MarkLogic Server

REST Application Developer's Guide

MarkLogic 6
September, 2012

Last Revised: 6.0-3, April, 2013

Copyright © 2013 MarkLogic Corporation. All rights reserved.

1.0 Introduction to the MarkLogic REST API 8
1.1 Getting Started with the MarkLogic REST API 8
1.1.1 Preparation 8
1.1.2 Create a REST API Instance 9
1.1.3 Load Documents Into the Database 10
1.1.4 Search the Database 11
1.1.5 Tear Down the REST API Instance 13
1.2 Overview of *MarkLogic REST API* Services 13
1.3 Security Requirements 15
1.4 Terms and Definitions 16
1.5 Understanding REST Resources 17
1.5.1 Addressing a Resource 17
1.5.2 Specifying the REST API Version 18
1.5.3 Specifying Parameters in a Resource Address 18
1.6 Understanding the Example Commands 19
1.6.1 Introduction to the curl Tool 19
1.6.2 Modifying the Example Commands for Windows 20
1.7 Controlling Input Content Type 20
1.8 Controlling Output Content Type 21
1.9 Error Reporting 22

2.0 Administering *MarkLogic REST API* Instances 24
2.1 Creating an Instance 24
2.2 Removing an Instance 26
2.3 Retrieving Configuration Information 27
2.3.1 Retrieving Configuration for All Instances 27
2.3.2 Retrieving Instance Configuration by Content Database 28
2.3.3 Retrieving Instance Configuration by Instance Name 29
2.4 Configuring Instance Properties 30
2.4.1 Instance Configuration Properties 30
2.4.2 Listing Instance Property Settings 31
2.4.3 Setting Instance Properties 32
2.4.4 Resetting Instance Properties 33

3.0 Manipulating Documents 35
3.1 Summary of the /documents Service 35
3.2 Loading Content into the Database 36
3.2.1 Loading Content 36
3.2.2 Adding Metadata 37
3.2.2.1 Example: Replacing One Metadata Category Using XML 38
3.2.2.2 Example: Replacing Multiple Metadata Categories Using XML . 38
3.2.2.3 Example: Replacing Multiple Metadata Categories Using JSON 39
3.2.3 Loading Content and Adding Metadata in the Same Request 39

3.2.3.1 Loading Content and Metadata Using Request Parameters 39
3.2.3.2 Loading Content and Metadata Using a Multipart Message 40
3.2.4 Loading JSON Documents Into the Database 41
3.2.5 Controlling Access to Documents Created with the REST API 42
3.2.6 Transforming Content During Ingestion 43
3.3 Retrieving Documents from the Database 43
3.3.1 Retrieving the Contents of a Document 44
3.3.2 Retrieving Metadata About a Document 44
3.3.3 Retrieving Content and Metadata in a Single Request 45
3.3.4 Transforming Content During Retrieval 46
3.4 Performing a Lightweight Document Check 46
3.5 Removing Documents from the Database 47
3.5.1 Removing a Document or Metadata 48
3.5.2 Removing Multiple Documents 48
3.5.3 Removing All Documents 49
3.6 Using Optimistic Locking to Update Documents 49
3.6.1 Understanding Optimistic Locking 50
3.6.2 Enabling Optimistic Locking 50
3.6.3 Using Optimistic Locking 51
3.6.4 Example: Updating a Document Using Optimistic Locking 52
3.7 Client-Side Cache Management Using Content Versioning 53
3.7.1 Enabling Content Versioning 54
3.7.2 Using Content Versioning for Cache Refreshing 54
3.7.3 Example: Refreshing a Cached Document 55
3.8 Working with Binary Documents 56
3.8.1 Types of Binary Documents 56
3.8.2 Streaming Binary Content 57
3.9 Working with Metadata 57
3.9.1 Metadata Categories 58
3.9.2 XML Metadata Format 58
3.9.3 JSON Metadata Format 59

4.0 Using and Configuring Query Features 61
4.1 Query Feature Overview 61
4.2 Querying Documents and Metadata 62
4.2.1 Constraining a Query by Collection or Directory 63
4.2.2 Searching With String Queries 63
4.2.3 Searching With Structured Queries 64
4.2.4 Searching By JSON Key and Value 65
4.2.5 Searching By Element or Attribute Name and Value 67
4.2.6 Debugging /search Queries With Logging 68
4.3 Querying Lexicons and Range Indexes 70
4.3.1 Querying the Values in a Lexicon or Range Index 71
4.3.2 Finding Value Co-Occurrences in Lexicons 73
4.3.3 Using a Query to Constrain Results 76
4.3.4 Identifying Lexicon and Range Index Values in Query Options 77

4.3.4.1 Defining Queryable Lexicon or Range Index Values 77
4.3.4.2 Defining Queryable Lexicon or Range Index Co-Occurrences 79
4.3.5 Creating Indexes on JSON Keys 80
4.4 Analyzing Lexicons and Range Indexes With Aggregate Functions 81
4.4.1 Aggregate Function Overview 82
4.4.2 Using Query Options to Apply Aggregate Functions 82
4.4.3 Using Request Parameters to Apply Aggregate Functions 84
4.4.4 Example: Applying a Builtin Aggregate Function 86
4.4.5 Example: Applying an Aggregate UDF 86
4.5 Specifying Dynamic Query Options with Combined Query 87
4.5.1 Syntax 88
4.5.2 Interaction with Queries in Request Parameters 88
4.5.3 Interaction with Persistent Query Options 89
4.5.4 Performance Considerations 90
4.5.5 Combined Query Examples 91
4.5.5.1 Example: Overriding Persistent Constraints 91
4.5.5.2 Example: Modifying the Search Response 92
4.6 Configuring Query Options 92
4.6.1 Controlling Queries With Options 93
4.6.2 Adding Query Options to a Request 93
4.6.3 Creating or Modifying Query Options 94
4.6.4 Creating or Modifying One Option 96
4.6.5 Checking Index Availability 97
4.6.6 Retrieving Options 99
4.6.7 Retrieving a List of Installed Query Options 100
4.6.8 Removing Query Options 100
4.6.8.1 Removing Query Options 101
4.6.8.2 Removing a Single Option 101
4.6.8.3 Removing All Named Query Options 101
4.7 Using Namespace Bindings 102
4.7.1 When Do You Need a Namespace Binding 102
4.7.2 Creating or Updating a Namespace Binding 102
4.7.3 Creating or Updating Multiple Namespace Bindings 104
4.7.4 Listing Available Namespace Bindings 106
4.7.5 Deleting Namespace Bindings 107
4.8 Generating Search Facets 108
4.9 Paginating Results 110
4.10 Customizing Search Snippets 111
4.10.1 Customizing Search Snippets With Query Options 112
4.10.2 Creating Your Own Snippet Extension 115

5.0 Managing Transactions 121
5.1 Service Summary 121
5.2 Overview of RESTful Transactions 121
5.3 Creating a Transaction 122
5.4 Associating a Transaction with a Request 123

5.5 Committing or Rolling Back a Transaction 124
5.6 Checking Transaction Status 124

6.0 Working With Content Transformations 127
6.1 Writing Transformations 127
6.1.1 Guidelines for Writing Transforms 127
6.1.2 Writing XQuery Transformations 128
6.1.3 Writing XSLT Transformations 129
6.1.4 Reporting Errors 129
6.1.5 Context Map Keys 131
6.2 Installing Transformations 131
6.2.1 Basic Installation 131
6.2.2 Defining Transform-Specific Parameters 132
6.2.3 Including Transform Metadata 132
6.3 Applying Transformations 132
6.4 Discovering Transformations 133
6.5 Retrieving the Implementation of a Transformation 134
6.6 XQuery Example: Adding an Attribute During Ingestion 134
6.7 XSLT Example: Adding an Attribute During Ingestion 137

7.0 Extending the REST API 139
7.1 Understanding Resource Service Extensions 139
7.2 Creating a Resource Service Extension 139
7.2.1 Guidelines for Writing Resource Service Extensions 139
7.2.2 The Resource Extension Interface 140
7.2.3 Returning JSON Documents 141
7.2.4 Reporting Errors 142
7.2.5 Context Map Keys 142
7.2.6 Controlling Transaction Mode 143
7.3 Installing a Resource Service Extension 143
7.3.1 Installing the Extension XQuery Module 144
7.3.2 Installing Dependent XQuery Library Modules 145
7.3.2.1 Installing Into the Modules Database 145
7.3.2.2 Installing Into the Modules Directory 146
7.3.2.3 Maintenance of Dependent Libraries 147
7.4 Using a Resource Service Extension 147
7.5 Discovering Resource Service Extensions 147
7.6 Retrieving the Implementation of a Resource Service Extension 148
7.7 Example Resource Service Extension 149
7.7.1 Overview 149
7.7.2 Installing the Example Extension 149
7.7.3 Using the Example Extension 150
7.7.4 Example Extension Implementation 151

8.0 Migrating REST Applications 153

8.1 Before You Begin 153
8.2 Upgrading Your Application to a New REST API Instance 153
8.2.1 Upgrading an Application Builder Application 154
8.2.2 Upgrading Other Applications 155
8.3 Migrating the Content Database 156
8.4 Migrating the REST API Instance 158

9.0 Appendix A: JSON Query Options Reference 160
9.1 How to Use This Reference 161
9.2 Query Options 161
9.3 attribute-descriptor 167
9.4 bucket 168
9.5 collection-constraint 169
9.6 computed-bucket 170
9.7 constraint 172
9.8 custom-constraint 174
9.9 element-descriptor 175
9.10 extract-metadata 176
9.11 field-descriptor 178
9.12 geo-attr-pair-constraint 178
9.13 geo-elem-constraint 180
9.14 geo-elem-pair-constraint 181
9.15 grammar 181
9.16 heatmap-descriptor 183
9.17 joiner 183
9.18 operator 185
9.19 path-expression 187
9.20 properties-constraint 188
9.21 range-constraint 188
9.22 sort-order 191
9.23 starter 193
9.24 suggestion-source 195
9.25 term-definition 196
9.26 transform-results 198
9.27 value-constraint 200
9.28 values-or-tuples 201
9.29 word-constraint 203
9.30 word-lexicon 204
9.31 facet-options 205
9.32 term-options 206
9.33 values-options 207
9.34 geo-options 208

10.0 Appendix B: XML Query Options Reference 210
10.1 <options> 210

10.2 <additional-query> 211
10.3 <concurrency-level> 211
10.4 <constraint> 211
10.4.1 <value> 212
10.4.2 <word> 212
10.4.3 <collection> 213
10.4.4 <range> 213
10.4.5 <element-query> 215
10.4.6 <properties> 215
10.4.7 <geo-*> 215
10.4.8 <custom> 216
10.4.9 <term-option> 217
10.4.10<facet-option> 218
10.5 <debug> 219
10.6 <default-suggestion-source> 219
10.6.1 <collection> 219
10.6.2 <range> 219
10.6.3 <word> 220
10.7 <extract-metadata> 220
10.8 <forest> 220
10.9 <fragment-scope> 220
10.10 <grammar> 221
10.10.1<implicit> 222
10.10.2<joiner> 222
10.10.3<starter> 222
10.11 <operator> 223
10.12 <page-length> 224
10.13 <quality-weight> 224
10.14 <return-constraints> 224
10.15 <return-facets> 224
10.16 <return-metrics> 224
10.17 <return-plan> 224
10.18 <return-qtext> 224
10.19 <return-query> 224
10.20 <return-results> 225
10.21 <return-similar> 225
10.22 <search-option> 225
10.23 <searchable-expression> 225
10.24 <sort-order> 225
10.25 <suggestion-source> 226
10.26 <term> 227
10.27 <transform-results> 228
10.28 <tuples> 229
10.29 <values> 230

11.0 Technical Support 231

Combined Product Notices 232
COPYRIGHT 232
TRADEMARK NOTICE 244

1.0 Introduction to the MarkLogic REST API

The *MarkLogic REST 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.

- Getting Started with the MarkLogic REST API
- Overview of MarkLogic REST API Services
- Security Requirements
- Terms and Definitions
- Understanding REST Resources
- Understanding the Example Commands
- Controlling Input Content Type
- Controlling Output Content Type
- Error Reporting

1.1 Getting Started with the MarkLogic REST API

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

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

1.1.1 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 may use any tool 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 19.

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."
  }
}
```

1.1.2 Create a REST API Instance

A *MarkLogic REST API* instance is an HTTP App Server configured with special code installed to service HTTP requests against the API. The /`rest-apis` service on port 8002 allows you to set up, query, and tear down REST API instances. The /`rest-apis` service can optionally create the content database when setting up an instance.

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

This section covers creating an instance using HTTP. To use Information Studio instead, see Creating a REST API Instance in the *Information Studio Developer's Guide*.

Follow this procedure to create a "rest-example" database with one forest and a "rest-example" HTTP App Server attached to the database.

1. Create the instance and database by sending a POST request to `/rest-apis` on port 8002:

```
$ curl --anyauth --user user:password -X POST \
    -d'{"rest-api":{"name":"rest-example"}}' \
    -H "Content-type: application/json" \
    http://localhost:8002/v1/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 a forest named "rest-example-1", 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>app-server-name</name>
</rest-api>
```

1.1.3 Load Documents Into the Database

This procedure loads sample content into the "rest-example" database 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 `rest-example` database:

1. Navigate to the directory containing the sample documents you created in "Preparation" on page 8.

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' \
       'http://localhost:8003/v1/documents?uri=/xml/one.xml&format=xml'
   ```

   The URL tells the `/documents` service to create an XML document (`format=xml`) 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' \
     'http://localhost:8003/v1/documents?uri=/json/two.json&format=json'
   ```

4. Optionally, use Query Console to explore the `rest-example` 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 35.

1.1.4 Search the Database

The *MarkLogic REST API* provides several query services. This procedure uses the `search` service to perform a simple string query search of the "rest-examples" database, finding documents containing "caesar". For details, see "Using and Configuring Query Features" on page 61.

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:8003/v1/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 adding `format=json`:

```
$ curl --anyauth --user user:password \
    'http://localhost:8004/v1/search?format=json&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, to search by JSON key and value or 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 61.

1.1.5 Tear Down the REST API Instance

This procedure uses the `rest-apis` service on port 8002 to remove the *MarkLogic REST API* instance created in "Create a REST API Instance" on page 9. 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 the "rest-examples" instance and associated content and modules databases:

1. Execute the following shell command to remove the instance and database:

```
$ curl --anyauth --user user:password -X DELETE \
  'http://localhost:8002/v1/rest-apis/rest-example?include=content&inc
lude=modules'
```

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

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

1.2 Overview of *MarkLogic REST API* Services

This sections gives a brief overview of the services provided by the *MarkLogic REST 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 *MarkLogic REST API*. For details, see "Creating an Instance" on page 24.

In addition to these services, you can create resource service extensions using XQuery. For details, see "Extending the REST API" on page 139.

Service	Description	More Information
`/rest-apis`	*MarkLogic REST API* instance administration, including creating and tearing down instances.	"Administering MarkLogic REST API Instances" on page 24
`/documents`	Document manipulation, including creating, updating and deleting documents and meta data.	"Manipulating Documents" on page 35
`/search`	Search content and metadata using string and structured queries.	"Using and Configuring Query Features" on page 61
`/keyvalue`	Search by XML content by element, element attribute, and field name and value. Search JSON content by JSON key and value.	"Using and Configuring Query Features" on page 61
`/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 61
`/transactions`	Support for evaluating REST requests in multi-statement transactions. Create, commit, rollback, and monitor transactions.	"Managing Transactions" on page 121
`/resources`	Access to user-defined resource service extensions.	"Extending the REST API" on page 139
`/config/query`	Create, modify, delete, and read configuration options used to control queries made using /search, /keyvalue, and /values.	"Configuring Query Options" on page 92
`/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 97

Service	Description	More Information
`/config/properties`	Configure instance-wide properties, such as enabling debug output and setting the content type of error messages.	“Configuring Instance Properties” on page 30
`/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.	“Working With Content Transformations” on page 127
`/config/resources`	Manage resource service extensions.	“Extending the REST API” on page 139
`/config/namespaces`	Create and manage instance-wide namespace prefix bindings. Such bindings allow you to use namespace prefixes in queries.	“Using Namespace Bindings” on page 102

1.3 Security Requirements

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

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

Role	Description
`rest-reader`	Enables read operations through the *MarkLogic REST 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 *MarkLogic REST 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 *MarkLogic REST 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.

If no permissions are explicitly set, documents you create with the *MarkLogic REST API* have a read permission for the rest-reader role and an update permission for the rest-writer role.

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

1.4 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 *MarkLogic REST 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 *MarkLogic REST 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 17.
rewriter	An XQuery module that interprets the URL of an incoming HTTP request and rewrites it to an internal URL that services the request.

Term	Definition
REST API instance	An instantiation of the *MarkLogic REST 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 MarkLogic REST API Instances” on page 24.
extension	An user-defined XQuery module that implements additional resource services that are made available through the *MarkLogic REST API*. For details, see “Extending the REST API” on page 139.
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 62.
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 62 and Using Structured Search as an Alternate to cts:query 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 70 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.

1.5 Understanding REST Resources

This section covers the basic structure of a *MarkLogic REST 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

1.5.1 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 *MarkLogic REST API* instance running on that port. A *MarkLogic REST API* instance is served by an HTTP App Server. For details, see "Creating an Instance" on page 24.

A resource address always includes the API version in URL. For details, see "Specifying the REST API Version" on page 18.

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 18.

1.5.2 Specifying the REST API Version

To guarantee stable behavior of the *MarkLogic REST API* as new versions are released, each resource address in the *MarkLogic REST API* includes a version number. The examples in this chapter show the version as `v1` or simply *version*. The version has the format:

```
v#
```

Where # is the version number. For example, in the initial version of the API:

```
http://localhost:8003/v1/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.

1.5.3 Specifying Parameters in a Resource Address

Resource addresses 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....
```

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

1.6 Understanding the Example Commands

The examples in this guide use the curl command line tool to send HTTP requests that exercise the *MarkLogic REST 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 17.

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

1.6.1 Introduction to the curl Tool

`curl` is a command line tool for sending HTTP requests. 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. 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 15.
-X *http_method*	The type of HTTP request (GET, PUT, POST, DELETE) that `curl` should send. If `-X` is not given, GET is assumed.
-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.

Option	Description
-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.

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:8003/v1/config/query/my-options
```

1.6.2 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.

1.7 Controlling Input Content Type

Many services of the *MarkLogic REST API* accept input in either XML or JSON. Unless otherwise noted, where content type control is supported, you can specify the content type using either the `format` request parameter or the HTTP Content-type header. If both the `format` parameter and the Content-type header are present, the format parameter takes precedence.

For POST methods that have data in both the request body and the response body, the `format` parameter overrides both Content-type and Accept headers and applies to both request and response data. If you want to use different content types for input and output on such methods, use the HTTP headers only; do not set `format`. For an example, see the reference documentation for `POST /v1/search`.

For JSON input, set the format request parameter to “json” or set the HTTP Content-type header to `application/json`. For example:

```
$ curl -X POST -d @'my-body.json' --anyauth --user user:password \
    http://localhost:8003/v1/config/query/my-opts?format=json

$ curl -X POST -d @'my-body.json' --anyauth --user user:password \
    -H "Content-type: application/json" \
    http://localhost:8003/v1/config/query/my-opts
```

For XML input, set the `format` request parameter to “xml” or set the HTTP Content-type header to `application/xml`. For example:

```
$ curl -X POST -d @'my-body.xml' --anyauth --user user:password \
    http://localhost:8003/v1/config/query/my-opts?format=xml

$ curl -X POST -d @'my-body.json' --anyauth --user user:password \
    -H "Content-type: application/xml" \
    http://localhost:8003/v1/config/query/my-opts
```

1.8 Controlling Output Content Type

Many services of the *MarkLogic REST API* that return metadata or configuration information allow you to select JSON or XML output. XML output is the default unless otherwise specified in the *MarkLogic REST API Reference* for a specific operation.

Where output content type control is supported, use the `format` request parameter or the HTTP Accept header to specify the response content type. If both the parameter and the Accept header are set, the `format` parameter takes precendence.

For POST methods that have data in both the request body and the response body, the `format` parameter overrides both Content-type and Accept headers and applies to both request and response data. If you want to use different content types for input and output on such methods, use the HTTP headers only; do not set `format`. For an example, see the reference documentation for `POST /v1/search`.

To request JSON output, set the `format` request parameter to “json” or set the HTTP `Accept` header to `application/json`. For example:

```
$ curl -X GET --anyauth --user user:password
http://localhost:8002/v1/rest-apis?format=json

$ curl -X GET --anyauth --user user:password -H "Accept:
application/json" http://localhost:8002/v1/rest-apis
```

To explicitly request XML output, set format to `xml` or set the HTTP `Accept` header to `application/xml`. For example:

```
$ curl -X GET --anyauth --user user:password
http://localhost:8002/v1/rest-apis?format=xml

$ curl -X GET --anyauth --user user:password -H "Accept:
application/xml" http://localhost:8002/v1/rest-apis
```

1.9 Error Reporting

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

If a request to a *MarkLogic REST API* instance fails, an error status code is returned and additional error detail is provided in the response body. You can configure your instance to return error details in either XML or JSON using the `error-format` configuration property. XML is the default content type. For details, see "Configuring Instance Properties" on page 30.

The following example shows error output in XML for a request containing 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

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

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

{
  "error": {
    "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.

XQuery and XSLT transforms and XQuery extensions you write should report errors using `RESTAPI-EXTNERR`, as described in “Reporting Errors” on page 129.

2.0 Administering *MarkLogic REST API* Instances

To use the *MarkLogic REST 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.

- Creating an Instance
- Removing an Instance
- Retrieving Configuration Information
- Configuring Instance Properties

2.1 Creating an Instance

You can create an instance of the *MarkLogic REST API* using Information Studio or by making an HTTP request. This section covers creating an instance using HTTP. To use Information Studio, see Creating a REST API Instance in the *Information Studio Developer's Guide*.

> **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.

To create a new instance using HTTP, send a POST 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 HTTP status 201 if the instance is successfully created.

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.

The configuration data must specify a name for the instance and may also specify a group name, database name, modules database name, and port, as described by the table below:

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 one forest.
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.

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.

The following example creates an instance using XML configuration information. The result is an HTTP App Server named “RESTstop” on port 8020 that serves data from the “Documents database. The modules database is pre-populated with the modules that support the *MarkLogic REST API*.

