.0" encoding="UTF-8"?>
<root>a doc with document-specific metadata</root>
--BOUNDARY--
```

5.2.9.2 Generating the POST Body

This section contains an XQuery query you can use to generate the POST body, using the technique described in as described in “Generating Example Payloads with XQuery” on page 285. Skip this section if you use other tools to generate the request body.

Before running this query in Query Console, modify `$OUTPUT-FILENAME` for your environment:

```
xquery version "1.0-ml";

declare variable $OUTPUT-FILENAME := "/example/default_metadata";
declare variable $BOUNDARY := "BOUNDARY";

let $sys-default:=
  document{ <root>a doc with system default metadata</root> }
let $req-specific :=
  text{'{"key":"a doc with request default metadata"}' }
let $doc-specific :=
  document{ <root>a doc with document-specific metadata</root> }
let $req-metadata := text { '{"quality" : 2 }'}
let $sys-default-json :=
  text { '{ "key": "a doc with system default metadata" }' }
let $empty-metadata := text { '{ }' }
let $doc-metadata := document {
    <rapi:metadata xmlns:rapi="http://marklogic.com/rest-api">
      <rapi:quality>1</rapi:quality>
    </rapi:metadata>
  }
let $manifest :=
   <manifest>
     <!-- content, using system default metadata -->
     <part>
       <headers>
          <Content-Type>application/xml</Content-Type>
          <Content-Disposition>attachment;
filename="sys-default.xml"</Content-Disposition>
       </headers>
     </part>
     <!-- Request default metadata -->
     <part>
       <headers>
          <Content-Type>application/json</Content-Type>
          <Content-Disposition>inline;
category=metadata</Content-Disposition>
       </headers>
     </part>
     <!-- JSON document using request default metadata -->
     <part>
       <headers>
          <Content-Type>application/json</Content-Type>
          <Content-Disposition>attachment;
filename="req-default.json"</Content-Disposition>
       </headers>
     </part>
      <!-- document specific metadata, in XML -->
      <part>
       <headers>
          <Content-Type>application/xml</Content-Type>
         <Content-Disposition>attachment; filename="doc-specific.xml";
category=metadata</Content-Disposition>
       </headers>
     </part>
```

```
      <!-- XML document with document specific metadata -->
      <part>
        <headers>
           <Content-Type>application/xml</Content-Type>
           <Content-Disposition>attachment;
filename="doc-specific.xml"</Content-Disposition>
        </headers>
      </part>
    </manifest>
return
xdmp:save($OUTPUT-FILENAME,
  xdmp:multipart-encode(
    $BOUNDARY, $manifest,
    ($sys-default,
     $req-metadata,
     $req-specific,
     $doc-metadata,
     $doc-specific)
  )
)
```

5.2.9.3 Executing the Request

The following command executes the bulk write request, assuming `/example/default_metadata` contains the POST body. Modify the value of the `--data-binary` option to the file that contains your POST body.

```
$ curl --anyauth --user user:password -X POST -i \
  --data-binary @/example/default_metadata \
  -H "Content-type: multipart/mixed; boundary=BOUNDARY" \
  http://localhost:8000/LATEST/documents
```

MarkLogic Server returns output similar to the following. For details, see “Response Overview” on page 254.

```
HTTP/1.1 200 OK
Server: MarkLogic
Content-Type: text/xml; charset=UTF-8
Content-Length: 656
Connection: Keep-Alive
Keep-Alive: timeout=5

<?xml version="1.0"?>
<rapi:documents xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:document>
    <rapi:uri>sys-default.xml</rapi:uri>
    <rapi:category>metadata</rapi:category>
    <rapi:category>content</rapi:category>
    <rapi:mime-type>application/xml</rapi:mime-type>
  </rapi:document>
  <rapi:document>
    <rapi:uri>req-default.json</rapi:uri>
    <rapi:category>metadata</rapi:category>
```

```
    <rapi:category>content</rapi:category>
    <rapi:mime-type>application/json</rapi:mime-type>
  </rapi:document>
  <rapi:document>
    <rapi:uri>doc-specific.xml</rapi:uri>
    <rapi:category>metadata</rapi:category>
    <rapi:category>content</rapi:category>
    <rapi:mime-type>application/xml</rapi:mime-type>
  </rapi:document>
</rapi:documents>
```

For a JSON response, set the Accept header to `application/json`:

```
$ curl --anyauth --user user:password -X POST -i \
  --data-binary @/example/default_metadata \
  -H "Content-type: multipart/mixed; boundary=BOUNDARY" \
  -H "Accept: application/json" \
  http://localhost:8000/LATEST/documents
```

5.2.9.4 Verifying the Results

You can retrieve the document quality for the created documents individually, or you can use the following bulk read command to retrieve the quality for all the documents in a single request. This example requests JSON metadata; you can also retrieve metadata as XML. MarkLogic Server returns a `multipart/mixed` response where each response body part contains the requested data for a different document. For details, see “Reading Multiple Documents by URI” on page 288.

```
$ curl --anyauth --user user:password -X GET -i \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/documents?category=quality&format=json
&uri=sys-default.xml&uri=req-default.json&uri=doc-specific.xml'
...
HTTP/1.1 200 OK
Content-type: multipart/mixed; boundary=BOUNDARY
Server: MarkLogic
Content-Length: 701
Connection: Keep-Alive
Keep-Alive: timeout=5

--BOUNDARY
Content-Type: application/json
Content-Disposition: attachment; filename=sys-default.xml;
category=quality; format=json
Content-Length: 13

{"quality":0}
--BOUNDARY
Content-Type: application/json
Content-Disposition: attachment; filename=req-default.json;
category=quality; format=json
Content-Length: 13
```

```
{"quality":2}
--BOUNDARY
Content-Type: application/json
Content-Disposition: attachment; filename=doc-specific.xml;
category=quality; format=json
Content-Length: 13

{"quality":1}
--BOUNDARY--
```

5.2.10 Example: Reverting to System Default Metadata

This example demonstrates how to "erase" request default metadata, bringing the system default metadata back into scope in the middle of a request. This example builds on "Example: Controlling Metadata Through Defaults" on page 269.

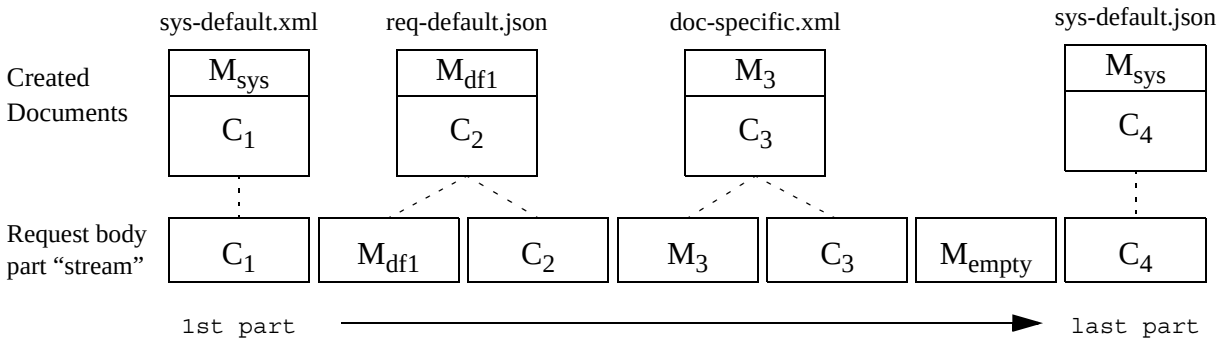
- Payload Description
- Generating the POST Body
- Executing the Request
- Verifying the Results

5.2.10.1 Payload Description

To revert to the system default metadata, include a metadata part that contains no metadata, as described in "Disabling Request Default Metadata" on page 268.

This example extends the POST body from "Example: Controlling Metadata Through Defaults" on page 269 by appending 2 parts to the payload: An empty metadata part and a 4th document, `sys-default.json`, that uses the system default metadata. This gives `sys-default.json` a document quality of 0 (the system default). All documents created after M_{empty} use the system default metadata in the absence of additional metadata parts.

The graphic below illustrates the parts in the request body and the documents it creates:



The following POST body implements the bulk write illustrated by the graphic. The annotations on the right indicate the start of each part in the graphic. These annotations are not present in the actual POST body.

```
--BOUNDARY
Content-Type: application/xml                                    (C1)
Content-Disposition: attachment; filename="sys-default.xml"
Content-Length: 86

<?xml version="1.0" encoding="UTF-8"?>
<root>a doc with system default metadata</root>
--BOUNDARY
Content-Type: application/json                                  (Mdfl)
Content-Disposition: inline; category=metadata
Content-Length: 16

{"quality" : 2 }
--BOUNDARY
Content-Type: application/json                                   (C2)
Content-Disposition: attachment; filename="req-default.json"
Content-Length: 45

{"key":"a doc with request default metadata"}
--BOUNDARY
Content-Type: application/xml                                    (M3)
Content-Disposition: attachment; filename="doc-specific.xml";
category=metadata
Content-Length: 147

<?xml version="1.0" encoding="UTF-8"?>
<rapi:metadata xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:quality>1</rapi:quality>
</rapi:metadata>
--BOUNDARY
Content-Type: application/xml                                    (C3)
Content-Disposition: attachment; filename="doc-specific.xml"
Content-Length: 89

<?xml version="1.0" encoding="UTF-8"?>
<root>a doc with document-specific metadata</root>
--BOUNDARY--
Content-Type: application/json                                (Mempty)
Content-Disposition: inline; category=metadata
Content-Length: 3

{ }
--BOUNDARY
Content-Type: application/json                                   (C4)
Content-Disposition: attachment; filename="sys-default.json"
Content-Length: 53
```

```
{ "key": "another doc with system default metadata" }
--BOUNDARY--
```

5.2.10.2 Generating the POST Body

This section contains an XQuery query you can use to generate the POST body, using the technique described in as described in “Generating Example Payloads with XQuery” on page 285. Skip this section if you use other tools to generate the request body.

Before running this query in Query Console, modify `$OUTPUT-FILENAME` for your environment:

```
xquery version "1.0-ml";

declare variable $OUTPUT-FILENAME := "/example/reset_metadata";
declare variable $BOUNDARY := "BOUNDARY";

let $sys-default:=
  document{ <root>a doc with system default metadata</root> }
let $req-specific :=
  text{'{"key":"a doc with request default metadata"}' }
let $doc-specific :=
  document{ <root>a doc with document-specific metadata</root> }
let $req-metadata := text { '{"quality" : 2 }'}
let $sys-default-json :=
  text { '{ "key": "a doc with system default metadata" }' }
let $empty-metadata := text { '{ }' }
let $doc-metadata := document {
  <rapi:metadata xmlns:rapi="http://marklogic.com/rest-api">
    <rapi:quality>1</rapi:quality>
  </rapi:metadata>
}
let $manifest :=
   <manifest>
     <!-- content, using system default metadata -->
     <part>
       <headers>
          <Content-Type>application/xml</Content-Type>
          <Content-Disposition>attachment;
filename="sys-default.xml"</Content-Disposition>
       </headers>
     </part>
     <!-- Request default metadata -->
     <part>
       <headers>
          <Content-Type>application/json</Content-Type>
          <Content-Disposition>inline;
category=metadata</Content-Disposition>
       </headers>
     </part>
     <!-- JSON document using request default metadata -->
     <part>
       <headers>
          <Content-Type>application/json</Content-Type>
```

```
          <Content-Disposition>attachment;
filename="req-default.json"</Content-Disposition>
       </headers>
     </part>
      <!-- document specific metadata, in XML -->
      <part>
       <headers>
          <Content-Type>application/xml</Content-Type>
         <Content-Disposition>attachment; filename="doc-specific.xml";
category=metadata</Content-Disposition>
       </headers>
     </part>
     <!-- XML document with document specific metadata -->
     <part>
       <headers>
          <Content-Type>application/xml</Content-Type>
          <Content-Disposition>attachment;
filename="doc-specific.xml"</Content-Disposition>
       </headers>
     </part>
     <!-- reset request default metadata -->
     <part>
       <headers>
          <Content-Type>application/json</Content-Type>
          <Content-Disposition>inline;
category=metadata</Content-Disposition>
       </headers>
     </part>
     <!-- Content that gets the system default metadata -->
     <part>
       <headers>
          <Content-Type>application/json</Content-Type>
          <Content-Disposition>attachment;
filename="sys-default.json"</Content-Disposition>
       </headers>
     </part>
     </manifest>
return
xdmp:save($OUTPUT-FILENAME,
  xdmp:multipart-encode(
    $BOUNDARY, $manifest,

($sys-default,$req-metadata,$req-specific,$doc-metadata,$doc-specific,
$empty-metadata,$sys-default-json)
  )
)
```

5.2.10.3 Executing the Request

The following command executes the bulk write request, assuming `/example/reset_metadata` contains the POST body. Modify the value of the `--data-binary` option to refer to the path to the file containing your POST body.

```
$ curl --anyauth --user user:password -X POST -i \
  --data-binary @/example/reset_metadata \
  -H "Content-type: multipart/mixed; boundary=BOUNDARY"
  http://localhost:8000/LATEST/documents
```

MarkLogic Server returns output similar to the following. For details, see “Response Overview” on page 254.

```
<?xml version="1.0"?>
<rapi:documents xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:document>
    <rapi:uri>sys-default.xml</rapi:uri>
    <rapi:category>metadata</rapi:category>
    <rapi:category>content</rapi:category>
    <rapi:mime-type>application/xml</rapi:mime-type>
  </rapi:document>
  <rapi:document>
    <rapi:uri>req-default.json</rapi:uri>
    <rapi:category>metadata</rapi:category>
    <rapi:category>content</rapi:category>
    <rapi:mime-type>application/json</rapi:mime-type>
  </rapi:document>
  <rapi:document>
    <rapi:uri>doc-specific.xml</rapi:uri>
    <rapi:category>metadata</rapi:category>
    <rapi:category>content</rapi:category>
    <rapi:mime-type>application/xml</rapi:mime-type>
  </rapi:document>
  <rapi:document>
    <rapi:uri>sys-default.json</rapi:uri>
    <rapi:category>metadata</rapi:category>
    <rapi:category>content</rapi:category>
    <rapi:mime-type>application/json</rapi:mime-type>
  </rapi:document>
</rapi:documents>
```

For a JSON response, set the Accept header to application/json:

```
$ curl --anyauth --user user:password -X POST -i \
  --data-binary @/example/reset_metadata \
  -H "Accept: application/json" \
  -H "Content-type: multipart/mixed; boundary=BOUNDARY"
  http://localhost:8000/LATEST/documents
```

5.2.10.4 Verifying the Results

Use the following command to retrieve the document quality for the inserted documents. Notice that `req-default.json` uses the request default document quality of 2, but `sys-default.json`, which is created after resetting the request default metadata, has the default document quality, zero.

```
$ curl --anyauth --user user:password -X GET -i \
    -H "Accept: multipart/mixed; boundary=BOUNDARY" \
    'http://localhost:8000/LATEST/documents?category=quality&format=js
on&uri=req-default.json&uri=sys-default.json'
...
HTTP/1.1 200 OK
Content-type: multipart/mixed; boundary=BOUNDARY
Server: MarkLogic
Content-Length: 327
Connection: Keep-Alive
Keep-Alive: timeout=5

--BOUNDARY
Content-Type: application/json
Content-Disposition: attachment; filename=req-default.json;
category=quality; format=json
Content-Length: 13

{"quality":2}
--BOUNDARY
Content-Type: application/json
Content-Disposition: attachment; filename=sys-default.json;
category=quality; format=json
Content-Length: 13

{"quality":0}
--BOUNDARY--
```

5.2.11 Example: Adding Documents to a Collection

This example demonstrates using request default metadata to add all documents to the same collection during insertion.

You can add selected documents to a different collection using document-specific metadata or by including an additional request default metadata part that uses a different collection; see “Example: Controlling Metadata Through Defaults” on page 269.

Since the metadata in the example request only includes settings for collections metadata, other metadata categories such as permissions and quality use the system default settings. For an example of more complex metadata, see “Extending the Example” on page 285.

For more information on metadata syntax, see “Working with Metadata” on page 142.

- Payload Description
- Generating the POST Body
- Executing the Request
- Verifying the Results
- Extending the Example

5.2.11.1 Payload Description

The payload contains 3 parts: A metadata part that specifies a collection and JSON and XML content parts for documents that are placed into that collection. The documents are added to the collection “April 2014”. The metadata is expressed as JSON. Both documents use server-assigned URIs.

	Multipart/mixed Request Body
Request Default Metadata	--BOUNDARY Content-Type: application/json Content-Disposition: inline; category=metadata Content-Length: 33 {"collections" : ["April 2014"] }
JSON Content	--BOUNDARY Content-Type: application/json Content-Disposition: inline; extension=json; directory=/example/ Content-Length: 18 { "key": "value" }
XML Content	--BOUNDARY Content-Type: application/xml Content-Disposition: inline; extension=xml; directory=/example/ Content-Length: 64 <?xml version="1.0" encoding="UTF-8"?> <root>some content</root> --BOUNDARY--

To use XML metadata instead of JSON, modify the Content-Type of the first part to `application/xml` and change the part contents to the following:

```
<rapi:metadata xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:collections>
    <rapi:collection>April 2014</rapi:collection>
  </rapi:collections>
</rapi:metadata>
```

5.2.11.2 Generating the POST Body

This section contains an XQuery query you can use to generate the POST body, using the technique described in as described in “Generating Example Payloads with XQuery” on page 285. Skip this section if you use other tools to generate the request body.

Before running this query in Query Console, modify `$OUTPUT-FILENAME` for your environment:

```
xquery version "1.0-ml";
```

```
declare variable $OUTPUT-FILENAME := "/example/coll-body";
declare variable $BOUNDARY := "BOUNDARY";

let $xml-content := document{ <root>some content</root> }
let $metadata := text { '{"collections" : ["April 2014"] }'}
let $json-content := text { '{ "key": "value" }' }
let $manifest :=
   <manifest>
     <!-- Request default metadata -->
     <part>
       <headers>
          <Content-Type>application/json</Content-Type>
          <Content-Disposition>inline;
category=metadata</Content-Disposition>
       </headers>
     </part>
     <!-- JSON document -->
     <part>
       <headers>
          <Content-Type>application/json</Content-Type>
          <Content-Disposition>inline; extension=json;
directory=/example/</Content-Disposition>
       </headers>
     </part>
     <!-- XML document -->
     <part>
       <headers>
          <Content-Type>application/xml</Content-Type>
          <Content-Disposition>inline; extension=xml;
directory=/example/</Content-Disposition>
       </headers>
     </part>
     </manifest>
return
xdmp:save($OUTPUT-FILENAME,
  xdmp:multipart-encode(
    $BOUNDARY, $manifest, ($metadata,$json-content,$xml-content)
  )
)
```

To use XML metadata instead of JSON metadata, replace the value of `$metadata` with the following declaration and change the Content-Type header for the first part to `application/xml`.

```
<rapi:metadata xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:collections>
    <rapi:collection>April 2014</rapi:collection>
  </rapi:collections>
</rapi:metadata>
```

5.2.11.3 Executing the Request

Use the following command to create the two documents. MarkLogic Server responds with status code 200 OK and includes the documents URIs and additional information about the created documents in the response body, as XML.

```
$ curl --anyauth --user user:password -X POST -i \
  --data-binary @/example/coll-body \
  -H "Content-type: multipart/mixed; boundary=BOUNDARY" \
  http://localhost:8000/LATEST/documents
...
HTTP/1.1 200 OK
Server: MarkLogic
Content-Type: text/xml; charset=UTF-8
Content-Length: 498
Connection: Keep-Alive
Keep-Alive: timeout=5
<?xml version="1.0"?>
<rapi:documents xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:document>
    <rapi:uri>/example/11572018736249878991.json</rapi:uri>
    <rapi:category>metadata</rapi:category>
    <rapi:category>content</rapi:category>
    <rapi:mime-type>application/json</rapi:mime-type>
  </rapi:document>
  <rapi:document>
    <rapi:uri>/example/5255172137182404803.xml</rapi:uri>
    <rapi:category>metadata</rapi:category>
    <rapi:category>content</rapi:category>
    <rapi:mime-type>application/xml</rapi:mime-type>
  </rapi:document>
</rapi:documents>
```

To get JSON results, set the Accept header to `application/json` or use the `format=json` request parameter. The following is an example of the equivalent JSON response data:

```
{ "documents": [
    { "uri": "/example/11572018736249878991.json",
      "mime-type": "application/json",
      "category": [
        "metadata",
        "content"
      ]
    },
    { "uri": "/example/5255172137182404803.xml",
      "mime-type": "application/xml",
      "category": [
        "metadata",
        "content"
      ]
    }
] }
```

For details, see “Response Overview” on page 254.

5.2.11.4 Verifying the Results

You can use the following combined query with the `/search` service to verify the documents added to the “April 2014” collection, or use the Explore feature of Query Console to examine the database contents. For more information, see “Specifying Dynamic Query Options with Combined Query” on page 179 or the *Query Console User Guide*.

