MarkLogic Server

Java Application Developer's Guide

MarkLogic 10
May, 2019

Last Revised: 10.0, May, 2019

Table of Contents

Java Application Developer's Guide

1.0 Introduction to the Java API

The Java Client API is an open source API for creating applications that use MarkLogic Server for document and search operations. This chapter includes the following sections:

1.1 Java Client API Overview

The Java Client API provides the following capabilities:

- Insert, update, or remove documents and document metadata, either individually or in batches. For details, see “Single Document Operations” on page 36, “Synchronous Multi-Document Operations” on page 70, or “Asynchronous Multi-Document Operations” on page 92.
- Query documents, lexicons, and semantic data. For details, see “Searching” on page 144.
- Extract data from MarkLogic as tables. For details, see “Optic Java API for Relational Operations” on page 218.
- Persist, retrieve, and query Java objects in stored in MarkLogic. For details, see “POJO Data Binding Interface” on page 226.
- Configure persistent and dynamic query options. For details, see “Query Options” on page 190.
- Apply transformations to new content and search results. For details, see “Content Transformations” on page 282.
- Extend the Java API to expose custom capabilities you install on MarkLogic Server. For details, see “Extending the Java API” on page 288.

When working with the Java API, you first create a manager for the type of document or operation you want to perform on the database (for instance, a `JSONDocumentManager` to write and read JSON documents or a `QueryManager` to search the database). To write or read the content for a database operation, you use standard Java APIs such as `InputStream`, DOM, StAX, JAXB, and Transformer as well as Open Source APIs such as JDOM and Jackson.

The Java API provides a handle (a kind of adapter) as a uniform interface for content representation. As a result, you can use APIs as different as `InputStream` and DOM to provide content for one `read()` or `write()` method. In addition, you can extend the Java API so you can use the existing `read()` or `write()` methods with new APIs that provide useful representations for your content.

This chapter covers a number of basic architecture aspects of the Java API, including fundamental structures such as *database clients*, *managers*, and *handles* used in almost every program you will write with it. Before starting to code, you need to understand these structures and the concepts behind them.

The MarkLogic Java Client API is built on top of the MarkLogic REST API. The REST API, in turn, is built using XQuery that is evaluated against an HTTP App Server. For this reason, you need a REST API instance on MarkLogic Server to use the Java API. A suitable REST API instance on port 8000 is pre-configured when you install MarkLogic Server. You can also create your own on another port. For details, see “Choose a REST API Instance” on page 15.

1.2 Java Client API or Java XCC?

The Java API co-exists with the previously developed XCC API, as they are intended for different use cases.

You can use the Java Client API to quickly become productive in your existing Java environment, using the Java interfaces for search and document management. You can also use the Java Client API extension capability to invoke XQuery and Server-Side JavaScript code on MarkLogic Server. This enables you to take advantage of MarkLogic functionality not exposed directly through the Java Client API.

XCC provides a lower-level interface for running remote or ad hoc XQuery or Server-Side JavaScript. While XCC provides significant flexibility, it also has a somewhat steeper learning curve for developers. You can think of XCC as being to ODBC or JDBC: A low level API for sending query language directly to the server. By contrast, the Java Client API is a higher level API for working with database constructs in Java.

In terms of performance, the Java API is very similar to Java XCC for compatible queries. The Java API is a very thin wrapper over a REST API with negligible overhead.

For more information about XCC, see the *XCC Developer’s Guide*.

1.3 Getting Started

To get started with the Java Client API, do the following:

- Required Software
- Make the Libraries Available to Your Application
- Choose a REST API Instance
- Create Users
- Explore the Examples

1.3.1 Required Software

For information about Java platform requirements, see the following page:

https://github.com/marklogic/java-client-api

The Java Client API also requires access to a MarkLogic Server installation configured with a REST Client API instance. When you install MarkLogic 8 or later, a pre-configured REST API instance is available on port 8000. For more details, see Administering REST Client API Instances in the *REST Application Developer's Guide*.

For information specific to rolling upgrades, see Java Client API in the *Administrator's Guide*.

1.3.2 Make the Libraries Available to Your Application

You can make the Java Client API libraries available to your project in one of the following ways:

- ZIP File
- Maven
- Gradle

For more details, see the following page:

http://developer.marklogic.com/products/java

The Java Client API is an open-source project, so you can also access the sources and build your own library. For details, see "Downloading the Library Source Code" on page 34.

1.3.2.1 ZIP File

You can download a ZIP file from the following URL:

http://developer.marklogic.com/products/java

Download the ZIP file and uncompress it to a directory of your choice. The jar files you need to add to your class path are in the `lib/` subdirectory.

1.3.2.2 Maven

To use the Maven repository, add the following to dependency to your Maven project POM file. (You may need to change the `version` data to match the release you're using.)

```
<dependency>
    <groupId>com.marklogic</groupId>
    <artifactId>marklogic-client-api</artifactId>
    <version>4.0.3</version>
</dependency>
```

You must also add the following to the repositories section of your `pom.xml`.

```
<repository>
    <id>jcenter</id>
    <url>http://jcenter.bintray.com</url>
</repository>
```

1.3.2.3 Gradle

If you use Gradle as your build tool, you must use Gradle version 1.7 or later. Add the following to your `build.gradle` file. Modify the version number as needed.

```
compile group: 'com.marklogic',
name: 'marklogic-client-api',
version: '4.0.3'
```

Add the following to your `build.gradle` repositories section:

```
jcenter()
```

1.3.3 Choose a REST API Instance

The Java API implementation interacts with MarkLogic Server using the MarkLogic REST Client API. Therefore you must have access to a REST API instance in MarkLogic Server before you can run an application that uses the Java Client API.

A REST API instance includes a specially configured HTTP App Server capable of handling REST Client API requests, a content database, and a modules database. MarkLogic Server comes with a suitable REST API instance attached to the Documents database, listening on port 8000.

The examples in this guide assume you're using the pre-configured REST API instance on port 8000 of localhost. If you want to create and use a different REST instance, see , see Administering REST Client API Instances in the *REST Application Developer's Guide*.

Note: Each application must use a separate modules database and REST API instance.

1.3.4 Create Users

You might need to create MarkLogic Server users with appropriate security roles, or give additional privileges to existing users.

Any user who reads data will need at least the `rest-reader` role and any user that writes data will need at least the `rest-writer` role.

REST instance configuration operations, such as setting instance properties require the `rest-admin` role. For details, see "REST Server Configuration" on page 276.

Some operations require additional privileges. For example, a `DatabaseClient` that connects to a database other than the default database associated with the REST instance must have the `http://marklogic.com/xdmp/privileges/xdmp-eval-in` privilege. Using the `ServerEvaluationCall` interface also requires special privileges; for details, see "Evaluating an Ad-Hoc Query or Server-Side Module" on page 298.

Note that MarkLogic Server Administration is *not* exposed in Java, so operations such as creating indices, creating users, creating databases, etc. must be done via the Admin Interface, REST Management API, or other MarkLogic Server administration tool. The server configuration component of the Java API is restricted to configuration operations on the REST instance.

For details, see Security Requirements in the *REST Application Developer's Guide*.

1.3.5 Explore the Examples

The Java Client API distribution includes several examples in the `examples/` directory. The examples include the following packages:

- `com.marklogic.client.example.cookbook`: A collection of small examples of using the core features of the API, such as document operations and search. Most of the example code in this guide is drawn from the Cookbook examples.
- `com.marklogic.client.example.handle`: Examples of using handles based on open source document models, such as JDOM or Jackson. Examples of handle extensions that read or write database documents in a new way.
- `com.marklogic.client.example.extension`: A collection of extension classes and examples for manipulating documents in batches.

For instructions on building and running the examples, see the project wiki on GitHub:

http://github.com/marklogic/java-client-api/wiki/Running-the-Examples

1.4 Creating, Working With, And Releasing a Database Client

Your application must create at least one `DatabaseClient` object before it can interact with MarkLogic using the Java Client API. The following topics cover key things you should know about the `DatabaseClient` interface.

1.4.1 The Role of a Database Client

A `DatabaseClient` object encapsulates the information needed to connect to MarkLogic, such as the host and port of a REST API instance, the database to operate on, and the authentication context. Internally, each `DatabaseClient` object is associated with a connection pool, as described in “Connection Management and Configuration” on page 17.

Most tasks you perform using the Java Client API are handled by a manager object. For example, you use a `QueryManager` to search the database and a `DocumentManager` to read, update, and delete documents. You create manager objects using factory methods on `DatabaseClient`, such as `newQueryManager` and `newDocumentManager`.

1.4.2 Expected Database Client Lifetime

Best practice is to maintain a single, shared reference to a `DatabaseClient` object for the lifetime of your application’s interaction MarkLogic, rather than frequently creating and destroying client objects.

You need multiple `DatabaseClient` objects if you need to connect to multiple databases or to connect to MarkLogic as multiple users. You must create a different `DatabaseClient` instance for each combination of (host, port, database, authentication context). Again, it is best to keep these instances around throughout their potential useful lifetime, rather than repeatedly recreating them.

You can one `DatabaseClient` object across multiple threads. After initial configuration, a `DatabaseClient` object is thread safe.

1.4.3 Connection Management and Configuration

Internally, the Java Client API maintains an `OkHttpClient` connection pool that is shared by all DatabaseClient objects. The connection pool efficiently re-uses connections whether you use a single `DatabaseClient` instance throughout the lifetime of your application or create and discard `DatabaseClient` objects on demand.

Whenever a `DatabaseClient` object makes a request to MarkLogic, an available connection is drawn from the connection pool. New connections are created on demand, as needed.

A `DatabaseClient` object returns its connection to the pool once it receives and processes the HTTP request on whose behalf it claimed the connection. A connection in the pool persists until it is explicitly released or times out due to idleness. The default maximum idle time is 5 minutes.

No state information is maintained with a connection. All cookies are discarded unless a multi-statement (multi-request) transaction is in use. The cookies associated with a multi-statement transaction are cached on the transaction object rather than with the connection.

You can adjust the connection pool configuration by implementing `OkHttpClientConfigurator` and calling its `configure` method. However, such adjustments depend on Java Client API internals and will be ignored if a future version of the API uses a different HTTP client implementation.

1.4.4 Creating a Database Client

To create a database client, use the `com.marklogic.client.DatabaseClientFactory.newClient()` method. For example, the following client connects to the default content database associated with the REST instance on port 8000 of localhost using digest authentication.

```
DatabaseClient client =
  DatabaseClientFactory.newClient(
    "localhost", 8000,
    new DatabaseClientFactory.DigestAuthContext("myuser", "mypassword"));
```

You can also create clients that connect to a specific content database. For example, the following client also connects to the REST instance on port 8000 of localhost, but all operations are performed against the database “MyDatabase”:

```
DatabaseClient client =
  DatabaseClientFactory.newClient(
    "localhost", 8000, "MyDatabase",
    new DatabaseClientFactory.DigestAuthContext("myuser", "mypassword"));
```

Note: To use a database other than the default database associated with the REST instance requires a user with the following privilege or the equivalent role: `http://marklogic.com/xdmp/privileges/xdmp-eval-in`.

The `host` and `port` values must be those of a REST API instance. When you install MarkLogic, a REST API instance associated with the Documents database is pre-configured for port 8000. You can also create your own instance.

The authentication context object should match the configuration of the REST API instance. For more details, see “Authentication and Connection Security” on page 20.

1.4.5 Connecting Through a Load Balancer

When your application connects to MarkLogic through a load balancer, you should follow these guidelines:

- Configure your `DatabaseClient` objects to make a `GATEWAY` type connection. This tells the Java Client API that direct connections to hosts in your MarkLogic cluster are not available.
- Configure your load balancer and MarkLogic cluster timeouts to be consistent with each other. Unavailable hosts should be invalidated by the load balancer only after the MarkLogic host timeout. Cookies should expire only after the MarkLogic session timeout.

For most Java Client API operations, the connection type is transparent. However, features such as the Data Movement SDK need to know whether or not all traffic must go through a gateway host.

The default connection type for a `DatabaseClient` is `DIRECT`, meaning that the Java Client API can make direct connections to hosts in your MarkLogic cluster if necessary.

To configure a `DatabaseClient` for a gateway connection, pass a `DatabaseClient.ConnectionType` value of `GATEWAY` as the last parameter to `DatabaseClientFactory.newClient`. For example:

```
DatabaseClient client =
  DatabaseClientFactory.newClient(
    "localhost", 8000, "MyDatabase",
    new DatabaseClientFactory.DigestAuthContext("myuser", "mypassword"),
    DatabaseClient.ConnectionType.GATEWAY);
```

For additional, context-specific load balancer guidelines, see the following topics:

- Multi-statement transactions: “Transaction Management When Using a Load Balancer” on page 268.
- Asynchronous batch-oriented document operations: “Working with a Load Balancer” on page 134.

1.4.6 Releasing a Database Client

When you no longer need a client and want to release connection resources, use the `DatabaseClient` object’s `release()` method.

```
client.release();
```

DatabaseClient objects efficiently manage connection resources and are expected to be long lived. You do not need to release and re-create client objects just because your application might not require a connection for an extended time. For more details, see “Expected Database Client Lifetime” on page 17 and “Connection Management and Configuration” on page 17.

1.5 Authentication and Connection Security

This section provides an overview of several methods for securing the communication between your client application and MarkLogic. See the following topics for details:

- Creating a SecurityContext Object
- Using Kerberos Authentication
- Connecting to MarkLogic with SSL
- Using SAML Authentication

1.5.1 Creating a SecurityContext Object

One of the inputs to `DatabaseClientFactory.newClient` is a `SecurityContext` object. This object tells the API what credentials to use to authenticate with MarkLogic. You can select from authentication methods such as Kerberos, digest, and basic.

For example, the database client created by the following statement uses digest authentication. The username and password are those of a user configured into MarkLogic.

```
import com.marklogic.client.DatabaseClientFactor.DigestAuthContext;
...
DatabaseClient client = DatabaseClientFactory.newClient(
    "localhost", 8000, new DigestAuthContext(username, password));
```

The authentication context object should match the configuration of the REST API instance. Kerberos based authentication is most secure. Basic authentication sends the password in obfuscated, but not encrypted, mode. Digest authentication encrypts passwords sent over the network.

You can connect to MarkLogic using SSL by attaching SSL configuration information to the security context. For details, see “Connecting to MarkLogic with SSL” on page 22.

For more information about user authentication, see Authenticating Users in the *Security Guide*.

1.5.2 Using Kerberos Authentication

Use the following steps to configure your MarkLogic installation and client application environment for Kerberos authentication:

- Configuring MarkLogic to Use Kerberos
- Configuring Your Client Host for Kerberos
- Creating a Database Client that Uses Kerberos

Your client host must be running Linux in order to use Kerberos with the Java Client API.

1.5.2.1 Configuring MarkLogic to Use Kerberos

Before you can use Kerberos authentication, you must configure MarkLogic to use external security. If your installation is not already configured for Kerberos, you must perform at least the following steps:

1. Create a Kerberos external security configuration object. For details, see Creating an External Authentication Configuration Object in the *Security Guide*.

2. Create a Kerberos keytab file and install it in your MarkLogic installation. For details, see Creating a Kerberos keytab File in the *Security Guide*.

3. Create one or more users associated with an external name. For details, see Assigning an External Name to a User in the *Security Guide*.

4. Configure your App Server to use "kerberos-ticket" authentication. For details, see Configuring an App Server for External Authentication in the *Security Guide*.

For more details, see External Security in the *Security Guide*.

1.5.2.2 Configuring Your Client Host for Kerberos

On the client, the Java Client API must be able to access a Ticket-Granting Ticket (TGT) from the Kerberos Key Distribution Center. The API uses the TGT to obtain a Kerberos service ticket.

Follow these steps to make a TGT available to the client application:

1. Install MIT Kerberos in your client environment if it is not already installed. You can download MIT Kerberos from http://www.kerberos.org/software/index.html.

2. If this is a new installation of MIT Kerberos, configure your installation by editing the `krb5.conf` file. For details, see https://web.mit.edu/kerberos/krb5-1.15/doc/admin/conf_files/krb5_conf.html.

   On Linux, Java expects this file to be located in `/etc/` by default. Java uses the conf file to determine your default realm and the KDC for that realm.

   If your `krb5.conf` file contains a setting for `default_ccache_name`, the value must be a file reference of the form `FILE:/tmp/krb5cc_%{uid}`. This is required because the Java Client API sets the `useTicketCache` option of `Krb5LoginModule` to `true`. For more details, see the javadoc for `com.sun.security.auth.module.Krb5LoginModule`.

3. Use `kinit` or a similar tool on your client host to create and cache a TGT with the Kerberos Key Distribution Center. The principal supplied to `kinit` must be one you associated with a MarkLogic user when performing the steps in "Configuring MarkLogic to Use Kerberos" on page 21.

For more details, see the following topics:

- Using Kerberos with Java: http://docs.oracle.com/javase/8/docs/technotes/guides/security/jgss/tutorials/KerberosReq.html
- Kinit command: https://web.mit.edu/kerberos/krb5-1.15/doc/user/user_commands/kinit.html
- Obtaining a ticket: http://web.mit.edu/kerberos/krb5-current/doc/user/tkt_mgmt.html#obtaining-tickets-with-kinit
- Krb5LoginModule javadoc: https://docs.oracle.com/javase/9/docs/api/com/sun/security/auth/module/Krb5LoginModule.html

1.5.2.3 Creating a Database Client that Uses Kerberos

In your client application, use `KerberosAuthContext` for your security context object. For example:

```
import com.marklogic.client.DatabaseClientFactory.KerberosAuthContext;
...
DatabaseClient client = DatabaseClientFactory.newClient(
    "localhost", 8000, new KerberosAuthContext());
```

You do not need to pass an explicit `externalName` parameter to `KerberosAuthContext` unless you have multiple principals authenticated in your ticket cache and need to specify which one to use.

For a working example, see the project on GitHub:

Note: The working example includes comments that provide suggestions for setting up a Kerberos configuration in a production environment.

https://github.com/marklogic/java-client-api/blob/master/marklogic-client-api/src/main/java/com/marklogic/client/example/cookbook/KerberosSSLClientCreator.java

1.5.3 Connecting to MarkLogic with SSL

You can use the security context to specify whether or not to use a secure SSL connection to communicate with MarkLogic. The App Server you connect to must also be configured to accept SSL connections. By default, the Java Client API does not use SSL.

For example, the database client created by the following statement uses digest authentication and an SSL connection:

```
// create a trust manager
// (note: a real application should verify certificates. This
// naive trust manager which accepts all the certificates should be replaced
// by a valid trust manager or get a system default trust manager
// which would validate whether the remote authentication credentials
// should be trusted or not.)
TrustManager naiveTrustMgr[] = new X509TrustManager[] {
    new X509TrustManager() {
      @Override
      public void checkClientTrusted(X509Certificate[] chain, String authType)
```

```
        {
        }

        @Override
        public void checkServerTrusted(X509Certificate[] chain, String authType)
        {
}

        @Override
        public X509Certificate[] getAcceptedIssuers() {
          return new X509Certificate[0];
        }
    }
};

// create an SSL context
SSLContext sslContext = SSLContext.getInstance("TLSv1.2");
/*
 * Here, we use a naive TrustManager which would accept any certificate
 * which the server produces. But in a real application, there should be a
 * TrustManager which is initialized with a Keystore which would determine
 * whether the remote authentication credentials should be trusted or not.
 *
 * If we init the sslContext with null TrustManager, it would use the
 * <java-home>/lib/security/cacerts file for trusted root certificates, if
 * javax.net.ssl.trustStore system property is not set and
 * <java-home>/lib/security/jssecacerts is not present. See this link for
 * more information on TrustManagers -
 * http://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/
 * JSSERefGuide.html
 *
 * If self signed certificates, signed by CAs created internally are used,
 * then the internal CA's root certificate should be added to the keystore.
 * See this link -
 * https://docs.oracle.com/cd/E19226-01/821-0027/geygn/index.html for adding
 * a root certificate in the keystore.
 */
sslContext.init(null, naiveTrustMgr, null);

// create the client
// (note: a real application should use a COMMON, STRICT, or implemented
hostname verifier)

DatabaseClient client = DatabaseClientFactory.newClient(
  props.host, props.port,
  new DigestAuthContext(props.writerUser, props.writerPassword)
    .withSSLContext(sslContext, (X509TrustManager) naiveTrustMgr[0])
    .withSSLHostnameVerifier(SSLHostnameVerifier.ANY));
```

The `SSLContext` object represents a secure socket protocol implementation which acts as a factory for secure socket factories. For more information about creating and working with `SSLContext` objects, see Accessing SSL-Enabled XDBC App Servers in the *XCC Developer's Guide*.

For even more security, you can also include a `DatabaseClientFactory.SSLHostnameVerifier` object to check if a hostname is acceptable.

For a working example, see the project on GitHub:

Note: The working example includes comments that provide suggestions for configuring SSL in a production environment.

https://github.com/marklogic/java-client-api/blob/master/marklogic-client-api/src/main/java/com/marklogic/client/example/cookbook/SSLClientCreator.java

For more information about secure communication with MarkLogic, see the *Security Guide*.

1.5.4 Using SAML Authentication

Your client application is responsible for acquiring a SAML assertions token from the SAML Identity Provider (IDP). You can then use the SAML assertions token to make requests to the MarkLogic App Server with the MarkLogic Client Java API. That division of responsibility makes it possible for your application to adapt to a wide variety of possible SAML scenarios and IDPs.

After configuring the MarkLogic App Server to authenticate with the SAML IDP, specify a `SAMLAuthContext` as the `SecurityContext` when calling `DatabaseClientFactory` to create a new `DatabaseClient`.

You can construct a `SAMLAuthContext` in any of three ways, depending on your approach to authorization:

- If you plan to finish using the `DatabaseClient` before the SAML assertions token expires, you can call the `SAMLAuthContext` constructor with the SAML assertions token.
- If you need to extend the expiration of the SAML assertions token before you finish using the `DatabaseClient`, you can call the `SAMLAuthContext` constructor with an `ExpiringSAMLAuth` object and a callback that renews the SAML assertions token with the SAML IDP.

  The `ExpiringSAMLAuth` object provides getters for the SAML assertions token and the expiration timestamp. Your client application can construct an `ExpiringSAMLAuth` object by calling the `SAMLAuthContext.newExpiringSAMLAuth` factory method.

  The renewer callback conforms to the `SAMLAuthContext.RenewerCallback` functional interface by taking the initial `ExpiringSAMLAuth` object as input and returning an Instant with the new expiration timestamp for the renewed SAML assertions token, as in the following example:

```
class MyClass {
   Instant renewer(ExpiringSAMLAuth authorization) {
     .... call to IDP ....
   }
}
```

- If you need to get a new SAML assertions token before you finish using the `DatabaseClient`, you can call the `SAMLAuthContext` constructor with a callback that authorizes with the SAML IDP by getting a new SAML assertions token.

  The authorizer callback conforms to the `SAMLAuthContext.AuthorizerCallback` functional interface that takes an `ExpiringSAMLAuth` object as input and returns an `ExpiringSAMLAuth` object with the new SAML assertions token and an expiration timestamp, as shown in the following example:

  ```
  class MyClass {
     ExpiringSAMLAuth authorizer(ExpiringSAMLAuth previous) {
       .... call to IDP ....
     }
  }
  ```

  On the first call, the `ExpiringSAMLAuth` parameter is null because no existing authorization exists. Your callback can construct an `ExpiringSAMLAuth` object by calling the `SAMLAuthContext.newExpiringSAMLAuth` factory method.

Tradeoffs to consider when choosing whether to renew or reauthorize include the following:

- The renewer callback executes in a background thread, allowing continued requests to the MarkLogic appserver while renewing the SAML assertions token, improving utilization and performance. Extending the life of the SAML assertions token, however, could increase vulnerability in less secure environments.
- The authorizer callback blocks requests to the MarkLogic appserver while getting a new SAML assertions token, reducing utilization and performance but maintaining the highest level of security.

You can reduce the expiration time to allow for network latency and the IDP response generation. Renewer and authorizer callbacks are called in advance of the stated expiration time to reduce the possibility that the SAML assertions token expires as a request is sent to the MarkLogic appserver.

If you need to maintain state between calls to a renewer or authorizer callback, you can implement the `ExpiringSAMLAuth` interface with your own class instead of calling the `SAMLAuthContext.newExpiringSAMLAuth` factory method to construct a default instance.

Apart from the specifics of acquiring the SAML assertions token, the use of a `DatabaseClient` remains the same:

- Multiple threads can use the same `DatabaseClient` object.
- `DatabaseClient` objects can be created with different authorizations (including different SAML assertions tokens).

1.6 A Basic "Hello World" Method

The following code is a basic method that creates a new document in the database. Digest authentication is used in this example; for more details, see "Authentication and Connection Security" on page 20.

```
public static void run(String host, int port, String user, String
                       password, Authentication authType) {

// Create the database client
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));

// Make a document manager to work with text files.
TextDocumentManager docMgr = client.newTextDocumentManager();

// Define a URI value for a document.
String docId = "/example/text.txt";

// Create a handle to hold string content.
StringHandle handle = new StringHandle();

// Give the handle some content
handle.set("A simple text document");

// Write the document to the database with URI from docId
// and content from handle
docMgr.write(docId, handle);

// release the client
client.release();
}
```

The above code is a slightly modified version of the `run` method from the `com.marklogic.client.example.cookbook.ClientCreator` cookbook example. It, along with a number of other basic example applications for the Java API, is located in `example/com/marklogic/client/example/cookbook` directory found in the zip file containing the Java API.

1.7 Document Managers

Different document formats are handled by different *document manager* objects, which serve as an interface between documents and the database connection. The package `com.marklogic.client.document` includes document managers for binary, XML, JSON, and text. If you don't know the document format, or need to work with documents of multiple formats, use a generic document manager. `DatabaseClient` instances have factory methods to create a new `com.marklogic.client.document.DocumentManager` of any subtype.

```
BinaryDocumentManager binDocMgr = client.newBinaryDocumentManager();
XMLDocumentManager XMLdocMgr = client.newXMLDocumentManager();
JSONDocumentManager JSONDocMgr = client.newJSONDocumentManager();
TextDocumentManager TextDocMgr = client.newTextDocumentManager();
```

```
GenericDocumentManager genericDocMgr =
client.newGenericDocumentManager();
```

Your application only needs to create one document manager for any given type of document, no matter how many of that type of document it works with. So, even if you expect to work with, say, 1,000,000,000 JSON documents, you only need to create one `JSONDocumentManager` object.

Document managers are thread safe once initially configured; no matter how many threads you have, you only need one document manager per document type.

If you make a mistake and try to use the wrong type of document with a document manager, the result depends on the combination of types. For example, a `BinaryDocumentManager` will try to interpret the document content as binary. `JSONDocumentManager` and `XMLDocumentManager` are the most particular, since if a document is not in their format, it will not parse. Most of the time, you will get an exception error, with `FailedRequestException` the default if the manager cannot determine the document type.

1.8 Streaming

To stream, you supply an `InputStream` or `Reader` for the data source, not only when reading from the database but also when writing to the database. This approach allows for efficient write operations that do not buffer the data in memory. You can also use an `OutputWriter` to generate data as the API is writing the data to the database.

When reading from the database using a stream, be sure to close the stream explicitly if you do not read all of the data. Otherwise, the resources that support reading continue to exist.

1.9 Using Handles for Input and Output

The Java Client API uses Handles to for I/O when interacting with MarkLogic. See the following topics for more details:

- Handle Overview
- Specifying Content Format
- Handle Type Quick Reference
- Handle Example

1.9.1 Handle Overview

Content handles are key to working with the Java Client API. Handles make use of the Adapter design pattern to enable strongly typed reading and writing of a diverse and extensible set of content formats. For example, you can create a `com.marklogic.client.io.DOMHandle` to read or write XML DOM data.

```
// reading
XMLDocumentManager docMgr = client.newXMLDocumentManager();
Document doc = docMgr.read(docURI, new DOMHandle()).get();
```

```
// writing
docMgr.write(docURI, new DOMHandle(someDocument));
```

You can also create a `com.marklogic.client.io.JacksonHandle` to read or write JSON data.

```
// reading
JSONDocumentManager JSONDocMgr = client.newJSONDocumentManager();
JsonNode node = JSONDocMgr.read(docURI, new JacksonHandle()).get();

// writing
JSONDocMgr.write(docURI, new JacksonHandle(someJsonNode));
```

The Java Client API pre-defines many handle implementations. The following packages contain handle classes:

- `com.marklogic.client.io` - Handles classes for standard representations such as `String`, `File`, and `DOM`.
- `com.marklogic.extra` - Handle classes for 3rd party formats such as DOM4J and GSON. Using these handle classes requires 3rd party libraries that are not included in the Java Client API distribution.

Some handles support both read and write operations. For example, you can use a `FileHandle` for reading and writing files. Some handles have a special purpose. For example, you use `SearchHandle` for processing the results of a search operation. For a complete list of handles and what they do, see the `com.marklogic.client.io` package in the *Java Client API Documentation*.

> **Note:** Handles are *not* thread safe. Whenever you create a new thread, you will have to also create new handle objects to use while in that thread.

Some Java Client API methods enable you to use I/O short cuts that do not require explicit creation of a handle. These shortcut methods always have an “As” suffix, such as “readAs”. For example, the `XMLDocumentManager.read` method shown above has an `XMLDocumentManager.readAs` counterpart that implicitly creates the handle for you. For example:

```
// reading
Document doc = docMgr.readAs(docURI, Document.class);

// writing
docMgr.writeAs(docURI, someDocument);
```

Likewise, the `JSONDocumentManger.read` method shown above has an `JSONDocumentManager.readAs` counterpart that implicitly creates the handle for you.

```
// reading
JsonNode node = JSONDocMgr.readAs(docURI, JsonNode.class);

// writing
JSONDocMgr.writeAs(docURI, someJsonNode);
```

These shortcut methods are not more efficient, but they can improve the readability of your code. For more details, see "Shortcut Methods as an Alternative to Creating Handles" on page 31.

1.9.2 Specifying Content Format

Some handles can be used with multiple document formats. For example, an `InputStream` can provide content in any format, so `InputStreamHandle` can be used for any document format. Where content format is not explicit in the handle type, use the handle's `setFormat` method to specify it. For example, the following call tells the Java Client API that the handle can be used with JSON content:

```
new InputStreamHandle().setFormat(Format.JSON);
```

You cannot set a format for all handle types. For example, a DOMHandle can only be used for reading and writing XML, so you cannot specify a format.

1.9.3 Handle Type Quick Reference

Not all handles support all content types. In addition, though most handles can be used for either reading or writing, some are more limited. This section provides a quick guide to the content formats, operations, and data types supported by each handle class. Special purpose handle classes, such as `SearchHandle`, are not included.

Handle Class	Content Format				Supported Java Type
	XML	Text	JSON	Binary	
`BytesHandle`	RW	RW	RW	RW	`byte[]`
`DocumentMetadataHandle`	RW				MarkLogic proprietary XML format; for details, see XML Metadata Format in the *REST Application Developer's Guide*.
`DOMHandle`	RW				`org.w3c.dom.Document`
`FileHandle`	RW	RW	RW	RW	`java.io.File`
`InputSourceHandle`	RW				`org.xml.sax.InputSource`
`InputStreamHandle`	RW	RW	RW	RW	`java.io.InputStream`
`JacksonHandle`			RW		`com.fasterxml.jackson.databind.JsonNode`
`JacksonDatabindHandle`			RW		your POJO class

Handle Class	Content Format				Supported Java Type
	XML	Text	JSON	Binary	
JacksonParserHandle			RW		com.fasterxml.jackson.core.JsonParser
JAXBHandle	RW				your POJO class
OutputStreamHandle	W	W	W	W	java.io.OutputStream
ReaderHandle	RW	RW	RW		java.io.Reader
SourceHandle	RW				javax.xml.transform.Source
StringHandle	RW	RW	RW		String
XMLEventReaderHandle	RW				javax.xml.stream.XMLEventReader
XMLStreamReaderHandle	RW				javax.xml.stream.XMLStreamReader

1.9.4 Handle Example

The following code uses a `DOMHandle` to read an XML document from the server into an in-memory DOM object:

```
XMLDocumentManager docMgr = client.newXMLDocumentManager();
DOMHandle handle = new DOMHandle();
docMgr.read(docURI, handle);
org.w3c.dom.Document document = handle.get();
```

The following code uses a `JacksonHandle` to read a JSON document from the server into an in-memory JsonNode:

```
JSONDocumentManager JSONDocMgr = client.newJSONDocumentManager();
JacksonHandle handleJson = new JacksonHandle();
JSONDocMgr.read(docURI, handleJson);
com.fasterxml.jackson.databind.JsonNode node = handleJson.get();
```

The following code uses a `DOMHandle` to write an XML document to MarkLogic. Assume `document` is some previously initialized in-memory XML DOM document.

```
XMLDocumentManager docMgr = client.newXMLDocumentManager();
DOMHandle handle = new DOMHandle();
handle.set(document);
docMgr.write(docId, handle);
```

The following code uses a `JacksonHandle` to write a JSON document to MarkLogic. Assume `node` is some previously initialized in-memory JsonNode document.

```
JSONDocumentManager JSONDocMgr = client.newJSONDocumentManager();
JacksonHandle handleJson = new JacksonHandle();
handleJson.set(node);
JSONDocMgr.write(docId, handleJson);
```

For additional examples, see the examples in the following packages. The source is available on GitHub. For details, see “Downloading the Library Source Code” on page 34.

- `com.marklogic.client.example.cookbook`
- `com.marklogic.client.example.handle`

1.10 Shortcut Methods as an Alternative to Creating Handles

Shortcut methods enable you to pass supported data types directly into or out of an operation without explicitly creating a handle to reference the data. These convenience methods can make your code more readable.

For more details, see the following topics:

- Understanding Shortcut Methods
- When to Choose Strongly Typed Over Shortcut
- Extending Shortcuts by Registering Handle Factories

1.10.1 Understanding Shortcut Methods

Many Java Client API classes and interfaces include “shortcut” methods of the form *operation*`As`, such as `readAs` or `writeAs`. These methods enable you to bypass the equivalent, more strongly typed methods that require you to pass in a handle. Using shortcut methods instead of handles can make your code more readable.

For example, the `XMLDocumentManager` and `JSONDocument Manager` interfaces includes both `read` and `readAs` methods such as the following:

```
// strongly typed, handle based
read(String docId, T contentHanlde)

// shortcut equivalent
readAs(String docId, Class<T> as)
```

This means you can read a document from the database using a call of either of the following forms:

```
// strongly typed, returns the populated DOMHandle object
DOMHandle handle = docMgr.read(docURI, new DOMHandle());

// shortcut, returns a DOM Document
Document doc = docMgr.readAs(docURI, Document.class);
```

```
// strongly typed, returns the populated JacksonHandle object
JacksonHandle handleJSON = JSONDocMgr.read(docURI, new
JacksonHandle());

// shortcut, returns a JsonNode
JsonNode node = JSONDocMgr.readAs(docURI, JsonNode.class);
```

Similarly, you can use `XMLDocumentManager` or `JSONDocumentManager` to write a document to the database using either of the following calls:

```
// strongly typed
docMgr.write(docURI, new DOMHandle(theDocument));

// shortcut
docMgr.writeAs(docURI, theDocument);

// strongly typed
JSONDocMgr.write(docURI, new JacksonHandle(theJsonNode));

// shortcut
JSONDocMgr.writeAs(docURI, theJsonNode));
```

Shortcut methods are not limited to document read and write operations. For example, you can use either `QueryManager.newRawCombinedQueryDefinition` or `QueryManager.newRawCombinedQueryDefinitionAs` to create a `RawCombinedQueryDefinition`.

1.10.2 When to Choose Strongly Typed Over Shortcut

Shortcut methods are the best choice in most cases because they improve the readability and maintainability of your code. However, you should keep the following points in mind:

- A shortcut method is not more efficient than the equivalent strongly typed method. Internally, a `Handle` is still created to manage the data.
- Using a shortcut method introduces a small risk because you're side-stepping the strong typing provided by a handle. For example, an exception is thrown if there is no handle type corresponding to class type you provide to the shortcut method.

The typing exposure is limited since the Java Client API pre-defines Handle classes for a broad range of types. You can register your own class-to-handle pairings to extend the supported classes. For details, see "Extending Shortcuts by Registering Handle Factories" on page 33.

Consider the strongly typed call form in the following cases:

- You want compile-time checking of input and output types.
- You want a slight increase in efficiency over a large number of requests.
- You need to control the MIME type or format of a handle.

1.10.3 Extending Shortcuts by Registering Handle Factories

Though you do not have to create a handle when using a shortcut method, the shortcut implementation still creates a `Handle` to manage the data.

For example, when you issue a shortcut call such as the following, the implementation creates a `DOMHandle` to receive the document read from the database.

```
docMgr.readAs(docURI, Document.class);
```

Similarly, the following implementation creates a `JacksonHandle` to receive the document read from the database.

```
JSONDocMgr.readAs(docURI, JsonNode.class);
```

This means that a shortcut method must be able to create a handle capable of handling the targeted class type. This capability is provided by a registry for handle factories. The shortcut method can query the registry for a handle factory that can process a particular class type. For details, see `DatabaseClientFactory.HandleFactoryRegistry` in the *Java Client API Documentation*.

The Java Client API automatically registers factories for the following handle classes. For details on the data types supported by each handle type, see the handle class documentation in the *Java Client API Documentation*.

`BytesHandle`	`InputStreamHandle`	`SourceHandle`
`DOMHandle`	`JacksonHandle`	`StringHandle`
`FileHandle`	`JacksonParserHandle`	`XMLEventReaderHandle`
`InputSourceHandle`	`ReaderHandle`	`XMLStreamReaderHandle`

If you create your own handle class, you can register a handle factory for it so that you can use shortcut methods on the classes supported by your handle class.

Note: Handle factory registration must be completed before you create a `DatabaseClient`.

You can use the same mechansim with a `JAXBHandle` factory to enable shortcut methods on POJOs. For example, if you have a POJO class named `Product`, then you can add it to the registry as follows:

```
DatabaseClientFactory.getHandleRegistry().register(
    JAXBHandle.newFactory(Product.class);
```

You can also use `JacksonDatabindHandle` factory to enable shortcut methods on POJOs.

```
DatabaseClientFactory.getHandleRegistry().register(
    JacksonDatabindHandle.newFactory(Product.class);
```

Then you can subsequently write `Product` POJOs to MarkLogic and read them back as follows:

```
XMLDocumentManager docMgr = client.newXMLDocumentManager();
Product product = // ...create a Product

docMgr.writeAs(docURI, Product.class);
// ...
product = docMgr.readAs(docURI, Product.class);
```

Likewise with a `JSONDocumentManager`:

```
JSONDocumentManager JSONDocMgr = client.newJSONDocumentManager();
Product product = // ...create a Product

JSONDocMgr.writeAs(docURI, Product.class);
// ...
product = JSONDocMgr.readAs(docURI, Product.class);
```

Note that the Java Client API also includes a POJO data binding capability as an alternative to managing your own POJOs with JAXB. Using this feature eliminates the need for the above registration. For more details, see “POJO Data Binding Interface” on page 226.

1.11 Thread Safety of the Java API

You should be aware of the following API characteristics with respect to thread safety:

- `DatabaseClient` is thread safe after initialization.
- The various manager classes are thread safe after initial configuration. Examples: `DocumentManager`, `QueryManager`, `ResourceManager`.
- Handles are not thread safe. Examples: `StringHandle`, `FileHandle`, `SearchHandle`.
- Builders are not thread safe. Examples: `DocumentPatchBuilder`, `StructuredQueryBuilder`.

For example, you can create a `DocumentManager` for manipulating XML documents and share it across multiple threads. Similarly, you can create a `QueryManager`, set the page length, and then share it between multiple threads.

Handles can be used across multiple requests within the same thread, but cannot be used across threads, so whenever you create a new thread, you must create new Handle objects to use in that thread.

1.12 Downloading the Library Source Code

The Java API is an open source project. Though you do not need the source code to use the library, the source is available from GitHub at the following URL:

https://github.com/marklogic/java-client-api

Assuming you have a Git client and the `git` command is on your path, you can download a local copy of the latest source using the following command:

```
git clone https://github.com/marklogic/java-client-api.git
```

2.0 Single Document Operations

This chapter describes how to create, delete, update, and read a single document content and/or its metadata using the Java Client API. The Java Client API also enables you to work with multiple documents in a single request, as described in “Synchronous Multi-Document Operations” on page 70 and “Asynchronous Multi-Document Operations” on page 92.

When working with documents, it is important to keep in mind the difference between a document on your client and a document in the database. In particular, any changes you make to a document’s content and metadata on the client do not persist between sessions. Only if you write the document out to the database do your changes persist.

This chapter includes the following sections:

2.1 Document Creation

Document creation is not done via a document creation method. When you first write content via a Manager object to a document in the database as identified by its URI, MarkLogic Server creates a document in the database with that URI and content.

> **Note:** To call `write()`, an application must authenticate as a user with at least one of the `rest-writer` or `rest-admin` roles (or as a user with the `admin` role).

This section describes the following about document creation operations:

2.1.1 Writing an XML or JSON Document To The Database

Note that no changes you make to a document or its metadata persist until you write the document out to the database. Within your application, you are only manipulating it within system memory, and those changes will vanish when the application ends. The database content is constant until and unless a write or delete operation changes it.

The basic steps needed to write a document are:

1. If you have not already done so, connect to the database, storing the connection in a `com.marklogic.client.DatabaseClient` object. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

2. If you have not already done so, use the `DatabaseClient` object to create a `com.marklogic.client.document.DocumentManager` object of the appropriate subclass for the document content you want to access (XML, text, JSON, binary, generic).

   a. In this example code, an `XMLDocumentManager`.

```
XMLDocumentManager XMLDocMgr = client.newXMLDocumentManager();
```

   b. In this example code, an `JSONDocumentManager`.

```
JSONDocumentManager JSONDocMgr = client.newJSONDocumentManager();
```

3. Get the document's content. For example, by using an `InputStream`.

```
FileInputStream docStream = new FileInputStream(
                                "data"+File.separator+filename);
```

4. Create a handle associated with the input stream to receive the document's content. How you get content determines which handle you use. Use the handle's `set()` method to associate it with the desired stream.

```
InputStreamHandle handle = new InputStreamHandle(docStream);
```

5. Write the document's content by calling a `write()` method on the `DocumentManager`, with arguments of the document's URI and the handle.

   a. Calling a `write()` method on the `XMLDocumentManager`:

```
XMLDocMgr.write(docId, handle);
```

   b. Calling a `write()` method on the `JSONDocumentManager`:

```
JSONDocMgr.write(docId, handle);
```

6. When finished with the database, release the connection resources by calling the `DatabaseClient` object's `release()` method.

```
client.release();
```

2.1.2 Creating a Text Document In the Database

This procedure outlines a very basic creation operation for a simple text document is as follows:

1. Create a `com.marklogic.client.DatabaseClient` for the database. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

2. Create a `com.marklogic.client.document.DocumentManager` object of the appropriate format for your document; text, binary, JSON, XML, or generic if you are not sure.

```
TextDocumentManager TextDocMgr = client.newTextDocumentManager();
```

3. For convenience's sake, set a variable to your new document's URI. This is not required; the raw string could be used wherever `docId` is used.

```
String docId = "/example/text.txt";
```

4. As discussed previously in "Using Handles for Input and Output" on page 27, within MarkLogic Java applications you use handle objects to contain a document's content and metadata. Since this is a text document, we will use a `com.marklogic.client.io.StringHandle` to contain the text content. After creation, set the handle's value to the document's initial content.

```
StringHandle handle = new StringHandle();
handle.set("A simple text document");
```

5. Write the document content out to the database. This creates the document in the database if it is not already there (if it is already there, it updates the content to whatever is in the `handle` argument). The identifier for the document is the value of the `docId` argument.

```
TextDocMgr.write(docId, handle);
```

6. When finished with the database, release the connection resources by calling the `DatabaseClient` object's `release()` method.

```
client.release();
```

2.1.3 Automatically Generating Document URIs

MarkLogic Server can automatically generate database URIs for documents inserted using the Java API. You can only use this feature to create new documents. To update an existing document, you must know the URI.

To insert a document with a generated URI, use a `com.marklogic.client.document.DocumentUriTemplate` with `DocumentManager.create()`, as described by the following procedure.

1. If you have not already done so, connect to the database, storing the connection in a `com.marklogic.client.DatabaseClient` object. For example, if using digest authentication:

   ```
   DatabaseClient client = DatabaseClientFactory.newClient(
       host, port, new DigestAuthContext(username, password));
   ```

2. If you have not already done so, use the `DatabaseClient` object to create a `com.marklogic.client.document.DocumentManager` object of the appropriate subclass for the document content you want to access (XML, text, JSON, binary, generic).

   a. In this example code, an `XMLDocumentManager`.

   ```
   XMLDocumentManager XMLDocMgr = client.newXMLDocumentManager();
   ```

   b. In this example code, an `JSONDocumentManager`.

   ```
   JSONDocumentManager JSONDocMgr = client.newJSONDocumentManager();
   ```

3. Create a `DocumentUriTemplate` using the document manager. Specify the extension suffix for the URIs created with this template. Do not include a "." separator.

   a. The following example creates a template that generates URIs ending with `".xml"`.

   ```
   DocumentUriTemplate templateXML =
   XMLDocMgr.newDocumentUriTemplate("xml");
   ```

   b. The following example creates a template that generates URIs ending with `".json"`.

   ```
   DocumentUriTemplate templateJSON =
   JSONDocMgr.newDocumentUriTemplate("json");
   ```

4. Optionally, specify additional URI template attributes, such as a database directory prefix and document format. The following example specifies a directory prefix of "/my/docs/".

   ```
   templateXML.setDirectory("/my/docs/");
   // Or
   templateJSON.setDirectory("/my/docs/");
   ```

5. Get the document's content. For example, by using an `InputStream`.

```
FileInputStream docStream =
    new FileInputStream("data" + File.separator + filename);
```

6. Create a handle associated with the input stream to receive the document's content. How you get content determines which handle you use. Use the handle's `set()` method to associate it with the desired stream.

```
InputStreamHandle handle = new InputStreamHandle(docStream);
```

7. Insert the document into the database by calling a `create()` method on the `DocumentManager`, passing in a URI template and the handle. Use the returned `DocumentDescriptor` to obtain the generated URI.

```
DocumentDescriptor descXML = XMLDocMgr.create(templateXML, handle);
// Or
DocumentDescriptor descJSON = JSONDocMgr.create(templateJSON,handle);
```

8. When finished with the database, release the connection resources by calling the `DatabaseClient` object's `release()` method.

```
client.release();
```

2.1.4 Format-Specific Write Capabilities

When inserting or updating a binary document, you can request metadata extraction using `BinaryDocumentManager.setMetadataExtraction`. For an example, see "Writing A Binary Document" on page 43.

When inserting or updating an XML document, you can request XML repair using `XMLDocumentManager.setDocumentRepair`.

See the *Java Client API Documentation* for details.

2.2 Document Deletion

To delete one or more documents, call `DocumentManager.delete` and pass in the URI(s) of the documents.

> **Note:** To delete documents, an application must authenticate as a user with at least one of the `rest-writer` or `rest-admin` roles (or as a user with the `admin` role).

The following example shows how to delete an XML document from the database.

1. Create a `com.marklogic.client.DatabaseClient` for connecting to the database.For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

2. If you have not already done so, use the `DatabaseClient` object to create a `com.marklogic.client.document.DocumentManager` object of the appropriate subclass for the document format (XML, text, JSON, or binary).

   a. In this example code, an `XMLDocumentManager`.

   ```
   XMLDocumentManager XMLDocMgr = client.newXMLDocumentManager();
   ```

   b. In this example code, an `JSONDocumentManager`.

   ```
   JSONDocumentManager JSONDocMgr = client.newJSONDocumentManager();
   ```

3. Delete the document(s). For example, the following statement deletes 2 documents:

   ```
   XMLDocMgr.delete("/example/doc1.xml", "/example/doc2.json");
   // Or
   JSONDocMgr.delete("/example/doc1.xml", "/example/doc2.json");
   ```

4. When finished with the database, release the connection resources by calling the `DatabaseClient` object's `release()` method.

   ```
   client.release();
   ```

2.3 Reading Document Content

Reading requires a handle to access document content.

Note that no changes you make to a document or its metadata persist until you write the document out to the database. Within your application, you are only manipulating it on the client, and those changes will vanish when the application ends. The database content is persistent until and unless a write or delete operation changes it.

If you read content with a stream, you must close the stream when done. If you do not close the stream, HTTP clients do not know that you are finished and there are fewer connections available in the connection pool.

The basic steps to read a document from the database are:

1. Create a `com.marklogic.client.DatabaseClient` for connecting to the database.For example, if using digest authentication:

   ```
   DatabaseClient client = DatabaseClientFactory.newClient(
       host, port, new DigestAuthContext(username, password));
   ```

2. If you have not already done so, use the `DatabaseClient` object to create a `com.marklogic.client.document.DocumentManager` object of the appropriate subclass for the document format (XML, text, JSON, or binary).

   a. In this example code, an `XMLDocumentManager`.

   ```
   XMLDocumentManager XMLDocMgr = client.newXMLDocumentManager();
   ```

   b. In this example code, an `JSONDocumentManager`.

   ```
   JSONDocumentManager JSONDocMgr = client.newJSONDocumentManager();
   ```

3. Create a handle to receive the document's content. For information on handles and the wide variety of handle types, see "Using Handles for Input and Output" on page 27.

   a. This example uses a `com.marklogic.client.io.DOMHandle` object.

   ```
   DOMHandle handleXML = new DOMHandle();
   ```

   b. This example uses a `com.marklogic.client.io.JacksonHandle` object.

   ```
   JacksonHandle handleJSON = new JacksonHandle();
   ```

4. Read the document's content by calling a `read()` method on the `DocumentManager`, with arguments of the document's URI and the handle. Here, assume `docId` contains the document's URI.

   ```
   XMLDocMgr.read(docId, handleXML);
   // Or(
   JSONDocMgr.read(docId, handleJSON);
   ```

5. Access the content by calling a `get()` method on the handle.

   a. For example, `DOMHandle.get` returns a W3C `Document` object.

   ```
   Document document = handleXML.get();
   ```

   b. For example, `JacksonHandle.get` returns a `JsonNode` object.

   ```
   JsonNode node = handleJSON.get();
   ```

6. When finished with the database, release the connection resources by calling the `DatabaseClient` object's `release()` method.

   ```
   client.release();
   ```

2.4 Writing A Binary Document

To insert or update a binary document, use a handle containing your binary content with `com.marklogic.client.document.BinaryDocumentManager`. You can use any handle that implements `BinaryWriteHandle`, such as `BytesHandle` or `FileHandle`.

No metadata extraction is performed by default. You can request metadata extraction and specify how it is saved by calling `BinaryDocumentManager.setMetadataExtraction()`.

The following example reads a JPEG image from a file named `my.png` and inserts it into the database as a binary document with URI `/images/my.png`. During insertion, metadata is extracted from the binary content and saved as document properties.

```
String docId = "/example/my.png";
String mimetype = "image/png";

BinaryDocumentManager docMgr = client.newBinaryDocumentManager();
docMgr.setMetadataExtraction(MetadataExtraction.PROPERTIES);

docMgr.write(
    docId,
    new FileHandle().with(new File("my.png")).withMimetype(mimetype)
  );
```

2.5 Reading Content From A Binary Document

There are several ways to read content from a binary document.

To stream binary content, use InputStream as follows:

```
InputStream byteStream =
    docMgr.read(docID, new InputStreamHandle()).get();
```

To buffer the binary content, use `com.marklogic.client.io.BytesHandle` object as follows:

```
byte[] buf = docMgr.read(docID, new BytesHandle()).get();
```

Or you can read only part of the content:

```
BytesHandle handle = new BytesHandle();
buf = docMgr.read(docId, handle, 9, 10).get();
```

2.6 Reading, Modifying, and Writing Metadata

Reading and writing document metadata from and to the database are very similar operations to reading and writing document content. Each requires calling methods on `com.marklogic.client.document.DocumentManager`. The handle for metadata can be a `DocumentMetadataHandle` to modify metadata in a POJO, or it can be raw XML or JSON.

You can perform operations on the metadata associated with documents such as collections, permissions, properties, and quality. This section describes those metadata operations and includes the following parts:

2.6.1 Document Metadata

The enum `DocumentManager.Metadata` enumerates the metadata categories (including `ALL`). The following are the metadata types covered by this enumeration:

- `COLLECTIONS`: Document collections, a non-hierarchical way of organizing documents in the database. For details, see “Collections Metadata” on page 46.
- `METADATAVALUES`: Key-value metadata, sometimes called “metadata fields”. For details, see “Values Metadata” on page 47.
- `PERMISSIONS`: Document permissions. For details, see “Permissions Metadata” on page 49.
- `PROPERTIES`: Document properties. Property-value pairs associated with the document. For details, see “Properties Metadata” on page 48.
- `QUALITY`: Document search quality. Helps determine which documents are of the best quality. For details, see “Quality Metadata” on page 48.

These metadata types are described in detail later in this chapter.

2.6.2 Reading Document Metadata

The basic steps needed to read a document’s metadata are:

1. If you have not already done so, create a `com.marklogic.client.DatabaseClient` for connecting to the database. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

2. If you have not already done so, use the `DatabaseClient` object to create a `com.marklogic.client.document.DocumentManager` object of the appropriate subclass for the document format (XML, text, JSON, or binary).

   a. In this example code, an `XMLDocumentManager`.

   ```
   XMLDocumentManager XMLDocMgr = client.newXMLDocumentManager();
   ```

   b. In this example code, an `JSONDocumentManager`.

   ```
   JSONDocumentManager JSONDocMgr = client.newJSONDocumentManager();
   ```

3. Create a `com.marklogic.client.io.DocumentMetadataHandle` object, which will receive the document's metadata. Alternately, you can create raw XML or JSON.