```
$ 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/v1/rest-apis
```

The following example creates the same instance using a JSON configuration file:

```
$ cat config.json
{
  "rest-api": {
    "name": "RESTstop",
    "database": "Documents",
    "port": "8020"
  }
}

# Windows users, see Modifying the Example Commands for Windows
$ curl -X POST --anyauth --user user:password -d @"./config.json" -H
"Content-type: application/json" http://localhost:8002/v1/rest-apis
```

2.2 Removing an Instance

To remove an instance of the *MarkLogic REST API*, send a DELETE request to the /rest-apis service on port 8002. To remove the instance but leave the content database intact, send the request with a URL of the form:

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

`instance-name` is the name supplied when the instance is created, which is also the name of the App Server associated with the instance. 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 "RESTstop" instance, 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/v1/rest-apis/RESTstop?include=content&include
=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/v1/rest-apis/RESTstop?group=stage
```

2.3 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

2.3.1 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. For details on setting the output format, see “Controlling Output Content Type” on page 21.

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 \
    http://localhost:8002/v1/rest-apis?format=json
```

The response includes configuration details about 2 instances, named RESTstop and OscarsREST.

```
{
  "rest-apis":[
    {
      "name": "RESTstop",
      "group": "Default",
      "database": "bill",
      "modules-database": "RESTstop-modules",
      "port": "8003"
    },
    {
      "name":"OscarsREST",
      "group": "Default",
      "database":"Oscars",
      "modules-database": "OscarsREST-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>RESTstop</rapi:name>
    <rapi:group>Default</rapi:group>
    <rapi:database>bill</rapi:database>
    <rapi:modules-database>RESTstop-modules</rapi:modules-database>
    <rapi:port>8003</rapi:port>
  </rapi:rest-api>
  <rapi:rest-api>
    <rapi:name>OscarsREST</rapi:name>
    <rapi:group>Default</rapi:group>
    <rapi:database>Oscars</rapi:database>
    <rapi:modules-database>OscaresREST-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 28 and “Retrieving Instance Configuration by Instance Name” on page 29.

2.3.2 Retrieving Instance Configuration by Content Database

To retrieve configuration details 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. For details on setting the output format, see “Controlling Output Content Type” on page 21.

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

```
# Windows users, see Modifying the Example Commands for Windows
$ curl -X GET --anyauth --user user:password \
    'http://localhost:8002/v1/rest-apis?format=json&database=bill'
```

The JSON output in the response looks like the following (whitespace added to improve readability):

```
{
  "rest-apis": [
    {
      "name": "RESTstop",
      "group": "Default",
      "database": "bill",
      "modules-database": "RESTstop-modules",
      "port": "8003"
    }
  ]
}
```

The equivalent XML output is shown below:

```
<rapi:rest-apis xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:rest-api>
    <rapi:name>RESTstop</rapi:name>
    <rapi:group>Default</rapi:group>
    <rapi:database>bill</rapi:database>
    <rapi:modules-database>RESTstop-modules</rapi:modules-database>
    <rapi:port>8003</rapi:port>
  </rapi:rest-api>
</rapi:rest-apis>
```

2.3.3 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. For details on setting the output format, see “Controlling Output Content Type” on page 21.

The following example requests information in JSON about the “RESTstop” instance:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl -X GET --anyauth --user user:password \
    http://localhost:8002/v1/rest-apis/RESTstop?format=json
```

The response body contains JSON output of the following form:

```
{
  "name": "RESTstop",
  "group": "Default",
  "database": "bill",
  "modules-database": "RESTstop-modules",
  "port": "8003"
}
```

Sending the same request, but specifying XML output, the response body contains:

```
<rapi:rest-api xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:name>RESTstop</rapi:name>
  <rapi:group>Default</rapi:group>
  <rapi:database>bill</rapi:database>
  <rapi:modules-database>RESTstop-modules</rapi:modules-database>
  <rapi:port>8003</rapi:port>
</rapi:rest-api>
```

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.

2.4 Configuring Instance Properties

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

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

2.4.1 Instance Configuration Properties

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

Name	Description
content-versions	Controls the availability and behavior of conditional GET, PUT and DELETE on `/documents`. Allowed values: `required`, `optional`, `none`. Default: `none`. For details, see “Using Optimistic Locking to Update Documents” on page 49.

Name	Description
debug	Turn debug logging on and off. Allowed values: `true`, `false`. Default: `false`.
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 127.
error-format	The content type of errors returned by the *MarkLogic REST API*. Allowed values: `xml`, `json`. Default: `xml`.
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`.

2.4.2 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 30.

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

You can retrieve property settings as XML or JSON, using either the format parameter or HTTP Accept header, as described in "Controlling Output Content Type" on page 21. 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:8003/v1/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-out/>
  <rapi:error-format>xml</rapi:error-format>
  <rapi:validate-options>true</rapi:validate-options>
</rapi:properties>
```

The following example retrieves the same information as JSON:

```
$ curl --anyauth --user user:password \
    http://localhost:8003/v1/config/properties?format=json

{
  "content-versions": "none",
  "validate-options": true,
  "document-transform-out": "",
  "debug": false,
  "error-format": "xml"
}
```

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

```
$ curl --anyauth --user user:password \
    http://localhost:8003/v1/config/properties/debug?format=json

{ "debug" : false }
```

The equivalent XML output is shown below:

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

2.4.3 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 30.

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

Property settings can be specified in either XML or JSON, using the format parameter or the Conent-type headers to indicate the content type. For details, see “Controlling Input Content Type” on page 20.

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 admin:adm1n -X PUT -d@"./props.xml" \
    http://localhost:8003/v1/config/properties?format=xml
```

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 admin:adm1n -X PUT -d@"./debug-prop.xml" \
    http://localhost:8003/v1/config/properties/debug?format=xml
```

The equivalent JSON input is:

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

2.4.4 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:8003/v1/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 30.

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

```
$ curl --anyauth --user user:password -X DELETE \
    http://localhost:8003/v1/config/properties/debug
```

3.0 Manipulating Documents

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

- Summary of the /documents Service
- Loading Content into the Database
- Retrieving Documents from the Database
- 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 Metadata

3.1 Summary of the /documents Service

Use the `/documents` service to create, read, update, and delete document content and metadata. The following table summarizes the supported operations:

Operation	Method	Description
Create/Update	PUT	Create or update content or metadata.
Retrieve	GET	Retrieve content and/or metadata.
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 retrieval
- Conditional document insertion using optimistic locking
- Conditional reads based on content versioning

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

3.2 Loading Content into the Database

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
- Loading JSON Documents Into the Database
- Controlling Access to Documents Created with the REST API
- Transforming Content During Ingestion

3.2.1 Loading Content

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.

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

Documents you create with the MarkLogic REST API have a read permission for the rest-reader role and update permission for the rest-writer role. To restrict access, use custom roles. For details, see “Controlling Access to Documents Created with the REST API” on page 42.

If no permissions are explicitly set, documents you create with the MarkLogic REST API have a read permission for the `rest-reader` role and an update permission for the `rest-writer` role.

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:8003/v1/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 you installation.

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 39.

For additional information about working with JSON documents, see “Loading JSON Documents Into the Database” on page 41.

3.2.2 Adding Metadata

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 58.

> **Note:** You cannot supply metadata via request parameters when there is no document content. 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. For format details, see “Working with Metadata” on page 57.

3. Specify the metadata format in the HTTP `Content-type` header or the `format` parameter. You may supply either JSON or XML metadata.

Metadata 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.

Metadata can be supplied as either XML or JSON. Use the `format` parameter or the `Content-type` header to specify the format of the metadata. If both `format` and `Content-type` are given, `format` takes precedence. If neither `format` nor `Content-type` is specified, XML is assumed. For details on formats, see “Working with Metadata” on page 57.

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.

3.2.2.1 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:8003/v1/documents?uri=/xml/example.xml&category=
collections'
```

3.2.2.2 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:8003/v1/documents?uri=/xml/example.xml&category=
collections&category=properties'
```

3.2.2.3 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 \
    'http://localhost:8003/v1/documents?uri=/xml/example.xml&category=
collections&category=properties&format=json'
```

The example uses `format=json` to communicate the metadata format to MarkLogic Server. The format can also be specified through the HTTP `Content-type` header.

3.2.3 Loading Content and Adding Metadata in the Same Request

You can update content and metadata in a single request using two methods:

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

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`. For example, to set the 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.

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:8003/v1/documents?uri=/images/critter.jpg&collec
tion=animals&prop:species="canus lupus"'
```

You can also create a block of metadata as JSON or XML and pass it in the request body with the content. See "Loading Content and Metadata Using a Multipart Message" on page 40.

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 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 39.

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 58.

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 57.

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:8003/v1/documents?uri=/xml/box.xml&category=coll
ections&category=content'
```

You can also pass metadata in URL parameters instead of in the request body. See "Loading Content and Metadata Using Request Parameters" on page 39.

3.2.4 Loading JSON Documents Into the Database

You can insert JSON documents into the database using `/documents`. To insert JSON content, simply specify a Content-type of `application/json` when inserting the document. For example:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -T ./my.json \
    -H "Content-type: application/json" \
    http://localhost:8003/v1/documents?uri=/xml/example.xml
```

Internally, MarkLogic Server converts your JSON document into an XML representation so you can use the powerful query capabilities of MarkLogic Server on your JSON content. Usually, you do not need to be aware of this representation, though you can examine it using Query Console's Explore feature. You might need to examine the internal representation if you need to define a range index on JSON content. For details, see "Creating Indexes on JSON Keys" on page 80.

> **Note:** Element names in the XML representation are in the namespace "http://marklogic.com/xdmp/json/basic". You must use this namespace when defining indexes on keys in JSON documents.

If you use content transformations, the transformations are applied to the JSON representation. For details, see "Working With Content Transformations" on page 127.

For more information on JSON support in MarkLogic Server, see Working With JSON in the *Application Developer's Guide*.

3.2.5 Controlling Access to Documents Created with the REST API

By default, documents you create with the MarkLogic REST 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 permission controls access to a document whether it is through the REST API or through other code running on MarkLogic Server.

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 GroupA role. Use the privileges, not the base roles. For example, assign these privileges to the role:

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

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

4. Assign the GroupA role to the appropriate users.

5. 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. Similaly, users with the GroupB role can create documents and read or update them, but users with the GroupB 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 *Understanding and Using Security Guide*.

3.2.6 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 127.

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/v1/documents?...&transform=name&trans:arg=val
ue
```

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:8003/v1/documents?uri=/doc/theDoc.xml&transform=
example&trans:reviewer=me'
```

For a complete example, see “XQuery Example: Adding an Attribute During Ingestion” on page 134 or “XSLT Example: Adding an Attribute During Ingestion” on page 137.

3.3 Retrieving Documents from the Database

To retrieve documents 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

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 127.

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 58.

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 format (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 57.

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.

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 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 58.

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 57.
- 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:8004/v1/documents?uri=/xml/box.xml&category=meta
data&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>
--BOUNDARY
Content-Type: text/xml
Content-Length: 128

<?xml version="1.0" encoding="UTF-8"?>
```

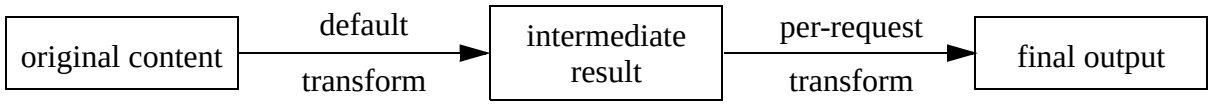
```
<data><mybox>This is my box. There are many like it, but this one is
mine.</mybox></data>
--BOUNDARY--
```

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 44 and "Working With Content Transformations" on page 127.

3.3.4 Transforming Content During Retrieval

You can apply custom transforms to a document before returning it to the requestor. 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 127.

You can configure a default transform that is automatically applied whenever a document is retried. 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 MarkLogic REST API instance. Instance-wide parameters are set using `/config/properties`. For details, see "Configuring Instance Properties" on page 30.

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/v1/documents?...&transform=name&trans:arg=val
ue
```

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:8003/v1/documents?uri=/doc/theDoc.xml&transform=
example&trans:reviewer=me'
```

3.4 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.
- Determining 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:8004/v1/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:8004/v1/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
```

3.5 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

3.5.1 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.

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
```

Where `category` can appear multiple times, with the values described in “Metadata Categories” on page 58.

Deleting permissions resets the document permissions to the default permissions for the current user. Deleting 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 56.

3.5.2 Removing Multiple Documents

You can remove all documents in a collection by sending a DELETE request with a URL of the following form:

```
http://host:port/version/search?collection=collection_name
```

Similarly, you can remove all documents in a directory by sending 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 documents 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 56.

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:8003/v1/search?directory=/plays/
...
HTTP/1.1 204 Updated
Server: MarkLogic
Content-Length: 0
Connection: close
```

3.5.3 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:8003/v1/search
```

3.6 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

- Using Optimistic Locking
- Example: Updating a Document Using Optimistic Locking

3.6.1 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.

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 MarkLogic REST 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 through your REST API instance. The version id is returned in the ETag header when you read a document, and you can pass it back in an If-Match header during an update or delete operation to test for changes prior to commit. For details, see “Using Optimistic Locking” on page 51.

> **Note:** Content versioning in the MarkLogic REST 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.

Enable content versioning using the `content-versions` instance configuration property. For details, see “Enabling Optimistic Locking” on page 50.

The MarkLogic REST API also supports multi-statement transactions. For details, see “Managing Transactions” on page 121.

3.6.2 Enabling Optimistic Locking

Enable optimistic locking using the `content-versions` instance configuration property, as described in “Configuring Instance Properties” on page 30.

The `content-versions` property can be set to `none` (the default), `optional`, or `required`. Set the property to `required` if you want every document update or delete operation to use optimistic locking. Set the property to `optional` to allow selective use of optimistic locking.

The table below describes how each setting for this property affects document operations.

Setting	Effect
none	This is the default setting. If you make a PUT or DELETE request to /documents and no document exists with the given URI, the operation succeeds. If an HTTP If-Match header is present, it is ignored.
optional	If you make a PUT or DELETE request to /documents without an If-Match header and the document does not exist, the operation succeeds. If an If-Match header is present, the operation fails if the document exists and the current version id does not match the version in the header.
required	If you make a PUT or DELETE request to /documents without an If-Match header and the document does not exist, the operation succeeds; if the document exists, the operation fails and returns 403. If an If-Match header is present, the operation fails if the document exists and the current version id does not match the version in the header.

3.6.3 Using Optimistic Locking

When content versioning is enabled, MarkLogic Server returns a document version id in the ETag header when you make a GET or HEAD request to /documents. For example:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -i -X HEAD \
    -H "Accept: application/xml" \
    http://localhost:8003/v1/documents?uri=/docs/example.xml
...
HTTP/1.1 200 Document Retrieved
Content-type: application/xml
ETag: "13473172834878540"
Server: MarkLogic
Connection: close
```

Pass the version id in the If-Match header of a PUT or DELETE request to /documents to have MarkLogic Server check for changes in the version id before committing the update or delete operation. For example:

```
# 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:8003/v1/documents?uri=/docs/example.xml
```

Follow this procedure to use optimistic locking to update a document. For a complete example, see “Example: Updating a Document Using Optimistic Locking” on page 52.

1. If content versioning is not already enabled, set the `content-versions` instance configuration property to `optional` or `required`; see "Enabling Optimistic Locking" on page 50.

2. Retrieve the document by making a GET request to `/documents`. The version id is included in the ETag HTTP header.

3. Update your local copy of the document.

4. Update the document in the database by sending a PUT request to `/documents` with the version id from Step 2 in the If-Match HTTP header.

   If the document has not been modified since Step 2, the update succeeds. If the document has been modified since Step 2, the update fails.

For a delete operation, follow a similar procedure, passing the version id in the If-Match header of a DELETE request to the `/documents` service. You can get the version id from a previous GET, HEAD, or PUT request.

3.6.4 Example: Updating a Document Using Optimistic Locking

The following example demonstrates using optimistic locking to attempting an document update. Both a successful and a failed update are shown.

1. If content versioning is not already enabled, enable it by setting `content-versions` to `optional`:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    -d'{"content-versions":"optional"}' \
    http://localhost:8003/v1/config/properties?format=json
```

2. Insert the 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:8003/v1/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:8003/v1/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 upate the document by new contents with the version id from Step 3 in the If-Match header. Since the document has not changed, 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:8003/v1/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:8003/v1/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>
```

3.7 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

3.7.1 Enabling Content Versioning

Enable content versioning using the `content-versions` instance configuration property, as described in “Configuring Instance Properties” on page 30.

The `content-versions` property can be set to `none` (the default), `optional`, or `required`. When the property is set to `optional` or `required`, 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 50.

> **Note:** Content versioning in the MarkLogic REST 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.

3.7.2 Using Content Versioning for Cache Refreshing

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.

Follow this procedure to use this feature. For a complete example, see “Example: Refreshing a Cached Document” on page 55.

1. If content versioning is not already enabled, set the `content-versions` instance configuration property to `optional` or `required`; see “Enabling Content Versioning” on page 54.

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 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 the If-None-Match header, the document is returned, along with the new version id in the ETag response header.

3.7.3 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 where the local cache is out of date.

1. If content versioning is not already enabled, enable it by setting `content-versions` to `optional`:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT \
    -d'{"content-versions":"optional"}' \
    http://localhost:8003/v1/config/properties?format=json
```

2. Insert the 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:8003/v1/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:8003/v1/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:8003/v1/documents?uri=/docs/example.xml
...
HTTP/1.1 304 Content Version Not Modified
ETag: "13473769780393030"
Server: MarkLogic
Content-Length: 0
Connection: close
```

5. Modify the document, which changes the version id.

```
$ curl --anyauth --user user:password -i -X PUT -d"<modified-data/>"  \
    -H "Content-type: application/xml" \
    http://localhost:8003/v1/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:8003/v1/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/>
```

3.8 Working with Binary Documents

This section covers the following topics:

- Types of Binary Documents
- Streaming Binary Content

3.8.1 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 MarkLogic REST API.

Large binary documents can be streamed out of the database using Range requests. For details, see "Streaming Binary Content" on page 57.

3.8.2 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:8003/v1/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
```

3.9 Working with Metadata

The `/documents` service supports inserting, updating, and retrieving document metadata. Metadata is the properties, collections, permissions, and quality of content.

Metadata manipulation is most often exposed by the MarkLogic REST 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, 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

3.9.1 Metadata Categories

Where the MarkLogic REST API accepts specification of metadata categories, the following categories are recognized:

- `collections`
- `permissions`
- `properties`
- `quality`
- `metadata`

The `metadata` category is shorthand for all the other categories. That is, `metadata` includes collections, permissions, properties, 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.

3.9.2 XML Metadata Format

Metadata contains information about document collections, permissions, properties, and quality. 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/>` or property elements. There can be only one `<quality/>` element. The element name and contents of each 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>
```

The following example shows a document in 2 collections, with one permission and 2 properties.

```
<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>
```

3.9.3 JSON Metadata Format

Metadata contains information about document collections, permissions, properties, and quality. A block of metadata can contain multiple collections, permissions and properties, but only 1 quality. The structure of each property depends on the property contents.

```
{
  "collections" : [ string ],
  "permissions" :
    {
      "role-name" : string,
      "capabilities" : [ string ]
    }
  ],
  "properties" : {
    property-name : property-value
  },
  "quality" : integer
}
```

The following example shows a document in 2 collections, with one permission and 2 properties.

```
{
  "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
}
```

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
- Analyzing Lexicons and Range Indexes With Aggregate Functions
- Specifying Dynamic Query Options with Combined Query
- Configuring Query Options
- Using Namespace Bindings
- Generating Search Facets
- Paginating Results
- Customizing Search Snippets

4.1 Query Feature Overview

The *MarkLogic REST 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 92.

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 102.

3. Use `/search`, `/keyvalue`, or `/values` to perform a query.

   a. Depending on the service, the query may be expressed using 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 93.

   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.

The table below gives a brief summary of each query related service.

Service	Description
`/search`	Use string and/or structured queries to search documents and metadata. Queries may be expressed in XML or JSON. For details, see "Querying Documents and Metadata" on page 62.
`/keyvalue`	Use key-value pair queries to search XML content and metadata, and JSON content. For example, search by element, element attribute, or JSON key values. Dynamic query options are not supported by this service. For details, see "Querying Documents and Metadata" on page 62.
`/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 70.
`/config/query`	Use XML or JSON to configure persistent query options for use with `/search`, `/keyvalue`, and `/values`. For details, see "Configuring Query Options" on page 92.
`/config/namespaces`	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 102.

The `/search`, `/keyvalue`, and `/values` services use query options to control and configure queries and search results. For details on options syntax, see:

- "Appendix A: JSON Query Options Reference" on page 160
- "Appendix B: XML Query Options Reference" on page 210

For more information on search concepts, see the Search Developer's Guide.

4.2 Querying Documents and Metadata

This section describes how to use the `/search` and `/keyvalue` services to search documents and metadata.

- Constraining a Query by Collection or Directory
- Searching With String Queries
- Searching With Structured Queries

- Searching By JSON Key and Value
- Searching By Element or Attribute Name and Value
- Debugging /search Queries With Logging

4.2.1 Constraining a Query by Collection or Directory

Use the `collection` and `directory` request parameters to the `/search` and `/keyvalue` services 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:8003/v1/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:8003/v1/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 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.

You can include both a string query and a structured query in the same request. In this case, the two queries are AND'd together. For more information, see "Searching With Structured Queries" on page 64 and "Specifying Dynamic Query Options with Combined Query" on page 87.

You can request search results in XML or JSON. To control the output format, use the `format` request parameter or `Accept` headers, as described in "Controlling Output Content Type" on page 21.

For details on the structure of search results, see “Appendix A: JSON Query Options Reference” on page 160 or “Appendix B: XML Query Options Reference” on page 210.

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 Using Structured Search as an Alternate to cts:query 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 the 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 63 and “Specifying Dynamic Query Options with Combined Query” on page 87.

You can request search results in XML or JSON. To control the output content type on GET, use the `format` request parameter or `Accept` headers. To control the input and output content type on POST, use the `format` parameter or the `Accept` and `Content-type` headers. For details, see “Controlling Output Content Type” on page 21.