```
$ cat coll-query.xml
<search xmlns="http://marklogic.com/appservices/search">
  <query>
    <collection-query>
      <uri>April 2014</uri>
    </collection-query>
  </query>
  <options>
    <transform-results apply="empty-snippet" />
    <return-metrics>false</return-metrics>
  </options>
</search>

$ curl --anyauth --user user:password -X POST -i \
  -H "Content-type: application/xml"
  -H "Accept: application/xml" -d @./coll-query.xml \
  http://localhost:8000/LATEST/search
```

You should get output similar to the following. The search results contain a match for the two documents previously created.

Format	Search Results
XML	<search:response snippet-format="empty-snippet" total="2" start="1" page-length="10" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns="" xmlns:search="http://marklogic.com/appservices/search"> <search:result index="1" **uri="/example/11572018736249878991.json"** path="fn:doc("/example/11572018736249878991.json")" score="0" confidence="0" fitness="0" href="/v1/documents?uri=%2Fexample%2F11572018736249878991.json" mimetype="application/json" format="json"> <search:snippet/> </search:result> <search:result index="1" **uri="/example/5255172137182404803.xml"** path="fn:doc("/example/5255172137182404803.xml")" score="0" confidence="0" fitness="0" href="/v1/documents?uri=%2Fexample%2F5255172137182404803.xml" mimetype="application/xml" format="xml"> <search:snippet/> </search:result> </search:response>

JSON	{ "snippet-format": "empty-snippet",
	"total": 2,
	"start": 1,
	"page-length": 10,
	"results": [
	{
	"index": 1,
	"uri": "/example/13110416418064966920.json",
	"path": "fn:doc(\"/example/13110416418064966920.json\")",
	"score": 0,
	"confidence": 0,
	"fitness": 0,
	"href": "/v1/documents?uri=%2Fexample%2F13110416418064966920.json",
	"mimetype": "application/json",
	"format": "json"
	},
	{
	"index": 1,
	"uri": "/example/55172137182404803.xml",
	"path": "fn:doc(\"/example/55172137182404803.xml\")",
	"score": 0,
	"confidence": 0,
	"fitness": 0,
	"href": "/v1/documents?uri=%2Fexample%2F55172137182404803.xml",
	"mimetype": "application/xml",
	"format": "xml"
	}
	]
	}

5.2.11.5 Extending the Example

You can include more than one category of metadata in a metadata part. For example, if you change the metadata part contents in this example to the following, then the created documents are readable by users with the `readers` role and have the user-defined `source` property, as well as being in the "April 2014" collection.

Format	Metadata

XML	`<rapi:metadata xmlns:rapi="http://marklogic.com/rest-api">`
	`  <rapi:collections>`
	`  <rapi:collection>April 2014</rapi:collection>`
	`  </rapi:collections>`
	`  <rapi:permissions>`
	`  <rapi:permission>`
	`  <rapi:role-name>readers</rapi:role-name>`
	`  <rapi:capability>read</rapi:capability>`
	`  </rapi:permission>`
	`  </rapi:permissions>`
	`  <prop:properties xmlns:prop="http://marklogic.com/xdmp/property">`
	`  <source>wikipedia.org</source>`
	`  </prop:properties>`
	`</rapi:metadata>`

JSON	`{`
	`  "collections": [ "April 2014" ],`
	`  "permissions": [{`
	`  "role-name": "readers",`
	`  "capabilities": ["read"]`
	`  }],`
	`  "properties": {`
	`  "source": "wikipedia.org"`
	`  }`
	`}`

For more details about working with metadata using the REST API, see the following topics:

- "Working with Metadata" on page 142
- "Adding Metadata" on page 57

5.2.12 Generating Example Payloads with XQuery

This section describes how to generate a `multipart/mixed` payload using XQuery and QueryConsole. Each bulk write example includes a suitable XQuery query for generating its payload. If your development environment includes other HTTP client tools or libraries for payload generation, you can use those instead.

A multipart payload must contain CRLF control characters in specific places. For details, see RFC 1341, availble at the following URL:

http://www.w3.org/Protocols/rfc1341/7_2_Multipart.html

Cutting and pasting text from the documentation will not preserve these control characters. Editing a multipart body with many text editors also corrupts the control characters. Therefore, it is best to generate the example multipart bodies programmatically.

The procedure below uses `xdmp:multipart-encode` to create a multipart payload and `xdmp:save` to save the result to a file you can use as input to the `curl` command.

The `xdmp:multipart-encode` builtin function accepts a manifest and a sequence of nodes as input. The nodes are the part contents. The manifest contains the headers for each part. The calling sequence is:

```
xdmp:multpart-encode($part-boundary, $manifest, $part-contents)
```

The manifest must contain a `<part/>` element for each content part in the `$part-contents` sequence, in the same order as the content parts. See the example below.

Follow this procedure to write a `multipart/mixed` payload to a file. If you are not familiar with Query Console, see the *Query Console User Guide*.

1. Open Query Console. For example, navigate to the following URL in your browser:

   ```
   http://yourhost:8000/qconsole
   ```

2. Create a new query by clicking on the “+” symbol at the top of the query editor.

3. Replace the default query contents with a body generation query. For example, replace the default query contents with the following query that generates 4 parts: JSON metadata, JSON content, XML metadata, and XML content.

   ```
   xquery version "1.0-ml";

   declare variable $OUTPUT-FILENAME := "/example/bulk-write-body";
   declare variable $BOUNDARY := "BOUNDARY";

   let $json-metadata1 := text { '{"quality" : 2 }'}
   let $json-doc1 := text { '{ "key": "some json content" }' }
   let $xml-metadata2 := document {
       <rapi:metadata xmlns:rapi="http://marklogic.com/rest-api">
         <rapi:quality>1</rapi:quality>
       </rapi:metadata>
     }
   let $xml-doc2 := document{ <root>some xml content</root> }
   let $manifest :=
     <manifest>
       <!-- JSON request default metadata -->
       <part>
         <headers>
            <Content-Type>application/json</Content-Type>
   ```

```
          <Content-Disposition>inline;
category=metadata</Content-Disposition>
      </headers>
    </part>
    <!-- JSON document -->
    <part>
       <headers>
          <Content-Type>application/json</Content-Type>
          <Content-Disposition>attachment;
filename="doc1.json"</Content-Disposition>
       </headers>
    </part>
    <!-- XML document-specific metadata -->
    <part>
      <headers>
        <Content-Type>application/xml</Content-Type>
        <Content-Disposition>attachment; filename="doc2.xml";
category=metadata</Content-Disposition>
      </headers>
    </part>
    <!-- XML content -->
    <part>
      <headers>
        <Content-Type>application/xml</Content-Type>
        <Content-Disposition>attachment;
filename="doc2.xml"</Content-Disposition>
      </headers>
    </part>
  </manifest>
return
xdmp:save($OUTPUT-FILENAME,
  xdmp:multipart-encode(
    $BOUNDARY, $manifest,
    ($json-metadata1,$json-doc1,$xml-metadata2,$xml-doc2)
  )
)
```

4. Change the `$OUTPUT-FILENAME` variable value to the full path where you want the payload to be saved. This directory must be writable by MarkLogic Server.

5. Click the Run button just below the query editor. The payload is saved to `$OUTPUT-FILENAME`.

If the output file is not created properly, check that the destination directory permissions permit MarkLogic Server to write a file to that location.

> **Note:** Depending on the editor you use, editing the saved file with a text editor can corrupt the payload by removing or converting the CRLF characters. Avoid editing the payload file unless you know your editor will preserve the CRLFs.

You can use the resulting output as input to the `curl` command using the `--data-binary` option. Use the same part boundary marker in your `curl` command that you used as the first parameter to `xdmp:http-encode` in the generation query.

For example, assuming you save the query output to `/example/bulk-write-body`, then the following command uses it as input to a bulk write request:

```
$ curl --anyauth --user user:password -X POST -i \
  --data-binary @/example/bulk-write-body \
  -H "Content-type: multipart/mixed; boundary=BOUNDARY" \
  http://localhost:8000/LATEST/documents
```

5.3 Reading Multiple Documents by URI

You can retrieve multiple documents by URI in a single request by sending a GET request to the `/documents` service with multiple `uri` parameters and an Accept header of `multipart/mixed`. The URL should be of the following form:

```
http://host:port/version/documents?uri=uri-1&uri=uri-2&...
```

For example, the following command retrieves content for two documents, `aardvark.xml` and `camel.xml`:

```
$ curl --anyauth --user user:password -X GET -i \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/documents?uri=aardvark.xml&uri=camel.x
ml'
```

MarkLogic Server returns a `multipart/mixed` response, with each part containing a requested document or its metadata. For details, see “Bulk Read Response Overview” on page 297.

The available request options are the same ones available when retrieving a single document and/or its metadata. For example:

- Use the `category` parameter to retrieve content, metadata (all or a subset), or both. For details, see “Retrieving Documents from the Database” on page 65.
- Use the `format` parameter to request metadata as either XML or JSON.
- Use the `transform` parameter to apply a read transform. For details, see “Transforming Content During Retrieval” on page 67.

  Note: Applying a transform creates an additional in-memory copy of each document, rather than streaming each document directly out of the database, so memory consumption is higher.

  Note: If you read both metadata and content for JSON documents, you should set format explicitly. Otherwise, the metadata format will vary depending upon whether your request includes one or multiple URIs.

For more details, see `GET:/v1/documents` in the *MarkLogic REST API Reference*.

5.4 Reading Multiple Documents Matching a Query

You can retrieve all documents that match a query by sending a GET or POST request to the `/search` or `/qbe` service with an Accept header value of `multipart/mixed`. You can retrieve content, metadata, or a combination of content and metadata for all matching documents.

- Overview of Bulk Read By Query
- Example: Using Query By Example (QBE)
- Example: Using a String, Structured, or Combined Query
- Extracting a Portion of Each Matching Document
- Including Search Results in the Response
- Paginating Results

5.4.1 Overview of Bulk Read By Query

To retrieve all documents from the database that match a query, use one of the following request methods, with the Accept header set to `multipart/mixed`:

Method	Description
`GET /`*`version`*`/search`	Retrieve documents matching a string and/or structured query with the query(s) in the `q` and/or `structuredQuery` request parameters.
`POST /`*`version`*`/search`	Retrieve documents matching a structured or combined query with the query in the POST body.
`GET /`*`version`*`/qbe`	Retrieve documents matching a Query By Example (QBE) with the query in the `query` request parameter.
`POST /`*`version`*`/qbe`	Retrieve documents matching a Query By Example (QBE) with the query in the POST body.

For example, the following command uses a string query to retrieve all documents containing the word bird:

```
$ curl --anyauth --user user:password -X GET -i \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  http://localhost:8000/LATEST/search?q=bird
```

MarkLogic Server returns a `multipart/mixed` response, with each part containing a matching document or its metadata. For details, see "Bulk Read Response Overview" on page 297.

For a more complete example, see "Example: Using a String, Structured, or Combined Query" on page 293.

Using a query to retrieve documents is the same as using `/search` and `/qbe` for a normal search, with the following exceptions:

- The Accept header must be `multipart/mixed`. This distinguishes your request from a normal search operation.
- No search response is included by default, but you can still request inclusion. For details, see "Including Search Results in the Response" on page 295.
- If you include a `transform` request parameter, the transform function is called on the returned documents and the search response but not on metadata. The transform must therefore be prepared to handle multiple kinds of input.
- Use the `category` parameter to specify the document data you want to retrieve: Content, metadata, or a metadata subset. You can retrieve a combination of these; for details, see "Bulk Read Response Overview" on page 297.
- Since a search response is not returned by default, the page range in the response is returned in vendor-specific response headers. For details, see "Paginating Results" on page 296.
- If there are no matches to the query, MarkLogic Server responds with status code 200 (OK) and an empty response body.

As with a normal search operation, you can include query options either by pre-installing them and naming them in the `options` request parameter, or by including them in a combined query in the POST body.

You can use a structured or combined query with the `/search` service to express complex queries. You can supply a structured query through the `structuredQuery` parameter on a GET request or in the body of a POST request. You can only supply a combined query in a POST request body.

See the following topics for more information on the query interfaces available through the REST API:

- "Querying Documents and Metadata" on page 151
- "Using Query By Example to Prototype a Query" on page 169
- "Specifying Dynamic Query Options with Combined Query" on page 179
- "Configuring Query Options" on page 208

5.4.2 Example: Using Query By Example (QBE)

This example demonstrates using a QBE to retrieve documents from the database using the `/qbe` service. You should be familiar with the basic `/qbe` REST API; for details, see "Using Query By Example to Prototype a Query" on page 169.

The following QBE matches documents with a kind XML element or JSON property with the value "bird":

Format	Query

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <kind>bird</kind>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{ "$query":`
	`  { "kind": "bird" }`
	`}`

The following command uses the above query to retrieve all matching documents. Since the request does not include any category parameters, only document content is returned, one document per part. The number of documents matching the input query is returned in the `vnd.marklogic.result-estimate` response header. This number is equivalent to `@total` on a search response. The document URI, document type, and part contents are returned in the Content-Disposition header; for details, see "Bulk Read Response Overview" on page 297.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X POST -d @query.xml -i \
  -H "Content-type: application/xml" \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/qbe'
...
HTTP/1.1 200 OK
Content-type: multipart/mixed; boundary=BOUNDARY
vnd.marklogic.start: 1
vnd.marklogic.pageLength: 10
vnd.marklogic.result-estimate: 6
Server: MarkLogic
Content-Length: 4222
Connection: Keep-Alive
Keep-Alive: timeout=5

--BOUNDARY
Content-Type: application/xml
Content-Disposition: attachment; filename=/animals/vulture.xml;
category=content; format=xml
...
```

To make the equivalent request using a JSON QBE, change the Content-Type header to `application/json`. Note that the format of a QBE (XML or JSON) can affect the kinds of documents that match the query. For details, see Scoping a Search by Document Type in the *Search Developer's Guide*.

To return metadata as JSON rather than XML, use the `format` request parameter. For example, the following command returns document quality expressed as JSON:

```
$ curl --anyauth --user user:password -X POST -d @query.xml -i \
  -H "Content-type: application/xml" \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/qbe?category=quality&format=json'
...
HTTP/1.1 200 OK
Content-type: multipart/mixed; boundary=BOUNDARY
vnd.marklogic.start: 1
vnd.marklogic.pageLength: 10
vnd.marklogic.result-estimate: 6
Server: MarkLogic
Content-Length: 4222
Connection: Keep-Alive
Keep-Alive: timeout=5

--BOUNDARY
Content-Type: application/json
Content-Disposition: attachment; filename=/animals/vulture.xml;
category=quality; format=json
...
```

If you use a GET request rather than a POST, you must URL-encode the `query` parameter value. For example, the following query uses the URL-encoded representation of the JSON query: `{ "$query": { "kind": "bird" } }`.

```
$ curl --anyauth --user user:password -X GET -i \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/qbe?query=%7B%20%22%24query%22%3A%20%7
B%20%22kind%22%3A%20%22bird%22%20%7D%20%7D'
```

Use the `options` request parameter to include persistent query options in your request. Use the `transform` parameter to apply a read transform. For example:

```
$ curl --anyauth --user user:password -X POST -d @query.json -i \
  -H "Content-type: application/json" \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/qbe?options=my-options&transform=my-en
richment
```

5.4.3 Example: Using a String, Structured, or Combined Query

This example demonstrates using a string, structured, or combined query to retrieve documents from the database using the `/search` service. The basic method is the same, no matter what query format you choose. You should be familiar with the basic `/search` REST API; for details, see "Querying Documents and Metadata" on page 151.

The following command uses a simple string query supplied through the `q` request parameter to retrieve all documents that contain "bird". Since the request does not include any `category` parameters, only document content is returned, one document per part. The number of documents matching the input query is returned in the `vnd.marklogic.result-estimate` response header. This number is equivalent to `@total` on a search response. The document URI, document type, and part contents are returned in the Content-Disposition header; for details, see "Bulk Read Response Overview" on page 297.

```
$ curl --anyauth --user user:password -X GET -i \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  http://localhost:8000/LATEST/search?q=bird
...
HTTP/1.1 200 OK
Content-type: multipart/mixed; boundary=BOUNDARY
vnd.marklogic.start: 1
vnd.marklogic.pageLength: 10
vnd.marklogic.result-estimate: 6
Server: MarkLogic
Content-Length: 1476
Connection: Keep-Alive
Keep-Alive: timeout=5

--BOUNDARY
Content-Type: text/xml
Content-Disposition: attachment; filename=/animals/vulture.xml;
category=content; format=xml
Content-Length: 93

<?xml version="1.0" encoding="UTF-8"?>
<animal><name>vulture</name><kind>bird</kind></animal>
...
```

To retrieve both documents and metadata, use the `category` parameter. The following example retrieves the document contents and the `quality` metadata. The response contains 2 parts for each matching document: A metadata part, immediately followed by the corresponding content part.

```
$ curl --anyauth --user user:password -X GET -i \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/search?q=bird&category=content&category=qualit
y'
...
HTTP/1.1 200 OK
Content-type: multipart/mixed; boundary=BOUNDARY
vnd.marklogic.start: 1
vnd.marklogic.pageLength: 10
```

```
vnd.marklogic.result-estimate: 6
Server: MarkLogic
Content-Length: 4222
Connection: Keep-Alive
Keep-Alive: timeout=5

--BOUNDARY
Content-Type: application/xml
Content-Disposition: attachment; filename=/animals/vulture.xml;
category=quality; format=xml
Content-Length: 299

?xml version="1.0" encoding="UTF-8"?>
<rapi:metadata uri="/animals/vulture.xml" ...>
  <rapi:quality>0</rapi:quality>
</rapi:metadata>
--BOUNDARY
Content-Type: text/xml
Content-Disposition: attachment; filename=/animals/vulture.xml;
category=content; format=xml
Content-Length: 93
...
```

To return the metadata as JSON rather than XML, add `format=json` to the request:

```
$ curl --anyauth --user user:password -X GET -i \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/search?q=bird&category=content&categor
y=quality&format=json'
```

You can supply a structured query through the `structuredQuery` parameter on a GET request or in the body of a POST request. You can only supply a combined query in a POST request body. For example, assuming `query.json` contains a structured or combined query, the following command retrieves metadata for documents containing “bird”.

```
$ curl --anyauth --user user:password -X POST -d @./query.json -i
  -H "Content-Type: application/xml" \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/search?category=metadata&format=json'
```

Use the `options` request parameter to include persistent query options in your request. Use the `transform` parameter to apply a read transform. For example:

```
$ curl --anyauth --user user:password -X GET -i \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  http://localhost:8000/LATEST/search?q=bird&options=my-options&transf
orm=my-enrichment
```

5.4.4 Extracting a Portion of Each Matching Document

You can use the `extract-document-data` query option to return selected portions of each matching document instead of the whole document.

For example, the following combined query specifies that the search should only return the portions of matching documents that match the path `/parent/body/target`.

```
<search xmlns="http://marklogic.com/appservices/search">
  <qtext>content</qtext>
  <options xmlns="http://marklogic.com/appservices/search">
    <extract-document-data selected="include">
      <extract-path>/parent/body/target</extract-path>
    </extract-document-data>
    <return-results>false</return-results>
  </options>
</search>
```

If one of the matching documents contains the following data:

```
{"parent": {
  "a": "foo",
  "body": { "target":"content" },
  "b": "bar"} }
```

Then the search returns the following sparse projection for this document:

```
{ "context":"fn:doc(\"/extract/doc2.json\")",
  "extracted":[{"target":"content"}]
}
```

For details, see Extracting a Portion of Matching Documents in the *Search Developer's Guide*.

If you use `extract-document-data` with a simple search, rather than a multi-document read, the sparse projections are embedded in the search response instead of returned as individual documents. That is, you get projected documents when the Accept header is `multipart/mixed` and embedded projections when the Accept header is `application/xml` or `application/json`.

5.4.5 Including Search Results in the Response

By default, no search response is included in the request response when you perform a bulk read based on a query. To return search results in addition to matching documents or metadata, set the `view` request parameter.

For the complete set of values accepted by the `view` parameter, see `GET:/v1/search` in the *MarkLogic REST API Reference*.

The search response is returned as XML by default. You can request a JSON search response by setting the `format` request parameter to `json`. If you request a JSON search response, your input query must also be JSON if it is not a string query. The `format` parameter also controls the format of returned metadata.

For example, the following command returns a search response, plus the documents matching the query:

```
$ curl --anyauth --user user:password -X GET -i \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/search?q=bird&view=results'
...
HTTP/1.1 200 OK
...

--BOUNDARY
Content-Type: application/xml
Content-Disposition: inline
Content-Length: 3177

<?xml version="1.0" encoding="UTF-8"?>
<search:response snippet-format="snippet" total="6" ...>
...
</search:response>
--BOUNDARY
Content-Type: text/xml
Content-Disposition: attachment; filename=/animals/vulture.xml;
category=content; format=xml
Content-Length: 93

<?xml version="1.0" encoding="UTF-8"?>
<animal><name>vulture</name><kind>bird</kind></animal>
...
```

5.4.6 Paginating Results

As with a normal search operation, you can use the `start` and `pageLength` request parameters to retrieve results in batches. Use `start` to specify the index of the first result to return and `pageLength` to control the number results to return.

By default, queries return the first 10 matches. That is, the default start position is 1 and the default page length is 10. You can fetch successive results by incrementing the start position by the page length in each call.

In a normal search operation, the page range included in the response body is returned via the `start` and `page-length` attributes (or JSON properties) on the search response. Since a bulk read via query does not necessarily include a search response, the page range is returned in the vendor-specific response headers `vnd.marklogic.start` and `vnd.marklogic.pageLength`. If a search response is returned by your request, the `start` and `page-length` value in the search response match the header values.

Similarly, an estimate of the total number of matches is returned in the vendor-specific header `vnd.marklogic.result-estimate`. This is equivalent to the `total` value in a search response. As with any search, the actual number of matches might be different, depending upon whether you use filtered or unfiltered search.

For more information, see the Search Developer's Guide and Fast Pagination and Unfiltered Searches in the *Scalability, Availability, and Failover Guide*.

The following example command fetches the first 5 documents containing “bird”. Notice that the response includes page range details in the `vnd.marklogic.*` headers.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
   'http://localhost:8000/LATEST/search?q=castle&start=1&pageLength=5'
...
HTTP/1.1 200 OK
Content-type: multipart/mixed; boundary=BOUNDARY
vnd.marklogic.start: 1
vnd.marklogic.pageLength: 5
vnd.marklogic.result-estimate: 16
Server: MarkLogic
Content-Length: 1225
Connection: Keep-Alive
Keep-Alive: timeout=5
...
```

To fetch the next set of matching documents, increment `start` by `pageLength` (5):

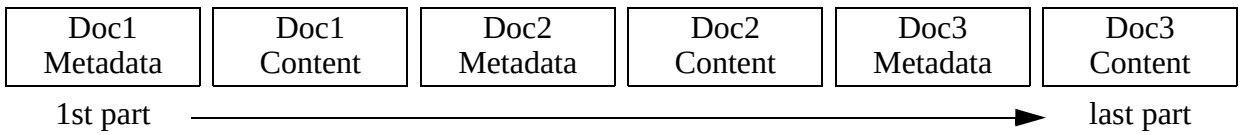
```
$ curl --anyauth --user user:password -X GET -i \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/search?q=bird&start=6&pageLength=5'
...
HTTP/1.1 200 OK
Content-type: multipart/mixed; boundary=BOUNDARY
vnd.marklogic.start: 6
vnd.marklogic.pageLength: 5
vnd.marklogic.result-estimate: 16
Server: MarkLogic
Content-Length: 1225
Connection: Keep-Alive
Keep-Alive: timeout=5
...
```

When the page length returned in `vnd.marklogic.pageLength` is less than the request `pageLength`, no more matching documents are available.

5.5 Bulk Read Response Overview

When you retrieve multiple documents in a single request using a list of URIs or a query, the results are returned in a `multipart/mixed` response that contains a part for each returned document or document metadata.

If you request both content and metadata, the content and metadata are returned in separate parts, with the metadata part immediately preceding the content part. If you request multiple metadata sub-categories, such as `quality` and `permissions`, then each metadata part contains all the requested subcategory values. The diagram below illustrates the response part stream for a request the returns both content and metadata.



When you retrieve multiple documents using a query, the response can also include the search response as the first part; for details, see "Including Search Results in the Response" on page 295.

The part Content-Type header contains the MIME type of the data in the part body. The Content-Type of a part is determined as follows:

- For a content part, the MIME type is determined by the MarkLogic Server MIME type mapping corresponding to the document URI extension. To review or change the MIME type mapping, see the "Mimetypes" subsection of the Groups section of the Admin Interface.
- For a metadata part, the MIME type is always `application/xml` or `application/json`. The default format is XML. Use the `format` request parameter to control the metadata MIME type.
- If a search response is included in the response, its MIME type is XML unless you use the `format=json` request parameter. For details, see "Including Search Results in the Response" on page 295.

The Content-Disposition header contains the source document URI, the kind of data in the part, and format of the data. The Content-Disposition header for a part has the following form:

```
Content-Disposition: attachment;filename=doc-uri;
category=data-category;...format=data-format
```

Where *`doc-uri`* is the database URI of the document from which the content or metadata in the part was extracted, and *`data-category`* is either `content`, `metadata`, or a specific metadata sub-category such as `permissions` or `quality`. For metadata, `format` is always `xml` or `json`. For content, `format` corresponds to the database document type: `xml`, `json`, `text`, or `binary`.

If optimistic locking or content versioning is enabled, the Content-Disposition header for a content part also contains a document version id of the form `versionId=`*`id`*; for details, see "Using Optimistic Locking to Update Documents" on page 131.

For example, the following bulk read request retrieves content, permissions, and document quality, so a content and a metadata part is returned for each document. The Content-Disposition header for each metadata part includes both `category=permissions` and `category=quality`.