   ```
   DocumentMetadataHandle metadataHandle = new DocumentMetadataHandle();
   ```

4. If you also want to get the document's content, create a handle to receive it. Note that you need separate handles for a document's content and metadata.

   a. This example uses a `com.marklogic.client.io.DOMHandle` object.

   ```
   DOMHandle handleXML = new DOMHandle();
   ```

   b. This example uses a `com.marklogic.client.io.JacksonHandle` object.

   ```
   JacksonHandle handleJSON = new JacksonHandle();
   ```

5. Read the document's metadata by calling a `readMetadata()` method on the `DocumentManager`, with an argument of the metadata handle. Note that you can also call `read()` with an additional argument of a content handle so that it will read the metadata into the metadata handle and the content into the content handle in a single operation. To call `read()`, an application must authenticate as `rest-reader`, `rest-writer`, or `rest-admin`. Below, `docId` is a variable containing a document URI.

   a. Calling methods on a `XMLDocumentManager`:

   ```
   //read only the metadata into a handle
   XMLDocMgr.readMetadata(docId, metadataHandle);

   //read metadata and content
   XMLDocMgr.read(docId, metadataHandle, handleXML);
   ```

   b. Calling methods on a `JSONDocumentManager`

   ```
   //read only the metadata into a handle
   JSONDocMgr.readMetadata(docId, metadataHandle);
   ```

```
//read metadata and content
JSONDocMgr.read(docId, metadataHandle, handleJSON);
```

6. Access the metadata by calling `get()` methods on the metadata handle. Later sections in this chapter show how to access the other types of metadata.

```
DocumentCollections collections = metadataHandle.getCollections();
Document document = handleXML.get();
JsonNode node = handleJSON.get();
```

7. When finished with the database, release the connection resources by calling the `DatabaseClient` object's `release()` method.

```
client.release();
```

By default, `DocumentManager` reads and writes all categories of metadata. To read or write a subset of the metadata categories, configure `DocumentManager` by calling `setMetadataCategories()`. For example, to retrieve just collection metadata, make the following call before calling `DocumentManager.read` or `DocumentManager.readMetadata`:

```
docMgr.setMetadataCategories(DocumentManager.Metadata.COLLECTIONS);
```

2.6.3 Collections Metadata

Collections are a way to organize documents in a database. A collection defines a set of documents in the database. You can set documents to be in any number of collections either at the time the document is created or by updating a document. Searches against collections are both efficient and convenient. For more details on collections, see Collections in the *Search Developer's Guide*.

The Java API allows you to read and manipulate collections metadata using the `com.marklogic.client.io.DocumentMetadataHandle.DocumentCollections`. Collections are named by specifying a URI. A collection URI serves as an identifier, and it can be any valid URI.

The code in this section assumes a `DocumentManager` object of an appropriate type for the document, `docMgr`, and a string containing a document URI, `docId`, have been created.

To get all collections for a document and put them in an array, do the following:

```
//Get the set of collections the document belongs to and put in array.
DocumentCollections collections = metadataHandle.getCollections();
```

To check if a collection URI exists in a document's set of collections, do the following:

```
collections.contains("/collection_name/collection_name2");
```

To add a document to one or more collections, do the following:

```
collections.addAll("/shakespeare/sonnets", "/shakespeare/plays");
```

To remove a document from a collection, do the following:

```
collections.remove("/shakespeare/sonnets");
```

To remove a document from all its collections, do the following:

```
collections.clear();
```

2.6.4 Values Metadata

The `METADATAVALUES` metadata category represents simple key-value metadata for a document. Both the key and the value are strings. You can define your own key-value pairs. MarkLogic also adds key-value pairs to this type of metadata to documents in certain situations, such as when you work with temporal documents.

MarkLogic stores values metadata separately from its associated document. To search values metadata, define a metadata field and use a field query. For more details, see Metadata Fields in the *Administrator's Guide*.

To access metadata values you've read from the database, use `DocumentMetadataHandle.getMetadataValues`. For example, if you read the metadata from a document using a call sequence similar to the following:

```
DocumentMetadataHandle metadataHandle = new DocumentMetadataHandle();
docMgr.setMetadataCategories(METADATAVALUES);
docMgr.readMetadata(docId, metadataHandle);
```

Then you can access the returned values metadata as follows:

```
DocumentMetadataValue mvMap = metadataHandle.getMetadataValues();
String someValue = mvMap.get("someKey");
```

`DocumentMetadataValue` is an extension of `java.util.Map`, so you can use the `Map` methods to explore the returned metadata.

To add a new key-value pair or change the value of an existing pair, in a document's metadata, use `DocumentMetadataValue.put` or `DocumentMetadataHandle.withMetadataValue`. For example, the following adds a key-value pair with key "myKey" and value "myValue":

```
mvMap.put("myKey", "myValue");
//or
metadataHandle.withMetadataValue("myKey", "myValue");
```

Once you initialize your map or handle with values metadata, write the new metadata to the database as described in "Writing Metadata" on page 50.

2.6.5 Properties Metadata

Manipulate properties metadata using the `com.marklogic.client.io.DocumentMetadataHandle.DocumentProperties` class.

The code in this section assumes a `DocumentManager` object, `docMgr`, and a string containing a document's URI, `docId`, have been created.

To get all of a document's properties metadata, do the following:

```
DocumentProperties properties = metadataHandle.getProperties();
```

`DocumentProperties` objects represent a document's properties as a map.

To check if a document's properties contain a specific property name, do the following:

```
exists = properties.containsKey("name");
```

To get a specific property's value do the following:

```
value = metadataHandle.getProperties("name");
```

To add a new property or change the value of an existing property in a document's metadata, build up the new set of properties using `DocumentProperties.put` or `DocumentMetadataHandle.withProperty`, and then write the new metadata to the database as described in "Writing Metadata" on page 50. For example, the following adds a property named "name" with the value "value".

```
metadataHandle.getProperties().put("name", "value");
```

2.6.6 Quality Metadata

The code in this section assumes a `com.marklogic.client.io.DocumentManager` object, `docMgr`, and a string containing a document's URI, `docId`, have been created.

The quality metadata affects the ranking of documents for use in searches by creating a multiplier for calculating the score for that document, and the default value for quality in the Java API is `0`.

To get a document's search quality metadata value do the following:

```
int quality = metadataHandle.getQuality();
```

To set a document's search quality value do the following:

```
metadataHandle.setQuality(3);
```

2.6.7 Permissions Metadata

Permissions on documents control who can access a document for the capabilities of read, update, insert, and execute. To perform one of these operations on a document, a user must have a role corresponding to the permission for each capability needed. For details on permissions and on the security model in MarkLogic Server, see the *Security Guide*.

The code in this section assumes a `DocumentManager` object, `docMgr`, and a string containing a document's URI, `docId`, have been created. Manipulate document properties using the class `com.marklogic.client.io.DocumentMetadataHandle.DocumentPermissions`.

MarkLogic Server defines permissions using roles and capabilities.

The allowed values for capabilities are those in the enum `com.marklogic.client.io.DocumentMetadataHandle.Capability`:

- `EXECUTE` - Permission to execute the document.
- `INSERT` - Permission to create but not modify or delete the document.
- `READ` - Permission to read the document but not modify it..
- `UPDATE` - Permission to create, modify, or delete the document, but not to read it.

Roles are assigned to users via the Admin Interface or through other administrative tools, and cannot be assigned via the Java Client API. You can, however, control permissions on documents as part of their metadata.

To get permissions metadata for a document, do the following:

```
DocumentPermissions permissions = metadataHandle.getPermissions()

metadataHandle.getPermissions().add("app-user",
   Capability.UPDATE, Capability.READ);
```

2.6.8 Manipulating Document Metadata In Your Application

A `DocumentMetadataHandle` represents metadata as a POJO. A `DocumentMetadataHandle` has several methods for manipulating a document's metadata. That may not be how you want to work with the metadata, however. If you would prefer to work with it as XML, then read it with an XML handle. If you would prefer to work with it as JSON, read it with a JSON handle. A `StringHandle` can use either XML or JSON, defaulting to XML.

To specify the format for reading content, use `withFormat()` or `setFormat()`, as in the following example:

```
StringHandle metadataHandle =
    new StringHandle().withFormat(Format.JSON);
```

2.6.9 Writing Metadata

When you are finished modifying metadata categories, you must write it to the database to persist it. Note that the above operations all only change the document's metadata stored on the client, and do not change the metadata for document in the database. To write the metadata changes to the database, as well as the document content, do the following:

```
InputStreamHandle handle = new InputStreamHandle(docStream);
docMgr.write(docId, metadataHandle, handle);
```

2.7 Working with Temporal Documents

Most document write operations on JSON and XML documents enable you to work with temporal documents. Temporal-aware document inserts and updates are made available through the `com.marklogic.client.document.TemporalDocumentManager` interface. `JSONDocumentManager` and `XMLDocumentManager` implement `TemporalDocumentManager`.

The `TemporalDocumentManager` interface exposes methods for creating, updating, patching, and deleting documents that accept temporal related parameters such as the following:

- `temporalCollection`: The URI of the temporal collection into which the new document should be inserted, or the name of the temporal collection that contains the document being updated.
- `temporalDocumentURI`: The "logical" URI of the document in the temporal collection; the temporal collection document URI. This is equivalent to the first parameter of the `temporal:statement-set-document-version-uri` XQuery function or of the `temporal.statementSetDocumentVersionUri` Server-Side JavaScript function.
- `sourceDocumentURI`: The temporal collection document URI of the document being operated on. Only applicable when updating existing documents. This parameter facilitates working with documents with user-maintained version URIs.
- `systemTime`: The system start time for an update or insert.

During an update operation, if you do not specify `sourceDocumentURI` or `temporalDocumentURI` parameters, then the `uri` parameter indicates the source document. If you specify `temporalDocumentURI`, but do not specify `sourceDocumentURI`, then the `temporalDocumentURI` identifies the source document.

The `uri` parameter always refers to the output document URI. When the MarkLogic manages the version URIs, the document URI and temporal document collection URI have the same value. When the user manages version URIs, they can be different.

The `TemporalDocumentManager.protect` method enables you to protect a temporal document from operations such as update, delete, and wipe for a specified period of time. This method is equivalent to calling the `temporal:document-protect` XQuery function or the `temporal.documentProtect` Server-Side JavaScript function.

Use `TemporalDocumentManager.advanceLsqt` to advance LSQT on a temporal collection. This method is equivalent to calling the `temporal:advance-lsqt` XQuery function or the `temporal.advanceLsqt` Server-Side JavaScript function.

For more details, see the *Temporal Developer's Guide* and the JavaDoc in the *Java Client API Documentation*.

2.8 Conversion of Document Encoding

The Java API handles encoding conversions for you, but you have to:

- know the encoding
- use the appropriate handle

If you specify the encoding and it turns out to be the wrong encoding, then the conversion will likely not turn out as you expect.

MarkLogic Server stores text, XML, and JSON as UTF-8. In Java, characters in memory and reading streams are UTF-16. The Java API converts characters to and from UTF-8 automatically.

When writing documents to the server, you need to know if they are already UTF-8 encoded. If a document is not UTF-8, you must specify its encoding or you are likely to end up with data that has incorrect characters due to the incorrect encoding. If you specify a non-UTF-8 encoding, the Java API will automatically convert the encoding to UTF-8 when writing to MarkLogic.

When writing characters to or reading characters from a file, Java defaults to the platform's standard encoding. For example, there is different platform encoding on Linux than Windows.

XML supports multiple encodings as defined by the header (called an XML declaration):

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

The XML declaration declares a file's encoding. XML parsing tools, including handles, can determine encoding from this and do the conversion for you.

When writing character data to the database, you need to pick an appropriate handle type, depending on your intent and circumstances.

Depending on your application, you may need to be aware that MarkLogic Server normalizes text to precomposed Unicode characters for efficiency. Unicode abstract characters can either be precomposed (one character) or decomposed (two characters). If you write a decomposed Unicode document to MarkLogic Server and then read it back, you will get back precomposed Unicode. Usually, you do not need to care if characters are precomposed or decomposed. This

Unicode issue only affects some characters, and many APIs abstract away the difference. For instance, the Java collator treats the precomposed and decomposed forms of a character as the same character. If your application needs to compensate for this difference, you can use `java.text.Normalizer`; for details, see:

http://docs.oracle.com/javase/6/docs/api/java/text/Normalizer.html

The following table describes possible cases for reading character data with recommended handles to use in each case.

Read Condition	Recommended Handle(s)
If reading binary data:	Use `BytesHandle`, `FileHandle`, or `InputStreamHandle`.
If reading character data from the database:	`BytesHandle`, `FileHandle`, `InputStreamHandle`, and the XML handles are encoded as UTF-8. `StringHandle` and `ReaderHandle` convert to UTF-16.

The following table describes possible cases for writing character data with recommended handles to use in each case.

Write Condition	Recommended Handle(s)
If the data you are writing is a Java string:	Use `StringHandle`; it converts on write from UTF-16 to UTF-8.
If writing binary data:	Use `BytesHandle`, `FileHandle`, or `InputStreamHandle`.
If the data you are writing is encoded as UTF-8 and you do not need to modify the data:	Use `BytesHandle`, `FileHandle`, or `InputStreamHandle`.
If it is XML that declares an encoding other than UTF-8 in the XML declaration and you do not need to modify the data:	Use `InputSourceHandle`, `XMLEventReaderHandle`, or `XMLStreamReaderHandle`; these convert to UTF-8.

Write Condition	Recommended Handle(s)
If the character data to write is XML that declares the encoding in a prolog and you need to modify the data:	Use `DOMHandle`, `SourceHandle`, or create a handle class on an open source DOM. For examples of the latter, see `JDOMHandle`, `XOMHandle`, or `DOM4JHandle` in the package `com.marklogic.client.extra`. All these classes convert to UTF-8.
If the character data to write has a known encoding other than UTF-8 and you don't need to modify the data:	Use `ReaderHandle` and specify the encoding when creating the `Reader` (as usual in Java); these convert to UTF-8.

If the character data to write is XML with a known but undeclared encoding and you need to modify the data:	Use `DOMHandle` with a `DocumentBuilder` parsing an `InputSource` with a specified encoding as in:
	
	`DOMHandle handle = new DOMHandle();`
	`handle.set(`
	`handle.getFactory().newDocumentBuilder()`
	`parse(newInputSource(...reader specifying charset ...)));`
	
	or Use `SourceHandle` with a `StreamReader` on a `Reader` with a specified encoding as in:
	
	`SourceHandle handle = new SourceHandle();`
	`handle.set(new StreamSource(... reader specifying charset ...));`

If the character data to write is JSON and you need to modify the data:	Consider using a JSON library such as Jackson or GSON. See `com.marklogic.client.extra.JacksonHandle` for an example.
If the character data to write is text other than JSON or XML and you need to modify the data:	Consider using a `StreamTokenizer` with a `Reader`, or `Pattern` with a `String`

2.9 Partially Updating Document Content and Metadata

The interface `com.marklogic.client.document.DocumentPatchBuilder` enables you to update a portion of an existing document or its metadata. This section covers the following topics:

- Introduction to Content and Metadata Patching

- Basic Steps for Patching Documents and Metadata
- Construct a Patch From Raw XML or JSON
- Defining the Context for a Patch Operation
- Example: Replacing Parts of a JSON Document
- Example: Patching Metadata
- Managing XML Namespaces in a Patch
- Construct Replacement Data on the Server

2.9.1 Introduction to Content and Metadata Patching

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

Use a partial update to do the following operations:

- Add, replace, or delete an XML element, XML attribute, or JSON object or array item of an existing document.
- Add, replace, or delete a subset of the metadata of an existing document. For example, modify a permission or insert a property.
- Dynamically generate replacement content or metadata on MarkLogic Server using builtin or user-defined functions. For details, see “Construct Replacement Data on the Server” on page 67.

You can apply multiple updates in a single patch, and you can update both content and metadata in the same patch.

A *patch* is a partial update descriptor, expressed in XML or JSON, that tells MarkLogic Server where to apply an update and what update to apply. Four operations are available in a patch: insert, replace, replace-insert, and delete. (A replace-insert operation functions as a replace, as long as at least one match exists for the target content; if there are no matches, then the operation functions as an insert.)

Patch operations can target XML elements and attributes, JSON property values and array items, and data values. You identify the target of an operation using XPath and JSONPath expressions. When inserting new content or metadata, the insertion point is further defined by specifying the position; for details, see How Position Affects the Insertion Point in the *REST Application Developer’s Guide*.

Note: You can only use a subset of XPath to define path expressions in patch operations. For details, see Patch Feature of the Client APIs in the *XQuery and XSLT Reference Guide*.

When applying a patch to document content, the patch format must match the document format: An XML patch for an XML document, a JSON patch for a JSON document. You cannot patch the content of other document types. You can patch metadata for all document types. A metadata-only patch can be in either XML or JSON. A patch that modifies both content and metadata must match the document content type.

You can construct a patch from raw JSON or XML, or using one of the following builder interfaces:

- `com.marklogic.client.document.DocumentPatchBuilder`
- `com.marklogic.client.document.DocumentMetadataPatchBuilder`

The patch builder interface contains value and fragment oriented methods, such as `replaceValue` and `replaceFragment`. You can use the `*Value` methods when the new value is an atomic value, such as a string, number, or boolean. Use the `*Fragment` methods when the new value is a complex structure, such as an XML element or JSON object or array.

Apply a patch by passing a handle to it to the `patch()` method of a `DocumentManager`. The following example sketches construction of a patch using a builder, and then applying the patch to an XML document. The patch inserts a `<child/>` element as the last child element of the node addressed by the XPath expression `/data`.

```
DocumentPatchBuilder xmlPatchBldr = XMLDocMgr.newPatchBuilder();
DocumentPatchHandle xmlPatch =
    xmlPatchBldr.insertFragment(
        "/data",
        Position.LAST_CHILD,
        "<child>the last one</child>")
      .build();
XMLDocMgr.patch(docId, xmlPatch);
```

The following example sketches construction of a patch using a builder, and then applying the patch to a JSON document. The patch inserts a `before` element as the element before the node adressed by the path expression `/data`.

```
DocumentPatchBuilder jsonPatchBldr = JSONDocMgr.newPatchBuilder();
DocumentPatchHandle jsonPatch =
    jsonPatchBldr.insertFragment(
        "/data",
        Position.BEFORE,
        "{\"before\":\"element before data attribute\"}")
      .build();
JSONDocMgr.patch(docId, jsonPatch);
```

For detailed instructions, see “Basic Steps for Patching Documents and Metadata” on page 56.

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

Content transformations are not directly supported in a partial update. However, you can implement a custom replacement content generation function to achieve the same effect. For details, see "Construct Replacement Data on the Server" on page 67.

2.9.2 Basic Steps for Patching Documents and Metadata

This section describes how to create a patch builder, use it to construct a patch descriptor, and then apply the patch. To construct a patch without using a builder, see "Construct a Patch From Raw XML or JSON" on page 58.

For JSON and XML documents, you can use a `com.marklogic.client.document.DocumentPatchBuilder` to patch content only, content plus metadata, or metadata only. For all document types, you can use a `com.marklogic.client.document.DocumentMetadataPatchBuilder` to patch metadata only. A `DocumentPatchBuilder` is also a `DocumentMetadataPatchBuilder`. Use a `DocumentManager` subclass such as `JSONDocumentManager` or `GenericDocumentManager` to create a patch builder.

When you combine content and metadata updates in the same patch, the patch format (XML or JSON) must match the content type of the patched documents.

Follow this procedure to use a builder to create and apply a patch to the contents of an XML or JSON document, or to the metadata of any type of document.

1. If you have not already done so, connect to the database, storing the connection in a `com.marklogic.client.DatabaseClient` object. For example, if using digest authentication:

   ```
   DatabaseClient client = DatabaseClientFactory.newClient(
       host, port, new DigestAuthContext(username, password));
   ```

2. If you have not already done so, use the `DatabaseClient` object to create a `com.marklogic.client.document.DocumentManager` object of the appropriate subclass for the document content you want to access (XML, JSON, binary, or text).

   a. In this example code, an `XMLDocumentManager`.

      ```
      XMLDocumentManager XMLDocMgr = client.newXMLDocumentManager();
      ```

   b. In this example code, an `JSONDocumentManager`.

      ```
      JSONDocumentManager JSONDocMgr = client.newJSONDocumentManager();
      ```

Note: You can only apply content patches to XML and JSON documents.

3. Create a document patch builder or metadata patch builder using the document manager.

   a. For example:

   ```
   DocumentPatchBuilder builderXML = XMLDocMgr.newPatchBuilder();
   ```

   b. Or:

   ```
   DocumentPatchBuilder builderJSON = JSONDocMgr.newPatchBuilder();
   ```

4. Call the patch builder methods to define insert, replace, replace-insert, and delete operations for the patch.

   a. The following example adds an element insertion operation:

   ```
   builderXML.insertFragment("/data", Position.LAST_CHILD,
       "<child>the last one</child>");
   ```

   b. The following example adds an element insertion operation:

   ```
   builderJSON.insertFragment("/data", Position.BEFORE,
       "{\"before\":\"element before data attribute\"}");
   ```

   For more details on identifying the target content for an operation, see “Defining the Context for a Patch Operation” on page 60.

5. Create a handle associated with the patch using `DocumentPatchBuilder.build()`.

   a. For example:

   ```
   DocumentPatchHandle handleXML = builderXML.build();
   ```

   b. Or:

   ```
   DocumentPatchHandle handleJSON = builderJSON.build();
   ```

   Note: Once you call `build()`, the patch contents are fixed. Subsequent calls to define additional operation, such as calling `insertFragment` again, will have no effect.

6. Apply the patch by calling a `patch()` method on the `DocumentManager`, with arguments of the document’s URI and the handle.

   a. For example:

   ```
   XMLDocMgr.patch(docId, handleXML);
   ```

b. Or:

```
JSONDocMgr.patch(docId, handleJSON);
```

7. When finished with the database, release the connection resources by calling the `DatabaseClient` object's `release()` method. For example:

```
client.release();
```

2.9.3 Construct a Patch From Raw XML or JSON

This section describes how to create and apply a patch that you construct directly using XML or JSON. To construct a patch using a Java builder, see "Basic Steps for Patching Documents and Metadata" on page 56.

When you construct a patch that modifies both content and metadata, the patch format must match the content type of the target XML or JSON document. When you construct a patch that only modifies metadata, the patch format can use either XML or JSON, and the patch can be applied to the metadata of any type of document (XML, JSON, text, or binary).

For examples of raw patches, see XML Examples of Partial Updates or JSON Examples of Partial Update in the *REST Application Developer's Guide*:

Follow this procedure to create and apply a raw XML or JSON patch to the contents of an XML or JSON document, or to the metadata of any type of document.

1. Create a JSON or XML representation of the patch operations, using the tools or library of your choice. For syntax, see XML Patch Reference and JSON Patch Reference and in the *REST Application Developer's Guide*.

   a. The following example uses a `String` representation of a patch that inserts an element in an XML document:

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

   b. The following example uses a `String` representation of a patch that inserts an element in a JSON document:

```
String jsonPatch = "{ \"patch\": " +
       "[ { \"insert\": { " +
                          "\"context\": \"/parent/child1\", " +
                          "\"position\": \"before\", " +
                          "\"content\": { \"INSERT1\": \"INSERTED1\" }
                  }}] }";
```

2. If you have not already done so, connect to the database, storing the connection in a `com.marklogic.client.DatabaseClient` object. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

3. If you have not already done so, use the `DatabaseClient` object to create a `com.marklogic.client.document.DocumentManager` object of the appropriate subclass for the document content you want to access (XML, JSON, binary, or text).

   a. In this example code, an `XMLDocumentManager`.

```
XMLDocumentManager XMLDocMgr = client.newXMLDocumentManager();
```

   b. In this example code, an `JSONDocumentManager`.

```
JSONDocumentManager JSONDocMgr = client.newJSONDocumentManager();
```

   Note: You can only apply content patches to XML and JSON documents.

4. Create a handle that implements `DocumentPatchHandle` and associate your patch with the handle. Set the handle content type appropriately. For example:

```
// For an XML patch
DocumentPatchHandle handle =
    new StringHandle(xmlPatch).withFormat(Format.XML);

// For a JSON patch
DocumentPatchHandle handle =
    new StringHandle(jsonPatch).withFormat(Format.JSON);
```

5. Apply the patch by calling a `patch()` method on the `DocumentManager`, with arguments of the document's URI and the handle.

```
XMLDocMgr.patch(docId, handle);
// Or
JSONDocMgr.patch(docId, handle);
```

6. When finished with the database, release the connection resources by calling the `DatabaseClient` object's `release()` method.

```
client.release();
```

2.9.4 Defining the Context for a Patch Operation

When you insert, replace, or delete content or metadata, the patch definition must include enough context to tell MarkLogic Server what XML or JSON components to operate on. For example, which XML element or JSON property to modify, where to insert a new element or object, or which element, object, or value to replace.

When you create a patch using a builder, you specify the context through the contextPath and selectPath parameters of builder methods such as `DocumentPatchBuilder.insertFragment()` or `DocumentPatchBuilder.replaceValue()`. When you create a patch from raw XML or JSON, you specify the operation context through the `context` and `select` XML attributes or JSON properties.

For XML documents, you specify the context using an XPath (XML) expression.The XPath you can use is limited to a subset of XPath. For details, see Patch Feature of the Client APIs in the *XQuery and XSLT Reference Guide*.

For JSON documents, use JSONPath (JSON). The JSONPath you can use has the same limitation as those that apply to XPath. For details, see Introduction to JSONPath and Patch Feature of the Client APIs in the *XQuery and XSLT Reference Guide*.

2.9.5 Example: Replacing Parts of a JSON Document

This example uses patch operations to perform the document transformation shown in the table below. The patch replaces one JSON property with another, replaces the simple value of a property, and replaces the array value of a property.

Before Update	After Update

`{ "parent": {`
`  "child1": {`
`  "grandchild": "value"`
`  },`
`  "child2": "simple",`
`  "child3": [ "av1", "av2" ]`
`} }`
	`{ "parent": {`
	`  "child1": {`
	`  **"REPLACE1": "REPLACED1"**`
	`  },`
	`  "child2": **"REPLACED2"**,`
	`  "child3": **[**`
	`  **"REPLACED3a",**`
	`  **"REPLACED3b"**`
	`  **]**`
	`} }`

The raw patch that applies these changes is shown below.

```
{ "patch": [
    { "replace": {
        "select": "/parent/child1",
        "content": { "REPLACE1": "REPLACED1" }
    }},
    { "replace": {
```

```
            "select": "/parent/child2",
            "content": "REPLACED2"
        }},
        { "replace": {
            "select": "/parent/array-node('child3')",
            "content": [ "REPLACED3a", "REPLACED3b" ]
        }}
]}
```

The following code demonstrates how to use the PatchBuilder interface to create the equivalent raw patch. A Jackson `ObjectMapper` is used to construct the complex replacement values (the object value of `child1` and the array value of `child3`).

```
JSONDocumentManager jdm = client.newJSONDocumentManager();
DocumentPatchBuilder pb = jdm.newPatchBuilder();
pb.pathLanguage(DocumentPatchBuilder.PathLanguage.XPATH);
ObjectMapper mapper = new ObjectMapper();

pb.replaceFragment("/parent/child1",
        mapper.createObjectNode().put("REPLACE1", "REPLACED1"));
pb.replaceValue("child2", "REPLACED2");
pb.replaceFragment("/parent/array-node('child3')",
        mapper.createArrayNode().add("REPLACED3a").add("REPLACED3b"));
jdm.patch(URI, pb.build());
```

For additional (raw) patch examples, see XML Examples of Partial Updates and JSON Examples of Partial Update in the *REST Application Developer's Guide*. These examples can assist you with constructing appropriate XPath expressions and replacement context in Java.

2.9.6 Example: Patching Metadata

This example demonstrates using a patch builder to modify metadata such as collections, permissions, quality, document properties, and key-value metadata.

Assume a document exists in the database with the following metadata. The document is in one collection, has no document properties or key-value metadata, has default permissions, and has quality 2.

```
<rapi:metadata uri="/java/doc.json"
    xsi:schemaLocation="http://marklogic.com/rest-api restapi.xsd"
    xmlns:rapi="http://marklogic.com/rest-api"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <rapi:collections>
    <rapi:collection>original</rapi:collection>
  </rapi:collections>
  <rapi:permissions>
    <rapi:permission>
      <rapi:role-name>rest-writer</rapi:role-name>
      <rapi:capability>update</rapi:capability>
    </rapi:permission>
    <rapi:permission>
```

```
        <rapi:role-name>rest-reader</rapi:role-name>
        <rapi:capability>read</rapi:capability>
      </rapi:permission>
    </rapi:permissions>
    <prop:properties xmlns:prop="http://marklogic.com/xdmp/property"/>
    <rapi:quality>2</rapi:quality>
    <rapi:metadata-values/>
  </rapi:metadata>
```

The example modifies the metadata to do the following:

- Add the document to another collection.
- Set the quality to 3.
- Add some key-value metadata.
- Add a new role to the permissions

The following code builds and applies the patch using a `GenericDocumentManager` and `DocumentMetadataPatchBuilder`.

```
public static void metadataExample() {
    GenericDocumentManager gdm = client.newDocumentManager();
    DocumentMetadataPatchBuilder pb = gdm.newPatchBuilder(Format.XML);

    pb.addCollection("new");
    pb.setQuality(3);
    pb.addMetadataValue("newkey", "newvalue");
    pb.addPermission("newrole",
                     DocumentMetadataHandle.Capability.READ,
                     DocumentMetadataHandle.Capability.UPDATE);

    gdm.patch(URI, pb.build());
}
```

After applying the patch, the document has the following metadata. The portion modified by the patch are shown in bold.

```
<rapi:metadata uri="/java/doc.json"
    xsi:schemaLocation="http://marklogic.com/rest-api restapi.xsd"
    xmlns:rapi="http://marklogic.com/rest-api"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <rapi:collections>
    <rapi:collection>original</rapi:collection>
    <rapi:collection>new</rapi:collection>
  </rapi:collections>
  <rapi:permissions>
    <rapi:permission>
      <rapi:role-name>rest-writer</rapi:role-name>
      <rapi:capability>update</rapi:capability>
    </rapi:permission>
    <rapi:permission>
```

```
      <rapi:role-name>newrole</rapi:role-name>
      <rapi:capability>update</rapi:capability>
      <rapi:capability>read</rapi:capability>
    </rapi:permission>
    <rapi:permission>
      <rapi:role-name>rest-reader</rapi:role-name>
      <rapi:capability>read</rapi:capability>
    </rapi:permission>
  </rapi:permissions>
  <prop:properties xmlns:prop="http://marklogic.com/xdmp/property"/>
  <rapi:quality>3</rapi:quality>
  <rapi:metadata-values>
    <rapi:metadata-value key="newkey">newvalue</rapi:metadata-value>
  </rapi:metadata-values>
</rapi:metadata>
```

Assume a document exists in the database with the following metadata. The document is in one collection, has default permissions, and has quality 0.

```
{
  "collections": [
    "squares"
  ],
  "permissions": [
    {
      "role-name": "rest-writer",
      "capabilities": [
        "update"
      ]
    }
  ],
  "properties": {
    "myprop": "this is my prop",
    "myotherprop": "this is my other prop"
  },
  "quality": 0
```

```
}
```

The example modifies the metadata to do the following:

- Add the document to another collection.
- Set the quality to 3.
- Add some key-value metadata.
- Add a new role to the permissions

The following code builds and applies the patch using a `GenericDocumentManager` and `DocumentMetadataPatchBuilder`.

```
public static void metadataExample() {
    GenericDocumentManager gdm = client.newDocumentManager();
    DocumentMetadataPatchBuilder pb = gdm.newPatchBuilder(Format.JSON);

    pb.addCollection("new");
    pb.setQuality(3);
    pb.addMetadataValue("newkey", "newvalue");
    pb.addPermission("newrole",
                     DocumentMetadataHandle.Capability.READ,
                     DocumentMetadataHandle.Capability.UPDATE);

    gdm.patch(URI, pb.build());
}
```

After applying the patch, the document has the following metadata. The portion modified by the patch are shown in bold.

```
{
  "collections": [
    "shapes",
    "new"
  ],
  "permissions": [
    {
      "role-name": "rest-writer",
      "capabilities": [
        "update"
      ]
```

```
    }, {
      "role-name": "new-role",
      "capabilities": [
        "update",
        "read"
      ]
    }

  ],
  "properties": {
    "myprop": "this is my prop",
    "myotherprop": "this is my other prop"
  },
  "quality": 3,
  "metadataValues": {
    "newkey": "newvalue"
  }
}
```

You could also use a document type specific document manager to apply the patch. For example, you could use a `JSONDocumentManager` to create a `DocumentPatchBuilder` as shown below. The patch builder operations (`pb.addCollection`, etc.) do not change as a consequence.

```
JSONDocumentManager jdm = client.newJSONDocumentManager();
DocumentPatchBuilder pb = jdm.newPatchBuilder();
pb.pathLanguage(DocumentPatchBuilder.PathLanguage.XPATH);

// Construct and apply patch as previously shown
```

2.9.7 Managing XML Namespaces in a Patch

Namespaces potentially impact two parts of a patch operation:

- The XPath expression(s) that define the context for an operation, such as which nodes to replace or where to insert new content.

- New or replacement content.

Your patch must include definitions of any namespaces used in these contexts. The way you do so varies, depending on whether or not you use a builder to construct your patch. This section covers the following topics:

- Defining Namespaces With a Builder
- Defining Namespaces in Raw XML

2.9.7.1 Defining Namespaces With a Builder

When you construct a patch with `DocumentPatchBuilder`, define any namespaces used in XPath context or select expressions by calling `DocumentPatchBuilder.setNamespaces()`. Such namespace definitions are patch-wide. That is, they apply to all operations in the patch.

Namespaces used in insertion or replacement content can either be patch-wide, as with XPath expressions, or defined inline on content elements.

The patch generated by the builder pre-defines the following namespace aliases for you:

- `xmlns:rapi="http://marklogic.com/rest-api"`
- `xmlns:prop="http://marklogic.com/xdmp/property"`
- `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`
- `xmlns:xi="http://www.w3.org/2001/XMLSchema"`

The following example defines three namespace aliases (`r`, `t`, and `n`) and uses them in defining the insertion context and the content to be inserted.

```
import com.marklogic.client.util.EditableNamespaceContext;
...
// construct a list of namespace definitions
EditableNamespaceContext namespaces = new EditableNamespaceContext();
namespaces.put("r", "http://root.org");
namespaces.put("t", "http://target.org");
namespaces.put("n", "http://new.org");

// add the namespace definitions to the patch
DocumentPatchBuilder builder = docMgr.newPatchBuilder();
builder.setNamespaces(namespaces);

// use the namespace aliases when definition operations
String newElem = "<n:new>";
builder.insertFragment(
    "/r:root/t:target", Position.LAST_CHILD, newElem);
```

You can also define the content namespace element `n` inline, as shown in the following example:

```
String newElem = "<n:new xmlns:n=\"http://new.org\">";
```

2.9.7.2 Defining Namespaces in Raw XML

When you construct a patch directly in XML, define any namespaces used in XPath context or select expressions on the root `<patch/>` element. Namespace definitions are patch-wide and apply to both XPath expressions and insertion or replacement content.

The `<patch />` element must be defined in the namespace `http://marklogic.com/rest-api`. It is recommended that you use a namespace alias for this namespace so that element and attribute references in your patch that are not namespace qualified do not end up in the `http://marklogic.com/rest-api namespace`.

The following example defines four namespace aliases, one for the patch (`rapi`) and three content-specific aliases (`r`, `n`, and `t`). The content-specific aliases are used in defining the insertion context and the content to be inserted.

```
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api"
    xmlns:r="http://root.org" xmlns:t="http://target.org"
    xmlns n="http://new.org">
  <rapi:insert context="/r:root/t:target" position="last-child">
    <n:new />
  </rapi:insert>
</rapi:patch>
```

For more details, see Managing XML Namespaces in a Patch in the *REST Application Developer's Guide*.

2.9.8 Construct Replacement Data on the Server

This section describes using builtin or user-defined XQuery or Server-Side JavaScript replacement functions to generate the content for a partial update replace or replace-insert operation dynamically on MarkLogic Server.

The builtin functions support simple arithmetic and string manipulation. For example, you can use a builtin function to increment the current value of numeric data or concatenate strings. For more complex operations, create and install a user-defined function.

To create a user-defined replacement function, see Writing an XQuery User-Defined Replacement Constructor or Writing a JavaScript User-Defined Replacement Constructor in the *REST Application Developer's Guide*. Install your implementation into the modules database associated with your REST Server; for details, see "Managing Dependent Libraries and Other Assets" on page 295.

To apply a builtin or user-defined server-side function to a patch operation when you create a patch with a patch builder, use a `DocumentMetadataPatchBuilder.CallBuilder`, obtained by calling `DocumentMetadataPatchBuilder.call()`. The builtin functions are exposed as methods of `CallBuilder`. The following example adds a replace operation to a patch that multiplies the current data value in `child` elements by 3.

```
DocumentPatchBuilder builder = docMgr.newPatchBuilder();
builder.replaceApply("child", builder.call().multiply(3));
```

To apply the same operation to a raw XML or JSON patch, use the `apply` XML attribute or JSON property of the operation. The following raw patches are equivalent to the patch produced by the above builder example. For details, see Constructing Replacement Data on the Server in the *REST Application Developer's Guide*.

XML	JSON
<rapi:patch xmlns:rapi="http://marklogic.com/rest-api"> <rapi:replace select="child" **apply="ml.multiply"**>3</rapi:replace> </rapi:patch>	{"patch": [{"replace": { "select": "child", **"apply": "ml.multiply"**, "content": 3 } }] }

To apply a user-defined replacement function using a patch builder, first associate the module containing the function with the patch by calling `DocumentPatchBuilder.library()`, and then apply the function to an operation using one of the `CallBuilder.applyLibrary*` methods. The following example applies the function `my-func` in the module namespace `http://my/ns`, implemented in the XQuery library module installed in the modules database at `/my.domain/my-lib.xqy`.

```
DocumentPatchBuilder builder = docMgr.newPatchBuilder();

builder.library("http://my/ns", "/my.domain/my-lib.xqy");
builder.replaceApply("child", builder.call().applyLibrary("my-func");
```

When you construct a raw XML or JSON patch, associate the containing library module with the patch using the `replace-library` patch component, then apply the function to a replace or `replace-insert` operation using the apply XML attribute or JSON property. The following examples are equivalent to the above builder code. For more details, see Using a Replacement Constructor Function in the *REST Application Developer's Guide*.

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

3.0 Synchronous Multi-Document Operations

This chapter describes how to read and write multiple documents in a single request to MarkLogic Server using the Java Client API. You can operate on both document content and metadata. The interfaces described here are synchronous, meaning your application will block during the operation.

If you only need to work with one document at a time, you can use the simpler single document interfaces. For details, see “Single Document Operations” on page 36. If you have a potentially long running multi-document task, consider using the asynchronous interfaces described in “Asynchronous Multi-Document Operations” on page 92.

This chapter includes the following sections:

3.1 Write Multiple Documents

This section describes how to create or update content and/or metadata for multiple documents in a single request to MarkLogic Server. This section includes the following topics:

3.1.1 Overview of Multi-Document Write

You can perform a multi-document write by building up a `DocumentWriteSet` that describes the document content and metadata to write, and then passing it to a `DocumentManager` to execute the write operation.

For example, the following code snippet writes content for an XML document with URI doc1.xml and both content and metadata for a JSON document with URI doc2.json. For a complete example, see “Example: Loading Multiple Documents” on page 72.

```
import com.marklogic.client.document.DocumentManager;
import com.marklogic.client.document.DocumentWriteSet;
...

DocumentWriteSet batch = docMgr.newWriteSet();

batch.add("doc1.xml", doc1ContentHandle);
batch.add("doc2.json", doc2MetadataHandle, doc2ContentHandle);

docMgr.write(batch);
```

A `DocumentWriteSet` represents a batch of document content and/or metadata to be written to the database in a single transaction. If any insertion or update in a write set fails, the entire batch fails. You should size each batch according to the guidelines described in “Selecting a Batch Size” on page 91.

A `DocumentWriteSet` has the following key features:

- Document content can be either heterogeneous or homogeneous, depending on the type of `DocumentManager` you use. For example, you can create or update any combination of XML, JSON, Text, and Binary documents in a single operation if you use `GenericDocumentManager`.
- For each document, a batch can include just content, just metadata, or both. If you include only metadata for a document, then the document must already exist.
- You can create or update documents with the system default metadata, batch default metadata, or document-specific metadata. You can mix these metadata sources in the same operation. For details, see “Understanding Metadata Scoping” on page 73.

The write operation is carried out by a `DocumentManager`. If all documents in the write set are of the same type, then using a `DocumentManager` of the corresponding type has the following advantages:

- The database document type is implicitly set by the `DocumentManager`. For example, an `XMLDocumentManager` sets the document type to XML for you and a `JSONDocumentManager` sets the document type to JSON for you.
- You can use the `DocumentManager` to set batch-wide, type specific options. For example, you can use `BinaryDocumentManager.setMetadataExtraction()` to direct MarkLogic Server to extract metadata from each binary document and store it in the document properties.

If you create a heterogeneous write set that includes documents of more than one type, then you must use a `GenericDocumentManager` to perform the write. In this case, you must explicitly set the type of each document and you cannot use any type specific options, such as XML repair or Binary metadata extraction. For details, see "Example: Writing a Mixed Document Set" on page 81.

When you use bulk write, pre-existing document properties are preserved, but other categories of metadata are completely replaced. If you want to preserve pre-existing metadata, use a single document write. For details, see "Understanding When Metadata is Preserved or Replaced" on page 76.

You can apply a server-side write transformation to each document in a multi-document write. First, install your transform on MarkLogic Server, as described in "Installing Transforms" on page 282. Then, include a reference to the transform in your `write` call, similar to the following:

```
ServerTransform transform = new ServerTransform(TRANSFORM_NAME);
docMgr.write(batch, transform);
```

3.1.2 Example: Loading Multiple Documents

This example provides a quick introduction to multi-document write. It creates two JSON documents in one transaction. The first document uses the system default metadata and the second document uses document-specific metadata.

Three items are added to the `DocumentWriteSet` for this operation: JSON content for a document with URI `doc1.json`, metadata for a document with URI `doc2.json`, and content for a JSON document with URI `doc2.json`. The core of the example is the following lines that build up a DocumentWriteSet and send it to MarkLogicServer for committing to the database:

```
// Create and populate the batch of docs to write
JSONDocumentManager jdm = client.newJSONDocumentManager();
DocumentWriteSet batch = jdm.newWriteSet();
batch.add("doc1.json", doc1);
batch.add("doc2.json", doc2Metadata, doc2);

// Perform the write operation
jdm.write(batch);
```

The full example function is shown below. This example uses `StringHandle` for the content, but you can use other handle types, such as `JacksonHandle` or `FileHandle`.

```
package examples;
import com.marklogic.client.io.*;
import com.marklogic.client.document.JSONDocumentManager;
import com.marklogic.client.document.DocumentWriteSet;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.DatabaseClientFactory.DigestAuthContext;
import com.marklogic.client.DatabaseClient;
```

```
public class Example implements ConnInfo {
    // replace with your MarkLogic Server connection information
    static String HOST = "localhost";
    static int PORT = 8000;
    static String USER = "username";
    static String PASSWORD = "password";
    static DatabaseClient client = DatabaseClientFactory.newClient(
        HOST, PORT, new DigestAuthContext(USER, PASSWORD));

    /// Basic example of writing 2 JSON documents.
    public static void example1() {
        // Create some example content and metadata
        StringHandle doc1 = new StringHandle(
                "{\"animal\": \"dog\"}").withFormat(Format.JSON);
        StringHandle doc2 = new StringHandle(
                "{\"animal\": \"cat\"}").withFormat(Format.JSON);
        DocumentMetadataHandle doc2Metadata =
                new DocumentMetadataHandle();
        doc2Metadata.setQuality(2);

        // Create and populate the batch of docs to write
        JSONDocumentManager jdm = client.newJSONDocumentManager();
        DocumentWriteSet batch = jdm.newWriteSet();
        batch.add("doc1.json", doc1);
        batch.add("doc2.json", doc2Metadata, doc2);

        // Perform the write operation
        jdm.write(batch);
    }

    public static void main(String[] args) {
        example1();
    }
}
```

3.1.3 Understanding Metadata Scoping

This topic describes how metadata is selected for documents created or updated with a multi-document write.

Note: For performance reasons, pre-existing metadata other than properties is completely replaced during a bulk write operation, either with values supplied in the `DocumentWriteSet` or with system defaults.

Metadata in a bulk write can be drawn from 3 possible sources, as shown in the table below. The table lists the metadata sources from highest to lowest precedence, so a source supercedes those below it if both are present.

Metadata Type	Description
document-specific metadata	Metadata that applies to a single document. Specify document-specific metadata by including a `DocumentMetadataHandle` along with the content handle when you call `DocumentWriteSet.add()`.
default metadata	Batch-specific metadata that can apply to multiple documents in a `DocumentWriteSet`. Specify default metadata by calling `DocumentWriteSet.addDefaultMetadata()`.
system default metadata	Default metadata configured into MarkLogic server. This metadata applies when neither document-specific nor set default metadata is present.

The metadata associated with a document is determined when you add the document to a `DocumentWriteSet`. This means that when you add default metadata, it only applies to documents subsequently added to the batch, not to documents already in the batch. Default metadata applies from the point it is added to the batch until a subsequent call to `DocumentWriteSet.addDefaultMetadata()`. Passing `null` to `addDefaultMetadata()` causes subsequent documents to revert to using system default metadata rather than batch default metadata.

The following code snippet illustrates the metadata interactions:

```
DatabaseClient client = ...;
JSONDocumentManager jdm = client.newJSONDocumentManager();
DocumentWriteSet batch = jdm.newWriteSet();

// using system default metadata
batch.add("doc1.json", doc1);    // use system default metadata

// using batch default metadata
batch.addDefaultMetadata(defaultMetadata1);
batch.add("doc2.json", doc2);    // use batch default metadata
batch.add("doc3.json", docSpecificMetadata, doc3);
batch.add("doc4.json", doc4);    // use batch default metadata

// replace batch default metadata with new metadata
batch.addDefaultMetadata(defaultMetadata2);
batch.add("doc5.json", doc5);    // use batch default metadata
```

```
// revert to system default metadata
batch.addDefaultMetadata(null);
batch.add("doc6.json", doc6);    // use system default metadata

// Execute the write operation
jdm.write(batch);
```

For a complete example, see “Example: Controlling Metadata Through Defaults” on page 77.

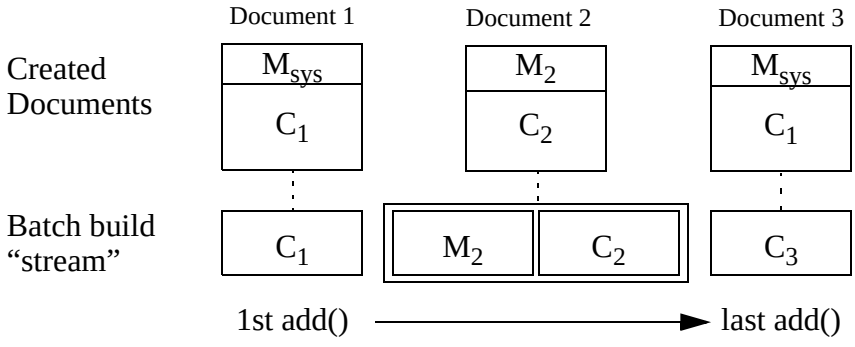
The following rules determine what metadata applies during document creation.

- Document-specific metadata always takes precedence over other metadata sources. Document-specific metadata is not merged with default metadata.
- System default metadata is used when there is no batch default metadata and no documents-specific metadata for a given document.
- Each time you add default metadata to a batch, the new default completely replaces any old default.
- When setting metadata for a document, any missing metadata category is either set to the system default metadata value or left unchanged, depending upon whether or not the batch includes a content update for the document. For details, see “Understanding When Metadata is Preserved or Replaced” on page 76.

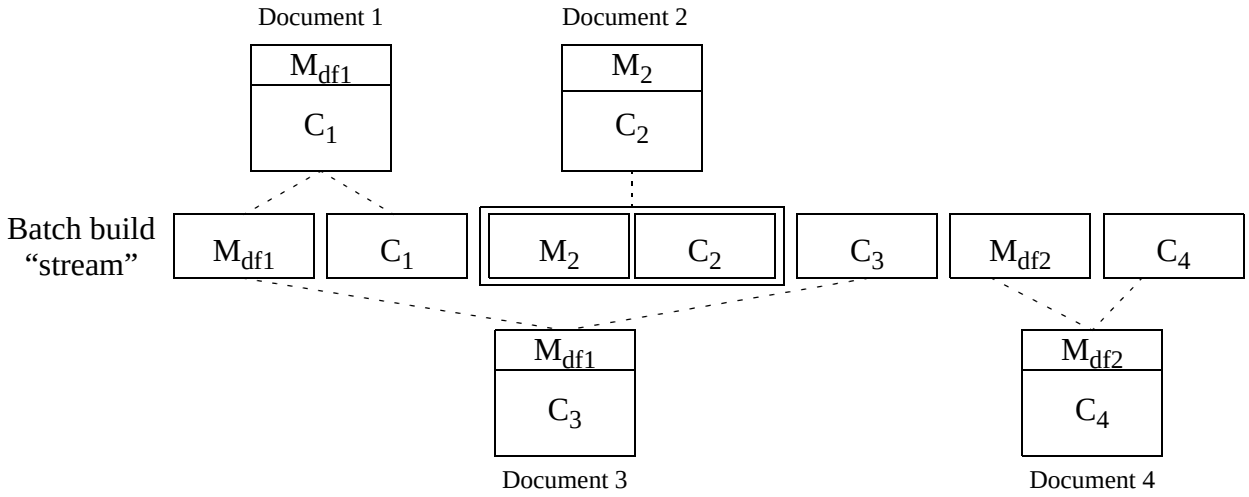
For performance reasons, no merging of document-specific or batch default metadata occurs. For example, if a document-specific metadata part contains only a collections setting, it inherits quality, permissions and properties from the system default metadata, not from any preceding batch default metadata.

The following examples illustrate application of these rules. In these examples, C_n represents a content part for the Nth document, M_n represents document-specific metadata for the Nth document, M_{dfn} represents the Nth occurrence of batch default metadata, and M_{sys} is the system default metadata. The batch build stream represents the order in which content and metadata is added to the batch.

The following input creates 3 documents. Documents 1 and Document 3 use system default metadata. Document 2 uses document-specific metadata.



The following input creates four documents, using a combination of batch default metadata and document-specific metadata. Document 1, Document 3, and Document 4 use batch default metadata. Document 2 uses document-specific metadata. Document 1 and Document 3 use the first block of batch default metadata, M_{df1}. After Document 3 is added to the batch, M_{df2} replaces M_{df1} as the default metadata, so Document 4 uses the metadata in M_{df2}.



3.1.4 Understanding When Metadata is Preserved or Replaced

This topic discusses when a multi-document write preserves or replaces pre-existing metadata. You can skip this section if your multi-document write operations only create new documents or you do not need to preserve pre-existing metadata such as permissions, document quality, collections, and properties.

When there is no batch default metadata and no document-specific metadata, all metadata categories other than properties are set to the system default values. Properties are unchanged.

In all other cases, either batch default metadata or document-specific metadata is used when creating a document, as described in "Understanding Metadata Scoping" on page 73.

When you update both content and metadata for a document in the same multi-document write operation, the following rules apply, whether applying batch default metadata or document-specific metadata:

- The metadata in scope is determined as described in "Understanding Metadata Scoping" on page 73.
- Any metadata category that has a value in the in-scope metadata completely replaces that category.
- Any metadata category other than properties that is missing or empty in the in-scope metadata is completely replaced by the system default value.

- If the in-scope metadata does not include properties, then existing properties are preserved.
- If the in-scope metadata does not include collections, then collections are reset to the default. There is no system default for collections, so this results in a document being removed from all collections if no default collections are specified for the user role performing the update.

When your write set includes metadata for a document, but no content, you update only the metadata for a document. In this case, the following rules apply:

- Any metadata category that has a value in the document-specific metadata completely replaces that category.
- Any metadata category that is missing or empty in the document-specific metadata is preserved.

The table below shows how pre-existing metadata changes if a multi-document write updates just the content, just the collections metadata (via document-specific metadata), or both content and collections metadata (via batch default metadata or document-specific metadata).

Metadata Category	Update Content Only	Update Metadata Only	Update Content & Metadata
collections	reset	modified to new value	modified to new value
quality	reset	preserved	reset
permissions	reset	preserved	reset
properties	preserved	preserved	preserved

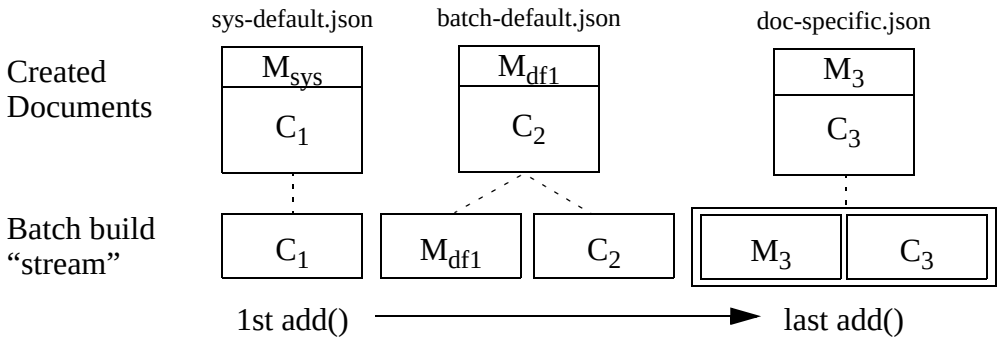
The results are similar if the metadata update modifies other metadata categories.