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' \
  'http://localhost:8003/v1/search?q=knew&format=json'
```

4.2.4 Searching By JSON Key and Value

When you store JSON documents in the database using the `/documents` service of the *MarkLogic REST API*, you can search based on the values of keys in the JSON content using the `/keyvalue` service.

> **Note:** Efficient key-value searches require an element range index on the key. For details, see “Creating Indexes on JSON Keys” on page 80.

To search by key value, send a GET request to the `/keyvalue` service with a URL of the form:

```
http://host:port/version/keyvalue?key=the_key&value=the_value
```

Where `the_key` is the name of the key to match against `the_value`. For example, the following search matches all JSON documents containing a key called “species” with a value of “canis lupus”:

```
http://localhost:2003/v1/keyvalue?key=species&value=canis%20lupus
```

Only JSON documents are considered in the search when you use the `key` parameter. To query XML content, see “Searching By Element or Attribute Name and Value” on page 67.

You can request search results in XML or JSON. To control the output format, use the `format` request parameter or `Accept` headers, as described in “Controlling Output Content Type” on page 21.

For a complete list of parameters available with the `/keyvalue` service, see “GET /v1/keyvalue” in the *REST Resources API*.

The following example searches for occurrences of the key “SPEAKER” with the value “BERNARDO”. First, insert a JSON document into the database:

```
$ cat hamlet.json
{
  "TITLE":"Hamlet",
  "ACT":{
    "TITLE":"ACT I",
    "SCENE":{
      "TITLE":"SCENE I. Elsinore. A platform before the castle.",
      "STAGEDIR":"FRANCISCO at his post. Enter to him BENARDO",
```

```
      "SPEECHES":[
        {"SPEECH":{
          "SPEAKER":"BERNARDO",
          "LINE":"Whos there?"
        }},
        {"SPEECH":{
          "SPEAKER":"FRANCISCO",
          "LINE":"Nay, answerme: stand, and unfold yourself."
        }},
        {"SPEECH":{
          "SPEAKER":"BERNARDO",
          "LINE":"Long live the king!"
        }}
      ]
    }
  }
}

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -d @"./hamlet.json" \
  -H 'Content-type: application/json' \
  http://localhost:8003/v1/documents?uri=/example/hamlet.json
```

Then search for occurrences of SPEAKER with the value BERNARDO. The search matches the document just inserted and returns a snippet for the two speeches by Bernardo:

```
$ curl --anyauth --user user:password -X GET \
  'http://localhost:8003/v1/keyvalue?key=SPEAKER&value=BERNARDO'
...
{
  "snippet-format":"snippet",
  "total":1,
  "start":1,
  "page-length":10,
  "results":[{
    "index":1,
    "uri":"/example/hamlet.json",
    "path":"fn:doc(\"/example/hamlet.json\")",
    "score":154880,
    "confidence":0.790972,
    "fitness":0.790972,
    "matches":[
      {"path":"fn:doc(\"/example/hamlet.json\")/*:json/*:ACT/*:SCENE/*
:SPEECHES/*:json[1]/*:SPEECH/*:SPEAKER",
       "match-text":[{"highlight":"BERNARDO"}]},
      {"path":"fn:doc(\"/example/hamlet.json\")/*:json/*:ACT/*:SCENE/*
:SPEECHES/*:json[3]/*:SPEECH/*:SPEAKER",
       "match-text":[{"highlight":"BERNARDO"}]}
    ]
  }],
  "metrics":{
    "query-resolution-time":"PT0.001398S",
    "facet-resolution-time":"PT0.000052S",
```

```
    "snippet-resolution-time":"PT0.001411S",
    "total-time":"PT0.003032S"
  }
}
```

4.2.5 Searching By Element or Attribute Name and Value

The `/keyvalue` service provides a simple interface to match element and element attribute values in XML content. For more sophisticated element and element attribute value searches, use structured queries; see "Searching With Structured Queries" on page 64.

Note: Efficient element and element attribute searches require element and attribute range indexes. For details, see Range Indexes and Lexicons in the *Administrator's Guide*.

To find all XML elements with a specific element name and value, send a GET request to the `/keyvalue` service with a URL of the form:

```
http://host:port/version/keyvalue?element=elem_name&value=elem_value
```

Note: If the element name is defined within a namespace, you must pre-define a namespace binding and use the defined prefix in *elem_name*. For details, see "Using Namespace Bindings" on page 102.

The following example matches documents containing a `<SPEAKER>` element with the value "BERNARDO":

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
   'http://localhost:8003/v1/keyvalue?element=SPEAKER&value=BERNARDO'
```

To find all XML elements with a specific element name that have an attribute with a specific value, send a GET request to the /keyvalue service with a URL of the form:

```
http://host:port/version/keyvalue?element=elem_name&attribute=attr_name&
value=elem_value
```

This example matches documents containing `<nominee>` elements with a `year` attribute value of 1946:

```
$ curl --anyauth --user user:password -X GET \
    'http://localhost:8003/v1/keyvalue?element=nominee&attribute=year&
value=1946'
```

MarkLogic Server can return search results in XML or JSON. To specify the output format, use the `format` request parameter or `Accept` headers, as described in "Controlling Output Content Type" on page 21.

For a complete list of parameters available with the `/keyvalue` service, see "GET /v1/keyvalue" in the *REST Resources API*.

4.2.6 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 30.

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 Using Structured Search as an Alternate to cts:query 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 ocurrences 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:8003/v1/config/properties
```

2. Issue a query to `/search`.

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
    http://localhost:8003/v1/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:8003/v1/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 81.

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 Keys

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 77.

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 93.

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 range 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 77 and "Adding Query Options to a Request" on page 93.

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` parameter, or a structured or combined query in the POST body. For details, see "Using a Query to Constrain Results" on page 76.

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 Output Content Type" on page 21.

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:8003/v1/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, structured query, or both 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:8003/v1/values/speaker?options=index-options&q="
HAMLET"'
```

4.3.2 Finding Value Co-Occurrences in Lexicons

A co-occurence is a set of index or lexicon values occurring in the same document fragment. The *MarkLogic REST API* enables you to query for 2-way co-occurrences. That is, for pairs of values from two 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` range specification for the target lexicons or indexes. For details, see “Defining Queryable Lexicon or Range Index Co-Occurrences” on page 79 and “Adding Query Options to a Request” on page 93.

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 range 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 77 and "Adding Query Options to a Request" on page 93.

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 76.

5. Specify the input (POST only) and output content type (XML or JSON) using the `format` parameter or the HTTP Accept and Content-type headers. For details, see "Controlling Output Content Type" on page 21.

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:

```
$ curl --anyauth --user user:password -X GET \
   http://localhost:8003/v1/values/speaker-scene?options=index-options
...
<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, structured query, or both 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":

```
$ curl --anyauth --user user:password -X GET \
    'http://localhost:8003/v1/values/speaker-scene?options=index-optio
ns&q="HAMLET"'
```

4.3.3 Using a Query to Constrain Results

You can use a string query, structured 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` element or or sub-object of a combined query supplied in the body of a POST request.

Specify a structured query in either XMl or JSON using one of the following:

- The `structuredQuery` request parameter on a GET request.
- The `query` element or sub-object of 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:8003/v1/values/speaker?options=index-options&q=m
oon'
```

For the full example without a query constraint, see "Querying the Values in a Lexicon or Range Index" on page 71.

For details on query syntax, see the following sources:

- Automatic Query Text Parsing and Grammar in the *Search Developer's Guide*

- Searching Using Structured Queries in the *Search Developer's Guide*
- "Specifying Dynamic Query Options with Combined Query" on page 87

4.3.4 Identifying Lexicon and Range Index Values in Query Options

Before you can use the `/values/{name}` service, you must install query options that identify lexicons and range indexes available to queries. 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:

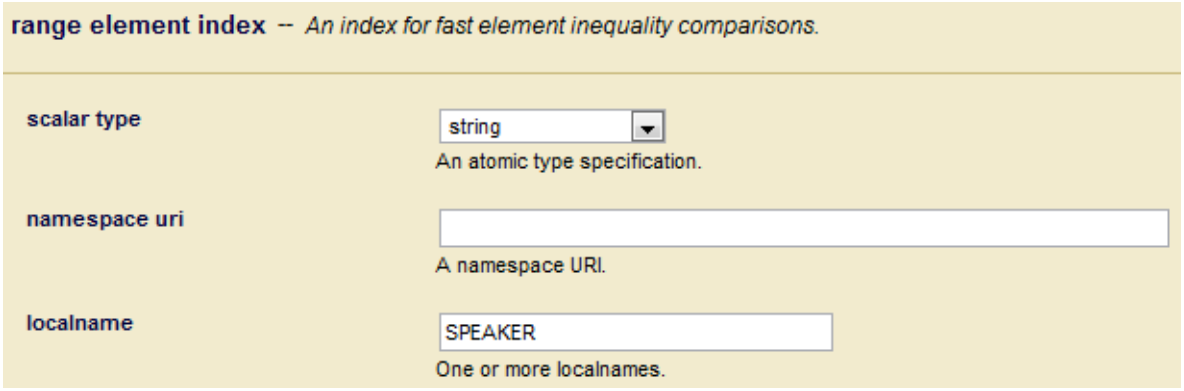
1. Create a lexicon or range index on the database, as described in Range Indexes and Lexicons in the *Administrator's Guide* and

2. Associate a name with the index or lexicon by defining a `<values/>` element 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 93.

For more information on lexicons and range indexes, see Browsing With Lexicons in the *Search Developer's Guide* and "Creating Indexes on JSON Keys" on page 80.

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`, once you install the 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" }`
	`  }`
	`  }`
	`  ]`
	`  }`
	`}`

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:8003/v1/config/query/index-options
```

Now, you can query the `SPEAKER` index using the `/values/speaker` resource with the query options named “index-options”.

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. Though MarkLogic Server supports N-way co-occurrences, the *MarkLogic REST API* only supports 2-way 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 two 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 pair 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 93.

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",
	  }
	  }
	  }
	  }
	}

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:8003/v1/config/query/index-options
```

Now, you can query the SPEAKER index using the `/values/speaker-scene` resource with the query options named “index-options”.

4.3.5 Creating Indexes on JSON Keys

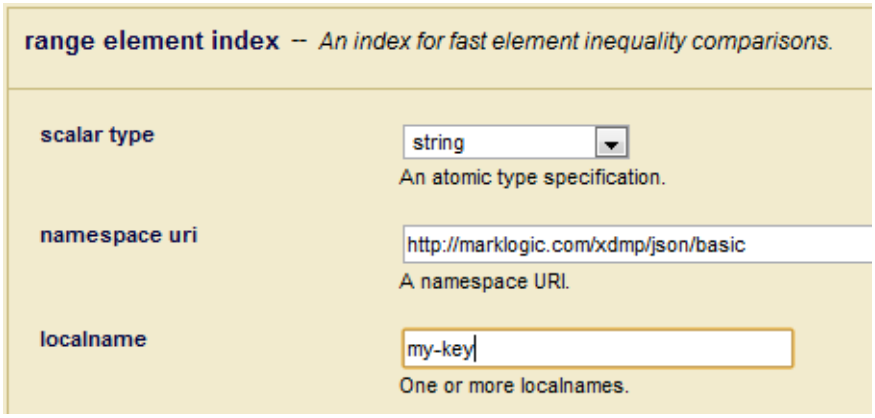
To efficiently search using JSON keys, you should define indexes on the keys. For example, using `/keyvalue` to query by JSON key values performs best when you define a value range index on the key you want to query against. To create such an index, you treat the JSON key as an XML element and define a corresponding element index, as described here.

JSON documents loaded using the *MarkLogic REST API* are stored internally as XML. Each JSON key has a corresponding XML element in the internal representation. The element name usually matches the JSON key name, but can be different if the key contains whitespace or other characters that cannot appear in an XML element name. Key names are encoded using the XQuery builtin `xdmp:encode-for-NCName`.

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

For more information on JSON support in MarkLogic Server, see Working With JSON in the *Application Developer's Guide*.

To create an index on a JSON key, you must determine the XML element name that corresponds to the key in the XML representation of your JSON content, and then create an element range index on that element. For example, to efficiently search the values of the JSON key "my-key", use the Admin Interface to create a range index with the following settings:



4.4 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

4.4.1 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 *MarkLogic REST API* to apply aggregate functions using `/values/{name}` in two ways:

- Include one or more `<aggregate/>` elements (XML) or "aggregate" 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 *MarkLogic REST API* to apply aggregate UDFs that require additional parameters.

4.4.2 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 93.

3. Apply the aggregate by sending a GET or POST request to `/values/{name}` and including the options from Step 2. For example: `GET /v1/values/speaker?options=index-options`.

For details on installing and using query options, see "Configuring Query Options" on page 92.

4.4.3 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=options_name
```

To analyze lexicon or index values or co-occurrences with a previously installed aggregate UDF, 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&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 93.

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 76.

7. Specify the result content type (XML or JSON) using the `format` parameter or the HTTP Accept headers. The default format is XML. For details, see "Controlling Output Content Type" on page 21.

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 User-Defined Functions in the *Application Developer's Guide* and the XQuery builtin function `cts:aggregate`.

4.4.4 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:8003/v1/values/speaker?options=index-options&agg
regate=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>
```

4.4.5 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 84.

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:8003/v1/values/speaker?options=index-options&agg
regate=count&aggregatePath=native/sampleplugin&view=aggregate&q=HAMLET
'
```

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"`
	`  }`
	`  }`
	`}`

4.5 Specifying Dynamic Query Options with Combined Query

A *combined query* is an XML or JSON wrapper around a string and/or structured query and query options. Use a combined query to specify query options on the fly, without first persisting them 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
- Interaction with Queries in Request Parameters
- Interaction with Persistent Query Options

- Performance Considerations
- Combined Query Examples

4.5.1 Syntax

A combined query can contain a string query, a structured query, query options, or any 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. If the combined query contains both a structured query and a string query, the two sub-queries are AND'd together.

The following table shows the structure of a combined query. Within the combined query wrapper, the queries and options use the same syntax as when they occur standalone.

Format	Combined Query Template

XML	`<search xmlns="http://marklogic.com/appservices/search">`
	`  <query>`
	`  <!-- structured query, same syntax as standalone -->`
	`  </query>`
	`  <qtext>your string query</qtext>`
	`  <options>`
	`  <!-- same syntax as standalone query options -->`
	`  </options>`
	`</search>`

JSON	`{"search":`
	`  "query": { structured query, same syntax as standalone },`
	`  "qtext": { "your string query here" }`
	`  "options": { same syntax as standalone query options },`
	`}`

For details on sub-query and query options syntax, see the following sections of the *Search Developer's Guide*:

- Searching Using Structured Queries
- Automatic Query Text Parsing and Grammar
- Search Customization Via Options and Extensions

4.5.2 Interaction with Queries in Request Parameters

When making a POST request to `/v1/search` or `/v1/values/{name}`, 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*

4.5.3 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.

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.

For example, suppose the following query options are installed under the name `my-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:8003/v1/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>
```

4.5.4 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 92.

4.5.5 Combined Query Examples

This section includes the following examples:

- Example: Overriding Persistent Constraints
- Example: Modifying the Search Response

4.5.5.1 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 94.

```
<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>
```

4.5.5.2 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:8003/v1/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 89.

4.6 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`, `/keyvalue`, and `/values`. 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

4.6.1 Controlling Queries With Options

Use the `options` request parameter to the `/search`, `/values`, and `/keyvalue` 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 A: JSON Query Options Reference” on page 160 or “Appendix B: XML Query Options Reference” on page 210.

MarkLogic Server comes configured with default query options. You can extend and modify the default options using `/config/query/default`.

You can also create custom query options. Before you can use a set of custom query options, you must install them as named options, using `/config/query/{name}`. For details, see “Configuring Query Options” on page 92.

Once you install query options under a name, you can apply them to a `/search`, `/keyvalue`, or `/values` request using the `options` request parameter. The following example searches for the word “julius”, using the query options named “my-options”:

```
http://localhost:8003/v1/search?q=julius&options=my-options
```

4.6.2 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`, `/keyvalue`, or `/values/{name}` to supply the name of pre-installed persistent options.

- Use the `options` parameter of a POST request to `/search` or `/values/{name}` 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.

Both 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 94.

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 87.

4.6.3 Creating or Modifying Query Options

To install or modify 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
```

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 93.

2. Place the XML or JSON option data in the request body.

   For syntax details, see "Appendix A: JSON Query Options Reference" on page 160 or "Appendix B: XML Query Options Reference" on page 210.

3. Specify the MIME type of the body content in the `format` parameter or the HTTP `Content-type` header, as described in "Controlling Input Content Type" on page 20. 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 30.

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:8003/v1/config/query/title-only
...
HTTP/1.1 204 Content Updated

$ curl --anyauth --user user:password -X GET \
  'http://localhost:8003/v1/search?q=title:julius&options=title-only'
...
<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 `format` request parameter instead of 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'
http://localhost:8003/v1/config/query/title-only?format=json
...
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:8003/v1/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>
```

4.6.4 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 Content Type” on page 20. 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 A: JSON Query Options Reference” on page 160 or “Appendix B: XML Query Options Reference” on page 210.

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 30.

4.6.5 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 94.

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 \
    http://localhost:8003/v1/config/indexes/tuples?format=xml
```

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"`
	`  }`
	`}`

4.6.6 Retrieving Options

To retrieve previously installed 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. Use the `format` request parameter or `Accept` header to control the response format. XML is the default format. For details, see “Controlling Output Content Type” on page 21.

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 A: JSON Query Options Reference” on page 160 or “Appendix B: XML Query Options Reference” on page 210.

As when retrieving a whole set, results can be requested as XML or JSON, using the `format` parameter or `Accept` header. For details, see “Controlling Output Content Type” on page 21. 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:8003/v1/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:8003/v1/config/query/title-only/term
...
<options xmlns="http://marklogic.com/appservices/search">
  <term>
    <term-option>case-sensitive</term-option>
```

```
  </term>
</options>
```

4.6.7 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/name
```

MarkLogic Server responds with a list of names in XML or JSON. XML is the default format. Use the `format` request parameter or the HTTP `Accept` header to specify the response format, as described in "Controlling Output Content Type" on page 21.

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".

```
$ curl --anyauth --user user:password -X GET \
    http://localhost:8003/v1/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:8003/v1/config/query
...
[
  {"name":"title-only","uri":"/v1/config/query/title-only"}
  {"name":"play-type","uri":"/v1/config/query/play-type"}
]
```

4.6.8 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

4.6.8.1 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.

4.6.8.2 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:8003/v1/config/query/my-options/constraint
```

For details on query options names and structure, see "Appendix A: JSON Query Options Reference" on page 160 or "Appendix B: XML Query Options Reference" on page 210.

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 30.

For details on query option names, see "Appendix A: JSON Query Options Reference" on page 160 or "Appendix B: XML Query Options Reference" on page 210.

4.6.8.3 Removing All Named Query Options

To remove all named query options from a *MarkLogic REST 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.

4.7 Using Namespace Bindings

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

4.7.1 When Do You Need a Namespace Binding

Use the `/config/namespaces` service to define namespace prefixes that you can use in your queries. For example, to specify an element with QName `theNS:theElement` in a `/keyvalue` search, you might refer to it as follows:

```
http://host:port/version/keyvalue?element=theNS:theElement&value=foo
```

There is no way to dynamically specify the namespace in your request, so you must pre-define the namespace prefix binding using `/config/namespaces` or `/config/namespace/{name}`.

4.7.2 Creating or Updating a Namespace Binding

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 104.

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 Content Type” on page 20. 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:8004/v1/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:8004/v1/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 \
    http://localhost:8004/v1/config/namespaces/bill?format=json

{
  "prefix":"bill",
  "uri":"http://marklogic.com/examples/shakespeare"
}
```

4.7.3 Creating or Updating Multiple Namespace Bindings

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 102.

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 Content Type” on page 20. 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:8004/v1/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"
    }
  ]
}

# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X PUT -d@./ns-bindings.json \
    -H "Content-type: application/json" \
    http://localhost:8004/v1/config/namespaces
```

You can examine the binding by sending a GET request to `/config/namespaces/{name}` or to `/config/namespaces`.

4.7.4 Listing Available Namespace Bindings

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 using the format request parameter or the Accept headers. For details, see “Controlling Output Content Type” on page 21.

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 \
    http://localhost:8004/v1/config/namespaces?format=xml
```

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:

```
$ curl --anyauth --user user:password -X GET \
    http://localhost:8004/v1/config/namespaces/one?format=xml

<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 \
    http://localhost:8004/v1/config/namespaces/one?format=json

{"prefix":"one","uri":"http://marklogic.com/examples/one"}
```

4.7.5 Deleting Namespace Bindings

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:8004/v1/config/namespaces/one
```

4.8 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. To search on a collection facet, you must enable the “collection lexicon” database configuration parameter in the Admin Interface.