```
$ curl --anyauth --user user:password -X GET -i \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/documents?category=content&category=pe
rmissions&category=quality&uri=...'
```

```
--BOUNDARY
Content-Type: application/xml
Content-Disposition: attachment; filename=...; category=permissions;
category=quality; format=xml
...
--BOUNDARY
Content-Type: text/xml
Content-Disposition: attachment; filename=...; category=content;
format=xml
...
```

The table below contains additional examples of returned part headers.

Part Description	Example Response Part Headers

XML content	`Content-type: application/xml`
	`Content-Disposition: attachment; filename=some.xml;`
	`category=content; format=xml`

JSON content	`Content-type: application/json`
	`Content-Disposition: attachment;filename=some.json;`
	`category=content; format=json`

Binary content	`Content-type: image/jpeg`
	`Content-Disposition: attachment;filename=some.jpg;`
	`category=content; format=binary`

All metadata for a document, expressed as XML	`Content-type: application/xml`
	`Content-Disposition: attachment; filename=some.xml;`
	`category=metadata; format=xml`

Selected metadata, expressed as json	`Content-type: application/json`
	`Content-Disposition: attachment; filename=some.json;`
	`category=quality; category=properties;`
	`format=json`

6.0 Managing Transactions

The REST Client API provides transaction control through the `/transactions` service and a `txid` parameter available on document manipulation and search requests. This section covers the following topics:

- Service Summary
- Overview of RESTful Transactions
- Creating a Transaction
- Associating a Transaction with a Request
- Committing or Rolling Back a Transaction
- Checking Transaction Status
- Managing Transactions When Using a Load Balancer

6.1 Service Summary

Many of the services available through the REST Client API allow you to perform an operation in the context of a specific transaction by passing a transaction id in a `txid` parameter. The `/transactions` service supports the creation and administration of these transactions.

The `/transactions` service supports the following operations:

Operation	Method	Description	More Information
Create	POST	Create a multi-statement transaction. See “Creating a Transaction” on page 303.	“Creating a Transaction” on page 303

The `/transactions/{txid}` service supports the following operations:

Operation	Method	Description	More Information
Commit or Rollblack	POST	Commit or rollback an open transaction.	“Committing or Rolling Back a Transaction” on page 304
Status	GET	Retrieve transaction status in XML or JSON.	“Checking Transaction Status” on page 305

6.2 Overview of RESTful Transactions

This section gives a brief introduction to the MarkLogic Server transaction model as it applies to the REST Client API. For a full discussion of the transaction model, see Understanding Transactions in MarkLogic Server in the *Application Developer's Guide*..

By default, each request to a REST Client API service is equivalent to a single statement transaction. That is, the request is evaluated as single transaction.

For example, when you update a document in the database by making a PUT request to the `/documents` service, the service effectively creates a new transaction, updates the document, commits the transaction, and then sends back a response. The update is immediately visible in the database and available to other requests.

The REST Client API also allows your application to take direct control of transaction boundaries so that multiple request are evaluated in the same transaction context. This is equivalent to the multi-statement transactions described in Multi-Statement Transaction Concept Summary in the *Application Developer's Guide*.

Using multi-statement transactions, you can make several update requests and commit them as a single transaction, ensuring either all or none of the updates appear in the database. The document manipulation and search capabilities of the REST Client API support multi-statement transactions.

To use multi-statement transactions:

1. Create a multi-statement transaction by sending a POST request to the `/transactions` service. The service returns a transaction id. See "Creating a Transaction" on page 303.

2. Evaluate one or more requests in the context of the transaction by including the transaction id in the `txid` request parameter. See "Associating a Transaction with a Request" on page 304.

3. Commit or rollback the transaction by sending a POST request to the `/transactions` service. See "Committing or Rolling Back a Transaction" on page 304.

If your application interacts with MarkLogic Server through a load balancer, you might need to include a load balancer session cookie in your requests to preserve session affinity. For details, see "Managing Transactions When Using a Load Balancer" on page 306.

When you explicitly create a transaction, you must explicitly commit or roll it back. Failure to do so leaves the transaction open until the request or transaction timeout expires. Open transactions can hold locks and consume system resources.

If the request or transaction timeout expires before a transaction is committed, the transaction is automatically rolled back and all updates are discarded. Configure the request timeout of the App Server using the Admin UI. Configure the timeout of a single transaction by setting the `timeLimit` request parameter during transaction creation.

Note that if you're using multi-statement transactions, you must create, use, and commit (or rollback) on the transaction using the same database. You cannot create a transaction on one database and then attempt to perform an operation such as read, write, or search using the transaction id and a different database.

6.3 Creating a Transaction

To create a transaction, send a POST request to the `/transactions` service with a URL of the form:

```
http://host:port/version/transactions
```

The service responds with a transaction id in the `Location` response header. Use the transaction id to control the transaction in which MarkLogic Server evaluates future requests; see "Associating a Transaction with a Request" on page 304.

The transaction id returned in the Location header is of the form. You can use this resource path to subsequently commit or rollback the transaction, as described in "Committing or Rolling Back a Transaction" on page 304.

```
/version/transactions/txnid
```

For example:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl -X POST -d "" --anyauth --user user:password \
    -H "Content-type: text/plain" \
    http://localhost:8000/LATEST/transactions
...
HTTP/1.1 303 See Created Transaction
Location: /v1/transactions/3148548124558550433
```

When supplying the transaction id to future requests, use only the `txnid` portion. For example:

```
$ curl -X PUT -T ./example.xml --anyauth --user user:password \
  'http://localhost:8000/LATEST/documents?uri=/xml/example.xml&txid=31
48548124558550433'
```

Transactions created using the `/transactions` service must be explicitly committed or rolled back. Failure to commit or rollback the transaction before the request timeout expires causes an automatic rollback. You can assign a shorter time limit to a transaction using the timeLimit URL parameter:

```
http://host:port/version/transactions?timeLimit=seconds
```

As a convenience for identifying a transaction in server status reports, you can assign a name to the transaction. The default transaction name is "client-txn". To assign a transaction, add the `name` URL parameter to your request:

```
http://host:port/version/transactions?name=txn_name
```

6.4 Associating a Transaction with a Request

Once you create a transaction, as described in "Creating a Transaction" on page 303, you can evaluate document manipulation and search requests in that transaction by supplying the transaction id wherever a `txid` URL parameter is supported.

For example, to perform a document update in the context of a specific transaction, include the transaction id in a PUT request to the `/documents` service:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl -X PUT -T ./example.xml --anyauth --user user:password \
  'http://localhost:8000/LATEST/documents?uri=/xml/example.xml&txid=31
48548124558550433'
```

Updates associated with an explicit transaction are visible to other requests executed within the same transaction, but they are not visible outside the transaction until the transaction is committed. See "Committing or Rolling Back a Transaction" on page 304.

Note: The database context in which you evaluate the request must be the same as the database context in which the transaction was created.

6.5 Committing or Rolling Back a Transaction

To commit a transaction created by making a POST request to the `/transactions` service, send a POST request to the `/transactions/{txid}` service with a URL of the form:

```
http://host:port/version/transactions/txid?result=outcome
```

Where *`txid`* is a transaction id returned in the `Location` header of the transaction creation request response, and *`outcome`* is either `commit` or `rollback`.

MarkLogic Server responds with a 204 when a transaction is successfully committed or rolled back, or if the transaction no longer exists.

Transactions created using the `/transactions` service should be explicitly commited or rolled back. Failure to commit or rollback the transaction before the request timeout expires causes an automatic rollback. Leaving transactions open unnecessarily potentially holds locks on documents and consume system resources.

6.6 Checking Transaction Status

To query the status of a transaction, send a GET request to the `/transactions` service with a URL of the form:

```
http://host:port/version/transactions/txid
```

Where `txid` is a transaction id returned in the `Location` header of the transaction creation request response.

MarkLogic Server responds with transaction status information in XML or JSON, depending upon the format requested through the HTTP `Accept` header or the `format` URL `parameter`. The default format is XML. If both the `Accept` header and the `format` parameter are set, the format parameter takes precedence.

To request a specific format, set the `Accept` header to `application/json` or `application/xml`, or set the `format` URL parameter to `xml` or `json`. The following example requests JSON output using the `Accept` header:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl -X GET --anyauth --user user:password \
    -H "Accept: application/json" \
    http://localhost:8000/LATEST/transactions/216104635458437451
```

The XML status information returned in the response takes the following form:

```
<rapi:transaction-status xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:host>
    <rapi:host-id>1506593501555860493</rapi:host-id>
    <rapi:host-name>somehost</rapi:host-name>
  </rapi:host>
  <rapi:server>
    <rapi:server-id>17994162681305741146</rapi:server-id>
    <rapi:server-name>App-Services</rapi:server-name>
  </rapi:server>
  <rapi:database>
    <rapi:database-id>8896678293064963766</rapi:database-id>
    <rapi:database-name>Documents</rapi:database-name>
  </rapi:database>
  <rapi:transaction-id>216104635458437451</rapi:transaction-id>
  <rapi:transaction-name>client-txn</rapi:transaction-name>
  <rapi:transaction-mode>update</rapi:transaction-mode>
  <rapi:transaction-timestamp>0</rapi:transaction-timestamp>
  <rapi:transaction-state>idle</rapi:transaction-state>
  <rapi:canceled>false</rapi:canceled>
  <rapi:start-time>2012-08-18T17:23:00-07:00</rapi:start-time>
  <rapi:time-limit>600</rapi:time-limit>
  <rapi:max-time-limit>3600</rapi:max-time-limit>
  <rapi:user>7071164303237443533</rapi:user>
  <rapi:admin>true</rapi:admin>
</rapi:transaction-status>
```

The JSON status information returned in the response takes the following form:

```
{
  "rapi:transaction-status": {
    "rapi:host": {
      "rapi:host-id": "1506593501555860493",
      "rapi:host-name": "somehost"
    },
    "rapi:server": {
      "rapi:server-id": "17994162681305741146",
      "rapi:server-name": "App-Services"
    },
    "rapi:database": {
      "rapi:database-id": "8896678293064963766",
      "rapi:database-name": "Documents"
    },
    "rapi:transaction-id": "216104635458437451",
    "rapi:transaction-name": "client-txn",
    "rapi:transaction-mode": "update",
    "rapi:transaction-timestamp": "0",
    "rapi:transaction-state": "idle",
    "rapi:canceled": "false",
    "rapi:start-time": "2012-08-18T17:23:00-07:00",
    "rapi:time-limit": "600",
    "rapi:max-time-limit": "3600",
    "rapi:user": "7071164303237443533",
    "rapi:admin": "true"
  }
}
```

You can also check transaction status using the Admin Interface or `xdmp:host-status`.

6.7 Managing Transactions When Using a Load Balancer

This section applies only to client applications that use multi-statement (multi-request) transactions and interact with a MarkLogic Server cluster through a load balancer. Requests that include a transaction id (`txid`) request parameter are part of a multi-statement transaction.

When you use a load balancer, it is possible for requests from your application to MarkLogic Server to be routed to different hosts, even within the same session. This has no effect on most interactions with MarkLogic Server, but all requests that are part of the same multi-statement transaction must be routed to the same host within your MarkLogic cluster. This consistent routing through a load balancer is called *session affinity.*

Most load balancers provide a mechanism that supports session affinity. This usually takes the form of a session cookie that originates on the load balancer. The client acquires the cookie from the load balancer, and passes it on any requests that belong to the session. You can use a load balancer session cookie to preserve session affinity across requests in the same transaction. The exact steps required to configure a load balancer to generate session cookies depends on the load balancer. Consult your load balancer documentation for details.

The following steps describe to use load balancer cookies to preserve session affinity across requests in the same transaction. A graphical representation of this process follows.

1. Create a multi-statement transaction.

    a. Send a POST request to the `/transactions` service. The load balancer directs the request to some host in your MarkLogic cluster.

    b. The response from MarkLogic includes a transaction id in the Location header, and the load balancer adds a session cookie. The client application caches the transaction id and session cookie for use on subsequent requests in the same transaction.

2. Perform operations in the context of the multi-statement transaction.

    a. Send a request to evaluate as part of the transaction, such as a document insertion. Include the transaction id (as a request parameter) and session cookie in your request. The load balancer uses the cookie to direct the request to the same host that originally created the transaction.

    b. MarkLogic responds without inculding any cookies, but the load balancer adds a session cookie.

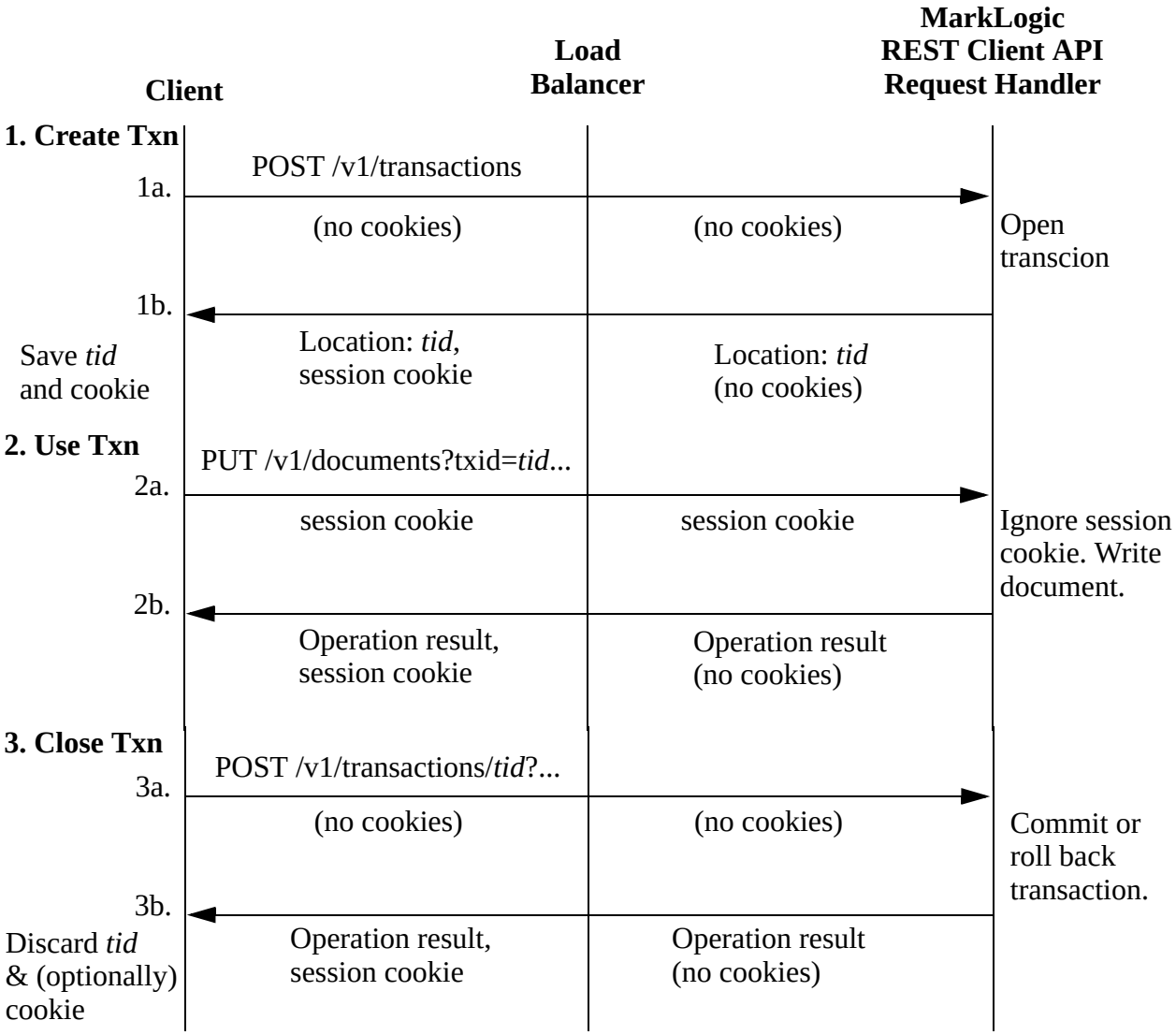
3. When ready, close the transaction by committing or rolling back.

    a. Send a POST request to the `/transactions/{txid}` service that includes the transaction id (in the URL) and session cookie.

    b. MarkLogic closes the transaction and responds with no cookies. The load balancer adds a session cookie. The client application discards the cached transaction id and, usually, the session cookie because the transaction is no longer valid.

The following diagram illustrates the steps described above. The steps in the diagram correspond to the steps in the procedure. The transaction id is represented in the diagram by *tid*.

Using Multi-Statement Transactions with a Load Balancer



The load balancer attaches a session cookie to responses to all REST Client API requests, but your application is only required to pass it back as part of a multi-statement transaction operation.

A REST Client API application can use a session cookie beyond the boundaries of a single multi-statement transaction by attaching the session cookie to other requests. However, most operations do not require session affinity. Omitting the session cookie from requests that are not part of a multi-statement transaction enables the load balancer to distribute work across your MarkLogic cluster more effectively.

You can have multiple multi-statement transactions open concurrently, and you can intermix single-statement transaction requests and multi-statement transaction requests, as long as you preserve the correct association between a transaction id and its session cookie in the multi-statement transaction requests.

You must not send a request that includes a transaction id for one transaction with a session cookie associated with a different one. Some HTTP connection pooling implementations may not properly handle session cookies. For example, a pooled connection might return a cookie that is not the correct one for the transaction you are trying to use.

7.0 Alerting

The following topics are covered:

- Summary of /alert Services
- Alerting Pre-Requisites
- Alerting Concepts
- Defining an Alerting Rule
- Installing an Alerting Rule
- Testing for Matches to Alerting Rules
- Retrieving Rule Definitions
- Testing for the Existence of Rule
- Deleting a Rule

7.1 Summary of /alert Services

The table below provides a brief summary of the alerting services provided by REST Client API:

Operation	Methods	Description
`/alert/rules`	`GET`	Retrieve definitions of all installed alerting rules.

`/alert/rules/{name}`	`HEAD`
	`GET`
	`PUT`
	`DELETE`
		Maintain alerting rules. You can create/modify, delete, and test for the existence of rules by name. Rules can be expressed in XML or JSON.

`/alert/match`	`GET` `POST`	Test one or more documents for matches against all installed rules or a specified subset of rules. The input documents can be specified using a query, URIs, or by passing the document content into the request. You can define an XQuery function to transform the match results in an application-specific way.

7.2 Alerting Pre-Requisites

You should also enable “fast reverse searches” on the content database associated with your REST API instance. Enable fast reverse searches using the Admin Interface, as described in Indexes for Reverse Queries in *Search Developer’s Guide*, or using the XQuery function `admin:database-set-fast-reverse-searches`.

The `rest-writer` role or equivalent privileges is required to create and delete alerting rules. All other operations require the `rest-reader` or equivalent privileges. Additional privileges may be required to access input documents during rule match operations.

7.3 Alerting Concepts

An *alerting application* is one that takes action whenever content matches a pre-defined set of criteria. For example, send an email notification to a user whenever a document about influenza is added to the database. In this case, the criteria might be “the abstract contains the word influenza”, and the action is “send an email”.

MarkLogic Server supports server-side alerting through the XQuery API and Content Processing Framework (CPF), and client-side alerting through the REST and Java APIs.

A server-side alerting application usually uses a “push” model. You register alerting rules and XQuery action functions with MarkLogic Server. Whenever content matches the rules, MarkLogic Server evaluates the action functions. For details, see Creating Alerting Applications in *Search Developer’s Guide*.

By contrast, a client-side alerting application uses a “pull” alerting model. You register alerting rules with MarkLogic Server, as in the push model. However, your application must poll MarkLogic Server for matches to the configured rules, and the application initiates actions in response to matches. This is the model used by the REST Client API.

An *alerting rule* is a query used in a reverse query to determine whether or not a search using that query would match a given document. A normal search query asks “What documents match these search criteria?” A reverse query asks “What criteria match this document?” In the influenza example above, you might define a rule that is a word query for “influenza”, with an element constraint of `<abstract/>`. Queries associated with alerting rules are stored in the database.

MarkLogic Server provides fast, scalable rule matching by storing queries in alerting rules in the database and indexing them in the reverse query index. You must explicitly enable “fast reverse searches” on your content database to take advantage of the reverse quer index. For details, see Indexes for Reverse Queries in *Search Developer’s Guide*.

Use the procedures described in this chapter to create and maintain search rules and to test documents for matches to the rules installed in your REST API instance. Determining what actions to take in response to a match and initiating those actions is left to the application.

7.4 Defining an Alerting Rule

An alerting rule is defined by a name, a query, and optional metadata. The core of a rule is the combined query that describes the search criteria to use in future match operation. A combined query is a `search` wrapper around a string and/or structured query plus query options; for details, see "Specifying Dynamic Query Options with Combined Query" on page 179.

All rules must have a name. You can either specify the name explicitly in the rule definition, or have MarkLogic Server internally add a name derived from the *`name`* path step in the URI used to install the rule in the REST API instance. If you specify an explicit name in the definition, it must match the *`name`* in the PUT request URI.

Use the following template for defining the rule in the request body.

Format	Alerting RuleTemplate

XML	`<rapi:rule xmlns:rapi="http://marklogic.com/rest-api">`
	`  <rapi:name>`*`rule-name`*`</rapi:name>`
	`  <rapi:description>`*`optional description`*`</rapi:description">`
	`  <rapi:rule-metadata>`
	`  `*`optional user-defined metadata`*
	`  </rapi:rule-metadata>`
	`  <!--`*` required combined query `*`-->`
	`</rapi:rule>`

JSON	`{"rule":`
	`  "name": "`*`rule-name`*`",`
	`  "description": "`*`optional description`*`",`
	`  "rule-metadata":`*` optional user-defined metadata`*`,`
	`  "search" :`*` a combined query`*
	`}`

For example, the following rule enables you to test whether a document or a set of documents matches the string query "xdmp" in a case-sensitive search.

```
<rapi:rule xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:name>Example Rule</rapi:name>
  <search:search
      xmlns:search="http://marklogic.com/appservices/search">
    <search:qtext>xdmp</search:qtext>
    <search:options>
      <search:term>
        <search:term-option>case-sensitive</search:term-option>
      </search:term>
    </search:options>
  </search:search>
</rapi:rule>
```

You can include an optional description and metadata. This data is not used by MarkLogic Server, but it is returned in match results and when you fetch back a rule definition. The structure of the rule metadata is user-defined. The following example rule adds a description and an author to the above rule:

```
<rapi:rule xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:name>Example Rule</rapi:name>
  <search:search
      xmlns:search="http://marklogic.com/appservices/search">
    <search:qtext>xdmp</search:qtext>
    <search:options>
      <search:term>
        <search:term-option>case-sensitive</search:term-option>
      </search:term>
    </search:options>
  </search:search>
  <rapi:description>An example rule.</rapi:description>
  <rapi:rule-metatdata>
    <author>me</author>
  </rapi:rule-metadata>
</rapi:rule>
```

The following example shows an equivalent rule expressed as JSON.

```
{ "rule": {
  "name" : "json-example",
  "search" : {
    "qtext" : "xdmp",
    "options" : {
      "term" : { "term-option" : "case-sensitive" }
    }
  },
  "description": "A JSON example rule.",
  "rule-metadata" : { "author" : "me" }
}}
```

Note: If you insert metadata as JSON and retrieve it as XML, reference the XML metadata elements in the namespace `http://marklogic.com/rest-api`.

7.5 Installing an Alerting Rule

Before you can perform a match operation against an alerting rule, you must install the rule in your REST API instance. To persist a rule in your REST API instance, send a PUT request to the `/alert/rules/{name}` service with a URL of the following form. The request body must contain an XML or JSON rule definition of the form described in “Defining an Alerting Rule” on page 313.

```
http://host:port/version/alert/rules/name
```

Where *name* is the name of the rule. If you specify an explicit name in the rule definition, it must match `name` in the PUT request URI. If you do not specify an explicit name in the rule definition, MarkLogic Server adds a name corresponding to the `name` in the request URI.

The following example command installs a rule under the name “example”, assuming the file `rule.xml` contains the rule definition:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -d @./rule.xml \
    -i -H "Content-type: application/xml" \
    'http://localhost:8000/LATEST/alert/rules/example'
```

To install a rule expressed as JSON, set the Content-type header to `application/json` or set the format request parameter to `json`. For example:

```
$ curl --anyauth --user user:password -X PUT -d @./rule.json \
    -i -H "Content-type: application/json" \
    'http://localhost:8000/LATEST/alert/rules/json-example'
```

7.6 Testing for Matches to Alerting Rules

Once you install alerting rules in your REST API instance, use the `/alert/match` service to determine which rules match one or more input documents. You can select the input documents using a database query or database URIs, or by passing a transient document.

This section covers the following topics:

- Basic Steps
- Identifying Input Documents Using a Query
- Identifying Input Documents Using URIs
- Matching Against a Transient Document
- Filtering Match Results
- Transforming Match Results

7.6.1 Basic Steps

To test one or more documents for a match against the rules, send a GET or POST request to the `/alert/match` service with a URL of the following form:

```
http://host:port/version/alert/match
```

The following example command returns all rules that match the document with the URI `/example/doc.xml`:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
```

```
-i -H "Accept: application/xml" \
'http://localhost:8000/LATEST/alert/match?uri=/example/doc.xml'
```

You must use one of the following methods to select the document(s) to test for a match:

- Query: Use a string, structured, or combined query to select documents in the database. For details, see “Identifying Input Documents Using a Query” on page 317. Use one of the following forms of request:

  ```
  GET
  http://host:port/version/alert/match?q=q1&structuredQuery=q2&options=q
  uery-options

  POST http://host:port/version/alert/match
  (post body contains a combined query)
  ```

- URI: Use one or more document URIs to select documents in the database. For details, see “Identifying Input Documents Using URIs” on page 318. Use a request of the following form:

  ```
  GET http://host:port/version/alert/match?uri=document-uri
  ```

- Content: Supply a transient document in the request body instead of using documents in the database. For details, see “Matching Against a Transient Document” on page 318. Use a request of the following form:

  ```
  POST http://host:port/version/alert/match
  (request body contains an XML document)
  ```

The response is a `rules` object that contains the definition of every rule that matches at least one of the input documents. You can request output in either XML or JSON using the HTTP Accept header. The data in the response body has the following form:

Format	Alerting RuleTemplate

XML	`<rapi:rules xmlns:rapi="http://marklogic.com/rest-api">`
	`  <rapi:rule>`*rule definition*`</rapi:rule>`
	`  <rapi:rule>`*rule definition*`</rapi:rule>`
	`</rapi:rules>`

JSON	`{"rules": [`
	`  {"rule": {` *rule definition* `},`
	`  {"rule": {` *rule definition* `}`
	`] }`

For the structure of a rule definition, see “Defining an Alerting Rule” on page 313.