3.1.5 Example: Controlling Metadata Through Defaults

This example uses document quality to illustrate how default metadata affects the documents you create. The document quality setting used in this example result in creation of the following documents:

- `sys-default.json` with document quality 0, from the system default metadata
- `batch-default.json` with document quality 2, from M_{df1}
- `doc-specific.json` with document quality 1, from M_3

The following graphic illustrates the construction of the batch and the documents created from it. In the picture, M_n represents metadata, C_n represents content. Note that the metadata is not literally embedded in the created documents; content and metadata are merely grouped here for illustrative purposes.



The following code snippet is the core of the example, building up a batch of document updates and inserting them into the database:

```
// Create and build up the batch
JSONDocumentManager jdm = client.newJSONDocumentManager();
DocumentWriteSet batch = jdm.newWriteSet();

batch.add("sys-default.json", content1);
batch.addDefault( defaultMetadata);
batch.add("batch-default.json", content2);
batch.add("doc-specific.json", docSpecificMetadata, content3);

// Create the documents
jdm.write(batch);
```

The full example function is shown below. This example uses `StringHandle` for the content, but you can use other handle types, such as `JacksonHandle` or `FileHandle`.

```
package examples;
import com.marklogic.client.io.*;
import com.marklogic.client.document.JSONDocumentManager;
import com.marklogic.client.document.DocumentWriteSet;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.DatabaseClient;


public class Example {
  // replace with your MarkLogic Server connection information
  static String HOST = "localhost";
  static int PORT = 8000;
  static String USER = "user";
  static String PASSWORD = "password";
  static DatabaseClient client = DatabaseClientFactory.newClient(
          HOST, PORT, new DigestAuthContext(USER, PASSWORD));
```

```
  static void example2() {
    // Synthesize input content
    StringHandle content1 = new StringHandle(
            "{\"number\": 1}").withFormat(Format.JSON);
    StringHandle content2 = new StringHandle(
            "{\"number\": 2}").withFormat(Format.JSON);
    StringHandle content3 = new StringHandle(
            "{\"number\": 3}").withFormat(Format.JSON);

    // Synthesize input metadata
    DocumentMetadataHandle defaultMetadata =
            new DocumentMetadataHandle().withQuality(1);
    DocumentMetadataHandle docSpecificMetadata =
            new DocumentMetadataHandle().withQuality(2);

    // Create and build up the batch
    JSONDocumentManager jdm = client.newJSONDocumentManager();
    DocumentWriteSet batch = jdm.newWriteSet();

    batch.add("sys-default.json", content1);
    batch.addDefault( defaultMetadata);
    batch.add("batch-default.json", content2);
    batch.add("doc-specific.json", docSpecificMetadata, content3);

    // Create the documents
    jdm.write(batch);

    // Verify results
    System.out.println(
        "sys-default.json quality: Expected=0, Actual=" +
        jdm.readMetadata("sys-default.json",
                         new DocumentMetadataHandle()).getQuality()
        );
    System.out.println("batch-default.json quality: Expected=" +
        defaultMetadata.getQuality() + ", Actual=" +
        jdm.readMetadata("batch-default.json",
                         new DocumentMetadataHandle()).getQuality()
        );
    System.out.println("doc-specific.json quality: Expected=" +
        docSpecificMetadata.getQuality() + ", Actual=" +
        jdm.readMetadata("batch-default.json",
                         new DocumentMetadataHandle()).getQuality()
        );
  }

  public static void main(String[] args) {
    example2();
  }
}
```

3.1.6 Example: Adding Documents to a Collection

This example demonstrates using batch default metadata to add all documents to the same collection during a multi-document write. For general information about working with metadata, see “Reading, Modifying, and Writing Metadata” on page 43.

Since the metadata in this example request only includes settings for collections metadata, other metadata categories such as permissions and quality use the system default settings. You can add individual documents to a different collection using document-specific metadata or by including additional batch default metadata that uses a different collection; see “Example: Controlling Metadata Through Defaults” on page 77.

The code snippet below inserts 2 JSON documents into the database with a collection named “April 2014”.

```
// Synthesize input metadata
DocumentMetadataHandle defaultMetadata =
    new DocumentMetadataHandle().withCollections("April 2014");

// Create and build up the batch
JSONDocumentManager jdm = client.newJSONDocumentManager();
DocumentWriteSet batch = jdm.newWriteSet();

batch.addDefault(defaultMetadata);
batch.add("coll-doc1.json", content1);
batch.add("coll-doc2.json", content2);
jdm.write(batch);
```

The full example is shown below. This example uses `StringHandle` for the content, but you can use other handle types, such as `JacksonHandle`, `XMLHandle`, or `FileHandle`.

```
package examples;
import com.marklogic.client.io.*;
import com.marklogic.client.query.MatchDocumentSummary;
import com.marklogic.client.query.QueryManager;
import com.marklogic.client.query.StructuredQueryBuilder;
import com.marklogic.client.document.JSONDocumentManager;
import com.marklogic.client.document.DocumentWriteSet;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.DatabaseClient;


public class Example {
  // replace with your MarkLogic Server connection information
  static String HOST = "localhost";
  static int PORT = 8000;
  static String USER = "username";
  static String PASSWORD = "password";
  static DatabaseClient client = DatabaseClientFactory.newClient(
          HOST, PORT, new DigestAuthContext(USER, PASSWORD));

  /// Inserting all documents in a batch into the same collection
```

```
    public static void example3() {
        // Synthesize input content
        StringHandle content1 = new StringHandle(
            "{\"number\": 1}").withFormat(Format.JSON);
        StringHandle content2 = new StringHandle(
            "{\"number\": 2}").withFormat(Format.JSON);
        // Synthesize input metadata
        DocumentMetadataHandle defaultMetadata =
            new DocumentMetadataHandle().withCollections("April 2014");

        // Create and build up the batch
        JSONDocumentManager jdm = client.newJSONDocumentManager();
        DocumentWriteSet batch = jdm.newWriteSet();

        batch.addDefault(defaultMetadata);
        batch.add("coll-doc1.json", content1);
        batch.add("coll-doc2.json", content2);
        jdm.write(batch);

        // Verify results by finding all documents in the collection
        QueryManager qm = client.newQueryManager();
        StructuredQueryBuilder builder = qm.newStructuredQueryBuilder();

        SearchHandle results = qm.search(
            builder.collection("April 2014"), new SearchHandle());
        for (MatchDocumentSummary summary : results.getMatchResults()) {
            System.out.println(summary.getUri());
        }
    }

    public static void main(String[] args) {
        example3();
    }
}
```

3.1.7 Example: Writing a Mixed Document Set

This example uses `GenericDocumentManager` to create a batch that contains documents with a mixture of document types in a single operation. The batch contains a JSON document, an XML document, and a binary document. The following code snippet demonstrates construction of a mixed document batch:

```
GenericDocumentManager gdm = client.newDocumentManager();
DocumentWriteSet batch = gdm.newWriteSet();
batch.add("doc1.json", jsonContent);
batch.add("doc2.xml", xmlContent);
batch.add("doc3.jpg", binaryContent);
gdm.write(batch);
```

When you use `GenericDocumentManager`, you must either use handles that imply a specific document or content type, or explicitly set it. In this example, the JSON and XML contents are provided using a `StringHandle`, and the document type is specified using `withFormat()`. The binary content is read from a file on the local filesystem, using `FileHandle.withMimeType()` to explicitly specify the a MIME type of `image/jpeg`, which implies a binary document.

> **Note:** Document type specific options such as XML repair and binary document metadata extract cannot be performed using `GenericDocumentManager`. You must use a document type specific document manager and a homogeneous batch to use these features.

The full example, including setting of the document/MIME types, is shown below. To run this example in your environment, you need a binary file to subsitute for `/some/jpeg/file.jpg`. If your file is not a JPEG image, change the MIME type in the call to `FileHandle.withMimeType()`.

```
package examples;
import java.io.File;

import com.marklogic.client.io.*;
import com.marklogic.client.document.GenericDocumentManager;
import com.marklogic.client.document.DocumentWriteSet;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.DatabaseClient;

public class standalone {
  // replace with your MarkLogic Server connection information
  static String HOST = "localhost";
  static int PORT = 8000;
  static String USER = "user";
  static String PASSWORD = "password";
  static DatabaseClient client = DatabaseClientFactory.newClient(
          HOST, PORT, new DigestAuthContext(USER, PASSWORD));

  /// Inserting documents with different document types
  static void example4() {
      // Synthesize input content
      StringHandle jsonContent = new StringHandle(
          "{\"key\": \"value\"}").withFormat(Format.JSON);
      StringHandle xmlContent = new StringHandle(
          "<data>some xml content</data>").withFormat(Format.XML);
      String filename = new String("/some/jpeg/file.jpg");
      FileHandle binaryContent =
          new FileHandle().with(new
File(filename)).withMimetype("image/jpeg");

      // Create and build up the batch
      GenericDocumentManager gdm = client.newDocumentManager();
      DocumentWriteSet batch = gdm.newWriteSet();
      batch.add("doc1.json", jsonContent);
      batch.add("doc2.xml", xmlContent);
      batch.add("doc3.jpg", binaryContent);
```

```
        gdm.write(batch);

        // Verify results
        System.out.println("doc1.json exists as: " +
          gdm.exists("doc1.json").getFormat().toString());
        System.out.println("doc2.xml exists as: " +
          gdm.exists("doc2.xml").getFormat().toString());
        System.out.println("doc3.jpg exists as: "
          + gdm.exists("doc3.jpg").getFormat().toString());
    }

    public static void main(String[] args) {
        example4();
    }
}
```

3.2 Read Multiple Documents by URI

You can retrieve multiple documents by URI in a single request by passing multiple URIs to `DocumentManager.read()`. For example, the following code snippet reads 3 documents from the database:

```
DocumentPage documents =
    docMgr.read("doc1.json", "doc2.json", "doc3.json");
while (documents.hasNext()) {
    DocumentRecord document = documents.next();
    // do something with the contents
}
```

The multi-document read operation returns a `DocumentRecord` for each matched URI. Use the `DocumentRecord` to access content and/or metadata about each document. By default, only content is available. To retrieve metadata, use `DocumentManager.setMetadataCategories()`. For example, the following code snippet retrieves both content and document quality for three documents:

```
DatabaseClient client = DatabaseClientFactory.newClient(...);
JSONDocumentManager jdm = client.newJSONDocumentManager();

jdm.setMetadataCategories(Metadata.QUALITY);

DocumentPage documents =
    jdm.read("doc1.json", "doc2.json", "doc3.json");
while (documents.hasNext()) {
    DocumentRecord document = documents.next();
    DocumentMetadataHandle metadata =
        document.getMetadata(new DocumentMetadataHandle());
    System.out.println(
        document.getUri() + ": " + metadata.getQuality());
    // do something with the content
}
```

For more information about metadata categories, see “Reading, Modifying, and Writing Metadata” on page 43.

Multi-document read also supports server side transformations and transaction controls. For more details on these features, see “Apply a Read Transformation” on page 90 and “Multi-Statement Transactions” on page 264.

> **Note:** Applying a transform creates an additional in-memory copy of each document on the server, rather than streaming each document directly out of the database, so memory consumption is higher.

3.3 Read Multiple Documents Matching a Query

Use `com.marklogic.client.document.DocumentManager.search()` to retrieve all documents that match a query. This section covers the following topics:

3.3.1 Overview of Multi-Document Read by Query

To retrieve all documents from the database that match a query, use `DocumentManager.search()`.

The `search` methods of `DocumentManager` differ from `QueryManager.search()` methods in that `DocumentManager` search returns document contents while `QueryManager` search returns search results and facets. Though you can retrieve search results along with contents using `DocumentManager.search()`, and you can retrieve document contents using `QueryManager.search()`, the interfaces are optimized for different use cases.

You can pass a string, structured, or combined query or a QBE to `DocumentManager.write()`. For example, the following code snippet reads all documents that contain the phrase “bird”:

```
JSONDocumentManager jdm = client.newJSONDocumentManager();
QueryManager qm = client.newQueryManager();
StringQueryDefinition query =
        qm.newStringDefinition().withCriteria("bird");

DocumentPage documents = jdm.search(query, 1);
while (documents.hasNext()) {
    DocumentRecord document = documents.next();
    // do something with the contents
}
```

Documents are returned as a `DocumentPage` that you can use to iterate over returned content and metadata. You might have to call `DocumentManager.search()` multiple times to retrieve all matching documents. The number of documents per `DocumentPage` is controlled by `DocumentManager.setPageLength()`. For details, see "Read Documents Incrementally" on page 88.

To return search results along with matching documents, include a `SearchHandle` in your call to `DocumentManager.search()`. For details, see "Return Search Results" on page 88. For example:

```
docMgr.search(query, 1, new SearchHandle());
```

You can apply server-side content transformations to matching documents by configuring a `ServerTransform` on the `QueryDefinition`. For details, see "Apply a Read Transformation" on page 90.

3.3.2 Example: Read Documents Matching a Query

This example demonstrates using a query to retrieve documents from the database using `DocumentManager.search()`. Though you can use any query type, this example focuses on Query By Example.You should be familiar with QBE basics. For details, see "Prototype a Query Using Query By Example" on page 156.

The following QBE matches documents with an XML element or JSON property named "kind" that has a of value "bird":

Format	Query

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <kind>bird</kind>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{ "$query":`
	`  { "kind": "bird" }`
	`}`

The following example code uses the above query to retrieve matching documents. Only document content is returned because no metadata categories are set on the `DocumentManager`.

The number of documents matching the input query is available using `DocumentPage.getTotalResults()`. This number is equivalent to `@total` on a search response and is only an estimate. The document URI, document type, and contents are available on each `DocumentRecord` in the `DocumentPage`.

```
package examples;
```

```
import com.marklogic.client.DatabaseClient;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.document.DocumentPage;
import com.marklogic.client.document.DocumentRecord;
import com.marklogic.client.document.JSONDocumentManager;
import com.marklogic.client.io.Format;
import com.marklogic.client.io.StringHandle;
import com.marklogic.client.query.QueryManager;
import com.marklogic.client.query.RawQueryByExampleDefinition;

public class QueryExample {
    // replace with your MarkLogic Server connection information
    static String HOST = "localhost";
    static int PORT = 8000;
    static String USER = "user";
    static String PASSWORD = "password";
    static DatabaseClient client = DatabaseClientFactory.newClient(
          HOST, PORT, new DigestAuthContext(USER, PASSWORD));

    public static void qbeExample() {
        JSONDocumentManager jdm = client.newJSONDocumentManager();
        QueryManager qm = client.newQueryManager();

        // Build query
        String queryAsString = "{ \"$query\": { \"kind\": \"bird\" }}";
        StringHandle handle = new StringHandle();
        handle.withFormat(Format.JSON).set(queryAsString);
        RawQueryByExampleDefinition query =
            qm.newRawQueryByExampleDefinition(handle);

        // Perform the multi-document read and process results
        DocumentPage documents = jdm.search(query, 1);
        System.out.println("Total matching documents: "
            + documents.getTotalSize());
        for (DocumentRecord document: documents) {
            System.out.println(document.getUri());
           // Do something with the content using document.getContent()
        }
    }

    public static void main(String[] args) {
        qbeExample();
        client.release();
    }
}
```

To perform the equivalent operation using an XML QBE, use an `XMLDocumentManager`. Note that the format of a QBE (XML or JSON) can affect the kinds of documents that match the query. For details, see Scoping a Search by Document Type in the *Search Developer's Guide*.

To use a string, structured, or combined query instead of a QBE, change the `QueryDefinition`. The search operation and results processing are unaffected by the type of query. For more details on query construction, see "Searching" on page 144.

For example, to use a string query to find all documents containing the phrase “bird”, replace the query building section of the above example with the following:

```
StringQueryDefinition query =
    qm.newStringDefinition().withCriteria("bird");
```

To return metadata in addition to content, set one or more metadata categories on the `DocumentManager` prior to the search. Use `DocumentPage.getMetadata()` to access it. For example, the following changes to the above example returns the quality of each document, along with the contents.

```
jdm.setMetadataCategories(Metadata.QUALITY);
DocumentPage documents = jdm.search(query, 1);
System.out.println("Total matching documents: "
    + documents.getTotalSize());
for (DocumentRecord document: documents) {
    System.out.println(document.getUri() + "quality: " +
        document.getMetadata(
            new DocumentMetadataHandle()).getQuality());
    // Do something with the content using document.getContent()
}
```

Use `QueryDefinition.setOptionsName()` to include persistent query options in your search; for details, see “Add Query Options to a Search” on page 87. For example, to apply persistent query options previously installed under the name “myOptions”, pass the options name during query creation:

```
RawQueryByExampleDefinition query =
    qm.newRawQueryByExampleDefinition(handle, "myOptions");
```

3.3.3 Add Query Options to a Search

You can customize your multi-document read using query options in the same way you use them with `QueryManager.search()`:

- Pre-install persistent query options and configure them by name into your `QueryDefinition`.
- Embed dynamic query options into a combined query or QBE. Note that QBE supports only a limited set of query options.

For example, if you previously installed persistent query options under the name “myOptions”, then you can use them in a multi-document read as follows:

```
JSONDocumentManager jdm = client.newJSONDocumentManager();
QueryManager qm = client.newQueryManager();
StringQueryDefinition query =
qm.newStringDefinition("myOptions").withCriteria("bird");

DocumentPage documents = jdm.search(query, 1);
```

For details, see “Query Options” on page 190 and “Apply Dynamic Query Options to Document Searches” on page 159.

3.3.4 Return Search Results

When you use `QueryManager.search()` to find matching documents, you receive a search response that can contain snippets, facets, and other match details. This information is not returned by default with `DocumentManager.search()`, but you can request it by including a SearchHandle in your call. When you include a SearchHandle, you receive both a search response and the matching documents.

For example, the following code snippet requests search results in addition the content of matching documents.

```
SearchHandle results = new SearchHandle().withFormat(Format.XML);
DocumentPage documents = jdm.search(query, 1, results);
for (MatchDocumentSummary match : results.getMatchResults()) {
    // process snippets, facets, and other result info
}
```

3.3.5 Read Documents Incrementally

When you read documents using `DocumentManager.search()`, the page size defined on the `DocumentManager` determines how many documents are returned. You can use this feature, plus the `start` parameter of `DocumentManager.search()` to incrementally read matching documents. The defualt page size is 10 documents. Incrementally reading batches of documents limits resource consumption on both the client and server.

For example, the following function sets the page size and reads all matching documents in batches of no more than 5 documents.

```
public static void pagingExample() {
    JSONDocumentManager jdm = client.newJSONDocumentManager();
    QueryManager qm = client.newQueryManager();
    StringQueryDefinition query =
        qm.newStringDefinition().withCriteria("bird");

    // Retrieve 5 documents per read
    jdm.setPageLength(5);

    // Fetch and process documents incrementally
    int start = 1;
    DocumentPage documents = null;
    while (start == 1 || documents.hasNextPage()) {
        // Read and process one batch of matching documents
        documents = jdm.search(query, start);
        for (DocumentRecord document : documents) {
            // process the content
        }
        // advance starting position to the next page of results
```

```
            start += documents.getPageSize();
        }
    }
```

3.3.6 Extracting a Portion of Each Matching Document

This section illustrates how to use the `extract-document-data` query option with the Java Client API to return selected portions of each matching document instead of the whole document. For details about the option components, see Extracting a Portion of Matching Documents in the *Search Developer's Guide*.

The following example code snippet uses a combined query to specify that the search should only return the portions of matching documents that match the path `/parent/body/target`.

```
String rawQuery =
  "<search xmlns=\"http://marklogic.com/appservices/search\">" +
  "  <qtext>content</qtext>" +
  "  <options xmlns=\"http://marklogic.com/appservices/search\">" +
  "    <extract-document-data selected=\"include\">" +
  "      <extract-path>/parent/body/target</extract-path>" +
  "    </extract-document-data>" +
  "    <return-results>false</return-results>" +
  "  </options>" +
  "</search>";
StringHandle qh = new StringHandle(rawQuery).withFormat(Format.XML);

GenericDocumentManager gdm = client.newDocumentManager();
QueryManager qm = client.newQueryManager();
RawCombinedQueryDefinition query =
qm.newRawCombinedQueryDefinition(qh);

DocumentPage documents = gdm.search(query, 1);
System.out.println("Total matching documents: " +
documents.getTotalSize());
for (DocumentRecord document: documents) {
  System.out.println(document.getUri());
  // Do something with the content using document.getContent()
}
```

You can also use a JSON raw query to search the portions of matching documents that match the path `/parent/body/target`.

portions of matching documents that match the path `/parent/body/target`.

```
String rawQuery =
"{\"options\": {" +
        "\"extract-document-data\": {" +
        "\"selected\": \"include\"," +
        "\"extract-path\": \"/parent/body/target\" } },
\"qtext\" : \"content\" }";
StringHandle qh = new StringHandle(rawQuery).withFormat(Format.JSON);
```

```
GenericDocumentManager gdm = client.newDocumentManager();
QueryManager qm = client.newQueryManager();
RawCombinedQueryDefinition query =
qm.newRawCombinedQueryDefinition(qh);

DocumentPage documents = gdm.search(query, 1);
System.out.println("Total matching documents: " +
documents.getTotalSize());
for (DocumentRecord document: documents) {
  System.out.println(document.getUri());
  // Do something with the content using document.getContent()
```

If one of the matching documents looked like the following:

```
{"parent": {
  "a": "foo",
  "body": { "target":"content" },
  "b": "bar"} }
```

Then the search returns the following sparse projection for this document. There will be one item in the “extracted” array (or one “extracted” element in XML) for each projection in a given context.

```
{ "context":"fn:doc(\"/extract/doc2.json\")",
  "extracted":[{"target":"content"}]
}
```

If you set the `selected` attribute to “all”, “include-with-ancestors”, or “exclude”, then the resulting document just contains the extracted content. For example, if you set `selected` to “include-with-ancestors” in the previous example, then the projected document conains the following. Notice that there are no “context” or “extracted” wrappers.

```
{"parent":{"body":{"target":"content1"}}}
```

You can also use `extract-document-data` to embed sparse projections in the search result summary returned by `QueryManager.search`. For details, see “Extracting a Portion of Matching Documents” on page 180.

3.4 Apply a Read Transformation

When you perform a multi-document read using `DocumentManager.read()` or `DocumentManager.search()`, you can apply a server-side document read transformation by configuring a `ServerTransform` into your `DocumentManager`.

The transform function is called on the returned documents, but not on metadata. If you include search results when reading documents with `DocumentManager.search()`, the transform function is called on both the returned documents and the search response, so the transform must be prepared to handle multiple kinds of input.

For more details, see “Content Transformations” on page 282.

The following example code demonstrates applying a read transform when reading documents that match a query.

```
ServerTransform transform = new ServerTransform(TRANSFORM_NAME);

docMgr.setReadTransform(transform);
docMgr.search(query, start);
```

Note: Applying a transform creates an additional in-memory copy of each document, rather than streaming each document directly out of the database, so memory consumption is higher.

3.5 Selecting a Batch Size

The best batch size for reading and writing multiple documents in a single request depends on the nature of your data. A batch size of 100 is a good starting place for most document collections. Experiment with different batch sizes of data characteristic to your application until you find one that fits within the limits of your MarkLogic Server installation and acceptable request timeouts.

If you need to ingest or retrieve a very large number of documents, you can also consider MarkLogic Content Pump (mlcp), a command line tool for loading and retrieving documents from a MarkLogic database. For details, see Loading Content Using MarkLogic Content Pump in the *Loading Content Into MarkLogic Server Guide*.

For additional tuning tips, see the *Query Performance and Tuning Guide*.

4.0 Asynchronous Multi-Document Operations

The Data Movement Software Development Kit (SDK) is a package in the Java Client API intended for manipulating large numbers of documents and/or metadata through an asynchronous interface that efficiently distributes workload across a MarkLogic cluster. This framework is best suited for long running operations and/or those that manipulate large numbers of documents.

You can use the Data Movement SDK "out-of-the-box" to insert, extract, delete, and transform documents in MarkLogic. You can also easily extend the framework to perform other operations.

The Java Client API also includes simpler interfaces for single-document operations and synchronous multi-document operations. For details, see "Alternative Interfaces" on page 142.

This chapter includes the following topics:

4.1 Terms and Definitions

You should be familiar with the following terms and definitions when working with the Data Movement SDK.

Term	Definition
`job`	An operation or large amount of work to be performed using the Data Movement SDK, such as loading documents into or reading documents from MarkLogic. For details, see “Basic Data Movement Job Life Cycle” on page 96.
`batch`	A small unit of work for a Data Movement job. For details, see “Basic Data Movement Job Life Cycle” on page 96.
`batcher`	An object that encapsulates the characteristics of a job and coordinates the work. The batcher is the job controller. It splits the work requested by a job into batches, coordinates distribution of work, and notifies listeners of events. For details, see “Basic Data Movement Job Life Cycle” on page 96.
`listener`	A callback object that is notified whenever an “interesting” job event occurs. You register listeners through a batcher. For details, see “Working With Listeners” on page 140.
`write job`	A job whose purpose is writing documents and optional metadata to MarkLogic. Write jobs are driven by a `WriteBatcher`. For details, see “Job Types” on page 98 and “Creating and Managing a Write Job” on page 102.
`query job`	A job whose purpose is gathering a set of URIs for documents in the database, and dispatching batches of URIs to listeners for action. The listeners determine the outcome. For example, you can use a query job to read or delete documents from MarkLogic. For details, see “Job Types” on page 98 and “Creating and Managing a Query Job” on page 110.
`job ticket`	An identifier for a job that can be used to retrieve status and other information about a job.
`job report`	A job status report. For details, see “Checking the Status of a Job” on page 131.
`read transformation`	A content, metadata, or search response transformation that is applied on MarkLogic server when you read a document from the database. For details, see “Applying a Read or Write Transformation” on page 130.

Term	Definition
`write transformation`	A content or metadata transformation that is applied on MarkLogic server when you insert a document into the database. The transformation is applied before committing the content. For details, see “Applying a Read or Write Transformation” on page 130.
`in-database transformation`	A content or metadata transformation that is applied on MarkLogic server to content already in the database. The content is not fetched from MarkLogic to the client or sent from the client to MarkLogic. For details, see “Applying an In-Database Transformation” on page 124.
`consistent snapshot`	A consistent snapshot is a conceptual snapshot of the state of the database at a specific point in time. Consistent snapshots are useful for securing an unchanging view of the database for a long-running that accesses documents in the database. For details, see “Using a Consistent Snapshot” on page 114.

4.2 Data Movement Feature Overview

The Data Movement SDK is designed to efficiently operate on large amounts of data. The operations are carried out asynchronously to facilitate spreading the workload across a cluster and to enable your application to continue other processing during a long-running job.

You can use the Data Movement SDK to perform the following operations out-of-the-box. You can easily customize the framework to perform other operations.

- Write data into MarkLogic.
- Read data from MarkLogic.
- Delete data from MarkLogic.
- Apply in-database transformations without fetching data to the client.

The Data Movement SDK provides the following additional benefits.

- A programmatic interface that enables easy integration into existing ETL and data flow tool chains.
- Asynchronous operation. Your application does not block while importing, exporting, deleting, or transforming data. You can incrementally change the workload. For example, as you receive data from an ETL stream, you can add the new input to a running import job.
- Control over workload characteristics, such as thread count and batch size.

- Data format flexibility. When importing documents, you can use any input source supported by Java, such as a file or a stream. The same applies to output when exporting documents.
- Data consistency. You can ensure that a long running export, delete, or transform job operates on the database state in effect when the job started.
- High performance and efficient use of client and server resources. You can tune client and server resource consumption through configuration. The API automatically distributes the server-side workload across your MarkLogic cluster.

Since the Data Movement SDK is part of the Java Client API, your data movement application can leverage the full power of the Java Client API to support high volume operations. For example, you can do the following:

- Use the full suite of search features in the Java Client API to select documents for export, deletion, or in-database transformation. For example, select documents using a string or structured query.
- Operate on documents and document metadata.
- Apply server-side XQuery or JavaScript transformations when importing or exporting documents. You can use the same transformation code and deployment for both data movement and lighter weight document operations.

If you prefer a command line interface, consider using the mlcp command line tool. Be aware that the Data Movement SDK offers some features unavailable to mlcp, and vice versa. For details, see “Alternative Interfaces” on page 142.

4.3 Data Movement Concepts

This section discusses the basic concepts behind the Data Movement SDK.

- Summary of Key Classes and Interfaces
- Basic Data Movement Job Life Cycle
- Job Types
- Object Lifetime Considerations
- How Work is Distributed Across a Cluster

4.3.1 Summary of Key Classes and Interfaces

The following table summarizes the classes and interfaces that drive work in Data Movement SDK. This is not a complete list of available classes and interfaces. For details, see the `com.marklogic.client.datamovement` package in the *Java Client API Documentation*.

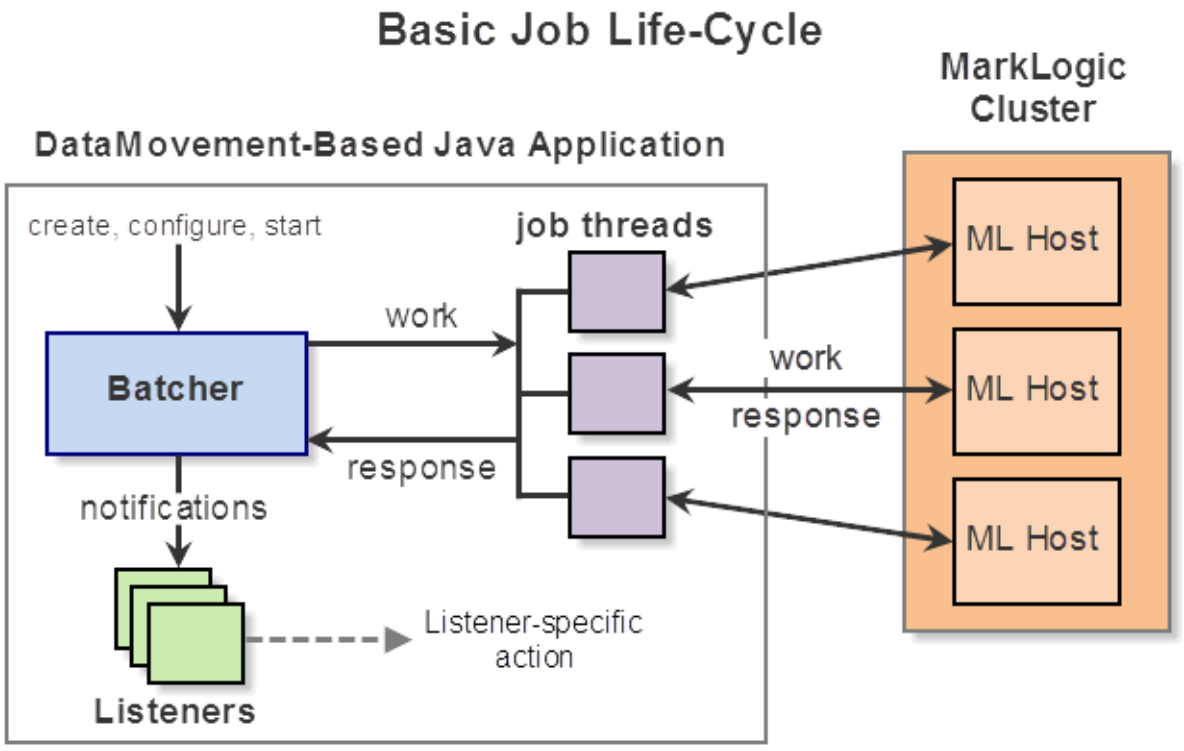
Class	Description
`DataMovementManager`	The primary job control interface. You use a `DataMovementManager` object to create, start, and stop jobs.
`Batcher`	A batcher encapsulates the characteristics of a job (threads, batch size, listeners) and controls the workflow. The subinterfaces of `Batcher` determine the workflow, such as read or write.
`WriteBatcher`	A `Batcher` for jobs that write documents to MarkLogic.
`QueryBatcher`	A `Batcher` for jobs that read documents in MarkLogic. Documents are selected by query or by URI. The action taken on read depends on the `BatchListener`'s configured for the job. For example, you might fetch the documents back to the client, delete them, or apply an in-place transformation.
`BatchListener`	The interface through which you respond to interesting job state changes. For example, you might log a message whenever a batch of documents is successfully written to the database. The events to which you can attach a listener depend on the type of `Batcher`. The DataMovement SDK includes several implementations, and you can define your own.
`BatchFailureListener`	The listener interface for responding to job failure events. The DataMovement SDK includes several implementations, and you can define your own.

4.3.2 Basic Data Movement Job Life Cycle

Data Movement is based on an asynchronous "job" model of interaction with MarkLogic. You create a job (represented by a `Batcher` object), configure its characteristics, and then start the job. Your application does not block while the job runs. Rather, you interact with the job asynchronously via one or more event listeners (represented by a `BatchListener`).

Once you configure and start a job, the underlying API manages distribution of the workload for you, both across the resources available to your client application and across your MarkLogic cluster.

The following diagram illustrates key operations and components common to all Data Movement jobs. Details vary depending on the type of job; for details on specific job types, see “Job Types” on page 98.



The following procedure describes the high level flow in more detail. The details vary, depending on the job type; see “Job Types” on page 98.

1. Create a `DataMovementManager` to manage jobs. This object is intended to be long-lived, and can manage multiple jobs. The `DataMovementManager` is not represented in the above diagram, but it is the agent through which you create, start, and stop jobs.

2. Create a batcher. The batcher acts as the job controller. The type of batcher you create determines the basic job flow (write or query); for details, see “Job Types” on page 98.

3. Configure job characteristics such as batch size and thread count.

4. Attach one or more listeners to interesting job events. The available events depend on the type of job.

5. Start the job. The job runs asynchronously, so this is a non-blocking operation.

6. Depending on the type of job, your application might periodically interact with the batcher to update the state of the running job. For example, periodically add documents to the work queue of a write job.

7. The batcher interacts with MarkLogic on behalf of each batch of work using one of the configured job threads.

8. Whenever an important job life cycle event occurs, the batcher notifies all listeners for that event. For example, a write job notifies batch success listeners whenever a batch of documents is successfully written to MarkLogic.

9. Stop the job when you no longer need it. A job can run indefinitely. Graceful shutdown of a job includes waiting for in-progress batches to complete. For more details, see “Job Control” on page 131.

4.3.3 Job Types

The Data Movement SDK supports the following job types. The job type determines the detailed workflow and the kind of operation a job can perform.

- Write Job
- Query Job

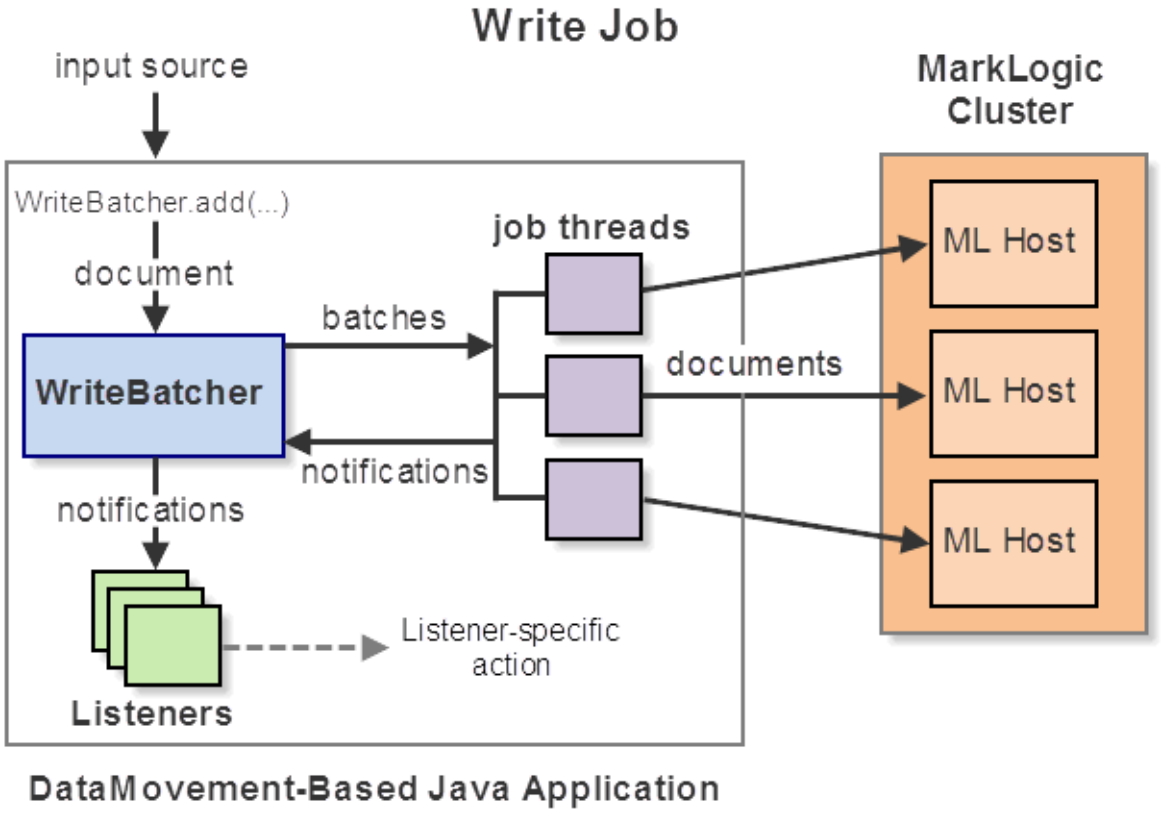
4.3.3.1 Write Job

A *write job* sends batches of documents to MarkLogic for insertion into a database. You can insert both content and metadata.

Your code submits documents to the batcher (job controller), and the batcher submits a batch of documents to MarkLogic whenever a full batch of documents is queued by your application. The number of documents in a batch is a configuration parameter of the job.

Batches are processed in multiple client application threads and distributed across the cluster. The batcher notifies listeners of the success or failure of each batch.

The following diagram gives an overview of the key components and actions of a write job:



For more details, see “Creating and Managing a Write Job” on page 102.

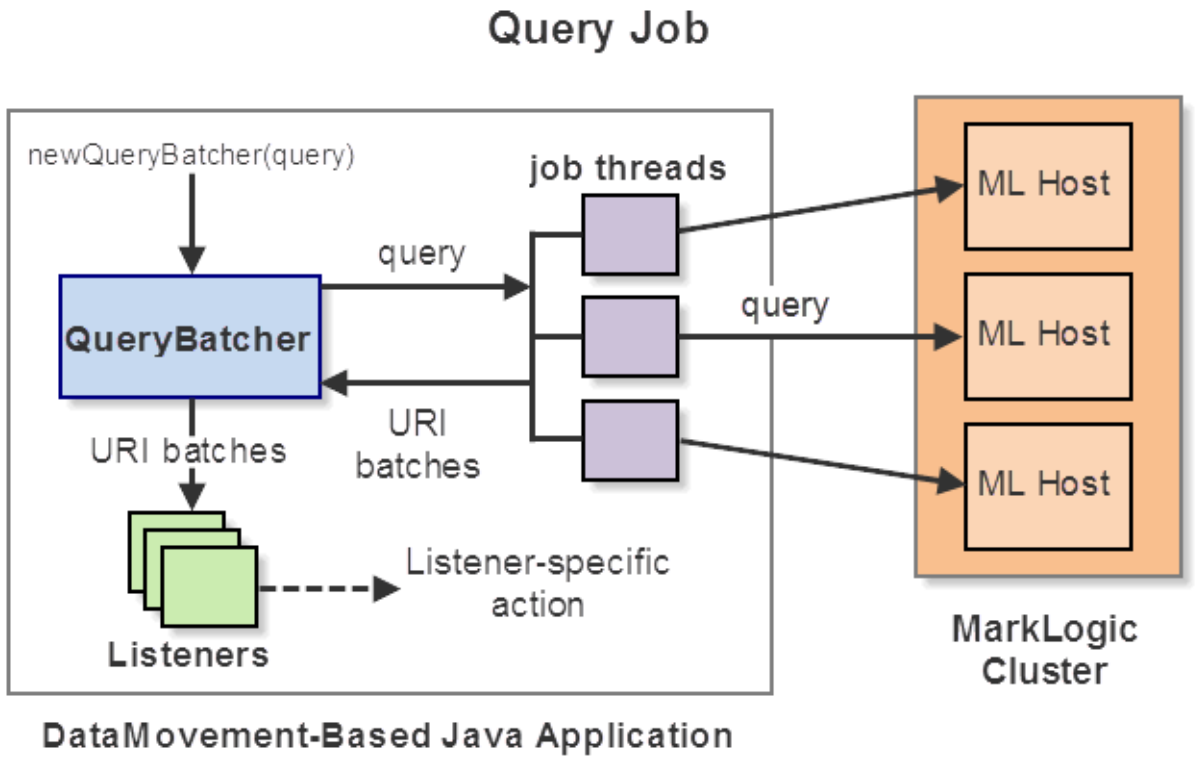
4.3.3.2 Query Job

A query job creates batches of URIs and dispatches each batch to listeners. The batcher gets URIs either by identifying documents that match a query or from a list of URIs you provide as an `Iterator`.

When the job is driven by a query, the batches of URIs are obtained by evaluating the query on MarkLogic and fetching the URIs of subsets of the matching documents. This enables the job to handle large query result sets efficiently.

The action applied to a URI batch is dependent on the listener. For example, a listener might read the documents specified by batch from the database and then save them to the filesystem.

The following diagram gives an overview of the key components and actions of a typical query job.



The Data Movement SDK pre-defines query job listeners that support the following actions:

- Read documents from MarkLogic (`ExportListener` and `ExportToWriterListener`).
- Delete documents from MarkLogic (`DeleteListener`).
- Apply an in-database transformation to documents in MarkLogic (`ApplyTransformListener`).
- Save the URIs of matched documents to a file or other output sink (`UrisToWriterListener`).

You can also create custom listeners to accomplish these and other operations. The pre-defined listeners are meant to serve as guides for creating your own listeners. For more details, see “Working With Listeners” on page 140.

You can also create query jobs that operate on a pre-defined set of URIs, rather than querying MarkLogic to find URIs. In this case, the `Batcher` does not interact with MarkLogic to collect URIs, but your listeners can still interact with MarkLogic to act on the URIs.

For more details, see “Creating and Managing a Query Job” on page 110.

4.3.4 Object Lifetime Considerations

A `DataMovementManager` object is usually a long-lived object. For example, create one when your data movement application starts up, and keep it until your application exits. A `DataMovementManager` object is the agent through which you create, start, and stop jobs. It also manages the MarkLogic connection resources used by jobs (in the form of `DatabaseClient` objects).

A `Batcher` can be released after you stop the job. Jobs cannot be restarted, so a `Batcher` cannot be re-used once the job is stopped.

When you pass a `Closeable` handle to `WriteBatcher.add` or `WriteBatcher.addAs`, the batcher takes responsibility for closing the handle. All `Closeable` content and metadata handles held by the batcher will be closed as soon as possible after a batch is written.

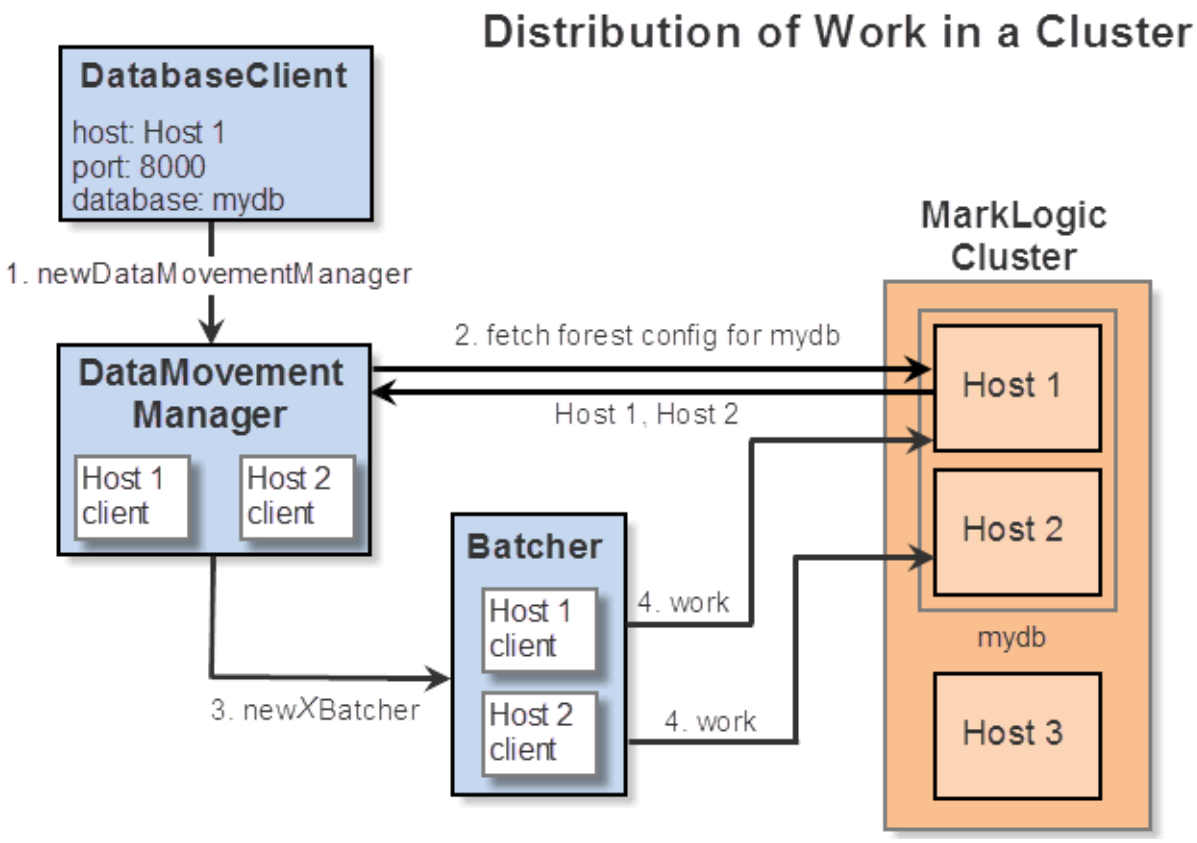
4.3.5 How Work is Distributed Across a Cluster

This section describes how a Data Movement job distributes its workload across a MarkLogic cluster. You do not need to understand this to use the Data Movement SDK, but you might find it useful in understanding the impact of host failures and cluster topology changes.

When you create a `DataMovementManager` object using `DatabaseClient.newDataMovementManager`, the `DataMovementManager` is implicitly associated with the connection held by the creating client. This connection is used to discover which hosts in your MarkLogic cluster contain available forests for the target database.

When you create a batcher using the `DataMovementManager`, the batcher's default configuration includes this forest host data. The batcher distributes its work among these hosts, helping to ensure no single host becomes a chokepoint or gets overloaded.

The following diagram illustrates this discovery process and propagation of forest configuration to a batcher. Assume the job targets the database named “mydb” in cluster that contains three hosts (Host 1, Host 2, and Host 3). Only Host 1 and Host 2 contains forests from “mydb”.



When a forest host becomes unavailable, the batcher attempts to recover by removing the failed host from its host list and redirecting work elsewhere. If the batcher runs out of viable hosts, the job stops.

If you change the forest topology of the database operated on by a job, the batcher will not be aware of this change unless you update the batcher forest configuration information. For details, see “Updating Forest Configuration for a Job” on page 133.

4.4 Creating and Managing a Write Job

A write job inserts documents into a database. The following topics describe creating and managing a write job. The flow of a write job is also illustrated in “Job Types” on page 98.

- Creating a Batcher and Configuring a Write Job
- Attaching Listeners to a Write Job
- Starting a Write Job
- Adding Documents and Metadata to a Job

4.4.1 Creating a Batcher and Configuring a Write Job

You can use a `WriteBatcher` object to load documents into MarkLogic. You can include both content and metadata. Use the batcher to configure runtime characteristics of the job, such as the batch size, and register listeners for batch success and failure events.

The following code snippet configures batch size and thread count. For additional configuration options see “Attaching Listeners to a Write Job” on page 103 and the *Java Client API Documentation*..

```
// Assume "dmm" is a previously created DataMovementManager object.
WriteBatcher batcher = dmm.newWriteBatcher();
batcher.withBatchSize(1000)
       .withThreadCount(10)
       /* ... additional configuration ... */
       ;
```

The order in which you configure job characteristics and attach listeners is not significant, other than that listeners for the same event are invoked in the order in which they’re attached.

For an end-to-end example, see “Example: Loading Documents From the Filesystem” on page 108.

4.4.2 Attaching Listeners to a Write Job

Whenever a `WriteBatcher` accumulates a batch of documents, it dispatches the batch to MarkLogic for writing. The success or failure of committing the batch to the database is reported back to the batcher, which in turn notifies appropriate listeners.

You can attach listeners to a `WriteBatcher` for the following events:

- Batch success: A batch success event occurs whenever all the documents in a batch are successfully committed to MarkLogic. Use `WriteBatcher.onBatchSuccess` to attach a listener to this event.
- Batch failure: A batch failure event occurs whenever at least one document in a batch cannot be committed to MarkLogic. Use `WriteBatcher.onBatchFailure` to attach a listener to this event.

You are not required to attach a listener, but doing so gives your application access to information that may not be included in the default logging and error handling, as well as more control over your job. Tracking success and failure details can also assist in error recovery.

Listeners for the same event are invoked in the order in which they are attached to the batcher.

The following code snippet illustrates attaching a success and a failure listener, both in the form of a lambda function.

```
// Assume "dmm" is a previously created DataMovementManager object.
WriteBatcher batcher = dmm.newWriteBatcher();
batcher.onBatchSuccess(batch-> {/* take some action */})
       .onBatchFailure((batch,throwable) -> {/* take some action */})
       // ...additional configuration...

dmm.startJob(batcher);
```

To learn more about listeners, see “Working With Listeners” on page 140.

For an end-to-end example, see “Example: Loading Documents From the Filesystem” on page 108.

4.4.3 Starting a Write Job

Start a job using `DataMovementManager.startJob`. For example:

```
// Assume "dmm" is a previously created DataMovementManager object.
WriteBatcher batcher = dmm.newWriteBatcher();
// ... configure the job and attach listeners ...

JobTicket ticket = dmm.startJob(batcher);
```

You receive a `JobTicket` that can be used to check status or stop the job. You can also retrieve the ticket later from the batcher.

You should not change the configuration of a job after you start it, with the possible (rare) exception of updating the forest configuration if your cluster topology changes; for details, see “Updating Forest Configuration for a Job” on page 133. The job will run until you stop it or a fatal error occurs. For more details, see “Job Control” on page 131.

For an end-to-end example, see “Example: Loading Documents From the Filesystem” on page 108.

4.4.4 Adding Documents and Metadata to a Job

While the job is running, add documents to the job using `WriteBatcher.add` or `WriteBatcher.addAs`. You can add document content or a combination of content and metadata.

A `WriteBatcher` object is thread safe, so you can add data to the job from multiple threads.

Whenever your application adds enough documents to the batcher to compose a full batch, the batcher dispatches the batch to one of its job threads for uploading to MarkLogic. Each batch of documents is committed as a single transaction, so if any document in a batch cannot be committed, the whole batch fails. The success or failure of the batch is reported to appropriate attached listeners.

The batcher will always wait for a full batch by default. If your input rate is very slow, you can periodically flush partial batches using `WriteBatcher.flushAsync`.

The following code snippet adds files from a directory (signified by the `DATA_DIR` variable) to a job. For an end-to-end example, see "Example: Loading Documents From the Filesystem" on page 108.

```
try {
    Files.walk(Paths.get(DATA_DIR))
         .filter(Files::isRegularFile)
         .forEach(p -> {
                   String uri = "/dmsdk/" + p.getFileName().toString();
                   FileHandle handle =
                       new FileHandle().with(p.toFile());
                   batcher.add(uri, handle);
                 });
} catch (IOException e) {
    e.printStackTrace();
}
```

The batcher takes responsibility for closing any `Closeable` content or metadata handles you pass in. Such handles are closed by the batcher as soon as possible after the resource is written to MarkLogic.

> **Note:** If you have a resource that needs to be closed after writing, but is not closed by the handle, you should override the `close` method of your handle and dispose of your resource there.

4.4.5 Stopping a Write Job

Graceful shutdown of a write job should include draining the document queue before shutting down the job. You usually want to ensure that all documents that have been added to the job are fully processed (either committed to the database or rejected due to an error).

You can achieve graceful shutdown with the following steps:

1. Stop any activity adding work to the job. That is, stop calling `WriteBatcher.add` or `WriteBatcher.addAs`. As long as you keep adding work to the job, the batcher will keep dispatching work to job threads whenever a batch accumulates.

2. Call `WriteBatcher.flushAndWait`. The batcher dispatches any partial batch in its work queue, and then waits for in-progress batches to complete.

3. Call `DataMovementManager.stopJob`. The job is marked as stopped. Queued (but not yet started) tasks are cancelled. Subsequent calls to `WriteBatcher.add`, `WriteBatcher.addAs`, `WriteBatcher.flushAndWait`, and `WriteBatcher.flushAsync` will throw an exception.

If you are concerned that the JVM might exit before all work completes, you can call `WriteBatcher.awaitCompletion` after you call `stopJob`.

The following code snippet demonstrates a graceful shutdown.

```
DataMovementManager dmm = ...;
WriteBatcher batcher = ...;

// ... disable any input sources ...

batcher.flushAndWait();
dmm.stopJob(ticket);
```

The following walkthrough explores the interactions between flush and stop in more detail to help you understand the tradeoff if you to shut a job down prematurely by just calling `stopJob`.