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:8003/v1/config/query/facet-options
```

If you query `/search` or `/keyvalue` 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:8003/v1/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:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
  'http://localhost:8003/v1/search?options=facet-options&q=castle&form
at=json'
...
{
  "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:8003/v1/search?options=facet-options&q=castle
type:Comedy'
```

4.9 Paginating Results

When you query the database using `/search` and `/keyvalues`, 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:8003/v1/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:8003/v1/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>
```

4.10 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 in two ways:

- Use query options to customize the results returned by the default snippeting function.
- Write your own snippeting function in XQuery and install it as an extension.

4.10.1 Customizing Search Snippets With Query Options

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 options, 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` element (XML) or child object (JSON).

2. Install these query options under a name or as the default options using the `/config/query` service. For details, see "Configuring Query Options" on page 92.

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 93.

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:8003/v1/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>
```

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-elements>`
	`  <element ns="" name="LINE" />`
	`  </preferred-elements>`
	`  </transform-results>`
	`  <search-option>filtered</search-option>`
	`</options>`

JSON	`{`
	`  "options": {`
	`  "transform-results": {`
	`  "apply": "snippet",`
	`  "max-matches": 3,`
	`  "preferred-elements": [`
	`  {`
	`  "ns": "",`
	`  "name": "LINE"`
	`  }`
	`  ]`
	`  },`
	`  "search-option": true`
	`  }`
	`}`

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:8003/v1/config/query/snippet-lines

$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
    "http://localhost:8003/v1/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:8003/v1/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:8003/v1/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,"
          ]
        }
      ]
    }
  ],
  ...
}
```

4.10.2 Creating Your Own Snippet Extension

If the `tranform-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*.

Custom snippet transformations cannot be installed or applied directly using the *MarkLogic REST API*. Instead, you must manually install your transformation function in the modules database of your REST instance and apply by specifying it in custom query options. In particular, you cannot install your custom snippeting function using the `/config/transforms` service.

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. Manually install your module in the modules database of your REST API instance, similar to installing a dependent library for a resource service extension. For details, see "Installing Dependent XQuery Library Modules" on page 145.

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 93.

The following example builds on the example from "Customizing Search Snippets With Query Options" on page 112. 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:

```
# 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:8003/v1/config/query/snippet-lines

$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
    "http://localhost:8003/v1/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 of 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["), "\[")
          return
          text {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], "]"))}
        }
        else $child
    }
};
```

If the above module is installed in the instance modules database with the URI `/my.domain/my-snippets.xqy`, then we can use the following options to generate custom snippets by supplying the localname 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="/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": "/my.domain/my-snippets.xqy",`
	`  "max-matches": 3,`
	`  "preferred-elements": [`
	`  {`
	`  "ns": "",`
	`  "name": "LINE"`
	`  }`
	`  ]`
	`  },`
	`  "search-option": true`
	`  }`
	`}`

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
$ curl --anyauth --user user:password -X PUT \
    -d @./transform-results.xml \
    -H "Content-type: application/xml" \
    http://localhost:8003/v1/config/query/custom-snippet
```

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/xml" \
    "http://localhost:8003/v1/search?q=hamlet&options=custom-snippet"

<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/xml` or use the `format` request parameter. For example:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/json" \
    "http://localhost:8003/v1/search?q=hamlet&options=custom-snippet"

{
  "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]\
/SPEECH[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]\
/SPEECH[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"
          ]
        }
      ]
    }
  ],
  ...
}
```

5.0 Managing Transactions

The *MarkLogic REST 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

5.1 Service Summary

Many of the services available through the *MarkLogic REST 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 122.	“Creating a Transaction” on page 122
Commit	PUT	Commit an open transaction.	“Committing or Rolling Back a Transaction” on page 124
Rollback	DELETE	Rollback a transaction, discarding all related changes.	“Committing or Rolling Back a Transaction” on page 124
Status	GET	Retrieve transaction status in XML or JSON.	“Checking Transaction Status” on page 124

5.2 Overview of RESTful Transactions

This section gives a brief introduction to the MarkLogic Server transaction model as it applies to the *MarkLogic REST 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 *MarkLogic REST 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 *MarkLogic REST 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 *MarkLogic REST 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 122.

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 123.

3. Commit the transaction by sending a PUT request to the `/transactions` service, or rollback the transaction by sending a DELETE request to the `/transactions` service. See "Committing or Rolling Back a Transaction" on page 124.

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.

5.3 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 123.

The transaction id returned in the Location header is of the form:

```
/transactions/hostid_txnid
```

For example:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl -X POST -d "" --anyauth --user user:password \
    http://localhost:8003/v1/transactions
...
HTTP/1.1 303 See Created Transaction
Location: /transactions/9146583258914468644_3148548124558550433
```

When supplying the transaction id to future requests, use only the `hostid_txnid` portion. For example:

```
$ curl -X PUT -T ./example.xml --anyauth --user user:password \
  'http://localhost:8003/v1/documents?uri=/xml/example.xml&txid=914658
3258914468644_3148548124558550433'
```

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
```

5.4 Associating a Transaction with a Request

Once you create a transaction, as described in "Creating a Transaction" on page 122, 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:8003/v1/documents?uri=/xml/example.xml&txid=914658
3258914468644_3148548124558550433'
```

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 124.

5.5 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 of the form *`hostid_txnid`*, as 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.

5.6 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 of the form *`hostid_txnid`*, as 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:8003/v1/transactions/1506593501555860493_21610463
5458437451
```

To make the equivalent request using the format parameter:

```
$ curl -X GET --anyauth --user user:password \
    http://localhost:8003/v1/transactions/1506593501555860493_21610463
5458437451?format=json
```

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>kim-3.marklogic.com</rapi:host-name>
  </rapi:host>
  <rapi:server>
    <rapi:server-id>17994162681305741146</rapi:server-id>
    <rapi:server-name>RESTstop</rapi:server-name>
  </rapi:server>
  <rapi:database>
    <rapi:database-id>8896678293064963766</rapi:database-id>
    <rapi:database-name>bill</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": "kim-3.marklogic.com"
    },
    "rapi:server": {
      "rapi:server-id": "17994162681305741146",
      "rapi:server-name": "RESTstop"
    },
    "rapi:database": {
      "rapi:database-id": "8896678293064963766",
      "rapi:database-name": "bill"
    },
    "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`.

6.0 Working With Content Transformations

The *MarkLogic REST API* allows you to create custom content transformations and apply them during operations such as document ingestion and retrieval. Transforms can be implemented as XQuery library modules or XSLT stylesheets, and can accept transform-specific parameters. Transforms can be applied on any *MarkLogic REST 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
- XQuery Example: Adding an Attribute During Ingestion
- XSLT Example: Adding an Attribute During Ingestion

6.1 Writing Transformations

This section covers the following topics:

- Guidelines for Writing Transforms
- Writing XQuery Transformations
- Writing XSLT Transformations
- Reporting Errors
- Context Map Keys

6.1.1 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 *MarkLogic REST 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`, 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 *MarkLogic REST API* framework manages the request context.
- Report errors by throwing one of the exceptions provided by the *MarkLogic REST API*. For details, see "Reporting Errors" on page 129.

6.1.2 Writing XQuery Transformations

To create an XQuery transformation, implement an XQuery library module in the 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 131.
$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 131.
$content	The input document to which to apply the transformation.

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*.

For a complete example, see "XQuery Example: Adding an Attribute During Ingestion" on page 134.

6.1.3 Writing XSLT Transformations

To create an XSLT transformation, implement an XSLT stylesheet that accepts the following parameters:

Parameter	Type	Description
`$context`	`map:map`	Service request context information such as input document type and URI, and output types accepted by the caller. For details, see "Context Map Keys" on page 131.
`$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 131.

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]'/>
...
```

For a complete example, see "XSLT Example: Adding an Attribute During Ingestion" on page 137.

6.1.4 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 22.

To report an error from your implementation, use the `fn:error` XQuery function with the `RESTAPI-EXTNERR` 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", "payload-format", "response-payload")`. That is, when using `fn:error` to raise `REST-EXTNERR`, 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.
- Payload Format. Allowed values: xml or json. If there is a payload format, there must be a payload.
- Response payload.

Note: Best practice is to use `RESTAPI-EXTNERR`. 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-EXTNERR` 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-EXTNERR",
          ("415","Raven","xml",xdmp:quote(<word>nevermore</word>)))
    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/xml
Server: MarkLogic
Set-Cookie: SessionID=714070bdf4076536; path=/
Content-Length: 62
Connection: close
```

```
<?xml version="1.0" encoding="UTF-8"?>
<word>nevermore</word>
```

6.1.5 Context Map Keys

The context map parameter made available to XQuery and XSLT transformations can contain the following keys:

Key	Description
`input-type`	The MIME type of the input document.
`uri`	The URI of the input document.
`accept-types`	The MIME types accepted by the requestor. Multiple MIME types are separated by whitespace.
`output-type`	The expected output document MIME type. If the transform supports content negotiation, the transform function can set/modify this value.

6.2 Installing Transformations

This section covers the following topics:

- Basic Installation
- Defining Transform-Specific Parameters
- Including Transform Metadata

6.2.1 Basic Installation

To install or update a transformation, 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.

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 `application/xquery` for an XQuery transformation and `application/xslt+xml` for an XSLT transformation.

If your transform requires additional user-defined XQuery library modules, you must install them separately before installing your transform. For details, see "Installing Dependent XQuery Library Modules" on page 145.

For a complete example, see "XSLT Example: Adding an Attribute During Ingestion" on page 137 or "XQuery Example: Adding an Attribute During Ingestion" on page 134.

6.2.2 Defining Transform-Specific Parameters

If the transform expects parameters, include the parameters 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=stri
ng\?
```

6.2.3 Including Transform Metadata

You can include metadata for `provider`, `version`, `title`, and `description` as request parameters. For example:

```
http://host:port/version/config/transforms/example?trans:reviewer=stri
ng\?&provider=MarkLogic%20Corporation
```

Metadata can be retrieved by sending a GET request to the /config/transforms service. See "Discovering Transformations" on page 133.

6.3 Applying Transformations

All *MarkLogic REST 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" with one parameter named "reviewer", the PUT request URL might look like the following:

```
http://host:port/version/documents?uri=/doc/theDoc.xml&transform=examp
le&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 "XSLT Example: Adding an Attribute During Ingestion" on page 137 or "XQuery Example: Adding an Attribute During Ingestion" on page 134.

6.4 Discovering Transformations

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` headers or the `format` request parameter to explicitly specify expected content type, as described in "Controlling Output Content Type" on page 21.

The following example requests an XML report:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X GET \
    http://localhost:8003/v1/config/transforms?format=xml
...
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>
```

6.5 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 `application/xquery` for an XQuery transform. Set the request Accept header to `application/xslt+xml` for an XSLT transform.

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:8003/v1/config/transforms/example
...
xquery version "1.0-ml";
module namespace
example="http://marklogic.com/rest-api/transform/example";
declare function example:transform( ...
```

6.6 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. Create a *MarkLogic REST API* instance on port 8003. For instructions, see “Creating an Instance” on page 24.

2. Cut and paste the code above into a file called “add-attr.xqy”.

3. 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:8003/v1/config/transforms/add-attr?trans:name=st
ring\?&trans:value=string\?'
```

4. 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 \
    http://localhost:8003/v1/config/transforms?format=xml
...
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: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. 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:8003/v1/documents?uri=/doc/transformed.xml&trans
form=add-attr&trans:name=example&trans:value=new'
```

6. 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:8003/v1/documents?uri=/doc/transformed.xml
<?xml version="1.0" encoding="UTF-8"?>
<data example="new"/>
```

6.7 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 *MarkLogic REST API* instance on port 8003. For instructions, see “Creating an Instance” on page 24.

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:8003/v1/config/transforms/add-attr-xsl?trans:nam
e=string\?&trans:value=string\?'
```

4. 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 \
    http://localhost:8003/v1/config/transforms?format=xml
...
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/>
    <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. Make a PUT request to the `/documents` service 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:8003/v1/documents?uri=/doc/transformed.xml&trans
form=add-attr-xsl&trans:name=example&trans:value=new'
```

6. 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:8003/v1/documents?uri=/doc/transformed.xml
<?xml version="1.0" encoding="UTF-8"?>
<data example="new"/>
```

7.0 Extending the REST API

You can extend the *MarkLogic REST API* by installing a resource service extension. This chapter covers the following topics:

- Understanding Resource Service Extensions
- Creating a Resource Service Extension
- Installing a Resource Service Extension
- Using a Resource Service Extension
- Discovering Resource Service Extensions
- Retrieving the Implementation of a Resource Service Extension
- Example Resource Service Extension

7.1 Understanding Resource Service Extensions

Use the resource service extension capability to make new REST service endpoints available through a *MarkLogic REST API* instance. To make a resource service extension available, follow these steps, which are discussed in detail in this chapter.

1. Create an XQuery library module that implements the extension.

2. Install the extension in a *MarkLogic REST API* instance to make it available to applications.

3. Access the extension service through a `/resources/{extensionName}` service endpoint.

7.2 Creating a Resource Service Extension

This section covers the following topics:

- Guidelines for Writing Resource Service Extensions
- The Resource Extension Interface
- Returning JSON Documents
- Reporting Errors
- Context Map Keys
- Controlling Transaction Mode

7.2.1 Guidelines for Writing 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 XQuery library modules depends on other user-defined modules, those modules must be installed in the modules database of the *MarkLogic REST API* instance.
- Report errors by throwing one of the exceptions provided by the *MarkLogic REST API*. For details, see "Reporting Errors" on page 142.

7.2.2 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 *MarkLogic REST API* instance in which it is installed.
- *yourNSAlias* is a namespace alias 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 localnames shown (`get`, `put`, etc.).

```
xquery version "1.0-ml"
module namespace yourNSAlias =
    "http://marklogic.com/rest-api/resource/extensionName";

declare function yourNSAlias:get(
    $context as map:map,
    $params  as map:map
) as document-node()*

declare function yourNSAlias:put(
    $context as map:map,
    $params  as map:map,
    $input   as document-node()*
) as document-node()?

declare function yourNSAlias:post(
    $context as map:map,
    $params  as map:map,
```

```
    $input   as document-node()*
) as document-node()*

declare function yourNSAlias: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 must define the argument names and types when installing your extension; for details see "Installing a Resource Service Extension" on page 143. Arguments are passed to your methods through the `$params` map. See the `get` implementation in "Example Resource Service Extension" on page 149.

The following template summarizes the meaning of the function parameters:

Parameter	Description
`$context`	Service request context information such as input document type and URI, and output types accepted by the caller. For details, see "Context Map Keys" on page 142.
`$params`	Extension-specific parameters. The map contains a key for each parameter on a request to the extension service. Define parameters when installing the extension. For details, see "Installing a Resource Service Extension" on page 143.
`$input`	The input document to which to apply the transformation.

Your extension can return XML, text, or binary documents. Set `output-types` 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.

7.2.3 Returning JSON Documents

This section covers important information for extensions that support JSON documents.

To return a constructed JSON document, create a text document containing JSON and set `output-types` to `application/json` in the `$context` map. For example:

```
declare function example:get(
    $context as map:map,
    $params  as map:map
```

```
) as document-node()* {
    map:put($context,"output-types","application/json"),
    document { text { '{"key":"value"}' } }
};
```

If your extension operates on a document previously inserted into the database as a JSON document using the `/documents` service, the document will be in the internal XML representation when you access it in your extension function. If you want to return such a document from your extension as JSON, you are responsible for converting it back to JSON using the XQuery builtin `json:transform-to-json`. For example:

```
declare function example:get(
    $context as map:map,
    $params  as map:map
) as document-node()* {
    map:put($context,"output-types","application/json"),
    document { text {
      json:transform-to-json(fn:doc("/example/from-json.xml"))
    } }
};
```

For details, see Working With JSON in *Application Developer's Guide*.

7.2.4 Reporting Errors

Extensions and transforms use the same mechanism to report errors to the calling application: Use `fn:error` to report a `RESTAPI-EXTNERR` 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 details, see "Reporting Errors" on page 129.

7.2.5 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-types`	The output document MIME type. The service must set `output-types` for all output documents.

For a complete example of a resource service extension, see “Example Resource Service Extension” on page 149.

7.2.6 Controlling Transaction Mode

This section discusses the default transaction mode under which the implementation of a resource service extension method 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
- `put` - update
- `post` - update
- `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()?
{...};
```

7.3 Installing a Resource Service Extension

You install a resource service extension XQuery module using the *MarkLogic REST API*. Any dependent user-defined modules must be installed separately before installing your extension.

This section covers the following topics:

- Installing the Extension XQuery Module
- Installing Dependent XQuery Library Modules

7.3.1 Installing the Extension XQuery Module

To install or update an extension, send a PUT request to the `/config/resources` service 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: *extensionName* must match the name in the extension module namespace declaration. For example:

```
xquery version "1.0-ml";
module namespace yourNSAlias =
  "http://marklogic.com/rest-api/resource/extensionName";
...
```

Set the Content-type of the request body to `application/xquery`.

If the extension service expects parameters, "declare" the parameters using request parameters when installing the extension. Use a request parameter of the following form to declare each extension parameter:

```
method:paramName=paramTypeAndCardinality
```

Where *method* 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 "Example Resource Service Extension" on page 149.

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 “Installing Dependent XQuery Library Modules” on page 145.

7.3.2 Installing Dependent XQuery Library Modules

If your resource extension or transform XQuery library module depends on additional library modules, install these modules into the modules database of the REST API instance or in the Modules directory of your MarkLogic Server installation.

Installation into the modules database is strongly recommended because this ensures MarkLogic Server will propagate the dependent library modules throughout your cluster.

Note: Install dependent modules before using the *MarkLogic REST API* to install your extension or transform that requires them.

This section covers the following topics:

- Installing Into the Modules Database
- Installing Into the Modules Directory
- Maintenance of Dependent Libraries

7.3.2.1 Installing Into the Modules Database

It is recommended that you install dependent libraries into the modules database of your REST API instance so that MarkLogic Server will propagate them throughout your cluster automatically.

Note: This feature is supported in REST API instances created with MarkLogic Server 6.0-2 and later.

You can install your dependent libriaries under any document URI, but it is good practice to prefix the URI by your domain name and the name of your extension. To avoid name collisions, you should not install your module using the URI prefixes reserved by the *MarkLogic REST API* for installing extension and transformation components (`/marklogic.rest.resource/` or `/marklogic.rest.transform/`).

To use a dependent library installed in the modules database 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 `/my.domain/my-ext/lib/dependent.xqy`, then the extension module using this library should include an import of the form:

```
import module namespace dep="mylib" at
    "/my.domain/my-ext/lib/dependent.xqy";
```

Follow these instructions to use Query Console to install a dependent library module:

1. Place your dependent library modules in a directory on your MarkLogic Server host that is accessible to MarkLogic Server. For example:

```
$ cp /user/me/dependent.xqy /space/resources/
```

2. Open Query Console in your browser by navigating to `http://yourhost:8000/qconsole`.

3. In Query Console, create a query similar to the following to load the module.

```
xdmp:document-load(
  "/space/resources/dependent.xqy",
  <options xmlns="xdmp:document-load">
    <uri>/my.domain/my-ext/lib/dependent.xqy</uri>
  </options>
  )
```

4. In the Query Console Content source dropdown list, select the modules database for your REST API instance. For example, if your instance is named “RESTstop”, select “RESTstop-modules”.

5. Click Run to execute the query and install the dependent module.

6. Optionally, click Explore to see that the library is now in the modules database.

7.3.2.2 Installing Into the Modules Directory

If you choose to install dependent libraries into the Modules directory of your MarkLogic Server installation, you must manually do so on each node in your cluster. Best practice is to install your libraries into the modules database of your REST API instance.

To install a dependent library into the Modules directory, copy the file into *MARKLOGIC_INSTALL_DIR*`/Modules` or into a subdirectory of this directory. The default location of this directory is:

- Unix: `/opt/MarkLogic/Modules`
- Windows: `C:\Program Files\MarkLogic\Modules`

To use a dependent library installed in the Modules directory, use a module URI that is the same as the file pathname after stripping `MARKLOGIC_INSTALL_DIR/Modules`. For example, if your extension module contains an import of the form:

```
import module namespace dep="mylib" at "/my-ext/dependent.xqy";
```

Then `dependent.xqy` should be located in:

MARKLOGIC_INSTALL_DIR`/my-ext/dependent.xqy`

MarkLogic Server does not automatically replicate modules installed in the Modules directory across a cluster. You must manually copy the files to each host in the cluster.

7.3.2.3 Maintenance of Dependent Libraries

When you install or update a dependent library module installed to the modules database of your REST API instance, the module is replicated across your cluster automatically. There can be a delay of up to one minute between updating and availability.

If you install a dependent library module in the Modules directory of your MarkLogic Server installation, then you must manually install and update the module throughout your cluster.

MarkLogic Server does not automatically remove dependent modules when you delete the related extension or transform.

Dependent modules installed in the modules database 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 26.

You must always manually delete library modules installed in the Modules directory of your MarkLogic Server installation.

7.4 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:8003/v1/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 147.

7.5 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` headers or the `format` request parameter to explicitly specify a content type, as described in "Controlling Output Content Type" on page 21.

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:8004/v1/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>
```

7.6 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:8004/v1/config/resources/example
...
xquery version "1.0-ml";
```

```
module namespace
example="http://marklogic.com/rest-api/resource/example";
declare function example:get( ...
```

7.7 Example Resource Service Extension

This section demonstrates the implementation, installation and invocation of a resource service extension.

- Overview
- Installing the Example Extension
- Using the Example Extension
- Example Extension Implementation

7.7.1 Overview

This example 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 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.

If you 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 142.

7.7.2 Installing the Example Extension

Installing the example requires identifying all the implemented methods, as well as the argument expected by the get method. To install the example:

1. If you do not already have a REST API instance, create one. This example assumes an instance is running on localhost:8004.

2. Copy the code from "Example Extension Implementation" on page 151 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 and description 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:8004/v1/config/resources/example?title=test
resource&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 the installation URL does.

Request Parameter	Description
`title=test resource`	Optional metadata.
`version=1.0`	Optional metadata.
`provider=marklogic`	Optional metadata.
`description=A simple resource example`	Optional metadata.
`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.

7.7.3 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:8004/v1/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:8004/v1/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.

7.7.4 Example Extension Implementation

```
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-types', used to generate content-type header :)
    let $output-types :=
```

```
        map:put($context,"output-types","application/xml")
    let $arg1 := map:get($params,"arg1")
    let $content :=
        <args>
            {for $arg in $arg1
             return <arg1>{$arg1}</arg1>
            }
        </args>
    return (xdmp:set-response-code(200,"OK"),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!")
};
```

8.0 Migrating REST Applications

This chapter describes how to migrate an application hosted on a *MarkLogic REST 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 Application Builder, the *MarkLogic REST API*, or the MarkLogic Java API.

The following topics are covered:

- Before You Begin
- Upgrading Your Application to a New REST API Instance
- Migrating the Content Database
- Migrating the REST API Instance

8.1 Before You Begin

The migration procedure outlined by this chapter assumes you have the following:

- MarkLogic Server 6.0-2 or later 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.

  To download and install mlcp, see 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 19.
- Administrative user privileges on the source and destination MarkLogic Server instances.

In addition, your application must be running on a *MarkLogic REST API* instance created with MarkLogic Server 6.0-2 or later. If your source application was not created or deployed for the first time on the source host using MarkLogic Server 6.0-2, see “Upgrading Your Application to a New REST API Instance” on page 153.

8.2 Upgrading Your Application to a New REST API Instance

You can skip this section if you never installed MarkLogic Server 6.0-1 on your source host.

Between versions 6.0-1 and 6.0-2, changes were made to the *MarkLogic REST API* to facilitate application migration. If you are migrating an application running on a REST API instance created with MarkLogic Server 6.0-1, you must first upgrade the instance to MarkLogic Server6.0-2 or later.

Your application is likely to be running on a MarkLogic Server 6.0-1 REST API instance if any of the following conditions are met, even if you subsequently upgraded from MarkLogic Server 6.0-1 to 6.0-2 or later:

- You created your application on the source host using MarkLogic Server 6.0-1.
- You deployed an existing application to the source host when it was running MarkLogic Server 6.0-1.

If you meet one of these criteria, you must re-host your application on a new REST API instance. Choose from one of the following procedures:

- Upgrading an Application Builder Application
- Upgrading Other Applications

8.2.1 Upgrading an Application Builder Application

This procedure moves your source application to a new REST API instance. You only need to perform this procedure if your application was created with Application Builder and one of the following conditions is true.

- You created your application on the source host using Application Builder and MarkLogic Server 6.0-1.
- You initially deployed your application on the source host using Application Builder and MarkLogic Server 6.0-1. This applies even if you subsequently upgraded the source host to MarkLogic Server 6.0-2 or later because the upgrade does not recreate the REST API instance.

If your application was not created with Application Builder, see “Upgrading Other Applications” on page 155.

If one of the above conditions is met, you must redeploy your application on a REST API instance created with MarkLogic Server 6.0-2 or later before migrating it to a new host. Doing so picks up changes that support seamless application migration across hosts.

> **Note:** This procedure will change the REST API instance name and the port number on which your application runs.

To create a new REST API instance for your application:

1. If the source host is not already running MarkLogic Server 6.0-2 or later, upgrade to MarkLogic Server 6.0-2 or later.

2. In your browser, navigate to the Application Services page on your source host. For example, navigate to:

```
http://localhost:8000/appservices
```

3. Click the name of your application in the list of Application Builder Applications. The Search page appears.

4. Click on Deploy at the top of the page. The Deploy page appears.

5. Select New REST API Instance on the Deploy page.

6. Accept the default name and port number, or enter a name and port number different from the App Server name and port number currently used by your application.

7. Click Deploy. Your application is deployed on a new REST API instance.

8. If your application includes extensions or customizations, re-install them to the modules database of the new App Server. For details, see Extending Applications Built With Application Builder in the *Application Builder Developer's Guide*.

9. Test your application on the new REST API instance.

10. Optionally, delete the old REST API instance and associated modules database using `curl` or an equivalent tool. For details, see "Removing an Instance" on page 26. Deleting the REST API instance will cause a server restart. For example, if your source host is localhost and your REST API instance name is "Oscars":

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X DELETE -i \
    http://localhost:8002/v1/rest-apis/Oscars?include=modules
```

8.2.2 Upgrading Other Applications

This procedure moves your application to a new REST API instance. Use this procedure only if both of the following conditions are met.