The results do not indicate which input documents match each rule. If you need to know this, you should match against one document at a time.

7.6.2 Identifying Input Documents Using a Query

To use a query to select the documents to test for rule matches, do one of the following:

- Make a GET request that includes a string query in the `q` request parameter and/or a structured query in the `structuredQuery` request parameter. If you include both, the two queries are AND'd together.
- Make a POST request that includes a combined query in the request body.

The syntax and semantics of the queries are the same as when making a GET or POST request to the `/search` service. For details, see "Querying Documents and Metadata" on page 151 and "Specifying Dynamic Query Options with Combined Query" on page 179.

The following example GET request uses a string query to select all documents that contain the search terms "cat" and "dog". The search is performed using the persistent query options stored under the name "my-options".

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    -i -H "Accept: application/xml" \
    'http://localhost:8000/LATEST/alert/match?options=my-options&q=cat
AND dog'
```

The following example POST request uses a combined query to perform a similar match. Note that query options can be defined at request time when you use a combined query.

```
$ cat match-body.xml
<search xmlns="http://marklogic.com/appservices/search">
  <qtext>xdmp</qtext>
  <options>
    <term>
      <term-option>case-sensitive</term-option>
    </term>
  </options>
</search>

$ curl --anyauth --user user:password -X POST -d @./match-body.xml \
    -i -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/alert/match
```

To express the combined query in JSON, use a query and command similar to the following:

```
$ cat match-body.json
{ "search" : {
    "qtext" : "xdmp",
    "options" : {
      "term" : { "term-option" : "case-sensitive" }
```

```
    }
}}

$ curl --anyauth --user user:password -X POST -d @./match-body.json \
    -i -H "Content-type: application/json" \
    http://localhost:8000/LATEST/alert/match
```

7.6.3 Identifying Input Documents Using URIs

To specify the input documents for an alert match using URIs, include one or more `uri` request parameters in a GET request to `/alert/match`. Each URI must identify a document, not a database directory.

The following example command tests the documents `/example/doc1.xml` and `/example/doc2.xml` for rule matches.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    -i -H "Accept: application/xml" \
    'http://localhost:8000/LATEST/alert/match?uri=/example/doc1.xml&ur
i=/example/doc2.xml'
```

7.6.4 Matching Against a Transient Document

You can test for rules matches against a document that is not stored in the database by making a POST request to `/alert/match` with the document in the request body.

The following example command tests which rules match the content in the local file `transient.xml`:

```
$ cat > transient.xml
<function>
  <prefix>xdmp</prefix>
  <name>document-delete</name>
</function>

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X POST \
    -i -H "Content-type: application/xml" -d @./transient.xml \
    'http://localhost:8000/LATEST/alert/match'
```

You can use either an XML or JSON document as input. The following example command use JSON input:

```
$ curl --anyauth --user user:password -X POST \
    -i -H "Content-type: application/json" -d @./transient.json \
    'http://localhost:8000/LATEST/alert/match'
```

7.6.5 Filtering Match Results

By default, the response to an alert match includes all matching rules. Use the `rule` request parameter to limit the results to a subset of the matching rules. For example, the response to the following command will include at most the definition of the rules named “one” and “two”, even if more rules match the input document.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    -i -H "Accept: application/xml" \
    'http://localhost:8000/LATEST/alert/match?uri=/example/doc.xml&rul
e=one&rule=two'
...
<rapi:rules xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:rule>
    <rapi:name>one</rapi:name>
    <rapi:description>Rule 1</rapi:description>
    <search:search
        xmlns:search="http://marklogic.com/appservices/search">
      <search:qtext>xdmp</search:qtext>
    </search:search>
  </rapi:rule>
</rapi/rules>
```

The following is the equivalent command using JSON output.

```
$ curl --anyauth --user user:password -X GET \
    -i -H "Accept: application/json" \
    'http://localhost:8000/LATEST/alert/match?uri=/example/doc.xml&rul
e=one&rule=two'
...
{ "rules": [
    { "rule": {
        "name": "one",
        "description": "Rule 1",
        "search": { "qtext": ["xdmp"] }
    } }
] }
```

7.6.6 Transforming Match Results

You can make arbitrary changes to the response from match request by applying a user-defined XQuery transformation function to the response. This section covers the following topics:

- Writing a Match Result Transform
- Using a Match Result Transform

7.6.6.1 Writing a Match Result Transform

Alert match transforms use the same interface and framework as content transformations applied during document ingestion, described in “Working With Content Transformations” on page 323.

Your transform function receives the raw XML match result data as input, such as a document with a <rapid:rules/> root element. For example:

```
<rapi:rules xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:rule>
    <rapi:name>one</rapi:name>
    <rapi:description>Rule 1</rapi:description>
    <search:search
        xmlns:search="http://marklogic.com/appservices/search">
      <search:qtext>xdmp</search:qtext>
    </search:search>
  </rapi:rule>
</rapi/rules>
```

If your function produces XML output and the client application requested JSON output, MarkLogic Server will transform your output to JSON only if one of the following conditions are met.

- Your function produces an XML document that conforms to the "normal" output from the search operation. For example, a document with a `<rapi:rules/>` root element whose contents are changed in a way that preserves the normal structure.
- Your function produces an XML document with a root element in the namespace `http://marklogic.com/xdmp/json/basic` that can be transformed by `json:transform-to-json`.

Under all other circumstances, the response body contains exactly the output returned by your transform function.

7.6.6.2 Using a Match Result Transform

Follow this procedure to apply a transformation:

1. Create a transform function according to the interface described in "Writing Transformations" on page 323.

2. Install your transform function on the REST API instance following the instructions in "Installing Transformations" on page 331.

3. Apply your transform to a request to the `/alert/match` service by specifying the name in the `transform` request parameter, as described in "Applying Transformations" on page 333.

If the transform function expects additional parameters, specify them by name using `trans:{name}` request parameters.

The following example command performs an alert match using the document with URI `/example/doc.xml` as input. The match results are transformed by calling the XQuery transform function installed with the name `my-alert-txfm`. A single parameter named `my-param` is passed into the transform function.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    -i -H "Accept: application/xml" \
    'http://localhost:8000/LATEST/alert/match?uri=/example/doc.xml&tra
nsform=my-alert-txfm&trans:my-param=value'
```

7.7 Retrieving Rule Definitions

You can retrieve the definition of all installed rules or the definition of a single installed rule identified by name.

To retrieve the definitions of all installed rules, send a GET request to the `/alert/rules` service with a URL of the following form:

```
http://host:port/version/alert/rules
```

To retrieve the definition of a single rule by name, send a GET request to the `/alert/rules/{name}` service with a URL of the following form:

```
http://host:port/version/alert/rules/{name}
```

Use the HTTP Accept header or the `format` request parameter to request results in either XML or JSON.

The following example command returns the definition of the rule named “Rule1”, in XML:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    -i -H "Accept: application/xml" \
    'http://localhost:8000/LATEST/alert/rules/Rule1'

<rapi:rule xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:name>Rule1</rapi:name>
  <search:search
      xmlns:search="http://marklogic.com/appservices/search">
    <search:qtext>xdmp</search:qtext>
  </search:search>
</rapi:rule>
```

7.8 Testing for the Existence of Rule

To test for the existence of a particular rule without fetching back the defining, send a HEAD request to the `/alert/rules/{name}` service with a URL of the following form:

```
http://host:port/version/alert/rules/{name}
```

If the named rule exists, MarkLogic Server responds with status code 200 (OK). If no such rule exists, MarkLogic Server responds with status 404 (Not Found).

7.9 Deleting a Rule

To delete a previously installed rule, send a DELETE request to the `/alert/rules/{name}` service with a URL of the following form:

```
http://host:port/version/alert/rules/{name}
```

8.0 Working With Content Transformations

The REST Client API allows you to create custom content transformations and apply them during operations such as document ingestion and search. Transforms can be implemented as XQuery library modules, XSLT stylesheets, or server-side JavaScript modules, and can accept transform-specific parameters. Transforms can be applied on any REST Client API service that accepts a `transform` parameter.

This chapter covers the following topics:

- Writing Transformations
- Installing Transformations
- Applying Transformations
- Discovering Transformations
- Retrieving the Implementation of a Transformation
- JavaScript Example: Adding a JSON Property During Ingestion
- XQuery Example: Adding an Attribute During Ingestion
- XSLT Example: Adding an Attribute During Ingestion
- XQuery Example: Modifying Document Type During Ingestion

8.1 Writing Transformations

This section covers the following topics related to authoring transform functions:

- Guidelines for Writing Transforms
- Writing JavaScript Transformations
- Writing XQuery Transformations
- Writing XSLT Transformations
- Expected Input and Output
- Reporting Errors
- Context Map Keys
- Controlling Transaction Mode

8.1.1 Guidelines for Writing Transforms

Keep the following guidelines in mind when authoring a transform:

- A transform is one step in a document processing pipeline controlled by the REST Client API framework. As such, transforms typically should not have side-effects.

- A transform operates on an in-memory input document. MarkLogic Server builtin functions that operate on in-database documents, such as `xdmp:node-replace` or `xdmp.nodeReplace`, cannot be applied directly to the input document. To modify the structure of the input document, perform the modifications during construction of a copy of the input content.
- A transform executes in the transaction context of the HTTP request on whose behalf it is called.
- A transform should not alter the HTTP request context. For example, you should not set the response code or headers directly. The REST Client API framework manages the request context.
- Report errors by throwing one of the exceptions provided by the REST Client API. For details, see “Reporting Errors” on page 328.

8.1.2 Writing JavaScript Transformations

To create a JavaScript transformation, implement an server-side JavaScript module that implements function with the following interface and exports it with the name `transform`:

```
function yourTransformName(context, params, content)
{
    ...
};

exports.transform = yourTransformName;
```

Your implementation must use the MarkLogic server-side JavaScript dialect. For details, see the *JavaScript Reference Guide*.

The table below describes the parameters:

Parameter	Description
`context`	An object containing service request context information such as input document types and URIs, and output types accepted by the caller. For details, see “Context Map Keys” on page 330.
`params`	An object containing the extension-specific parameters, if any. The object contains an object property for each request parameter with a `trans:` prefix. If you specify the same parameter name more than once, the property value is an array. You can optionally define parameter name and type metadata when installing the extension. For details, see “Installing Transformations” on page 331.
`content`	The input document to which to apply the transformation. For example, the document being inserted into the database if the transform is applied during ingestion, or a `search:response` if the transform is applied during a search. For details, see “Expected Input and Output” on page 328.

Note that `content` is document node, not a JavaScript object. If `content` is a JSON document, you must call `toObject` on it before you can manipulate the content as a JavaScript object or modify it. For example:

```
var mutableDoc = content.toObject();
```

The expected output from your transform function depends upon the situation in which it is used. For details, see “Expected Input and Output” on page 328.

For a complete example, see “JavaScript Example: Adding a JSON Property During Ingestion” on page 335.

8.1.3 Writing XQuery Transformations

To create an XQuery transformation, implement an XQuery library module in the following namespace:

```
http://marklogic.com/rest-api/transform/yourTransformName
```

Where *yourTransformName* is the name used to refer to the transform in subsequent REST requests, such as installing and applying the transform.

The module must implement a public function called `transform` with the following interface:

```
declare function yourNS:transform(
  $context as map:map,
  $params as map:map,
  $content as document-node())
as document-node()
```

The table below describes the parameters:

Parameter	Description
`$context`	Service request context information such as input document type and URI, and output types accepted by the caller. For details, see “Context Map Keys” on page 330.
`$params`	Transformation specific parameters. The map contains a key for each transform parameter passed to the request for which the transform is applied. Define parameters when installing the transformation. For details, see “Installing Transformations” on page 331.
`$content`	The input document to which to apply the transformation. For example, the document being inserted into the database if the transform is applied during ingestion, or a `search:response` if the transform is applied during a search. For details, see “Expected Input and Output” on page 328.

The `map:map` type is a MarkLogic Server extended XQuery type representing a key-value map. For details, see Using the map Functions to Create Name-Value Maps in the *Application Developer's Guide*.

The expected output from your transform function depends upon the situation in which it is used. For details, see “Expected Input and Output” on page 328.

For a complete example, see “XQuery Example: Adding an Attribute During Ingestion” on page 338.

8.1.4 Writing XSLT Transformations

To create an XSLT transformation, implement an XSLT stylesheet that accepts the following parameters:

Parameter	Type	Description
`$context`	`map:map`	Service request context information such as input document type and URI, and output types accepted by the caller. For details, see "Context Map Keys" on page 330.
`$params`	`map:map`	Transformation specific parameters. The map contains a key for each transform parameter passed to the request for which the transform is applied. Define any parameters when installing the transformation. For details, see "Installing Transformations" on page 331.

The `map:map` type is a MarkLogic Server extended XQuery type representing a key-value map. For details, see Using the map Functions to Create Name-Value Maps in the *Application Developer's Guide*.

To use the `map` type in your stylesheet, include the namespace `http://marklogic.com/xdmp/map`. Reference the map contents using the `map:get` XQuery function.

The following example illustrates declaring the `map` namespace and `$context` and `$params` parameters, and referencing values in the `$params` map:

```
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:example="http://marklogic.com/rest-api/example/transform"
    xmlns:map="http://marklogic.com/xdmp/map">
<xsl:param name="context" as="map:map"/>
<xsl:param name="params"  as="map:map"/>
<xsl:template match="/*">
  <xsl:copy>
    <xsl:attribute name='{(map:get($params,"name"),"transformed")[1]}'
      select='(map:get($params,"value"),"UNDEFINED")[1]'/>
...
```

The expected output from your transform function depends upon the situation in which it is used. For details, see "Expected Input and Output" on page 328.

For a complete example, see "XSLT Example: Adding an Attribute During Ingestion" on page 340.

8.1.5 Expected Input and Output

The input document (`$content`) to a transform is always a document. The format and contents of the document depend on where the transform is used. The following table summarizes the expected input and output for a transform function for the supported use cases.

Transform Use Case	Input	Output
Document Ingestion	The input document, as received from the client application. In JavaScript, this is an immutable document node. Use `toObject` to create a mutable copy.	The document as it is to be stored in the database.
Document Retrieval	The document as stored in the database.	The document as it is to be returned to the client application.
Search	An XML or JSON document node representing either the search response or matching documents, depending on the operation.	The document to put in the response body. The data is returned to the client as-is.
Alert Match	An XML document with a `<rapi:rules/>` root element.	The document to put in the response body. The data is returned to the client as-is.

A transform applied to a search operation should expect to receive either a search results summary, such as a `<search:response/>`, or a document matched by the query. A search request can return a search result summary, a set of matching documents, or both, depending on the request parameters, Accept headers, and query options. For details, see “What to Expect as Input Content” on page 239.

8.1.6 Reporting Errors

This section discusses proper error reporting in your transform or service resource extension implementation. Use the approach described here for both XQuery and XSLT implementations. For more information about error reporting conventions, see “Error Reporting” on page 37.

To report an error from your implementation, use the `fn:error` XQuery function with the `RESTAPI-SRVEXERR` error code, and provide the HTTP response code and additional error information in the 3rd parameter to `fn:error`.

The 3rd parameter to `fn:error` should be a sequence of the form `("status-code", "status-message", "response-payload")`. That is, when using `fn:error` to raise `RESTAPI-SRVEXERR`, the `$data` parameter to `fn:error` is a sequence with the following members, all optional:

- HTTP status code. Default: 400.
- HTTP status message. Default: Bad Request.
- Response payload (optional). This should either be plain text or data in a format compatible with the default error format configured for the REST API instance.

Note: Best practice is to use `RESTAPI-SRVEXERR`. If you report any other error or raise any other exception, it is reported to the calling application as a 500 Server Internal Error.

For example, this resource extension function raises `RESTAPI-SRVEXERR` if the input content type is not as expected:

```
declare function example:transform(
    $context as map:map,
    $params  as map:map,
    $input   as document-node()
) as document-node()
{
    (: get 'input-types' to use in content negotiation :)
    let $input-types := map:get($context,"input-types")
    let $negotiate :=
        if ($input-types = "application/xml")
        then () (: process, insert/update :)
        else fn:error((),"RESTAPI-SRVEXERR",
          ("415","tsk tsk","whoops"))
    return document { "Done"}  (: may return a document node :)
};
```

If a PUT request is made to the transform with an unexpected content type, the `fn:error` call causes the request to fail with a status 415 and to include the additional error description in the response body:

```
HTTP/1.1 415 tsk tsk
Content-type: application/xml
Server: MarkLogic
Set-Cookie: SessionID=714070bdf4076536; path=/
Content-Length: 62
Connection: close

{
  "message": "js-example: response with invalid 400 status",
  "statusCode": 415,
  "body": {
    "errorResponse": {
```

```
            "statusCode": 415,
            "status": "tsk tsk",
            "messageCode": "RESTAPI-SRVEXERR",
            "message": "whoops"
          }
        }
      }
```

8.1.7 Context Map Keys

The context map parameter made available to transformations can contain the following keys:

XQuery Map Key	JavaScript Property Name	Description
`input-type`	`inputType`	The MIME type of the input document (the `content` parameter).
`uri`	`uri`	The URI of the input document, in a read or write transform.
`accept-types`	`acceptTypes`	On a read transform, the MIME types accepted by the requestor. Multiple MIME types are separated by whitespace. If there are multiple Accept types, the value is a JavaScript array or an XQuery sequence.
`output-type`	`outputType`	The expected output document MIME type. It is set for you by the REST API, but if your XQuery or XSLT read or write transform supports content negotiation, the transform function can set/modify this value.

8.1.8 Controlling Transaction Mode

This section discusses the default transaction mode under which your content transformation is evaluated, and how you can override the default behavior when using an XQuery transform. You cannot control transaction context for a JavaScript transform.

This discussion assumes you are familiar with the MarkLogic Server transaction mode concept; for details, see Transaction Mode in *Application Developer's Guide*.

The default transaction mode for a transformation function depends on the request on whose behalf it is applied:

- PUT or POST to `/documents`: update
- GET `/documents`: query
- GET or POST for a query service (`/search`, `/qbe`, `/values`): query

Choosing the appropriate transaction mode can improve the resource utilization of your implementation. For example, an update transaction acquires exclusive locks on documents, as described in Update Transactions: Readers/Writers Locks in *Application Developer's Guide*. If you know that your transformation is actually a read-only operation, you can override the default transaction mode so that the function is evaluated in "query" transaction mode and does not acquire exclusive locks.

To override the default behavior, include a transaction mode annotation in the declaration of your function:

```
declare %rapi:transaction-mode("the-mode") function yourNS:transform(...)
```

The value for `%rapi:transaction-mode` can be either "update" or "query".

For example, to specify query transaction mode for your `transform` function, use the following declaration:

```
declare %rapi:transaction-mode("query") function example:transform(
  $context as map:map,
  $params as map:map,
  $content as document-node())
as document-node()
{...};
```

8.2 Installing Transformations

This section covers the following topics:

- Basic Installation
- Including Transform Metadata

8.2.1 Basic Installation

To install or update a transformation, use `PUT:/v1/config/transforms/{name}`. Send a PUT request with a URL of the following form, with the transform code in the request body:

```
http://host:port/version/config/transforms/yourTransformName
```

Where *yourTransformName* is the name used when administering and applying the transformation. You can optionally include informational metadata about your transform during installation; for details, see "Including Transform Metadata" on page 332.

Note: If you are installing an XQuery transform, *yourTransformName* must match the leaf name in the transform module namespace declaration. For example:

```
xquery version "1.0-ml";
module namespace example =
```

```
  "http://marklogic.com/rest-api/transform/yourTransformName";
...
```

Set the `Content-type` of the request to one of the following, depending on the language in which you implement your transformation:

- XQuery: `application/xquery`
- JavaScript: `application/vnd.marklogic-javascript`
- XSLT: `application/xslt+xml`

For example, the following command installs a JavaScript transform under the name "js-example":

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -i \
    --data-binary @"./trans-ex.sjs" \
    -H "Content-type: application/vnd.marklogic-javascript" \
    'http://localhost:8000/LATEST/config/transforms/js-example'
```

If your transform requires additional assets such as user-defined XQuery or JavaScript library modules, you must install them separately before installing your transform. For details, see "Using a Resource Service Extension" on page 362.

For a complete example, see one of the following topics:

- "JavaScript Example: Adding a JSON Property During Ingestion" on page 335
- "XSLT Example: Adding an Attribute During Ingestion" on page 340
- "XQuery Example: Adding an Attribute During Ingestion" on page 338

8.2.2 Including Transform Metadata

You can optionally include metadata for `provider`, `version`, `title`, `description` and caller-supplied parameters as request parameters during installation. Metadata can be retrieved using `GET:/v1/config/transforms`. See "Discovering Transformations" on page 333.

For example, the following transform installation includes metadata about the provider:

```
http://host:port/version/config/transforms/example?provider=MarkLogic%
20Corporation
```

The optional parameter metadata identifies the name, type, and cardinality of parameters the client can supply when apply the transform to a read, write or search. Include the parameter metadata as request parameters of the following form:

```
trans:paramName=paramTypeAndCardinality
```

Where *paramName* is the name of the parameter and *paramTypeAndCardinality* reflects the type and cardinality the parameter would have in an XQuery function declaration.

For example, “string?” means a string type parameter which may appear 0 or 1 time. The following URL illustrates a transform called “example” with an optional string parameter named “reviewer”:

```
http://host:port/version/config/transforms/example?trans:reviewer=string\?
```

8.3 Applying Transformations

All REST Client API services that support transformations use the following mechanism for specifying application of a transformation:

1. Specify the name of an installed transform as the value of the `transform` request parameter:

   ```
   transform=transformName
   ```

2. If the transform expects parameters, specify each parameter with a request parameter of the form:

   ```
   trans:paramName=value
   ```

For example, to apply a transform during document insertion, you can send a PUT request to the `/documents` service with a `transform` request parameter. Assuming the transform called “example” expects one parameter named “reviewer”, the PUT request URL might look like the following:

```
http://host:port/version/documents?uri=/doc/theDoc.xml&transform=example&trans:reviewer=me
```

You could apply the same transformation during document retrieval by sending a GET request to the `/documents` service with the same URL.

For a complete example, see one of the following topics:

- “JavaScript Example: Adding a JSON Property During Ingestion” on page 335
- “XSLT Example: Adding an Attribute During Ingestion” on page 340
- “XQuery Example: Adding an Attribute During Ingestion” on page 338

8.4 Discovering Transformations

Use `GET:/v1/config/transforms` to discover the name, interface, and other metadata about installed transformations. Send a GET request to the `/config/transforms` service of the form:

```
http://host:port/version/config/transforms
```

MarkLogic Server returns a summary of the installed transforms in XML or JSON. The default summary format is XML. Use the Accept header or `format` request parameter to select the output content type. For details, see “Controlling Input and Output Content Type” on page 31.

By default, this request rebuilds the transform metadata each time it is called to ensure the metadata is up to date. If you find this refresh makes discovery take too long, you can disable the refresh by setting the `refresh` request parameter to false. Disabling the refresh can result in this request returning inaccurate information, but it does not affect the “freshness” or availability of the implementation of any transforms.