Suppose you have a write job with a batch size of 100, and the job is in the following state:

- Completed: Batches 1-3. That is, 300 documents have been written to MarkLogic and the listeners for these batches have completed their work.
- In-Progress: Batch 4 is being written to MarkLogic, but has not yet completed.
- In-Progress: Batch 5 has been written to MarkLogic, but the listeners have not completed their work.
- Not Started: 75 documents are sitting in the batcher's work queue, waiting for a full batch to accumulate.

Now, consider the following possible shutdown scenarios:

1. Stop calling `WriteBatcher.add` and `WriterBatcher.addAs`, then call `WriteBatcher.flushAndWait`, followed by `DataMovementManager.stopJob`.

    - The `flushAndWait` call creates a batch from the 75 documents in queue, then blocks until this batch and batches 4 and 5 complete.
    - No new batches will be started, and no batches will be in progress when you call `stopJob` because no new work is flowing into the job when you call flush.

2. You call `WriteBatcher.flushAndWait`, followed by `DataMovementManager.stopJob`.

    - The `flushAndWait` call creates a batch from the 75 documents in queue, then blocks until this batch and batches 4 and 5 complete.
    - Any batches that start between calling `flushAndWait` and `stopJob` will complete, assuming the JVM does not exit.
    - Any partial batch that accumulates between the calls is discarded.

    Calling `flushAsync` instead of `flushAndWait` has the same outcome, if the JVM does not exit before in-progress batches complete.

3. You call `DataMovementManager.stopJob`.

    - The 75 documents in the queue are discarded.
    - Batches 4 and 5 will complete, assuming the JVM does not exit.
    - Any subsequent attempt to call `WriteBatcher.add` or `WriteBatcher.addAs` throws an exception, so no additional batches are started or documents lost.

Only sequence #1 ensures that no submitted documents are lost.

4.4.6 Write Job Performance Considerations

You should consider the following factors when configuring and tuning a write job:

- Batch Size
- Thread Count
- Work Item Input Rate
- Listener Design

4.4.6.1 Batch Size

The batch size configuration parameter of a `WriteBatcher` is the number of items that are sent to MarkLogic at once. The “ideal” batch size depends on many factors, including the size of the input documents and network latency. A batch size in the range 100-1000 works for most applications.

The following list calls out some factors you should consider when choosing a batch size:

- All items in a batch are sent to MarkLogic in a single request and committed as a single transaction.
- If your job updates existing documents, locks must be acquired on those documents and held for the lifetime of the transaction. A large batch size can thus potentially increase lock contention and affect overall application performance.
- Selecting a batch size is a speed vs. memory tradeoff. Each request to MarkLogic introduces overhead, but all the items in a batch must stay in memory until the batch is processed, so a larger batch size consumes more memory.
- Since the batcher will not send any queued items until a full batch accumulates, you should also consider the input rate of your application. A large batch size and a slow input rate can cause items to be in a pending state for a long time. You can avoid this by periodically calling `WriteBatcher.flushAsync` or `WriteBatcher.flushAndWait`.

4.4.6.2 Thread Count

The thread count configuration parameter of a `WriteBatcher` is the number of threads in the client JVM that will be dedicated to writing batches to MarkLogic. The threads operate in parallel, each servicing one batch at a time.

Ideally, you should choose a thread count that will keep most of the job threads busy and keep MarkLogic busy without overwhelming your cluster. You should usually configure at least as many client threads as hosts containing forests in the target database. The default is one thread per forest host.

4.4.6.3 Work Item Input Rate

Write job performance can be affected by the input rate. That is, by the rate at which you add documents to the batcher.

If you queue documents much faster than the batcher's job threads can process batches, you can overwhelm the batcher. When this happens, the batcher adopts a strategy that uses submitting threads instead of the busy job threads. This effectively throttles submitting threads and prevents the task queue from using too much memory, while still enabling the job to progress.

To tune performance, you can adjust the number of threads adding work to the batcher or the rate at which items are added.

4.4.6.4 Listener Design

When a batch succeeds or fails, the job thread that submitted the batch invokes all the appropriate listeners. If you register a listener that takes a long time to complete, it slows down the notification of other listeners for the same event, and slows down the rate at which the job can complete batches.

A listener can also slow down a job if it calls synchronized resources since lock contention can occur.

4.4.7 Example: Loading Documents From the Filesystem

The following example creates and configures a `WriteBatcher` job, and then feeds the job files all the files in a directory on the filesystem.

Though this example simply pulls input from the filesystem, it could come from any source supported by Java. For example, the application could asynchronously receive data from an ETL pipeline, a message queue, or periodically pull from a file system drop box.

The example attaches listeners to the batch success and batch failure events. The success listener logs the number of documents written so far, and the failure listener simply rethrows the failure exception. A production application would have more sophisticated listeners.

```
package examples;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

import com.marklogic.client.io.*;
import com.marklogic.client.datamovement.DataMovementManager;
import com.marklogic.client.datamovement.WriteBatcher;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.DatabaseClient;
import com.marklogic.client.DatabaseClientFactory.DigestAuthContext;


public class DMExamples {
    // replace with your MarkLogic Server connection information
    static String HOST = "localhost";
    static int PORT = 8000;
    static String USER = "username";
    static String PASSWORD = "password";
    private static DatabaseClient client =
            DatabaseClientFactory.newClient(
                HOST, PORT, new DigestAuthContext(USER, PASSWORD));
    private static String DATA_DIR = "/your/input/dir/";

    // Loading files into the database asynchronously
    public static void importDocs() {
        // create and configure the job
        DataMovementManager dmm = client.newDataMovementManager();
        WriteBatcher batcher = dmm.newWriteBatcher();
        batcher.withBatchSize(5)
               .withThreadCount(3)
               .onBatchSuccess(batch-> {
                   System.out.println(
                       batch.getTimestamp().getTime() +
                       " documents written: " +
                       batch.getJobWritesSoFar());
                })
               .onBatchFailure((batch,throwable) -> {
                   throwable.printStackTrace();
                });

        // start the job and feed input to the batcher
        dmm.startJob(batcher);
        try {
            Files.walk(Paths.get(DATA_DIR))
                 .filter(Files::isRegularFile)
                 .forEach(p -> {
                   String uri = "/dmsdk/" + p.getFileName().toString();
                   FileHandle handle =
```

```
                        new FileHandle().with(p.toFile());
                    batcher.add(uri, handle);
                });
            } catch (IOException e) {
                e.printStackTrace();
            }

            // Start any partial batches waiting for more input, then wait
            // for all batches to complete. This call will block.
            batcher.flushAndWait();
            dmm.stopJob(batcher);
        }

        public static void main(String[] args) {
            importDocs();
        }
    }
```

4.5 Creating and Managing a Query Job

A query job takes either a query or a list of URIs as input, and distributes batches of URIs to listeners for action. The flow of a query job is outlined in “Job Types” on page 98.

The outcome of a query job is dependent on the actions taken by the listeners. This section covers the following topics common to all query jobs, regardless of the end goal.

- Creating and Configuring a Query Job
- Attaching Listeners to a Query Job
- Starting a Query Job
- Stopping a Query Job
- Using a Consistent Snapshot
- Performance Considerations for Query Jobs

To learn more about specific query job use cases, see the following topics:

- Reading Documents from MarkLogic
- Applying an In-Database Transformation
- Deleting Documents from a Database

4.5.1 Creating and Configuring a Query Job

To run a query job, use a `QueryBatcher` object created with `DataMovementManager.newQueryBatcher`. A `QueryBatcher` distributes batches of URIs to listeners registerd for the “URIs ready” event.

The set of URIs that a query job operates on can come from the following sources:

- A string query, structured query, or combined query. The job retrieves batches of URIs of matching documents from MarkLogic.
- A raw or unstructured query. Because it requires no transformation on the server, a raw query is faster than a structured query.
- An application-defined list of URIs (in the form of an `Iterator`). The job splits these URIs into batches.

The following code snippet constructs a `QueryBatcher` based on a structured query. The query is a directory query on the path "`/dmsdk/`".

```
// Assume "client" is a previously created DatabaseClient object.
QueryManager qm = client.newQueryManager();
StructuredQueryBuilder sqb = qm.newStructuredQueryBuilder();
StructuredQueryDefinition query = sqb.directory(true, "/dmsdk/");

// Create the batcher
DataMovementManager dmm = client.newDataMovementManager();
QueryBatcher batcher = dmm.newQueryBatcher(query);
```

The following code snippet takes a raw query (`querydefRawCts`).

```
QueryBatcher queryBatcher2 =
dmManager.newQueryBatcher(querydefRawCts);
```

Note: The raw CTS query is the representation of a query that executes most quickly. Although the Java API supports other kinds of raw queries, including a raw query that is equivalent to a structured query, raw queries are not as fast as a raw CTS query.

The following code snippet constructs a `QueryBatcher` based on a list of URIs.

```
// Assume "client" is a previously created DatabaseClient object.
DataMovementManager dmm = client.newDataMovementManager();
String uris[] =
    {"/dmsdk/doc1.xml", "/dmsdk/doc3.xml", "/dmsdk/doc5.xml"};
QueryBatcher batcher =
    dmm.newQueryBatcher(Arrays.asList(uris).iterator());
```

You can configure runtime characteristics of the job, such as the batch size, thread count and whether or not to use a consistent snapshot of the documents in the database.

Note: Whether or not to use a consistent snapshot is an important consideration for query jobs. For details, see "Using a Consistent Snapshot" on page 114.

The following code snippet sets the batch size and thread count, and imposes a consistent snapshot requirement for a previously created batcher.

```
batcher.withBatchSize(100)
       .withThreadCount(10)
       .withConsistenSnapshot()
       /* ... additional configuration ... */
       ;
```

For more complete examples, see the following topics:

- “Example: Exporting Documents that Match a Query” on page 122
- “Example: Applying an In-Database Transformation” on page 127
- “Deleting Documents from a Database” on page 129

The order in which you configure job characteristics and attach listeners is not significant, other than that listeners for the same event are invoked in the order in which they’re attached.

You should also attach at least one listener; for details, see “Attaching Listeners to a Query Job” on page 112.

4.5.2 Attaching Listeners to a Query Job

Whenever a `QueryBatcher` accumulates a batch of URIs, it dispatches the URIs to the listeners attached using `QueryBatcher.onUrisReady`. If you do not attach at least one `onUrisReady` listener, the job will not do anything meaningful.

You can attach listeners to a `QueryBatcher` for the following events:

- URIs ready: This event occurs whenever the batcher accumulates a batch of URIs to be processed. Use `QueryBatcher.onUrisReady` to attach a `QueryBatchListener` to this event.
- Query failure: This event can occur when you use a query to derive the list of URIs for a job, and the query fails for some reason. Use `QueryBatcher.onQueryFailure` to attach a `QueryFailureListener` to this event.

You should attach at least one success listener and one failure listener to perform application-specific monitoring and business logic. A listener has access to information that may not be captured by the default logging from the Java Client API.

The action taken when a batch of URIs is available is up to the `onUrisReady` listeners. Data Movement SDK comes with listeners that support the following operations.

- Read documents from MarkLogic (`ExportListener`, `ExportToWriterListener`). For details, see“Reading Documents from MarkLogic” on page 118.
- Apply an in-database transformation to documents in MarkLogic (`ApplyTransformListener`). For details, see “Applying an In-Database Transformation” on page 124.
- Delete documents in MarkLogic (`DeleteListener`). For details, see “Deleting Documents from a Database” on page 129.

- Log or otherwise track progress of a query job (`ProgressListener`).

You can also create your own listeners. The listeners that come with Data Movement SDK are meant to serve as a starting point for your customizations.

The following code snippet illustrates attaching listeners to a query job. This job prints the URIs in each batch to stdout.

```
// Assume "dmm" is a previously created DatabaseMovementManager object
// and "query" is a previously created StructuredQueryDefinition.

DataMovementManager dmm = client.newDataMovementManager();
QueryBatcher batcher = dmm.newQueryBatcher(query);

batcher.onUrisReady(batch -> {
            for (String uri : batch.getItems()) {
                System.out.println(uri);
            }
        })
       .onQueryFailure( exception -> exception.printStackTrace() );
       // ...additional configuration...

dmm.startJob(batcher);
```

The order in which you configure job characteristics and attach listeners is not significant, other than that listeners for the same event are invoked in the order in which they're attached.

To learn more about listeners, see "Working With Listeners" on page 140.

4.5.3 Starting a Query Job

Start a job using `DataMovementManager.startJob`. For example:

```
// Assume "client" is a previously created DatabaseClient object
DataMovementManager dmm = client.newDataMovementManager();
QueryBatcher batcher = dmm.newQueryBatcher(someQuery);
// ... configure the job and attach listeners ...

JobTicket ticket = dmm.startJob(batcher);
```

You receive a `JobTicket` that can be used to check status or stop the job. You can also retrieve the ticket later from the batcher.

You should not change the configuration of a job after you start it. The job will run until you stop it or a fatal error occurs. For more details, see "Job Control" on page 131.

4.5.4 Stopping a Query Job

A query job will go on dispatching batches of URIs to its listeners until all batches have been dispatched or you call `DataMovementManager.stopJob`. Follow these steps to ensure the listeners complete processing all URI batches before shutdown:

1. Call `QueryBatcher.awaitCompletion`. This call blocks until all URIs are processed. You can set a time limit on how long to block, but the job will go on processing batches after the timeout expires.

2. Call `DataMovementManager.stopJob`. The job will not start any additional batches. In-progress batches will run to completion unless the JVM exits. Resources are released as the in-progress work completes.

For example, suppose you have a query job that will ultimately fetch 10 batches of URIs from MarkLogic, and the job is in the following state:

- Completed: Batches 1-3. That is, the URIs were dispatched to listeners and the listeners completed their work.
- In-Progress: Batch 4 is awaiting query results from MarkLogic.
- In-Progress: Batch 5 has been dispatched to the listeners, but the listeners have not completed their work.
- Not Started: Batches 6-10 not yet assigned to any job threads.

If you call `awaitCompletion`, the call will block until batches 4-10 are completed.

If you instead call `stopJob`, batches 4 and 5 will complete (unless the JVM exits), but batches 6-10 will not be processed, even if they could have been started while waiting on batches 4 and 5.

The following code gracefully shuts down a query job after it completes all work:

```
DataMovementManager dmm = ...;
QueryBatcher batcher = ...;

batcher.awaitCompletion();
dmm.stopJob(ticket);
```

The following code shuts down a job without necessarily completing all work. Work in progress when you call stopJob completes, but no additional work is done. The call to `awaitCompletion` is optional, but can be useful to prevent the application from exiting before work is completed.

```
DataMovementManager dmm = ...;
QueryBatcher batcher = ...;

dmm.stopJob(ticket);
batcher.awaitCompletion();
```

4.5.5 Using a Consistent Snapshot

"Consistent snapshot" is a configuration option for a query job that causes the query driving the job to be evaluated against the state of the database at the point in time when the job begins.

- When to Use a Consistent Snapshot

4.5.5.1 When to Use a Consistent Snapshot

You must use a consistent snapshot if your job meets the following criteria:

- The job is driven by a query (rather than an application-defined list of URIs), and
- The job (or other activity) modifies the database in way that can cause successive evaluations of the query to return different results.

Failing to use a consistent snapshot under these circumstances can cause inconsistent and unpredictable job results. For details, see "The Problem Solved by a Consistent Snapshot" on page 115.

For example, you should always use a consistent snapshot when using `DeleteListener` or `ApplyTransformListener` with a query-driven job.

You might also want to use a consistent snapshot when reading documents from the database if you need to capture a deterministic set of documents and there is a possibility of the database contents changing while your job runs.

4.5.5.2 How to Use a Consistent Snapshot

To enable the use of a consistent snapshot, call `QueryBatcher.withConsistentSnapshot` and ensure your database configuration supports point-in-time queries.

The following code snippet configures a query job to use a consistent snapshot:

```
QueryBatcher batcher = dmm.newQueryBatcher(someQuery);
batcher.withConsistentSnapshot();
```

This causes the job to evaluate the query as a point-in-time query. You might have to change your database configuration to enable point-in-time queries by setting a merge timestamp. For details, see Enabling Point-In-Time Queries in the Admin Interface in the *Application Developer's Guide*.

You might also want to use a consistent snapshot in your listeners. For example, `ExportListener` and `ExportToWriterListener` have a `withConsistentSnapshot` method you can use to ensure the listeners capture exactly the same set of documents as were matched by the query.

4.5.5.3 The Problem Solved by a Consistent Snapshot

When you drive a query job using a query (rather than a static list of URIs), the batcher fetches the URIs for matching documents incrementally, rather than fetching them all at once and holding them in memory.

The batches are fetched using the same pagination model that the search interfaces use to fetch results incrementally, specifying the desired page by a starting position in the results plus a page length. The examples below illustrate the problems that can occur if the query results are changing as the job runs.

Suppose the initial query for a job matches documents with the following URIs, and that the batch (page) size is 3. When the job fetches the first page, it gets the URIs for `doc1`, `doc2`, `doc3`.

```
doc1 doc2 doc3  doc4 doc5 doc6  doc7 doc8 doc9  doc10
--------------  --------------  --------------  -----
    page 1          page 2          page 3      page 4
```

While that batch of URIs is being processed, a change in the database causes `doc3` to no longer match the query. Thus, the query results now look like the following:

```
doc1 doc2 doc4  doc5 doc6 doc7  doc8 doc9 doc10
--------------  --------------  ---------------
    page 1          page 2          page 3
```

When the job requests the next page of matches, beginning at position 4, it gets back the URIs for `doc5`, `doc6`, and `doc7`. Notice that `doc4` has been skipped because it is now in the first page of results, which has already been processed from the perspective of the job.

A similar problem can occur if the database changes in a way that adds a new document to the query results. Imagine that, after the job processes the first batch of URIs, a new `docA` matches the query and is part of the first page, as follows:

```
doc1 doc2 docA  doc3 doc4 doc5  doc6 doc7 doc8  doc9 doc10
--------------  --------------  --------------  ----------
    page 1          page 2          page 3         page 4
```

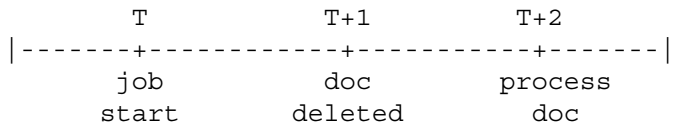
When the job fetches page 2, the batch includes `doc3` again, which has already been processed. If the job is applying an in-database transformation, this double processing could have an undesirable effect.

If you use a consistent snapshot of the database state at the beginning of a query job, then the query always matches the same documents.

You might also want to use a consistent snapshot in your query job listeners, depending on the operation.

Consider a query job that uses `ExportListener` to read documents from the database. Say the batcher is running at a consistent snapshot, but the listener is not. Some time after the start of the job, one of the documents matching the query is deleted. The deleted document URI will still be included in a batch because of the consistent snapshot. However, the listener will get an error trying to read the nonexistent document.

The following diagram illustrates this case. The job starts at some time *T*. The document is deleted at time *T+1*. At *T+2*, the job produces a batch that includes the URI for the deleted document and passes it to the listener. If the listener is not pinned to a point-in-time, then it will find the deleted document does not exist, which might result in an error.



If you call `ExportListener.withConsistentSnapshot` as well as `QueryBatcher.withConsistentSnapshot`, then both the query evaluation and the URI processing will be carried out against a fixed snapshot of the database.

`ExportToWriterListener` also has a `withConsistentSnapshot` method.

4.5.6 Performance Considerations for Query Jobs

You should consider the following factors when configuring and tuning a query job:

- Batch Size
- Thread Count
- Listener Design

4.5.6.1 Batch Size

For a query-driven job, the batch size configuration parameter of a `QueryBatcher` is the number of URIs that are fetched from MarkLogic at once. For a URI iterator driven job, batch size is the number of URIs the batcher picks off the list at once. In both cases, the batch size determines the number of items sent to the listeners for processing.

The “ideal” batch size depends on many factors, including the size of the input documents and network latency. A batch size in the range 100-1000 works for most applications.

The following list calls out some factors you should consider when choosing a batch size:

- Selecting a batch size is a speed vs. memory tradeoff. Each request to MarkLogic introduces overhead, but all the items in a batch must stay in memory until the batch is processed, so a larger batch size consumes more memory.
- Consider how batch size interacts with the implementation of your listener. For example, ExportListener fetches all the documents in a batch from MarkLogic in a single request, so a large batch size causes the listener to hold many documents in memory. For more details, see “Listener Design” on page 118.

4.5.6.2 Thread Count

The thread count configuration parameter of a `QueryBatcher` is the number of threads in the client JVM that will be dedicated to processing URI batches. The threads operate in parallel, each servicing one batch at a time.

Ideally, you should choose a thread count that will keep most of the job threads busy. If your listener interacts with MarkLogic, you should ideally also keep MarkLogic busy without overwhelming the cluster. For a job that interacts with MarkLogic, you should usually have more client threads than hosts containing forests in the target database.

4.5.6.3 Listener Design

The performance of a query job is heavily depending on the processing performed by the `QueryBatcher.onUrisReady` listeners.

When a batch of URIs is ready for processing, the batcher invokes each `onUrisReady` listener, in the order in which they were register. If you register a listener that takes a long time to complete, it delays the execution of other listeners for the same event, and slows down the rate at which the job can complete batches.

A listener can also slow down a job if it calls synchronized resources since lock contention can occur.

If one of your listeners is too slow, you can design it to do its processing in a separate thread, allowing control to return to the job and other listeners to execute.

Listener performance can be affected by batch size. For example, an `ApplyTransformListener` performs all the transformations for a batch of URIs as a single transaction. An open transaction holds locks on fragments with pending updates, potentially increasing lock contention and affecting overall application performance. If you run into lock contention, you might be able to address it by using a smaller batch size.

4.6 Reading Documents from MarkLogic

To read documents and/or metadata from MarkLogic using the Data Movement SDK, use a `QueryBatcher` and attach an `ExportListener`, `ExportToWriterListener`, or equivalent custom `QueryBatchListener` to `onUrisReady`. An export listener also has attached listeners. These listeners take action when the export listener has a document available for processing.

This section only details how to use `ExportListener` and `ExportToWriterListener` to read documents from MarkLogic with a query job. However, you can create your own listener for reading documents.

For more details, see the following topics:

- Using ExportListener to Read Documents

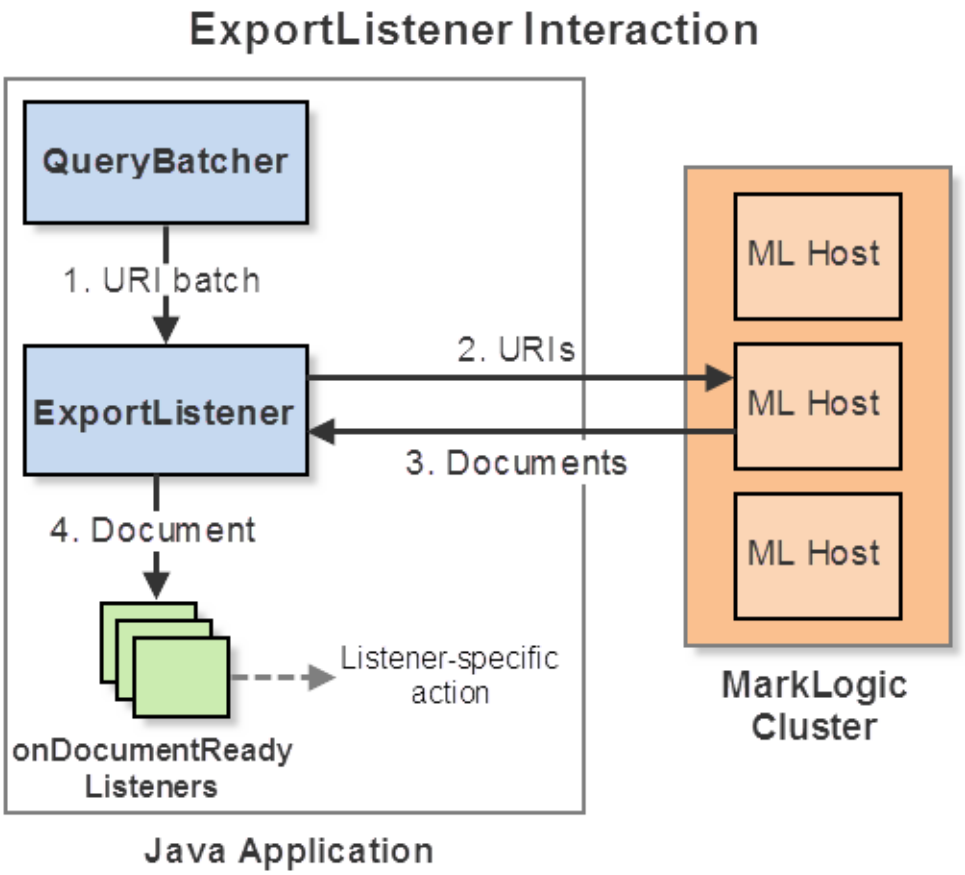
- Using ExportToWriterListener to Read Documents
- Example: Exporting Documents that Match a Query

This section assumes you are familiar with query job basics. If not, review “Creating and Managing a Query Job” on page 110.

4.6.1 Using ExportListener to Read Documents

When an `ExportListener` receives a batch of URIs from a `QueryBatcher`, it reads these documents from MarkLogic, and then dispatches each document to its own listener(s). Attach per-document listeners using `ExportListener.onDocumentReady`. For example, you might register a document listener that writes a document to the filesystem.

The following diagram illustrates the flow between QueryBatcher, ExportListener, and document listeners.



You can configure aspects of the `ExportListener` document read operation such as the following. For a complete list, refer to the *Java Client API Documentation*.

- Fetch metadata such as collections or properties, as well as document content. See `ExportListener.withMetadataCategory`.

- Use a consistent snapshot to fetch documents as they were when the query job started. See `ExportListener.withConsistenSnapshot` and "Using a Consistent Snapshot" on page 114.
- Apply a server-side read transform to each document before returning it to the client application. See `ExportListener.withTransform`.

The `ExportListener` uses the interfaces described in "Synchronous Multi-Document Operations" on page 70 to fetch the documents, so the listener blocks during the fetch. Each fetched document (and its metadata) is made available to the listeners as a `DocumentRecord`. This is the same interface used by the synchronous interfaces, such as the multi-document read shown in "Read Multiple Documents by URI" on page 83.

The following code snippet attaches a document listener in the form of a lambda function to an ExportListener. The document listener simply writes the return document to a known place in the filesystem (*`DATA_DIR`*), with a filename corresponding to the last path step in the URI.

```
// ...construct a query...
QueryBatcher batcher = dmm.newQueryBatcher(query);

batcher.onUrisReady(
    new ExportListener()
        .onDocumentReady(doc-> {
            String uriParts[] = doc.getUri().split("/");
            try {
                Files.write(
                    Paths.get(DATA_DIR, "output",
                              uriParts[uriParts.length - 1]),
                    doc.getContent(new StringHandle()).toBuffer());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }))
    // ...additional configuration...
```

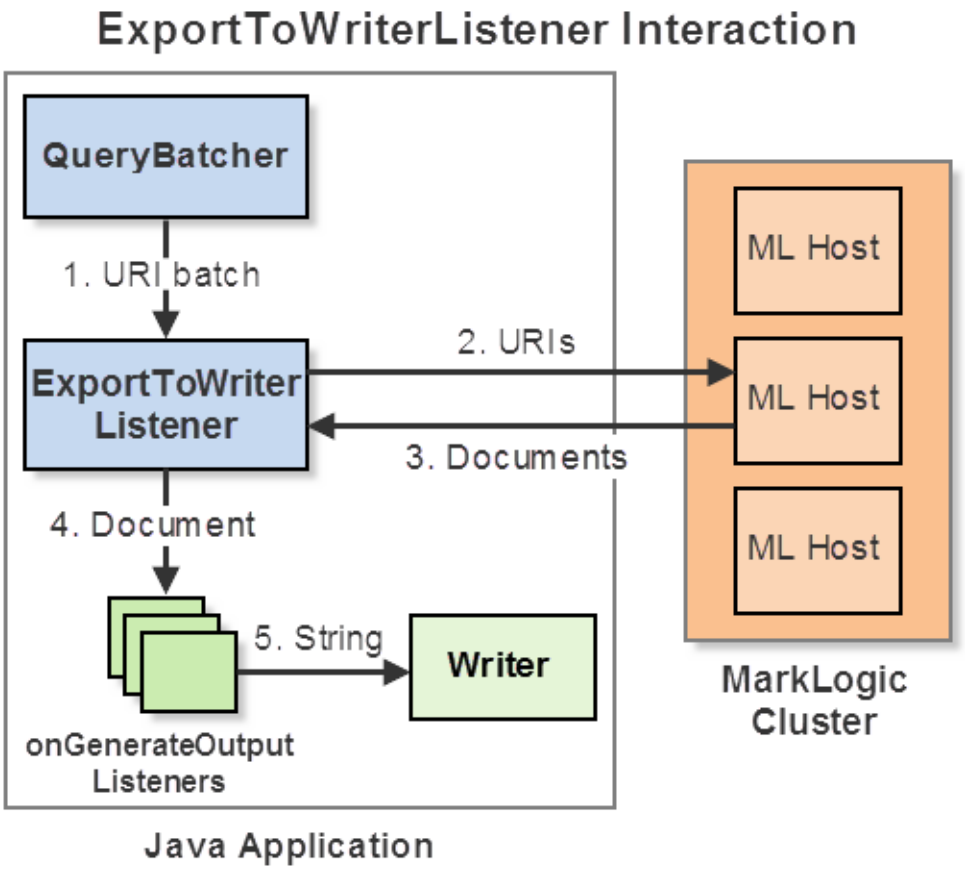
For a more complete example, see "Example: Exporting Documents that Match a Query" on page 122.

4.6.2 Using ExportToWriterListener to Read Documents

When you create an `ExportToWriterListener`, you must supply a `Writer` that will receive the documents read from MarkLogic. When an `ExportToWriterListener` receives a batch of URIs from a `QueryBatcher`, it reads these documents from MarkLogic, and then calls `Writer.write` on each document.

If sending the contents of each document to the writer as-is does not meet the needs of your application, you can register an output listener to prepare custom input for the writer. Use `ExportToWriterListener.onGenerateOutput` to register such a listener.

The following diagram illustrates the flow when you register an `onGenerateOutput` listener.



If you do not register an `onGenerateOutput` listener, then the flow in the above diagram skips Step 4. That is, the `ExportToWriterListener` sends content of each document directly to the `Writer`; metadata is ignored.

You can configure aspects of the `ExportToWriterListener` document read operation such as the following. For a complete list, refer to the *Java Client API Documentation*.

- Fetch metadata such as collections or properties, as well as document content. See `ExportToWriterListener.withMetadataCategory`. You should register an `onGenerateOutputListener` if you fetch metadata because the default flow with no listener ignores metadata.
- Use a consistent snapshot, fetching documents as they were when the query job started. See `ExportToWriterListener.withConsistenSnapshot` and “Using a Consistent Snapshot” on page 114.
- Apply a server-side read transform to each document before returning it to the client application. See `ExportToWriterListener.withTransform` and “Applying a Read or Write Transformation” on page 130.

- Prepend a string to the output sent to the `Writer` for each document. This prefix is included whether or not control flow goes through an `onGenerateOutputListener`. See `ExportToWriterListener.withRecordPrefix`.
- Append a string to the output sent to the `Writer` for each document. This suffix is included whether or not control flow goes through an `onGenerateOutputListener`. See `ExportToWriterListener.withRecordSuffix`.

The `ExportToWriterListener` uses the interfaces described in "Synchronous Multi-Document Operations" on page 70 to fetch the documents, so the listener blocks during the fetch. Each fetched document (and its metadata) is made available to the `onGenerateOutput` listeners as a `DocumentRecord`. This is the same interface used by the synchronous interfaces, such as the multi-document read shown in "Read Multiple Documents by URI" on page 83.

The following example creates an `ExportToWriterListener` that is configured to fetch documents and collection metadata. The `onGenerateOutput` listener generates a comma-separated string containing the document URI, first collection name, and the document content. `ExportToWriterListener.withRecordSuffix` is used to emit a newline after each document is processed. The end result is a three-column CSV file.

```
FileWriter writer = new FileWriter(outputFile));
ExportToWriterListener listener = new ExportToWriterListener(writer)
    .withRecordSuffix("\n")
    .withMetadataCategory(DocumentManager.Metadata.COLLECTIONS)
    .onGenerateOutput(
      record -> {
        try{
          String uri = record.getUri();
          String collection =
              record.getMetadata(new DocumentMetadataHandle())
                    .getCollections().iterator().next();
          String contents = record.getContentAs(String.class);
          return uri + "," + collection + "," + contents;
        } catch (Exception e) {
            e.printStackTrace();
        }
      }
    );
```

For the complete example, see `ExportToWriterListenerTest` in `com.marklogic.client.test.datamovement`. The test source is available on GitHub. For more details, see "Downloading the Library Source Code" on page 34.

4.6.3 Example: Exporting Documents that Match a Query

The following function uses `QueryBatcher` and `ExportListener` to read documents from MarkLogic and save them to the filesystem. The job uses a structured query to select the documents to be exported. Further explanation follows the code sample.

```
// Assume "client" is a previously created DatabaseClient object.
private static String EX_DIR = "/your/directory/here";
```

```
private static DataMovementManager dmm =
    client.newDataMovementManager();

// ...

public static void exportByQuery() {
    // Construct a directory query with which to drive the job.
    QueryManager qm = client.newQueryManager();
    StructuredQueryBuilder sqb = qm.newStructuredQueryBuilder();
    StructuredQueryDefinition query = sqb.directory(true, "/dmsdk/");

    // Create and configure the batcher
    QueryBatcher batcher = dmm.newQueryBatcher(query);
    batcher.onUrisReady(
        new ExportListener()
            .onDocumentReady(doc-> {
                String uriParts[] = doc.getUri().split("/");
                try {
                    Files.write(
                        Paths.get(EX_DIR, "output",
                                  uriParts[uriParts.length - 1]),
                        doc.getContent(
                            new StringHandle()).toBuffer());
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }))
           .onQueryFailure( exception -> exception.printStackTrace() );

    dmm.startJob(batcher);

    // Wait for the job to complete, and then stop it.
    batcher.awaitCompletion();
    dmm.stopJob(batcher);
}
```

The query driving the job is a simple directory query that matches all documents in the directory "/dmsdk/", such as the documents inserted in "Example: Loading Documents From the Filesystem" on page 108:

```
QueryManager qm = client.newQueryManager();
StructuredQueryBuilder sqb = qm.newStructuredQueryBuilder();
StructuredQueryDefinition query = sqb.directory(true, "/dmsdk/");
```

You can use any string, structured, or combined query. For details on query construction, see "Searching" on page 144.

The `ExportListener.onDocumentsReady` listener attached by the example writes each document to the filesystem, using the last path step in the URI as the file name. That is, if the document URI is `/dmsdk/doc1.xml`, then a file named `doc1.xml` is written to the output directory. The output directory is `EX_DIR/output/`, where `EX_DIR` is a variable holding the path of your choice.

```
new ExportListener()
    .onDocumentReady(doc-> {
        String uriParts[] = doc.getUri().split("/");
        try {
            Files.write(Paths.get(EX_DIR, "output",
                                  uriParts[uriParts.length - 1]),
                        doc.getContent(new StringHandle()).toBuffer());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }))
```

The `ExportListener.onQueryFailure` listener is just a lambda function that emits a stack trace. You would use a more sophisticated listener in a production application.

4.7 Applying an In-Database Transformation

You can use the Data Movement SDK to orchestrate in-place transformations of documents already in the database by using an `ApplyTransformListener` with a `QueryBatcher`. This section includes the following topics:

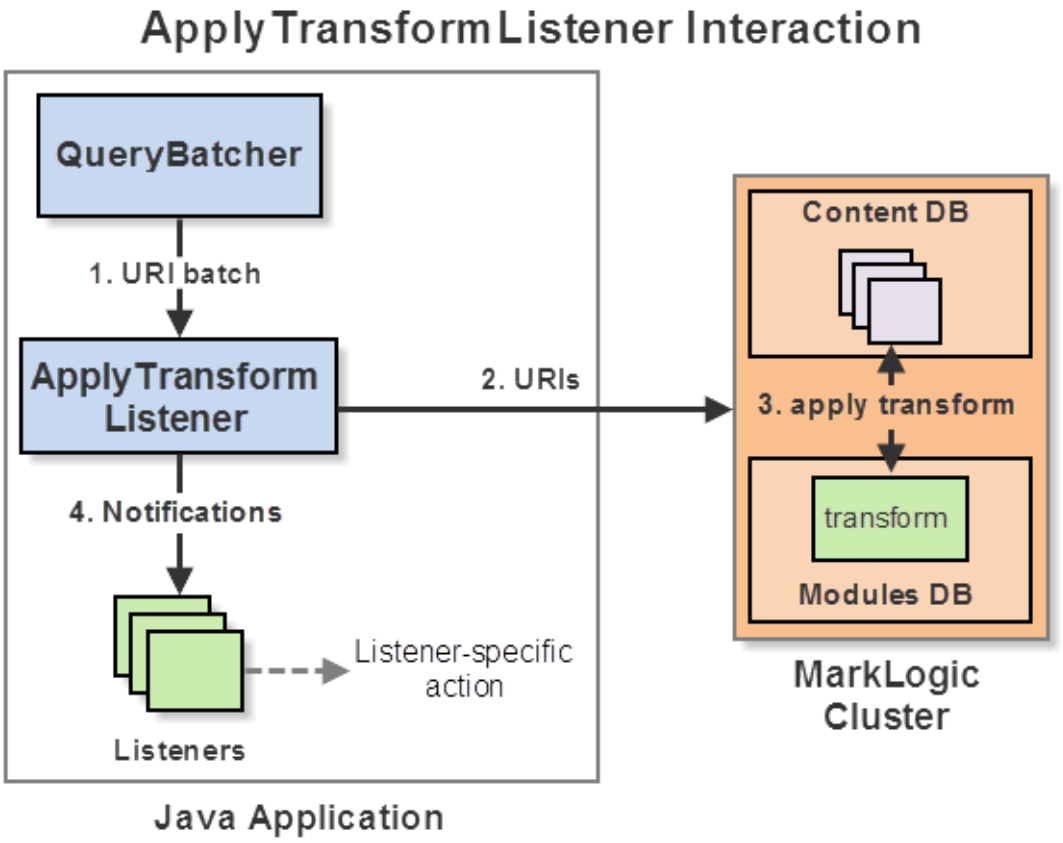
- Applying an In-Database Transformation with QueryBatcher
- Example: Applying an In-Database Transformation

4.7.1 Applying an In-Database Transformation with QueryBatcher

An in-database transformation is driven by a client-side query job, but carried out entirely inside MarkLogic, without fetching any documents to the client. Use a `QueryBatcher` with an `ApplyTransformListener` attached to the batcher's `onUrisReady` event. You could also create a custom transform listener.

This section assumes you are familiar with query job basics. If not, review "Creating and Managing a Query Job" on page 110.

The following diagram illustrates the default flow of query job that performs an in-database transformation.



By default, the output of the transform replaces the original document in MarkLogic.You can configure the listener to run the transform without updating the source document by calling `ApplyTransformListener.withApplyResult`. For example, you could use this approach if your "transform" computes an aggregate over the documents matching a query and stores the result elsewhere in the database.

The transform to be applied by the job must be installed on MarkLogic before you can use it. Data Movement SDK uses the same transform framework as the single document operations and synchronous multi-document operations. For details on authoring and installing a transform, see "Content Transformations" on page 282.

You identify the transform by supplying a `ServerTransform` object when you create the `ApplyTransformListener` for the job.

The following example applies a previously installed transformation to a set of URIs using a query job. The contents of the three target documents are replaced by the documents created by the transform function.

```
private static DataMovementManager dmm =
    client.newDataMovementManager();
// ...
public static void inPlaceTransform(String txName) {
    ServerTransform txform = new ServerTransform(txName);
    String uris[] =
        {"/dmsdk/doc1.xml", "/dmsdk/doc3.xml", "/dmsdk/doc5.xml"};
    QueryBatcher batcher =
        dmm.newQueryBatcher(Arrays.asList(uris).iterator());
    batcher.withConsistentSnapshot()
           .onUrisReady(
               new ApplyTransformListener().withTransform(txform))
          .onQueryFailure( exception -> exception.printStackTrace() );
    dmm.startJob(batcher);
    batcher.awaitCompletion();
    dmm.stopJob(batcher);
}
```

For a more complete example, see “Example: Applying an In-Database Transformation” on page 127.

All the transformed documents associated with a batch of URIs are committed as a single transaction, so if the transformation of any document fails, the whole batch fails. The absence of a targeted document in the database is not treated as an error and does not cause the batch to fail. Such documents are simply skipped.

> **Note:** If you use a query to select the documents to be transformed, then you should use `QueryBatcher.withConsistentSnapshot` with `ApplyTransformListener`. For details, see “Using a Consistent Snapshot” on page 114.

You can attach listeners to an `ApplyTransformListener` to receive notifications about batch success, batch failure, and skipped document events. These listeners use the `QueryBatchListener` interface. Use the following methods to attach listeners:

- `ApplyTransformListener.onSuccess`: Register a listener that is called whenever all the documents corresponding to a batch of URIs have been successfully transformed or skipped. The URIs of the batch of transformed documents are accessible through the registered listener’s `getItems` method.
- `ApplyTransformListener.onSkipped`: Register a listener that is called whenever one or more documents corresponding to a batch of URIs were not found in the database. The URIs of the missing documents are accessible through `getItems` method of the batch passed to the listener.

- `ApplyTransformListener.onBatchFailure`: Register a listener that is called whenever an entire batch of transformations is rejected due to an error transforming at least one document.

4.7.2 Example: Applying an In-Database Transformation

The example in this section applies an in-database XQuery transformation using `QueryBatcher` and `ApplyTransformListener`.

The following XQuery module implements a trivial transform that inserts a `<now/>` XML child element into the input document if the root element is `<data/>`. (This matches the document structure created by “Example: Loading Documents From the Filesystem” on page 108.) The element value is the current `xs:dateTime` when the transform is applied. For more details, see “Writing Transformations” on page 287.

```
xquery version "1.0-ml";
module namespace dmex =
"http://marklogic.com/rest-api/transform/dm-in-place";

(: Add an element named "now" that contains the current dateTime. :)
declare function dmex:transform(
  $context as map:map,
  $params as map:map,
  $content as document-node())
as document-node() {
  if (fn:empty($content/data)) then $content
  else document {
    let $root := $content/*
    return
      element {fn:name($root)} {
        element now { fn:current-dateTime() },
        $root/@*,
        $root/node()
      }
    }
};
```

The following server-side Javascript module implements a trivial transform that adds a property named `writeTimestamp` corresponding to the current dateTime to the document stored in the database. If the input document is not JSON, the content is unchanged.

```
function insertTimestamp(context, params, content)
{

  if (context.inputType.search('json') >= 0) {

    const result = content.toObject();

    result.writeTimestamp = fn.currentDateTime();

    return result;
```

```
  } else {

    /* Pass thru for non-JSON documents */

    return content;

  }

};


exports.transform = insertTimestamp;
```

If you any of the above codes to a file (`namefile.xqy` or `namefile.sjs`), you can install it on MarkLogic with code similar to the following. This function expects the transform name (which is subsequently used to identify the transform during operations), and the name of the file containing the code as input. It reads the file from *`EX_DIR`*`/ext/`*`txFilename`* and installs it under the specified name.

```
// Assume "client" is a previously created DatabaseClient object.
// The example also assumes the following context:
private static String EX_DIR = "/your/data/dir/here/";
private static DataMovementManager dmm =
    client.newDataMovementManager();

// Helper function for installing transformations.
public static void installTransform(String txName, String txFilename) {
    FileHandle txImpl = new FileHandle().with(
            Paths.get(EX_DIR, "ext", txFilename).toFile());
    TransformExtensionsManager txmgr =
        client.newServerConfigManager()
              .newTransformExtensionsManager();
    txmgr.writeXQueryTransform(txName, txImpl);
    // Or, if you use a servser-side JavaScript module
    txmgr.writeJavascriptTransform(txName, txImpl);
}
```

For more details, see “Installing Transforms” on page 282.

Assuming the transformation is installed, the following function creates a query job to apply it to a set of documents specified by a URI list. You could also apply it to documents matching a query.

```
public static void inPlaceTransform(String txName) {
    ServerTransform txform = new ServerTransform(txName);
    String uris[] = {
        "/dmsdk/doc1.xml", "/dmsdk/doc3.xml", "/dmsdk/doc5.xml"};
    QueryBatcher batcher =
        dmm.newQueryBatcher(Arrays.asList(uris).iterator());
    batcher.onUrisReady(
               new ApplyTransformListener().withTransform(txform))
```

```
            .onQueryFailure( exception -> exception.printStackTrace() );
    dmm.startJob(batcher);
    batcher.awaitCompletion();
    dmm.stopJob(batcher);
}
```

The example accepts the transform name as input and constructs a `ServerTransform` object from this name. The `ServerTransform` is required to configure the `ApplyTransformListener`. For more details, see “Using Transforms” on page 283.

```
ServerTransform txform = new ServerTransform(txName);
...
new ApplyTransformListener().withTransform(txform)
```

Whenever a batch of URIs is ready for processing, the `ApplyTransformListener` applies the transform to all the documents in the batch.

If the job was driven by a query rather than a list of URIs, you would include a call to `QueryBatcher.withConsistentSnapshot` in the job configuration. You should use a consistent snapshot when running query driven jobs that modify the database. For details, see “Using a Consistent Snapshot” on page 114.

4.8 Deleting Documents from a Database

You can use the Data Movement SDK to delete documents stored in MarkLogic by using a `DeleteListener` with a `QueryBatcher`. This section assumes you are familiar with query job basics. If not, review “Creating and Managing a Query Job” on page 110.

As with any query job, the target URIs are fetched to the client so that the delete operation can be distributed across the cluster. No documents are fetched to the client. You can select the documents to be deleted by specifying a query or supplying a list of URIs.

Note: A job that deletes documents alters the state of the database in a way that affects query results. If you use a query to select the documents for deletion, you should enable merge timestamps on the database and use a consistent snapshot. For more details, see “Using a Consistent Snapshot” on page 114.

All the deletions associated with a batch of URIs are committed as a single transaction, so if the deletion of any document fails, the whole batch fails. Note that the absence of a targeted document in the database is not treated as an error and does not cause the batch to fail.

The following example deletes all documents where the “data” element has a value of 5:

```
// Assume "client" is a previously created DatabaseClient object and
// "dmm" is a previously created DataMovementManager.
public static void deleteDocs() {
    QueryManager qm = client.newQueryManager();
    StructuredQueryBuilder sqb = qm.newStructuredQueryBuilder();
    StructuredQueryDefinition query = sqb.value(sqb.element("data"),5);
```

```
        QueryBatcher batcher = dmm.newQueryBatcher(query);
        batcher.withConsistentSnapshot()
               .onUrisReady(new DeleteListener())
               .onQueryFailure( exception -> exception.printStackTrace() );
        dmm.startJob(batcher);

        batcher.awaitCompletion();
        dmm.stopJob(batcher);
    }
```

4.9 Applying a Read or Write Transformation

You can apply a server-side transformation to documents when you insert them into MarkLogic with a write job. Similarly, you can apply a server-side transformation to documents when you read them from MarkLogic using a query job.

Applying a read or write transformation uses the same framework as an in-database transformation (and other Java Client API document operations), but the flow is different. A write transform is applied to content received from the client; this content may not already be present in the database. A read transform is applied to content just before it is returned to the client, leaving the document in the database unchanged.

You must install a transformation in MarkLogic before you can use it in a job. Other Java Client API document operations use the same transformation framework, including single document operations and synchronous multi-document operations. Authoring and installation of transformations are discussed in “Content Transformations” on page 282.

Configure a write transformation using `WriteBatcher.withTransform`. Supply a `ServerTransform` object that represents a previously installed transformation. When creating the `ServerTransform`, you must use the name under which the transform is installed on MarkLogic.

The following code snippet configures a `WriteBatcher` with a write transform.

```
DataMovementManager dmm = ...;

WriteBatcher batcher = dmm.newWriteBatcher();
batcher.withBatchSize(5)
       .withThreadCount(3)
       .withTransform(new ServerTransform(txName))
       // ...additional configuration
       ;
```

For a query job, the listener determines whether or not to support a transform because the action performed by the job is determined by the listener. For example, `ExportListener` and `ExportToWriterListener` both have a `withTransform` method through which you can specify a server-side read transform. However, a transform makes no sense in the context of a `DeleteListener`, so it has no such method.

The following code snippet configures an ExportListener with a read transform.

```
DataMovementManager dmm = ...;

QueryBatcher batcher = dmm.newQueryBatcher(query);
batcher.onUrisReady(
          new ExportListener()
            .withTransform(new ServerTransform(txName))
            .onDocumentReady(...))
       .onQueryFailure(...);
```

4.10 Job Control

- Checking the Status of a Job
- Pausing and Restarting a Job
- Graceful Termination of a Job
- Terminating a Job Prematurely
- Updating Forest Configuration for a Job
- Working with a Load Balancer
- Restricting the Hosts Used by a Job

4.10.1 Checking the Status of a Job

When you start a job, you receive a `JobTicket`. You can use the `JobTicket` to retrieve the type and id of a job, and to get a job report (using `DataMovementManager.getJobReport`). The job report provides statistics such as the number of successfully processed batches. The meaning of the statistics depends on the type of job; refer to the javadoc for `JobReport` for details.

The following code snippet retrieves a job report from the ticket for a write job:

```
DataMovementManager dmm = ...;
WriterBatcher batcher = dmm.newWriteBatcher();
//...
JobTicket ticket = dmm.startJob(batcher);
//...
JobReport report = dmm.getJobReport(ticket);
```

You can also retrieve batch-level information about a job within a listener. For example, a `WriteBatcher.onBatchSuccessListener` can call `WriteBatch.getJobWritesSoFar`.

A `JobReport` gathers its statistics by querying listeners that are automatically attached to query and write job batchers. For example, a `WriteJobReportListener` is automatically attached to the `onBatchSuccess` and `onBatchFailure` events when you create a `WriteBatcher`.

You can use the implementation of these listeners as a starting point for your own reporting, and even replace the default reporting listeners with your own. For more information on replacing listeners, see “Working With Listeners” on page 140.

4.10.2 Pausing and Restarting a Job

The Data Movement SDK does not support restarting jobs. Once you call `DataMovementManager.stopJob`, you cannot perform additional work with the job.

You can effectively mimic pausing and restarting a write job by controlling the flow of work into the job. For example, the following steps “pause” and “restart” a write job:

1. Stop any activity that calls `WriteBatcher.add` or `WriteBatcher.addAs`.

2. Call `WriteBatcher.flushAndWait` or `WriteBatcher.flushAsync`. This ensure any partial batch is processed and in-progress batches get completed.

3. When you’re ready to resume work, start calling `WriteBatcher.add` and `WriteBatcher.addAs` again.

After Step 2, above, the job is effectively paused since it has finished all available work and new work is not arriving.

A query job always runs until all URIs are processed unless you shut it down prematurely. However, you can effectively pause a query job by blocking the listener(s). For example, you could create a listener that conditionally blocks on an object by calling `Object.wait`. For a timed pause, pass a timeout period to `wait`. You can use `Object.notifyAll` to reactivate all listeners and resume processing.

4.10.3 Graceful Termination of a Job

Graceful termination means shutting down a job in a way that leaves it in a deterministic state. For example, if you were to abruptly terminate a write job, some queued documents might not be written to the database.

Graceful termination usually means draining the work queue of a job before calling `DataMovementManager.stopJob`. These steps differ between write jobs and query jobs. For details on shutting down each type of job, see the following topics:

- “Stopping a Write Job” on page 105
- “Stopping a Query Job” on page 113

A job cannot be restarted after calling `DataMovementManager.stopJob`.

4.10.4 Terminating a Job Prematurely

If you need to stop a job without waiting for work to be completed, you can call `DataMovementManager.stopJob` without first calling methods that drain the work queue like `WriteBatcher.flushAndWait` or `QueryBatcher.awaitCompletion`.

If you do not follow the graceful shutdown procedure, you cannot guarantee that queued work will be started or in-progress work will be completed after calling `stopJob`. Any work that started prior to calling `stopJob` will be allowed to complete as long as the JVM does not exit.

For example, if documents have been added to a write job, but a full batch has not yet accumulated, the partial batch will not be processed.

For details on shutting down each type of job, see the following topics:

- “Stopping a Write Job” on page 105
- “Stopping a Query Job” on page 113

A job cannot be restarted after calling `DataMovementManager.stopJob`.

4.10.5 Updating Forest Configuration for a Job

This section describes how to update a batcher’s understanding of which hosts in a cluster include forests for the database on which the job operates. You are unlikely to need to do this unless you have a very long running job and change your cluster topology.

As mentioned in “How Work is Distributed Across a Cluster” on page 101, when you create a batcher, the `DataMovementManager` initializes the batcher with information about which hosts in your MarkLogic cluster contain forests in the database targeted by the job. The batcher uses this forest configuration information to determine how to distribute work across the cluster.

If you change the database forest locations in such a way that this list of forest hosts becomes inaccurate, the batcher will not be aware of the change. For example, if you add a forest to a host that previously contained no forests for the database, the batcher will not direct work to the new host.

To refresh a batcher’s forest model, pass the output of `DataMovementManager.readForestConfiguration` to `Batcher.withForestConfig`. When you call `DataMovementManager.readForestConfig()`, the `DataMovementManager` queries the cluster for the current forest configuration and returns the new configuration. For example:

```
DataMovementManager dmm = ...;
WriteBatcher batcher = ...;
dmm.startJob(batcher);

// some time later...
batcher.withForestConfig(dmm.readForestConfig());
```

4.10.6 Working with a Load Balancer

By default, a job tries to connect directly to multiple hosts in your cluster in order to efficiently distribute work. However, if there is a load balancer sitting between your client application and your MarkLogic cluster, these direct connections may not be possible.