- Your application was not created using Application Builder. If your application was created with Application Builder, see "Upgrading an Application Builder Application" on page 154.
- Your application is running on the source host on a REST API instance created using MarkLogic Server 6.0-1.

If you meet both these conditions, perform the following steps on your source host before migrating your application to a new host:

1. If the source host is not already running MarkLogic Server 6.0-2 or later, upgrade to MarkLogic Server 6.0-2 or later.

2. Delete the current REST API instance and associated modules database. For details, see "Removing an Instance" on page 26. Deleting the REST API instance will cause a server

restart. For example, assuming your source host is localhost and your REST API instance name is “Oscars”:

```
# Windows users, see Modifying the Example Commands for Windows
$ curl --anyauth --user user:password -X DELETE -i \
    http://localhost:8002/v1/rest-apis/Oscars?include=modules
```

3. Recreate the REST API instance using Information Studio or the *MarkLogic REST API*. For details, see “Creating an Instance” on page 24. This example uses the REST API with a previously saved configuration file:

```
$ cat config.xml
<rest-api xmlns="http://marklogic.com/rest-api">
  <name>Oscars</name>
  <database>Oscars</database>
  <port>8003</port>
</rest-api>

$ curl -X POST --anyauth --user user:password -d @"./config.xml" \
    -H "Content-type: application/xml" \
    http://localhost:8002/v1/rest-apis
```

4. If you modified the default REST API configuration properties, set them again. For details, see “Configuring Instance Properties” on page 30.

5. If your application includes custom query options, content transformations, or resource service extensions, redeploy them to the new instance. For details, see “Creating or Modifying Query Options” on page 94, “Installing Transformations” on page 131, “Installing a Resource Service Extension” on page 143.

8.3 Migrating the Content Database

Use this procedure to migrate your application content database configuration and contents to the destination host. You must do this before migrating the REST API instance as described in “Migrating the REST API Instance” on page 158.

If the content database already exists on the destination host, its configuration may be changed by this procedure. You will have a chance to review the changes when importing the source configuration.

If the content database does not already exist on the destination, it will be created for you.

1. Export the content database configuration on the source MarkLogic Server instance using Configuration Manager. For details, see Exporting a Configuration in the *Administrator’s Guide*. An XML file that contains the configuration is created.

2. Import the content database configuration file created in the previous step to the destination MarkLogic Server instance using Configuration Manager. For details, see

Importing a Configuration in the *Administrator's Guide*. An empty content database is created on the destination instance.

3. Copy the content from the source database to the destination database using MarkLogic Content Pump (mlcp), XQSync, or a similar tool. To use mlcp to copy your content:

   a. Create an XDBC App Server attached to the content database on the source host. For details, see Creating a New XDBC Server in the *Administrator's Guide*.

   b. Create an XDBC App Server attached to the content database on the destination host.

   c. 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 *Loading Content Into MarkLogic Server Guide*. For example:

```
$ mlcp.sh copy -mode local \
    -input_host src-host -input_port src-xdbc-port \
    -input_username user -input_password password \
    -output_host dest-host -output_port dest-xdbc-port \
    -output_username user -output_password password
```

   d. If you cannot reach the 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 to an Archive and Importing Content Into MarkLogic Server in the *Loading Content Into MarkLogic Server Guide*. For example:

```
# Export the source database content to an archive
$ mlcp.sh export -mode local -output_type archive \
    -host src-host -port src-xdbc-port \
    -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 -port dest-xdbc-port \
    -username user -password password
    -input_file_path dir-containing-archive
```

You can now remove the XDBC App Servers created during this procedure, or save them to re-use for migrating the REST API instance.

8.4 Migrating the REST API Instance

Use this procedure to migrate your application configuration and logic to a new REST API instance on the destination host. This step creates a new REST API instance on the destination host, and then overwrites the configuration and code it contains with the equivalent information from the source instance.

If the destination is a multi-host MarkLogic Server cluster, you only need to perform this procedure on one E-node.

> **Note:** The content database must exist on the destination host before performing this procedure. For details, see “Migrating the Content Database” on page 156.

1. On the destination host, create a new REST API instance with the same group, name and port number as the source App Server. For details, see “Creating an Instance” on page 24. Use the database created in “Migrating the Content Database” on page 156 as the content database.

2. On the destination host, create any MarkLogic Server users and roles required by your application. Use the same user and role name as on the source host.

3. Use MarkLogic Content Pump, XQSync, or a similar tool to copy the REST API instance modules database from the source to the destination. To use mlcp to copy your content:

   a. On the source host, create an XDBC App Server attached to the REST API instance modules database. For example, if your REST API instance is named “Oscars”, the default modules database name will be “Oscars-modules”.

   b. On the destination host, create an XDBC App Server attached to the modules database of the REST API instance created in Step 1.

   c. If the source and destination hosts are both reachable from the host where you are running mlcp, use the mlcp `copy` command, as described in Copying Content Between Databases in the *Loading Content Into MarkLogic Server Guide*. For example, you can use a command of this form:

```
$ mlcp.sh copy -mode local \
    -input_host src-host -input_port src-xdbc-port \
    -input_username user -input_password password \
    -output_host dest-host -output_port dest-xdbc-port \
    -output_username user -output_password password
```

   d. If you cannot reach the both the source and destination host from the same host, use the mlcp `export` and `import` commands, as described in Exporting to an Archive and Importing Content Into MarkLogic Server in the *Loading Content Into MarkLogic Server Guide*. For example:

```
# Export the source modules database content to an archive
$ mlcp.sh export -mode local -output_type archive \
    -host src-host -port src-xdbc-port \
    -username user -password password \
    -output_file_path an-existing-directory
# one or more ZIP files are created in the output directory

# Import the archive to the destination modules database
$ mlcp.sh import -mode local -input_file_type archive \
    -host dest-host -port dest-xdbc-port \
    -username user -password password
    -input_file_path dir-containing-archive
```

Your application should now be usable with the destination MarkLogic Server instance.

9.0 Appendix A: JSON Query Options Reference

This chapter defines the JSON structure of query options (`search:options`) used by the search related features of the *MarkLogic REST API* and the MarkLogic Java API.

Query options can also be expressed as XML. For details, see "Appendix B: XML Query Options Reference" on page 210.

- Query Options
- attribute-descriptor
- bucket
- collection-constraint
- computed-bucket
- constraint
- custom-constraint
- element-descriptor
- extract-metadata
- field-descriptor
- geo-attr-pair-constraint
- geo-elem-constraint
- geo-elem-pair-constraint
- grammar
- heatmap-descriptor
- joiner
- operator
- path-expression
- properties-constraint
- range-constraint
- sort-order
- starter
- suggestion-source
- term-definition
- transform-results
- value-constraint

- values-or-tuples
- word-constraint
- word-lexicon
- facet-options
- term-options
- values-options
- geo-options

9.1 How to Use This Reference

This reference describes the layout of the options structure usable with the query capabilities of the *MarkLogic REST API* and MarkLogic Java API, when expressed as JSON. The structures are shown with all possible properties, but not all properties may be usable together.

For specific semantics, refer to the description of each structure. For more details, refer to the options descriptions for the XQuery function `search:search` in the *XQuery and XSLT Reference Guide*.

9.2 Query Options

```
"options": {
  "additional-query": [ string ],
  "concurrency-level": string,
  "constraint": [ constraint ],
  "debug": boolean,
  "default-suggestion-source": [ suggestion-source ],
  "extract-metadata": extract-metadata,
  "forest": unsigned long,
  "fragment-scope": string,
  "grammar": grammar,
  "operator": [ operator ],
  "page-length": unsigned long,
  "quality-weight": double,
  "return-aggregates": boolean,
  "return-constraints": boolean,
  "return-facets": boolean,
  "return-frequencies": boolean,
  "return-metrics": boolean,
  "return-plan": boolean,
  "return-qtext": boolean
  "return-query": boolean,
  "return-results": boolean,
  "return-similar": boolean,
  "return-values": boolean,
  "search-option": [ string ],
  "searchable-expression": { path-expression }
  "sort-order": [ sort-order ],
```

```
  "suggestion-source": [ suggestion-source ],
  "term": term-definition,
  "transform-results": transform-results,
  "tuples": values-or-tuples,
  "values": values-or-tuples
}
```

Key	Value	Description

additional-query[]	[*string*]	Additional serialized `cts:query`'s, as XML literals. The queries are AND'd with the string and/or structured query that is supplied to the request. The query results are constrained by the `additional-query`, but any terms matching the addtional-query are not highlighted in the snippet result output. For example, the following constrains the results to the directory named /my/directory/:
		`"additional-query":[`
		`"<directory-query`
		`xmlns='http://marklogic.com/cts'>`
		`<uri>/my/dir/</uri>`
		`</directory-query>"`
		`]`

concurrency-level	*unsigned integer*	The maximum number of threads used to resolve facets. The default is 8, which specifies that at most 8 threads will be used concurrently to resolve facets.
constraint[]	[constraint]	Constraints define facets which can be returned as part of the search results. The default is no defined constraints. Each constraint must have an name, which is the unique name of this constraint. The name can be used in the search grammar to specify the constraint. The list can contain zero or more constraint objects.
debug	*boolean*	Activates debugging mode. Additional report elements will be present in the output. Set to `true` to activate. The default is `false`.

Key	Value	Description
default-suggestion-source[]	[suggestion-source]	Defines the content to be used as the default source of suggestions (see `search:suggest`). The source may be expressed as a reference to an existing named constraint, or as a collection, value, word or word-lexicon element. Note that the use of `word-lexicon` (the database-wide word lexicon) is not recommended as best practice; collection and range lexicons will yield the best performance. Each default-suggestion-source object can optionally have a collation value, which specifies the collation of the value lexicon used during query evaluation. If no collation is specified, then the query uses the default collation for the context in which the query is evaluated.
extract-metadata	extract-metadata	Specifies data to return in the `metadata` section of the `response` output. You can specify element or attribute content from the search hits to be returned or values from any constraints in your search.
forest[]	[*unsigned long*]	One or more forest IDs.

Key	Value	Description
fragment-scope	*string*	Allowable values: • properties • documents (default) Controls the global fragment scope over which to search. The global scope applies both to what the search returns (that is, if it returns results from document fragments or from property fragments) and the global scope is inherited by any constraints that do not explicitly override the `fragment-scope`. You can override the global fragment scope by setting a local fragment scope inside a constraint definition. For example, a `fragment-scope` of properties on a range-constraint enables you to facet on a value stored in a property, even if you are searching over documents.
grammar	grammar	A custom search grammar definition. The default grammar defines "Google-style" parsing.
operator[]	[operator]	A list of state elements, each representing a unique run-time configuration option. For example, if an operator with the name "sort" is defined, query text [sort:foo] will select the `state` child with the name "foo" at query runtime, using the option specified on that state element. Options affecting query parsing (such as `constraint`, `grammar`, `term`, `empty`) may not be configured via operators.
page-length	*unsigned long*	Specifies the number of results per page. The default value is 10.
quality-weight	*double*	Specifies a a weighting factor to use in the query. The default value is 1.0.
return-aggregates	*boolean*	Include the result of running a builtin or user-defined aggregate function. Applies only to queries against values or tuples.

Key	Value	Description
return-constraints	*boolean*	Include original constraint definitions in the results. The default is false.
return-facets	*boolean*	Include resolved facets in the results. The default is true.
return-frequencies	*boolean*	Include frequencies in the results.
return-metrics	*boolean*	Include statistics in the results. The default is true.
return-plan	*boolean*	Include `xdmp:plan` output in the results. The default is false. The output is useful in examining the evaluation plan of the search.
return-qtext	*boolean*	Include the original query text in the results. The default is true.
return-query	*boolean*	Include the XML query representation in the results. The default is false.
return-results	*boolean*	Include search results in the output. (Use transform-results to specify how each result should be formatted.) The default is true.
return-similar	*boolean*	Include with each search result a list of URLs of similar documents in the database. The default is false.
return-values	*boolean*	When querying a range index or values lexicon, whether or not to include the index/lexicon values in the results. Default: true.
search-option[]	[*string*]	For advanced users, one or more options to be passed in to the query operation (for example, `filtered`, `unfiltered`, `score-logtfidf`, and so on). See the XQuery function `cts:search`.

Key	Value	Description
searchable-expression	path-expression	An expression to be searched. Whatever expression is specified is returned from the search. For example, if you specify `//p`, then `p` elements that match the search criteris are returned. The expression must be an inline fully searchable XPath expression, and all necessary namespaces must be declared. The default value is `fn:collection()`, which searches all documents in the database. This option does not affect facet results. To constrain facet results, use `additional-query`.
sort-order[]	[sort-order]	Set the default sort order. The first such value is the primary sort order, the second secondary sort order, and so on. The default is to sort by score, descending. Note that the default scoring algorithm can be set just like any other option with the option named `search-option`.

Key	Value	Description
suggestion-source	[suggestion-source]	Specifies a constraint source to override a named constraint when using search:suggest. The suggestions are often used for type-ahead suggestions in a search user interface. If empty, no suggestions are generated when that constraint is applied. Specifying an alternate suggestion-source is useful in cases where you have a named constraint to use for searching and facets, but you might want to use a slightly (or completely) different source for type-ahead suggestions without needed to re-parse your search terms. Each suggestion source must have a `ref` property corresponding to a named constraint (one suggestion-source per named constraint).
transform-results	transform-results	Specifies a function to use to process a search result for the snippet output. The default is that each result is formatted using the built-in default snippetting function.
tuples	values-or-tuples	Defines one or more value lexicons query against, matching value co-occurrences. That is, tuples of values, each of which appear in the same fragment.
values	values-or-tuples	Defines one or more value lexicons to query against.

9.3 attribute-descriptor

```
{
  "name": string,
  "ns": string
}
```

The definition of an XML element attribute. An attribute descriptor is generally accompanied by an element descriptor to identify the containing element.

Key	Value	Description
name	*string*	Required. The name of an attribute.
ns	*string*	The namespace in which the attribute is defined.

Example:

```
{ "name": "bar", "ns": "http://marklogic.com/examples" }
```

9.4 bucket

```
{
  "name": string,
  "label": string,
  "lt": atomic-value,
  "ge": atomic-value
}
```

A named range of static values within a range constraint. For dynamic value ranges see computed-bucket. Values assigned to a bucket meet the criteria:

```
bucket.lt < value <= bucket.gt
```

Key	Value	Description
name	*string*	Required. Identifies this bucket.
label	*string*	Text that can be used to label the bucket. Corresponds to the text() in an XML `<bucket/>` definition.
lt	*atomic-value*	The lower bound for values assigned to this bucket.
ge	*atomic-value*	The upper bound for values assigned to this bucket.

Example:

```
{
  "options": {
    "constraint": [
      {
        "name": "decade",
        "range": {
```

```
            "type": "xs:gYear",
            "facet": true,
            "facet-option": [
              "limit=10"
            ],
            "bucket": [
              {
                "lt": "1930",
                "ge": "1920",
                "name": "1920s",
                "label": "1920s"
              },
              {
                "lt": "1940",
                "ge": "1930",
                "name": "1930s",
                "label": "1930s"
              }
            ],
            "attribute": {
              "ns": "",
              "name": "year"
            },
            "element": {
              "ns": "http:\/\/marklogic.com\/wikipedia",
              "name": "nominee"
            }
          }
        }
    ]
  }
}
```

9.5 collection-constraint

```
{
  "prefix": string,
  "facet-option": [ string ],
  "facet": boolean
}
```

Key	Value	Description
prefix	*string*	Required. Constrain a query to documents in collections matching this prefix.
facet-option[]	[*string*]	Zero or more facet-options.
facet	*boolean*	Whether or not to generate facets from this constraint.

The following example defines a collection constraint called “my-collection” that constrains a query to documents in collections with the prefix “/my/path/”:

```
{
  "options": {
    "constraint": [
      {
        "name": "my-collection",
        "collection": {
          "prefix": "\/my\/path\/"
        }
      }
    ]
  }
}
```

9.6 computed-bucket

```
{
  "name": string,
  "label: string,
  "lt": atomic-value,
  "ge": atomic-value,
  "anchor": string,
  "lt-anchor": string,
  "ge-anchor": string
}
```

A named range of dynamic temporal values within a range constraint. For static value ranges see bucket. Values assigned to a bucket meet the criteria:

```
(bucket.anchor + bucket.lt) < value <= (bucket.anchor + bucket.gt)
```

Use `lt-anchor` and `ge-anchor` to override the anchor value used to compute the lower and upper bounds, respectively.

Key	Value	Description
name	*string*	Required. The name of this bucket.
label	*string*	Text that can be used to label the bucket. Corresponds to the text() in an XML `<bucket/>` definition.
lt	*atomic-value*	The lower bound for values assigned to this bucket, relative to `anchor` or `lt-anchor`. The value should be an `xs:duration`, `xs:yearMonthDuration`, or `xs:dayTimeDuration`.
ge	*atomic-value*	The lower bound for values assigned to this bucket, relative to `anchor` or `ge-anchor`. The value should be an `xs:duration`, `xs:yearMonthDuration`, or `xs:dayTimeDuration`.
anchor	*string*	A dynamic temporal anchor value for the bucket. The bucket bounds are relative to this dynamic value. Allowed values: • now • start-of-day • start-of-month • start-of-year
lt-anchor	*string*	Overrides `anchor` for computing the bucket lower bound. For allowed values, see `anchor`.
ge-anchor	*string*	Overrides `anchor` for computing the bucket upper bound. For allowed values, see `anchor`.

Example:

```
{
  "options": {
    "constraint": [
      {
        "name": "made",
        "range": {
          "type": "xs:dateTime",
          "computed-bucket": [
            {
              "name": "today",
              "ge": "P0D",
              "lt": "P1D",
              "anchor": "now",
              "label": "Today"
            },
            {
              "name": "30-days",
              "ge": "-P30D",
```

```
            "lt": "P1D",
            "anchor": "now",
            "label": "Last 30 days"
          },
          {
            "name": "60-days",
            "ge": "-P60D",
            "lt": "P1D",
            "anchor": "now",
            "label": "Last 60 Days"
          },
          {
            "name": "year",
            "ge": "-P1Y",
            "lt": "P1D",
            "anchor": "now",
            "label": "Last Year"
          }
        ],
        "element": {
          "ns": "http:\/\/example.com",
          "name": "manufactured"
        },
        "attribute": {
          "ns": "",
          "name": "date"
        }
      }
    }
  ]
 }
}
```

9.7 constraint

```
{
  "name": string,
  "annotation": [ string ],
  "range": range-constraint,
  "word": word-constraint,
  "value": value-constraint,
  "collection": collection-constraint,
  "properties": properties-constraint,
  "element-query": element-descriptor,
  "custom": custom-constraint,
  "geo-elem": geo-elem-constraint,
  "geo-elem-pair": geo-elem-pair-constraint,
  "geo-attr-pair": geo-attr-pair-constraint
}
```

Every constraint must have a unique `name` and may have an annotation. The constraint semantics are expressed through one or more of the remaining constraint object types (range, word, value, etc.).

Key	Value	Description
name	*string*	Required. The name must be unique within these query options and may not contain whitespace.
annotation	[*string*]	Your comments. Annotations have no effect.
range	range-constraint	Specifies the element or attribute on which to constrain by range. There must be a range index of the specified type (and collation for string range indexes) defined.
word	word-constraint	Specifies the element, attribute, or field on which to constrain by word. You cannot create facets from a word constraint.
value	value-constraint	Specifies element or attribute values on which to constrain by value. You cannot create facets from a value constraint.
collection	collection-constraint	Specifies a collection which to constrain.
properties	properties-constraint	Specifies a constraint that finds matches on the corresponding properties documents. You cannot create facets from a properties constraint.
element-query	element-descriptor	Specifies a constraint that restricts the search to the specified element. You cannot create facets from an element-query constraint.