The following example requests an XML report:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
    http://localhost:8000/LATEST/config/transforms
...
HTTP/1.1 200 Transform List Retrieved
Content-type: application/xml
Server: MarkLogic
Content-Length: 348
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<rapi:transforms xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:transform>
    <rapi:name>add-attr-xsl</rapi:name>
    <rapi:title/>
    <rapi:version/>
    <rapi:provider-name>MarkLogic Corporation</rapi:provider-name>
    <rapi:description/>
    <rapi:transform-parameters>
      <rapi:parameter>
        <rapi:parameter-name>value</rapi:parameter-name>
        <rapi:parameter-type>string?</rapi:parameter-type>
      </rapi:parameter>
      <rapi:parameter>
        <rapi:parameter-name>name</rapi:parameter-name>
        <rapi:parameter-type>string?</rapi:parameter-type>
      </rapi:parameter>
    </rapi:transform-parameters>
    <rapi:transform-source>
      /v1/transforms/add-attr
    </rapi:transform-source>
  </rapi:transform>
</rapi:transforms>
```

8.5 Retrieving the Implementation of a Transformation

To retrieve the XQuery code or XSLT stylesheet that implements a transformation, send a GET request to the `/config/transforms/{name}` service of the form:

```
http://host:port/version/config/transforms/transformName
```

Where *transformName* is the name under which the extension is installed.

Set the request Accept header to one of the following MIME types. The type must match the implementation language of the transform.

- XQuery: `application/xquery`
- JavaScript: `application/vnd.marklogic-javascript`
- XSLT: `application/xslt+xml`

MarkLogic Server returns the XQuery library module or XSLT stylesheet that implements the transform in the response body. For example:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xquery" \
    http://localhost:8000/LATEST/config/transforms/example
...
xquery version "1.0-ml";
module namespace
example="http://marklogic.com/rest-api/transform/example";
declare function example:transform( ...
```

8.6 JavaScript Example: Adding a JSON Property During Ingestion

This example demonstrates a transformation that adds a property to a JSON document. The first example in this section adds a property with a fixed name and a value corresponding to the current dateTime. The second example adds a property corresponding to a name and value passed in as transform parameters.

When used as a read transform, the following transform adds a property named `readTimestamp` to the content returned to the client. When used as a write transform, it adds a property named `writeTimestamp` to the document stored in the database. In both cases, the value of the property is the dateTime at the time of the operation. If the input document is not JSON, the content is unchanged on both read and write.

Follow these steps to exercise the transform:

1. Copy the following code into a file called “trans-ex.sjs”.

```
function insertTimestamp(context, params, content)
{
  if (context.inputType.search('json') >= 0) {
    var result = content.toObject();
    if (context.acceptTypes) {                  /* read */
      result.readTimestamp = fn.currentDateTime();
    } else {                                    /* write */
      result.writeTimestamp = fn.currentDateTime();
```

```
    }
    return result;
  } else {
    /* Pass thru for non-JSON documents */
    return content;
  }
};

exports.transform = insertTimestamp;
```

2. Send a `PUT:/v1/config/transforms/{name}` request to install the transform with the name "js-example". For example:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -i \
    --data-binary @"./trans-ex.sjs" \
    -H "Content-type: application/vnd.marklogic-javascript" \
    'http://localhost:8000/LATEST/config/transforms/js-example'
```

3. Optionally, verify installation of the transformation using `GET:/v1/config/transforms`. For example:

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/json" \
    http://localhost:8000/LATEST/config/transforms
{ "transforms": {
    "transform": [
      {
        "name": "js-example",
        "source-format": "javascript",
        "transform-parameters": "",
        "transform-source": "/v1/config/transforms/example"
      }
] }
```

4. Send a `PUT:/v1/documents` request to insert a document into the database, using the transform to add the "writeTimestamp" property. For example:

```
$ curl --anyauth --user user:password -X PUT -d'{"name":"value"}' \
    -H "Content-type: application/json" \
    'http://localhost:8000/LATEST/documents?uri=/transforms/example.js
on&transform=js-example'
```

5. Send a `GET:/v1/documents` request to retrieve the inserted document. Note the added `writeTimestamp` attribute. For example:

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/json" \
    http://localhost:8000/LATEST/documents?uri=/transforms/example.jso
n
{ "key":"value",
```

```
  "writeTimestamp":"2014-11-25T16:59:00.459241-08:00"
}
```

6. Send a second `GET:/v1/documents` request to retrieve the inserted documents and apply the transform as a read transform. Note the added "readTimestamp" property. For example:

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/json" \
    'http://localhost:8000/LATEST/documents?uri=/transforms/example.js
on&transform=js-example'
{ "key":"value",
  "writeTimestamp":"2014-11-25T16:59:00.459241-08:00",
  "readTimestamp":"2014-11-25T17:38:28.417881-08:00"
}
```

The `writeTimestamp` is unchanged since it is part of the document. The `readTimestamp` is an artifact of the read, rather than part of the document. It will change each time you apply the transform during the read.

The following function is an alternative transform that demonstrates passing in transform parameters. For each parameter passed in via `trans:name=value`, the transform inserts a JSON property with the same name and value. The transform is applicable to both read and write operations. As before, non-JSON documents are unchanged.

```
function insertProperty(context, params, content)
{
  if (context.inputType.search('json') >= 0 && params) {
    var result = content.toObject();
    Object.keys(params).forEach(function(key) {
      result[key] = params[key];
    });
    return result;
  } else {
    // Pass thru unchanged
    return content;
  }
};

exports.transform = insertProperty;
```

The following example commands demonstrate using this transform on a read operation, assuming the transform is installed with the name "ex-params":

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/json" -i \
    'http://localhost:8000/LATEST/documents?uri=/transforms/example.js
on&transform=ex-params&trans:prop1=42&trans:prop2=dog'
...
{ "key":"value",
  "writeTimestamp":"2014-11-25T16:59:00.459241-08:00",
  "prop1":"42",
```

```
  "prop2":"dog"
}
```

You can specify the same parameter more than once. For example, if the request parameter "prop1" is supplied more than once, the value of "prop1" is an array.

```
http://localhost:8000/LATEST/documents?...&trans:prop1=42&trans:prop1=
43
==>
{ ...,
  "prop1":["42", "43"]
}
```

8.7 XQuery Example: Adding an Attribute During Ingestion

This example demonstrates using a transformation to add an attribute to the root node of an XML document during ingestion. The name and value of the added attribute are passed in as parameters to the transform function. You can use the same transform for document retrieval by applying it to a GET request instead of a PUT.

The transformation XQuery module shown below reads the attribute name and value from parameters, and then makes a copy of the input document, adding the requested attribute to the root node. The transformation cannot perform direct transformations on the input XML, such as calling `xdmp:node-replace`, because the document has not yet been inserted into the database when the transform function runs.

```
xquery version "1.0-ml";
module namespace example =
  "http://marklogic.com/rest-api/transform/add-attr";

declare function example:transform(
  $context as map:map,
  $params as map:map,
  $content as document-node()
) as document-node()
{
  if (fn:empty($content/*)) then $content
  else
    let $value := (map:get($params,"value"),"UNDEFINED")[1]
    let $name := (map:get($params, "name"), "transformed")[1]
    let $root  := $content/*
    return document {
      $root/preceding-sibling::node(),
      element {fn:name($root)} {
        attribute { fn:QName("", $name) } {$value},
        $root/@*,
        $root/node()
      },
      $root/following-sibling::node()
    }
};
```

To exercise this example:

1. Cut and paste the code above into a file called “add-attr.xqy”.

2. Make a PUT request to the `/config/transforms/{name}` service to install the transform as “add-attr”, accepting 2 string parameters called “name” and “value”. For example:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -d@"./add-attr.xqy" \
    -H "Content-type: application/xquery" \
    'http://localhost:8000/LATEST/config/transforms/add-attr?trans:nam
e=string\?&trans:value=string\?'
```

3. Optionally, verify installation of the transformation by making a GET request to the `/config/transforms` service. For example:

```
$ curl --anyauth --user user:password -X GET -i \
    -H "Accept: application/xml" \
    http://localhost:8000/LATEST/config/transforms
...
HTTP/1.1 200 Transform List Retrieved
Content-type: application/xml
Server: MarkLogic
Content-Length: 348
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<rapi:transforms xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:transform>
    <rapi:name>add-attr</rapi:name>
    <rapi:source-format>xquery</rapi:source-format>
    <rapi:title/>
    <rapi:version/>
    <rapi:provider-name/>
    <rapi:description/>
    <rapi:transform-parameters>
      <rapi:parameter>
        <rapi:parameter-name>value</rapi:parameter-name>
        <rapi:parameter-type>string?</rapi:parameter-type>
      </rapi:parameter>
      <rapi:parameter>
        <rapi:parameter-name>name</rapi:parameter-name>
        <rapi:parameter-type>string?</rapi:parameter-type>
      </rapi:parameter>
    </rapi:transform-parameters>
    <rapi:transform-source>
      /v1/transforms/add-attr
    </rapi:transform-source>
  </rapi:transform>
</rapi:transforms>
```

4. Make a PUT request to the `/documents` service to insert a document into the database, using the "add-attr" transform to add the attribute "example" with a value of "new". For example:

```
$ curl --anyauth --user user:password -X PUT -d"<data/>" \
    'http://localhost:8000/LATEST/documents?uri=/doc/transformed.xml&t
ransform=add-attr&trans:name=example&trans:value=new'
```

5. Make a GET request to the `/documents` service to retrieve the inserted document. Note the added "example" attribute. For example:

```
$ curl --anyauth --user user:password -X GET \
    http://localhost:8000/LATEST/documents?uri=/doc/transformed.xml
<?xml version="1.0" encoding="UTF-8"?>
<data example="new"/>
```

8.8 XSLT Example: Adding an Attribute During Ingestion

This example demonstrates using a transformation to add an attribute to the root node of an XML document during ingestion. The name and value of the added attribute are passed in as parameters to the stylesheet. You can use the same transform for document retrieval by applying it to a GET request instead of a PUT.

The transformation stylesheet shown below reads the attribute name and value from parameters, and then makes a copy of the input document, adding the requested attribute to the root node.

```
<xsl:stylesheet version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:example="http://marklogic.com/rest-api/example/transform"
    xmlns:map="http://marklogic.com/xdmp/map">
<xsl:param name="context" as="map:map"/>
<xsl:param name="params"  as="map:map"/>
<xsl:template match="/*">
    <xsl:copy>
        <xsl:attribute
name='{(map:get($params,"name"),"transformed")[1]}'
            select='(map:get($params,"value"),"UNDEFINED")[1]'/>
        <xsl:copy-of select="@*"/>
        <xsl:copy-of select="node()"/>
    </xsl:copy>
</xsl:template>
</xsl:stylesheet>
```

To exercise this example:

1. Create a REST Client API instance on port 8003. For instructions, see "Creating an Instance" on page 40.

2. Cut and paste the stylesheet above into a file called "add-attr.xsl".

3. Make a PUT request to the `/config/transforms/{name}` service to install the transform as "add-attr-xsl", accepting 2 string parameters called "name" and "value". For example:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -d@"./add-attr.xsl" \
    -H "Content-type: application/xslt+xml" \
    'http://localhost:8000/LATEST/config/transforms/add-attr-xsl?trans
:name=string\?&trans:value=string\?'
```

4. Optionally, verify installation of the transformation by sending a `GET:/v1/config/transforms` request. For example:

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
    http://localhost:8000/LATEST/config/transforms
...
HTTP/1.1 200 Transform List Retrieved
Content-type: application/xml
Server: MarkLogic
Content-Length: 348
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<rapi:transforms xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:transform>
    <rapi:name>add-attr-xsl</rapi:name>
    <rapi:source-format>xslt</rapi:source-format>
    <rapi:title/>
    <rapi:version/>
    <rapi:provider-name/>
    <rapi:description/>
    <rapi:transform-parameters>
      <rapi:parameter>
        <rapi:parameter-name>value</rapi:parameter-name>
        <rapi:parameter-type>string?</rapi:parameter-type>
      </rapi:parameter>
      <rapi:parameter>
        <rapi:parameter-name>name</rapi:parameter-name>
        <rapi:parameter-type>string?</rapi:parameter-type>
      </rapi:parameter>
    </rapi:transform-parameters>
    <rapi:transform-source>
      /v1/transforms/add-attr
    </rapi:transform-source>
  </rapi:transform>
</rapi:transforms>
```

5. Send a `PUT:/v1/documents` request to insert a document into the database, using the "add-attr-xsl" transform to add the attribute "example" with a value of "new". For example:

```
$ curl --anyauth --user user:password -X PUT -d"<data/>" \
    'http://localhost:8000/LATEST/documents?uri=/doc/transformed.xml&t
ransform=add-attr-xsl&trans:name=example&trans:value=new'
```

6. Send a `GET:/v1/documents` request to retrieve the inserted document. Note the added "example" attribute. For example:

```
$ curl --anyauth --user user:password -X GET \
    http://localhost:8000/LATEST/documents?uri=/doc/transformed.xml
<?xml version="1.0" encoding="UTF-8"?>
<data example="new"/>
```

8.9 XQuery Example: Modifying Document Type During Ingestion

This example is a write transform that converts JSON input into XML, inserting an XML document into the database. It also demonstrates modifying the context map to change the document URI.

When you insert a document with a PUT or POST request to the `/documents` service and specify a transform function name, the MIME type in the Content-type request header takes precedence over the URI extension in determining the format of the data in the request body.

The following transform takes advantage of this behavior to accept JSON input and convert it to XML. The transform also modifies the URI to match, if necessary: If the URI has a ".json" URI suffix, the URI is modified to have a ".xml" suffix by modifying the value of the "uri" key in the `$context` map.

```
xquery version "1.0-ml";
module namespace xqjson2xml =
  "http://marklogic.com/rest-api/transform/xqjson2xml";

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


declare function xqjson2xml:transform(
  $context as map:map,
  $params as map:map,
  $content as document-node()
) as document-node()
{
  if (fn:contains(map:get($context, "input-type"), "json"))
  then (
    map:put($context, "output-type", "application/xml"),
    let $uri := map:get($context, "uri")
    return
      if (fn:ends-with($uri, ".json")) then
        map:put($context, "uri",
                fn:concat(fn:substring-before($uri, ".json"), ".xml"))
      else (),
    document {json:transform-from-json($content)}
```

```
  )
  else $content
};
```

To exercise this example:

1. Cut and paste the code above into a file called “json2xml.xqy”.

2. Make a PUT request to the `/config/transforms/{name}` service to install the transform as “xqjson2xm”. For example:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -d@"./json2xml.xqy" \
    -H "Content-type: application/xquery" \
    'http://localhost:8000/LATEST/config/transforms/xqjson2xml'
```

3. Optionally, verify installation of the transformation by making a GET request to the `/config/transforms` service. For example:

```
$ curl --anyauth --user user:password -X GET -i \
    -H "Accept: application/xml" \
    http://localhost:8000/LATEST/config/transforms
```

4. Make a PUT request to the `/documents` service to insert a document into the database with a .xml extension, using the “xqjson2xml” transform. Note the request payload is JSON, but an XML document is inserted into the database.

```
$ curl --anyauth --user user:password -i -X PUT \
    -d'{"a": "b"}' -H "Content-type: application/json" \
    'http://localhost:8000/LATEST/documents?uri=/doc/j2x.xml&transform
=xqjson2xml'
```

5. Make a GET request to the `/documents` service to retrieve the inserted XML document. For example:

```
$ curl --anyauth --user user:password -X GET \
    'http://localhost:8000/LATEST/documents?uri=/doc/j2x.xml'
...
HTTP/1.1 200 OK
vnd.marklogic.document-format: xml
Content-type: application/xml; charset=utf-8
Server: MarkLogic
Content-Length: 134
Connection: Keep-Alive
Keep-Alive: timeout=5

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

6. Make another PUT request to the `/documents` service to insert a document into the database, using the "xqjson2xml" transform, but this time with a URI with a ".json" suffix. The transform modifies both the document type and the URI extension.

```
$ curl --anyauth --user user:password -i -X PUT \
    -d'{"c": "d"}' -H "Content-type: application/json" \
    'http://localhost:8000/LATEST/documents?uri=/doc/j2x_2.json&transf
orm=xqjson2xml'
```

7. Make a GET request to the `/documents` service to retrieve the inserted XML document, using the modified URI. For example:

```
$ curl --anyauth --user user:password -X GET \
    'http://localhost:8000/LATEST/documents?uri=/doc/j2x_2.xml'
...
HTTP/1.1 200 OK
vnd.marklogic.document-format: xml
Content-type: application/xml; charset=utf-8
Server: MarkLogic
Content-Length: 134
Connection: Keep-Alive
Keep-Alive: timeout=5

<?xml version="1.0" encoding="UTF-8"?>
<json type="object" xmlns="http://marklogic.com/xdmp/json/basic"><c
type="string">d</c></json>
```

Note that you can write a similar transformation in JavaScript, but you could not modify the document URI because any changes to the context input object are discarded when your JavaScript transform returns.

9.0 Extending the REST API

You can extend the REST Client API in a variety of ways, including resource service extensions and evaluation of ad-hoc queries and server-side modules. This chapter covers the following topics:

- Available REST API Extension Points
- Understanding Resource Service Extensions
- Creating a JavaScript Resource Service Extension
- Creating an XQuery Resource Service Extension
- Installing a Resource Service Extension
- Using a Resource Service Extension
- Example: JavaScript Resource Service Extension
- Example: XQuery Resource Service Extension
- Controlling Access to a Resource Service Extension
- Discovering Resource Service Extensions
- Retrieving the Implementation of a Resource Service Extension
- Managing Dependent Libraries and Other Assets
- Evaluating an Ad-Hoc Query
- Evaluating a Module Installed on MarkLogic Server

9.1 Available REST API Extension Points

The REST Client API offers several ways to extend and customize the available services with user-defined code that is either pre-installed on MarkLogic Server or supplied at request time. The following extension points are available:

- Content transformations: A user-defined transform function can be applied when documents are written to the database or read from the database; for details, see “Working With Content Transformations” on page 323. You can also define custom replacement content generators for the patch feature; for details, see “Constructing Replacement Data on the Server” on page 112.
- Search result customization: Customization opportunities include constraint parsers for string queries, search result snippet generation, and search result customization. For details, see “Customizing Search Results” on page 227 and the *Search Developer’s Guide*.
- Resource service extensions: Define your own REST endpoints, accessible through GET, PUT, POST and/or DELETE requests. Resource service extensions are covered in detail in this chapter.

- Ad-hoc query execution: Send an arbitrary block of XQuery or JavaScript code to MarkLogic Server for evaluation. For details, see “Evaluating an Ad-Hoc Query” on page 380.
- Server-side module evaluation: Evaluate user-defined XQuery or JavaScript modules pre-installed on MarkLogic Server. For details, see “Evaluating a Module Installed on MarkLogic Server” on page 388.

9.2 Understanding Resource Service Extensions

Use the resource service extension capability to make new REST service endpoints available through a REST Client API instance. To make a resource service extension available, follow these steps, which are discussed in detail in this chapter.

1. Create a JavaScript or XQuery implementation of your extension. For details, see “Creating a JavaScript Resource Service Extension” on page 346 or “Creating an XQuery Resource Service Extension” on page 353.

2. Install the extension in a REST Client API instance to make it available to applications. For details, see “Installing a Resource Service Extension” on page 360.

3. Access the extension service through a `/resources/{extensionName}` service endpoint. For details see “Using a Resource Service Extension” on page 362.

The `/resources` service also supports dynamic discovery of installed extensions. When you install an extension, you can specify extension metadata, including method parameter name and type information to make it easier to use dynamically discovered extensions.

If your extension depends on other modules or assets, you can install them in the modules database using the /ext service. For details, see “Managing Dependent Libraries and Other Assets” on page 376.

9.3 Creating a JavaScript Resource Service Extension

This section covers the following topics related to authoring a Resource Service Extension using server-side JavaScript:

- Guidelines for Writing JavaScript Resource Service Extensions
- The JavaScript Resource Extension Interface
- Reporting Errors
- Setting the Response Status Code
- Setting Additional Response Headers
- Context Object Properties
- Controlling Transaction Mode

9.3.1 Guidelines for Writing JavaScript Resource Service Extensions

Keep the following guidelines in mind when authoring a resource extension using JavaScript:

- You must implement your extension using MarkLogic Server-Side JavaScript. For details, see the *JavaScript Reference Guide*.
- Methods such as `put` and `post` that accept an input document receive an immutable in-memory representation. MarkLogic Server builtin functions that operate on in-database documents, such as `xdmp.nodeReplace`, cannot be applied directly to the input document. To modify the structure of the input document, you must create a mutable instance of the document by calling `toObject` on it.
- When the client sends a single document, the `input` parameter passed to your implementation is a document node. When the client sends multiple documents, the input parameter is a `Sequence` over document nodes. You should be prepared to handle either.
- Methods that can return multiple documents must return a `Sequence` over document nodes or JavaScript objects. Use `Sequence.from` to create a `Sequence` from an array.
- Your code executes in the transaction context of the HTTP request on whose behalf it is called. You should not commit or rollback the current transaction. You can use `xdmp.invokeFunction`, `xdmp.eval`, and `xdmp.invoke` to perform operations in a different transaction.
- If your extension depends on other user-defined modules, those modules must be installed in the modules database of the REST Client API instance. For details, see “Managing Dependent Libraries and Other Assets” on page 376.
- Report errors back to the client application using `fn.error`, not through an Error object or other mechanism. For details, see “Reporting Errors” on page 349.

Many of these points are illustrated by “Example: JavaScript Resource Service Extension” on page 362.

9.3.2 The JavaScript Resource Extension Interface

To create a resource service extension using server-side JavaScript, implement a module based on the template below. The module exports and function signatures must be exactly as they appear in the template, with the following exceptions:

- You only need to implement and export functions for the HTTP verbs supported by the extension.

Base your extension implementation on the following template.

```
function get(context, params) {
  // return zero or more document nodes
};

function post(context, params, input) {
  // return zero or more document nodes
```

```
};

function put(context, params, input) {
  // return at most one document node
};

function deleteFunction(context, params) {
  // return at most one document node
};

exports.GET = get;
exports.POST = post;
exports.PUT = put;
exports.DELETE = deleteFunction;
```

The following template summarizes the meaning of the function parameters passed to your implementation functions:

Parameter	Description
`context`	An object containing service request context information such as input document types and URIs, and output types accepted by the caller. For details, see “Context Object Properties” on page 351.
`params`	An object containing extension-specific parameter values supplied by the client, if any. The object contains a property for each resource parameter included in the request to the extension service. Properties specified more than once by the caller have array value. For example: If the request is `PUT /v1/resource/my-ext?rs:p=1&rs:p=2`, then the value of `params.p` is `["1", "2"]`. You can optionally define parameter name and type metadata when installing the extension. For details, see “Installing a Resource Service Extension” on page 360.
`input`	The data from the request body. For a request with only part, input is a single document node. For a multipart request, input is a `Sequence` over document nodes.

The `input` parameter can be a single document node or a `Sequence`, depending on whether the request body contains a single part or multiple parts. You can normalize the input by converting the single document to a `Sequence`, if desired. For example:

```
function normalizeInput(item)
{
  return (item instanceof Sequence)
         ? item                      // many
```

```
            : Sequence.from([item]);    // one
};
```

The `input` document(s) are immutable. Use `toObject` to create a mutable copy. For example, assuming input a single JavaScript object created from passing JSON data in the request body, the following code enables you to modify the input object:

```
var modifiableDoc = input.toObject();
modifiableDoc.newProperty = someValue;
```

Your methods can accept input arguments in addition to the input documents in the request body. Arguments are passed to your methods through the `params` object. For an example, see the `get` implementation in “Example: JavaScript Resource Service Extension” on page 362.

Your extension can return JavaScript objects and XML, JSON, text, or binary document nodes. A JavaScript object is serialized as JSON before returning it to the client. Set `context.outputTypes` in the `$context` map to the MIME type appropriate for the returned content. Except for the JavaScript to JSON conversion, the content is returned as-is, with the MIME type you specify.

To return multiple documents, create a `Sequence` over your JavaScript objects or document nodes. Use `xdmp.arrayValues` or `Sequence.from` to create a `Sequence` from an array. For example, the following code snippet results in a multi-part response containing 2 parts, with each part containing a JSON document.

```
context.outputTypes = [ 'application/json', 'application/json' ];
return Sequence.from([{one: 1}, {two:2}];
```

If you want to report errors back to the client application, you must use the `fn.error` function, with a very specific calling convention. For details, see “Reporting Errors” on page 349.