In such a case, you must configure your `DatabaseClient` objects to specify a `GATEWAY` connection, instead of the default `DIRECT` connection. For example:

```
DatabaseClient client =
  DatabaseClientFactory.newClient(
    "localhost", 8000, "MyDatabase",
    new DatabaseClientFactory.DigestAuthContext("myuser", "mypassword"),
    DatabaseClient.ConnectionType.GATEWAY);
```

You cannot use a `FilteredForestConfiguration` with a `GATEWAY` connection since all traffic will be routed through the gateway.

You should configure your load balancer timeout periods to be consistent with your MarkLogic cluster timeouts. For more details, see “Connecting Through a Load Balancer” on page 19.

For details on failover handling, see “Failover When Connecting Through a Load Balancer” on page 136.

4.10.7 Restricting the Hosts Used by a Job

By default, a job tries to connect to all hosts in your cluster that contain forests in the database. This optimizes the performance of your job. However, if you need to restrict host list for a reason *other than connecting through a load balancer*, you can use `FilteredForestConfiguration` to configure that list.

> **Note:** If you connect to MarkLogic through a load balancer, see “Working with a Load Balancer” on page 134, instead of using `FilteredForestConfiguration`.

You can configure a white list (hosts allowed) or a black list (host disallowed). The Java Client API uses the same mechanism internally to manage failover.

The following example restricts a job to connecting to MarkLogic through only the hosts “good-host-1” and “good-host-2”:

```
// Assume "dmm" is a previously created DataMovementManager object.
batcher.withForestConfig(
  new FilteredForestConfiguration(
    dmm.readForestConfig()
  ).withWhiteList("good-host-1", "good-host-2")
);
```

Note that limiting a job to connect to a restricted host list can negatively impact the performance of your job.

4.11 Failover Handling

Failover occurs when a forest or a host in a cluster becomes unavailable due to events such as a forest restart or a host becoming unreachable. The unavailable host might become available again or be replaced by a failover host that is configured for the database as described in High Availability of Data Nodes With Failover in the *Scalability, Availability, and Failover Guide*. The Data Movement SDK attempts to recover from such events with no data loss.

This section covers the following topics:

- Default Failover Handler
- Failover When Connecting Through a Load Balancer
- Interaction with In-Database Transform
- Failover Handling in Custom Listeners

4.11.1 Default Failover Handler

The Data Movement SDK provides a default error handling listener, `HostAvailabilityListener`, for managing failover events. Whenever you create a `QueryBatcher` or a `WriteBatcher` object, a `HostAvailabilityListener` is attached to it. You can also use `HostAvailabilityListener` as an example for creating your own failover handler.

> **Note:** This discussion applies when you connect directly to MarkLogic. If you connect through a load balancer, see "Failover When Connecting Through a Load Balancer" on page 136.

When the `HostAvailabilityListener` detects an unavailable host, the Data Movement SDK responds as follows:

1. Check to see if the configured minimum number of hosts remain in the forest configuration (minus the failed host). If not, stop the job with an error. If so, proceed with the recovery procedure.

2. To avoid repeated occurrences of the same error, remove the failed host from the forest configuration on which the job operates. The failed host is considered "suspended" for a configurable time period and will not be used by the job while in this state.

3. Schedule an asynchronous task to re-acquire the forest configuration from MarkLogic when the suspension time period expires. This enables the failed host to come back into rotation or be replaced by a failover host.

4. Retry the failed batch with one of the hosts remaining in the forest configuration modified in Step 2.

Use `HostAvailabilityListener.withSuspendTimeForHostUnavailable` to configure the suspension time period. The default suspension period is 10 minutes.

Use `HostAvailabilityListener.withMinHosts` to configure the minimum number of host required to enable retrying a failed batch.

Use `HostAvailabilityListener.withHostUnavailableExceptions` to configure the exceptions that trigger the retry flow. By default, `HostAvailabilityListener` acts on the following exceptions classes: `SocketException`, `SSLException`, `UnknownHostException`.

For example, the following code configures the default `HostAvailability` listener attached to a batcher with a suspension period of 5 minutes and a two host minimum:

```
HostAvailabilityListener.getInstance(batcher)
    .withSuspendTimeForHostUnavailable(Duration.ofMinutes(5))
    .withMinHosts(2);
```

If the behavior of `HostAvailabilityListener` does not meet the needs of your application, you can use it as a basis for developing your own failover handler. To review the implementation on GitHub or download a local copy of the source code, see "Downloading the Library Source Code" on page 34.

4.11.2 Failover When Connecting Through a Load Balancer

When you connect to MarkLogic through a load balancer, you must configure your DatabaseClient objects to use a `GATEWAY` connection, as described in "Working with a Load Balancer" on page 134.

When you use a `GATEWAY` connection, all traffic goes through the load balancer host, so it is not possible for the job to modify its host list if a host in your MarkLogic cluster becomes unavailable, as described in "Default Failover Handler" on page 135.

Instead, `HostAvailabilityListener` retries against the load balancer for some time. When the MarkLogic cluster successfully recovers from the host failure, batches submitted through the load balancer start succeeding again.

If the MarkLogic cluster is not able to recover within the timeout period, then the job fails. If the load balancer host becomes unavailable, your job is cancelled.

4.11.3 Interaction with In-Database Transform

When you attach an `ApplyTransformListener` to a `QueryBatcher`, the retry mechanism described in "Default Failover Handler" on page 135 applies only to the process of fetching batches of URIs from MarkLogic by default because the Java Client API cannot assume it is safe to retry the intended in-database transformation or deletion.

If a failover event occurs while fetching a batch of URIs, `HostAvailabilityListener` retries the failed URI fetch, just as it does when handling failovers for reading and writing documents. If a failover event occurs after a batch of URIs is dispatched to an attached `onUrisReady` listener such as an `ApplyTransformListener`, the batch will fail by default if a failover event occurs.

To handle this more complex situation, the Java Client API supports the following types of listener for failover handling:

- `HostAvailabilityListener`: If a failover event occurs while fetching a batch of URIs, `HostAvailabilityListener` retries the failed URI fetch, just as it does when handling failovers for reading and writing documents.
- `NoResponseListener`: Handles the case where no response is received from MarkLogic. The default `NoResponseListener` handles the case where no response is received while fetching URIs. This listener is register by default for all listeners created by the Java Client API.
- `BatchFailureListener<QueryBatch>` for `HostAvailabilityListener`: Implements the retry logic when a qualifying exception is raised while fetching URIs. Such a retry listener is associated with all listeners created by the Java Client API, including `ApplyTransformListener`.
- `BatchFailureListener<QueryBatch> for NoResponseListener`: Implements the retry logic when no response is received from MarkLogic during the transform operation. The Java Client API adds this listener to listeners for idempotent operations, such as DeleteListener. It is not added to `ApplyTransformListener` by default

If you know that your transform is idempotent and can safely be repeated, then you can enable failover handling for the no response case by attaching a retry listener to the `NoResponseListener`. For example:

```
ApplyTransformListener txformListener = new ApplyTransformListener()
  .withTransform(txform);
QueryBatcher batcher = ...;

NoResponseListener noResponseListener =
  NoResponseListener.getInstance(batcher);
if (noResponseListener != null) {
  BatchFailureListener<QueryBatch> retryListener =
    noResponseListener.intializeRetryListener(txformListener);
  if (retryListener != null) {
    txformListener.onFailure(retryListener);
  }
}
```

If your in-database transform is not idempotent, but you want to retry in some no-response cases, you implement your own BatchFailureListener. For details, see “Conditionally Retry” on page 139.

4.11.4 Failover Handling in Custom Listeners

This section describes how to implement failover handling in a custom listener. Your listener can respond to failover events in the following ways:

- Never retry. Allow the batch to fail. You do not need to write any special code to address this case. This is the default behavior of `ApplyTransformListener`.
- Always Retry. If the operation performed by the listener is idempotent, such as document write or delete, then you can always safely retry. `DeleteListener` implements this approach.
- Conditionally Retry. You must implement a custom `BatchFailureListener` for this case.

4.11.4.1 Always Retry

If you create a custom `QueryBatchListener` that should always retry on a qualifying error, override the `initializeListener` method to do the following:

1. Obtain the `HostAvailabilityListener` from the batcher.

```
HostAvailabilityListener hostAvailabilityListener =
    HostAvailabilityListener.getInstance(queryBatcher);
```

2. Obtain a `RetryListener` by calling `HostAvailabilityListener.intializeRetryListener`.

```
BatchFailureListener<QueryBatch> retryListener =
    hostAvailabilityListener.initializeRetryListener(this);
```

3. Register the `RetryListener` as an `onFailureListener` of your custom listener.

```
if ( retryListener != null )  onFailure(retryListener);
```

4. Obtain a `NoResponseListener` from the batcher.

```
NoResponseListener noResponseListener =
    NoResponseListener.getInstance(queryBatcher);
```

5. Obtain a `RetryListener` by calling `NoResponseListener.initializeRetryListener`.

```
BatchFailureListener<QueryBatch> noResponseRetryListener =
    noResponseListener.initializeRetryListener(this);
```

6. Register the `RetryListener` as an `onFailure` listener of your custom listener.

```
if ( noResponseRetryListener != null )
    onFailure(noResponseRetryListener);
```

The `RetryListener` for the `noResponseListener` is required to handle cases where a host becomes unavailable without returning any response from MarkLogic, rather than raising an error.

The following code puts these steps together into an implementation of `initializeListener` for a custom query batch listener:

```
public class myListener : extends Object implements QueryBatchListener
{

  // ...

  @Override
  public void initializeListener(QueryBatcher queryBatcher) {
    HostAvailabilityListener hostAvailabilityListener =
      HostAvailabilityListener.getInstance(queryBatcher);
    if ( hostAvailabilityListener != null ) {
      BatchFailureListener<QueryBatch> retryListener =
        hostAvailabilityListener.initializeRetryListener(this);
      if ( retryListener != null )  onFailure(retryListener);
    }
    NoResponseListener noResponseListener =
      NoResponseListener.getInstance(queryBatcher);
    if ( noResponseListener != null ) {
      BatchFailureListener<QueryBatch> noResponseRetryListener =
        noResponseListener.initializeRetryListener(this);
      if ( noResponseRetryListener != null )
        onFailure(noResponseRetryListener);
    }
  }

};
```

See the implementation of `com.marklogic.client.datamovement.DeleteListener` for a complete example. To review the implementation on GitHub or download a local copy of the source code, see “Downloading the Library Source Code” on page 34.

4.11.4.2 Conditionally Retry

If you only want to retry your operation under certain circumstances, do the following:

- Create a class that implements `BatchFailureListener<QueryBatch>`. Implement your retry logic in the `processFailure` method.
- Attach an instance of your `BatchFailureListener` as an `onFailure` listener of your custom listener.

To initiate a retry from your batch failure listener, invoke `QueryBatcher.retry`. This enables a retry if an error occurs when fetching URIs. For example:

```
public void processFailure(QueryBatch batch, Throwable throwable) {
  // ...
  batch.getBatcher().retry(batch);
  // ...
}
```

To create a custom availability listener, override `QueryBatchListener.intializeListener`. The default implementation of this method does nothing. Your implementation should be similar to the following:

```
@Override
public void initializeListener(QueryBatcher queryBatcher) {
    HostAvailabilityListener hostAvailabilityListener =
        HostAvailabilityListener.getInstance(queryBatcher);
    if ( hostAvailabilityListener != null ) {
      BatchFailureListener<QueryBatch> retryListener =
          hostAvailabilityListener.initializeRetryListener(this);
      if( retryListener != null ) onFailure(retryListener);
    }
}
```

The batcher calls the `initializeListener` method on each attached `QueryBatchListener`.

The retry listener should call `QueryBatchListener.retryListener` in its `processFailure` method to re-attempt the failed operation. That is, to retry in cases where a batch of URIs is successfully retrieved from MarkLogic, but a failure occurs during the in-database operation. For an example, see the implementation of `HostAvailabilityListener.RetryListener.processFailure`.

4.12 Working With Listeners

A listener is a callback through which your application responds to interesting job state changes, such as when a write job successfully inserts a batch of documents, or a query job prepares a batch of URIs for processing.

This section covers the following listener-related topics:

- Guidelines for Creating Listeners
- Attaching Multiple Listeners to a Job
- Removing or Replacing a Listener

4.12.1 Guidelines for Creating Listeners

Data Movement SDK pre-defines several listener classes that are fully functional, but also meant to serve as a starting place for you to implement your own listeners.

For example, Data Movement SDK includes an `ExportToWriterListener` class for reading documents from the database and sending the contents as a string to a `Writer`. You might create a custom listener that also emits metadata, or one that generates zip file entries instead of strings.

When creating your own listeners, keep the following points in mind:

- All listener code must be thread safe because listeners are executed asynchronously across all job threads. For example, you should not have multiple listeners updating a shared

collection unless the collection is thread safe (`Collections.synchronizedMap<T>`, `Collections.synchronizedList<T>`, `ConcurrentHashMap`, `ConcurrentLinkedQueue`, etc.).

- In query jobs driven by a query (rather than a fixed set of URIs), each `QueryBatchListener` has access to the host and forest that contain the documents identified by a URI batch. Your job will be more efficient if you use the same host for your per batch operations. See `QueryBatch.getClient` and `QueryBatch.getForest`.

The thread safety requirement also applies to "listener listeners". For example, if you attach document ready event listeners to an `ExportListener` (`ExportListener.onDocumentReady`) that code must also be thread safe.

4.12.2 Attaching Multiple Listeners to a Job

You can attach listeners to multiple events, and you can attach multiple listeners to a single event. When there are multiple listeners for an event, they are invoked serially, in the order in which they were attached to the job. An event is not complete until all listeners complete their processing.

For example, when you create a `WriteBatcher`, the `DataMovementManager` automatically attaches a `WriteJobReportListener` to the batch success event. When you attach your own batch success or failure event listeners using `WriteBatcher.onBatchSuccess`, it doesn't replace the `WriteJobReportListener`. Rather, the batch success event now has multiple listeners.

You can probe the listeners attached to a job using methods such as `WriteBatcher.getBatchSuccessListeners` and `QueryBatcher.getQueryFailureListeners`.

4.12.3 Removing or Replacing a Listener

You can add a listener to a batcher using the appropriate `on`*`Event`* method, such as `WriteBatcher.onBatchSuccess`. You should not attach a listener to a running job.

To remove or replace a listener, you must retrieve the list of listeners attached to an event, modify the list, and set the listener list on the batcher to the value of the new list.

Note that the Data Movement SDK attaches a default set of listeners to `WriteBatcher` and `QueryBatcher` in support of job reporting, error recovery, and job management. If you replace or remove the entire set of listeners attached to an event, you will lose these automatically attached listeners.

The `WriteBatcher` and `QueryBatcher` interfaces include setters and getters for their respective event listener lists. For example, the `QueryBatcher` interface includes `getUrisReadyListeners` and `getQueryFailureListeners` methods.

The listener classes provided by Data Movement SDK, such as `ExportListener`, do not expose any kind of listener id. You can only distinguish them on the listener list by probing the type.

The following code snippet demonstrates removing a custom batch success listener from a `WriteBatcher`.

```
WriteBatchListener oldListeners[] =
    batcher.getBatchSuccessListeners();
ArrayList<WriteBatchListener> newListeners =
    new ArrayList<WriteBatchListener>();
for (WriteBatchListener listener : oldListeners) {
    if (!(listener instanceof MyWriteBatchListener)) {
        newListeners.add(listener);
    }
}
batcher.setBatchSuccessListeners(
    Stream.of(batcher.getBatchSuccessListeners())
      .filter(listener -> !(listener instanceof MyWriteBatchListener))
      .toArray(WriteFailureListener[]::new)
);
```

4.13 Alternative Interfaces

If your application is not working with large workloads or does not require an asynchronous interface, consider using the interfaces described in the following sections:

- "Single Document Operations" on page 36. Synchronous document operations on one document at a time. You can create, read, update and delete documents.
- "Synchronous Multi-Document Operations" on page 70. Synchronous document operations on multiple documents. You can create, read, update, and delete documents. You might find this interface simpler if you do not require asynchrony or the level of control provided by the Data Movement SDK.

If you want to move data into, out of, or between MarkLogic clusters using the command line, consider the mlcp tool. This tool provides many of the capabilities and performance characteristics of the Data Movement interfaces. For details, see the *mlcp User Guide*.

5.0 Searching

This chapter describes how to submit searches using the Java API, and includes the following sections:

5.1 Overview of Search Using the Java API

The MarkLogic Java API provides the following fundamental ways of querying the database:

- Searches on documents, which return search results, snippets, and facets.
- Value or Tuple (co-occurrences) searches, which return data from range indexes and the results of aggregate functions (including user-defined aggregate functions) from range indexes.

In addition to typical document searches, you can search Java POJOs that have been stored in the database. For details, see “POJO Data Binding Interface” on page 226.

When you search documents you can express search criteria using one of the following kinds of query:

- String query: Use a Google-style query string to search documents and metadata. For details, see “Search Using String Query Definition” on page 146.
- Query By Example: Search documents by constructing a query that directly models the structure of the documents you want to match. For details, see “Prototype a Query Using Query By Example” on page 156.
- Structured query: A simple and easy way to construct queries as a Java, XML, or JSON structure, allowing you to manipulate complex queries (such as geospatial polygons) in

the Java client. For details, see “Search Documents Using Structured Query Definition” on page 147

- Combined query: Combine a string or structured query with dynamic query options. For details, see “Apply Dynamic Query Options to Document Searches” on page 159.

When you query aggregate range indexes, you express your search criteria using a values query.

All search methods can also use persistent query options. *Persistent query options* are stored on the REST Server and referenced by name in future queries. Once created and persisted, you can apply query options to multiple searches, or even set to be the default options for all searches. Note that in XQuery, query option configurations are called *options nodes*.

Some search methods support dynamic query options that you specify at search time. A combined query allows you to bundle a string and/or structured query with dynamic query options to further customize a search on a per search basis. You can also specify persistent query options with a combined query search. The search automatically merges the persistent (or default) query options and the dynamic query options together. For details, see “Apply Dynamic Query Options to Document Searches” on page 159.

Query options can be very simple or very complex. If you accept the defaults, for example, there is no need to specify explicit query options. You can also make them as complex as is needed.

For details on how to create and work with query option configurations, see “Query Options” on page 190. For details on individual query options and their values, see Appendix: Query Options Reference in the *Search Developer’s Guide*. For more information on search concepts, see the *Search Developer’s Guide*.

In the examples in this chapter, assume a `DatabaseClient` called `client` has already been defined.

5.2 Using SearchHandle to Examine Query Results

Usually, you will use a `SearchHandle` object to contain your query results. The exact nature of results varies, depending on both the handle’s configuration and what query options and values were used for the search operation.

You can specify snippets to return in various ways. By default, they return as Java objects. But for custom or raw snippets, they are returned as DOM documents by using the `forceDOM` flag.

There are several ways to access different parts of the search result or control search results from a `SearchHandle`.

- The `getMatchResults()` method returns an array of `MatchDocumentSummary` objects of the matched documents, from which you can further extract for each result its match locations, path, metadata, an array of snippets, fitness, confidence measure, and URI. For details, see the `MatchDocumentSummary` entry in Java API JavaDoc.

- `getMetrics()` returns a `SearchMetrics` object containing various timing metrics about the search.
- `getFacetNames()`, `getFacetResult(name)`, `getFacetResults()` return, respectively, a list of returned facet names, the specified named facet result, and an array of facet results for this search.
- `getTotalResults()` returns an estimate of the number of results from the search.
- `setForceDOM(boolean)` sets the force DOM flag, which if `true` causes snippets to always be returned as DOM documents.

See the Java API JavaDoc for SearchHandle for the full interface.

The following is a typical programming technique for accessing search results using a search handle:

```
// iterate over MatchDOcumentSummary array locations, getting
// the snippet text for each location (you would then do something
// with the snippet text)
MatchDocumentSummary[] summaries = results.getMatchResults();
for (MatchDocumentSummary summary : summaries ) {
    MatchLocation[] locations = summary.getMatchLocations();
    for (MatchLocation location : locations) {
        location.getAllSnippetText();
      // do something with the snippet text
    }
}
```

5.3 Search Using String Query Definition

The MarkLogic Server Search API lets you do searches on string arguments, including the usual search operators such as AND and OR. For example, you could search on “Batman”, “Batman AND Robin”, “Batman OR Robin”, etc. For details, see Search Grammar in the *Search Developer’s Guide*.

1. Instantiate a `QueryManager`. The manager deals with interaction between the client and the database.

   ```
   QueryManager queryMgr = client.newQueryManager();
   ```

2. Instantiate a `StringQueryDefinition` object. Use `StringQueryDefinition.setCriteria()` to specify your search string.

   ```
   StringQueryDefinition qd = queryMgr.newStringDefinition();

   qd.setCriteria("Batman AND Robin");
   ```

3. Run a search with the `StringQueryDefinition` object as an argument, returning a `SearchHandle` object or an XML or JSON handle to get the search results in either of those formats:

```
SearchHandle results = queryMgr.search(qd, new SearchHandle());
DOMHandle results = queryMgr.search(qd, new DOMHandle());
JacksonHandle results = queryMgr.search(qd, new JacksonHandle());
```

4. Process and/or display the results using the handle.

5.4 Search Documents Using Structured Query Definition

Structured queries let you construct and modify complex queries in Java, XML, or JSON. For details, see Searching Using Structured Queries in the *Search Developer's Guide*. This section includes the following parts:

- Ways to Create a Structured Query
- Basic Steps to Define a Structured Query Definition
- Creating a Structured Query From Raw XML or JSON
- Structured Query Examples

5.4.1 Ways to Create a Structured Query

You can create a structured query in XML, in JSON, or using the `StructuredQueryBuilder` or `PojoQueryBuilder` interfaces in the Java API.

To specify a structured query directly in XML or JSON, use `RawStructuredQueryDefinition`; for details, see "Creating a Structured Query From Raw XML or JSON" on page 148. If you construct a structured query directly, it is up to you to make sure the query is constructed correctly. Incorrectly constructed queries can result in syntax errors, a query that does not do what you expect, or other exceptions. For syntax details, see Searching Using Structured Queries in the *Search Developer's Guide*.

The `StructuredQueryBuilder` interface in the Java API enables you build out a structured query one piece at a time in Java. The `PojoQueryBuilder` interface is similar, but you use it specifically for searching persistent POJOs; for details see "Searching POJOs in the Database" on page 232.

5.4.2 Basic Steps to Define a Structured Query Definition

The following are the basic steps needed to define a structured query definition in the Java API. This procedure creates a structured query definition using `StructuredQueryBuilder`. You can also create one directly in XML/JSON; for details, see "Creating a Structured Query From Raw XML or JSON" on page 148.

1. Instantiate a `QueryManager`. The manager deals with interaction between the client and the database.

```
QueryManager queryMgr = client.newQueryManager();
```

2. Instantiate a `StructuredQueryBuilder`, optionally passing in the name of persistent query options to use with your search.

```
StructuredQueryBuilder qb = new StructuredQueryBuilder(OPTIONS_NAME);
```

3. Use the query builder to create a `StructuredQueryDefinition` object with the desired search criteria.

```
StructuredQueryDefinition querydef =
    qb.and(qb.term("neighborhood"),
           qb.valueConstraint("industry", "Real Estate"));
```

4. Run a search with the `StringQueryDefinition` object as an argument, returning a result handle:

```
SearchHandle results = queryMgr.search(querydef, new SearchHandle());
```

5.4.3 Creating a Structured Query From Raw XML or JSON

To create a structured query from a raw XML or JSON representation, use any handle class that implements `com.marklogic.client.io.marker.StructureWriteHandle`.

The Java API includes `StructureWriteHandle` implementations that support creating a structure in XML or JSON from a string (`StringHandle`), a file (`FileHandle`), a stream (`InputStreamHandle`), and popular abstractions (`DOMHandle`, `DOM4JHandle`, `JDOMHandle`). For a complete list of implementations, see the Java API JavaDoc.

Follow this procedure to create a structured query using a handle:

1. Instantiate a `QueryManager`. The manager deals with interaction between the client and the database.

```
QueryManager queryMgr = client.newQueryManager();
```

2. Create a JSON or XML representation of the query, using a text editor or other tool or library. Use the syntax detailed in Searching Using Structured Queries in the *Search Developer's Guide*. The following example uses `String` for the raw representation:

```
String rawXMLQuery =
    "<search:query "+
          "xmlns:search='http://marklogic.com/appservices/search'>"+
      "<search:term-query>"+
          "<search:text>neighborhoods</search:text>"+
      "</search:term-query>"+
      "<search:value-constraint-query>"+
          "<search:constraint-name>industry</search:constraint-name>"+
          "<search:text>Real Estate</search:text>"+
      "</search:value-constraint-query>"+
    "</search:query>";

String rawJSONQuery =
        "{\"query\": {" +
                "   \"term-query\": {" +
```

```
                    "      \"text\": \"neighborhoods\"" +
                    "   }," +
                    "   \"value-constraint-query\": {" +
                    "      \"constraint-name\": \"industry\"," +
                    "      \"text\": \"Real Estate\"" +
                    "   }" +
                    "}" +
          "}";
```

3. Create a handle on your raw query using a class that that implements `StructureWriteHandle`. Set the handle content format appropriately. For example:

```
// For an XML query
StringHandle rawHandle =
    new StringHandle(rawXMLQuery).withFormat(Format.XML);

// For a JSON query
StringHandle rawHandle =
    new StringHandle(rawJSONQuery).withFormat(Format.JSON);
```

4. Create a `RawStructuredQueryDefinition` from the handle. Optionally, include the name of persistent query options. For example:

```
// Use the default persistent query options
RawStructuredQueryDefinition querydef =
    queryMgr.newRawStructuredQueryDefinition(rawHandle);

// Use the persistent options previously saved as "myoptions"
RawStructuredQueryDefinition querydef =
    queryMgr.newRawStructuredQueryDefinition(rawHandle, "myoptions");
```

5. Perform a search using the `RawStructuredQueryDefinition` and a results handle.

```
SearchHandle resultsHandle =
    queryMgr.search(querydef, new SearchHandle());
```

5.4.4 Structured Query Examples

This section shows some structured query examples, showing the XML for a structured query and the corresponding Java code using `StructuredQueryBuilder`. You can put each of these examples in context by inserting the `StructuredQueryDefinition` line in the following code:

```
QueryManager queryMgr = dbClient.newQueryManager();
StructuredQueryBuilder sb =
   queryMgr.newStructuredQueryBuilder("myopt");

// put code from examples here
StructuredQueryDefinition criteria =
   ... example of building query definition ...
// end code from examples

StringHandle searchHandle =
```

```
  queryMgr.search(
    criteria, new StringHandle()).get();
```

Additionally, these examples use query options from the following code:

```
String xmlOptions =
    "<search:options " +
        "xmlns:search='http://marklogic.com/appservices/search'>" +
      "<search:constraint name='date'>" +
        "<search:range type='xs:date'>" +
          "<search:element name='date'
ns='http://purl.org/dc/elements/1.1/'/>" +
        "</search:range>" +
      "</search:constraint>" +
      "<search:constraint name='popularity'>" +
        "<search:range type='xs:int'>" +
          "<search:element name='popularity' ns=''/>" +
        "</search:range>" +
      "</search:constraint>" +
      "<search:constraint name='title'>" +
        "<search:word>" +
          "<search:element name='title' ns=''/>" +
        "</search:word>" +
      "</search:constraint>" +
      "<search:return-results>true</search:return-results>" +
      "<search:transform-results apply='raw' />" +
    "</search:options>";

//JSON equivalant
String jsonOptions =
        "{\"options\":{" +
                "   \"constraint\": [" +
                "      {" +
                "         \"name\": \"date\"," +
                "         \"range\": {" +
                "            \"type\":\"xs:date\", " +
                "            \"element\": {" +
                "               \"name\": \"date\"," +
                "               \"ns\":
\"http://purl.org/dc/elements/1.1/\"" +
                "            }" +
                "         }" +
                "      }," +
                "      {" +
                "         \"name\": \"popularity\"," +
                "         \"range\": {" +
                "            \"type\":\"xs:int\", " +
                "            \"element\": {" +
                "               \"name\": \"popularity\"," +
                "               \"ns\": \"\"" +
                "            }" +
                "         }" +
                "      }," +
```

```
"       {" +
"          \"name\": \"title\"," +
"          \"word\": {" +
"             \"element\": {" +
"                \"name\": \"title\"," +
"                \"ns\": \"\"" +
"             }" +
"          }" +
"       }" +
"    ]," +
"    \"return-results\": \"true\"," +
"    \"transform-results\": {" +
"       \"apply\": \"raw\"" +
"    }" +
"}}";
```

```
QueryOptionsManager optionsMgr =
  dbClient.newServerConfigManager().newQueryOptionsManager();
optionsMgr.writeOptions("myopt",
    new StringHandle(xmlOptions).withFormat(Format.XML));
// Or, with JsonOptions:
    new StringHandle(jsonOptions).withFormat(Format.JSON));
```

This section contains the following examples:

- Example: Date Range Structured Query
- Example: Element Index Structured Query
- Example: Document Property Structured Query
- Example: Directory Structured Query
- Example: Document Structured Query
- Example: JSON Property Structured Query
- Example: Collection Structured Query

5.4.4.1 Example: Date Range Structured Query

For the boilerplate code environment in which this example runs, see the code snippet in "Structured Query Examples" on page 149.

The following example defines a query that searches for the `"2005-01-01"` value in the date range index.

```
StructuredQueryDefinition criteria =
   sb.containerQuery("date", Operator.EQ, "2005-01-01");

/* XML equivalent
<search:query xmlns:search=
   "http://marklogic.com/appservices/search">
  <search:range-constraint-query>
    <search:constraint-name>date</search:constraint-name>
    <search:value>2005-01-01</search:value>
  </search:range-constraint-query>
</search:query>
*/

/* JSON equivalent
{"query":{
      "range-constraint-query": {
              "constraint-name": "date",
              "value": "2005-01-01"
                      }
          }
}
*/
```

5.4.4.2 Example: Element Index Structured Query

For the boilerplate code environment in which this example runs, see the code snippet in "Structured Query Examples" on page 149.

The following example defines a query that searches for the `"Bush"` value within an element range index on `title`.

```
StructuredQueryDefinition criteria =
   sb.wordConstraint("title", "Bush");

/* XML equivalent
<search:query xmlns:search=
   "http://marklogic.com/appservices/search">
  <search:word-constraint-query>
    <search:constraint-name>title</search:constraint-name>
      <search:text>Bush</search:text>
  </search:word-constraint-query>
</search:query>
*/
```

```
/* JSON equivalent
{"query":{
      "word-constraint-query": {
              "constraint-name": "title",
              "text": "Bush"
                      }
          }
}
*/
```

5.4.4.3 Example: Document Property Structured Query

For the boilerplate code environment in which this example runs, see the code snippet in “Structured Query Examples” on page 149.

The following example defines a query that searches for the `"hello"` term in the value of any property.

```
StructuredQueryDefinition criteria =
   sb.properties(sb.term("hello"));

/* XML equivalent
<search:query xmlns:search=
   "http://marklogic.com/appservices/search">
  <search:properties-fragment-query>
    <search:term-query>
      <search:text>hello</search:text>
    </search:term-query>
  </search:properties-fragment-query>
</search:query>
*/

/* JSON equivalent
{"query":{
      "property-fragment-query": {
              "term-query": {,
              "text": "hello"
                        }
                   }
          }
}
*/
```

5.4.4.4 Example: Directory Structured Query

For the boilerplate code environment in which this example runs, see the code snippet in “Structured Query Examples” on page 149.

The following example defines a query that searches for documents in the `"http://testdoc/doc6/"` directory.

```
StructuredQueryDefinition criteria =
   sb.directory(true, "http://testdoc/doc6/");

/* XML equivalent
<search:query xmlns:search=
   "http://marklogic.com/appservices/search">
  <search:directory-query>
    <search:uri>
      <search:text>http://testdoc/doc6/</search:text>
    </search:uri>
  </search:directory-query>
</search:query>
*/

/* JSON equivalent
{"query":{
      "directory-query": {
              "uri": {,
              "text": "http://testdoc/doc6/"
                        }
                   }
          }
}
*/
```

5.4.4.5 Example: Document Structured Query

For the boilerplate code environment in which this example runs, see the code snippet in “Structured Query Examples” on page 149.

The following example defines a query that searches for the `"http://testdoc/doc6/"` document.

```
StructuredQueryDefinition criteria =
   sb.document("http://testdoc/doc2");

/* XML equivalent
<search:query xmlns:search=
   "http://marklogic.com/appservices/search">
  <search:document-query>
    <search:uri>
      <search:text>http://testdoc/doc2</search:text>
    </search:uri>
  </search:document-query>
</search:query>
```

```
*/
/* JSON equivalent
{"query":{
      "document-query": {
              "uri": {,
                  "text": "http://testdoc/doc2/"
                        }
                   }
          }
}
*/
```

5.4.4.6 Example: JSON Property Structured Query

For the boilerplate code environment in which this example runs, see the code snippet in “Structured Query Examples” on page 149.

The following example defines a query that searches for documents containing a JSON property named .

```
StructuredQueryDefinition criteria =
   sb.containerQuery(sb.jsonProperty("myProp"), sb.term("theValue"));

/* XML equivalent
<search:query xmlns:search=
   "http://marklogic.com/appservices/search">
  <search:container-query>
    <search:json-property>myProp</search:json-property>
    <search:term-query>
      <search:text>theValue</search:text>
    </search:term-query>
  </search:container-query>
</search:query>
*/

/* JSON equivalent
{"query":{
      "container-query": {
              "json-property" : "myProp",
              "term-query": {,
                  "text": "the-value"
                        }
                   }
          }
}
*/
```

5.4.4.7 Example: Collection Structured Query

For the boilerplate code environment in which this example runs, see the code snippet in “Structured Query Examples” on page 149.

The following example defines a query that searches documents belonging to the `"http://test.com/set3/set3-1"` collection.

```
StructuredQueryDefinition criteria =
   sb.collection("http://test.com/set3/set3-1");

/* XML equivalent
<search:query xmlns:search=
   "http://marklogic.com/appservices/search">
   <search:collection-query>
    <search:uri>
      <search:text>http://test.com/set3/set3-1</search:text>
      </search:uri>
  </search:collection-query>
</search:query>
*/
/* JSON equivalent
{"query":{
      "collection-query": {
              "uri": {,
                  "text": "http://test.com/set3/set3-1"
                        }
                   }
          }
}
*/
```

5.5 Prototype a Query Using Query By Example

This section describes how to use the Java API to perform a search using a Query By Example (QBE). A QBE enables rapid prototyping of queries for “documents that look like this” using search criteria that resemble the structure of documents in your database. If you are not familiar with QBE, see Searching Using Query By Example in *Search Developer’s Guide*.

This section covers the following topics:

- What is QBE
- Search Documents Using a QBE
- Validate a QBE
- Convert a QBE to a Combined Query

5.5.1 What is QBE

A Query By Example (QBE) enables rapid prototyping of queries for "documents that look like this" using search criteria that resemble the structure of documents in your database. If you are not familiar with QBE, see Searching Using Query By Example in *Search Developer's Guide*.

If your documents include an `author` XML element or JSON property, you can use the following example QBE to find documents with an `author` value of "Mark Twain".

Format	Example

XML	`<q:qbe xmlns:q="http://marklogic.com/appservices/querybyexample">`
	`  <q:query>`
	`  <author>Mark Twain</author>`
	`  </q:query>`
	`</q:qbe>`

JSON	`{`
	`  "$query": { "author": "Mark Twain" }`
	`}`

You can only use QBE to search XML and JSON documents. Metadata search is not supported. You can search by element, element attribute, and JSON property; fields are not supported. For details, see Searching Using Query By Example in *Search Developer's Guide*

A QBE is represented by `com.marklogic.client.query.RawQueryByExampleDefinition` in the Java API. Operations on a QBE are performed through a `QueryManager`.

The Java API supports the following operations on a QBE:

- Search XML and JSON documents.
- Validate the correctness of a QBE.
- Convert a QBE to a combined query for improved performance and full expressiveness.

5.5.2 Search Documents Using a QBE

To create a QBE from a raw XML or JSON representation, use any handle class that implements `com.marklogic.client.io.marker.StructureWriteHandle` to create a `RawQueryByExampleDefinition`.

The Java API includes `StructureWriteHandle` implementations that support creating a structure in XML or JSON from a string (`StringHandle`), a file (`FileHandle`), a stream (`InputStreamHandle`), and popular abstractions (`DOMHandle`, `DOM4JHandle`, `JDOMHandle`). For a complete list of implementations, see the Java API JavaDoc.

Follow this procedure to create a QBE and use it in a search:

1. Instantiate a `QueryManager`. The manager deals with interaction between the client and the database.

```
QueryManager queryMgr = client.newQueryManager();
```

2. Create a JSON or XML representation of the query, using a text editor or other tool or library. Use the syntax detailed in Searching Using Query By Example in the *Search Developer's Guide*. The following example uses `String` for the raw representation:

```
String rawXMLQuery =
  "<q:qbe xmlns:q='http://marklogic.com/appservices/querybyexample'>"+
    "<q:query>" +
      "<author>Mark Twain</author>" +
    "</q:query>" +
  "</q:qbe>";

//Or
String rawJSONQuery =
        "{" +
            "\"$query\": { \"author\": \"Mark Twain\" }" +
        "}";
```

3. Create a handle using a class that implements `StructureWriteHandle`, set the handle content format, and associate your query with the handle. For example:

```
// For an query expressed as XML
StringHandle rawHandle =
    new StringHandle(rawXMLQuery).withFormat(Format.XML);

// For a query expressed as JSON
StringHandle rawHandle =
    new StringHandle(rawJSONQuery).withFormat(Format.JSON);
```

4. Create a `RawQueryByExampleDefinition` from the handle. Optionally, include the name of persistent query options. For example:

```
// Use the default persistent query options
RawQueryByExampleDefinition querydef =
    queryMgr.newRawQueryByExampleDefinition(rawHandle);

// Use the persistent options previously saved as "myoptions"
RawQueryByExampleDefinition querydef =
    queryMgr.newRawQueryByExampleDefinition(rawHandle, "myoptions");
```

5. Perform a search using the `RawQueryByExampleDefinition` and a results handle.

```
SearchHandle resultsHandle =
    queryMgr.search(querydef, new SearchHandle());
```

5.5.3 Validate a QBE

When you perform a search, MarkLogic Server does not verify the correctnesss of your QBE. If your QBE is syntactically or semantically incorrect, you might get errors or surprising results. To avoid such issues, you can validate your QBE.

To validate a QBE, construct a query as described in "Search Documents Using a QBE" on page 157, and then pass it to `QueryManager.validate()` instead of `QueryManager.search()`. The validation report is returned in a `StructureReadHandle`. For example:

```
StringHandle validationReport =
    queryMgr.validate(qbeDefn, new StringHandle());
```

The report can be in XML or JSON format, depending on the format of the input query and the format you set on the handle. By default, validation returns a JSON report for a JSON input query and an XML report for an XML input query. You can override this behavior using the `withFormat()` method of your response handle.

5.5.4 Convert a QBE to a Combined Query

Generating a combined query from a QBE has the following potential benefits:

- Improve search performance.
- Access a wider array of search features.
- Debug your QBE by examining the lower level Search API constructs it generates.

A combined query combines a structured query and query options into a single XML or JSON query. For details, see "Apply Dynamic Query Options to Document Searches" on page 159.

To generate a combined query from a QBE, construct a query as described in "Search Documents Using a QBE" on page 157, and then pass it to `QueryManager.convert()` instead of `QueryManager.search()`. The results are returned in a `StructureReadHandle`. For example:

```
StringHandle combinedQueryHandle =
    queryMgr.convert(qbeDefn, new StringHandle());
```

The resulting handle can be used to construct a `RawCombinedQueryDefinition`; for details, see "Searching Using Combined Query" on page 160.

For more details on the query component of a combined query, see Searching Using Structured Queries in *Search Developer's Guide*.

5.6 Apply Dynamic Query Options to Document Searches

You can use a combined query to specify query options at query time, without first persisting them as named options. A *combined query* is an XML or JSON wrapper around a string query and/or a structured, cts, or QBE query, plus query options.

Note: The Java Client API does not support using a QBE in a combined query at this time. Use a standalone QBE and persistent query options instead.

This section covers the following topics:

5.6.1 Searching Using Combined Query

Combined queries are useful for rapid prototyping during development and for applications that need to modify query options on a per query basis. The `RawCombinedQueryDefinition` class represents a combined query in the Java API.

You can only create a combined query from raw XML or JSON; there is no builder class. A combined query can contain the following components, all optional:

- A string query
- A serialized structured query or cts query
- Query options

If you include both a string query and a structured query or cts query, the two queries are AND'd together.

For example, the following raw combined query uses a string query and a structured query to match all documents where the TITLE element contains the word “henry” and the term “fourth”. The options embedded in the query suppress the generation of snippets and extract just the /PLAY/TITLE element from the matched documents.

Format	Example

XML	`<search:search xmlns:search="http://marklogic.com/appservices/search">`
	`  <search:query>`
	`  <search:word-query>`
	`  <search:element name="TITLE"/>`
	`  <search:text>henry</text>`
	`  </search:word-query>`
	`  </search:query>`
	`  <search:qtext>fourth</search:qtext>`
	`  <search:options>`
	`  <search:extract-document-data>`
	`  <search:extract-path>/PLAY/TITLE</search:extract-path>`
	`  </search:extract-document-data>`
	`  <search:transform-results apply="empty-snippet"/>`
	`  </search:options>`
	`</search:search>`

JSON	`{"search" : {`
	`  "query": {`
	`  "word-query": {`
	`  "element": { "name": "TITLE" },`
	`  "text": [ "henry" ]`
	`  }`
	`  },`
	`  "qtext": "fourth",`
	`  "options": {`
	`  "extract-document-data": {`
	`  "extract-path": "/PLAY/TITLE"`
	`  },`
	`  "transform-results": {`
	`  "apply": "empty-snippet"`
	`  }`
	`  }`
	`} }`

For syntax details, see Syntax and Semantics in the *REST Application Developer’s Guide*.

Since there is no builder for `RawCombinedQueryDefinition`, you must construct the contents “by hand”, associate a handle with the contents, and then attach the handle to a `RawCombinedQueryDefinition` object. For example:

```
RawCombinedQueryDefinition xmlCombo =
    qm.newRawCombinedQueryDefinition(new StringHandle().with(
      // your raw XML combined query here
```

```
    ).withFormat(Format.XML));
// your raw JSON combined query here
    ).withFormat(Format.JSON));
```

For more complete examples, see “Combined Query Examples” on page 166.

Use any handle class that implements `com.marklogic.client.io.marker.StructureWriteHandle`. The Java API includes `StructureWriteHandle` implementations that support creating a structure in XML or JSON from input sources such as a string (`StringHandle`), a file (`FileHandle`), a stream (`InputStreamHandle`), and popular abstractions (`DOMHandle`, `DOM4JHandle`, `JDOMHandle`). For a complete list of implementations, see the *Java Client API Documentation*.

Though there is no builder for combined queries, you can use `StructuredQueryBuilder` to create the structured query portion of a combined query; for details, see “Creating a Combined Query Using StructuredQueryBuilder” on page 164.

The following procedure provides more detailed instructions for binding a handle on the raw representation `RawCombinedQueryDefinition` object usable for searching.

1. Instantiate a `QueryManager`. The manager deals with interaction between the client and the database. For example:

   ```
   QueryManager queryMgr = client.newQueryManager();
   ```

2. Create a JSON or XML representation of the query, using a text editor or other tool or library. For syntax details, see Syntax and Semantics in the *REST Application Developer's Guide*. The following example uses `String` for the raw representation of a combined query that contains a structured query:

   ```
   String rawXMLQuery =
       "<search:search "+
           "xmlns:search='http://marklogic.com/appservices/search'>"+
         "<search:query>"+
           "<search:term-query>"+
             "<search:text>neighborhoods</search:text>"+
           "</search:term-query>"+
           "<search:value-constraint-query>"+
             "<search:constraint-name>industry</search:constraint-name>"+
             "<search:text>Real Estate</search:text>"+
           "</search:value-constraint-query>"+
         "</search:query>"+
         "<search:options>"+
           "<search:constraint name='industry'>"+
             "<search:value>"+
               "<search:element name='industry' ns=''/>"+
             "</search:value>"+
           "</search:constraint>"+
         "</search:options>"+
       "</search:search>";
   ```

```
//Or
String rawJSONQuery =
        "{\"search\":{" +
                "   \"query\": {" +
                "      \"term-query\": {" +
                "         \"text\": \"neighborhoods\"" +
                "      }," +
                "      \"value-constraint-query\": {" +
                "         \"constraint-name\": \"industry\"," +
                "         \"text\": \"Real Estate\"" +
                "      }" +
                "   }," +
                "   \"options\": {" +
                "      \"constraint\": {" +
                "         \"name\": \"industry\"," +
                "         \"value\": {" +
                "            \"element\": {" +
                "               \"name\": \"industry\"," +
                "               \"ns\": \"\"" +
                "            }" +
                "         }" +
                "      }" +
                "   }" +
                "}" +
            "}";
```

3. Create a handle on your raw query, using a class that implements `StructureWriteHandle`. For example:

```
// Query as XML
StringHandle rawHandle =
    new StringHandle().withFormat(Format.XML).with(rawXMLQuery);

// Query as JSON
StringHandle rawHandle =
    new StringHandle().withFormat(Format.JSON).with(rawJSONQuery);
```

4. Create a `RawCombinedQueryDefinition` from the handle. Optionally, include the name of persistent query options. For example:

```
// Use the default persistent query options
RawCombinedQueryDefinition querydef =
    queryMgr.newRawCombinedQueryDefinition(rawHandle);

// Use persistent options previously saved as "myoptions"
RawCombinedQueryDefinition querydef =
    queryMgr.newRawCombinedQueryDefinition(rawHandle, "myoptions");
```

5. Perform a search using the `RawCombinedQueryDefinition` and a results handle.

```
SearchHandle resultsHandle =
    queryMgr.search(querydef, new SearchHandle());
```

For a complete example of searching with a combined query, see `com.marklogic.client.example.cookbook.RawCombinedSearch` in the `example/` directory of your Java API installation.

5.6.2 Creating a Combined Query Using StructuredQueryBuilder

When building a `RawCombinedQuery` that contains a structured query, you can use `StructuredQueryBuilder` to create the structured query portion of a combined query. This technique always produces an XML combined query.

Create a `StructuredQueryDefinition` using `StructuredQueryBuilder`, just as you would when searching with a standalone structured query. Then, extract the serialized structured query using `StructuredQueryDefinition.serialize`, and embed it in your combined query. For example:

```
QueryManager qm = client.newQueryManager();

StructuredQueryBuilder qb = qm.newStructuredQueryBuilder();
StructuredQueryDefinition structuredQuery =
    qb.word(qb.element("TITLE"), "henry");
String comboq =
    "<search xmlns=\"http://marklogic.com/appservices/search\">" +
        structuredQuery.serialize() +
    "</search>";
RawCombinedQueryDefinition query =
    qm.newRawCombinedQueryDefinition(
        new StringHandle(comboq).withFormat(Format.XML));
```

You can also include a string query and/or query options in your combined query. For a more complete example, see “Combined Query Examples” on page 166.

5.6.3 Interaction with Persistent Query Options

Dynamic query options supplied in a combined query are merged with persistent and default options that are in effect for the search. If the same non-constraint option is specified in both the combined query and persistent options, the setting in the combined query takes precedence.

Constraints are overridden by name. That is, if the dynamic and persistent options contain a `<constraint/>` element with the same `name` attribute, 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 use the following raw XML combined query to define dynamic query options:

```
<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>
```

You can create a `RawQueryDefinition` that encapsulates the combined query and the persistent options:

```
StringHandle rawQueryHandle =
    new StringHandle(...).withFormat(Format.XML);
RawCombinedQueryDefinition querydef =
    queryMgr.newRawCombinedQueryDefinition(
        rawQueryHandle, "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>
  <constraint name="not-same">
    <element-query name="title" ns="http://my/namespace" />
  </constraint>
</options>
```

5.6.4 Combined Query Examples

The examples in this section demonstrate constructing different types of combined queries using the Java Client API. The example queries are constructed as in-memory strings to keep the example self-contained, but you could just as easily read them from a file or other external source.

Unless otherwise noted, the examples all use equivalent queries and query options. The query is a word query on the term “henry” where it appears in a TITLE element, AND’d with a string query for the term “henry”.

The examples also share the scaffolding in “Shared Scaffolding for Combined Query Examples” on page 168, which defines the query options and drives the search. However, the primary point of the examples is the query construction.

See the following topics for example code:

- Example: Structured and String Query
- Example: cts and String Query
- Shared Scaffolding for Combined Query Examples

5.6.4.1 Example: Structured and String Query

The following two functions perform a search using a combined query that contains a string query, a structured query, and query options.

The first function expresses the query in XML, using StructuredQueryBuilder to create the structured query portion of the combined query. The second function expresses the query in JSON. Both functions use the options and search driver from “Shared Scaffolding for Combined Query Examples” on page 168.

```
// Use a combined query containing a structured query, string query,
// and query options. A StructuredQueryBuilder is used to create the
// structured query portion. The combined query is expressed as XML.
//
public static void withXmlStructuredQuery() {
    StructuredQueryBuilder qb = new StructuredQueryBuilder();
    StructuredQueryDefinition builtSQ =
        qb.word(qb.element("TITLE"), "henry");

    System.out.println("** Searching with an XML structured query...");
    doSearch(new StringHandle().with(
        "<search xmlns=\"http://marklogic.com/appservices/search\">" +
            "<qtext>fourth</qtext>" +
            builtSQ.serialize() +
            XML_OPTIONS +
        "</search>").withFormat(Format.XML));
}

// Use a combined query containing a structured query, string query,
```

```
// and query options. The combined query is expressed as JSON.
public static void withJsonStructuredQuery() {
    System.out.println("** Searching with a JSON structured query...");
    doSearch(new StringHandle().with(
        "{\"search\" : {" +
            "\"query\": {" +
                "\"word-query\": {" +
                    "\"element\": { \"name\": \"TITLE\"}," +
                        "\"text\": [ \"henry\" ]" +
                "}" +
            "}, " +
            "\"qtext\": \"fourth\"," +
            JSON_OPTIONS +
        "} }").withFormat(Format.JSON));
}
```

5.6.4.2 Example: cts and String Query

The following two functions perform a search using a combined query that contains a string query, a cts query, and query options.

The first function expresses the query in XML. The second function expresses the query in JSON. Both functions use the options and search driver from “Shared Scaffolding for Combined Query Examples” on page 168.

```
// Use a combined query containing a cts query, string query,
// and query options. The combined query is expressed as XML.
public static void withXmlCtsQuery() {
    System.out.println("** Searching with an XML cts query...");
    doSearch(new StringHandle().with(
        "<search xmlns=\"http://marklogic.com/appservices/search\">" +
          "<cts:element-word-query xmlns:cts=\"http://marklogic.com/cts\">" +
            "<cts:element>TITLE</cts:element>" +
            "<cts:text xml:lang=\"en\">henry</cts:text>" +
          "</cts:element-word-query>" +
          "<qtext>fourth</qtext>" +
          XML_OPTIONS +
        "</search>").withFormat(Format.XML));
}

// Use a combined query containing a cts query, string query,
// and query options. The combined query is expressed as JSON.
public static void withJsonCtsQuery() {
    System.out.println("** Searching with a JSON cts query...");
    doSearch(new StringHandle().with(
        "{\"search\" : {" +
            "\"ctsquery\": {" +
              "\"elementWordQuery\": {" +
                "\"element\" : [\"TITLE\"]," +
                "\"text\" : [\"henry\"]," +
                "\"options\" : [\"lang=en\"]" +
              "}" +
            "}, " +
```

```
            "\"qtext\": \"fourth\"," +
            JSON_OPTIONS +
        "} }").withFormat(Format.JSON));
}
```

5.6.4.3 Shared Scaffolding for Combined Query Examples

The examples in “Combined Query Examples” on page 166 share the scaffolding in this section for connecting to MarkLogic, defining query options, performing a search, and displaying the search results.

The query options are designed to strip down the search results into something easy for the example code to process while still emitting simple but meaningful output. This is done by suppressing snippeting and using the `extract-document-data` option to return just the TITLE element from the matches.