Key	Value	Description
custom	custom-constraint	Specifies a custom constraint along with the name of the function implementations used to evaluate the custom constraint.
geo-elem	geo-elem-constraint	Specifies a geospatial constraint. If the constraint has a `heatmap` child, then it can be used as a facet, and can contain `facet-options`.
geo-elem-pair	geo-elem-pair-constraint	Specifies a geospatial constraint. If the constraint has a `heatmap` child, then it can be used as a facet, and can contain `facet-options`.
geo-attr-pair	geo-attr-pair-constraint	Specifies a geospatial constraint. If the constraint has a `heatmap` child, then it can be used as a facet, and can contain `facet-options`.

9.8 custom-constraint

```
{
  "parse": {
    "apply": string,
    "ns": string,
    "at": string
  },
  "start-facet": {
    "apply": string,
    "ns": string,
    "at": string
  },
  "finish-facet": {
    "apply": string,
    "ns": string,
    "at": string
  },
  "facet-option": [ string ],
  "term-option": string,
  "facet": boolean
}
```

Specifies a custom constraint along with the name of the function implementations used to evaluate the custom constraint.

Key	Value	Description
parse	*object*	Required. Identifies the function used to parse the query text to produce a `cts:query`.
parse.apply	*string*	Required. The parsing function localname.
parse.ns	*string*	The namespace in which the function is declared.
parse.at	*string*	The URI of the XQuery library module implementing function.
start-facet	*object*	Identifies a function to make lexicon API calls that return the values and counts used to construct facets.
start-facet.apply	*string*	Required if `start-facet` is defined.
start-facet.ns	*string*	The namespace in which the function is declared.
start-facet.at	*string*	The URI of the XQuery library module implementing function.
finish-facet	*object*	Identifies the function takes input from the `start-facet` function (if it is used) and constructs the facet element.
finish-facet.apply	*string*	Required if `finish-facet` is defined. The function localname.
finish-facet.ns	*string*	The namespace in which the function is declared.
finish-facet.at	*string*	The URI of the XQuery library module implementing function.
facet-option[]	[*string*]	Zero or more facet-options.
term-option[]	[*string*]	Zero or more term-options.
facet	*boolean*	Whether or not to use this constraint for faceting.

9.9 element-descriptor

```
{
  "name": string,
  "ns": string
}
```

This object defines an XML element for use in options such as constraints.

Key	Value	Description
name	*string*	Required. The name of an element.
ns	*string*	The namespace in which the named element is defined.

Example:

```
{ "name": "foo", "ns": "http://marklogic.com/examples" }
```

9.10 extract-metadata

```
{
  "constraint-value": [
    { "ref": string }
  ],
  "qname": [
    {
      "elem-ns": string,
      "elem-name": string,
      "attr-ns": string,
      "attr-name": string
    }
  ]
  "json-key": [ string ]
}
```

Use `extract-metadata` to select one or more metadata elements from each document in the query result set. Selections are made by specifying qnames (element or element and attribute), JSON keys, or constraint references (collection, range and geospatial types only) for extraction from indexes.

Key	Value	Description
constraint-value[]	*array*	An array of references to named constraints defined in these query options. Corresponding indexes must exist when the query executes.
constraint-value.ref	*string*	The name of a collection, range, or geospatial constraint defined in these query options.
qname[]	*array*	An array of QName definitions for element or element attribute content from the search hits to be returned.
qname.elem-name	*string*	The name of an element from which content should be returned. Required (if using qname).
qname.elem-ns	*string*	The namespace URI in which `qname.elem-name` is defined. Required (if using qname)
qname.attr-name	*string*	The name of an attribute of `qname.elem-name` whose value should be returned.
qname.attr-ns	*string*	The namespace URI in which `qname.attr-name` is defined.
json-key[]	[*string*]	The names of json keys from which content should be returned. Applicable only to JSON documents loaded using the *MarkLogic REST API* or MarkLogic Java API.

Example:

```
{
  "options": {

    "extract-metadata": {
      "constraint-value": [
        { "ref": "title" },
        { "ref": "speaker" }
      ],
      "qname": [
        {
          "elem-ns": "http:\/\/example.com",
          "elem-name": "foo"
        }
      ],
      "json-key": [ "foo", "bar" ]
    }
```

```
    "constraint": [
      {
        "name": "title",
        "range": {
          "type": "xs:string",
          "element": {
            "ns": "",
            "name": "TITLE"
          }
        }
      },
      {
        "name": "speaker",
        "range": {
          "type": "xs:string",
          "element": {
            "ns": "",
            "name": "SPEAKER"
          }
        }
      }
    ]
  }
}
```

9.11 field-descriptor

```
{
  "name": string,
  "collation": string
}
```

Key	Value	Description
name	*string*	Required. The name of a field.
collation	*string*	An optional collation URI.

9.12 geo-attr-pair-constraint

```
{
  "parent": element-descriptor,
  "lat": attribute-descriptor,
  "lon": attribute-descriptor,
  "facet-option": [ string ],
  "geo-option": [ string ],
  "heatmap": heatmap-descriptor
}
```

A geospatial constraint modelling a geospatial index with coordinates stored as element attributes. By default coordinates are stored as latitude,longitude points. To reverse the coordinate order, add the geo-option "long-lat-points" to the query options configuration. For more details, see `cts:element-attribute-pair-geospatial-query`.

Key	Value	Description
parent	element-descriptor	Defines the parent element.
lat	*object*	The latitude-defining attribute.
lon	*object*	The longitude-defining attribute.
facet-option[]	[*string*]	Zero or more facet-options.
geo-option[]	[*string*]	Zero or more geo-options.
heatmap	heatmap-descriptor	A heatmap that to use for faceting.

Example:

```
{
  "options": {
    "constraint": [
      {
        "name": "my-geo-attr-pair",
        "geo-attr-pair": {
          "facet-option": [
            "empties"
          ],
          "heatmap": {
            "s": 23.2,
            "w": -118.3,
            "n": 23.3,
            "e": -118.2,
            "latdivs": 4,
            "londivs": 4
          },
          "parent": {
            "ns": "ns1",
            "name": "elem1"
          },
          "lat": {
            "ns": "ns2",
            "name": "attr2"
          },
          "lon": {
            "ns": "ns3",
            "name": "attr3"
```

```
            }
          }
        }
      ]
    }
}
```

9.13 geo-elem-constraint

```
{
  "parent": element-descriptor,
  "element": element-descriptor,
  "facet-option": [ string ],
  "geo-option": [ string ],
  "heatmap": heatmap-descriptor
}
```

A geospatial constraint modelling a geospatial index with coordinates stored in a single element. By default coordinates are stored as latitude,longitude points. To reverse the coordinate order, add the geo-option "long-lat-points" to the query options configuration. For more details, see `cts:element-geospatial-query`.

Key	Value	Description
parent	element-descriptor	The parent element.
element	element-descriptor	The element defining the latitude and longitude.
facet-option[]	[*string*]	Zero or more facet-options.
geo-option[]	[*string*]	Zero or more geo-options.
heatmap	heatmap-descriptor	Describes a heatmap to use for faceting.

Example:

```
{
  "options": {
    "constraint": [
      {
        "name": "geo-elem-child",
        "geo-elem": {
          "parent": {
            "ns": "",
            "name": "g-elem-child-parent"
          },
          "element": {
            "ns": "",
```

```
            "name": "g-elem-child-point"
          }
        }
      }
    ]
  }
}
```

9.14 geo-elem-pair-constraint

```
{
  "parent": element-descriptor,
  "lat": element-descriptor,
  "lon": element-descriptor,
  "facet-option": [ string ],
  "geo-option": [ string ],
  "heatmap": heatmap-descriptor
}
```

A geospatial constraint modelling a geospatial index with coordinates stored in a two elements. By default coordinates are stored as latitude,longitude points. To reverse the coordinate order, add the geo-option "long-lat-points" to the query options configuration. For details, see `cts:element-pair-geospatial-query`.

Key	Value	Description
parent	element-descriptor	The parent element.
lat	element-descriptor	The latitude-defining element.
lon	element-descriptor	The longitude-defining element.
facet-option[]	[*string*]	Zero or more facet-options.
geo-option[]	[*string*]	Zero or more geo-options.
heatmap	heatmap-descriptor	Defines a heatmap to be used for faceting.

9.15 grammar

```
{
  "implict": string,
  "quotation": string,
  "starter": [ starter ],
  "joiner": [ joiner ]
}
```

Specifies a custom search grammar. For details, see Modifying and Extending the Search Parsing Grammar in the *Search Developer's Guide*.

Key	Value	Description
implict	*string*	A `cts:query`, as a serialized string literal, to use by default to join two search terms together. By default, the Search API uses a `cts:and-query`, but you can change it to any `cts:query` with the implicit grammar option.
quotation	*string*	The string to use to indicate the start and end of a phrase.
starter	starter	Zero or more custom starters that denote special parsing for terms.
joiner	[joiner]	Zero or more custom joiners for joining two terms (cat AND dog) or for separating a constraint or operator from its value.

Example:

```
{
  "options": {
    "grammar": {
      "starter": [
        {
          "strength": 30,
          "apply": "grouping",
          "delimiter": ")",
          "label": "("
        }
      ],
      "joiner": [
        {
          "strength": 10,
          "apply": "infix",
          "cts-element": "cts:or-query",
          "tokenize": "word",
          "label": "OR"
        }
      ],
      "quotation": "\"",
      "implicit": "<cts:and-query strength=\"20\"
xmlns=\"http:\/\/marklogic.com\/appservices\/search\"
xmlns:cts=\"http:\/\/marklogic.com\/cts\"\/>"
    }
  }
}
```

9.16 heatmap-descriptor

```
{
  "n": double,
  "s": double,
  "e": double,
  "w": double,
  "latdivs": number,
  "londivs": number
}
```

Models a two-dimensional grid used to categorize data along two dimenstions. Use with geospatial indexes.

Key	Value	Description
n	*double*	Required. The north coordinate.
s	*double*	Required. The south coordinate.
e	*double*	Required. The east coordinate.
w	*double*	Required. The west coordinate.
latdivs	*unsigned integer*	Required. The number of latitude divisions.
londivs	*unsigned integer*	Required. The number of longitude divisions.

9.17 joiner

```
{
  "apply": string,
  "ns": string,
  "at": string,
  "label": string,
  "strength": integer,
  "element": string,
  "options": string,
  "tokenize": string,
  "compare": string,
  "consume": integer
}
```

Key	Value	Description
label	*string*	Required. The joiner operator token. For example, the default grammar includes a joiner with the label “AND”, which enables queries like “cat AND dog”.
apply	*string*	The local-name of a function to use to apply the joiner.
ns	*string*	The namespace in which the `apply` function is defined, if using a user-defined function.
at	*string*	The module from which the `apply` function is imported, if using a user-defined function.
strength	*integer*	Required. The order of precedence of this joiner over others. Higher strength tokens or groups are processed before lower ones.
element	*string*	A `cts:query` element name (for example, `cts:and-query`).
options	*string*	A space-separated list of options that are passed through to the underlying `cts:query` constructor
tokenize	*string*	Allowed values: “word” or “default”.
compare	*string*	The comparison operator used in range queries. Allowed values: LT, LE, GT, GE, NE
consume	*integer*	The number of tokens to consume when parsing N-token joiners. For example, in the expression “A NEAR B”, one token is consumed, but in the expression “A NEAR/2 B” two tokens are consumed.

Example:

```
{
  "options": {
    "grammar": {
      "joiner": [
        {
          "strength": 10,
          "apply": "infix",
          "cts-element": "cts:or-query",
```

```
            "tokenize": "word",
            "label": "OR"
          }
        ]
      }
    }
  }
```

9.18 operator

```
{
  "name": string,
  "state": [
    {
      "name": string,
      "additional-query": string,
      "annotation": [ string ],
      "debug": boolean,
      "forest": unsigned long,
      "page-length": unsigned long,
      "quality-weight": double,
      "search-option": [ string ],
      "searchable-expression": path-expression,
      "sort-order": [ sort-order ],
      "transform-results": transform-results
    }
  ]
}
```

A named wrapper for one or more state elements, each representing a unique run-time configuration option. For example, if an operator with the name "sort" is defined, query text [sort:foo] will select the state child with the name "foo" at query runtime, using the option specified on that state element. Options affecting query parsing (such as `constraint`, `grammar`, `term`, `empty`) may not be configured via operators.

Key	Value	Description
name	*string*	Required. The name of this operator. The name must be unique across all operators and constraints
state[]	*array*	One or more state definitions.
state.name	*string*	Required. The name of this state.
state.additional-query	*string*	A query to perform when the operator is encountered.

Key	Value	Description
state.annotation	[*string*]	Your comments. Annotations have no effect.
state.debug	*boolean*	Whether or not to enable debug logging. Default: false.
state.forest	*unsigned long*	A forest id.
state.page-length	*unsigned long*	The page length into which results are chunked.
state.quality-weight	*double*	Specifies a a weighting factor to use in the query. The default value is 1.0.
state.search-option[]	[*string*]	Zero or search options.
state.searchable-expression	path-expression	A query to perform when the operator is encountered.
state.sort-order[]	[sort-order]	Sort order rules to apply when the operator is encountered in a query.
state.transform-results	transform-results	Transformations to apply to the results produced by this operator.

The following example defines 2 operators, named “sort” and “page-length”. The “sort” operator has 2 states, “down” and “up”. The “page-length” operator has 2 states, “short” and “long”. This allows you, for example, to include a search term of the form `sort:up` to request results in ascending order.

```
{
  "options": {
    "operator": [
      {
        "name": "sort",
        "state": [
          {
            "name": "down",
            "sort-order": [{
                "direction": "descending",
                "score": null
              }]
          },
          {
            "name": "up",
            "sort-order": [{
                "direction": "ascending",
                "score": null
              }]
          }]
```

```
      },
      {
        "name": "page-length",
        "state": [
          {
            "name": "short",
            "page-length": 10
          },
          {
            "name": "long",
            "page-length": 50
          }]
      }]
  }
}
```

9.19 path-expression

```
{
  "text": string,
  "namespaces": { prefix-list }
}
```

Key	Value	Description
text	*string*	Required. The XPath expression. If the expression uses namespace prefixes, the prefixes must be defined in the `namespaces` property. Path elements with no prefix are assumed to be in no namespace.
namespaces	*prefix-list*	Optional. A list of `prefix:uri` pairs, defining any namespace prefixes used in the `text` property of this expression. See the example below.

The following example demonstrates using the path-expression “a:foo/b:bar/qux” as the definition of a searchable-expression:

```
{
  "options": {
    "searchable-expression": {
      "text": "\/a:foo\/b:bar\/qux",
      "namespaces": {
        "a": "http:\/\/a",
        "b": "http:\/\/b"
      }
    }
  }
}
```

9.20 properties-constraint

```
{ "properties": null }
```

Constrains a search to matches found in document properties.

Example:

```
{
  "options": {
    "constraint": [
      { "name": "just-props", "properties": null" }
    ]
  }
}
```

9.21 range-constraint

```
{
  "type": string,
  "element": element-descriptor,
  "attribute": attribute-descriptor,
  "field": field-descriptor,
  "json-key": string,
  "path-index": path-expression,
  "fragment-scope": string,
  "bucket": [ bucket ],
  "computed-bucket": [ computed-bucket ],
  "facet-option" : [ string ]
  "collation" : string,
  "facet": boolean
}
```

Specifies an element, element attribute, field, JSON key, or XPath expression on which to constrain by range. The constraint must include one of `element`, `field`, `json-key`, or `path-index`. If there is an `element`, an `attribute` may also be included.

There must be a range index of the specified type (and collation for string range indexes) defined for the specified element, attribute, field, or path.

For range constraints with either bucket or computed-bucket specifications, for maximum performance and sortability, the buckets should be in a continuous order; if the order is not continuous (either ascending or descending), then the buckets are returned in the order specified, regardless of any sorting facet-option in the specification.

Key	Value	Description
type	*string*	Required. The XQuery type of the values in the range. There database configuration must include a range index with this value type.
element	element-descriptor	Defines an XML element to constrain by or the element containing `attribute`. If there is an `element`, there can also be an `attribute`. There must not be a `field`, `json-key`, or `path-index`. The database configuration must include a corresponding element or element attribute range index.
attribute	attribute-descriptor	Defines an XML element attribute to constrain by. There must be an accompanying element property. There must not be a `field`, `json-key`, or `path-index`. The database configuration must include a corresponding attribute range index.
field	field-descriptor	Defines field to constrain by. There must be an accompanying element property. There must not be a `element, attribute`, `json-key`, or `path-index`. The database configuration must include a corresponding field range index.
json-key	*string*	Defines JSON key to constrain by. There must be an accompanying element property. There must not be an `element, attribute`, `field`, or `path-index`. JSON key constraints apply only to JSON documents loaded using the *MarkLogic REST API* or the MarkLogic Java API.
path-index	path-expression	Defines an XPath expression to constrain by. There must be an accompanying element property. There must not be a `element, attribute`, `json-key`, or `field`. The database configuration must include a corresponding path range index.

Key	Value	Description
fragment-scope	*string*	Allowable values: • properties • documents (default) Set a local fragment scope for this constraint. The local fragment-scope overrides a global fragment-scope. For example, a fragment-scope of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents.
bucket[]	[bucket]	Zero or more named ranges of static values.
computed-bucket[]	[computed-bucket]	Zero or more named ranges of dynamic values.
facet-option[]	[*string*]	Zero or more facet-options.
collation	*string*	A collation URI, if `type` is `xs:string`.
facet	*boolean*	Whether or not to include facets.

Example:

```
{
  "options": {
    "constraint": [
      {
        "name": "size",
        "range": {
          "type": "xs:unsignedInt",
          "bucket": [
            {
              "name": "small",
              "lt": "5",
              "label": "Under 5"
            },
            {
              "name": "medium",
              "ge": "5",
              "lt": "12",
              "label": "5 to 12"
            },
            {
              "name": "large",
              "ge": "12",
              "label": "12+"
            }
          ],
          "element": {
            "name": "shoe",
```

```
            "ns": "http:\/\/implementation-specific.com"
          },
          "attribute": {
            "name": "size",
            "ns": ""
          }
        }
      }
    ]
  }
}
```

9.22 sort-order

```
{
  "element": element-descriptor,
  "attribute": attribute-descriptor,
  "field": field-descriptor,
  "json-key": string,
  "score": string,
  "annotation": [ string ],
  "type": string,
  "collation": string,
  "direction": string
}
```

Defines the sort order of an element, element attribute, field, or JSON property. A sort-order object must have one of a single element child, a single score child, a single field child, or a single json-key child.

If there is an element child it can optionally have an attribute sibling (to specify an attribute of the preceding element). Both the element and attribute elements must have `ns` and `name` attributes to specify the namespace and local-name of the specified element and attribute.

You should not apply `sort-order` to a field that has more than one included element.

If there is a score child, it specifies to sort by the score of the search result.

Key	Value	Description
element	element-descriptor	Identifies the element to which this sort order applies, or the element containing the attribute to which this sort order applies. If specifying only an element, you must set `type` to match the range index type of this element. If an element property is present, you cannot also specify `field` or `json-key`.
attribute	attribute-descriptor	Identifies the element attribute to which this sort order applies. You must set `type` to match the range index type of this attribute. If an `attribute` property is present, you cannot also specify `field` or `json-key`, and you must specify `element`.
field	field-descriptor	Identifies the field to which this sort order applies. Cannot be used with `element`, `attribute` or `json-key`.
json-key	*string*	Identifies the JSON property to which this sort order applies. This applies only to matches in JSON documents loaded using the *MarkLogic REST API*. Cannot be used with `element`, `attribute` or `field`.
score	*string*	Allowed values: • null • score-logtfidf • score-logtf • score-simple • score-random
annotation[]	[*string*]	Your comments. Annotations have no effect.
type	*string*	If you are sorting by an element or an attribute, you must specify a type property with a value corresponding to the range index type of that element or attribute (for example, `xs:string`, `xs:dateTime`, and so on).
collation	*string*	A collation URI. Optionally specify a collation when-sort-order.type is xs:string; otherwise, use the collation of the query.
direction	*string*	Specify the sorting direction. Default: `descending`. Allowed values: • ascending • descending

For example, the following specifies a primary sort order using the element value for `my-element` (which needs a string range index with the specified collation), and a secondary sort order of score ascending:

```
{
  "options": {
    "sort-order": [
      {
        "direction": "descending",
        "type": "xs:string",
        "collation": "http://marklogic.com/collation/",
        "element": {
          "name": "my-element",
          "ns": "my-namespace",
        },
        "annotation": [ "some user comment can go here" ] },
      {
        "direction": "ascending",
        "score": null
      }
    ]
  }
```

9.23 starter

```
{
  "label": string,
  "apply": string,
  "ns": string,
  "at": string,
  "strength": integer,
  "element": string,
  "options": string,
  "tokenize": string,
  "delimiter": string
}
```

A `starter` is a component of a customer search grammar that specifies the text to use to delimit and group terms.

Key	Value	Description
label	*string*	Required. The joiner operator token. For example, the default grammar includes a joiner with the label “AND”, which enables queries like “cat AND dog”.
apply	*string*	The local-name of a function to use to apply the starter.
ns	*string*	The namespace in which the `apply` function is defined, if using a user-defined function.
at	*string*	The module from which the `apply` function is imported, if using a user-defined function.
strength	*integer*	Required. The order of precedence of this starter over others. Higher strength tokens or groups are processed before lower ones.
element	*string*	A `cts:query` element name (for example, `cts:and-query`).
options	*string*	A space-separated list of options that are passed through to the underlying `cts:query` constructor
tokenize	*string*	Allowed values: “word” or “default”.
delimiter	*string*	The string to use as a delimiter for the starter.

Example:

```
{
  "options": {
    "grammar": {
      "starter": [
        {
          "strength": 30,
          "apply": "grouping",
          "delimiter": ")",
          "label": "("
        },
        {
          "strength": 40,
          "apply": "prefix",
          "cts-element": "cts:not-query",
          "label": "-"
        }
```

```
        ]
      }
    }
  }
```

9.24 suggestion-source

```
{
  "ref": string,
  "suggestion-option": [ string ]
  "annotation": [ string ]
  "range": range-constraint,
  "word": word-constraint,
  "value": value-constraint,
  "collection": collection-constraint,
  "properties": properties-constraint,
  "custom": custom-constraint,
  "geo-elem": geo-elem-constraint,
  "geo-elem-pair": geo-elem-pair-constraint,
  "geo-attr-pair": geo-attr-pair-constraint
  "word-lexicon": word-lexicon
}
```

Key	Value	Description
ref	*string*	The name of a constraint defined by the query options.
suggestion-option	[*string*]	Query options to use in generating suggestions, as name-value pairs.
annotation	[*string*]	Your comments. Annotations have no effect.
range	range-constraint	Defines a constraint that overrides the one named by `ref` when generating search suggestions.
word	word-constraint	Defines a constraint that overrides the one named by `ref` when generating search suggestions.
collection	collection-constraint	Defines a constraint that overrides the one named by `ref` when generating search suggestions.
word-lexicon	word-lexicon	Defines a constraint that overrides the one named by `ref` when generating search suggestions.