9.3.3 Reporting Errors

This section describes how to report errors back to the client application. Extensions and transforms use the same mechanism to report errors to the calling application. If you attempt to return an error in any other fashion, such as through an `Error` object, the client just receives a 500 Internal Error response.

To report an error from your implementation, call the `fn.error` function with the `RESTAPI-SRVEXERR` error code, and provide the HTTP response code and additional error information in the 3rd parameter to `fn.error`. Calling `fn.error` interrupts execution of your code. Control does not return.

> **Note:** Best practice is to use `RESTAPI-SRVEXERR`. If you report any other error or raise any other exception, it is reported to the calling application as a 500 Server Internal Error.

The 3rd parameter to `fn:error` should be a `Sequence` containing a sequence of the form `(status-code, status-message, response-payload)`. You can use `xdmp.arrayValues` or `Sequence.from` to construct a sequence from a JavaScript array.

For example, if you wrap the `fn.error` call in a function, then you can call it as shown below:

```
function returnErrToClient(statusCode, statusMsg, body)
{
  fn.error(null, 'RESTAPI-SRVEXERR',
           Sequence.from([statusCode, statusMsg, body]));
  // unreachable - control does not return from fn.error.
};

// ... use the func ...
returnErrToClient(400, 'Bad Request', 'additional info');
// unreachable
```

If you include an error response payload, it should either be plain text or compatible with the error format configured for the REST instance. You cannot override the default error payload format.

9.3.4 Setting the Response Status Code

By default, your resource service extension does not need to specify the HTTP response status code. The default behavior is as follows:

- If your implementation function returns content, MarkLogic Server responds with a status 200.
- If your function raises no errors but does not return content, MarkLogic Server responds with status 204.
- If your function raises an error, the response code depends on the error; for details, see “Reporting Errors” on page 349.

To return a status code different from the default, set the numeric code and message in the `context` object. Use the `outputStatus` property with `[code, message]` ts value.

> **Note:** Do not use `output-status` to report an error to the client. Instead, use `fn.error`. For details, see “Reporting Errors” on page 349.

The following example returns a `201 Created` status to the client instead of `200 OK`:

```
function put(context, params, input)
{
    ...
    context.outputStatus = [201, 'Created'];
    ...
};
```

9.3.5 Setting Additional Response Headers

The REST API sets headers such as Content-type and Content-Length for you when constructing the response from a resource extension request. You can specify additional headers by setting `context.outputHeaders`.

> **Note:** Do not use `context.outputHeaders` to set the response Content-type header. Use `context.outputTypes` to specify the response MIME type(s) instead.

If you use `context.outputHeaders`, the value should be an object of the following form:

```
{header-name: header-value, header-name: header-value, ...}
```

The following example adds headers named “X-MyHeader1” and “X-MyHeader2” to the response by setting `context.outputHeaders`.

```
function put(context, params, input)
{
  ...
  context.outputHeaders =
    {"X-MyHeader1", "header1 value", "X-MyHeader2", "header2 value"};
  ...
};
```

The resulting response headers are similar to the following:

```
HTTP/1.1 200 OK
X-MyHeader1: header1 value
X-MyHeader2: header2 value
Server: MarkLogic
Content-Type: text/plain; charset=UTF-8
Content-Length: 4
Connection: Keep-Alive
Keep-Alive: timeout=5
```

9.3.6 Context Object Properties

The `context` parameter to resource service extension functions is an object that can contain the following properties. Unless otherwise noted, all the property values are arrays. Not all properties are always present. For example, the `inputTypes` property is not meaningful for a GET request.

Key	Description
`inputTypes`	The MIME type of the input document(s). There is an array element for each item in the `input` parameter passed to your method implementation.
`acceptTypes`	The MIME types accepted by the requestor.

Key	Description
`outputHeaders`	Extra headers to include in the response to the client. Optional. Specify headers as a JavaScript object of the form: `{header1: value1, header2: value2, ...}`. For details, see "Setting Additional Response Headers" on page 351.
`outputStatus`	The HTTP response status code and message to return to the client. Optional. For example, to return a 201 Created status, set `outputStatus` to `[201, "Created"]`. For details, see "Setting the Response Status Code" on page 350.
`outputTypes`	The output document MIME type(s). The service must set `outputTypes` for all output documents.

You can create a resource extension to probe `context`. For example, the following "extension" probes the `context` and `params` parameters of PUT extension requests:

```
function put(context, params, input) {
  context.outputTypes = ["application/json"];
  return { context: context, params: params };
};
exports.PUT = put;
```

If the above example is installed as an extension named "dumper", then you can probe the context as follows. There are two items in `context.inputTypes` because the multipart body contained one JSON part and one XML part.

```
curl --anyauth --user user:password -X PUT -i \
    -H "Content-type: multipart/mixed; boundary=BOUNDARY"
    -H "Accept: application/json" \
    --data-binary @./my-multipart-body \
    http://localhost:8000/LATEST/resources/dumper?rs:myparam=foo

HTTP/1.1 200 OK
Content-type: application/json; charset=UTF-8
Server: MarkLogic
Content-Length: 176
Connection: Keep-Alive
Keep-Alive: timeout=5

{"context": {
    "acceptTypes": [],
    "inputBoundary": "BOUNDARY",
    "inputTypes": [
      "application/json",
      "application/xml"
    ],
    "outputTypes": [ "application/json" ]
  },
```

```
    "params": { "myparam": "foo" }
}
```

For a complete example of a resource service extension, see “Example: JavaScript Resource Service Extension” on page 362.

9.3.7 Controlling Transaction Mode

In a JavaScript extension, you cannot control the transaction mode in which your `get`, `put`, `post`, or `delete` function runs. Transaction mode is controlled by the REST API App Server.

- `GET` - query for a single statement transaction, update for a multi-statement transaction; that is, if your request includes a transaction id, the function runs in update mode.
- `PUT` - update
- `POST` - query for a single statement transaction, update for a multi-statement transaction; that is, if your request includes a transaction id, the function runs in update mode.
- `DELETE` - update

If you need to execute code in a transaction context different from the default, use one of the following options:

- Execute the transaction mode sensitive code in a different transaction, using `xdmp.invokeFunction`, `xdmp.eval`, or `xdmp.invoke`. For details, see the *JavaScript Reference Guide*.
- Implement your extension in XQuery. For details, see “Creating an XQuery Resource Service Extension” on page 353.

To learn more about transaction mode and the MarkLogic transaction model, see Understanding Transactions in MarkLogic Server in the *Application Developer’s Guide*.

9.4 Creating an XQuery Resource Service Extension

This section covers the following topics related to authoring a Resource Service Extension using XQuery:

- Guidelines for Writing XQuery Resource Service Extensions
- The Resource Extension Interface
- Reporting Errors
- Setting the Response Status Code
- Setting Additional Response Headers
- Context Map Keys
- Controlling Transaction Mode

9.4.1 Guidelines for Writing XQuery Resource Service Extensions

Keep the following guidelines in mind when authoring a resource extension:

- Methods such as `put` and `post` that accept an input document receive an in-memory representation. MarkLogic Server builtin functions that operate on in-database documents, such as `xdmp:node-replace`, cannot be applied directly to the input document. To modify the structure of the input document, perform the modifications during construction of a copy of the input content.
- Your code executes in the transaction context of the HTTP request on whose behalf it is called. You should not commit or rollback the current transaction. You can use `xdmp:eval` and `xdmp:invoke` to perform operations in a different transaction.
- If your extension depends on other user-defined modules, those modules must be installed in the modules database of the REST Client API instance. For details, see "Managing Dependent Libraries and Other Assets" on page 376.
- Report errors by throwing one of the exceptions provided by the REST Client API. For details, see "Reporting Errors" on page 356.

9.4.2 The Resource Extension Interface

To create a resource service extension, implement an XQuery library module based on the template below. The module namespace, function names, and function signatures must be exactly as they appear in the template, with the following exceptions:

- The *extensionName* in the module namespace is the user-defined name under which the extension is installed and by which applications access the service. Each extension should have a unique name within the REST Client API instance in which it is installed.
- *yourNSPrefix* is a namespace prefix of your choosing.
- You only need to implement functions for the HTTP verbs supported by the extension.

Base your extension implementation on the following template. You must use the function local names shown (`get`, `put`, etc.).

```
xquery version "1.0-ml";
module namespace yourNSPrefix =
    "http://marklogic.com/rest-api/resource/extensionName";

declare function yourNSPrefix:get(
    $context as map:map,
    $params  as map:map
) as document-node()*

declare function yourNSPrefix:put(
    $context as map:map,
    $params  as map:map,
    $input   as document-node()*
) as document-node()?
```

```
declare function yourNSPrefix:post(
    $context as map:map,
    $params  as map:map,
    $input   as document-node()*
) as document-node()*

declare function yourNSPrefix:delete(
    $context as map:map,
    $params  as map:map
) as document-node()?
```

The `map:map` type is a MarkLogic Server extended XQuery type representing a key-value map. For details, see Using the map Functions to Create Name-Value Maps in the *Application Developer's Guide*.

Your methods can accept input arguments. You can define the argument names and types when installing your extension; for details see "Installing a Resource Service Extension" on page 360. Arguments are passed to your methods through the `$params` map. See the `get` implementation in "Example: XQuery Resource Service Extension" on page 370.

The following template summarizes the meaning of the function parameters:

Parameter	Description
`$context`	Service request context information such as input document type and URI, and output types accepted by the caller. For details, see "Context Map Keys" on page 359.
`$params`	Extension-specific parameters. The map contains a key for each parameter on a request to the extension service. You can optionally define parameters when installing the extension. For details, see "Installing a Resource Service Extension" on page 360.
`$input`	The input document to which to apply the transformation.

Your extension can return XML, JSON, text, or binary documents. Set `output-type` in the `$context` map to the MIME type appropriate for the returned content. The content is returned as-is, with the MIME type you specify.

If your extension function returns data, it must be in the form document nodes. In addition to getting documents from the database or a builtin or library function, you can create documents using node constructors. The following expression demonstrate various ways of constructing XML and JSON document nodes:

```
(: JSON, from a serialized string representation:)
xdmp:unquote('{"from": "string"}')

(: JSON, from a node constructor :)
```

```
document { object-node {"from": "constructor"} }

(: XML from a constructor :)
document { element from { text {"constructor"} } }

(: XML from a document constructor and XML :)
document { <from>XML literal</from> },
```

For details, see XML and XQuery in the *XQuery and XSLT Reference Guide* and Working With JSON in XQuery in the *Application Developer's Guide*.

9.4.3 Reporting Errors

Extensions and transforms use the same mechanism to report errors to the calling application: Use `fn:error` to report a `RESTAPI-SRVEXERR` error, and provide additional information in the `$data` parameter of `fn:error`. You can control the response status code, status message, and provide an additional error reporting response payload. For example:

```
fn:error((),"RESTAPI-SRVEXERR",
  (415, "Unsupported Input Type",
   "Only application/xml is supported"))
```

The 3rd parameter to `fn:error` should be a sequence of the form `("status-code", "status-message", "response-payload")`. That is, when using `fn:error` to raise `RESTAPI-SRVEXERR`, the `$data` parameter to `fn:error` is a sequence with the following members, all optional:

- HTTP status code. Default: 400.
- HTTP status message. Default: Bad Request.
- Response payload. Plain text or data compatible with the error message format configured for the REST API instance.

Note: Best practice is to use `RESTAPI-SRVEXERR`. If you report any other error or raise any other exception, it is reported to the calling application as a 500 Server Internal Error.

For example, this resource extension function raises `RESTAPI-SRVEXERR` if the input content type is not as expected:

```
declare function example:put(
    $context as map:map,
    $params  as map:map,
    $input   as document-node()
) as document-node()
{
    (: get 'input-types' to use in content negotiation :)
    let $input-types := map:get($context,"input-types")
    let $negotiate :=
        if ($input-types = "application/xml")
```

```
        then () (: process, insert/update :)
        else fn:error((),"RESTAPI-SRVEXERR",
          ("415","Raven","nevermore"))
    return document { "Done"}  (: may return a document node :)
};
```

If a PUT request is made to the extension with an unexpected content type, the `fn:error` call causes the request to fail with a status 415 and to include the additional error description in the response body:

```
HTTP/1.1 415 Raven
Content-type: application/json
Server: MarkLogic
Set-Cookie: SessionID=714070bdf4076536; path=/
Content-Length: 62
Connection: close

{
  "message": "js-example: response with invalid 400 status",
  "statusCode": 415,
  "body": {
    "errorResponse": {
      "statusCode": 415,
      "status": "Raven",
      "messageCode": "RESTAPI-SRVEXERR",
      "message": "nevermore"
    }
  }
}
```

9.4.4 Setting the Response Status Code

By default, your resource service extension does not need to specify the HTTP response status code. The default behavior is as follows:

- If your implementation function returns content, MarkLogic Server responds with a status 200.
- If your function raises no errors but does not return content, MarkLogic Server responds with status 204.
- If your function raises an error, the response code depends on the error; for details, see “Reporting Errors” on page 356.

To return a different status code other than the default, set the numeric code and message in `$context`. Use the key `output-status` with a `(`*`code`*`, `*`message`*`)` pair as its value.

> **Note:** Do not use `output-status` to report an error to the client. Instead, use `fn:error`. For details, see “Reporting Errors” on page 356.

The following example returns a `201 Created` status to the client instead of `200 OK`:

```
declare function example:put(
    $context as map:map,
    $params  as map:map,
    $input   as document-node()*
) as document-node()?
{
    ...
    map:put($context, "output-status", (201, "Created"));
};
```

9.4.5 Setting Additional Response Headers

The REST API sets headers such as Content-type and Content-Length for you when constructing the response from a resource extension request. You can specify additional headers by setting `output-headers` in the `$context` map.

Note: Do not use `output-headers` to set Content-type. Use `output-type` to specify MIME type(s) instead.

The following example adds headers named “X-My-Header1” and “X-My-Header2” to the response by setting the output-headers to a sequence of the form (*header-name*, *header-value*, *header-name*, *header-value*, ...)

```
declare function example:put(
    $context as map:map,
    $params  as map:map,
    $input   as document-node()*
) as document-node()?
{
    ...
    map:put($context, "output-headers",
      ("X-MyHeader1", "header1 value", "X-MyHeader2", "header2 value"))
2"));
};
```

You can also set `output-headers` using a map with the header names as keys. The following code snippet has the same effect on the response headers as the previous example.

```
let $headers := map:map()
...
(map:put($headers, "X-MyHeader1", "header1 value"),
 map:put($headers, "X-MyHeader2", "header2 value"),
 map:put($context, "output-headers", $headers))
```

In both cases, the resulting response headers are similar to the following:

```
HTTP/1.1 200 OK
X-MyHeader1: header1 value
X-MyHeader2: header2 value
```

```
Server: MarkLogic
Content-Type: text/plain; charset=UTF-8
Content-Length: 4
Connection: Keep-Alive
Keep-Alive: timeout=5
```

9.4.6 Context Map Keys

The `$context` parameter to resource service extension functions can contain the following keys. Not all keys are always present. For example, there is no input-type on a GET request.

Key	Description
`input-types`	The MIME type of the input document(s). If there are multiple input documents, the value is a sequence of strings, one for the MIME type of each input document.
`accept-types`	For a GET, the MIME types accepted by the requestor. If there are multiple MIME types, the value is a sequence of strings.
`output-headers`	Response headers to include extra headers in the response to the client. Optional. Specify headers as a sequence of alternating header and value pairs: `(header1, value1, header2, value2, ...)`, or as a `map`. For details, see "Setting Additional Response Headers" on page 358.
`output-status`	The HTTP response status code and message to return to the client. Optional. If unset, 200 OK is returned, unless an error is raised. For example, to return a 201 Created status, set `output-status` to `(201, "Created")`. For details, see "Setting the Response Status Code" on page 357.
`output-type`	The output document MIME type. The service must set `output-type` for all output documents.

For a complete example of a resource service extension, see "Example: XQuery Resource Service Extension" on page 370.

9.4.7 Controlling Transaction Mode

This section discusses the default transaction mode under which the implementation of a resource service extension is evaluated, and how you can override the default behavior. This discussion assumes you are familiar with the MarkLogic Server transaction mode concept; for details, see Transaction Mode in *Application Developer's Guide*.

Choosing the appropriate transaction mode can improve the resource utilization of your implementation. For example, the function implementing your POST method runs in "update" transaction mode by default. This means it acquires exclusive locks on documents, as described in Update Transactions: Readers/Writers Locks in *Application Developer's Guide*. If you know that your POST implementation is actually a read-only operation, you can override the default transaction mode so that the function is evaluated in "query" transaction mode and does not acquire exclusive locks.

The XQuery functions in a resource service extension implementation are evaluated using the following default transaction modes:

- `get` - query for a single statement transaction, update for a multi-statement transaction; that is, if your request includes a transaction id, the function runs in update mode.
- `put` - update
- `post` - query for a single statement transaction, update for a multi-statement transaction; that is, if your request includes a transaction id, the function runs in update mode.
- `delete` - update

To override the default behavior, include a transaction mode annotation in the declaration of your function:

```
declare %rapi:transaction-mode("the-mode") function yourNS:method(...)
```

The value for `%rapi:transaction-mode` can be either "update" or "query".

For example, to specify query transaction mode for your `put` function, use the following declaration:

```
declare %rapi:transaction-mode("query") function example:put(
    $context as map:map,
    $params  as map:map,
    $input   as document-node()*
) as document-node()?
{...};
```

9.5 Installing a Resource Service Extension

You install a resource service extension module using the REST Client API. Any dependent user-defined modules must be installed separately before installing your extension; for details, see "Managing Dependent Libraries and Other Assets" on page 376.

To install or update an extension, send a PUT request to `/config/resources` with a URL of the following form, with the extension code in the request body:

```
http://host:port/version/config/resources/extensionName
```

Where *extensionName* is the name used when administering and accessing the extension.

Note: For an XQuery extension, *extensionName* must match the name in the extension module namespace declaration. For example:

```
xquery version "1.0-ml";
module namespace yourNSPrefix =
  "http://marklogic.com/rest-api/resource/extensionName";
...
```

Set the Content-type of the request body to `application/xquery` or `application/vnd.marklogic-javascript`.

MarkLogic Server parses the extension implementation to determine which HTTP methods it supports. Only functions in the implementation that conform to the expected interface become available for use. For interface details, see "The Resource Extension Interface" on page 354.

If the extension service expects parameters, you can optionally "declare" the parameters using request parameters when installing the extension. This information is metadata that can be returned by a GET request to `/config/resources`. It is not used to check parameters on requests to the extension.

To include parameter metadata, declare each supported method and parameter using request parameters of the following form:

```
method=methodName

methodName:paramName=paramTypeAndCardinality
```

Where *methodName* is one of the extension functions: `get`, `put`, `post`, or `delete`; *paramName* is the name of the parameter; and *paramTypeAndCardinality* reflects the type and cardinality the parameter would have in an XQuery function declaration.

For example, "`xs:string?`" means a string type parameter which may appear 0 or 1 time. The following URL illustrates an extension named "example" with an optional string parameter named "reviewer" to the `put` method and a required string parameter "genre" to the `get` method:

```
'http://host:port/version/config/resources/example?method=put&put:revi
ewer=string\?&method=get&get:genre=xs:string'
```

You can also specify extension metadata for provider, version, title, and description as request parameters during installation. For example:

```
http://host:port/version/config/resources/example?method=get&get:genre
=string&provider=MarkLogic%20Corporation
```

For a complete example, see one of the following topics:

- "Example: JavaScript Resource Service Extension" on page 362
- "Example: XQuery Resource Service Extension" on page 370.

Note: If your module depends upon other user-defined modules, you must install them manually in the modules database of the REST API instance or in the Modules directory of your MarkLogic Server installation. For details, see "Using a Resource Service Extension" on page 362.

9.6 Using a Resource Service Extension

To access extension services, send a request of the appropriate type (GET, PUT, POST, or DELETE) to a URL of the following form:

```
http://host:port/version/resources/extensionName
```

Where *extensionName* is the name under which the extension is installed.

If the service expects parameters, supply the parameter names and values as request parameters, prefixing each parameter name with `rs:`. For example, if the "example" service for GET expects a parameter named "genre", the request URL would be similar to the following:

```
http://localhost:8000/LATEST/resources/example?rs:genre=literature
```

The names, types, and cardinality of method parameters are resource specific. You can send a GET request to `/config/resources` to dynamically discover resources extension interfaces. For details, see "Discovering Resource Service Extensions" on page 374.

9.7 Example: JavaScript Resource Service Extension

This section demonstrates the implementation, installation and invocation of a resource service extension implemented in Server-Side JavaScript.

- JavaScript Extension Implementation
- Installing the Example Extension
- Using the Example Extension

9.7.1 JavaScript Extension Implementation

This example extension implements all the HTTP methods supported by the resource extension interface. It demonstrates the use of input arguments, setting response content type, response status codes, additional response headers, and reporting errors to the calling application. The implementation follows the template described in "Creating a JavaScript Resource Service Extension" on page 346.

The methods implemented by this extension do the following:

Method	Description
GET	Accepts one or more caller-defined parameters. Returns a multipart response, with a part for each user-defined format. Each part contains a JSON document of the following form: `{ "name": param-name, "value": param-value }`
PUT	This method mimics some of the functionality of a multi-document write using `POST:/v1/documents`. For each input JSON document, adds a property to document and inserts it into the database. For each input XML document, inserts the document into the database unchanged. For any other document type, no action is taken. The document URIs are derived from a "basename" parameter supplied by the caller. The response contains a JSON document that reports the URIs of the inserted documents.
POST	Logs a message.
DELETE	Logs a message.

Helper functions are used to abstract away operations such as reporting errors to the client and normalizing the type of the input document set to `Sequence`. The later is demonstrated as a convenience since the PUT method might receive either a single document node or a `Sequence`, depending on whether the request body has one or multiple parts.