The `doSearch` method performs the search, independent of the structure of the combined query, and prints out the matched titles. The shown result processing is highly dependent on the query options and structured of the example documents.

```
package examples;

import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathFactory;

import org.w3c.dom.Document;

import com.marklogic.client.DatabaseClient;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.io.Format;
import com.marklogic.client.io.SearchHandle;
import com.marklogic.client.io.StringHandle;
import com.marklogic.client.io.marker.StructureWriteHandle;
import com.marklogic.client.query.ExtractedItem;
import com.marklogic.client.query.ExtractedResult;
import com.marklogic.client.query.MatchDocumentSummary;
import com.marklogic.client.query.QueryManager;
import com.marklogic.client.query.RawCombinedQueryDefinition;
import com.marklogic.client.query.StructuredQueryBuilder;
import com.marklogic.client.query.StructuredQueryDefinition;

import javax.xml.xpath.XPathExpressionException;

public class CombinedQuery {
    // replace with your MarkLogic Server connection information
    static String HOST = "localhost";
    static int PORT = 8000;
    static String DATABASE = "bill";
    static String USER = "username";
    static String PASSWORD = "password";
    private static DatabaseClient client =
            DatabaseClientFactory.newClient(
```

```
                HOST, PORT, DATABASE,
                new DatabaseClientFactory.DigestAuthContext(USER, PASSWORD));

        // Define query options to be included in our raw combined query.
        static String XML_OPTIONS =
            "<options xmlns=\"http://marklogic.com/appservices/search\">" +
              "<extract-document-data>" +
                "<extract-path>/PLAY/TITLE</extract-path>" +
              "</extract-document-data>" +
              "<transform-results apply=\"empty-snippet\"/>" +
              "<search-option>filtered</search-option>" +
            "</options>";
        static String JSON_OPTIONS =
            "\"options\": {" +
                "\"extract-document-data\": {" +
                    "\"extract-path\": \"/PLAY/TITLE\"" +
                "}," +
                "\"transform-results\": {" +
                     "\"apply\": \"empty-snippet\"" +
                "}" +
            "}";

        // Perform a search using a combined query. The input handle is
        // assumed to contain an XML or JSON combined query.
        //
        // The combined query must contain either the XML_OPTIONS or
        // JSON_OPTIONS defined above. The options produce a
        // search:response in which each search:match has this form:
        //
        // <search:result index="n" uri="..." path="..." score="..."
        //     confidence="....4450079" fitness="0.5848901" href="..."
        //     mimetype="..." format="xml">
        //   <search:snippet/>
        //   <search:extracted kind="element">
        //     <TITLE>a title</TITLE>
        //   </search:extracted>
        // </search:result>
        //
        // XML DOM is used to extract the title text from the extrace elems
        //
        public static void doSearch(StructureWriteHandle queryHandle) {
            // Create a raw combined query
            QueryManager qm = client.newQueryManager();
            RawCombinedQueryDefinition query =
                    qm.newRawCombinedQueryDefinition(queryHandle);

            // Perform the search
            SearchHandle results = qm.search(query, new SearchHandle());

            // Process the results, printint out the title of each match
            try {
              XPathExpression xpath = XPathFactory.newInstance()
                  .newXPath().compile("//TITLE");
              for (MatchDocumentSummary match : results.getMatchResults()) {
```

```
                    ExtractedResult extracted = match.getExtracted();
                    if (!extracted.isEmpty()) {
                        for (ExtractedItem item : extracted) {
                            System.out.println(
                                xpath.evaluate(item.getAs(Document.class)));
                        }
                    }
                }
            } catch (XPathExpressionException e) {
                e.printStackTrace();
            }
        }

        // with*Query methods go here

    public static void main(String[] args) {
        // call with*Query methods of interest to you
    }
```

5.6.5 Performance Considerations

Using persistent query options usually performs better than using dynamic query options. In most cases, the performance difference between the two methods 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 a `QueryOptionsManager` to persist your options under a name and associate the options with a simple `StringQueryDefinition` or `StructuredQueryDefinition`.

5.7 Search On Tuples (Tuples Query / Values Query)

You can return values and tuples (co-occurrences) through the Java API. Value and tuple searches require the appropriate range indexes are configured on your MarkLogic Server database. For background on values and co-occurrences, see Browsing With Lexicons in the *Search Developer's Guide*.

This section includes the following parts:

- Values Search
- Tuples Search
-

5.7.1 Values Search

The following returns values through the Java API:

The following are the basic steps to search on values:

1. Instantiate a `QueryManager`. The manager deals with interaction between the client and the database.

```
QueryManager queryMgr = client.newQueryManager();
```

2. Create a `ValuesDefinition` object using the query manager. In the following example, the parameters define a named values constraint (`myvalue`) defined in previously persisted query options (`valueoptions`):

```
// build a search definition
ValuesDefinition vdef =
    queryMgr.newValuesDefinition("myvalue", "valuesoptions");
```

3. Configure additional values or tuples search properties, as needed. For example, call `setAggregate()` to set the name of the aggregate function to be applied as part of the query.

```
vdef.setAggregate("correlation", "covariance");
```

4. Run a search with the `ValuesDefinition` object as an argument, returning a `ValuesHandle` object. Note that the tuples search method is called `values()`, not `search()`.

```
ValuesHandle results = queryMgr.values(vdef, new ValuesHandle());
```

You can retrieve results one page at a time by defining a page length and starting position with the `QueryManager` interface. For example, the following code snippet retrieves a “page” of 5 values beginning with the 10th value.

```
queryMgr.setPageLength(5);
ValuesHandle result = queryMgr.values(vdef, new ValuesHandle(), 10);
```

For more information on values search concepts, see Returning Lexicon Values With search:values and Browsing With Lexicons in the *Search Developer’s Guide*.

5.7.2 Tuples Search

The following returns tuples (co-occurrences) through the Java API:

1. Instantiate a `QueryManager`. The manager deals with interaction between the client and the database.

```
QueryManager queryMgr = client.newQueryManager();
```

2. Create a `ValuesDefinition` object using the query manager. In the following example, the parameters define a named tuples constraint (`co`) defined in previously persisted query options (`tupleoptions`):

```
// build a search definition
ValuesDefinition vdef =
    queryMgr.newValuesDefinition("co", "tupleoptions");
```

3. Run a search with the `ValuesDefinition` object as an argument, returning a `TuplesHandle` object. Note that the tuples search method is called `tuples()`, not `search()`.

```
TuplesHandle results = queryMgr.tuples(vdef, new TuplesHandle());
```

You can retrieve results one page at a time by defining a page length and starting position with the `QueryManager` interface. For example, the following code snippet retrieves a "page" of 5 tuples beginning with the 10th one.

```
queryMgr.setPageLength(5);
TuplesHandle result = queryMgr.tuples(vdef, new TuplesHandle(), 10);
```

For more information on tuples search concepts, see Returning Lexicon Values With search:values and Browsing With Lexicons in the *Search Developer's Guide*.

5.7.3 Adding a Constraining Query

You can constrain the results of a values or tuples query to only return values in documents matching the constraining query. The constraining query can be a string, structured, combined, or cts query.

To add a constraining query to a values or tuples query, construct the query definition as usual and then attach it to the values or tuples query using the `ValuesDefinition.setQueryDefintion` method.

The following example adds a constraining `cts:query` to a values query, assuming a set of query options are installed under the name "valopts" that defines a `values` option named "title". Only values in documents matching the `cts:element-word-query` will be returned.

```
QueryManager qm = client.newQueryManager();

// Create a cts:query with which to constrain the values query result
String serializedQuery =
    "<cts:element-word-query xmlns:cts=\"http://marklogic.com/cts\">" +
        "<cts:element>TITLE</cts:element>" +
        "<cts:text xml:lang=\"en\">fourth</cts:text>" +
    "</cts:element-word-query>";
RawCtsQueryDefinition ctsquery =
    qm.newRawCtsQueryDefinition(
        new StringHandle(serializedQuery).withFormat(Format.XML));

// Create a values query and evaluate it
```

```
ValuesDefinition vdef = qm.newValuesDefinition("title", "valopts");
vdef.setQueryDefinition(ctsquery);
ValuesHandle results = qm.values(vdef, new ValuesHandle());
```

5.8 Limiting A Search To Specific Collections And/Or A Directory

All query definition interfaces have `setCollections()` and `setDirectory()` methods. By calling `setDirectory(`*`directory_URI_string`*`)` on your query definition, you limit your search to that directory. By calling `setCollections(`*`list_of_collection_name_strings`*`)` on your query definition, you limit your search to those collections. You can call both and limit your search to collections and a single directory.

5.9 Searching Values Metadata Fields

Values metadata, sometimes called key-value metadata, can only be searched if you define a metadata field on the keys you want to search. Once you define a field on a metadata key, use the normal field search capabilities to include a metadata field in your search. For example, you can use a `cts:field-word-query` or a structured query `word-query` on a metadata field, or define a constraint on the field and use the constraint in a string query.

For more details, see Metadata Fields in the *Administrator's Guide*. For some examples, see Example: Structured Search on Key-Value Metadata Fields or Searching Key-Value Metadata Fields in the *Search Developer's Guide*.

5.10 Transforming Search Results

You can make arbitrary changes to the results of a search or values query by applying a server-side transformation function to the results. This section covers the following topics:

- Writing a Search Result Transform
- Using a Search Result Transform

5.10.1 Writing a Search Result Transform

Search response transforms use the same interface and framework as content transformations applied during document ingestion, described in Writing Transformations in the *REST Application Developer's Guide*.

Your transform function receives the XML or JSON search response prepared by MarkLogic Server in the `content` parameter. For example, if the response is XML, then the content passed to your transform is a document node with a `<search:response/>` root element. Any customizations made by the `transform-results` query option or result decorators are applied before calling your transform function.

You can probe the document type to test whether the input to your transform receives JSON or XML input. For example, in server-side JavaScript, you can test the documentFormat property of a document node:

```
function myTransform(context, params, content) {
  if (content.documentFormat == "JSON") {
    // handle as JSON or a JavaScript object
  } else {
    // handle as XML
}

  ...
}
```

In XQuery and XSLT, you can test the node kind of the root of the document, which will be `element` for XML and `object` for JSON.

```
declare function dumper:transform(
  $context as map:map,
  $params as map:map,
  $content as document-node()
) as document-node()
{
  if (xdmp:node-kind($content/node() eq "element")
  then(: process as XML :)
  else (: process as JSON :)
```

As with read and write transforms, the content object is immutable in JavaScript, so you must call toObject to create a mutable copy:

```
var output = content.toObject();
...modify output...
return output;
```

The type of document you return must be consistent with the `output-type` (`outputType`) context value. If you do not return the same type of document as was passed to you, set the new output type on the `context` parameter.

5.10.2 Using a Search Result Transform

To use a server transform function:

1. Create a transform function according to the interface described in Writing Transformations in the *REST Application Developer's Guide*.

2. Install your transform function on the REST API instance following the instructions in "Installing Transforms" on page 282.

3. Specify the transform function in your `QueryDefinition` by calling `setResponseTransform()`. For example:

```
QueryManager queryMgr = dbClient.newQueryManager();
StringQueryDefinition query = queryMgr.newStringDefinition();
query.setCriteria("cat AND dog");
```

```
query.setResponseTransform(new ServerTransform("example"));
```

You are responsible for specifying a handle type capable of interpreting the results produced by your transform function. The `SearchHandle` implementation provided by the Java API only understands the search results structure that MarkLogic Server produces by default.

5.11 Generating Search Term Completion Suggestions

Use `com.marklogic.client.query.QueryManager.suggest()` to generate search term completion suggestions that match a wildcard terminated string. For example, if the user enters the text “doc” into a search box, you can use `suggest()` with “doc” as string criteria to retrieve a list of terms matching “doc*”, and then display them to user. This service is analogous to calling the XQuery function `search:suggest` or the REST API method `GET /version/suggest`.

The following topics are covered:

5.11.1 Basic Steps

Use the following procedure to retrieve search term completion suggestions:

1. Configure at least one database index on the XML element, XML attribute, or JSON property values you want to include in the search for suggestions. For performance reasons, a range or collection index is recommended over a word lexicon; for details, see `search:suggest`.

2. Create and install persistent query options that use your index as a suggestion source by including it in the definition of a `default-suggestion-source` or `suggestion-source option`. For details, see Search Term Completion Using search:suggest in the *Search Developer’s Guide* and “Creating Persistent Query Options From Raw JSON or XML” on page 193.

3. Instantiate a `QueryManager`. The manager deals with interaction between the client and the database.

   ```
   QueryManager queryMgr = client.newQueryManager();
   ```

4. Use the query manager to obtain a `SuggestDefinition` object.

   ```
   SuggestDefinition sd = queryMgr.newSuggestDefinition();
   ```

5. Configure the definition with the string for which to retrieve suggestions. For example, the following call configures the operation to return matches to the wildcard string "doc*":

```
sd.setStringCriteria("doc");
```

6. Optionally, associate persistent query options with the suggest definition. You can skip this step if your default query options include one or more `suggestion-source` or `default-suggestion-source` options. Otherwise, specify the name of previously installed query options that include `suggestion-source` and/or `default-suggestion-source` settings.

```
sd.setOptions("opt-suggest");
```

7. Optionally, configure additional properties, such as the maximum number of suggestions to return or additional string queries with which to filter the results. For example:

```
sd.setLimit(5);
sd.setQueryStrings("prefix:xdmp");
```

8. Retrieve the suggestions using your suggest definition and query manager:

```
String[] results = queryMgr.suggest(sd);
```

5.11.2 Example: Generating Search Suggestions

This example walks you through configuring your database and REST instance to try retrieving search suggestions. The Documents database is assumed in this example, but you can use any database. This example has the following parts:

1. Initialize the Database
2. Install Query Options
3. Get Search Suggestions

5.11.2.1 Initialize the Database

Run the following query in Query Console to load the sample data into your database, or use a `DocumentManager` to insert equivalent documents into the database. The example will retrieve suggestions for the `<name/>` element, with and without a constraint based on the `<prefix/>` element.

```
xdmp:document-insert("/suggest/load.xml",
  <function>
    <prefix>xdmp</prefix>
    <name>document-load</name>
  </function>
  );
xdmp:document-insert("/suggest/insert.xml",
  <function>
    <prefix>xdmp</prefix>
    <name>document-insert</name>
  </function>
  );
```

```
xdmp:document-insert("/suggest/query.xml",
  <function>
    <prefix>cts</prefix>
    <name>document-query</name>
  </function>
  );
xdmp:document-insert("/suggest/search.xml",
  <function>
    <prefix>cts</prefix>
    <name>search</name>
  </function>
  );
```

The equivalent in Javascript:

```
declareUpdate();
xdmp.documentInsert("/suggest/load.json",
    {function:
        {prefix: "xdmp",
         name: "document-load"}
    });

xdmp.documentInsert("/suggest/insert.json",
    {function:
        {prefix: "xdmp",
         name: "document-insert"}
    });

xdmp.documentInsert("/suggest/query.json",
    {function:
        {prefix: "cts",
         name: "document-query"}
    });

xdmp.documentInsert("/suggest/load.search",
    {function:
        {prefix: "cts",
         name: "document-search"}
    });
```

To create the range index used by the example, run the following query in Query Console, or use the Admin Interface to create an equivalent index on the `name` element. The following query assumes you are using the Documents database; modify as needed.

```
xquery version "1.0-ml";
import module namespace admin = "http://marklogic.com/xdmp/admin"
  at "/MarkLogic/admin.xqy";
admin:save-configuration(
  admin:database-add-range-element-index(
    admin:get-configuration(),
    xdmp:database("Documents"),
    admin:database-range-element-index(
    "string", "http://marklogic.com/example",
```

```
      "name", "http://marklogic.com/collation/", fn:false())
    )
);
```

The equivalent in Javascript:

```
declareUpdate();
const admin = require("/MarkLogic/admin.xqy");
admin.saveConfiguration(
  admin.databaseAddRangeElementIndex(admin.getConfiguration(),
                                     xdmp.database("Documents"),
                                     admin.databaseRangeElementIndex("string"
                                           "http://marklogic.com/example",
                                           "name",
                                           "http://marklogic.com/collation/",
                                           fn.false()))
)
```

5.11.2.2 Install Query Options

The example relies on the following query options. These options use the `<name/>` element as the default suggestion source. The value constraint named “prefix” is included only to illustrate how to use additional query to filter suggestions. It is not required to get suggestions.

```
<options xmlns="http://marklogic.com/appservices/search">
 <default-suggestion-source>
   <range type="xs:string" facet="true">
      <element ns="http://marklogic.com/example" name="name"/>
   </range>
 </default-suggestion-source>
 <constraint name="prefix">
   <value>
      <element ns="http://marklogic.com/example" name="prefix"/>
   </value>
 </constraint>
</options>
```

The equivalent in JSON:

```
{"options":{
        "default-suggestion-source": {
            "range": {
                "facet": "true",
                        "element": {
                    "ns": "http://marklogic.com/example",
                            "name": "name"
                }
            }
        },
        "constraint": {
            "name": "prefix",
                    "value": {
                "element": {
```

```
                    "ns": "http://marklogic.com/example",
                            "name": "prefix"
                }
            }
        }
    }
}
```

Install the options under the name `"opt-suggest"` using `QueryOptionsManager`, as described in “Creating Persistent Query Options From Raw JSON or XML” on page 193. For example, to configure the options using a string literal, do the following:

```
String options =
  "<options xmlns=\"http://marklogic.com/appservices/search\">" +
    "<default-suggestion-source>" +
      "<range type="xs:string" facet="true">" +
        "<element ns="http://marklogic.com/example" name="name"/>" +
      "</range>" +
    "</default-suggestion-source>" +
    "<constraint name="prefix">" +
      "<value>
        "<element ns="http://marklogic.com/example" name="prefix"/>" +
      "</value>" +
    "</constraint>" +
  "</options>";

// Or the JSON equivalent:

String optionsJson =
        "{\"options\":{" +
                "   \"default-suggestion-source\": {" +
                "      \"range\": {" +
                "         \"facet\": \"true\"," +
                "         \"element\": {" +
               "           \"ns\": \"http://marklogic.com/example\"," +
                "            \"name\": \"name\"" +
                "         }" +
                "      }" +
                "   }," +
                "   \"constraint\": {" +
                "      \"name\": \"prefix\"," +
                "      \"value\": {" +
                "         \"element\": {" +
               "           \"ns\": \"http://marklogic.com/example\"," +
                "            \"name\": \"prefix\"" +
                "         }" +
                "      }" +
                "   }" +
                "}" +
            "}";

StringHandle handle =
    new StringHandle(options).withFormat(Format.XML);
```

```
QueryManager queryMgr = client.newQueryManager();

QueryOptionsManager optMgr =
    client.newServerConfigManager().newQueryOptionsManager();
optMgr.writeOptions("opt-suggest", handle);
```

5.11.2.3 Get Search Suggestions

To retrieve search suggestions, use `QueryManager.suggest()`. For example:

```
QueryManager queryMgr = client.newQueryManager();
SuggestDefinition sd = queryMgr.newSuggestDefinition();
sd.setStringCriteria("doc");
String[] results = queryMgr.suggest(sd);
```

The results contain the following suggestions derived from the sample input documents:

```
document-insert
document-load
document-query
```

Recall that the query options include a value constraint on the `prefix` element. You can use this constraint with the string query `prefix:xdmp` as filter so that the operation returns only suggestions occuring in a documents with a `prefix` value of xdmp. For example:

```
sd.setStringCriteria("doc");
sd.setQueryStrings("prefix:xdmp");
String[] results = queryMgr.suggest(sd);
```

Now, the results contain only `document-insert` and `document-load`. The function named `document-query` is excluded because the `prefix` value for this document is not xdmp.

5.11.3 Where to Find More Information

For more details on using search suggestions, including performance recommendations and additional examples, see the following:

- `search:suggest` (XQuery function)
- Search Term Completion Using search:suggest in *Search Developer's Guide*.

5.12 Extracting a Portion of Matching Documents

This section describes how to use the `extract-document-data` query option with `QueryManager.search` to extract a subset of each matching document and return it in your search results.

This section covers the following related topics:

- Overview of Extraction

- Basic Steps for Search Match Extraction
- Example: Extracting a Portion of Each Matching Document

You can also use this option with a multi-document read (`DocumentManager.search`) to retrieve the extracted subset instead of the complete document; for details, see “Extracting a Portion of Each Matching Document” on page 89.

5.12.1 Overview of Extraction

By default, `QueryManager.search` returns a search result summary. When you perform a search that includes the `extract-document-data` query option, you can embed selected portions of each matching document in the search results and access them through returned Handle.

The projected contents are specified through absolute XPath expressions in `extract-document-data` and a `selected` attribute that specifies how to treat the selected content.

The `extract-document-data` option has the following general form. For details, see extract-document-data in the *Search Developer's Guide* and Extracting a Portion of Matching Documents in the *Search Developer's Guide*.

```
<extract-document-data selected="howMuchToInclude">
  <extract-path>/path/to/content</extract-path>
</extract-document-data>
```

The equivalent in JSON:

```
{"extract-document-data":{
            "selected": "howMuchToInclude",
            "extract-path": "/path/to/content"
        }
}
```

The path expression in `extract-path` is limited to the subset of XPath described in The extract-document-data Query Option in the *XQuery and XSLT Reference Guide*.

Use the `selected` attribute to control what to include in each result. This attribute can take on the following values: “all”, “include”, “include-with-ancestors”, and “exclude”. For details, see *Search Developer's Guide*.

The document projections created with `extract-document-data` are accessible in the following way. For a complete example, see “Example: Extracting a Portion of Each Matching Document” on page 184.

```
QueryManager qm = client.newQueryManager();
SearchHandle results = qm.search(query, new SearchHandle());
MatchDocumentSummary matches[] = results.getMatchResults();
for (MatchDocumentSummary match : matches) {
    ExtractedResult extracts = match.getExtracted();
    for (ExtractedItem extract: extracts) {
```

```
            // do something with each projection
        }
    }
```

The `ExtractedItem` interface includes `get` and `getAs` methods for manipulating the extracted content through either a handle (`ExtractedItem.get`) or an object (`ExtractedItem.getAs`). For example, the following statement uses `getAs` to access the extracted content as a `String`:

```
String content = extract.getAs(String.class);
```

You can use `ExtractedResult.getFormat` with `ExtractedItem.get` to detect the type of data returned and access the content with a type-specific handle. For example:

```
for (MatchDocumentSummary match : matches) {
    ExtractedResult extracts = match.getExtracted();
    for (ExtractedItem extract: extracts) {
        if (match.getFormat() == Format.JSON) {
            JacksonHandle handle = extract.get(new JacksonHandle());
            // use the handle contents
        } else if (match.getFormat() == Format.XML) {
            DOMHandle handle = extract.get(new DOMHandle());
            // use the handle contents
        }
    }
}
```

The search returns an `ExtractedItem` for each match to a path in a given document when you set `select` to "include". For example, if your `extract-document-data` option includes multiple extraction paths, you can get an `ExtractedItem` for each path. Similarly, if a single document contains more than one match for a single path, you get an ExtractedItem for each match.

By contrast, when you set `select` to "all", "include-with-ancestors", or "exclude", you get a single `ExtractedItem` per document that contains a match.

5.12.2 Basic Steps for Search Match Extraction

Use the following technique to perform a search that includes extracted data in the search results. For a complete example of applying this pattern, see "Example: Extracting a Portion of Each Matching Document" on page 184.

1. Instantiate a `QueryManager`. The manager deals with interaction between the client and the database.

   ```
   QueryManager queryMgr = client.newQueryManager();
   ```

2. Define query options that include the `extract-document-data` option. Make the option available to your search by embedding it in the options of a combined query or installing it

as part of a named persistent query options set. The following example uses the option in a String that can be used to construct a `RawCombinedQuery`:

```
String rawQuery =
  "<search xmlns=\"http://marklogic.com/appservices/search\">" +
  "  <query><directory-query><uri>/extract/</uri></directory-query></query>" +
  "  <options xmlns=\"http://marklogic.com/appservices/search\">" +
  "    <extract-document-data selected=\"include\">" +
  "      <extract-path>/parent/body/target</extract-path>" +
  "    </extract-document-data>" +
  "  </options>" +
  "</search>";

//The equivalent in JSON:
String rawQueryJson =
        "{\"search\":{" +
                "   \"query\": {" +
                "      \"directory-query\": {" +
                "         \"uri\": \"/extract/\"" +
                "      }" +
                "   }," +
                "   \"options\": {" +
                "      \"extract-document-data\": {" +
                "         \"selected\": \"include\"," +
                "         \"extract-path\": \"/parent/body/target\"" +
                "      }" +
                "   }" +
                "}" +
            "}";
```

For details, see “Prototype a Query Using Query By Example” on page 156 or “Using QueryOptionsManager To Delete, Write, and Read Options” on page 192.

3. Create a query using any of the techniques discussed in this chapter. For example, the following snippet creates a RawCombinedQuery from the string shown in Step 2.

```
StringHandle qh = new StringHandle(rawQuery).withFormat(Format.XML);
//Or with rawQueryJson
StringHandle qh = new
StringHandle(rawQueryJson).withFormat(Format.JSON);
QueryManager qm = client.newQueryManager();
RawCombinedQueryDefinition query =
qm.newRawCombinedQueryDefinition(qh);
```

4. Perform a search using your query and options that include `extract-document-data`.

```
SearchHandle results = qm.search(query, new SearchHandle());
```

5. Use the search handle to access the extracted content through the match results. For example:

```
MatchDocumentSummary matches[] = results.getMatchResults();
for (MatchDocumentSummary match : matches) {
    ExtractedResult extracts = match.getExtracted();
    for (ExtractedItem extract: extracts) {
        // do something with each projection
    }
}
```

If you do not use a `SearchHandle` to capture your search results, you must access the extracted content from the raw search results. For details on the layout, see Extracting a Portion of Matching Documents in the *Search Developer's Guide*.

5.12.3 Example: Extracting a Portion of Each Matching Document

This example demonstrates the use of the `extract-document-data` query option to embed a selected subset of data from matched documents in the search results. For an example of using extract-document-data as part of a multi-document read, see "Extracting a Portion of Each Matching Document" on page 89.

The example documents are inserted into the "/extract/" directory in the database to make them easy to manage in the example. The example data includes one XML document and one JSON document, structured such that a single XPath expression can be used to demonstrate using `extract-document-data` on both types of document.

The example documents have the following contents, with the bold portion being the content extracted using the XPath expression `/parent/body/target`.

```
JSON:

{"parent": {
  "a": "foo",
  "body": {
    "target": "content1"
  },
  "b": "bar"
}}

XML:

<parent>
  <a>foo</a>
  <body>
    <target>content2</target>
  </body>
  <b>bar</b>
</parent>
```

The example uses a `RawCombinedQuery` that contains a `directory-query` structured query and query options that include the `extract-document-data` option. The example creates the combined query from a string literal, but you can also use `StructuredQueryBuilder` to create the query portion of the combined query. For details, see "Creating a Combined Query Using StructuredQueryBuilder" on page 164.

The following example program inserts some documents into the database, performs a search that uses the `extract-document-data` query option, and then deletes the documents. Before running the example, modify the values of `HOST`, `PORT`, `USER`, and `PASSWORD` to match your environment.

```
package com.marklogic.examples;

import org.w3c.dom.Document;

import com.marklogic.client.document.DocumentWriteSet;
import com.marklogic.client.document.GenericDocumentManager;
import com.marklogic.client.io.*;
import com.marklogic.client.query.DeleteQueryDefinition;
import com.marklogic.client.query.ExtractedItem;
import com.marklogic.client.query.ExtractedResult;
import com.marklogic.client.query.MatchDocumentSummary;
import com.marklogic.client.query.QueryManager;
import com.marklogic.client.query.RawCombinedQueryDefinition;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.DatabaseClient;
import com.marklogic.client.DatabaseClientFactory.DigestAuthContext;


public class ExtractExample {
    // replace with your MarkLogic Server connection information
    static String HOST = "localhost";
    static int PORT = 8000;
    static String USER = "username";
    static String PASSWORD = "password";
    static DatabaseClient client = DatabaseClientFactory.newClient(
            HOST, PORT,
            new DigestAuthContext(USER, PASSWORD));
    static String DIR = "/extract/";

    // Insert some example documents in the database.
    public static void setup() {
        StringHandle jsonContent = new StringHandle(
          "{\"parent\": {" +
            "\"a\": \"foo\"," +
            "\"body\": {" +
              "\"target\": \"content1\"" +
            "}," +
            "\"b\": \"bar\"" +
          "}}").withFormat(Format.JSON);
        StringHandle xmlContent = new StringHandle(
          "<parent>" +
            "<a>foo</a>" +
```

```
            "<body><target>content2</target></body>" +
            "<b>bar</b>" +
          "</parent>").withFormat(Format.XML);
        GenericDocumentManager gdm = client.newDocumentManager();

        DocumentWriteSet batch = gdm.newWriteSet();
        batch.add(DIR + "doc1.json", jsonContent);
        batch.add(DIR + "doc2.xml", xmlContent);
        gdm.write(batch);
    }

    // Perform a search with RawCombinedQueryDefinition that extracts
    // just the "target" element or property of docs in DIR.
    public static void example() {
        String rawQuery =
        "<search xmlns=\"http://marklogic.com/appservices/search\">" +
        "  <query>" +
        "    <directory-query><uri>" + DIR + "</uri></directory-query>" +
        "  </query>" +
        "  <options>" +
        "    <extract-document-data selected=\"include\">" +
        "      <extract-path>/parent/body/target</extract-path>" +
        "    </extract-document-data>" +
        "  </options>" +
        "</search>";
        //The equivalent in JSON:
        String rawQueryJson =
        "{\"search\":{" +
                "   \"query\": {" +
                "      \"directory-query\": {" +
                "         \"uri\": \"/extract/\"" +
                "      }" +
                "   }," +
                "   \"options\": {" +
                "      \"extract-document-data\": {" +
                "         \"selected\": \"include\"," +
                "         \"extract-path\": \"/parent/body/target\"" +
                "      }" +
                "   }" +
                "}" +
            "}";


        StringHandle qh =
            new StringHandle(rawQuery).withFormat(Format.XML);
// Or with rawQueryJson
            new StringHandle(rawQueryJson).withFormat(Format.JSON);

        QueryManager qm = client.newQueryManager();
        RawCombinedQueryDefinition query =
            qm.newRawCombinedQueryDefinition(qh);

        SearchHandle results = qm.search(query, new SearchHandle());
```

```
            System.out.println(
                "Total matches: " + results.getTotalResults());

            MatchDocumentSummary matches[] = results.getMatchResults();
            for (MatchDocumentSummary match : matches) {
               System.out.println("Extracted from uri: " + match.getUri());
                ExtractedResult extracts = match.getExtracted();
                for (ExtractedItem extract: extracts) {
                    System.out.println("  extracted content: " +
                        extract.getAs(String.class));
                }
            }
        }

        // Delete the documents inserted by setup.
        public static void teardown() {
            QueryManager qm = client.newQueryManager();
            DeleteQueryDefinition byDir = qm.newDeleteDefinition();
            byDir.setDirectory(DIR);
            qm.delete(byDir);
        }

        public static void main(String[] args) {
            setup();
            example();
            teardown();
        }
    }
```

When you run the example, you should see output similar to the following:

```
Total matches: 2
Extracted from uri: /extract/doc1.json
  extracted content: {"target":"content1"}
Extracted from uri: /extract/doc2.xml
  extracted content: <target xmlns="">content2</target>
```

If you add a second extract path, such as “//b”, then you get multiple extracted items for each matched document:

```
Extracted items from uri: /extract/doc1.json
  extracted content: {"target":"content1"}
  extracted content: {"b":"bar"}
Extracted items from uri: /extract/doc2.xml
  extracted content: <target xmlns="">content2</target>
  extracted content: <b xmlns="">bar</b>
```

By varying the value of the `selected` attribute of `extract-document-data`, you further control how much of the matching content is returned in each `ExtractedItem`. For example, if you modify the original example to set the value of selected to “include-with-ancestors”, then the output is similar to the following:

```
Extracted items from uri: /extract/doc1.json
  extracted content: {"parent":{"body":{"target":"content1"}}}
Extracted items from uri: /extract/doc2.xml
  extracted content:
    <parent xmlns=""><body><target>content2</target></body></parent>
```

For more examples of how `selected` affects the results, see Extracting a Portion of Matching Documents in the *Search Developer's Guide*.

6.0 Query Options

This chapter describes how to use, write, read, and delete *query options*. In the MarkLogic XQuery Search API, a query options object is called an *options node*.

This chapter contains the following sections:

For details on each of the query options, see Appendix: Query Options Reference in the *Search Developer's Guide*. While there are a large number of options, in order to configure your searches properly and build persistent query options, you will need to familiarize yourself with them.

6.1 Using Query Options

Query options let you specify a set of options for search and apply them repeatedly to multiple searches. The individual options can specify the following:

- Define constraints that do not require indexes, such as word, value and element constraints.
- Define constraints that do require indexes, such as collection, field-value, and other range constraints.
- Control search characteristics such as case sensitivity and ordering.
- Extend the search grammar.
- Customize query results including pagination, snippeting, and filtering.

Query options can be persistent or dynamic. Persistent query options are stored on the REST Server and referenced by name in future queries. Dynamic query options are options created on a per-request basis. Choosing between the two is a trade off between flexibility and performance: Dynamic query options are the more flexible, but persistent query options usually provide better performance. You can use both persistent and dynamic query options in the same query. Dynamic query options are only available for operations that accept a `RawCombinedQueryDefinition`. For details, see "Apply Dynamic Query Options to Document Searches" on page 159.

Use a `QueryOptionsManager` object to manage persistent query options and store them on the REST Server. To see individual option values, use the appropriate `get()` command on a handle class that implements `QueryOptionsReadHandle`.

The persistent query options are the static part of the search, and are generally used for many different queries. For example, you might create persistent query options that define a range constraint on a date range index so that you can facet the results by date.

Additionally, many queries have a component that is constructed dynamically by your Java code. For example, you might change the result page, the query criteria (terms, facet values, and so on), or other dynamic parts of the query. The static and dynamic parts of the query are merged together during a search.

For details on specific query options, see Appendix: Query Options Reference in the *Search Developer's Guide*. While there are a large number of options, in order to configure your searches properly, you will need to familiarize yourself with them.

For additional examples, see Query Options Examples in the *Search Developer's Guide*.

6.2 Default Query Options

The MarkLogic Java API comes with predefined persistent query options called `default`. It acts just like any other options and is used if options are not specified elsewhere. You can read it into a handle, change values, and write it back out, where it will still be used as the default query options. While changing its values should not be done casually, this can be very useful if your site needs different default behaviors.

The default options are selected for optimal performance; searches run unfiltered, and document quality is not taken into consideration during scoring. If you install different default options, consider including the following options unless your application requires filtering or the use of document quality.

```
<options xmlns="http://marklogic.com/appservices/search">
  <search-option>unfiltered</search-option>
  <quality-weight>0</quality-weight>
</options>
```

The equivalent in JSON:

```
{"options":{
            "search-option": "unfiltered",
            "quality-weight": "0"
    }
}
```

If you delete `default`, the server will fall back to its own defaults.

6.3 Using QueryOptionsManager To Delete, Write, and Read Options

Interactions with the database are done via a manager object, in this case `QueryOptionsManager`. Use `com.marklogic.client.admin.QueryOptionsManager` to manage persistent query options that are stored on the REST server. Since query options are associated with the REST server configuration, to create a `QueryOptionsManager` you call `ServerConfigManager.newQueryOptionsManager()`.

As with all operations on `ServerConfigManager`, an application must authenticate as `rest-admin`. Note that any application that authenticates as `rest-reader` and `rest-writer` can use query options, but to write or delete them from the server requires `rest-admin`.

```
// create a manager for writing, reading, and deleting query options
QueryOptionsManager qoManager=
            client.newServerConfigManager().newQueryOptionsManager();
```

The simplest `QueryOptions` operation is deleting a stored one:

```
qoManager.deleteOptions("myqueryoptions");
```

To read query options from the database, use a handle object of a class that implements `QueryOptionsReadHandle`. To write query options to the database, use a handle object of a class that implements `QueryOptionsWriteHandle`. The API includes several handle classes that implement these interfaces, including `StringHandle`, `BytesHandle`, `DOMHandle`, and `JacksonHandle`. These interfaces allow you to work with query options as raw strings, XML, and JSON.

The following example reads in the options configuration called `myqueryoptions` from the server, then writes it out again.

```
// read a query option configuration from the database
// qoHandle now contains the query option
// "myqueryoptions"
DOMHandle qoHandle =
    qoManager.readOptions("myqueryoptions", new DOMHandle());

//Or the equivalent with a JacksonHandle
JacksonHandle qoHandle =
    qoManager.readOptions("myqueryoptions", new JacksonHandle());

// write the query option to the database
qoManager.writeOptions("myqueryoptions", qoHandle);
```

You can get a list of all named `QueryOptions` from the server via the `QueryOptionsListHandle` object:

```
QueryManager queryMgr = dbclient.newQueryManager();
QueryOptionsListHandle qolHandle =
    queryMgr.optionsList(new QueryOptionsListHandle());
Set<String> results = qolHandle.getValuesMap().keySet();
```

6.4 Using Query Options With Search

You can customize a query with query options in the following ways:

- Create persistent query options, save them to the REST server with an associated name, and then reference them by name when you construct a query. To use the default query options, omit an options name when you construct the query. The following example creates a string query that uses the options stored as “myoptions”:

```
// Create a string query that uses persistent query options
QueryManager qMgr = new QueryManager();
StringQueryDefinition qDef = qMgr.newStringDefinition("myoptions");
...
qMgr.search(qDef, resultsHandle);
```

- Embed dynamic query options in a combined query definition.

You can use both persistent and dynamic query options in the same search by including a query options name when constructing a combined query (`RawCombinedQueryDefinition`).

Persistent query options must be stored on the REST server before you can use them in a search. For details, see “Using QueryOptionsManager To Delete, Write, and Read Options” on page 192.

To construct persistent query options, use a handle class that implements `QueryOptionsWriteHandle`, such as `StringHandle` or `DOMHandle`. Using a handle, you can create query options directly in XML or JSON; for details, see “Creating Persistent Query Options From Raw JSON or XML” on page 193.

To construct dynamic query options, use a handle that implements `StructureWriteHandle`, such as `StringHandle` or `DOMHandle` to create a combined query that includes an options component, then associate the handle with a `RawCombinedQueryDefinition`. For details, see “Apply Dynamic Query Options to Document Searches” on page 159.

6.5 Creating Persistent Query Options From Raw JSON or XML

To create persistent query options from a raw XML or JSON representation, use any handle class that implements `com.marklogic.client.io.marker.QueryOptionsWriteHandle`. Follow this procedure to create persistent query options using a handle:

1. Create a JSON or XML representation of the query options, using the tools of your choice. The following example uses a String representation:

```
String xmlOptions =
  "<search:options "+
        "xmlns:search='http://marklogic.com/appservices/search'>"+
      "<search:constraint name='industry'>"+
        "<search:value>"+
          "<search:element name='industry' ns=''/>"+
        "</search:value>"+
      "</search:constraint>"+
```

```
  "</search:options>";

String jsonOptions =
        "{\"search\":{" +
                "   \"constraint\": {" +
                "      \"name\": \"industry\"," +
                "      \"value\": {" +
                "         \"element\": {" +
                "            \"name\": \"industry\"," +
                "            \"ns\": \"\"" +
                "         }" +
                "      }" +
                "   }" +
                "}" +
            "}";
```

2. Create a handle that implements `QueryOptionsWriteHandle` and associate your options with the handle. Set the content format type appropriately. For example:

```
// For XML options
StringHandle writeHandle =
    new StringHandle(xmlOptions).withFormat(Format.XML);

// For JSON options
StringHandle writeHandle =
    new StringHandle(jsonOptions).withFormat(Format.JSON);
```

3. Save the options to the REST server using `QueryOptionsManager.writeOptions`. For example:

```
optionsMgr.writeOptions(optionsName, writeHandle);
```

For a complete example, see `com.marklogic.client.example.cookbook.QueryOptions` in the following directory of the Java API distribution:

```
example/com/marklogic/client/example/cookbook
```

The Java API includes `QueryOptionsWriteHandle` implementations that support constructing query options as XML or JSON using several alternatives to `String`. These alternatives include reading from a file (`FileHandle`) or stream (`InputStreamHandle`), and popular abstractions, such as DOM, DOM4J, and JDOM. For details, see the Java API JavaDoc.

You can use any handle that implements `QueryOptionsReadHandle` to fetch previously persisted query options from the REST server. The following example fetches the JSON representation of query options into a `String` object:

```
StringHandle jsonStringHandle = new StringHandle();
jsonStringHandle.setFormat(Format.JSON);

qoManager.readOptions("jsonoptions", jsonStringHandle);
```

6.6 Validating Query Options With setQueryOptionValidation()

Query options can be complex. By default, the server validates query options before writing them out to a database. This takes a small amount of time, but because the query options are usually created once and then persisted, it does not really make a difference.

If you do try to write out an invalid query options and validation is enabled (which is the default), you get a `400` error from the server and a `FailedRequestException` thrown.

If you want to turn validation off, you can do so by calling the following right after you create your `ServerConfigurationManager` object:

```
ServerConfigurationManager.setQueryOptionValidation(false)
```

Note that if validation is disabled and you have query options that turn out to be invalid, your searches will still run, but any invalid options will be ignored. For example, if you define an invalid constraint and then try to use it in a search, the search will run, but the constraint will not be used. The search results will contain a warning in cases where a constraint is not used. You can access those warnings via `SearchHandle.getWarnings()`.

7.0 Working With Semantic Data

This chapter discusses the following topics related to using the Java Client API to load semantic triples, manage semantics graphs, and query semantic data. The following topics are covered:

7.1 Introduction

This chapter focuses on details specific to using the Java Client API for semantic operations. For more details and general concepts, see the *Semantics Developer's Guide*.

The graph management capabilities of the Java Client API enable you to manipulate managed triples stored in MarkLogic. For example, you can create, modify, or delete a graph using a GraphManager. For details, see "Creating and Managing Graphs" on page 198.

You can insert unmanaged triples into MarkLogic using standard document operations. Use the `DocumentManager` interfaces to insert XML or JSON documents with triples embedded in them. Unmanaged triples are indexed and searchable, just like managed triples, but you use typical XML and JSON document permissions and interfaces to control them. Unmanaged triples enable you to store semantic data alongside the content to which it applies. "Example: Loading, Managing, and Querying Triples" on page 209 illustrates the use of an unmanaged triple.

Triples can also be made available for queries through the use of MarkLogic features such as the following. See the listed topics for details.

- Inferencing: Inference in the *Semantics Developer's Guide*.
- TDE templates: Using a Template to Identify Triples in a Document in the *Semantics Developer's Guide*.
- Entity Services modeling: Extending a Model with Additional Facts and Generating a TDE Template in the *Entity Services Developer's Guide*.

You can use the Java Client API to query all types of semantic data using the `SPARQLQueryManager` interface. You can evaluate both SPARQL and SPARQL Update queries. For more details, see "Querying Semantic Triples With SPARQL" on page 204 and "Using SPARQL Update to Manage Graphs and Graph Data" on page 213.

7.2 Overview of Common Semantic Tasks

You can use the SPARQL, semantic query, and semantic graph interfaces of MarkLogic

The following table lists some common tasks related to Semantics, along with the interfaces best suited for completing the task using the Java Client API. For a complete list of interfaces, see *Java Client API Documentation*. All of the following interfaces are in the package `com.marklogic.client.semantics`.

Load semantic triples into a named graph or the default graph without using SPARQL Update.	`GraphManager.write or GraphManager.writeAs` For details, see "Creating or Overwriting a Graph" on page 200.
Manage graphs or graph data with SPARQL Update.	`SPARQLQueryManager.executeUpdate` For details, see "Using SPARQL Update to Manage Graphs and Graph Data" on page 213.
Read a semantic graph from the database.	`GraphManager.read` or `GraphManager.readAs` For details, see "Reading Triples from a Graph" on page 202.

Query semantic data with SPARQL	`SPARQLQueryManager.executeAsk`
	
	`SPARQLQueryManager.executeConstruct`
	
	`SPARQLQueryManager.executeDescribe`
	
	`SPARQLQueryManager.executeSelect`
	
	For details, see "Querying Semantic Triples With SPARQL" on page 204.

Manage graph permissions.	`GraphManager.writePermissions`
	
	`GraphManager.mergePermissions`
	
	`GraphManager.deletePermissions`
	
	For details, see "Managing Permissions" on page 214.

7.3 Creating and Managing Graphs

Use the `GraphManager` interface to perform graph management tasks such as creating, reading, updating, and deleting graphs. This section contains the following topics related to graph management tasks:

7.3.1 GraphManager Interface Summary

The following table summarizes key `GraphManager` methods. For a complete list of methods, see the *Java Client API Documentation*.

`write` `writeAs`	Create or overwrite a graph. If the graph already exists, the effect is the same as removing the graph and then recreating it from the input data. For details, see “Creating or Overwriting a Graph” on page 200.
`read` `readAs`	Retrieve triples from a specific graph. For details, see “Reading Triples from a Graph” on page 202.
`replaceGraphs` `replaceGraphsAs`	Remove triples from all graphs, and then insert the quads in the input data set. Unmanaged triples are not affected. The effect is the same as first calling `GraphManager.deleteGraphs`, and then inserting the quads. For details, see “Replacing Quad Data in Graphs” on page 202.
`merge` `mergeAs`	Add triples to a named graph or the default graph. If the graph does not exist, it is created. For more details, see “Adding Triples to an Existing Graph” on page 202.
`mergeGraphs` `mergeGraphsAs`	Add quads to the graphs specified in the input quad data. Any graphs that do not already exist are created. “Adding Triples to an Existing Graph” on page 202

`delete`	Delete a specific graph. “Deleting a Graph” on page 203.
`deleteGraphs`	Delete all graphs. Unmanaged triples are not affected. For details, see “Deleting a Graph” on page 203.

`writePermissions`
`mergePermissions`
`deletePermissions`
	Manage graph permissions. You can also set graph permissions during write and merge operations. For details, see “Managing Permissions” on page 214.

7.3.2 Creating a GraphManager Object

Operations on graphs, such as loading triples and reading a graph, require a `com.marklogic.client.semantics.GraphManager` object. To create a `GraphManager`, use `DatabaseClient.newGraphManager`.

For example, the following code snippet creates a `DatabaseClient`, and then uses it to create a `GraphManager`.

```
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.DatabaseClient;
import com.marklogic.client.DatabaseClient.DigestAuthContext;
import com.marklogic.client.semantics.GraphManager;
...
DatabaseClient client = DatabaseClientFactory.newClient(
    "localhost", 8000, "myDatabase",
    new DigestAuthContext("username", "password"));
GraphManager gmgr = client.newGraphManager();
```

You do not have to create a new `DatabaseClient` object to create a `GraphManager`. You can re-use any client object previously created by your application that represents the desired connection to MarkLogic.

7.3.3 Specifying the Triple Format

When reading and writing triples, you must specify the triples format MIME type. You can specify the format in the following ways:

- Use the `withMimeType` method on your triples `Handle` to set the format on a per source basis. For example, the following code initializes a `FileHandle` for reading triples in Turtle format:

```
import com.marklogic.client.io.FileHandle;
...
FileHandle fileHandle =
  new FileHandle(new File("example.ttl"))
    .withMimetype(RDFMimeTypes.TURTLE);
```

- Use `GraphManager.setDefaultMimeType` to set a format to be used across all operations performed through a given `GraphManager`. For example, the following code sets the default MIME type to Turtle:

```
import com.marklogic.client.io.FileHandle;
...
GraphManager graphMgr = ...;
graphMgr.setDefaultMimeType(RDFMimeTypes.TURTLE);
```

Setting a default MIME type frees you from setting the MIME type on every triples handle and enables use of the `GraphManager.*As` methods, such as `GraphManager.writeAs` and `GraphManager.readAs`. For example:

```
graphMgr.setDefaultMimeType(RDFMimeTypes.TURTLE);
...
graphMgr.writeAs(
    someGraphURI, new FileHandle(new File("example.ttl")));
```

Set the MIME type to one of the values exposed by the `RDFMimeTypes` class, such as `RDFMimeTypes.RDFJSON` or `RDFMimeTypes.TURTLE`. For more details about triples formats accepted by MarkLogic, see Supported RDF Triple Formats in *Semantics Developer's Guide*.

To learn more about Handles, see "Using Handles for Input and Output" on page 27.

7.3.4 Creating or Overwriting a Graph

Use `GraphManager.write` and `GraphManager.writeAs` to create or overwrite a graph. If a graph already exists with the specified URI, the effect is the same as removing the existing graph and then recreating it f rom the input triple data.

Note that if you use `GraphManager.write` to load quads, any graph URI in a quad is ignored in favor of the graph URI parameter passed into `write`.

For example, the following code loads triples from a file into a graph. For the complete example, see "Example: Loading, Managing, and Querying Triples" on page 209.

```
public static void loadGraph(String filename, String graphURI, String
format) {
    System.out.println("Creating graph " + graphURI);
    FileHandle tripleHandle =
      new FileHandle(new File(filename)).withMimetype(format);
    graphMgr.write(graphURI, tripleHandle);
}
```

Use the following procedure to load semantic triples into a graph in MarkLogic.

1. If you have not already done so, create a `com.marklogic.client.semantics.GraphManager`. as described in “Creating a GraphManager Object” on page 199. For example:

```
GraphManager graphMgr = client.newGraphManager();
```

2. Create a `Handle` associated with the input triples. The `Handle` type depends on the source for your content, such as a file or in-memory data. For example, the following Handle can be used to read triples in Turtle format from a file:

```
FileHandle tripleHandle =
  new FileHandle(new File("example.ttl"));
```

3. If no default triples format is set on your `GraphManager`, specify the triples format for the `Handle`, using the `withMimetype` method. For more details, see “Querying Semantic Triples With SPARQL” on page 204. For example:

```
tripleHandle.withMimeType(RDFMimeTypes.TURTLE);
```

4. Write the triples to MarkLogic using `GraphManager.write` . For example:

   a. To load triples into a named graph, specify the graph URI as the graph URI parameter. For example:

```
graphMgr.write(someGraphURI, tripleHandle);
```

   b. To load triples into the default graph, specify `GraphManager.DEFAULT_GRAPH` as the graph URI parameter. For example:

```
graphMgr.write(GraphManager.DEFAULT_GRAPH, tripleHandle);
```

5. If your application no longer needs to connect to MarkLogic, release the connection resources by calling the `DatabaseClient` object’s `release()` method.

```
client.release();
```

As an alternative to `GraphManager.write`, if you already have triples in an in-memory object, you can use `GraphManager.writeAs` to short circuit explicit creation of a handle. For example:

```
graphManager.setDefaultMimeType(RDFMimeTypes.RDFJSON);
...
Object graphData = ...;
graphMgr.writeAs(someGraphURI, graphData);
```

For more details on this technique, see “Shortcut Methods as an Alternative to Creating Handles” on page 31.

7.3.5 Reading Triples from a Graph

Use `GraphManager.read` or `GraphManager.readAs` to read the contents of a graph in MarkLogic. You must specify the serialization format from the triples, either on the read Handle or the GraphManager; for details, see "Specifying the Triple Format" on page 199.

The following example retrieves the contents of the default graph, in Turtle format and makes the results available to the application through a `StringHandle`:

```
StringHandle triples = graphMgr.read(
    GraphManager.DEFAULT_GRAPH,
    new StringHandle().withMimetype(RDFMimeTypes.TURTLE));
//...work with the triples as one big string
```

For a complete example, see "Example: Loading, Managing, and Querying Triples" on page 209.

7.3.6 Replacing Quad Data in Graphs

Use `GraphManager.replaceGraphs` and `GraphManager.replaceGraphsAs` to remove all quads from all graphs and then insert quad data into the graphs specified in the new quads. Unmanaged triples are not affected by this operation. The end result is the same as first calling `GraphManager.delete` (or `deleteAs`) and then inserting the quads.

The quad data can be in either NQuad or TriG format. Set the MIME type appropriate, as described in "Specifying the Triple Format" on page 199.

The following example adds a single triple in Turtle format to the default graph. This triple is passed via a `StringHandle`.

```
StringHandle quadHandle = new StringHandle()
    .with(someQuadData)
    .withMimetype(RDFMimeTypes.NQUAD);
graphMgr.replaceGraphs(quadHandle);
```

The following example performs a similar operation, using `replaceAs`:

```
graphMgr.setDefaultMimeType(RDFMimeTypes.NQUAD);
...
File graphData = new File(filename);
graphMgr.replaceGraphsAs(graphData);
```

7.3.7 Adding Triples to an Existing Graph

Use `GraphManager.merge` or `GraphManager.mergeAs` to merge triples into an existing graph. You must specify the serialization format from the triples, either on the read Handle or the GraphManager; for details, see "Specifying the Triple Format" on page 199.

For quad data, use `mergeGraphs` or `mergeGraphsAs`. For details, see "Adding Quads into an Existing Graph" on page 203.

The following example adds a single triple in Turtle format to the default graph. This triple is passed via a `StringHandle`.

```
StringHandle stringHandle = new StringHandle()
    .with("<http://example.org/subject2> " +
          "<http://example.org/predicate2> " +
          "<http://example.org/object2> .")
    .withMimetype(RDFMimeTypes.TURTLE);
graphMgr.merge("myExample/graphUri", stringHandle);
```

The following example performs a similar operation, using `mergeAs`:

```
graphMgr.setDefaultMimeType(RDFMimeTypes.TURTLE);
...
Object graphData = ...;
graphMgr.mergeAs(someGraphURI, graphData);
```

7.3.8 Adding Quads into an Existing Graph

Use `GraphManager.mergeGraphs` and `GraphManager.mergeGraphsAs` to add quads to an existing graph. If a quad specifies the URI of an existing graph, the quad data is merged into that graph. If no such graph exists, the graph is created.

The quad data can be in either NQuad or TriG format. Set the MIME type appropriate, as described in “Specifying the Triple Format” on page 199.

The following example adds a single triple in Turtle format to the default graph. This triple is passed via a `StringHandle`.

```
StringHandle quadHandle = new StringHandle()
    .with(someQuadData)
    .withMimetype(RDFMimeTypes.NQUAD);
graphMgr.mergeGraphs(quadHandle);
```

The following example performs a similar operation, using `mergeAs`:

```
graphMgr.setDefaultMimeType(RDFMimeTypes.NQUAD);
...
File graphData = new File(filename);
graphMgr.mergeGraphsAs(graphData);
```

7.3.9 Deleting a Graph

Use GraphManager.delete to remove a single graph. Use GraphManager.deleteGraphs to delete all graphs.

The following example removes a single graph with the specified URI:

```
graphMgr.delete(someGraphURI);
```

The following example removes all graphs:

```
graphMgr.deleteGraphs(someGraphURI);
```

For a complete example, see “Example: Loading, Managing, and Querying Triples” on page 209.

7.4 Querying Semantic Triples With SPARQL

To query semantic data using SPARQL, create a `SPARQLQueryDefinition`, and then evaluate it using one of the `SPARQLQueryManager.execute*` methods. You can configure many aspects of your query, such as variable bindings, the graphs to which to apply the query, the query optimization level.