9.25 term-definition

Specifies handling of empty searches and controls options for how individual terms (that is, terms not associated with a constraint) will be represented when parsing the search.

```
{
  "term-option": [ string ],
  "default": {
    "ref": string
    "word": word-constraint,
    "value": value-constraint,
    "range": value-constraint,
    "annotation": [ string ],
  },
  "empty": {
    "apply": string,
    "ns": string,
    "at": string
  },
  "weight": double,
  "apply": string,
  "ns": string,
  "at": string
}
```

Key	Value	Description
term-option[]	[*string*]	Zero or more term-options.
default	*object*	Specify a default property which determines special handling to all terms, so you can have all terms apply a set of rules such as query weighting, specifying a element or attribute value or word query to be used as a default, or specifying a constraint to be applied by default.
default.ref	*string*	A reference to a constraint defined elsewhere in the query options.
default.range	range-constraint	A range constraint to apply to individual terms without other constraints.
default.value	value-constraint	A value constraint to apply to individual terms without other constraints.
default.word	word-constraint	A word constraint to apply to individual terms without other constraints.
default.annotation	[*string*]	Your comments. Annotations have no effect.

Key	Value	Description
apply	*string*	The name of a custom function to control individual term handling. If you specify a custom function name, you must also specify `term.ns` and `term.at`.
ns	*string*	Only required when using `term.apply`. The namespace of the function library module containing your function.
at	*string*	Only required when using `term.apply`. The path to the module containing your function.
empty	*object*	To control how empty searches (that is, the empty string passed into `search:search`) are resolved, specify an empty child element with an apply attribute. The value of the apply attribute specifies the behavior for empty searches: A value of `all-results` specifies that empty searches return everything in the database, a value of `no-results` (the default) specifies that an empty search returns nothing. Additionally, you create your own function to handle empty searches. To specify your own function, create a function that returns a `cts:query` and specify the local-name of your function, along with the namespace of the function library module, and the path to the module.
empty.apply	*string*	Allowed values: • all-results • no-results • your own function name When specifying your own function, you must also specify `empty.ns` and `empty.at`.
empty.ns	*string*	Only required when `empty.apply` is a custom function. The namespace of the function library module containing your function.
empty.at	*string*	Only required when `empty.apply` is a custom function. The path to the module containing your function.
weight	*double*	A weighting factor to apply to individual terms. Default: 1.0.

An example of using a custom function custom code:

```
{
  "options": {
```

```
    "term": {
      "apply": "my-term-query",
      "ns": "http:\/\/custom\/search",
      "at": "\/lib\/custom.xqy",
      "empty": {
        "apply": "all-results"
      }
    }
  }
}
```

An example parsing to a weighted element word query on element "foo:bar":

```
{
  "options": {
    "term": {
      "default": {
        "word": {
          "element": {
            "ns": "foo",
            "name": "bar"
          },
          "weight": 2
        }
      },
      "empty": {
        "apply": "all-results"
      }
    }
  }
}
```

9.26 transform-results

```
{
  "apply": string,
  "ns": string,
  "at": string,
  "per-match-tokens": unsigned integer,
  "max-matches": unsigned integer,
  "max-snippet-chars": unsigned integer,
  "preferred-elements": [ element-descriptor ],
  "user-defined-param": "param-value"
}
```

Key	Value	Description
apply	*string*	The local-name of a function to use to process a search result for the snippet output. Default: “snippet”. There are three other built-in snippetting functions whose behavior you can affect: "raw" (return the whole node), "empty-snippet" (return an empty snippet), and "metadata-snippet" (return the snippet from the specified element in the properties document).
ns	*string*	The namespace in which the `apply` function is defined, if using a user-defined function.
at	*string*	The module from which the `apply` function is imported, if using a user-defined function.
per-match-tokens	*unsigned integer*	Usable with the builtin snippeting functions.
max-matches	*unsigned integer*	Usable with the builtin snippeting functions.
max-snippet-chars	*unsigned integer*	Usable with the builtin snippeting functions.
preferred-elements	[element-descriptor]	Usable with the builtin snippeting functions.
user-defined-param	*value*	Pass parameters to your snippeting function by including *`param-name`*`:value pairs`. For example, to pass in a value of 1 for the “limit” parameter of your function: “limit”:1.

For example, to modify the default snippeting function, “snippet”:

```
{
  "options": {
    "transform-results": {
      "apply": "snippet",
      "per-match-tokens": "30",
      "max-matches": "4",
      "max-snippet-chars": "200",
      "preferred-elements": ""
    }
  }
}
```

To define a custom transformation using your own snippeting function, “my-snippet” and pass it two parameters:

```
{
  "options": {
    "transform-results": {
      "apply": "my-snippet",
      "ns": "my-namespace",
      "at": "/my-snippet.xqy",
      "my-param1": "42",
      "my-param2": "abc"
    }
  }
}
```

9.27 value-constraint

```
{
  "json-key": string,
  "element": element-descriptor,
  "attribute": attribute-descriptor,
  "field": field-descriptor,
  "weight": double,
  "fragment-scope": string,
  "term-option": string
}
```

Defines elements, element attributes, fields, and JSON keys on which to constrain values. You cannot create facets from a value constraint.

Key	Value	Description
json-key	*string*	A JSON key on which to constrain queries. Only applicable to JSON documents ingested using the *MarkLogic REST API* or the MarkLogic JSON API.
element	element-descriptor	An element on which to constrain queries, or the element containing `attribute` when constraining by element attribute.
attribute	attribute-descriptor	An element attribute on which to constrain queries. The constraint must also contain an `element` if there is an `attribute`.

Key	Value	Description
field	field-descriptor	A field on which to constrain queries.
weight	*double*	Higher weights move search results up in the relevance order. The default is 1.0. The weight should be less than or equal to the absolute value of 16 (between -16 and 16); weights greater than 16 will have the same effect as a weight of 16. Weights less than the absolute value of 0.0625 (between -0.0625 and 0.0625) are rounded to 0, which means that they do not contribute to the score.
fragment-scope	*string*	Allowable values: • properties • documents (default) Set a local fragment scope for this constraint. The local fragment-scope overrides a global fragment-scope. For example, a fragment-scope of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents.
term-option	*string*	Zero or more term-options.

Example:

```
{
  "options": {
    "constraint": [
      {
        "name": "my-value",
        "value": {
          "element": {
            "ns": "my-namespace",
            "name": "my-localname"
          }
        }
      }
    ]
  }
}
```

9.28 values-or-tuples

```
{
  "name": string,
  "uri": null,
  "annotation": [ string ],
```

```
    "collection": collection-constraint,
    "range": range-constraint,
    "field": field-descriptor,
    "geo-elem": geo-elem-constraint,
    "geo-elem-pair": geo-elem-pair-constraint,
    "geo-attr-pair": geo-attr-pair-constraint,
    "aggregate": {
      "apply": string,
      "udf": string
    },
    "values-option": [ string ]
  }
```

Key	Value	Description
name	*string*	Required. The name that identifies this values or tuples definition.
annotation	[*string*]	Your comments. Annotations have no effect.
uri	null	Specifies the database URI lexicon.
collection	collection-constraint	A collection constraint with which to extract values from the database collection lexicon.
range	range-constraint	A range constraint with which to extract values from a range index or value lexicon.
field	field-descriptor	A field constraint with which to extract values from a field index or value lexicon.
geo-elem	geo-elem-constraint	A geospatial constraint with which to extract values from a geospatial index.
geo-elem-pair	geo-elem-pair-constraint	A geospatial constraint with which to extract values from a geospatial index.
geo-attr-pair	geo-attr-pair-constraint	A geospatial constraint with which to extract values from a geospatial index.

Key	Value	Description
aggregate	*object*	Defines a builtin or user-defined aggregate function to apply to lexicon values or value co-occurrences.
aggregate.apply	*string*	Required. The name of a builtin aggregate or aggregate UDF.
aggregate.udf	*string*	The path to the native plugin library containing the implementation of the user-defined extension aggregate. Omit when using a builtin aggregate.
values-option[]	[*string*]	Zero or more values-options.

Example:

```
{
  "options": {
    "values": [
      {
        "name": "speaker",
        "range": {
          "type": "xs:string",
          "element": {
            "ns": "",
            "name": "SPEAKER"
          }
        }
      }
    ]
  }
}
```

9.29 word-constraint

```
{
  "json-key": string,
  "element": element-descriptor,
  "attribute": attribute-descriptor,
  "field": field-descriptor,
  "weight": double,
  "fragment-scope": string,
  "term-option": string,
  "collation": string
}
```

Constrains by a word query of JSON key, element, element attribute, or field. You cannot facet on word constraints.

Key	Value	Description
json-key	*string*	Constrain by JSON key. Mutually exclusive of `element`, `attribute`, and `field`.
element	element-descriptor	Constrain by JSON key. Mutually exclusive of `json-key` and `field`.
attribute	attribute-descriptor	Constrain by JSON key. Mutually exclusive of `json-key` and `field`. An `element` is also required.
field	field-descriptor	Constrain by JSON key. Mutually exclusive of `element`, `attribute`, and `field`.
weight	*double*	Higher weights move search results up in the relevance order. The default is 1.0. The weight should be less than or equal to the absolute value of 16 (between -16 and 16); weights greater than 16 will have the same effect as a weight of 16. Weights less than the absolute value of 0.0625 (between -0.0625 and 0.0625) are rounded to 0, which means that they do not contribute to the score.
fragment-scope	*string*	Allowable values: • properties • documents (default) Set a local fragment scope for this constraint. The local fragment-scope overrides a global fragment-scope. For example, a fragment-scope of `properties` on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents.
term-option	*string*	Zero or more term-options.
collation	*string*	A collation URI.

9.30 word-lexicon

```
{
  "fragment-scope": string,
  "collation": string
}
```

Key	Value	Description
fragment-scope	*string*	Allowable values: • properties • documents (default)
collation	*string*	A collation URI. If no collation is specified, then the query uses default collation for the context in which the query is evaluated.

9.31 facet-options

Facet options are specified as properties of range and other constraints that can be used as a facet (any constraints except word, value, element-query, or property).

Legal values for a facet-option property are generally any option that can be passed into the underlying query or lexicon API. The following list enumerates the facet-option values, but be aware that some options are only available with some range types. For more detail on these options, see the documentation for the underlying range or lexicon APIs

- ascending
- descending
- empties
- any
- document
- properties
- locks
- frequency-order
- item-order
- fragment-frequency
- item-frequency
- type=type
- timezone=TZ
- limit=N
- sample=N

- truncate=N
- skip=N
- score-logtfidf
- score-logtf
- score-simple
- score-random
- checked
- unchecked
- concurrent
- map

For example:

```
{
  "options": {
    "constraint": [
      {
        "name": "color",
        "range": {
          "type": "xs:string",
          "facet-option": [
            "item-frequency"
          ],
          "element": {
            "ns": "",
            "name": "bodycolor"
          }
        }
      }
    ]
  }
}
```

9.32 term-options

Specifies the options passed into the query (for example, `case-insensitive`). By default, the query uses the same default options as the underlying cts:query constructors, and the defaults change based on your index configuration.

You can use term options in several places, including as a property of a term-definition and within many constraints. Within constraints, term options typically occur as property the type of constraint (for example, inside a word-constraint). You can use the `/config/query` service to validate a set of options.

Legal term options values are:

- case-sensitive
- case-insensitive
- diacritic-sensitive
- diacritic-insensitive
- punctuation-sensitive
- punctuation-insensitive
- whitespace-sensitive
- whitespace-insensitive
- stemmed
- unstemmed
- wildcarded
- unwilcarded
- exact
- lang=*iso639code*

For example

```
{
  "options": {
    "term": {
      "apply": "term",
      "empty": {
        "apply": "all-results"
      },
      "term-option":[
        "diacritic-insensitive",
        "unwildcarded"
      ]
    }
  }
}
```

9.33 values-options

Options that can be used in defining values and tuples. For a description of these options, see `cts:values` and `cts:values-tuples` in the *XQuery and XSLT Reference Guide*. Some options can only be used when defining a values configurations; others can only be used when defining tuples.

- ascending
- descending
- any

- document
- properties
- locks
- frequency-order
- item-order
- fragment-frequency
- item-frequency
- timezone=*TZ*
- limit=*N*
- skip=*N*
- sample=*N*
- truncate=*N*
- score-logtfidf
- score-logtf
- score-simple
- score-random
- score-zero
- checked
- unchecked
- concurrent
- map (for values only)
- ordered (for tuples only)
- proximity=*N* (for tuples only)

9.34 geo-options

Options that can be used when defining geo constraints. The options usable in a given geo constraint depend on the type of constraint. See the constraint for limitations.

- coordinate-system=*wgs84*
- units=*miles*
- boundaries-included
- boundaries-excluded

- boundaries-latitude-excluded
- boundaries-longitude-excluded
- boundaries-south-excluded
- boundaries-north-excluded
- boundaries-east-excluded
- boundaries-west-excluded
- boundaries-circle-excluded
- cached
- uncached
- type=*point*
- type=*long-lat-point*

10.0 Appendix B: XML Query Options Reference

This chapter defines the XML structure of query options (`search:options`) used by the search related features of the *MarkLogic REST API* and the MarkLogic Java API.

Query options can also be expressed in JSON. For details, see "Appendix A: JSON Query Options Reference" on page 160.

For more information, see Understanding the Search API in the *Search Developer's Guide*.

10.1 <options>

The <options> node is the outer wrapper for query options. An options node can contain the following elements:

- <additional-query>
- <concurrency-level>
- <constraint>
- <debug>
- <default-suggestion-source>
- <extract-metadata>
- <forest>
- <fragment-scope>
- <grammar>
- <operator>
- <page-length>
- <quality-weight>
- <return-constraints>
- <return-facets>
- <return-metrics>
- <return-plan>
- <return-qtext>
- <return-query>
- <return-results>
- <return-similar>
- <search-option>

- <searchable-expression>
- <sort-order>
- <suggestion-source>
- <term>
- <transform-results>
- <tuples>
- <values>

10.2 <additional-query>

An additional serialized `cts:query` node, which is combined using a `cts:and-query` with the query resulting from the query text. The query results are constrained by the specified `additional-query`, but any terms matching the `addtional-query` are not highlighted in the snippet result output. For example, the following options node constrains the results to the directory named /my/directory/:

```
<options xmlns="http://marklogic.com/appservices/search">
  <additional-query>
    <cts:directory-query xmlns:cts="http://marklogic.com/cts">
      <cts:uri>/my/directory/</cts:uri>
    </cts:directory-query>
  </additional-query>
</options>
```

If you have multiple `additional-query` options, they are combined using a `cts:and-query`.

10.3 <concurrency-level>

The maximum number of threads used to resolve facets. The default is 8, which specifies that at most 8 threads will be used concurrently to resolve facets. For example, the following specifies a concurrency level of 16:

```
<options xmlns="http://marklogic.com/appservices/search">
  <concurrency-level>16</concurrency-level>
</options>
```

10.4 <constraint>

The outer wrapper element for a constraint definition. Constraints are used to define facets which can be returned as part of the search results. The default is no defined constraints.

Each constraint element must have an `@name` attribute (required), which is the unique name of this constraint. The name can then be used in the search grammar to specify the constraint. A constraint name must be unique within the containing query options and must not contain whitespace.

The constraint element can have zero or more of the following elements:

- <value>
- <word>
- <collection>
- <range>
- <element-query>
- <properties>
- <geo-*>
- <custom>
- <term-option>
- <facet-option>

10.4.1 <value>

Specifies element or attribute values on which to constrain. You cannot create facets from a value constraint. For example:

```
<constraint name="my-value">
    <value>
      <element ns="my-namespace" name="my-localname"/>
    </value>
</constraint>
<constraint name="my-attribute-value">
    <value>
      <attribute ns="" name="my-attribute"/>
      <element ns="my-namespace" name="my-localname"/>
    </value>
</constraint>
<constraint name="fieldvalue">
    <value>
      <field name="fieldvalue"/>
      <weight>2.0</weight>
    </value>
</constraint>
```

10.4.2 <word>

Specifies the element, attribute, or field on which to constrain by word. You cannot create facets from a word constraint. For example:

```
<constraint name="name">
    <word>
      <element ns="http://widgets-r-us.com" name="name"/>
    </word>
```

```
</constraint>
<constraint name="description">
    <word>
        <field name="my-field"/>
        <weight>2.0</weight>
    </word>
</constraint>
```

10.4.3 <collection>

Specifies the collection on which to constrain. For example:

```
<constraint name="my-collection">
    <collection prefix="http://server.com/my-collection/"/>
</constraint>
```

10.4.4 <range>

Specifies the element or attribute on which to constrain by range. There must be a range index of the specified type (and collation for string range indexes) defined for the specified element or attribute.

Each range element with a type attribute child, an optional collation attribute child (for string range indexes), an element child, an optional attribute child (for attribute ranges), and may have one or more computed-bucket children and/or one or more facet-option children (to pass options to the underlying lexicon apis). If it is a field range constraint, then the range element has a field child element with a name attribute. If it is a path-index range constraint, then the range element has a path-index child element with any namespaces defined in xmlns attributes. For example:

```
<options xmlns="http://marklogic.com/appservices/search">
 <constraint name="made">
    <range type="xs:dateTime"><!-- requires a dateTime range index -->
    <element ns="http://example.com" name="manufactured"/>
    <attribute ns="" name="date"/>
    <computed-bucket name="today" ge="P0D" lt="P1D"
            anchor="now">Today</computed-bucket>
    <computed-bucket name="30-days" ge="-P30D" lt="P1D"
            anchor="now">Last 30 days</computed-bucket>
    <computed-bucket name="60-days" ge="-P60D" lt="P1D"
            anchor="now">Last 60 Days</computed-bucket>
    <computed-bucket name="year" ge="-P1Y" lt="P1D"
            anchor="now">Last Year</computed-bucket>
  </range>
 </constraint>
 <constraint name="color">
   <range type="xs:string">
     <element ns="" name="bodycolor"/>
   </range>
 </constraint>
 <constraint name="name">
   <range type="xs:string" collation="http://marklogic.com/collation/">
     <field name="my-field-name"/>
```

```
    </range>
  </constraint>
  <constraint name="color-facet">
     <range type="xs:string" facet="true">
       <element ns="" name="bodycolor"/>
       <!-- the facet-option values are passed directly to the
            underlying lexicon calls -->
       <facet-option>frequency-order</facet-option>
       <facet-option>descending</facet-option>
     </range>
  </constraint>
  <constraint name="fieldrange">
     <range type="xs:string"
             collation="http://marklogic.com/collation/">
        <field name="fieldrange"/>
     </range>
  </constraint>
  <constraint name="fieldrangebucket">
    <range type="xs:dateTime">
      <facet-option>ascending</facet-option>
      <field name="fieldrangebucket"/>
      <computed-bucket name="older" lt="-P365D" anchor="now">
        Older than 1 years
      </computed-bucket>
      <computed-bucket name="year" ge="-P365D" lt="-P30D" anchor="now">
        1 month to 1 year ago
      </computed-bucket>
      <computed-bucket name="month" ge="-P30D" lt="-P7D" anchor="now">
        7 to 30 days ago
      </computed-bucket>
      <computed-bucket name="week" ge="-P7D" lt="-P1D" anchor="now">
        1 to 7 days ago
      </computed-bucket>
      <computed-bucket name="today" ge="-P1D" lt="P0D" anchor="now">
        Today
      </computed-bucket>
      <computed-bucket name="future" ge="P0D" anchor="now">
        Future
      </computed-bucket>
     </range>
  </constraint>
  <constraint name="year">
    <range type="xs:gYear" facet="true">
      <path-index xmlns:my="http://example.com">
        /publication/my:meta/my:year
      </path-index>
      <!--uses same options structure, options as element,
          element-attribute or field based range constraints -->
    </range>
  </constraint>
</options>
```

For range constraints with either bucket or computed-bucket specifications, for maximum performance and sortability, the buckets should be in a continuous order; if the order is not continuous (either ascending or descending), then the buckets are returned in the order specified, regardless of any sorting facet-option in the specification.

10.4.5 <element-query>

Specifies a constraint that restricts the search to the specified element. You cannot create facets from an element-query constraint.

```
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="sample-element-constraint">
    <element-query name="title" ns="http://my/namespace" />
  </constraint>
</options>
```

10.4.6 <properties>

Specifies a constraint that finds matches on the corresponding properties documents. You cannot create facets from a properties constraint.

```
<options xmlns="http://marklogic.com/appservices/search">
  <constraint name="sample-property-constraint">
    <properties />
  </constraint>
</options>
```

For an additional example, see the second example below.

10.4.7 <geo-*>

Specifies a geospatial constraint. Must be one of the following elements: `<geo-attr-pair>`, `<geo-elem-pair>`, or `<geo-elem>`. If the constraint has a `<heatmap>` child, then it can be used as a facet, and can contain `<facet-options>`. For example:

```
<constraint name="my-geo-attr-pair">
    <!-- Uses cts:element-attribute-pair-geospatial-query, and
    cts:element-attribute-pair-geospatial-boxes for the
    heatmap facet.  -->
  <geo-attr-pair>
    <heatmap s="23.2" w="-118.3" n="23.3" e="-118.2"
             latdivs="4" londivs="4"/>
    <facet-option>empties</facet-option>
    <parent ns="ns1" name="elem1"/>
    <lat ns="ns2" name="attr2"/>
    <lon ns="ns3" name="attr3"/>
  </geo-attr-pair>
</constraint>
<constraint name="my-geo-elem-pair">
    <!-- Uses cts:element-pair-geospatial-query, and
    cts:element-pair-geospatial-boxes for the
```

```
    heatmap facet (not shown).  -->
  <geo-elem-pair>
    <parent ns="ns1" name="elem2"/>
    <lat ns="ns2" name="attr2"/>
    <lon ns="ns3" name="attr3"/>
  </geo-elem-pair>
</constraint>
<constraint name="my-geo-elem">
    <!-- Uses cts:element-geospatial-query, and
    cts:element-geospatial-boxes for the
    heatmap facet (not shown).  -->
  <geo-elem>
    <element ns="ns1" name="elem3"/>
    <geo-option>type=long-lat-points</geo-option>
  </geo-elem>
</constraint>
<constraint name="my-geo-elem-child">
    <!-- Uses cts:element-child-geospatial-query, and
    cts:element-geospatial-boxes for the
    heatmap facet (not shown).  -->
  <geo-elem>
    <parent ns="ns1" name="elem4"/>
    <element ns="ns1" name="elem5"/>
  </geo-elem>
</constraint>
```

Note that when specifying coordinates using a geospatial constraint, you must put any coordinate values that contain the negative sign (-) in double-quotes, otherwise the search API will interpret the negative sign as the negation operator (when using the default grammar). For example, to specify a point using a geospatial constraint, specify qtext of the form my-geo-elem-child:"-12, 5" in your search string.