The following code implements the contract described above. See the comments in the code for details. For usage, see "Using the Example Extension" on page 367.

```
// GET
//
// This function returns a document node corresponding to each
// user-defined parameter in order to demonstrate the following
// aspects of implementing REST extensions:
// - Returning multiple documents
// - Overriding the default response code
// - Setting additional response headers
//
function get(context, params) {
  var results = [];
  context.outputTypes = [];
  for (var pname in params) {
    if (params.hasOwnProperty(pname)) {
      results.push({name: pname, value: params[pname]});
      context.outputTypes.push('application/json');
    }
  }
```

```
  // Return a successful response status other than the default
  // using an array of the form [statusCode, statusMessage].
  // Do NOT use this to return an error response.
  context.outputStatus = [201, 'Yay'];

  // Set additional response headers using an object
  context.outputHeaders =
    {'X-My-Header1' : 42, 'X-My-Header2': 'h2val' };

  // Return a Sequence to return multiple documents
  return Sequence.from(results);
};

// PUT
//
// The client should pass in one or more documents, and for each
// document supplied, a value for the 'basename' request parameter.
// The function inserts the input documents into the database only
// if the input type is JSON or XML. Input JSON documents have a
// property added to them prior to insertion.
//
// Take note of the following aspects of this function:
// - The 'input' param might be a document node or a Sequence
//   over document nodes. You can normalize the values so your
//   code can always assume a Sequence.
// - The value of a caller-supplied parameter (basename, in this case)
//   might be a single value or an array.
// - context.inputTypes is always an array
// - How to return an error report to the client
//
function put(context, params, input) {
  // normalize the inputs so we don't care whether we have 1 or many
  var docs = normalizeInput(input);
  var basenames = params.basename instanceof Array
                ? params.basename: [ params.basename ];

  // Validate inputs.
  if (docs.count > basenames.length) {
    returnErrToClient(400, 'Bad Request',
       'Insufficient number of uri basenames. Expected ' +
       docs.count + ' got ' + basenames.length + '.');
    // unreachable - control does not return from fn.error
  }

  // Do something with the input documents
  var i = 0;
  var uris = [];
  for (var doc of docs) {
    uris.push( doSomething(
        doc, context.inputTypes[i], basenames[i++]
    ));
  }

  // Set the response body MIME type and return the response data.
```

```
  context.outputTypes = ['application/json'];
  return { written: uris };
};

function post(context, params, input) {
  xdmp.log('POST invoked');
  return null;
};

function deleteFunction(context, params) {
  xdmp.log('POST invoked');
  return null;
};

// PUT helper func that demonstrates working with input documents.
//
// It inserts a (nonsense) property into the incoming document if
// it is a JSON document and simply inserts the document unchanged
// if it is an XML document. Other doc types are skipped.
//
// Input documents are imutable, so you must call toObject()
// to create a mutable copy if you want to make a change.
//
// The property added to the JSON input is set to the current time
// just so that you can easily observe it changing on each invocation.
//
function doSomething(doc, docType, basename)
{
  var uri = '/extensions/' + basename;
  if (docType == 'application/json') {
    // create a mutable version of the doc so we can modify it
    var mutableDoc = doc.toObject();
    uri += '.json';

    // add a JSON property to the input content
    mutableDoc.written = fn.currentTime();
    xdmp.documentInsert(uri, mutableDoc);
    return uri;
  } else if (docType == 'application/xml') {
    // pass thru an XML doc unchanged
    uri += '.xml';
    xdmp.documentInsert(uri, doc);
    return uri;
  } else {
    return '(skipped)';
  }
};

// Helper function that demonstrates how to normalize inputs
// that may or may not be multi-valued, such as the 'input'
// param to your methods.
//
// In cases where you might receive either a single value
// or a Sequence, depending on the request context,
```

```
// you can normalize the data type by creating a Sequence
// from the single value.
function normalizeInput(item)
{
  return (item instanceof Sequence)
         ? item                        // many
         : Sequence.from([item]);      // one
};

// Helper function that demonstrates how to return an error response
// to the client.

// You MUST use fn.error in exactly this way to return an error to the
// client. Raising exceptions or calling fn.error in another manner
// returns a 500 (Internal Server Error) response to the client.
function returnErrToClient(statusCode, statusMsg, body)
{
  fn.error(null, 'RESTAPI-SRVEXERR',
           Sequence.from([statusCode, statusMsg, body]));
  // unreachable - control does not return from fn.error.
};


// Include an export for each method supported by your extension.
exports.GET = get;
exports.POST = post;
exports.PUT = put;
exports.DELETE = deleteFunction;
```

9.7.2 Installing the Example Extension

This section installs the example resource extension under the name `js-example`.

Follow these steps to install the example extension in the modules database associated with your REST API instance:

1. If you do not already have a REST API instance, create one. This example uses the pre-configured instance on port 8000, and assumes the host is localhost.

2. Copy the code from “JavaScript Extension Implementation” on page 362 into a file. You can use any file name. This example assumes `example.sjs`.

3. Install the example using a command similar to the following.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -i \
    -H "Content-type: application/vnd.marklogic-javascript" \
    --data-binary @./example.sjs \
    http://localhost:8000/LATEST/config/resources/js-example
```

For usage examples, see “Using the Example Extension” on page 367.

Optionally, you can include metadata during installation so that a GET request to `/config/resources` returns more detail. For example:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    -H "Content-type: application/vnd.marklogic-javascript" \
    -d@"./example.sjs" \
    "http://localhost:8000/LATEST/config/resources/js-example?version=
1.0&provider=marklogic&description=A simple resource
example&method=get&method=post&method=put&method=delete&get:arg1=[stri
ng]"
```

The table below breaks down the metadata request parameters so it is easier to see what metadata is included.

Request Parameter	Description
`version=1.0`	The extension version number.
`provider=marklogic`	The extension provider.
`description=A simple resource example`	A brief description of the extension.
`method=get`	The extensions supports GET requests.
`method=post`	The extension supports POST requests.
`method=put`	The extension supports PUT requests
`method=delete`	The extension supports DELETE requests
`get:arg1=[string]`	The GET method accepts an argument named “arg1” which can occur 1 or more times.

9.7.3 Using the Example Extension

This section explores the behavior of the example extension and demonstrates how to exercise some of the methods.

The GET method of the example extension returns a JSON document for each parameter value passed in. Each document contains the name and value of a parameter. The GET method also demonstrates overriding the default response status and adding response headers.

```
function get(context, params) {
  ...
  for (var pname in params) {
    if (params.hasOwnProperty(pname)) {
```

```
      results.push({name: pname, value: params[pname]});
      context.outputTypes.push('application/json');
    }
  }
  context.outputStatus = [201, 'Yay'];
  context.outputHeaders =
    {'X-My-Header1' : 42, 'X-My-Header2': 'h2val' };
  return Sequence.from(results);
};
```

The following command exercises the GET method. The response contains 2 parts, one for each unique input parameter name. Notice that the value of the parameter "a" is an array, reflecting its inclusion twice in the request URL.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
  -H "Accept: multipart/mixed; boundary=BOUNDARY" \
  'http://localhost:8000/LATEST/resources/js-example?rs:a=1&rs:b=2&rs:
a=3'

HTTP/1.1 201 Yay
Content-type: multipart/mixed; boundary=BOUNDARY
X-My-Header1: 42
X-My-Header2: h2val
Server: MarkLogic
Content-Length: 207
Connection: Keep-Alive
Keep-Alive: timeout=5

--BOUNDARY
Content-Type: application/json
Content-Length: 25

{"name":"b", "value":"2"}
--BOUNDARY
Content-Type: application/json
Content-Length: 32

{"name":"a", "value":["1", "3"]}
--BOUNDARY--
```

The PUT method expects one or more input documents, and a "basename" request parameter value for each one. The basename is used to construct the URI for JSON and XML input; other input document types are ignored. A `written` property is added to each JSON input by the `doSomething` helper function, as shown below:

```
if (docType == 'application/json') {
  var mutableDoc = doc.toObject();
  mutableDoc.written = fn.currentTime();
  xdmp.documentInsert(uri, mutableDoc);
  ...
}
```

The following example invokes the PUT method, passing in a single JSON document as input. The response tells you the URI under which the document is inserted.

```
$ curl --anyauth --user user:password -X PUT -d '{"key":"value"}' \
  -H "Content-type: application/json" \
  'http://localhost:8000/LATEST/resources/js-example?rs:basename=one'

{"written": ["/extensions/one.json"]}
```

If you retrieve the inserted document, you can see that the PUT method added a JSON property named “written” to the input document:

```
$ curl --anyauth --user user:password -X GET \
  -H "Accept: application/json" \
  http://localhost:8000/LATEST/documents?uri=/extensions/one.json

{"key":"value", "written":"11:21:01-08:00"}
```

You can use a multi-part body to pass more than one document to the PUT method. JSON input is modified as shown above. XML documents are inserted unmodified. All other inputs are ignored. The following example command inserts two documents into the database if the multi-part body contains one JSON part and one XML part:

```
$ curl --anyauth --user user:password -X PUT -i \
  -H "Content-type: multipart/mixed; boundary=BOUNDARY" \
  -H "Accept: application/json" \
  --data-binary @./multipart-body
  'http://localhost:8000/LATEST/resources/js-example?rs:basename=a&rs:
basename=b'

HTTP/1.1 200 OK
Content-type: application/json; charset=UTF-8
Server: MarkLogic
Content-Length: 55
Connection: Keep-Alive
Keep-Alive: timeout=5

{"written":["/extensions/a.json", "/extensions/b.xml"]}
```

If you do not include a “basename” parameter value corresponding to each input document, the PUT implementation returns an error response. Extensions should always return errors to the client using `RESTAPI-SRVEXERR` and the pattern described in “Reporting Errors” on page 349. In the example extension, errors are reported using `reportErrToClient` as follows:

```
fn.error(null, 'RESTAPI-SRVEXERR',
         Sequence.from([statusCode, statusMsg, body]));
// unreachable - control does not return from fn.error.
```

For example, if we repeat the previous PUT request, but remove the second occurrence of "basename", a 400 error is returned:

```
$ curl --anyauth --user user:password -X PUT -i \
  -H "Content-type: multipart/mixed; boundary=BOUNDARY" \
  -H "Accept: application/json" \
  --data-binary @./multipart-body
  'http://localhost:8000/LATEST/resources/js-example?rs:basename=a'

HTTP/1.1 400 Bad Request
Content-type: application/json; charset=UTF-8
Server: MarkLogic
Content-Length: 162
Connection: Keep-Alive
Keep-Alive: timeout=5
{
  "errorResponse": {
    "statusCode": 400,
    "status": "Bad Request",
    "messageCode": "RESTAPI-SRVEXERR",
    "message": "Insufficient number of uri basenames. Expected 2 got 1."
  }
}
```

The POST and DELETE methods simply log a message when invoked. If you call them, check the MarkLogic Server error log for output.

9.8 Example: XQuery Resource Service Extension

This section demonstrates the implementation, installation and invocation of a resource service extension implemented in XQuery.

- XQuery Extension Implementation
- Installing the Example Extension
- Using the Example Extension

9.8.1 XQuery Extension Implementation

This example extension implements all the HTTP methods supported by the resource extension interface. It demonstrates the use of input arguments, as well as setting content type and response status codes. The implementation follows the template described in "Creating an XQuery Resource Service Extension" on page 353.

The methods implemented by this extension do the following:

Method	Description

GET	Accepts one or more string values through an “arg1” parameter. Returns an XML document that contains the input argument values using the template:
	`<args>`
	`  <arg1>value</arg1>`
	`<args>`

POST	Logs a message.
PUT	Accepts XML as input and returns “Done”.
DELETE	Logs a message.

The following code implements the contract described above.

```
xquery version "1.0-ml";

(: Namespace pattern must be:
 : "http://marklogic.com/rest-api/resource/{$rname}"
 : and prefix must match resource name :)
module namespace example =
  "http://marklogic.com/rest-api/resource/example";

declare default function namespace
  "http://www.w3.org/2005/xpath-functions";
declare option xdmp:mapping "false";

(: Conventions:
 : Module prefix must match resource name,
 : and function signatures must conform to examples below.
 : The $context map carries state between the extension
 : framework and the extension.
 : The $params map contains parameters set by the caller,
 : for access by the extension.
 :)

(: Function responding to GET method - must use local name 'get':)
declare function example:get(
    $context as map:map,
    $params  as map:map
) as document-node()*
{
    (: set 'output-type', used to generate content-type header :)
    let $output-type :=
        map:put($context,"output-type","application/xml")
    let $arg1 := map:get($params,"arg1")
```

```
    let $content :=
        <args>
            {for $arg in $arg1
             return <arg1>{$arg1}</arg1>
            }
        </args>
    return document { $content }
    (: must return document node(s) :)
};

(: Function responding to PUT method - must use local name 'put'. :)
declare function example:put(
    $context as map:map,
    $params  as map:map,
    $input   as document-node()*
) as document-node()?
{
    (: get 'input-types' to use in content negotiation :)
    let $input-types := map:get($context,"input-types")
    let $negotiate :=
        if ($input-types = "application/xml")
        then () (: process, insert/update :)
        else error((),"ACK",
          "Invalid type, accepts application/xml only")
    return document { "Done"}  (: may return a document node :)
};

(: Function responding to POST method - must use local name 'post'. :)
declare function example:post(
    $context as map:map,
    $params  as map:map,
    $input   as document-node()*
) as document-node()*
{
    xdmp:log("post!")
};

(: Func responding to DELETE method - must use local name 'delete'. :)
declare function example:delete(
    $context as map:map,
    $params  as map:map
) as document-node()?
{
    xdmp:log("delete!")
};
```

9.8.2 Installing the Example Extension

Follow these steps to install the example extension in the modules database associated with your REST API instance:

1. If you do not already have a REST API instance, create one. This example assumes an instance is running on `localhost:8000`.

2. Copy the code from "XQuery Extension Implementation" on page 370 into a file. You can use any file name. This example assumes `example.xqy`.

3. Install the example using a command similar to the following. The extension metadata such as title, version, provider, description, and method interface details are all optional.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    -H "Content-type: application/xquery" -d@"./example.xqy" \
    "http://localhost:8000/LATEST/config/resources/example"
```

Optionally, you can include metadata during installation so that a GET request to `/config/resources` returns more detail. For example:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    -H "Content-type: application/xquery" -d@"./example.xqy" \
    "http://localhost:8000/LATEST/config/resources/example?version=1.0
&provider=marklogic&description=A simple resource
example&method=get&method=post&method=put&method=delete&get:arg1=xs:st
ring*"
```

The table below breaks down the request parameters so it is easier to see what metadata is included.

Request Parameter	Description
`version=1.0`	The extension version number.
`provider=marklogic`	The extension provider.
`description=A simple resource example`	A brief description of the extension.
`method=get`	The extensions supports GET requests.
`method=post`	The extension supports POST requests.
`method=put`	The extension supports PUT requests
`method=delete`	The extension supports DELETE requests
`get:arg1=xs:string*`	The GET method accepts an argument named "arg1" which can occur 1 or more times.

9.8.3 Using the Example Extension

The following example command invokes the GET method of the example extension, passing the string "super fantastic" as arguments to the GET implementation:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
  -H "Accept: application/xml" \
  "http://localhost:8000/LATEST/resources/example?rs:arg1=super
fantastic"

<?xml version="1.0" encoding="UTF-8"?>
<args>
  <arg1>super fantastic</arg1>
</args>
```

The following example invokes the PUT method:

```
$ curl --anyauth --user user:password -X PUT -d "<input/>" \
    -H "Content-type: application/xml" \
    http://localhost:8000/LATEST/resources/example

Done
```

The POST and DELETE methods simply log a message when invoked. If you call them, check the MarkLogic Server error log for output.

9.9 Controlling Access to a Resource Service Extension

Only users with the `rest-extension-user` role (or equivalent privileges) can execute modules in the modules database, including installed REST resource service extensions.

The pre-defined `rest-reader`, `rest-writer`, and `rest-admin` roles inherit `rest-extension-user`. You do not need additional configuration to grant access to your extension to users with these roles.

However, if instead of using the pre-defined roles, you create a custom role with the equivalent privileges, then you must explicitly inherit the `rest-extension-user` role in your custom role. Users with the custom role cannot execute resource service extensions unless you do so.

For details, see "Controlling Access to Documents and Other Artifacts" on page 22.

9.10 Discovering Resource Service Extensions

To discover the name, interface, and other metadata about installed resource extensions, send a GET request to the `/config/resources` service of the form:

```
http://host:port/version/config/resources
```

MarkLogic Server returns a summary of the installed extensions in XML or JSON. The default summary content type is XML. Use the Accept header or `format` request parameter to select the output content type. For details, see "Controlling Input and Output Content Type" on page 31.

The amount of information available about a given extension depends on the amount of metadata provided during installation of the extension. The name and methods are always available. Details such as provider, version, and method parameter information are optional.

By default, this request rebuilds the extension metadata each time it is called to ensure the metadata is up to date. If you find this refresh makes discovery take too long, you can disable the refresh by setting the `refresh` request parameter to false. Disabling the refresh can result in this request returning inaccurate information, but it does not affect the "freshness" or availability of the implementation of any extensions.

The following example requests an XML report. One extension is installed, called "example". The extension implements only a get method, which accepts a string parameter named "genre". The extension provider is "Acme Widgets".

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -H "Accept: application/xml" \
    -X GET http://localhost:8000/LATEST/config/resources
...
<rapi:resources xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:resource>
    <rapi:name>example</rapi:name>
    <rapi:title/>
    <rapi:version/>
    <rapi:provider-name>Acme Widgets</rapi:provider-name>
    <rapi:description/>
    <rapi:methods>
      <rapi:method>
        <rapi:method-name>get</rapi:method-name>
        <rapi:parameter>
          <rapi:parameter-name>genre</rapi:parameter-name>
          <rapi:parameter-type>string</rapi:parameter-type>
        </rapi:parameter>
      </rapi:method>
    </rapi:methods>
    <rapi:resource-source>/v1/resources/example</rapi:resource-source>
  </rapi:resource>
</rapi:resources>
```

9.11 Retrieving the Implementation of a Resource Service Extension

To retrieve the XQuery code that implements an extension, send a GET request to the `/config/resources/{name}` service of the form:

```
http://host:port/version/config/resources/extensionName
```

Where *extensionName* is the name under which the extension is installed.

Set the request Accept header to `application/xquery`.

MarkLogic Server returns the XQuery library module that implements the extension in the response body. For example:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -H "Accept: application/xquery" \
    -X GET http://localhost:8000/LATEST/config/resources/example
...
xquery version "1.0-ml";
module namespace
example="http://marklogic.com/rest-api/resource/example";
declare function example:get( ...
```

9.12 Managing Dependent Libraries and Other Assets

Use the `/ext` service to install, update, retrieve, and delete assets required by your REST application, including additional XQuery library modules required by resource extensions and transforms, and partial update replace library modules.

> **Note:** You can only use the `/ext` service with REST API instances created with MarkLogic Server 7 or later. If you REST API instance was created with an earlier version, refer to the documentation for that version.

The following topics are covered:

- Maintenance of Dependent Libraries and Other Assets
- Installing or Updating an Asset
- Referencing an Asset
- Removing an Asset
- Retrieving an Asset List
- Retrieving an Asset

9.12.1 Maintenance of Dependent Libraries and Other Assets

When you install or update a dependent library module or other asset as described in this section, the asset is replicated across your cluster automatically. There can be a delay of up to one minute between updating and availability.

MarkLogic Server does not automatically remove dependent assets when you delete the related extension or transform.

Since dependent assets are installed in the modules database, they are removed when you remove the REST API instance if you include the modules database in the instance teardown. For details, see “Removing an Instance” on page 43.

If you installed assets in a REST API instance using MarkLogic 6, they cannot be managed using the `/ext` service unless you re-install them using `/ext`. Reinstalling the assets may require additional changes because the asset URIs will change. If you choose not to migrate such assets, continue to maintain them according to the documentation for MarkLogic 6.

9.12.2 Installing or Updating an Asset

Use the `/ext` service to install any assets required by resource service extensions or transformations, and to install partial update replace library modules. Using this service installs an asset into the modules database associated with the REST API instance. If the REST API instance is part of a cluster, the asset is automatically propagated throughout the cluster.

To install or update an asset, send a PUT request to `/ext/{directories}/{asset}` with a URL of the following form:

```
http://host:port/version/ext/directories/asset?perm:role=capability
```

Where *directories* is one or more database directories and *asset* is the asset document name. Use the `perm` request parameter to specify additional permissions on the asset, beyond the permissions granted to rest-admin privileges that are always present. The asset is installed into the modules database with the URI `/ext/directories/asset`.

The following example installs the XQuery library module `my-lib.xqy` into the modules database with the URI `/ext/my/domain/my-lib.xqy`. Execute privileges are granted to the role `my-users`.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -i -d @./my-lib.xqy \
  -H "Content-type: application/xquery" \
  "http://localhost:8000/LATEST/ext/my/domain/my-lib.xqy?perm:my-users
=execute"
```

Note: If you're installing an asset whose content is sensitive to line breaks, such as a JavaScript module that contains '//' style comments, use `--data-binary` rather than `-d` to pass the content to `curl`. For details, see "Introduction to the curl Tool" on page 27.

You can insert assets into the modules database as JSON, XML, text, or binary documents. MarkLogic Server determines the document format by examining the `format` request parameter, the MIME type in the `Content-type` HTTP header, or the document URI extension, in that order. For example, if the `format` parameter is present, it determines the document type. If there is no `format` parameter and no `Content-type` header, the URI extension (.xml, .jpg, etc.) determines the document type, based on the MIME type mappings defined for your installation; see the Mimetypes section of the Admin Interface.

9.12.3 Referencing an Asset

To use a dependent library installed with `/ext` in your extension or transform module, use the same URI under which you installed the dependent library. For example, if a dependent library is installed with the URI `/ext/my/domain/my-lib.xqy`, then the extension module using this library should include an import of the following form:

```
import module namespace dep="mylib" at "/ext/my/domain/lib/my-lib.xqy";
```

9.12.4 Removing an Asset

You can remove either a single asset or all the assets in a directory in the modules database associated with your REST API instance.

To remove a single asset, send a DELETE request with a URL of the following form:

```
http://host:port/version/ext/directories/asset
```

Where *directories* is one or more database directories and *asset* is the asset document name. For example, the following command removes an XQuery library module with the URI `/ext/my/domain/my-lib.xqy`:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X DELETE -i\
  "http://localhost:8000/LATEST/ext/my/domain/my-lib.xqy
```

To remove all assets in a modules database directory, send a DELETE request with a URL of the following form:

```
http://host:port/version/ext/directories/
```

Note: The URL must end with a terminating path separator (“/”) so that MarkLogic Server recognizes it as a directory.

Where *directories* is one or more database directories. For example, the following command removes all assets in the directory `/ext/my/domain/`:

```
$ curl --anyauth --user user:password -X DELETE -i\
  "http://localhost:8000/LATEST/ext/my/domain/
```

9.12.5 Retrieving an Asset List

You can retrieve a list of all assets in the modules database by sending a GET request with a URL of the following form:

```
http://host:port/version/ext/
```

You can retrieve a list of assets in a particular directory by sending a GET request with a URL of the following form, where *directories* is one or more database directory path steps:

```
http://host:port/version/ext/directories
```

You can request results as either XML or JSON, using either the HTTP Accept header or the `format` request parameter.

The following example retrieves a list of all assets installed under the database directory `/ext/my/domain/`, as XML:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET -i\
  -H "Accept: application/xml" \
  http://localhost:8000/LATEST/ext/my/domain/
...
HTTP/1.1 200 OK
Server: MarkLogic
Content-Type: text/xml; charset=UTF-8
Content-Length: 125
Connection: Keep-Alive
Keep-Alive: timeout=5

<rapi:assets xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:asset>/ext/patch/replace/my-lib.xqy</rapi:asset>
</rapi:assets>
```

The following example makes the same request, but returns JSON results.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET -i\
  -H "Accept: application/json" \
  http://localhost:8000/LATEST/ext/my/domain/
...
HTTP/1.1 200 OK
Server: MarkLogic
Content-Type: text/plain; charset=UTF-8
Content-Length: 54
Connection: Keep-Alive
Keep-Alive: timeout=5

{"assets":[
  {"asset":"/ext/my/domain/my-lib.xqy"}
]}
```

9.12.6 Retrieving an Asset

To retrieve the contents of an asset document, send a GET request with a URL of the following form:

```
http://host:port/version/ext/directories/asset
```

Where *directories* is one or more database directories and *asset* is the asset document name. For example, the following command retrieves an XQuery library module with the URI `/ext/my/domain/my-lib.xqy`:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET -i \
  -Accept: "application/xquery" \
  "http://localhost:8000/LATEST/ext/my/domain/my-lib.xqy
...
HTTP/1.1 200 Document Retrieved
Content-type: application/xquery
Server: MarkLogic
Content-Length: 633
Connection: Keep-Alive
Keep-Alive: timeout=5
...asset contents...
```

The response data content type is determined by the MIME type in the Accept header. No content conversions are applied. For example, if you retrieve a binary document containing PDF and set the Accept header to `application/xml`, the response Content-type will be `application/xml`, even though the data in the body is binary.

9.13 Evaluating an Ad-Hoc Query

The `/eval` service enables you to send blocks of XQuery and server-side JavaScript code to MarkLogic Server for evaluation. This is equivalent to calling the builtin server function `xdmp:eval` (XQuery) or `xdmp.eval` (JavaScript). The following topics are covered:

- Required Privileges
- Evaluating an Ad-Hoc JavaScript Query
- Evaluating an Ad-Hoc XQuery Query
- Specifying External Variable Values
- Interpreting the Results of an Ad-Hoc Query
- Managing Session State

9.13.1 Required Privileges

Using the `/eval` service requires special privileges not required for other REST Client API services. You must have the following privileges or the equivalent:

- `http://marklogic.com/xdmp/privileges/xdmp-eval`
- `http://marklogic.com/xdmp/privileges/xdmp-eval-in`
- `http://marklogic.com/xdmp/privileges/xdbc-eval`
- `http://marklogic.com/xdmp/privileges/xdbc-eval-in`

9.13.2 Evaluating an Ad-Hoc JavaScript Query

To send JavaScript code to MarkLogic Server for evaluation, use `POST:/v1/eval`. Send a POST request to the `/eval` service with a URL of the following form:

```
http://host:port/version/eval
```

The POST body MIME type must be `x-www-form-urlencoded`, with the ad-hoc code as the value of the `javascript` form parameter and any external variables in the `vars` form parameter. For details on passing in variable values, see “Specifying External Variable Values” on page 385.

Your query must be expressed using the MarkLogic server-side JavaScript dialect. For details, see the *JavaScript Reference Guide*.

For example, suppose you want to evaluate the following JavaScript code, where `word1` and `word2` are external variable values supplied by the request:

```
word1 + " " + word2
```

The following request causes the code to be evaluated by MarkLogic Server. Use the curl `--data-urlencode` option to force URL-encoding of the form parameter values in the POST body.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X POST -i \
    -H "Content-type: application/x-www-form-urlencoded" \
    -H "Accept: multipart/mixed" \
    --data-urlencode javascript='word1 + " " + word2' \
    --data-urlencode vars='{"word1":"hello","word2":"world"}' \
    http://localhost:8000/LATEST/eval
```

The response is always `multipart/mixed`, with one part for each value returned by the evaluated code. The part headers can contain additional type information about the returned data in the `X-Primitive` header. For details, see “Interpreting the Results of an Ad-Hoc Query” on page 386.