7.4.1 Basic Steps for SPARQL Query Evaluation

Evaluating a SPARQL query consists of the following basic steps:

1. Create a query manager using `DatabaseClient.newSPARQLQueryManager`. For example:

```
DatabaseClient client = ...;
SPARQLQueryManager sqmgr = client.newSPARQLManager();
```

2. Create a query using `SPARQLQueryManager.newSPARQLQueryDefinition` and configure the query as needed. For example:

```
SPARQLQueryDefinition query = sqmgr.newSPARQLQueryDefinition(
  "SELECT * WHERE { ?s ?p ?o } LIMIT 10")
  .withBinding("o", "http://example.org/object1");
```

3. Evaluate the query and receive results by calling one of the `execute*` methods of `SPARQLQueryManager`. For example, use `executeSelect` for a SELECT query:

```
JacksonHandle results = new JacksonHandle();
results.setMimetype(SPARQLMimeTypes.SPARQL_JSON));
results = sqmgr.executeSelect(query, results);
```

The format of the results you receive depends on the type of query you evaluate and how you configure your results Handle and/or the SPARQLQueryManager. For details, see “Handling Query Results” on page 205.

For example, the following evaluates a SPARQL SELECT query and returns the results as JSON. For a complete example, see “Example: Loading, Managing, and Querying Triples” on page 209.

```
SPARQLQueryManager qm = client.newSPARQLQueryManager();
SPARQLQueryDefinition query = qm.newQueryDefinition(
    "SELECT ?person " +
    "WHERE { ?person <http://example.org/marklogic/predicate/livesIn>
\"London\" }"
);

JsonNode results = qm.executeSelect(query, new JacksonHandle()).get();
// ... Process results
```

7.4.2 Handling Query Results

The layout and available format of the results from a SPARQL query depend on the type of query. For details, see the following topics:

- SELECT Results
- CONSTRUCT and DESCRIBE Results
- ASK Results

7.4.2.1 SELECT Results

A SPARQL SELECT query returns a SPARQL result set, serialized as JSON, XML, or plain text comma-separated values (CSV). You must set the MIME type on your results ReadHandle as appropriate for the results format you want to use. The supported MIME types for a SELECT query are defined by `com.marklogic.client.semantics.SPARQLMimeTypes`.

For example, `JacksonHandle` implements `SPARQLResultsReadHandle` and `JSONReadHandle`, so you should set the handle MIME type to `SPARQLMimeTypes.SPARQL_JSON` when receiving SELECT query results through a `JacksonHandle`:

```
JacksonHandle handle = new JacksonHandle();
handle.setMimeType(SPARQLMimeTypes.SPARQL_JSON);
```

The following table summarizes the supported Handle and MIME type combinations:

`JSONReadHandle`	`SPARQLMimeTypes.SPARQL_JSON`	SPARQL results, serialized as JSON. For details on this format, see https://www.w3.org/TR/sparql11-results-json/.
`XMLReadHandle`	`SPARQLMimeTypes.SPARQL_XML`	SPARQL results, serialized as XML. For details on this format, see https://www.w3.org/TR/2013/REC-rdf-sparql-XMLres-20130321/.
Other Handle types	`SPARQLMimeTypes.SPARQL_CSV`	SPARQL results, serialized as CSV, with line per query solution. For details on this format, see https://www.w3.org/TR/2013/REC-sparql11-results-csv-tsv-20130321/.

For examples of the raw XML, JSON, and CSV results, see the examples in Response Output Formats in the *Semantics Developer's Guide*.

7.4.2.2 CONSTRUCT and DESCRIBE Results

CONSTRUCT and DESCRIBE queries return triples. You can request the results in any of the triples formats defined by `com.marklogic.client.semantics.RDFMimeTypes`, except TRIG. For best performance, use the N-triples format (`RDFMimeTypes.NTRIPLES`).

When using a `JSONReadHandle`, set the handle MIME type to `RDFMimeTypes.RDF_JSON`. This produces results in RDF/JSON format.

When using an `XMLReadHandle`, set the handle MIME type to `RDFMimeTypes.RDF_XML`. This produces results in RDF/XML format.

Any TriplesReadHandle implementation that handle plain text can use any of the RDFMimeTypes, such as `NTRIPLE`, `NQUADS`, or `TURTLE`.

For RDF/XML. use an `XMLReadHandle` with the MIME type set to `RDFMimeTypes.RDF_XML`.

To set the handle MIME type, use the `setMimeType` method. For example:

```
JacksonHandle handle = new JacksonHandle();
handle.setMimeType(RDFMimeTypes.RDFJSON);
```

7.4.2.3 ASK Results

An ASK query always returns a simple boolean value. For details, see `QueryManager.executeAsk`.

7.4.3 Defining Variable Bindings

If your query depends on runtime variable definitions, you can define variable bindings one at a time using the fluent `SPARQLQueryDefinition.withBinding` definition, or build up a set of bindings using `SPARQLBindings` and then attach them to the query using `SPARQLQueryDefinition.setBindings`.

The following example incrementally attaches bindings to a query definition using withBindings:

```
// incrementally attach bindings to a query
SPARQLQueryDefinition query = ...;
query.withBinding("o", "http://example.org/object1")
     .withBinding(...);
```

The following example builds up a setting of bindings and then attaches them all to the query at once, using setBindings:

```
// build up a set of bindings and attach them to a query
SPARQLBindings bindings = new SPARQLBindings();
bindings.bind("o", "http://example.org/object1");
bindings.bind(...);
query.setBindings(bindings);
```

Both `SPARQLQueryDefinition.withBinding` and `SPARQLBindings` enable you to specify a language tag or RDF type for the bound value.

For more details, see SPARQLBindings in the *Java Client API Documentation*.

7.4.4 Limiting the Number of Results

When you evaluate a SPARQL `SELECT` query, by default, all results are returned. You can limit the number of results returned in a "page" using `SPARQLQueryManager.setPageLength` or a SPARQL `LIMIT` clause. You can retrieve successive pages of results by repeatedly calling `executeSelect` with a different page start position. For example:

```
// Change the max page length
sqmgr.setPageLength(NRESULTS);

// Fetch at most the first N results
long start = 1;
JacksonHandle results = sparqlMgr.executeSelect(query, handle, start);

// Fetch the next N results
start += N;
JacksonHandle results = sparqlMgr.executeSelect(query, handle, start);
```

7.4.5 Inferencing Support

Inferencing enables discovery of new "facts" based on a combination of data and rules for understanding that data. The Java Client API includes the following features that facilitate inferencing:

- Enabling or Disabling Automatic Inferencing
- Associating a Rule Set with a Query

7.4.5.1 Enabling or Disabling Automatic Inferencing

When automatic inferencing is enabled, MarkLogic can apply a default inferencing ruleset at query time. Default ruleset management is a function of the REST Management API. However, you can enable or disable the use of a default ruleset at query time using `SPARQLQueryDefinition.withIncludeDefaultRulesets`. Use of the default ruleset is enabled by default.

For more details, see Rulesets in the *Semantics Developer's Guide*.

7.4.5.2 Associating a Rule Set with a Query

You can explicitly apply a ruleset at query time rather than implicitly using the default ruleset.

The Java Client API includes the `com.marklogic.client.semantics.SPARQLRuleset` class with a set of built-in rulesets and a factory method to enable you to use custom rulesets. To associate a ruleset with a query, use `SPARQLQueryDefinition.withRuleset` or `SPARQLQueryDefinition.setRulesets`.

For more details, see Rulesets in the *Semantics Developer's Guide*.

7.5 Querying Triples with the Optic API

The Optic features of the Java Client API enable you to query semantic data without using SPARQL. The Optic features of the Java Client API are covered in detail in "Optic Java API for Relational Operations" on page 218.

To perform a semantic query using the Optic API, construct a query plan using `com.marklogic.client.expression.PlanBuilder`, and then evaluate it using `com.marklogic.client.row.RowManager`. For example, the following function creates a query plan that finds "persons who live in London", based on the data from "Example: Loading, Managing, and Querying Triples" on page 209.

```
public static void opticQuery() {
    RowManager rowMgr = client.newRowManager();
    PlanBuilder p = rowMgr.newPlanBuilder();
    PlanPrefixer predPrefixer =
        p.prefixer("http://example.org/marklogic/predicate/");
    Plan plan =
```

```
            p.fromTriples(
                p.pattern(p.col("person"),
                predPrefixer.iri("livesIn"),
                p.xs.string("London")));

        RowSet<RowRecord> results = rowMgr.resultRows(plan);

        System.out.println("OPTIC: Persons who live in London:");
        for (RowRecord row: results) {
            System.out.println("  " + row.getString("person"));
        }
    }
```

When the above function runs as part of the end to end example, it produces output of the following form:

```
OPTIC: Persons who live in London:
  http://example.org/marklogic/person/Jane_Smith
  http://example.org/marklogic/person/John_Smith
  http://example.org/marklogic/person/Mother_Goose
```

7.6 Example: Loading, Managing, and Querying Triples

The following example program demonstrates how to perform the following tasks:

- Load triples into a graph. These triples become “managed” triples. Use the operations discussed in “Creating and Managing Graphs” on page 198 for graph management.
- Load a document containing a triple. This becomes an unmanaged triple. It is indexed and can be queried, but it is not managed through the graph operations.
- Execute a semantic query using either SPARQL or the Optic API.
- Removed a graph, thereby removing all the managed triples in the graph.

This example creates a graph from the following input data. Copy and paste this data to a file, and then modify the variable `tripleFilename` in the example code to point to this file.

```
<http://example.org/marklogic/person/John_Smith>
<http://example.org/marklogic/predicate/livesIn>
"London"^^<http://www.w3.org/2001/XMLSchema#string> .
<http://example.org/marklogic/person/Jane_Smith>
<http://example.org/marklogic/predicate/livesIn>
"London"^^<http://www.w3.org/2001/XMLSchema#string> .
<http://example.org/marklogic/person/Jack_Smith>
<http://example.org/marklogic/predicate/livesIn>
"Glasgow"^^<http://www.w3.org/2001/XMLSchema#string> .
```

The example data contains triple data that define relationships of the form “Person X livesIn Y”. The query run by the example finds all persons who live in London. The program runs the query several times:

- After loading triples into the default graph. This query matches John Smith and Jack Smith.
- After loading a document containing an unmanaged triple that asserts Mother Goose lives in London. This query matches John Smith, Jack Smith, and Mother Goose.
- After removing the default graph. Only the unmanaged triple remains, so the query matches only Mother Goose.
- After removing the document containing the unmanaged triple. No matches are found.

The example code demonstrates how to use a SPARQL query and an Optic query to fetch the same information. For more details, see “Querying Semantic Triples With SPARQL” on page 204 and “Querying Triples with the Optic API” on page 208.

Before running the following program, modify the definition of the `client` and `tripleFilename` variables to match your environment.

```
package examples;

import java.io.File;

import com.fasterxml.jackson.databind.JsonNode;
import com.marklogic.client.DatabaseClient;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.document.JSONDocumentManager;
import com.marklogic.client.expression.PlanBuilder;
import com.marklogic.client.expression.PlanBuilder.Plan;
import com.marklogic.client.io.FileHandle;
import com.marklogic.client.io.Format;
import com.marklogic.client.io.JacksonHandle;
import com.marklogic.client.io.StringHandle;
import com.marklogic.client.row.RowManager;
import com.marklogic.client.row.RowRecord;
import com.marklogic.client.row.RowSet;
import com.marklogic.client.semantics.GraphManager;
import com.marklogic.client.semantics.RDFMimeTypes;
import com.marklogic.client.semantics.SPARQLQueryDefinition;
import com.marklogic.client.semantics.SPARQLQueryManager;
import com.marklogic.client.type.PlanPrefixer;

public class Graphs {
    static DatabaseClient client = DatabaseClientFactory.newClient(
        "localhost", 8000, "Documents",
        new DatabaseClientFactory.DigestAuthContext(
            "username", "password"));
    static private GraphManager graphMgr = client.newGraphManager();
    static private String graphURI = GraphManager.DEFAULT_GRAPH;
```

```
    static private String tripleFilename = "/path/to/your/file.ttl";
    static private String unmanagedTripleDocURI = "mothergoose.json";

    // Load managed triples from a file into a graph in MarkLogic
    public static void loadGraph(String filename, String graphURI, String
format) {
        System.out.println("Creating graph " + graphURI);
        FileHandle tripleHandle =
            new FileHandle(new File(filename)).withMimetype(format);
        graphMgr.write(graphURI, tripleHandle);
    }

    // Insert a document that includes an unmanaged triple.
    public static void addUnmanagedTriple() {
        System.out.println("Inserting doc containing an unmanaged triple...");
        StringHandle contentHandle = new StringHandle(
            "{ \"name\": \"Mother Goose\"," +
                "\"triple\" : {" +
                    "\"subject\" :
\"http://example.org/marklogic/person/Mother_Goose\"," +
                    "\"predicate\" :
\"http://example.org/marklogic/predicate/livesIn\"," +
                    "\"object\" : {" +
                      "\"value\" : \"London\"," +
                      "\"datatype\" :
\"http://www.w3.org/2001/XMLSchema#string\"" +
            "} }  }").withFormat(Format.JSON);
        JSONDocumentManager jdm = client.newJSONDocumentManager();
        jdm.write(unmanagedTripleDocURI, contentHandle);
    }

    public static void deleteUnmanagedTriple() {
        System.out.println("Removing doc containing unmanaged triple...");
        JSONDocumentManager jdm = client.newJSONDocumentManager();
        jdm.delete(unmanagedTripleDocURI);
    }

    public static void readGraph(String graphURI, String format) {
        System.out.println("Reading graph " + graphURI);
        StringHandle triples =
            graphMgr.read(graphURI, new StringHandle().withMimetype(format));
        System.out.println(triples);
    }

    // Delete a graph. Unmmanaged triples are unaffected.
    public static void deleteGraph(String graphURI) {
        System.out.println("Deleting graph " + graphURI);
        graphMgr.delete(graphURI);
    }

    // Evaluate a SPARQL query.
    public static void sparqlQuery() {
        SPARQLQueryManager qm = client.newSPARQLQueryManager();
        SPARQLQueryDefinition query = qm.newQueryDefinition(
```

```
            "SELECT ?person " +
            "WHERE { ?person
<http://example.org/marklogic/predicate/livesIn> \"London\" }"
        );

        JsonNode results =
            qm.executeSelect(query, new JacksonHandle()).get();
        JsonNode matches = results.path("results").path("bindings");
        System.out.println("SPARQL: Persons who live in London:");
        for (int i = 0; i < matches.size(); i++) {
            String subject =
                matches.get(i).path("person").path("value").asText();
            System.out.println("  " + subject);
        }
    }

    public static void opticQuery() {
        RowManager rowMgr = client.newRowManager();
        PlanBuilder pb = rowMgr.newPlanBuilder();
        PlanPrefixer predPrefixer =
            pb.prefixer("http://example.org/marklogic/predicate/");
        Plan plan = pb.fromTriples(
            pb.pattern(
                pb.col("person"),
                predPrefixer.iri("livesIn"),
                pb.xs.string("London")));

        RowSet<RowRecord> results = rowMgr.resultRows(plan);
        System.out.println("OPTIC: Persons who live in London:");
        for (RowRecord row: results) {
            System.out.println("  " + row.getString("person"));
        }
    }

    public static void main(String[] args) {
        loadGraph(tripleFilename, graphURI, RDFMimeTypes.TURTLE);
        readGraph(graphURI, RDFMimeTypes.TURTLE);

        // Query the graph for persons who live in London.
        // Should find 2 matches.
        sparqlQuery();

        // Add a document containing an unmanaged triple. Query again.
        // Should find 3 matches.
        addUnmanagedTriple();
        sparqlQuery();

        // Perform the same query using the Optic API
        opticQuery();

        // Delete the created graph. Unmanaged triple remains.
        // Query should find 1 match.
        deleteGraph(graphURI);
        sparqlQuery();
```

```
            // Remove the document containing the unmanaged triple.
            // Query should find no matches.
            deleteUnmanagedTriple();
            sparqlQuery();

            client.release();
        }
    }
```

When you run the example, you should see output similar to the following. Whitespace has been added to the output to more easily distinguish between the operations.

```
Creating graph com.marklogic.client.semantics.GraphManager.DEFAULT_GRAPH
@prefix p1: <http://example.org/marklogic/predicate/> .
@prefix p0: <http://example.org/marklogic/person/> .
p0:Jane_Smith   p1:livesIn      "London" .
p0:Jack_Smith   p1:livesIn      "Glasgow" .
p0:John_Smith   p1:livesIn      "London" .

SPARQL: Persons who live in London:
  http://example.org/marklogic/person/Jane_Smith
  http://example.org/marklogic/person/John_Smith
  http://example.org/marklogic/person/Mother_Goose

Inserting a document containing an unmanaged triple...
SPARQL: Persons who live in London:
  http://example.org/marklogic/person/Jane_Smith
  http://example.org/marklogic/person/John_Smith
  http://example.org/marklogic/person/Mother_Goose

OPTIC: Persons who live in London:
  http://example.org/marklogic/person/Jane_Smith
  http://example.org/marklogic/person/John_Smith
  http://example.org/marklogic/person/Mother_Goose

Deleting graph com.marklogic.client.semantics.GraphManager.DEFAULT_GRAPH
SPARQL: Persons who live in London:
  http://example.org/marklogic/person/Mother_Goose

Removing document containing unmanaged triple...
SPARQL: Persons who live in London:
```

7.7 Using SPARQL Update to Manage Graphs and Graph Data

You can use SPARQL Update to insert, update, or delete triples and graphs, as an alternative to the graph management interface described in “Creating and Managing Graphs” on page 198. You cannot use SPARQL Update to operate on unmanaged triples.

To learn about SPARQL Update, see SPARQL Update in the *Semantics Developer’s Guide*.

To use SPARQL Update with the Java Client API, follow the same procedure as for SPARQL query, but initialize your `SPARQLQueryDefinition` with update rather than query code, and use `SPARQLQueryManager.executeUpdate` to evaluate the update. For details, see “Basic Steps for SPARQL Query Evaluation” on page 204.

For example, the following code adds a single triple to the default graph. You can add this function to the example framework in “Example: Loading, Managing, and Querying Triples” on page 209 to experiment with SPARQL Update.

```
public static void sparqlUpdate() {
    SPARQLQueryManager qm = client.newSPARQLQueryManager();
    SPARQLQueryDefinition query = qm.newQueryDefinition(
        "PREFIX exp: <http://example.org/marklogic/people/>" +
        "PREFIX pre: <http://example.org/marklogic/predicate/>" +
        "INSERT DATA {" +
        "  exp:Humpty_Dumpty pre:livesIn \"London\" ." +
        "  }"
    );
    System.out.println("Inserting a triple using SPARQL Update");
    qm.executeUpdate(query);
}
```

A successful SPARQL Update returns no results.

You can bind variables to values using the procedure described in “Defining Variable Bindings” on page 207.

7.8 Managing Permissions

Permissions on semantic data are managed at either the graph or document level, depending on whether the triples are managed or unmanaged. Querying and reading semantic data requires read permissions on either the containing graph (managed triples) or document (unmanaged triples).

This section covers the following topics related to controlling permissions on semantic data:

- Default Graph Permissions and Required Privileges
- Setting Graph Permissions
- Retrieving Graph Permissions
- Managing Permissions on Unmanaged Triples

7.8.1 Default Graph Permissions and Required Privileges

All graphs created and managed using the Java, REST, or Node.js Client APIs grant “read” capability to the `rest-reader` role and “update” capability to the `rest-writer` role. These default permissions are always assigned to a graph, even if you do not explicitly specify them.

If you explicitly specify other permissions when creating a graph, the default permissions are still set, in addition to the permissions you specify.

You can use custom roles to limit access to selected users on a graph by graph basis. Your custom roles must include equivalent `rest-reader` and `rest-writer` privileges. Otherwise, users with these roles cannot use the Java Client API to manage or query semantic data. For details, see Controlling Access to Documents and Other Artifacts in the *REST Application Developer's Guide*.

For more information on the MarkLogic security model, see the *Security Guide*.

7.8.2 Setting Graph Permissions

When you create a graph with the Java Client API, MarkLogic assigns a set of default permissions, even if you do not specify any explicit permissions; for details, see "Default Graph Permissions and Required Privileges" on page 214. You can modify permissions on a graph as a standalone operation or as part of another operation, such as when creating or merging graphs.

Graph permissions are encapsulated in a `GraphPermissions` object. To create a set of graph permissions, use `GraphManager.newGraphPermissions` or `GraphManager.permission`.

To modify permissions standalone, use the following `GraphManager` methods:

- `GraphManager.writePermissions`
- `GraphManager.mergePermissions`
- `GraphManager.deletePermissions`

To modify permissions as part of another operation, such as `GraphManager.write` or `GraphManager.merge`, include a `GraphPermissions` object in your call.

The following example sets the permissions on the graph with URI "myExample/graphUri". The code grants the role "example_manager" read and update permissions on the graph.

```
graphMgr.writePermissions("myExample/graphUri",
    graphMgr.permission("example_manager", Capability.READ)
            .permission("example_manager", Capability.UPDATE));
```

The following example sets the graph permissions as part of a graph merge operation:

```
graphMgr.merge(
  "myExample/graphUri", someTriplesHandle,
  graphManager.permission("role1", Capability.READ)
              .permission("role2", Capability.READ, Capability.UPDATE));
```

7.8.3 Retrieving Graph Permissions

To retrieve permissions metadata for a named graph or the default graph, use `GraphManager.getPermissions`. Explore the resulting `GraphPermissions` object using `Map` operations. The Map keys are role names, such as “rest-reader”, and the values are the capabilities.

For example, the following code fetches the permissions on the default graph and prints the capabilities associated with the “rest-reader” role:

```
GraphPermissions permissions =
     graphMgr.getPermissions(GraphManager.DEFAULT_GRAPH);
System.out.println(permissions.get("rest-reader");
```

7.8.4 Managing Permissions on Unmanaged Triples

Unmanaged triples are stored in documents, alongside other content, rather than being inserted into the triple store. You control access to unmanaged triples through the permissions on the documents that contain them.

For example, a SPARQL query will only return a matching unmanaged triple if the user running the query has read permissions on the document that contains the triple.

Permissions are considered document metadata. Set permissions using the `DocumentManager` interface and a `MetadataHandle`. For example, set permissions using `DocumentMetadataHandle.setPermissions`, and then including the metadata handle in a call to `DocumentManager.write`. For more details, see “Reading, Modifying, and Writing Metadata” on page 43.

For more information document permissions, see Protecting Documents in the *Security Guide*.

8.0 Optic Java API for Relational Operations

The MarkLogic Optic API is implemented in JavaScript, XQuery, REST, and Java. A general overview and the JavaScript and XQuery implementations of the Optic API is described in Optic API for Multi-Model Data Access in the *Application Developer's Guide*. This chapter describes the Java Client implementation of the Optic API, which is very similar in structure to the JavaScript version of the Optic API.

This chapter has the following main sections:

8.1 Overview

The Optic Java Client API provides classes to support building a plan on the client, executing the plan on the server, and processing the response on the client.

On the server, the Optic API can yield a row that satisfies any of several common use cases:

- A traditional flat list of atomic values with names and XML Schema atomic datatypes.
- A dynamic JSON or XML document with substructure and leaf atomic values or mixed text.
- An envelope with out-of-band metadata properties and relations for a list of documents.

The second use case can take advantage of document joins or constructor expressions. The third use case can take advantage of document joins.

On the client, the Optic Java Client API can consume a set of rows in one of the following ways:

- As a single CSV, JSON, or XML payload with all of the rows in the set.
- By iterating over each row with a Java map key-value interface, a pre-defined Plain Old Java Object (POJO) tree structure, or a JSON or XML document structure.

A structured value in a format alien to the response format is encoded as a string. In particular, when getting a CSV payload, a JSON or XML column value is encoded as a string. Similarly, when getting a JSON payload or row, an XML value is encoded as a string (and vice versa).

8.2 Getting Started

The Optic Java Client communicates with a REST App Server on MarkLogic.

1. Download the MarkLogic Java Client API to your client system and configure, as described in “Getting Started” on page 14.

2. You can use the preconfigured REST App Server at port 8000, as described in “Choose a REST API Instance” on page 15, however it is generally better that you create your own REST App Server. You can use the `POST:/v1/rest-apis` call to quickly and conveniently create a REST App Server. For example, to create a REST App Server, named `Optic`, on a server named `MLserver`, you can simply enter:

   ```
   curl -X POST --anyauth -u admin:admin -H
   "Content-Type:application/json" \
   -d '{
      "rest-api": {"name": "Optic"}
   }' \
   http://MLserver:8002/v1/rest-apis
   ```

   The `Optic` App Server will be assigned an unused port number and all of the required forests and databases will be created for it. The `Optic` database created for you will use the default `Schemas` database. However, you should create a unique schemas database and assign it to the `Optic` database.

To run the examples described in this chapter, do the following:

1. Follow the steps in Load the Data in the *SQL Data Modeling Guide* to load the sample documents into the database. Use the database associated with your REST API instance (`Optic`) rather than the one used in the procedure.

2. Follow the stops in Create Template Views in the *SQL Data Modeling Guide* to create views and insert the template view documents into the schema database assigned to the `Optic` database.

8.3 Java Packages

The following packages implement the Optic features in the Java API:

Package	Description
`com.marklogic.client.expression`	Provides classes for building Optic plan pipelines and expressions for execution on the REST server.
`com.marklogic.client.row`	Provides classes for sending plan requests to and processing row responses from the REST server.
`com.marklogic.client.type`	Provides interfaces that specify the type of an expression or value passed to a `PlanBuilder` method or returned from a `RowRecord` method.

See the MarkLogic Java API JavaDoc reference for details.

8.4 Structure of the Java Optic API

The Java Optic API is similar to the server-side JavaScript and XQuery implementations of the Optic API described in Optic API for Multi-Model Data Access in the *Application Developer's Guide*. This chapter describes the Java Client implementation of the Optic API, which is similar in structure.

The Optic API for Multi-Model Data Access chapter in the *Application Developer's Guide* contains the following main topics of interest to Java Optic developers:

- Objects in an Optic Pipeline
- Data Access Functions
- Kinds of Optic Queries
- Expression Functions For Processing Column Values
- Functions Equivalent to Boolean, Numeric, and String Operators
- Node Constructor Functions
- Best Practices and Performance Considerations
- Optic Execution Plan
- Parameterizing a Plan
- Exporting and Importing a Serialized Optic Query

8.4.1 Values and Expressions

The *Val interfaces represent client-side values typed with server data types. For example, the `PlanBuilder.xs.decimal` method constructs a client value with an `xs.decimal` data type.

The *Expr interfaces represent server expressions typed with server data types. For example, the `PlanBuilder.fn.formatNumber` method constructs a server expression to format the result of a numeric expression as a string expression.

Server expressions executed to produce the boolean expression for a `where` operation or the expression assigned to a column by the `PlanBuilder.as` function can take columns as arguments. The function call calculates the result of the expression for each row using the values of the columns in the row. For example, if the first argument to `PlanBuilder.fn.formatNumber` is a column, the formatted string will be produced for each row with the value of the column. The column must have a value in each row with the data type required in the expression.

The API provides some overloads for typical literal arguments of expression functions as a convenience.

The `com.marklogic.client.type` package has the marker interfaces for the server data types.

8.4.2 Items and Sequences

Some functions can take multiple values or expressions for a parameter. Such parameters have a sequence data type. A sequence data type can take either a single item of the data type or a sequence of the data type. The API provides constructor functions that take a varargs of items of the appropriate data type and return the sequence.

For instance, `PlanBuilder.pattern` takes a sequence for the subject, predicate, and object parameters.

The call can pass either one `PlanTriplePosition` instance (an `XsAnyAtomicTypeVal`, `PlanColumn`, or `PlanParamExpr` object) as the subject or use `PlanBuilder.subject` to construct a sequence of such objects to pass as the subject.

8.4.3 Atomic Values and Nodes in RowRecord

`RowRecord` provides the `getKind` metadata method for discovering the `ColumnKind` of a column in the current row (ATOMIC_VALUE, CONTENT, or NULL).

For an ATOMIC_VALUE column, the `getDatatype` metadata method reports the atomic data type.

You can call a `get*` getter to cast the value to the appropriate primitive or to a *Val type.

For a CONTENT column, the `getContentFormat` and `getContentMimetype` metadata methods report the format and mime type. The caller can pass the appropriate handle to the `getContent` getter to read the JSON, XML, binary, or text content (consistent with the Java API elsewhere).

8.5 Examples

The following two examples are based on documents and template views described in the Creating Template Views chapter in the *SQL Data Modeling Guide*.

List all of the employees in order of ID number.

```
package Optic;

import com.marklogic.client.DatabaseClient;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.io.StringHandle;
import com.marklogic.client.expression.PlanBuilder;
import com.marklogic.client.expression.PlanBuilder.ModifyPlan;
import com.marklogic.client.row.RowManager;

public class optic4 {
   public static void main(String[] args) {
      DatabaseClient db = DatabaseClientFactory.newClient(
```

```
            "localhost", 8000,
            new DatabaseClientFactory.DigestAuthContext("admin", "admin")
        );

        RowManager rowMgr = db.newRowManager();
        PlanBuilder p = rowMgr.newPlanBuilder();

        ModifyPlan plan = p.fromView("main", "employees")
            .select("EmployeeID", "FirstName", "LastName")
            .orderBy("EmployeeID")
            .offsetLimit(0, 25);

        System.out.println(
            rowMgr.resultDoc(plan,
            new StringHandle().withMimetype("text/csv")).get()
            );

        return;
    }
}
```

Return the ID and full name for the employee with an EmployeeID of 3.

```
package Optic;

import com.marklogic.client.DatabaseClient;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.io.StringHandle;
import com.marklogic.client.expression.PlanBuilder;
import com.marklogic.client.expression.PlanBuilder.ModifyPlan;
import com.marklogic.client.row.RowManager;
import com.marklogic.client.type.XsIntVal;

public class optic {

   public static void main(String[] args) {
      DatabaseClient db = DatabaseClientFactory.newClient(
         "MLserver", 8000,
         new DatabaseClientFactory.DigestAuthContext("admin", "admin")
      );

      RowManager rowMgr = db.newRowManager();
          PlanBuilder p = rowMgr.newPlanBuilder();
          XsIntVal EmployeeID = p.xs.intVal(3);

      ModifyPlan plan = p.fromView("main", "employees")
          .where(p.eq(p.col("EmployeeID"), EmployeeID))
          .select("EmployeeID", "FirstName", "LastName")
          .orderBy("EmployeeID");

      System.out.println(
          rowMgr.resultDoc(plan,
          new StringHandle().withMimetype("text/csv")).get()
```

```
            );

        return;
        }
}
```

The following example returns a list of the people who were born in Brooklyn in the form of a table with two columns, `person` and `name`. This is executed against the example dataset described in Loading Triples in the *Semantics Developer's Guide*. This example is the Java equivalent of the last JavaScript example described in fromView Examples in the Optic API for Multi-Model Data Access chapter in the *Application Developer's Guide*.

```
package Optic;

import com.marklogic.client.DatabaseClient;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.DatabaseClientFactory.DigestAuthContext;
import com.marklogic.client.io.StringHandle;
import com.marklogic.client.expression.PlanBuilder;
import com.marklogic.client.row.RowManager;
import com.marklogic.client.type.PlanColumn;

public class optic2 {

    public static void main(String[] args) {
      DatabaseClient db = DatabaseClientFactory.newClient(
          "localhost", 8000,
          new DigestAuthContext("admin", "admin")
      );
      RowManager rowMgr = db.newRowManager();
      PlanBuilder p = rowMgr.newPlanBuilder();

      PlanBuilder.Prefixer foaf =
          p.prefixer("http://xmlns.com/foaf/0.1");
      PlanBuilder.Prefixer onto =
          p.prefixer("http://dbpedia.org/ontology");
      PlanBuilder.Prefixer resource =
          p.prefixer("http://dbpedia.org/resource");

      PlanColumn person = p.col("person");

      PlanBuilder.QualifiedPlan plan = p.fromTriples(
        p.pattern(person, onto.iri("birthPlace"),
                  resource.iri("Brooklyn")),
        p.pattern(person, foaf.iri("name"), p.col("name"))
      );

      System.out.println(
          rowMgr.resultDoc(plan,
          new StringHandle().withMimetype("text/csv")).get()
      );

      return;
```

```
        }
    }
```

9.0 POJO Data Binding Interface

You can use the Java Client API to persist POJOs (Plain Old Java Objects) as documents in a MarkLogic database. This feature enables you to apply the rich MarkLogic Server search and data management features to the Java objects that represent your application domain model without explicitly converting your data to documents.

This chapter includes the following topics:

9.1 Data Binding Interface Overview

The data binding feature of the Java Client API enables your data to flow seamlessly between application-level Java objects and JSON documents stored in a MarkLogic server. With the addition of minimal annotations to your class definitions, you can store POJOs in the database, search them with the full power of MarkLogic Server, and recreate POJOs from the stored objects.

The Java Client API data binding interface uses the data binding capabilities of Jackson to convert between Java objects and JSON. You can leverage Jackson annotations to fine tune the representation of your objects in the database, but generally you should not need to. Not all Jackson annotations are compatible with the Java Client API data binding capability. For details, see “Limitations of the Data Binding Interface” on page 227.

The data binding capabilities of the Java Client API are primarily exposed through the `com.marklogic.client.pojo.PojoRepository` interface. To get started with data binding, follow these basic steps:

- For each Java class you want to bind to a database representation, add source code annotations to your class definition that call out the Java property to be used as the object id.
- Use a `PojoRepository` to save your objects in the database. You can create, read, update, and delete persisted objects.
- Search your object data using a string (`StringQueryDefinition`) or structured query (`PojoQueryDefinition`). You can use search to identify and retrieve a subset of the stored POJOs.

The object id annotation is required. Additional annotations are available to support more advanced features, such as identifying properties on which to create database indexes and latitude and longitude identifiers for geospatial search. For details, see "Annotating Your Object Definition" on page 227.

9.2 Limitations of the Data Binding Interface

You should be aware of the following restrictions and limitations of the data binding feature:

- The Data Bind interface is intended for use in situations where the in-database representation of objects is not as important as using a POJO-first Java API.

  If you have strict requirements for how your objects must be structured in the database, use `JacksonDatabindHandle` with `JSONDocumentManager` and `StructuredQueryBuilder` instead of the Data Binding interface.

- You can only persist and restore objects of consistent type.

  That is, if you persist objects of type T, you must restore them and search them as type T. For example, you cannot persist an object as type T and then restore it as a some type T' that extends T, or vice versa.

- You cannot use the data binding interface with classes that contain inner classes.
- The object property you chose as the object id must not contain values that do not form valid database URIs when serialized. You should choose object properties that have atomic type, such as `Integer`, `String`, or `Float`, rather than a complex object type such as `Calendar`.
- Though the Java Client API uses Jackson to convert between POJOs and JSON, not all Jackson features are compatible with the Java Client API data binding capability. For example, you can add Jackson annotations to your POJOs that result in objects not being persisted or restored properly.

9.3 Annotating Your Object Definition

The data binding interface in the Java Client API is driven by simple annotations in your class definitions. Annotations are of the form `@annotationName`. You can attach an annotation to a public class field or a public getter or setter method.

Every bound class requires at least an `@Id` annotation to define the object property that holds the object id. A bound POJO class must contain exactly one `@Id` annotation. Each object must have a unique id.

Additional, optional annotations support powerful search features such as range and geospatial queries.

For example, the following annotation says the object id should be derived from the getter `MyClass.getMyId`. If you rely on setters and getters for object identity, your setters and getters should follow the Java Bean convention.

```
import com.marklogic.client.pojo.annotation.Id;
public class MyClass {
  Long myId;

  @Id
  public Long getMyId() {
    return myId;
  }
}
```

Alternatively, you can associated @Id with a member. The following annotation specifies that the `myId` member holds the object id for all instances of `myClass`:

```
import com.marklogic.client.pojo.annotation.Id;
public class MyClass {
  @Id
  public Long myId;
}
```

Annotations can be associated with a member, a getter or a setter because an annotation decorates a logical property of your POJO class.

The following table summarizes the supported annotations. For a complete list, see `com.marklogic.pojo.annotation` in the JavaDoc.

Annotation	Description
`@Id`	The object identifier. The value in the `@Id` property or the value returned by the `@Id` method is used to generate a unique database URI for each persistent object of the class. Each object must have a unique id. Each POJO class may have only one `@Id`.

Annotation	Description
`@PathIndexProperty`	Identifies a property for which a path range index is required. Any property on which you perform range queries must be indexed. For details, see “Creating Indexes from Annotations” on page 236.
`@GeospatialLatitude`	Identifies the property that contains the geospatial latitude coordinate value, in support of a geospatial element pair index. For details, see “Creating Indexes from Annotations” on page 236.
`@GeospatialLongitude`	Identifies the property that contains the geospatial longitude coordinate value, in support of a geospatial element pair index. For details, see “Creating Indexes from Annotations” on page 236.
`@GeoSpatialPathIndexProperty`	Identifies a property for which a geospatial point path range index is required. Any property on which you perform geospatial point queries must be indexed. For details, see “Creating Indexes from Annotations” on page 236.

9.4 Saving POJOs in the Database

Use `PojoRepository.write` to insert or update POJOs in a MarkLogic database. Your POJO class definition must include at least an `@Id` annotation and each object must have a unique id.

The class whose objects you want to persist must be serializable by Jackson. For details, see “Testing Your POJO Class for Serializability” on page 249.

Use the following procedure to persist POJOs in the database:

1. Ensure the class you want to work with includes at least an `@Id` annotation, as described in “Annotating Your Object Definition” on page 227.

1. If you have not already done so, create a `com.marklogic.client.DatabaseClient` object. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

2. Create a `PojoRepository` object associated with the class you want to bind. For example, if you want to bind the class named `MyClass` and the `@Id` annotation in `MyClass` identifies a field or method return type of type `Long`, create a repository as follows:

```
PojoRepository myClassRepo =
    client.newPojoRepository(MyClass.class, Long.class);
```

3. Call `PojoRepository.write` to save objects to the database. For example:

```
MyClass obj = new MyClass();
myClass.setId(42);

myClassRepo.write(obj);
```

4. When you are finished with the database, release the connection.

```
client.release();
```

For a working example, see "Example: Saving and Restoring POJOs" on page 231.

To load POJOs from the database into your application, use `PojoRepository.read` or `PojoRepository.search`. For details, see "Retrieving POJOs from the Database By Id" on page 230 and "Searching POJOs in the Database" on page 232

9.5 Retrieving POJOs from the Database By Id

Use `PojoRepository.read` to load POJOs from the database into your application. You should only use `PojoRepository.read` on objects created using `PojoRepository.write`.

Use the following procedure to load POJOs from the database by object id:

1. Ensure the class you want to work with includes at least an `@Id` annotation , as described in "Annotating Your Object Definition" on page 227.

2. If you have not already done so, create a `com.marklogic.client.DatabaseClient` object. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

3. Create a `PojoRepository` object associated with the class you want to work with. For example, if you want to restore objects of the class named `MyClass` and the `@Id` annotation in `MyClass` identifies a field or method return type of type `Long`, create a repository as follows:

```
PojoRepository myClassRepo =
    client.newPojoRepository(MyClass.class, Long.class);
```

4. Call `PojoRepository.read` to restore one or more objects from the database. For example:

```
MyClass obj = myClassRepo.read(42);

PojoPage<MyClass> objs = myClassRepo.read(new Long[] {1,3,5});
```

5. When you are finished with the database, release the connection.

```
client.release();
```

For a working example, see “Example: Saving and Restoring POJOs” on page 231.

To restore POJOs from the database using criteria other than object id, see “Searching POJOs in the Database” on page 232.

9.6 Example: Saving and Restoring POJOs

The following example saves several objects of type `MyType` to the database, recreates them as POJOs by reading them by id from the database, and then prints out the contents of the restored objects.

The objects are written to the database by calling `PojoRepository.write` and read back using `PojoRepository.read`. In this example, the objects are read back by id. You can retrieve objects by searching for a variety of object features. For details, see “Searching POJOs in the Database” on page 232.

```
package examples;

import com.marklogic.client.DatabaseClient;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.DatabaseClientFactory.DigestAuthContext;
import com.marklogic.client.pojo.PojoPage;
import com.marklogic.client.pojo.PojoRepository;
import com.marklogic.client.pojo.annotation.Id;


public class PojoExample {
  private static DatabaseClient client =
DatabaseClientFactory.newClient(
    "localhost", 8000, new DigestAuthContext(user, password));

  // The POJO class
  static public class MyClass {
    Integer myId;
    String otherData;

    public MyClass() { myId = 0; otherData = ""; }
    public MyClass(Integer id) { myId = id; otherData = ""; }
    public MyClass(Integer id, String data) {
      myId = id; otherData = data;
    }

    @Id
    public int getMyId() { return myId; }
    public void setMyId(int id) { myId = id; }

    public String getOtherData() { return otherData; }
```

```
    public void setOtherData(String data) { otherData = data; }

    public String toString() {
      return "myId=" + getMyId() + " " +
          "otherData=\"" + getOtherData() + "\"";
    }
  }

  static void tryPojos() {
    PojoRepository<MyClass,Integer> repo =
        client.newPojoRepository(MyClass.class, Integer.class);
    Integer ids[] = {1, 2, 3};
    String data[] = {"a", "b", "c"};

    // Save objects in the database
    for (int i = 0; i < ids.length; i++) {
      repo.write(new MyClass(ids[i], data[i]));
    }

    // Restore objects from the database by id
    PojoPage<MyClass> outputObjs = repo.read(ids);
    while (outputObjs.hasNext()) {
      MyClass obj = outputObjs.next();
      System.out.println(obj.toString());
    }
  }

  public static void main(String[] args) {
    tryPojos();
    client.release();
  }
}
```

9.7 Searching POJOs in the Database

You can use `PojoRepository.search` to search POJOs in the database that match a query. A rich set of query capabilities is available, including full text search using a simple string query grammar and more finely controllable search using structured query.

This section covers concept and procedural information on searching POJOs. For a complete example, see “Example: Searching POJOs” on page 240.

This section covers the following topics:

- Basic Steps for Searching POJOs
- Full Text Search with String Query
- Search Using Structured Query
- How Indexing Affects Searches
- Creating Indexes from Annotations

9.7.1 Basic Steps for Searching POJOs

This section describes the basic process for searching POJOs. The variations are in how you express your search criteria.

> **Note:** You should only use `PojoRepository.search` on objects created using `PojoRepository.write`. Using it to search JSON documents created in a different way can lead to errors.

1. Ensure the class you want to work with includes at least an `@Id` annotation, as described in "Annotating Your Object Definition" on page 227.

2. If you have not already done so, create a `com.marklogic.client.DatabaseClient` object. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

3. Create a `PojoRepository` object associated with the class you want to work with. For example, if you want to restore objects of the class named `MyClass` and the `@Id` annotation in `MyClass` identifies a field or method return type of type `Long`, create a repository as follows:

```
PojoRepository<MyClass, Long> myClassRepo =
    client.newPojoRepository(MyClass.class, Long.class);
```

4. Optionally, set the limit on the number of matching objects to return. The default is 10 objects.

```
myClassRepo.setPageLength(5);
```

5. Create a `StringQueryDefinition` or `StructuredQueryDefinition` that represents the objects you want to find.

   a. For a string query, create a `StringQueryDefinition` using a `QueryManager` object. For details, see "Full Text Search with String Query" on page 234. For example, the following query performs a full text search for the phrase "dog":

```
QueryManager qm = client.newQueryManager();
StringQueryDefinition query =
    qm.newStringDefinition().withCriteria("dog");
```

   b. For a structured query, use `PojoRepository.getQueryBuilder` to create a query builder, and then use the query builder to create your query. For details, see "Search Using Structured Query" on page 234. For example, the following query matches objects whose "otherData" property value is "dog":

```
StructuredQueryDefinition query =
    myClassRepo.getQueryBuilder().value("otherData", "dog");
```

6. Call `PojoRepository.search` to find matching objects in the database. Set the `start` parameter to 1 to retrieve results beginning with the first match, or set it to higher value to return subsequent pages of results, as described in “Retrieving POJOs Incrementally” on page 249.

```
PojoPage<MyClass> matchingObjs = repo.search(query,1);
while (matchingObjs.hasNext()) {
    MyClass ojb = matchingObjs.next();
    ...
}
```

7. When you are finished with the database, release the connection.

```
client.release();
```

Matching objects are returned as a `PojoPage`, which represents a limited number of results. You may not receive all results in a single page if you read a large number objects. You can fetch the matching objects in batches, as described in “Retrieving POJOs Incrementally” on page 249. You can configure the page size using `PojoRepository.setPageLength`.

9.7.2 Full Text Search with String Query

A string query is a plain text search string composed of terms, phrases, and operators that can be easily composed by end users typing into an application search box. For example, 'cat AND dog' is a string query for finding documents that contain both the term 'cat' and the term 'dog'. For details, see The Default String Query Grammar in the *Search Developer's Guide*.

Using a string query to search POJOs performs a full text search. That is, matches can occur anywhere in an object.

For example, if the sample data contains an object whose “title” property is “Leaves of Grass” and another object whose “author” property is “Munro Leaf”, then the following search matches both objects. (The search term “leaf” matches “leaves” because string search uses stemming by default.)

```
QueryManager qm = client.newQueryManager();
StringQueryDefinition query =
    qm.newStringDefinition().withCriteria("leaf");
PojoPage<Book> matches = repo.search(query, 1);
```

For a complete example, see “Searching POJOs in the Database” on page 232.

9.7.3 Search Using Structured Query

A structured query is an Abstract Syntax Tree representation of a search expression. You can use structured query to build up a complex query from a rich set of sub-query types. For example, structured query enables you to search specific object properties.

Use `PojoQueryBuilder` to create structured queries over your persisted POJOs. Though you can create structured queries in other ways, using a `PojoQueryBuilder` enables you to create queries without knowing the details of how your objects are persisted in the database or the syntax of a structured query. Also, `PojoQueryBuilder` exposes only those structured query capabilities that are applicable to POJOs.

To create a `PojoQueryBuilder`, use `PojoRepository.getQueryBuilder` to create a builder. For example:

```
PojoQueryBuilder<Person> qb = repo.getQueryBuilder();
```

Use the methods of `PojoQueryBuilder` to create complex, compound queries on your objects, equivalent to structured query constructs such as `and-query`, `value-query`, `word-query`, `range-query`, `container-query`, and geospatial queries. For details, see Structured Query Concepts in the *Search Developer's Guide*.

To match data in objects nested inside your top level POJO class, use `PojoQueryBuilder.containerQuery` (or `PojoQueryBuilder.containerQueryBuilder`) to constrain a query or sub-query to a particular sub-object.

For example, suppose your objects have the following structure:

```
public class Person {
  public Name name;
}
public class Name {
  public String firstName;
  public String lastName;
}
```

The following search matches the term "john" in `Person` objects only when it appears somewhere in the `name` object. It matches occurrences in either `firstName` or `lastName`.

```
PojoQueryBuilder qb = repo.getQueryBuilder();
PojoPage<Person> matches = repo.search(
    qb.containerQuery("name", qb.term("john")), 1);
```

The following query further constrains matches to occurrences in the `lastName` property of `name`.

```
qb.containerQuery("name",
    qb.containerQuery("lastName", qb.term("john")))
```

For a complete example, see "Searching POJOs in the Database" on page 232.

9.7.4 How Indexing Affects Searches

You can search POJOs with many query types without defining any indexes. This enables you to get started quickly. However, indexes are required for range queries (`PojoQueryBuilder.range`) and can significantly improve search performance by enabling unfiltered search, as described below.

A *filtered* search uses available indexes, if any, but then checks whether or not each candidate meets the query requirements. This makes a filtered search accurate, but much slower than an unfiltered search. An *unfiltered* search relies solely on indexes to identify matches, which is much faster, but can result in false positives. For details, see Fast Pagination and Unfiltered Searches in *Query Performance and Tuning Guide*.

By default, a POJO search is an unfiltered search. To force use of a filtered search, wrap your query in a call to `PojoQueryBuilder.filteredQuery`. For example:

```
repo.search(builder.filteredQuery(builder.word("john")))
```

Unless your database is small or your query produces only a small set of pre-filtering results, you should define an index over any object property used in a word, value, or range query. If your search includes a range query, you must either have an index configured on each object property used in the range query, or you must wrap your query in a call to `PojoRepository.filteredQuery` to force a filtered search.

The POJO interfaces of the Java API include the ability to annotate object properties that should be indexed, and then generate an index configuration from the annotation. For details, see “Creating Indexes from Annotations” on page 236.

9.7.5 Creating Indexes from Annotations

As described in “How Indexing Affects Searches” on page 236, you should usually create indexes on object properties used in range queries. Though no automatic index creation is provided, the POJO interface can simplify index creation for you by generating index configuration information from annotations.

Use the following procedure to create an index on an object property.

1. Attach an `@PathIndexProperty` annotation to each object property you want to index. You can attach the annotation to a member, setter, or getter. Set scalarType to a value compatible with the type of your object property. For example:

```
import com.marklogic.client.pojo.annotation.PathIndexProperty;

public class Person {
  ...

  @PathIndexProperty(scalarType=PathIndexProperty.ScalarType.INT)
  public int getAge() {
    return age;
```

```
  }
  ...
}
```

2. Run the `com.marklogic.client.pojo.util.GenerateIndexConfig` tool to generate an index configuration for your application. For example, if you run the following command against the example code in “Example: Searching POJOs” on page 240:

```
$ java com.marklogic.client.pojo.util.GenerateIndexConfig \
    -classes "examples.Person examples.Name"
    -file personIndexes.json
```

   Then the following index configuration is saved to the file `personIndexes.json`.

```
{
  "range-path-index" : [ {
    "path-expression" : "examples.Person/age",
    "scalar-type" : "int",
    "collation" : "",
    "range-value-positions" : "false",
    "invalid-values" : "ignore"
  } ],
  "geospatial-path-index" : [ ],
  "geospatial-element-pair-index" : [ ]
}
```

3. Use the generated index configuration to add the required indexes to the database in which you store your POJOs. See below for details.

You can use the output from `GenerateIndexConfig` to add the required indexes to your database in several ways, including the Admin Interface, the XQuery Admin API, and the Management REST API.

The output from `GenerateIndexConfig` is suitable for immediate use with the REST Management API method `PUT:/manage/v2/databases/{id|name}/properties`. However, be aware that this interface overwrites all indexes in your database with the configuration in the request.

To use the output of `GenerateIndexConfig` to create indexes with the REST Management API, run a command similar to the following. This example assumes you are using the Documents database for your POJO store and that the file personIndexes.json was generated by `GenerateIndexConfig`.

Warning The following command will replace all indexes in the database with the indexes in `personIndexes.json`. Do not use this procedure if your database configuration includes other indexes that should be preserved.

```
$ curl --anyauth --user user:password -X PUT -i
    -H "Content-type: application/json" -d @./personIndexes.json \
    http://localhost:8002/manage/LATEST/databases/Documents/properties
```

To create the required indexes with the REST Management API while preserving existing indexes follow this procedure:

1. Use `GET:/manage/v2/databases/{id|name}/properties` to retrieve the current database properties. For example, the following command saves the properties of the Documents database to the file `allProperties.json`:

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/json" -o allProperties.json
    http://localhost:8002/manage/LATEST/databases/Documents/properties
```

2. Locate the indexes of the same types as those generated by GenerateIndexConfig in the output from Step.

   a. If there are no indexes of the same type as those generated by GenerateIndexConfig, you can safely apply the generated configuration directly.

   b. If there are existing indexes of the same type as those generated by GeneratedIndexConfig, extract the existing indexes of that type from the output of Step 1 and combine this configuration information with the output from GenerateIndexConfig. See the example below.

3. Use `PUT:/manage/v2/databases/{id|name}/properties` to install the merged index configuration. For example:

```
$ curl --anyauth --user user:password -X PUT -i
    -H "Content-type: application/json" -d @./comboIndex.json \
    http://localhost:8002/manage/LATEST/databases/Documents/properties
```

For example, suppose `GenerateIndexConfig` generates the following output, which includes one path range index on `Person.age` and no geospatial indexes.

```
{
  "range-path-index" : [ {
    "path-expression" : "examples.Person/age",
    "scalar-type" : "int",
    "collation" : "",
    "range-value-positions" : "false",
    "invalid-values" : "ignore"
  } ],
  "geospatial-path-index" : [ ],
  "geospatial-region-path-indexes" : [ ],
  "geospatial-element-pair-index" : [ ]
}
```

Further suppose retrieving the current database properties reveals an existing `range-path-index` setting such as the following:

```
$ curl --anyauth --user user:password -X GET \
    -H "Accept: application/json" -o allProperties.json
    http://localhost:8002/manage/LATEST/databases/Documents/properties

==> Properties saved to allProperties.json include the following:

{
  "database-name": "Documents",
  "forest": [
    "Documents"
  ],
  "security-database": "Security",
  ...
  "range-path-index": [
    {
      "scalar-type": "string",
      "collation": "http://marklogic.com/collation/",
      "path-expression": "/some/other/data",
      "range-value-positions": false,
      "invalid-values": "reject"
    }
  ],
  ...
}
```

Then combining the existing index configuration with the generated POJO index configuration results in the following input to `PUT:/manage/v2/databases/{id|name}/properties`. (You can omit the generated `geospatial-path-index`, `geospatial-region-path-index`, and `geospatial-element-pair-index` configurations in this case because they are empty.)