10.4.8 <custom>

Specifies a custom constraint along with the name of the function implementations used to evaluate the custom constraint. For example:

```
<constraint name="my-custom">
  <custom facet="true">
    <parse apply="my-parse-function"
        ns="my-function-namespace" at="path-to-module.xqy"/>
    <start-facet apply="my-start-function"
        ns="my-function-namespace" at="path-to-module.xqy"/>
    <finish-facet apply="my-finish-function"
        ns="my-function-namespace" at="path-to-module.xqy"/>
  </custom>
</constraint>
<!--  The start-facet and finish-facet elements can be omitted if
      facet="false". When facet="true", the start-facet can be ommitted
      if you do not run in concurrent mode ("concurrent" option on the
      lexicon functions).
-->
```

10.4.9 <term-option>

Specifies the options passed into the search (for example, case-insensitive). There can be zero or more `term-option` elements. By default, the search uses the same default options as the underlying `cts:query` constructors, and the defaults change based on your index configuration. You can use `term-option` elements in several places, including as a child of the term element and within many constraint elements (except for `element-query` constraints). Within constraints, `term-option` specifications typically occur as a child of the type of constraint (for example, in a word constraint, as a child of word. Use `search:check-options` to make sure the usage is legal.

Legal term option values are:

- case-sensitive
- case-insensitive
- diacritic-sensitive
- diacritic-insensitive
- punctuation-sensitive
- punctuation-insensitive
- whitespace-sensitive
- whitespace-insensitive
- stemmed
- unstemmed
- wildcarded
- unwilcarded
- exact
- lang=*iso639code*

For example:

```
<term-option>diacritic-insensitive</term-option>
```

Or, for example, within in a word constraint:

```
<constraint name="name"
  xmlns="http://marklogic.com/appservices/search">
  <word>
    <element ns="http://widgets-r-us.com" name="name"/>
    <term-option>case-sensitive</term-option>
  </word>
</constraint>
```

10.4.10 <facet-option>

Specifies options for facets, and is specified as a child of any range constraint or other constraint that can be used as a facet (any constraints except word, value, element-query, or property).

Legal values for a `facet-option` element are generally any option that can be passed into the underlying `cts:range-query` or lexicon API. The following list enumerates the facet-option values, but be aware that some options are only available with some range types. For more detail on these options, see the documentation for the underlying range or lexicon APIs.

- ascending
- descending
- empties
- any
- document
- properties
- locks
- frequency-order
- item-order
- fragment-frequency
- item-frequency
- type=type
- timezone=TZ
- limit=N
- sample=N
- truncate=N
- skip=N
- score-logtfidf
- score-logtf
- score-simple
- score-random
- checked
- unchecked
- concurrent
- map

For example:

```
<constraint name="color"
xmlns="http://marklogic.com/appservices/search">
   <range type="xs:string">
     <element ns="" name="bodycolor"/>
     <facet-option>item-frequency</facet-option>
   </range>
</constraint>
```

10.5 <debug>

Activates debugging mode. Additional report elements will be present in the output. Set to true to activate. The default is false.

10.6 <default-suggestion-source>

Defines the content to be used as the default source of suggestions (see `search:suggest`). The source may be expressed as a reference to an existing named constraint, or as a collection, value, word or word-lexicon element. Note that the use of word-lexicon (the database-wide word lexicon) is not recommended as best practice; collection and range lexicons will yield the best performance.

Each default-suggestion-source element can optionally have an `@collation` attribute, which specifies the collation of the value lexicon used during query evaluation. If no collation is specified, then the query uses default collation for the context in which the query is evaluated.

The default-suggestion-source element can have zero or more of the following child elements:

- <collection>
- <range>
- <word>

10.6.1 <collection>

Specifies using the collection lexicon for suggestions. For example:

```
<default-suggestion-source>
        <collection/>
</default-suggestion-source>
```

10.6.2 <range>

Specifies the element or attribute lexicon to use for suggestions. For example:

```
<default-suggestion-source>
  <range type="xs:string">
    <element ns="my-namespace" name="my-localname"/>
    <attribute ns="" name="my-attribute"/>
```

```
  </range>
</default-suggestion-source>
```

10.6.3 <word>

Specifies using the word lexicon for suggestions. This option might not scale well for a large database. For example:

```
<default-suggestion-source>
    <word/>
</default-suggestion-source>
<default-suggestion-source>
    <word>
        <field name="my-field"/>
    </word>
</default-suggestion-source>
```

10.7 <extract-metadata>

Specifies data to return in the search:metadata section of the search:response output. You can specify element or attribute content from the search hits to be returned or values from any constraints in your search.

```
<search:extract-metadata>
           <!-- qnames specify elements or attributes in document
                content; multiples will be returned if they exist -->
   <search:qname elem-ns="" elem-name="bar"/>
   <search:qname elem-ns="http://example.com" elem-name="baz"
                 attr-ns="http://ick" attr-name="id"/>
   <search:qname elem-ns="http://example.com"
                 elem-name="blech" attr-ns="" attr-name="size"/>
           <!-- constraint-values extract values from the index
                associated with the constraint -->
   <search:constraint-value ref="idx"/>
   <search:constraint-value ref="id"/>
   <search:qname elem-ns="http://example.com" elem-name="foo"/>
</search:extract-metadata>
```

10.8 <forest>

A single forest ID to pass into cts:search. To specify multiple forests, use multiple forest elements in the options node.

The value must be an `xs:unsignedLong` type.

10.9 <fragment-scope>

Controls the scope over which to search. Valid values are documents and properties, and the default value is documents.

You can specify fragment-scope as a child of `<options>`, and that sets a global fragment scope which applies both to what the search returns (that is, if it returns results from document fragments or from property fragments) and the global scope is inherited by any constraints that do not explicitly override the fragment-scope.

You can also specify fragment-scope as a child of `<range>`, `<value>`, or `<word>` constraints. Specifying as a child of one of these constraints sets a local fragment scope for that constraint. The local fragment-scope overrides a global fragment-scope. For example, a fragment-scope of properties on a range constraint enables you to facet on a value stored in a property, even if you are searching over documents. For example:

```
<options xmlns="http://marklogic.com/appservices/search">
<fragment-scope>documents</fragment-scope>
  <constraint name="last-modified">
    <range type="xs:dateTime">
      <element ns="http://marklogic.com/xdmp/property"
               name="last-modified"/>
      <fragment-scope>properties</fragment-scope>
    </range>
  </constraint>
  <debug>true</debug>
</options>
```

10.10 <grammar>

Wrapper element for grammar definition. The default grammar defines "Google-style" parsing.

The grammar element has a quotation element that specifies the quotation character with which to surround phrases. The text between the quotation characters is treated as a phrase. You cannot specify a search that includes the quotation character; for example, to specify a search that includes the double quotation character (the default quotation character), modify your grammar to use a different quotation character.

The grammar element can have 0 or more joiner elements and 0 or more starter elements.

The grammar element should contain at least one `joiner`, `starter`, or `implicit` child element. If the grammar element is present but empty, then the grammar does nothing, and the search is parsed according to the term option.

- <implicit>
- <joiner>
- <starter>

10.10.1 <implicit>

Specifies the cts:query to use to implicitly combine terms when no joiner is specified in the qtext. Use this to modify the implicit search grammar (for example, if you want the default search to be a cts:or-query instead of a cts:and-query).

The following is an example of a options element with an implicit specification for a cts:or-query:

```
<options xmlns="http://marklogic.com/appservices/search">
  <grammar>
    <implicit>
      <cts:or-query strength="10"
xmlns:cts="http://marklogic.com/cts"/>
    </implicit>
  </grammar>
</options>
```

10.10.2 <joiner>

Specifies what text to use to combine terms together, and what is the underlying cts:query constructor to use to join the terms together. You specify the function to call for the joiner with the apply attribute, along with optional ns (for the module namespace) and at (for the module path) attributes. Additionally, the strength attribute determines the order of precedence over other joiner elements, the optional options attribute specifies a space-separated list of options that are passed through to the underlying `cts:query` constructor, and the element attribute specifies the `cts:query` element name (for example, `cts:and-query`).

10.10.3 <starter>

Specifies what text to use to delimit and group terms. You specify the function to call for the starter with the apply attribute, along with optional ns (for the module namespace) and at (for the module path) attributes. Additionally, the strength attribute determines the order of precedence over other starter elements, the optional options attribute specifies a space-separated list of options that are passed through to the underlying `cts:query` constructor, the element attribute specifies the cts:query element name (for example, `cts:and-query`), and the delimiter attribute specifies the string to use as a delimiter for the starter.

The following is an example of a grammar element.

```
<grammar xmlns="http://marklogic.com/appservices/search">
  <quotation>"</quotation>
  <implicit>
    <cts:and-query strength="20" xmlns:cts="http://marklogic.com/cts"/>
  </implicit>
  <starter strength="30" apply="grouping" delimiter=")">(</starter>
  <starter strength="40" apply="prefix"
    element="cts:not-query">-</starter>
  <joiner strength="10" apply="infix" element="cts:or-query"
    tokenize="word">OR</joiner>
  <joiner strength="20" apply="infix" element="cts:and-query"
```

```
    tokenize="word">AND</joiner>
  <joiner strength="30" apply="infix" element="cts:near-query"
    tokenize="word">NEAR</joiner>
  <joiner strength="30" apply="near2"
    element="cts:near-query">NEAR/</joiner>
  <joiner strength="50" apply="constraint">:</joiner>
  <joiner strength="50" apply="constraint" compare="LT"
    tokenize="word">LT</joiner>
  <joiner strength="50" apply="constraint" compare="LE"
    tokenize="word">LE</joiner>
  <joiner strength="50" apply="constraint" compare="GT"
    tokenize="word">GT</joiner>
  <joiner strength="50" apply="constraint" compare="GE"
    tokenize="word">GE</joiner>
  <joiner strength="50" apply="constraint" compare="NE"
    tokenize="word">NE</joiner>
</grammar>
```

10.11 <operator>

A named wrapper for one or more state elements, each representing a unique run-time configuration option. For example, if an operator with the name "sort" is defined, query text [sort:foo] will select the state child with the name "foo" at query runtime, using the option specified on that state element. Options affecting query parsing (such as constraint, grammar, term, empty) may not be configured via operators.

An operator element can have one or more state elements. Each state element can have one of the following elements:

- additional-query
- debug
- forest
- page-length
- quality-weight
- searchable-expression
- sort-order
- transform-results

In the following example, a search for special:hello constrains the search by the "hello world" query, and a search for special:forest constrains the search to the forest names "my-forest".

```
<operator name="special">
  <state name="hello">
    <additional-query>{cts:word-query("hello world")}
    </additional-query>
  </state>
  <state name="forest">
```

```
      <forest>{xdmp:forest("my-forest")}</forest>
    </state>
  </operator>
```

10.12 <page-length>

Specifies the number of results per page. The default value is 10.

The value must be an `xs:unsignedInt type`.

10.13 <quality-weight>

Specifies a a weighting factor to use in the query. The default value is 1.0.

The value must be an `xs:double type`.

10.14 <return-constraints>

Include original constraint definitions in the results. The default is false.

The value must be an `xs:boolean` type.

10.15 <return-facets>

Include resolved facets in the results. The default is true.

The value must be an `xs:boolean` type.

10.16 <return-metrics>

Include statistics in the results. The default is true.

The value must be an `xs:boolean` type.

10.17 <return-plan>

Include xdmp:plan output in the results. The default is false. The output is useful in examining the evaluation plan of the search.

The value must be an `xs:boolean` type.

10.18 <return-qtext>

Include the original query text in the results. The default is true.

The value must be an `xs:boolean` type.

10.19 <return-query>

Include the XML query representation in the results. The default is false.

The value must be an `xs:boolean` type.

10.20 <return-results>

Include search results in the output. (Use transform-results to specify how each result should be formatted.) The default is true.

The value must be an `xs:boolean` type.

10.21 <return-similar>

Include with each search result a list of URLs of similar documents in the database. The default is false.

The value must be an `xs:boolean` type.

10.22 <search-option>

For advanced users, a single option to be passed in with `cts:search` calls (for example, filtered, unfiltered, score-logtfidf, and so on). To pass in multiple options, specify multiple search-options elements in the options node. The default is no additional options. For example:

```
<search-option>unfiltered</search-option>
<search-option>score-logtf</search-option>
```

10.23 <searchable-expression>

An expression to be searched. Whatever expression is specified is returned from the search. For example, if you specify `//p`, then `p` elements that match the search criteris are returned. The expression must be an inline fully searchable XPath expression, and all necessary namespaces must be declared using xmlns attributes. For example:

```
<searchable-expression xmlns:ex="http:example.com"
    xmlns:com="http://company.com">/ex:orders/com:company
</searchable-expression>
```

The default value is `fn:collection()`, which searches all documents in the database.

> **Note:** This option does not affect facet results. To constrain facet results, use <additional-query>.

10.24 <sort-order>

Set the default sort order. The first such element is the primary sort order, the second secondary sort order, and so on. The default is to sort by score, descending. Note that the default scoring algorithm can be set just like any other option with the option named `search-option`. If you are sorting by an element or an attribute, you must specify a type attribute with a value corresponding

to the range index type of that element or attribute (for example, `xs:string`, `xs:dateTime`, and so on). If the corresponding range index is of type `xs:string`, then you can optionally specify a `collation` attribute (otherwise the collation of the query is used). To change the sorting direction, specify an optional `direction` attribute with a value of `descending` (the default) or `ascending`.

You should not apply `sort-order` to a field that has more than one included element.

The `sort-order` element must have one of a single `element` child, a single `score` child, a single `field` child, or a single `json-key` child. If there is a `score` child, it specifies to sort by the score of the search result. If there is an `element` child it can optionally have an `attribute` sibling (to specify an attribute of the preceding element). Both the `element` and `attribute` elements must have `ns` and `name` attributes to specify the namespace and local-name of the specified element and attribute. Additionally, the `sort-order` element can have 0 or more `annotation` elements (to add comments, for example).

For example, the following specifies a primary sort order using the element value for `my-element` (which needs a string range index with the specified collation), and a secondary sort order of score ascending:

```
<sort-order type="xs:string"
collation="http://marklogic.com/collation/"
            direction="ascending">
  <element ns="my-namespace" name="my-element"/>
  <annotation>some user comment can go here</annotation>
</sort-order>
<sort-order direction="ascending">
  <score/>
</sort-order>
```

10.25 <suggestion-source>

Specifies a constraint source to override a named constraint when using `search:suggest`. The suggestions are often used for type-ahead suggestions in a search user interface. If empty, no suggestions are generated when that constraint is applied. Specifying an alternate suggestion-source is useful in cases where you have a named constraint to use for searching and facets, but you might want to use a slightly (or completely) different source for type-ahead suggestions without needed to re-parse your search terms.

Each suggestion source must have a ref attribute corresponding to a named constraint (one suggestion-source per named constraint). A suggestion-source can have one of the following child elements: collection, range, word, or word-lexicon.

For example, the following overrides the tag: prefix, using the range index for the attribute `shortname` instead of the one for name when using `search:suggest`:

```
<constraint name="tag">
  <range collation="http://marklogic.com/collation/"
         type="xs:string" facet="true">
```

```
    <element ns="my-namespace" name="my-element"/>
    <attribute ns="" name="name"/>
  </range>
</constraint>
<suggestion-source ref="tag">
  <range collation="http://marklogic.com/collation/"
         type="xs:string" facet="true">
    <element ns="my-namespace" name="my-element"/>
    <attribute ns="" name="shortname"/>
  </range>
</suggestion-source>
```

10.26 <term>

Specifies handling of empty searches and controls options for how individual terms (that is, terms not associated with a constraint) will be represented when parsing the search.

To control how empty searches (that is, the empty string passed into `search:search`) are resolved, specify an empty child element with an apply attribute. The value of the apply attribute specifies the behavior for empty searches: a value of all-results specifies that empty searches return everything in the database, a value of no-results (the default) specifies that an empty search returns nothing. Additionally, you create your own function to handle empty searches. To specify your own function, create a function that returns a cts:query and specify the local-name of your function in the apply attribute, the namespace of the function library module in the ns attribute, and the path to the module in the at attribute.

You can also specify a default child element which determines special handling to all terms, so you can have all terms apply a set of rules such as query weighting, specifying a element or attribute value or word query to be used as a default, or specifying a constraint to be applied by default.

Additionally, you can specify zero or more term-option elements to control the behavior of the search terms.

For example:

```
<term>
   <empty apply="no-results" />
   <term-option>diacritic-insensitive</term-option>
   <term-option>unwildcarded</term-option>
</term>
```

The following examples show default term specifications for terms with no constraints:

Term element, parsing to a weighted word query.

```
<term xmlns="http://marklogic.com/appservices/search>
    <default>
        <word>
```

```
            <weight>2.0</weight>
        </word>
    </default>
    <empty apply="all-results"/>
</term>
```

Term element, parsing to a weighted element word query on element "foo:bar":

```
<term xmlns="http://marklogic.com/appservices/search">
    <default>
        <word>
            <element ns="foo" name="bar"/>
            <weight>2.0</weight>
        </word>
    </default>
    <empty apply="all-results"/>
</term>
```

Term element, parsing to an element-values query scoped to the properties fragment:

```
<term xmlns="http://marklogic.com/appservices/search">
    <default>
        <value>
            <element ns="http://example.com" name="copyright-year"/>
            <fragment-scope>properties</fragment-scope>
        </value>
    </default>
    <empty apply="all-results"/>
</term>
```

Term element, parsing to a field word query via an existing field-word constraint:

```
<term>
    <default ref="my-field-word-constraint"/>
    <empty apply="all-results"/>
</term>
```

Term element, referencing custom code (`<default>` configuration will be ignored in a conflict).

```
<term apply="my-term-query" ns="http://custom/search"
   at="/lib/custom.xqy">
    <default ref="my-field-word-constraint"/>
    <empty apply="all-results"/>
</term>
```

10.27 <transform-results>

Specifies a function to use to process a search result for the snippet output. The default is that each result is formatted using the built-in default snippetting function.

To use a different snippetting function, you must use either `search:search` or `search:resolve`; the `search:snippet` function always uses the default snippeting function. Specify the local-name of the function to pass in as the value of the apply attribute, the namespace as the value of the ns attribute, and the path to the module as the value of the at attribute. You can pass in parameters to the function by specifying zero or more param child elements (the parameters are passed in in the order specified).

In addition to the default snippetting function, there are three other built-in snippetting functions: `apply="raw"` (return the whole node), `apply="empty-snippet"` (return an empty snippet), and `apply="metadata-snippet"` (return the snippet from the specified element in the properties document).

The following example specifies a custom snippetting function:

```
<transform-results apply="snippet" ns="my-namespace"
                   at="/my-library.xqy"/>
```

The following example specifies a snippet returning the element named my-element in the properties document:

```
<transform-results apply="metadata-snippet">
  <preferred-elements>
    <!-- Specify namespace and localname for elements that exist
         in properties documents -->
    <element ns="" name="my-element"/>
  </preferred-elements>
</transform-results>
```

10.28 <tuples>

Identifies values range indexes or lexicons in which to find value co-occurrences. A `<values-response>` of co-occurrences is included in the search response. You can specify range, collection, geospatial, and uri indexes or lexicons.

A `<tuples/>` element can include an `<aggregate/>` child to specify an aggregate builtin or user-defined function to apply to the values. You can also specify one or more options using `<values-options/>`; see `cts:values` for a list of available options.

The following example specifies co-occurrences between editor and author element values. attribute range index and the URI lexicon.

```
<options xmlns="http://marklogic.com/appservices/search">
  <tuples name="editor-author">
    <range type="xs:string"
        collation="http://marklogic.com/collation/">
       <element name="editor" ns=""/>
    </range>
    <range type="xs:string"
         collation="http://marklogic.com/collation/">
```

```
        <element name="author" ns=""/>
      </range>
      <lexicon-option>limit=10</lexicon-option>
    </tuples>
  </options>
```

10.29 <values>

Identifies a values range index or lexicon with which to constrain the search. A `<values-response>` of matching index or lexicon values is included in the search response. You can specify range, collection, geospatial, and uri indexes or lexicons.

A `<values/>` element can include an `<aggregate/>` child to specify an aggregate builtin or user-defined function to apply to the values. You can also specify one or more options using `<values-options/>`; see `cts:values` for a list of available options.

The following example specifies values from an element attribute range index and from the URI lexicon. For more examples, see `search:values`.

```
<options xmlns="http://marklogic.com/appservices/search">
    <!-- element attribute range index values -->
    <values name="example">
        <range type="xs:integer">
            <element ns="" name="game"/>
            <attribute ns="" name="runs"/>
        </range>
        <aggregate apply="sum"/>
    </values>

    <!-- uri lexicon values -->
    <values name="uri">
      <uri/>
      <values-option>limit=10</values-option>
    </values>
</options>
```

11.0 Technical Support

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

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

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

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

support@marklogic.com

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

Combined Product Notices

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

COPYRIGHT

Copyright © 2013 MarkLogic Corporation. All rights reserved.
This technology is protected by U.S. Patent No. 7,127,469B2, U.S. Patent No. 7,171,404B2, U.S. Patent No. 7,756,858 B2, and U.S. Patent No 7,962,474 B2.

The MarkLogic software is protected by United States and international copyright laws, and incorporates certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers set forth below.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

TRADEMARK NOTICE

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