For example, the above request returns results similar to the following:

```
HTTP/1.1 200 OK
Server: MarkLogic 8.0-20141122
Set-Cookie: TxnMode=auto; path=/
Set-Cookie: TxnID=null; path=/
Content-Type: multipart/mixed; boundary=198d5dc521a0e8cf
Content-Length: 113
Connection: Keep-Alive
Keep-Alive: timeout=5


--198d5dc521a0e8cf
Content-Type: text/plain
X-Primitive: untypedAtomic
```

```
hello world
--198d5dc521a0e8cf--
```

You can return documents, objects, and arrays as well as atomic values. To return multiple items, you must return either a JavaScript `Sequence` or an XQuery sequence. You can construct a `Sequence` from an array-like or a generator, and many builtin functions that can return multiple values return a `Sequence`. To construct a sequence, apply `xdmp.arrayValues` or `Sequence.from` to a JavaScript array.

For example, to extend the previous example to return the two input values as well as the concatenated string, accumulate the results in an array and then apply `Sequence.from` to the array:

```
Sequence.from([word1, word2, word1 + " " + word2])
```

The following command executes the above code. The response contains 3 parts: one for the string value “hello”, one for the string value “world”, and one for the concatenation result, “hello world”.

```
$ curl --anyauth --user user:password -X POST -i \
    -H "Content-type: application/x-www-form-urlencoded" \
    -H "Accept: multipart/mixed" \
    --data-urlencode javascript='Sequence.from([word1, word2, word1 + "
" + word2])' \
    --data-urlencode vars='{"word1":"hello","word2":"world"}' \
    http://localhost:8000/LATEST/eval

HTTP/1.1 200 OK
Server: MarkLogic 8.0-20141122
Set-Cookie: TxnMode=query; path=/
Set-Cookie: TxnID=null; path=/
Content-Type: multipart/mixed; boundary=3d0b32f66d6d0f33
Content-Length: 279
Connection: Keep-Alive
Keep-Alive: timeout=5


--3d0b32f66d6d0f33
Content-Type: text/plain
X-Primitive: untypedAtomic

hello
--3d0b32f66d6d0f33
Content-Type: text/plain
X-Primitive: untypedAtomic

world
--3d0b32f66d6d0f33
Content-Type: text/plain
X-Primitive: untypedAtomic

hello world
--3d0b32f66d6d0f33--
```

By default, the ad-hoc code is evaluated in the context of the database associated with the REST API instance that receives the request. To evaluate the code against another database, specify the database name or id through the database URL parameter. For example, the following request uses the "example" database:

```
$ curl --anyauth --user user:password -X POST -i \
    -H "Content-type: application/x-www-form-urlencoded" \
    -H "Accept: multipart/mixed" \
    --data-urlencode javascript='word1 + " " + word2' \
    --data-urlencode vars='{"word1":"hello","word2":"world"}' \
    http://localhost:8000/LATEST/eval?database=example
```

9.13.3 Evaluating an Ad-Hoc XQuery Query

To send XQuery code to MarkLogic Server for evaluation, use `POST:/v1/eval`. Send a POST request to the `/eval` service with a URL of the following form:

```
http://host:port/version/eval
```

The POST body MIME type must be `x-www-form-urlencoded`, with the ad-hoc code as the value of the `xquery` form parameter and any external variables in the `vars` form parameter. For details on passing in variable values, see "Specifying External Variable Values" on page 385.

For example, suppose you want to evaluate the following XQuery code, where `word1` and `word2` are external variable values supplied by the request:

```
xquery version "1.0-ml";
declare variable $word1 as xs:string external;
declare variable $word2 as xs:string external;
fn:concat($word1, " ", $word2)
```

The following request body supplies the query code and the parameter values. (The values should be urlencoded. They're shown unencoded here for readability.)

```
xquery=
xquery version "1.0-ml";
declare variable $word1 as xs:string external;
declare variable $word2 as xs:string external;
fn:concat($word1, " ", $word2)

&vars={"word1":"hello","word2":"world"}
```

The following request causes the code to be evaluated by MarkLogic Server, assuming file `concat.xqy` contains the XQuery shown above. Use the curl `--data-urlencode` option to force URL-encoding of the form parameter values in the POST body.

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X POST -i \
    -H "Content-type: application/x-www-form-urlencoded" \
    -H "Accept: multipart/mixed" \
```

```
    --data-urlencode xquery@./concat.xqy \
    --data-urlencode vars='{"word1":"hello","word2":"world"}' \
    http://localhost:8000/LATEST/eval
```

The response is always `multipart/mixed`, with one part for each value returned by the evaluated code. The part headers can contain additional type information about the returned data in the `X-Primitive` header. For details, see “Interpreting the Results of an Ad-Hoc Query” on page 386.

For example, the above request returns results similar to the following:

```
HTTP/1.1 200 OK
Server: MarkLogic 8.0-20141122
Set-Cookie: TxnMode=auto; path=/
Set-Cookie: TxnID=null; path=/
Content-Type: multipart/mixed; boundary=7e235c5751341cff
Content-Length: 106
Connection: Keep-Alive
Keep-Alive: timeout=5


--7e235c5751341cff
Content-Type: text/plain
X-Primitive: string

hello world
--7e235c5751341cff--
```

You can return documents and other complex values, as well as atomic values. To return multiple items, return a sequence. The multipart response contains a part for each sequence item.

The following query extends the previous example to return the two input values as well as the concatenated string:

```
xquery version "1.0-ml";
declare variable $word1 as xs:string external;
declare variable $word2 as xs:string external;
($word1, $word2, fn:concat($word1, " ", $word2))
```

The following command evaluates the query, assuming the query is in a file named `concat2.xqy`. The response contains 3 parts, one for the value of `word1`, one for the value of `word2`, and one for the concatenation result, “hello world”.

```
$ curl --anyauth --user user:password -X POST -i \
    -H "Content-type: application/x-www-form-urlencoded" \
    -H "Accept: multipart/mixed" \
    --data-urlencode xquery@./concat2.xqy \
    --data-urlencode vars='{"word1":"hello","word2":"world"}' \
    http://localhost:8000/LATEST/eval

HTTP/1.1 200 OK
Server: MarkLogic 8.0-20141122
```

```
Set-Cookie: TxnMode=query; path=/
Set-Cookie: TxnID=null; path=/
Content-Type: multipart/mixed; boundary=3d0b32f66d6d0f33
Content-Length: 279
Connection: Keep-Alive
Keep-Alive: timeout=5


--3d0b32f66d6d0f33
Content-Type: text/plain
X-Primitive: string

hello
--3d0b32f66d6d0f33
Content-Type: text/plain
X-Primitive: string

world
--3d0b32f66d6d0f33
Content-Type: text/plain
X-Primitive: string

hello world
--3d0b32f66d6d0f33--
```

By default, the ad-hoc query is evaluated in the context of the database associated with the REST API instance that receives the request. To evaluate the query against another database, specify the database name or id through the database URL parameter. For example, the following request evaluates the query against the “example” database:

```
$ curl --anyauth --user user:password -X POST -i \
    -H "Content-type: application/x-www-form-urlencoded" \
    -H "Accept: multipart/mixed" \
    --data-urlencode xquery@./concat.xqy \
    --data-urlencode vars='{"word1":"hello","word2":"world"}' \
    http://localhost:8000/LATEST/eval?database=example
```

9.13.4 Specifying External Variable Values

You can pass values to an ad-hoc query (or invoked module) at runtime using external variables. Specify the variable values in the POST body using the `vars` form parameter. The `vars` parameter has the following form:

```
vars={name:value, name:value, ...}
```

For example, the following vars parameter supplies values for 2 external variables, named `word1`, and `word2`:

```
vars={word1:"hello",word2:"world"}
```

If you're evaluating or invoking XQuery code, you must declare the variables explicitly in the ad-hoc query or module. For example, the following prolog declares two external variables whose values can be supplied through the above `vars` parameter.

```
xquery version "1.0-ml";
declare variable $word1 as xs:string external;
declare variable $word2 as xs:string external;
...
```

If you're evaluating or invoking XQuery code that depends on variables in a namespace, use Clark notation on the variable name. That is, specify the name using notation of the form `{`*`namespaceURI`*`}`*`name`*. For example, given the following query saved to the file `nsquery.xqy`:

```
xquery version "1.0-ml";
declare namespace my = "http://example.com";
declare variable $my:who as xs:string external;
fn:concat("hello", " ", $my:who)
```

You can specify the value of `$my:who` as shown in the command below. Note that you must quote the namespace qualified variable name.

```
$ curl --anyauth --user user:password -i -X POST \
    -H "Content-type: application/x-www-form-urlencoded" \
    --data-urlencode xquery@./nsquery.xqy
    --data-urlencode vars='{"{http://example.com}who":"world"}' \
    http://localhost:8000/LATEST/eval

HTTP/1.1 200 OK
Server: MarkLogic 8.0-20141122
Set-Cookie: TxnMode=auto; path=/
Set-Cookie: TxnID=null; path=/
Content-Type: multipart/mixed; boundary=ea9250df22f6543f
Content-Length: 106
Connection: Keep-Alive
Keep-Alive: timeout=5


--ea9250df22f6543f
Content-Type: text/plain
X-Primitive: string

hello world
--ea9250df22f6543f--
```

9.13.5 Interpreting the Results of an Ad-Hoc Query

When you make a POST request to the `/eval` or `/invoke` service, the result is always a `multipart/mixed` response, with a part for each value returned by the query. Depending on the type of value, the part header can contain information that helps your application interpret the value.

For example, an atomic value (`anyAtomicType`, a type derived from `anyAtomicType`, or an equivalent JavaScript type) has an X-Primitive part header that can specify an explicit type such as `integer`, `string`, or `date`. The reported type may be more general than the actual type. Types derived from `anyAtomicType` include anyURI, boolean, dateTime, double, and string. For details, see http://www.w3.org/TR/xpath-functions/#datatypes.

Returning a JavaScript `null` value or an XQuery empty sequences results in an empty response.

The table below lists the Content-type and X-Primitive part header values for a variety of types. The response for other output is undefined, including the output from constructors such as `attribute()`, `comment()`, `processing-instruction()`, `boolean-node()`, `cts:query()`, or a JavaScript object.

Value Type	Content-type Header Value	X-Primitive Header Value
`document-node[object-node()]`	`application/json`	*undefined*
`object-node()`	`application/json`	*undefined*
`document-node[array-node()]`	`application/json`	*undefined*
`array-node()`	`application/json`	*undefined*
`map:map`	`application/json`	*undefined*
`json:array`	`application/json`	*undefined*
`document-node[element()]`	`text/xml`	*undefined*
`element()`	`text/xml`	*undefined*
`document-node[binary()]`	`application/x-unknown-content-type`	*undefined*
`binary()`	`application/x-unknown-content-type`	*undefined*
`document-node[text()]`	`text/plain`	`text`
`text()`	`text/plain`	`text`

Value Type	Content-type Header Value	X-Primitive Header Value
`any atomic value`	`text/plain`	`anyAtomicType` or a derived type
`JavaScript string`	`text/plain`	`string`
`JavaScript number`	`text/plain`	`decimal`, a derived type such as `integer`, or `string` (for infinity)
`JavaScript boolean`	`text/plain`	`Boolean`

9.13.6 Managing Session State

When you use `POST:/v1/invoke` or `POST:/v1/eval` MarkLogic returns a cookie. Normally, your client application code should ignore the cookie. However, if your server-side code sets a session field, then your client application must send the same cookie to the same host for any subsequent eval or invoke requests that access the session field.

You should also include any load balancer cookies if you are using a load balancer.

If you make an eval or invoke request that requires access to an existing session field in the context of a multi-statement transaction, include both the transaction cookie and the session cookie in the request.

9.14 Evaluating a Module Installed on MarkLogic Server

`POST:/v1/invoke` enables you to evaluate an XQuery or server-side JavaScript module installed on MarkLogic Server. This is equivalent to calling the builtin server function `xdmp:invoke` (XQuery) or `xdmp.invoke` (JavaScript).

Using the `/invoke` service requires special privileges not required for other REST Client API services. You must have the following privileges or their equivalent:

- `http://marklogic.com/xdmp/privileges/xdmp-invoke`
- `http://marklogic.com/xdmp/privileges/xdmp-invoke-in`
- `http://marklogic.com/xdmp/privileges/xdbc-invoke`
- `http://marklogic.com/xdmp/privileges/xdbc-invoke-in`

To evaluate an XQuery or JavaScript module installed on MarkLogic Server, make a POST request to the /invoke service with a URL of the following form:

```
http://host:port/version/invoke
```

The POST body MIME type must be `x-www-form-urlencoded`, with the module path as the value of the `module` form parameter and any external variables in the `vars` form parameter. For details on passing in variable values, see “Specifying External Variable Values” on page 385.

For example, assuming you previously used a `PUT /LATEST/ext/invoke/example.sjs` request to install a JavaScript module on MarkLogic Server, then the following command invokes the module:

```
$ curl --anyauth --user user:password -i -X POST \
    -H "Content-type: application/x-www-form-urlencoded" \
    --data-urlencode module=/ext/invoke/example.sjs \
    http://localhost:8000/LATEST/invoke
```

The `module` parameter must be the path to a module installed in the modules database associated with the REST API instance. You can use the `/ext` service to install your module, as described in “Managing Dependent Libraries and Other Assets” on page 376. The module path is resolved using the rules described in “Rules for Resolving Import, Invoke, and Spawn Paths” on page 90 in the *Application Developer's Guide*. If you install your module using `PUT /v1/ext/{name}`, the module path always begins with `/ext/`.

If the invoked module expects input variable values, specify them using the `vars` form parameter, as described in “Specifying External Variable Values” on page 385.

The response is always `multipart/mixed`, with one part for each value returned by the invoked module. The part headers can contain additional type information about the returned data in the `X-Primitive` header. For details, see “Interpreting the Results of an Ad-Hoc Query” on page 386.

If your server-side code sets a session field that subsequent eval or invoke requests depend upon, your subsequent calls must pass along the session cookie returned by the first such request. For details, see “Managing Session State” on page 388.

The following example first installs a module in the modules database associated with the REST API instance using the `/ext` service, then invokes module through the `/invoke` service.

1. Save the following code to file. This is the code to be invoked. The rest of this example demonstrates using the JavaScript module, saved to a file named `module.sjs`.

   a. For JavaScript, copy the following code into a file named `module.sjs`.

   ```
   Sequence.from([word1, word2, word1 + " " + word2])
   ```

   b. For XQuery, copy the following code into a file named `module.xqy`.

   ```
   xquery version "1.0-ml";
   declare variable $word1 as xs:string external;
   declare variable $word2 as xs:string external;

   ($word1, $word2, fn:concat($word1, " ", $word2))
   ```

2. Install the module in the modules database of the REST API instance. Modify the username and password in the example commands to fit your environment.

   a. To install the JavaScript module, use the following command:

```
curl --anyauth --user user:password -X PUT -i \
    --data-binary @./module.sjs \
    -i -H "Content-type: application/vnd.marklogic-javascript"
    http://localhost:8000/LATEST/ext/invoke/example.sjs
```

   b. To install the XQuery module, use the following command:

```
curl --anyauth --user user:password -X PUT -i -d @./module.xqy \
    -H "Content-type: application/xquery"
    http://localhost:8000/LATEST/ext/invoke/example.xqy
```

3. Invoke the module. Modify the username and password in the example commands to fit your environment.

   a. To invoke the JavaScript module, use the following command:

```
curl --anyauth --user user:password -i -X POST \
    -H "Content-type: application/x-www-form-urlencoded" \
    --data-urlencode module=/ext/invoke/example.sjs \
    --data-urlencode vars='{"word1":"hello","word2":"world"}' \
    http://localhost:8000/LATEST/invoke
```

   b. To invoke the XQuery module, use the following command:

```
curl --anyauth --user user:password -i -X POST \
    -H "Content-type: application/x-www-form-urlencoded" \
    --data-urlencode module=/ext/invoke/example.xqy \
    --data-urlencode vars='{"word1":"hello","word2":"world"}' \
    http://localhost:8000/LATEST/invoke
```

The result of running this example is similar to the following. There are 3 parts in the response body, corresponding to the 3 values returned by the invoked module.

```
HTTP/1.1 200 OK
Server: MarkLogic 8.0-20141122
Set-Cookie: TxnMode=query; path=/
Set-Cookie: TxnID=null; path=/
Content-Type: multipart/mixed; boundary=09ff9ca463ea6272
Content-Length: 279
Connection: Keep-Alive
Keep-Alive: timeout=5


--09ff9ca463ea6272
Content-Type: text/plain
X-Primitive: string
```

```
hello
--09ff9ca463ea6272
Content-Type: text/plain
X-Primitive: string

world
--09ff9ca463ea6272
Content-Type: text/plain
X-Primitive: string

hello world
--09ff9ca463ea6272--
```

10.0 Migrating REST Applications

This chapter describes how to migrate an application hosted on a REST Client API instance from one MarkLogic Server installation to another, such as migrating an application from your development environment to a test or production environment.

> **Note:** These instructions only apply to applications that depend on a REST API instance, such as those created with the REST Client API or the MarkLogic Java API.

The following topics are covered:

- Before You Begin
- Migrating the REST API Instance and Database Configuration
- Migrating the Contents of the Database

10.1 Before You Begin

Migrating a REST API application requires migrating the REST API instance (an App Server and modules database) and your content database. The REST API instance and the configuration of your content database are migrated using Configuration Manager. You can migrate the contents of your content database using any tool, but this guide assumes MarkLogic Content Pump (`mlcp`).

The migration procedure outlined by this chapter assumes you have the following:

- MarkLogic 9 on both the source and destination hosts.
- MarkLogic Content Pump (`mlcp`), XQSync, or an equivalent tool capable of copying content and metadata from one MarkLogic Server database to another. These instructions assume `mlcp`.

  To download and install mlcp, see Loading Content Using MarkLogic Content Pump in the *Loading Content Into MarkLogic Server Guide*.

- `curl` or an equivalent tool for sending HTTP requests; see “Introduction to the curl Tool” on page 27. These instructions assume `curl`.
- Administrative user privileges on the source and destination MarkLogic Server instances.

10.2 Migrating the REST API Instance and Database Configuration

Use Configuration Manager to migrate your REST API instance and the configuration of your content database from to the destination MarkLogic Server cluster. If you are not familiar with Configuration Manager, see Using the Configuration Manager in *Administrator's Guide*.

- Export the Configuration from the Source Cluster
- Import the Configuration to the Destination Cluster

10.2.1 Export the Configuration from the Source Cluster

This procedure saves the configuration of your content database and REST API instance App Server, and the configuration and contents of the modules database to a ZIP file using Configuration Manager. For detailed instructions on using Configuration Manager, see Exporting a Configuration in *Administrator's Guide*.

> **Note:** You are not required to include the content database in this step, but if you do not, you must ensure it exists on the destination cluster before importing the REST API instance configuration.

1. Open the Configuration Manager on the source cluster in your browser. Authenticate as a user with the `manage-admin` role or equivalent privileges. For example, navigate to:

   ```
   http://source-host:8002/nav
   ```

2. Click on the Export tab. The Package Export interface is displayed.

3. If you want to migrate your content database configuration, place a check in the Settings column for your content database.

4. Click on Servers in the left column. Configuration Manager displays the App Servers available for export.

5. Place a check in the Settings and Modules columns for your REST API instance App Server.

6. Click the Export button in the upper right corner and select a location and name for the exported configuration ZIP file.

10.2.2 Import the Configuration to the Destination Cluster

This procedure uses Configuration Manager and the ZIP file created in "Export the Configuration from the Source Cluster" on page 394 to re-create your content database and REST API instance on the destination MarkLogic Server cluster. For detailed instructions on using Configuration Manager, see Importing a Configuration in *Administrator's Guide*.

> **Note:** If you did not include your content database configuration in the ZIP file, you must ensure that the content database exists on the destination cluster before performing this procedure.

1. Open the Configuration Manager on the destination cluster in your browser. Authenticate as a user with admin privileges. For example, navigate to:

   ```
   http://dest-host:8002/nav
   ```

2. Click on the Import tab. The Package Import interface is displayed.

3. Click the Browse button and browse to the ZIP containing your configuration.

4. Click Compare to import the package. The package contents are displayed. No configuration changes have been made at this time.

5. Click Apply. Configuration Manager re-creates the packaged configurations and the contents of the modules database.

6. On the destination cluster, create any MarkLogic Server users and roles required by your application. Use the same user and role name as on the source host.

Your REST API instance is fully functional at this point, including any resource extensions, persistent query options, or transformations stored in the modules database. If you imported your content database, then the database and its forests exist, but the database is still empty.

You can use the Admin Interface or Configuration Manager to confirm the existence of your REST API instance and content database. If you application includes resource extensions, transformations, or persistent query options, you can query them. For example, you can get a list of persistent query options using `GET /v1/config/query`.

10.3 Migrating the Contents of the Database

Use the procedure in this section to migrate the contents of your application content database to the destination cluster. The database must already exist on the destination cluster using `mlcp`. You can also use XQSync or a similar tool to copy the contents.

The following procedure copies the entire contents of the source content database into the destination content database, using mlcp and the XDBC App Server pre-configured on port 8000 when you install MarkLogic 8 or later. If you are using an older version of MarkLogic, you may need to create an XDBC App Server for use with mlcp and explicitly specify the input and/or output port on the mlcp command line.

1. If the source and destination hosts are both reachable from the host where you perform the copy, use the mlcp `copy` command, as described in Copying Content Between Databases in the *mlcp User Guide*. For example:

```
$ mlcp.sh copy -mode local \
   -input_host src-host -input_username user -input_password password \
   -output_host dest-host -output_username user \
   -output_password password
```

2. If you cannot reach both the source and destination host from the host where you perform the copy, use the mlcp `export` and `import` commands, as described in Exporting Content from MarkLogic Server and Importing Content Into MarkLogic Server in the *mlcp User Guide*. For example:

```
# Export the source database content to an archive
$ mlcp.sh export -mode local -output_type archive \
    -host src-host -username user -password password \
    -output_file_path an-existing-directory
# One or more ZIP files are created in the output directory.

# Copy the archive files to a host that can reach the destination host

# Import the archive to the destination database
$ mlcp.sh import -mode local -input_file_type archive \
    -host dest-host -username user -password password
    -input_file_path dir-containing-archive
```

11.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at http://help.marklogic.com to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the Support Handbook for instructions on registering support contacts and on working with the MarkLogic Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at http://developer.marklogic.com. For technical questions, we encourage you to ask your question on Stack Overflow.

12.0 Copyright

MarkLogic Server 9.0 and supporting products.
Last updated: November 18, 2019

COPYRIGHT

© 2019 MarkLogic Corporation. All rights reserved.
This technology is protected by U.S. Patent No. 7,127,469B2, U.S. Patent No. 7,171,404B2, U.S. Patent No. 7,756,858 B2, and U.S. Patent No 7,962,474 B2, US 8,892,599, and US 8,935,267.

The MarkLogic software is protected by United States and international copyright laws, and incorporates certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers set forth below.

For all copyright notices, including third-party copyright notices, see the Combined Product Notices for your version of MarkLogic.