```
{ "range-path-index" : [
    {
    "path-expression" : "examples.Person/age",
    "scalar-type" : "int",
    "collation" : "",
    "range-value-positions" : "false",
    "invalid-values" : "ignore"
    },
    {
      "scalar-type": "string",
      "collation": "http://marklogic.com/collation/",
      "path-expression": "/some/other/data",
      "range-value-positions": false,
      "invalid-values": "reject"
    }
] }
```

As shown above, it is not necessary to merge the generated index configuration into the entire properties file and reapply all the property settings. However, you can safely do so if you know that none of the other properties have changed since you retrieved the properties.

For more information on the REST Management API, see the *Monitoring MarkLogic Guide* and the *Scripting Administrative Tasks Guide*.

9.8 Example: Searching POJOs

The example in this section demonstrates using string and structured queries to search POJOs, as well as pagination of search results. The following topics are covered:

- Overview of the Example
- Source Code
- Exploring the Example Queries

9.8.1 Overview of the Example

The example uses `Person` objects as POJOs. Each `Person` contains data such as name, age, gender, unique id, and birthplace. The name is represented by a `Name` object that contains the first and last name. Age is an integer value. Gender is an enumeration. The remaining properties are strings. Thus, the data available for a person has the following conceptual structure:

```
name:
  firstName: John
  lastName: Doe
gender: MALE
age: 27
id: 123-45-6789
birthplace: Hometown, NY
```

The `id` object property is used as the unique POJO identifier.

The example is driven by the `PeopleSearch` class. Running `PeopleSearch.main` loads `Person` objects into the database, performs several searches using string and structured queries, and then removes the objects from the database.

The following methods are the operations of `PeopleSearch`.

- `dbInit`: Load `Person` objects into the database
- `dbTeardown`: Remove all `Person` objects from the database
- `stringQuery`: Perform a string query and print the first page of results
- `doQuery`: Perform a structured query and print the first page of results

The PeopleSearch class uses the helper methods `stringQuery` and `doQuery` to abstract the invariant mechanics of the search from the query construction.

The `stringQuery` and `doQuery` helper methods simply encapsulate the invariant parts of performing a search and displaying the results in order to make it easier to focus on query construction.

9.8.2 Source Code

This section contains the full source code for the example. Copy this code to files in order run the example.

- Person Class Definition
- Name Class Definition
- PeopleSearch Class Definition

9.8.2.1 Person Class Definition

Person is the top level POJO class used by the example. `Person.getId` is annotated as the object id. Additional annotations call out the need for an index on the age property so it can be used in range queries.

Copy the following code into a file with the relative pathname examples/Person.java.

```
package examples;

import com.fasterxml.jackson.annotation.JsonIgnore;
import com.marklogic.client.pojo.annotation.Id;
import com.marklogic.client.pojo.annotation.PathIndexProperty;

public class Person {
    public Person() {}
    public Person(String first, String last, Gender gender,
                  int age, String id, String birthplace) {
        this.name = new Name(first, last);
        this.age = age;
        this.id = id;
        this.gender = gender;
        this.birthplace = birthplace;
    }

    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }

    @PathIndexProperty(scalarType=PathIndexProperty.ScalarType.INT)
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
```

```
        this.age = age;
    }

    @Id
    public String getSSN() {
        return id;
    }
    public void setSSN(String ssn) {
        this.id = ssn;
    }

    public Gender getGender() {
        return gender;
    }
    public void setGender(Gender gender) {
        this.gender = gender;
    }

    @JsonIgnore
    public String getFullName() {
        return this.name.getFullName();
    }

    public String getBirthplace() {
        return birthplace;
    }
    public void setBirthplace(String birthplace) {
        this.birthplace = birthplace;
    }

    enum Gender {MALE, FEMALE}

    private Name name;
    private Gender gender;
    private int age;
    private String id;
    private String birthplace;
}
```

9.8.2.2 Name Class Definition

The Name class exists to demonstrate searching sub-objects of your top level POJO class. Each Person object contains a Name.

Copy the following code into a file with the relative pathname `examples/Name.java`.

```
package examples;

import com.fasterxml.jackson.annotation.JsonIgnore;
import com.marklogic.client.pojo.annotation.PathIndexProperty;

public class Name {
    public Name() { }
```

```
    public Name(String first, String last) {
        this.firstName = first;
        this.lastName = last;
    }

    @PathIndexProperty(scalarType=PathIndexProperty.ScalarType.STRING)
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    @PathIndexProperty(scalarType=PathIndexProperty.ScalarType.STRING)
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    @JsonIgnore
    public String getFullName() {
        return this.firstName + " " + this.lastName;
    }

    private String firstName;
    private String lastName;
}
```

9.8.2.3 PeopleSearch Class Definition

`PeopleSearch` is the class that drives the examples. The main method loads `Person` POJOs into the database, performs some searches, and then removes the POJOs from the database.

Copy the following code into a file with the relative path `examples/PeopleSearch.java`. Modify the call to `DatabaseClientFactory.newClient` to use your connection information. You will need to change at least the username and password parameter values.

```
package examples;

import com.marklogic.client.DatabaseClient;
import com.marklogic.client.DatabaseClientFactory;
import com.marklogic.client.DatabaseClientFactory.DigestAuthContext;
import com.marklogic.client.pojo.PojoPage;
import com.marklogic.client.pojo.PojoQueryBuilder;
import com.marklogic.client.pojo.PojoQueryBuilder.Operator;
import com.marklogic.client.pojo.PojoQueryDefinition;
import com.marklogic.client.pojo.PojoRepository;
import com.marklogic.client.query.QueryManager;
import com.marklogic.client.query.StringQueryDefinition;

import examples.Person.Gender;
```

```
public class PeopleSearch {
  private static DatabaseClient client = DatabaseClientFactory.newClient(
    "localhost", 8000, new DigestAuthContext(USER, PASSWORD));
  private static PojoRepository<Person,String> repo =
    client.newPojoRepository(Person.class, String.class);


  // The pojos to be stored in the database for searching
  private static Person people[] = {
    new Person("John", "Doe", Gender.MALE, 27, "123-45-6789", "Albany, NY"),
    new Person("John", "Smith",
               Gender.MALE, 41, "234-56-7891", "Las Vegas, NV"),
    new Person("Mary", "John",
               Gender.FEMALE, 19, "345-67-8912", "Norfolk, VA"),
    new Person("Jane", "Doe",
               Gender.FEMALE, 72, "456-78-9123", "St. John, FL"),
    new Person("Sally", "St. John", Gender.MALE, 34,
               "567-89-1234", "Reno, NV"),
    new Person("Kate", "Peters",
               Gender.FEMALE, 17, "678-91-2345", "Denver, CO")
  };

  // Save the example pojos to the database
  static void dbInit() {
    // Save objects to the database
    for (int i = 0; i < people.length; i++) {
      repo.write(people[i]);
    }
  }

  // Remove the pojos from the database
  static void dbTeardown() {
    repo.deleteAll();
  }

  // Print one page of results
  static void printResults(PojoPage<Person> matchingObjs) {
    if (matchingObjs.hasContent()) {
      while (matchingObjs.hasNext()) {
        Person person = matchingObjs.next();
        System.out.println("  " + person.getFullName() + " from "
            + person.getBirthplace());
      }
    } else {
      System.out.println("  No matches");
    }
    System.out.println();
  }

  // Perform a structured query and print the first page of results
  public void doQuery(PojoQueryDefinition query) {
    printResults(repo.search(query,1));
  }

  // Perform a full text search and print first page of results
  public void stringQuery(String qtext) {
    QueryManager qm = client.newQueryManager();
    StringQueryDefinition query = qm.newStringDefinition().withCriteria(qtext);
```

```
    printResults(repo.search(query,1));
  }

  // Fetch all matches, one page at a time
  public void fetchAll(PojoQueryDefinition query) {
    PojoPage<Person> matches;
    int start = 1;
    do {
      matches = repo.search(query, start);
      System.out.println("Results " + start +
                         " thru " + (start + matches.size() - 1));
      printResults(matches);
      start += matches.size();
    } while (matches.hasNextPage());
  }

  public static void main(String[] args) {
    PeopleSearch ps = new PeopleSearch();

    // load the POJOs
    dbInit();

    // Perform a string query
    System.out.println("Full text search for 'john'");
    ps.stringQuery("john");

    System.out.println(
      "Full text search for 'john' only where there is no 'NV'");
    ps.stringQuery("john AND -NV");

    // Perform structured queries
    PojoQueryBuilder<Person> qb = repo.getQueryBuilder();

    System.out.println("'john' appears anywhere in the person record");
    ps.doQuery(qb.term("john"));

    System.out.println("name contains 'john'");
    ps.doQuery(qb.containerQuery("name", qb.term("john")));

    System.out.println("last name exactly matches 'John'");
    ps.doQuery(qb.value("lastName","John"));

    System.out.println("last name contains the term 'john'");
    ps.doQuery(qb.word("lastName", "john"));

    System.out.println("First name or last name contains 'john'");
    ps.doQuery(
      qb.containerQuery("name",
        qb.or(qb.value("firstName", "John"),
              qb.value("lastName", "John"))));

    System.out.println("'john' occurs in lastName property of name");
    ps.doQuery(
      qb.containerQuery("name",
        qb.containerQuery("lastName", qb.term("john"))));

    System.out.println("find all females");
    ps.doQuery(qb.value("gender", "FEMALE"));
```

```
    // This query requires the existence of a range index on age
    System.out.println("all persons older than 30");
    ps.doQuery(qb.range("age", Operator.GT, 30));

    // Demonstrate retrieving successive pages of results.
    // Page length is set artificially low to force multiple pages of results.
    repo.setPageLength(2);
    System.out.println("Retrieve multiple pages of results");
    ps.fetchAll(qb.range("age", Operator.GT, 30));

    // comment this line out to leave the objects in the database between runs
    dbTeardown();
    client.release();
  }
}
```

9.8.3 Exploring the Example Queries

This section provides an overview of the queries performed by the `PeopleSearch` example. The searches are driven by the helper functions `stringSearch` and `doQuery`. These are simply wrappers around `PojoRepository.search` to abstract the invariant parts of each search, such as displaying the results. For example, the following call to `doQuery`:

```
ps.doQuery(qb.value("gender", "FEMALE"));
```

Is equivalent to the following code, fully unrolled. Additional calls to `doQuery` in the example vary only by the query that is passed to `PojoRepository.search`.

```
PojoPage<Person> matchingObjs =
  repo.search(qb.value("gender", "FEMALE"),1));
if (matchingObjs.hasContent()) {
  while (matchingObjs.hasNext()) {
    Person person = matchingObjs.next();
    System.out.println("  " + person.getFullName() + " from " +
                       person.getBirthplace());
  }
} else {
  System.out.println("  No matches");
}
System.out.println();
```

The example begins with some simple string queries. The table below describes the interesting features of these queries.

Query Text	Description
`"john"`	Match the term "john" wherever it appears in the `Person` objects. The match is not case-sensitive and will match portions of values, such as “St. John”.

Query Text	Description
`"john AND -NV"`	Match `Person` objects that contain the phrase "john" and do not contain "NV". The “-” operator is a NOT operator in string queries. Since the search term “NV” is capitalized, that term is matched in a case-sensitive manner. By contrast, the term “-nv” is a case-insensitive match that would match “nv”, “NV”, “nV”, and “nV”.

The default treatment of case sensitivity in string queries is that phrases that are all lower-case are matched case-insensitive. Upper case or mixed case phrases are handled in a case-sensitive manner. You can control this behavior through the `term` query option; for details, see term in the *Search Developer's Guide*.

The remaining queries in the example are structured queries. The table below describes the key characteristics of these queries.

Query	Description
`qb.term("john")`	Match the phrase "john" anywhere in the `Person` objects. The match is not case-sensitive and will match portions of values, such as “St. John”.

`qb.containerQuery(`
`  "name",`
`  qb.term("john"))`
	Match the phrase "john" only in the value of the `name` object property. Matches can be at any level within name.

`qb.value("lastName","John")`	Match objects whose `lastName` object property has the exact value "John". Values such as "john" or "St. John” do not match.
`qb.word("lastName", "john")`	Match objects whose `lastName` object property value includes the phrase "john". The match is not case-sensitive and will match portions of values, such as “St. John”. The search does not recurse through sub-objects. For example, since `Person.name` is an object, `qb.word("name", "john")` finds no matches because it will not look into the values of `lastName` and `firstName` object properties. The `lastName` object property can appear at any level. That is, it is not restricted to occurrences within name.

Query	Description
`qb.containerQuery( "name", qb.or( qb.value("firstName","John"), qb.value("lastName","John")))`	Match objects whose `lastName` or `firstName` object property is exactly "John". You can combine arbitrarily complex queries together.
`qb.containerQuery( "name", qb.containerQuery( "lastName", qb.term("john")))`	Match objects whose `name` property contains a `lastName` property that includes the phrase "john" at any level.
`qb.value("gender", "FEMALE")`	Match objects whose `gender` property is exactly the value "FEMALE". The match must be exact.
`qb.range("age", Operator.GT, 30)`	Match objects whose `age` property value is greater than 30. The database configuration must include a path range index on `age` of type `int`. If a matching index is not found, a `XDMP-PATHRIDXNOTFOUND` error occurs. For details, see "How Indexing Affects Searches" on page 236.

The final query in the example demonstrates pagination of query results, using the `PeopleSearch.fetchAll` helper function. The query result page length is first set to 2 to force pagination to occur on our small results. After this call, each call to `PojoRepository.search` or `PojoRepository.readAll` will return at most 2 results.

```
repo.setPageLength(2);
```

The `fetchAll` helper function below repeatedly call `PojoRepository.search` (and prints out the results) until there are no more pending matches. Each call to search includes the starting position of the first match to return. This parameter starts out as 1, to retrieve the first match, and is incremented each time by the number of matches on the fetched page (`PojoPage.size`). The loop terminates when there are no more results (`PojoPage.hasNextPage` returns false).

```
public void fetchAll(PojoQueryDefinition query) {
    PojoPage<Person> matches;
    int start = 1;
    do {
      matches = repo.search(query, start);
      System.out.println("Results " + start +
                         " thru " + (start + matches.size() - 1));
      printResults(matches);
      start += matches.size();
    } while (matches.hasNextPage());
  }
```

9.9 Retrieving POJOs Incrementally

By default, when you retrieve POJOs using `PojoRepository.read` or `PojoRepository.search`, the number of results returned is limited to one "page". Paging results enables you to retrieve large result sets without consuming undue resources or bandwidth.

The number of results per page is configurable on `PojoRepository`. The default page length is 10, meaning at most 10 objects are returned. You can change the page length using `PojoRepository.setPageLength`. When you're reading POJOs by id, you can also retrieve an unconstrained number of results by calling `PojoRepository.readAll`.

All PojoRepository methods for retrieving POJOs include a "start" parameter you can use to specify the 1-based index of the first object to return from the result set. Use this parameter in conjunction with the page length to iteratively retrieve all results.

For example, the following function fetches successive groups of Person objects matching a query. For a runnable example, see "Example: Searching POJOs" on page 240.

```
public void fetchAll(PojoQueryDefinition query) {
    PojoPage<Person> matches;
    int start = 1;
    do {
      matches = repo.search(query, start);
      // ...do something with the matching objects...
      start += matches.size();
    } while (matches.hasNextPage());
  }
```

Both PojoRepository.search and PojoRepository.read return results in a PojoPage. Use the same basic strategy whether fetching objects by id or by query.

A PojoPage can container fewer than `PojoRepository.getPageLength` objects, but will never contain more.

9.10 Removing POJOs from the Database

You can delete POJOs from the database in two ways:

- By id, using `PojoRepository.delete`. You can specify one or more object ids.
- By POJO class, using `PojoRepository.deleteAll`.

Since a PojoRepository is bound to a specific POJO class, calling `PojoRepository.deleteAll` removes all POJOs of the bound type from the database.

9.11 Testing Your POJO Class for Serializability

You can only use the data binding interfaces with Java POJO classes that can be serialized and deserialized by Jackson. You can use a test such as the following to check whether or not your POJO class is serializable.

```
try {
  String value = objectMapper.writeValueAsString(
    new MyClass(42,"hello"));
  MyClass newobj = objectMapper.readValue(value, MyClass.class);
  // class is serializable if no exception is raised by objectMapper
} catch (Exception e) {
  e.printStackTrace();
}
```

9.12 Troubleshooting

This section contains topics for troubleshooting errors and surprising behaviors you might encounter while working with the POJO interfaces. The following topics are covered:

- Error: XDMP-UNINDEXABLEPATH
- Error: XDMP-PATHRIDXNOTFOUND
- Unexpected Search Results

9.12.1 Error: XDMP-UNINDEXABLEPATH

If you see an error similar to the following:

```
search failed: Internal Server Error. Server Message:
XDMP-UNINDEXABLEPATH: examples.PojoSearch$Person/id
```

Then you are probably using an object property of a nested class as the target of your @Id annotation. You cannot use the POJO interfaces with nested classes.

Nested class names serialize with a “$” in their name, such as examples.PojoSearch$Person, above. Path expressions with such symbols in them cannot be indexed.

9.12.2 Error: XDMP-PATHRIDXNOTFOUND

If you see an error similar to the following:

```
search failed: Bad Request. Server Message: XDMP-PATHRIDXNOTFOUND:
cts:search(...)
```

Then you need to configure a supporting index in the database in which you store your POJOs. For details, see “How Indexing Affects Searches” on page 236 and “Creating Indexes from Annotations” on page 236.

9.12.3 Unexpected Search Results

If your POJO search does not return the results you expect, you can dump out the serialization of the query produced by PojoQueryBuilder to see if the resulting structured query is what you expect. For example:

```
System.out.println(qb.range("age", Operator.GT, 30).serialize());
==>
<query xmlns="http://marklogic.com/appservices/search"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:search="http://marklogic.com/appservices/search"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <range-query type="xs:int">
    <path-index>examples.Person/age</path-index>
    <value>30</value>
    <range-operator>GT</range-operator>
  </range-query>
</query>
```

If your query looks as you expect, the surprising results might be the result of using unfiltered search. Search on POJOs are unfiltered by default, which makes the search faster, but can produce false positives. For details, see “How Indexing Affects Searches” on page 236.

10.0 Alerting

The MarkLogic Java API enables you to create applications that include client-side alerting capabilities through the `com.marklogic.client.alerting` package. You can use the RuleDefinition and RuleManager interfaces to create and maintain alerting rules and to test documents for matches to rules.

This chapter covers the following topics:

- Alerting Pre-Requisites
- Alerting Concepts
- Defining Alerting Rules
- Testing for Matches to Alerting Rules

10.1 Alerting Pre-Requisites

You should enable "fast reverse searches" on the content database associated with your REST API instance. Enable fast reverse searches using the Admin Interface, as described in Indexes for Reverse Queries in *Search Developer's Guide*, or using the XQuery function `admin:database-set-fast-reverse-searches`.

Creating or delete alerting rules requires the `rest-writer` role, or equivalent privileges. All other alerting operations require the `rest-reader` role, or equivalent privileges.

10.2 Alerting Concepts

An *alerting application* is one that takes action whenever content matches a pre-defined set of criteria. For example, send an email notification to a user whenever a document about influenza is added to the database. In this case, the criteria might be "the abstract contains the word influenza", and the action is "send an email".

MarkLogic Server supports server-side alerting through the XQuery API and Content Processing Framework (CPF), and client-side alerting through the REST and Java APIs.

A server-side alerting application usually uses a "push" model. You register alerting rules and XQuery action functions with MarkLogic Server. Whenever content matches the rules, MarkLogic Server evaluates the action functions. For details, see Creating Alerting Applications in *Search Developer's Guide*.

By contrast, a client-side alerting application uses a "pull" alerting model. You register altering rules with MarkLogic Server, as in the push model. However, your application must poll MarkLogic Server for matches to the configured rules, and the application initiates actions in response to matches. This is the model used by the REST Client API.

An *alerting rule* is a query used in a reverse query to determine whether or not a search using that query would match a given document. A normal search query asks “What documents match these search criteria?” A reverse query asks “What criteria match this document?” In the influenza example above, you might define a rule that is a word query for “influenza”, with an element constraint of `<abstract/>`. Alerting rules are stored in the content database associated with your REST API instance.

MarkLogic Server provides fast, scalable rule matching by storing queries in alerting rules in the database and indexing them in the reverse query index. You must explicitly enable “fast reverse searches” on your content database to take advantage of the reverse quer index. For details, see Indexes for Reverse Queries in *Search Developer’s Guide*.

Use the procedures described in this chapter to create and maintain search rules and to test documents for matches to the rules installed in your REST API instance. Determining what actions to take in response to a match and initiating those actions is left to the application.

10.3 Defining Alerting Rules

An alerting rule is defined by a name, a query, and optional metadata. The core of a rule is the combined query that describes the search criteria to use in future match operations. A combined query encapsulates a string and/or structured query plus query options; for syntax details and examples, see Specifying Dynamic Query Options with Combined Query in *REST Application Developer’s Guide*.

Choose one of the following methods to define a rule:

- Defining a Rule Using RuleDefinition
- Defining a Rule in Raw XML
- Defining a Rule in Raw JSON

Note that although you can define a rule in JSON, it will be returned as XML when you read it back from the database.

10.3.1 Defining a Rule Using RuleDefinition

Follow this procedure to define a rule using `com.marklogic.client.alerting.RuleDefinition`:

1. If you have not already done so, connect to the database, storing the connection in a `com.marklogic.client.DatabaseClient` object. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

2. If you have not already done so, create a `com.marklogic.client.alerting.RuleManager`.

```
RuleManager ruleMgr = client.newRuleManager();
```

3. Create a `com.marklogic.client.admin.RuleDefinition` object and populate it with your rule name and data. Optionally, you can include a description and metadata.

```
RuleDefinition rule = new RuleDefinition(RULE_NAME, RULE_DESC);

String combinedQuery = ...; // see complete example, below
StringHandle qHandle = new StringHandle(combinedQuery);
rule.importQueryDefinition(qHandle);

RuleMetadata metadata = rule.getMetadata();
metadata.put(new QName("author"), "me");
```

4. Save the rule to the database by calling `RuleManager.writeRule()`.

```
ruleMgr.writeRule(rule);
```

The following example code snippet puts all the steps together. The example rule matches documents containing the term “xdmp”.

```
// create a manager for configuring rules
RuleManager ruleMgr = client.newRuleManager();
RuleDefinition rule = new RuleDefinition(RULE_NAME, RULE_DESC);

// Configure metadata
RuleMetadata metadata = rule.getMetadata();
metadata.put(new QName("author"), "me");

// Configure the match query
String combinedQuery =
    "<search:search "+
            "xmlns:search='http://marklogic.com/appservices/search'>"+
      "<search:qtext>xdmp</search:qtext>"+
      "<search:options>"+
        "<search:term>"+
          "<search:term-option>case-sensitive</search:term-option>"+
        "</search:term>"+
      "</search:options>"+
    "</search:search>";

//Or the JSON equivalent
String combinedQueryJson =
        "{\"search\":{" +
                "   \"qtext\": \"xdmp\"," +
                "   \"options\": {" +
                "      \"term\": {" +
                "         \"term-option\": \"case-sensitive\"" +
                "      }" +
                "   }" +
                "}" +
            "}";

StringHandle qHandle =
    new StringHandle(combinedQuery).withFormat(Format.XML);
```

```
//JSON equvalent
    new StringHandle(combinedQueryJson).withFormat(Format.JSON);
rule.importQueryDefinition(qHandle);

// Write the rule to the database
ruleMgr.writeRule(rule);
```

10.3.2 Defining a Rule in Raw XML

Follow this procedure to define a rule directly in XML. When creating the rule, use the template in Defining an Alerting Rule in *REST Application Developer's Guide*.

1. If you have not already done so, connect to the database, storing the connection in a `com.marklogic.client.DatabaseClient` object. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

2. If you have not already done so, create a `com.marklogic.client.alerting.RuleManager`.

```
RuleManager ruleMgr = client.newRuleManager();
```

3. Create an XML representation of the rule, using a text editor or other tool or library.cription and metadata. The following example uses `String` for the raw representation.

```
String rawRule =
    "<rapi:rule xmlns:rapi='http://marklogic.com/rest-api'>"+
      "<rapi:name>"+RULE_NAME+"</rapi:name>"+
      "<rapi:description>An example rule.</rapi:description>"+
      "<search:search "+
              "xmlns:search='http://marklogic.com/appservices/search'>"+
        "<search:qtext>xdmp</search:qtext>"+
        "<search:options>"+
          "<search:term>"+
              "<search:term-option>case-sensitive</search:term-option>"+
          "</search:term>"+
        "</search:options>"+
      "</search:search>"+
      "<rapi:rule-metadata>"+
          "<author>me</author>"+
      "</rapi:rule-metadata>"+
    "</rapi:rule>";
```

4. Create a handle on your raw query, using a class that implements `RuleWriteHandle`. For example:

```
StringHandle handle =
    new StringHandle(rawRule).withFormat(Format.XML);
```

5. Save the rule to the database by calling `RuleManager.writeRule()`.

```
ruleMgr.writeRule(RULE_NAME, handle);
```

The following example code snippet puts all the steps together. The example rule matches documents containing the term “xdmp”.

```
// create a manager for configuring rules
RuleManager ruleMgr = client.newRuleManager();

// Define the rule in raw XML
String rawRule =
  "<rapi:rule xmlns:rapi='http://marklogic.com/rest-api'>"+
    "<rapi:name>"+RULE_NAME+"</rapi:name>"+
    "<rapi:description>An example rule.</rapi:description>"+
    "<search:search "+
            "xmlns:search='http://marklogic.com/appservices/search'>"+
      "<search:qtext>xdmp</search:qtext>"+
      "<search:options>"+
        "<search:term>"+
          "<search:term-option>case-sensitive</search:term-option>"+
        "</search:term>"+
      "</search:options>"+
    "</search:search>"+
    "<rapi:rule-metadata>"+
        "<author>me</author>"+
    "</rapi:rule-metadata>"+
  "</rapi:rule>";

// create a handle for writing the rule
StringHandle handle =
    new StringHandle(rawRule).withFormat(Format.XML);

// write the rule to the database
ruleMgr.writeRule(RULE_NAME, handle);
```

10.3.3 Defining a Rule in Raw JSON

Follow this procedure to define a rule directly in XML. When creating the rule, use the template in Defining an Alerting Rule in *REST Application Developer’s Guide*.

1. If you have not already done so, connect to the database, storing the connection in a `com.marklogic.client.DatabaseClient` object. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

2. If you have not already done so, create a `com.marklogic.client.alerting.RuleManager`.

```
RuleManager ruleMgr = client.newRuleManager();
```

3. Create a JSON representation of the rule, using a text editor or other tool or library.cription and metadata. The following example uses `String` for the raw representation.

```
String rawRule =
          "{ \"rule\": {"+
            "\"name\" : \""+RULE_NAME3+"\","+
            "\"search\" : {"+
              "\"qtext\" : \"xdmp\","+
              "\"options\" : {"+
                "\"term\" : { \"term-option\" : \"case-sensitive\" }"+
              "}"+
            "},"+
            "\"description\": \"A JSON example rule.\","+
            "\"rule-metadata\" : { \"author\" : \"me\" }"+
          "}}";
```

4. Create a handle using a class that implements `RuleWriteHandle` and associate your raw rule with the handle. For example:

```
StringHandle handle =
    new StringHandle(rawRule).withFormat(Format.JSON);
```

5. Save the rule to the database by calling `RuleManager.writeRule()`.

```
ruleMgr.writeRule(RULE_NAME, handle);
```

The following example code snippet puts all the steps together. The example rule matches documents containing the term “xdmp”.

```
// create a manager for configuring rules
RuleManager ruleMgr = client.newRuleManager();

// Define the rule in raw JSON
String rawRule =
  "{ \"rule\": {"+
    "\"name\" : \""+RULE_NAME3+"\","+
    "\"search\" : {"+
      "\"qtext\" : \"xdmp\","+
      "\"options\" : {"+
        "\"term\" : { \"term-option\" : \"case-sensitive\" }"+
      "}"+
    "},"+
    "\"description\": \"A JSON example rule.\","+
    "\"rule-metadata\" : { \"author\" : \"me\" }"+
  "}}";

// Create a handle for writing the rule
StringHandle qHandle =
    new StringHandle(rawRule).withFormat(Format.JSON);

// Write the rule to the database
ruleMgr.writeRule(RULE_NAME3, qHandle);
```

10.4 Testing for Matches to Alerting Rules

Once you install alerting rules in your REST API instance, use `RuleManager.match()` to determine which rules match one or more input documents. You can select the input documents using a database query or database URIs, or by passing a transient document.

This section covers the following topics:

- Identifying Input Documents Using a Query
- Identifying Input Documents Using URIs
- Matching Against a Transient Document
- Filtering Match Results
- Transforming Alert Match Results

10.4.1 Basic Steps

Follow this procedure to test one or more documents to see if they match installed alerting rules. Identify the input documents using a query or URIs, or by passing in a transient input document.

1. If you have not already done so, connect to the database, storing the connection in a `com.marklogic.client.DatabaseClient` object. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

2. If you have not already done so, create a `com.marklogic.client.alerting.RuleManager`.

```
RuleManager ruleMgr = client.newRuleManager();
```

3. Find the rules that match your input documents by calling `RuleManager.match()`. The result is a list of RuleDefinition objects. The following example uses a query to identify the input documents.

```
StringQueryDefinition querydef = ...;
RuleDefinitionList matchedRules =
    ruleMgr.match(querydef, new RuleDefinitionList());
```

The `match()` method returns the definition of any rules matching your input documents.

You can further customize rule matching by limiting the match results to a subset of the installed rules or applying a server-side transformation to the match results. For details, see the JavaDoc for `RuleManager`.

For a complete example, see `com.marklogic.client.example.cookbook.RawClientAlert`.

10.4.2 Identifying Input Documents Using a Query

You can use a string query, structured query, or combined query to select the documents in the database that you want to test for rule matches. These instructions assume you are familiar with constructing queries using the Java API; for details, see “Searching” on page 144.

Use the following procedure to select input documents using a query:

1. Construct a string, structured, or combined query definition as described in “Searching” on page 144. The following example uses `StringQueryDefinition`.

```
QueryManager queryMgr = client.newQueryManager();
String criteria = "document";
StringQueryDefinition querydef = queryMgr.newStringDefinition();
querydef.setCriteria(criteria);
```

2. If you constructed a raw XML or JSON query definition, create a handle using a class that implements `StructureWriteHandle`. For example, if you created an XML query using `String`, create a `StringHandle`:

```
StringHandle rawHandle =
    new StringHandle(rawXMLQuery).withFormat(Format.XML);
//Or
    new StringHandle(rawJSONQuery).withFormat(Format.JSON);
```

3. Call `RuleManager.match()`, passing in either a `QueryDefinition` or `StructureWriteHandle` to the document selection query.

```
RuleDefinitionList matchedRules =
    ruleMgr.match(querydef, new RuleDefinitionList());
```

For a complete example, see `com.marklogic.client.example.cookbook.RawClientAlert`.

You can limit the rules under consideration by passing an array of rule names to `RuleManager.match()`. You can limit the input documents to a subset of the input query results by specifying start and page length. For details, see the JavaDoc for `RuleManager`.

10.4.3 Identifying Input Documents Using URIs

You can select the documents you want to test for rule matches by specifying a list of document URIs to `RuleManager.match()`. Each URI must identify a document, not a database directory.

Use the following procedure to select input documents using URIs:

1. Construct a String array of document URIs.

```
String[] docIds = { "/example/doc1.xml", "/suggest/doc2.xml" };
```

2. Call `RuleManager.match()`, passing in the list of URIs.

```
RuleDefinitionList matchedRules =
    ruleMgr.match(docIds, new RuleDefinitionList());
```

You can limit the rules under consideration by passing an array of rule names to `RuleManager.match()`. For details, see the JavaDoc for `RuleManager`.

10.4.4 Matching Against a Transient Document

You can test for rule matches against a document that is not in the database by passing the transient document to `RuleManager.match()`.

1. Create a handle using a class that implements `StructureWriteHandle`. The following example uses a `String` as the source document.

```
String doc = "<prefix>xdmp</prefix>";
//Or
String doc = "{\"prefix\": \"xdmp\"}"

StringHandle handle = new StringHandle(doc).withFormat(Format.XML);
//Or
StringHandle handle = new StringHandle(doc).withFormat(Format.JSON);
```

2. Call `RuleManager.match()`, passing in a `StructureWriteHandle` to the document.

```
RuleDefinitionList matchedRules =
    ruleMgr.match(handle, new RuleDefinitionList());
```

You can limit the rules under consideration by passing an array of rule names to `RuleManager.match()`. For details, see the JavaDoc for `RuleManager`.

10.4.5 Filtering Match Results

By default, the result of an alert match includes all matching rules. You can limit the result to a subset of matching rules by passing a list of candiate rule names to `RuleManager.match()`. For example, the result of the following match includes at most the definitions of the rules named “one” and “two”, even if more rules match the input query definition:

```
RuleManager ruleMgr = client.newRuleManager();
StringQueryDefinition querydef = ...;
String [] candidateRules = new String[] {"one", "two"};
RuleDefinitionList matchedRules =
    ruleMgr.match(querydef, 0L, QueryManager.DEFAULT_PAGE_LENGTH,
        candidateRules, new RuleDefinitionList());
```

10.4.6 Transforming Alert Match Results

You can make arbitrary changes to the results from a match request by applying a server-side XQuery transformation function to the results. This section covers the following topics:

- Writing a Match Result Transform
- Using a Match Result Transform

10.4.6.1 Writing a Match Result Transform

Alert match transforms use the same interface and framework as content transformations applied during document ingestion, described in Writing Transformations in the *REST Application Developer's Guide*.

Your transform function receives the raw XML match result data prepared by MarkLogic Server as input, such as a document with a `<rapi:rules/>` root element. For example:

```
<rapi:rules xmlns:rapi="http://marklogic.com/rest-api">
  <rapi:rule>
    <rapi:name>one</rapi:name>
    <rapi:description>Rule 1</rapi:description>
    <search:search
        xmlns:search="http://marklogic.com/appservices/search">
      <search:qtext>xdmp</search:qtext>
    </search:search>
  </rapi:rule>
</rapi/rules>
```

If your function produces XML output and the client application requested JSON output, MarkLogic Server will transform your output to JSON only if one of the following conditions are met.

- Your function produces an XML document that conforms to the "normal" output from the search operation. For example, a document with a `<rapi:rules/>` root element whose contents are changed in a way that preserves the normal structure.
- Your function produces an XML document with a root element in the namespace `http://marklogic.com/xdmp/json/basic` that can be transformed by `json:transform-to-json`.

Under all other circumstances, the output returned by your transform function is what is returned to the client application.

10.4.6.2 Using a Match Result Transform

To use a server transform function:

1. Create a transform function according to the interface described in Writing Transformations in the *REST Application Developer's Guide*.

2. Install your transform function on the REST API instance following the instructions in "Installing Transforms" on page 282.

3. In your application, create a `ServerTransform` object to represent the installed transform, and pass it as a parameter on your call to `RuleManager.match()`. For example:

```
RuleManager ruleMgr = client.newRuleManager();
StringQueryDefinition querydef = ...;
RuleDefinitionList matchedRules =
    ruleMgr.match(querydef, 0L, QueryManager.DEFAULT_PAGE_LENGTH,
        new String[] {}, new RuleDefinitionList(),
        new ServerTransform("your-transform-name"));
```

You are responsible for specifying a handle type capable of interpreting the results produced by your transform function. The `RuleDefinitionList` implementation provided by the Java API only understands the match results structure that MarkLogic Server produces by default.

11.0 Transactions and Optimistic Locking

This chapter covers two different ways for locking documents during MarkLogic Server operations, *multi-statement transactions* and *optimistic locking.*

This chapter includes the following sections:

- Multi-Statement Transactions
- Optimistic Locking

11.1 Multi-Statement Transactions

The following sections cover how to put multiple MarkLogic Server operations in a single *multi-statement transaction.* Specifically, you *open* a transaction, perform multiple operations in it, and then either *rollback* or *commit* the transaction. This section includes the following parts:

- Transactions and the Java API
- Transaction Interface
- Starting A Transaction
- Operations Inside A Transaction
- Rolling Back A Transaction
- Committing A Transaction
- Cookbook: Multistatement Transaction
- Transaction Management When Using a Load Balancer

For detailed information about transactions in MarkLogic Server, see Understanding Transactions in MarkLogic Server in the *Application Developer's Guide*.

11.1.1 Transactions and the Java API

By default, most Java Client API interactions with MarkLogic happen in a single transaction. For example, if you use `DocumentManager.write` to insert a document into the database, the insertion happens as a single transaction that is automatically committed by MarkLogic before a response is sent back to the Java client application.

> **Note:** Requests without multi-statement transactions commit automatically and atomically and can be load balanced

You can use a multi-statement transaction to perform multiple interactions with MarkLogic Server in the context of a single transaction. A multi-statement transaction must be explicitly created and committed or rolled back. For example, you could use a multi-statement transaction to make several calls to DocumentManager.write, and then commit all the writes at once. None of the writes would be visible outside the transaction context unless or until you commit the transaction.

Note: If at all possible, developers should avoid using multi-statement transactions, because 1 atomic network request is more efficient than 3 network requests to open, work, and commit, and because atomic requests can be retried, whereas multi-statement transactions cannot be retried. If you have a use case that requires multi-statement transactions (e.g., where multiple separate requests that mutate the database must complete together or not at all), consider using optimistic locking as a lighter-weight but safe alternative for reading before writing.

Database updates performed in a multi-statement transactions either all succeed or all roll back, depending on whether the transaction is committed or rolled back.

For example, suppose you open a transaction, create a document, and then try to perform a metadata operation on a different document that fails. If, in response to the failure, you roll back the transaction, then neither the document creation nor the metadata update is successful. If you commit the transaction instead, then the document creation succeeds.

Rollbacks do not take place automatically on operation failure. Your application must check for operation success or failure and explicitly rollback the transaction if that is the desired outcome. Failure can be detected by tests of your devising or by trapping and handling a related exception.

Transactions have an associated time limit. If a transaction is not committed before the time limit expires, it is automatically rolled back. The time limit is only a failsafe. You should not design your code with the expectation that a timeout will handle needed rollbacks. Leaving transactions open unnecessarily ties up server-side resources and holds locks on documents. The default time limit is the session time limit configured for your App Server. You can also specify a per transaction time limit when you create a multi-statement transaction; for details, see “Starting A Transaction” on page 266.

A multi-statement transaction must honor host affinity within your MarkLogic cluster. For example, all requests within the context of a transaction should be serviced by the same host. If you use multi-statement transactions in an environment where a load balancer sits between your client application and MarkLogic, then you might need to configure your load balancer to preserve session affinity. For more details, see “Transaction Management When Using a Load Balancer” on page 268.

Note that a document operation performed in the default single statement transaction context locks the document until that operation succeeds or fails. If MarkLogic detects a deadlock, then the transaction is automatically restarted until either it completes or an exception is thrown (for example, by reaching a time limit for the update transaction). This happens automatically, and you normally do not need to worry about it.

11.1.2 Transaction Interface

Use the `com.marklogic.client.Transaction` interface to manage a transaction. The following are the key operations for managing multi-statement transactions in the Java Client API:

- Start a multi-statement transaction. For more details, see "Starting A Transaction" on page 266.

  ```
  DatabaseClient.openTransaction()
  ```

- Commit a multi-statement transaction when it successfully finishes. For more details, see "Committing A Transaction" on page 268.

  ```
  Transaction.commit()
  ```

- Rollback a multi-statement transaction, resetting any actions that previously took place in that transaction. For example, delete any created documents, restore any deleted documents, revert updates, etc. For more details, see "Rolling Back A Transaction" on page 267.

  ```
  Transaction.rollback()
  ```

You perform operations inside a given multi-statement transaction by passing the `Transaction` object returned by `DatabaseClient.openTransaction` into the operation. For details, see "Operations Inside A Transaction" on page 267

Use the `Transaction.readStatus` method to check whether or not a transaction is still open. That is, whether or not it has been committed or rolled back.

11.1.3 Starting A Transaction

To create a multi-statement transaction and obtain a `Transaction` object, call the `openTransaction()` method on a `DatabaseClient` object. To call `openTransaction()`, an application must authenticate as `rest-writer` or `rest-admin`. For example:

```
Transaction transaction = client.openTransaction();
```

You can also include a transaction name and time limit arguments. The `timeLimit` value is the number of seconds the transaction has to finish and commit before it is automatically rolled back. As previously noted, you should not depend on the time limit rolling back your transaction; it is only meant as a failsafe to end the transaction if all else fails.

```
Transaction transaction1 = client.openTransaction("MyTrans", 10);
```

11.1.4 Operations Inside A Transaction

To perform an operation within the context of a multi-statement transaction, pass the `Transaction` object returned by `DatabaseClient.openTransaction` into the operation. For example, pass a `Transaction` object into `DocumentManager.read`, `DocumentManager.write`, or `QueryManager.search`. For example:

```
// read a document inside a transaction
docMgr.read(myDocId1, handle, myTransaction);

// write a document inside a transaction
docMgr.write(myDocId1, handle, myTransaction);

// delete a document inside a transaction
docMgr.delete(myDocId2, myTransaction);
```

You can have more than one transaction open at once. Other users can also be running transactions on or sending requests to the same database. To prevent conflicts, whenever the server does something to a document while in a transaction, the database locks the document until that transaction either commits or rolls back. Because of this, you should commit or roll back your transactions as soon as possible to avoid slowing down your and possibly others' applications.

You can intermix commands which are not part of a transaction with transaction commands. Any command without a `Transaction` object argument is not part of a multi-statement transaction. However, you usually group all operations for a given transaction together without interruption so you can commit or roll it back as fast as possible.

Note: The database context in which you perform an operation in a multi-statement transaction must be the same as the database context in which the transaction was created. The database is set when you create a `DatabaseClient`, so consistency is assured as long as you do not attempt to use a Transaction object created by one `DatabaseClient` with an operation performed through a `DatabaseClient` with a different configuration.

11.1.5 Rolling Back A Transaction

In case of an error or exception, call a transaction's `rollback()` method:

```
transaction.rollback()
```

The `rollback()` method cancels the remainder of the transaction, and reverts the database to its state prior to the transaction start. Proactively rolling back a transaction puts less load on MarkLogic than waiting for the transaction to time out.

To roll back a transaction, your application must authenticate as `rest-writer` or `rest-admin`.

11.1.6 Committing A Transaction

Once your application successfully completes all operations associated with a multi-statement transaction, *commit* the transaction so that the actions are reflected in the database. Commit a transaction by calling `Transaction.commit`:

```
transaction.commit();
```

To commit a multi-statement transaction, your application must authenticate as `rest-writer` or `rest-admin`.

Once a transaction has been committed, it cannot be rolled back and the `Transaction` object is no longer available for use. To perform another transaction, you must create a new `Transaction` object.

11.1.7 Cookbook: Multistatement Transaction

See `com.marklogic.client.example.cookbook.MultiStatementTransaction` for a full example of how to use multi-statement transactions. The Cookbook examples are in the Java API distribution in the following directory:

```
example/com/marklogic/client/example/cookbook
```

11.1.8 Transaction Management When Using a Load Balancer

This section applies only to client applications that use multi-statement transactions and interact with a MarkLogic Server cluster through a load balancer. For additional general-purpose load balancer guidelines, see “Connecting Through a Load Balancer” on page 19.

When you use a load balancer, it is possible for requests from your application to MarkLogic Server to be routed to different hosts, even within the same session. This has no effect on most interactions with MarkLogic Server, but operations that are part of the same multi-statement transaction need to be routed to the same host within your MarkLogic cluster. This consistent routing through a load balancer is called *session affinity*.

Most load balancers provide a mechanism that supports session affinity. This usually takes the form of a session cookie that originates on the load balancer. The client acquires the cookie from the load balancer, and passes it on any requests that belong to the session. The exact steps required to configure a load balancer to generate session cookies depends on the load balancer. Consult your load balancer documentation for details.

To the load balancer, a session corresponds to a browser session, as defined in RFC 2109 (https://www.ietf.org/rfc/rfc2109.txt). However, in the context of a Java Client API application using multi-statement transactions, a session corresponds to a single multi-statement transaction.

The Java Client API leverages a session cookie to preserve host affinity across operations in a multi-statement transaction in the following way. This process is transparent to your application; the information is provided to illustrate the expected load balancer behavior.

1. When you create a transaction using `DatabaseClient.openTransaction`, the Java Client API receives a transaction id from MarkLogic and, if the load balancer is properly configured, a session cookie from the load balancer. This information is cached in the `Transaction` object.

2. Each time you perform a Java Client API operation that includes a `Transaction` object, the Java Client API attaches the transaction id and the session cookie to the request(s) it sends to MarkLogic. The session cookie causes the load balancer to route the request to the same host in your MarkLogic cluster that created the transaction.

3. When MarkLogic receives a request, it ignores the session cookie (if present), but uses the transaction id to ensure the operation is part of the requested transaction. When MarkLogic responds, the load balancer again adds a session cookie, which the Java Client API caches on the `Transaction` object.

4. When you commit or roll back a transaction, any cookies returned by the load balancer are discarded since the transaction is no longer valid. This effectively ends the session from the load balancer's perspective because the Java Client API will no longer pass the session cookie around.

Any Java Client API operation that does not include a `Transaction` object will not include a session cookie (or transaction id) in the request to MarkLogic, so the load balancer is free to route the request to any host in your MarkLogic cluster.

11.2 Optimistic Locking

An application under *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 an object.

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 includes the following sub-sections:

11.2.1 Activating Optimistic Locking

Optimistic locking relies on an opaque numeric identifier that is associated with the state of the document's content at a point of time. By default, the REST Server to which the Java API connects does not keep track of this identifier, but you can enable it for use by setting a property, and make it optional or required.

To expand, there is a number associated with every document. Whenever a document's content changes, the value of its number changes. By comparing the stored value of that number at a point in time with the current value, the REST Server can determine if a document's content has changed since the time the stored value was stored.

Note: While this numeric identifier lets you compare state, and uses a numeric value to do so, this is *not* document versioning. The numeric identifier only indicates that a document has been changed, nothing more. It does not store multiple versions of the document, nor does it keep track of what the changes are to a document, only that it has been changed at some point. You cannot use this for change-tracking or archiving previous versions of a document.

Since this App Server configuration parameter applies either to all documents or none, it is implemented in the REST Server. This means it is part of the overall server configuration, and must be turned on and off via a `ServerConfigurationManager` object and thus requires `rest-admin` privileges. For more about server configuration management, see "REST Server Configuration" on page 276.

To activate optimistic locking, do the following:

```
// if not already done, create a database client
DatabaseClient client = DatabaseClientFactory.newClient(...);

// create server configuration manager
ServerConfigurationManager configMgr =
                           client.newServerConfigManager();

// read the server configuration from the database
configMgr.readConfiguration();

// require content versions for updates and deletes
// use UpdatePolicy.VERSION_OPTIONAL to allow but not
// require identifier use. Use UpdatePolicy.MERGE_METADATA
// (the default) to deactive identifier use
configMgr.setUpdatePolicy(UpdatePolicy.VERSION_REQUIRED);

// write the server configuration to the database
configMgr.writeConfiguration();

// release the client
client.release();
```

Allowed values for `UpdatePolicy` are in the Enum `ServerConfigurationManager.UpdatePolicy`.

11.2.2 DocumentDescriptors

To work with a document's change identifier, you must create a `DocumentDescriptor` for the document. A `DocumentDescriptor` describes exactly one document and is created via use of an appropriately typed method for the document. For more information on document managers, see "Document Managers" on page 26.

```
// create a descriptor for versions of the document
DocumentDescriptor desc = docMgr.newDescriptor(docId);
```

You can also get a document's `DocumentDescriptor` by checking to see if the document exists. This code returns the specified document's `DocumentDesciptor` or, if the document does not exist, `null`:

```
DocumentDescriptor desc = docMgr.exists(docId);
```

11.2.3 Using Optimistic Locking

Each `read()`, `write()`, and `delete()` method for `DocumentManager` has both a version that uses a URI string parameter to identify the document to be read, written, or deleted, and an identical version that uses a `DocumentDescriptor` object instead. The descriptor is only populated with state when you read a document or when you check for a document's existence. When you write, the state changes, but is not reflected in the descriptor.

When `UpdatePolicy` is set to `VERSION_REQUIRED`, you must use the `DocumentDescriptor` versions of the `write()` (when modifying a document) and `delete()` methods. If the change identifier has not changed, the write or delete operation succeeds. If someone else has changed the document so that a new version has been created, the operation fails by throwing an exception.

> **Note:** There is no general notification when UpdatePolicy changes to VERSION_REQUIRED. If the policy changes to required and an application uses the URI string version of `read()`, etc., such requests will now fail and throw exceptions.

If you are creating a document under `VERSION_REQUIRED`, you either must not supply a descriptor, or if you do pass in a descriptor it must not have state. A descriptor is stateless if it is created through a `DocumentManager` and has not yet been populated with state by a `read()` or `exists()` method. If the document does not exist, the operation succeeds. If the document exists, the operation fails and throws an exception.

When `UpdatePolicy` is set to `VERSION_OPTIONAL`, if you do not supply an identifier value via the descriptor and use the `docId` versions of `write()` and `delete()`, the operation always succeeds. If you do supply an identifier value by using the `DocumentDescriptor` versions of `write()` and `delete()`, the same rules apply as above when the update policy is `VERSION_REQUIRED`.

The identifier value always changes on the server when a document's content changes there.

The “optimistic” part of optimistic locking comes from this not being an actual lock, but rather a means of checking if another application has changed a document since you last accessed it. If another application does try to modify the document, the Server does not even try to stop it from doing so. It just changes the document’s identifier value.

So, the next time your application accesses the document, it compares the number it stored for that document with its current number. If they are different, your application knows the document has been changed since it last accessed the document. It could have been changed once, twice, a hundred times; it does not matter. All that matters is that it has been changed. If the numbers are the same, the document has not been changed since you last accessed it.

11.2.4 Cookbook: Version Control and Optimistic Locking

See `com.marklogic.client.example.OptimisticLocking` in the Cookbook for a full example of how to use and optimistic locking. The Cookbook examples are in the Java API distribution in the following directory:

```
example/com/marklogic/client/example/cookbook
```

12.0 Logging

`RequestLogger` objects are supplied to individual manager objects, most commonly document and query managers. You can choose to log content sent to the server as well as any requests. It is located in `com.marklogic.client.util`.

This chapter includes the following sections:

12.1 Starting Logging

First, you must obtain a `RequestLogger` object via `DatabaseClient`'s `newLogger()` method, which takes an argument of an output stream to send the log messages to. This output stream can be shared with other loggers outside of the MarkLogic Server Java API. You are responsible for flushing the output stream.

```
out = new ByteArrayOutputStream();
RequestLogger logger = client.newLogger(out);
```

To start logging, call the `startLogging()` method on a manager object with an argument of a `RequestLogger` object. For example:

```
MyDocumentManager.startLogging(logger)
```

There is only one logger for any given object. However, you can share a `RequestLogger` object among multiple manager objects, just by specifying the same `RequestLogger` object in multiple `startLogging()` method calls.

12.2 Suspending and Resuming Logging

By using `RequestLogger`'s `setEnabled()` method, you can pause and resume logging on any logger object. For example, to suspend logging:

```
logger.setEnabled(false)
```

To reenable logging:

```
logger.setEnabled(true)
```

To check if logging is enabled or not:

```
logger.isEnabled(); //returns a boolean
```

When you change a logger's enable status, it applies to all manager objects for which that `RequestLogger` object was used as an argument to `StartLogging()`.

12.3 Stopping Logging

To stop logging on a manager, call the `stopLogging()` method. If called on a manager not currently logging, nothing happens, not even an error or exception. The `RequestLogger` object associated with the manager is not destroyed by this method and you can reuse and restart it.

```
MyDocumentManager.stopLogging()
```

12.4 Log Entry Format

Two types of things can be logged once logging is turned on and enabled. Requests to the server are always logged. These include search requests, configuration requests, and all database requests. By default, only requests are logged.

You can use `RequestLogger`'s `setContentMax()` method to control how much content is logged. By giving it the constant `ALL_CONTENT` value, all content is logged. To revert to no content being logged, use the constant `NO_CONTENT`. If you use a numeric value, such as `1000`, the first that many content bytes are logged. Note that if the request is for a deletion, no content is logged.

`FileHandle` is an exception to the ability to log content. Only the name of the file is logged.

You can also retrieve a request logger's underlying print stream by calling `getPrintStream()` on the `RequestLogger` object. Once you access the log's print stream, writing to it adds your own messages to the log.

12.5 Logging To The Server's Error Log

You can also use `ServerConfigurationManager.setServerRequestLogging()` to turn logging requests to the server's error log on or off, based on the boolean argument you provide. This log's location is platform dependent. For details about log files in MarkLogic Server, see Log Files in the *Administrator's Guide*.

13.0 REST Server Configuration

REST Server configuration is done through a `ServerConfigurationManager` object located in package `com.marklogic.client.admin`. REST Server configuration deals with the underlying REST instance running in MarkLogic. You can configure REST Server properties, namespace bindings, query options, and transform and resource extensions.

Note that you can only configure aspects of the underlying REST instance with the Java API. MarkLogic Server administration is not exposed in Java, so things such as creating indexes, creating users, creating databases, and assigning roles to users must be done via the MarkLogic Admin Interface or other means (for example, the Admin API or REST Management API). For more information about administering MarkLogic Server, see the *Administrator's Guide*.

This chapter includes the following sections:

13.1 Creating a Server Configuration Manager Object

Using a `com.marklogic.client.DatabaseClient` object, call `newServerConfigManager()`

```
DatabaseClient client = DatabaseClientFactory.newClient(...);

// create a manager for server configuration
ServerConfigurationManager configMgr =
                           client.newServerConfigManager();
```

Your application only needs one active `ServerConfigurationManager` at any time.

13.2 Reading and Writing Server Configuration Properties

Use `com.marklogic.client.admin.ServerConfigurationManager` to manage server configuration properties. To read the current server configuration values into the `ServerConfigurationManager` object, do:

```
configMgr.readConfiguration();
```

If your application changes these values, they will not persist unless written out to the server. To write the REST Server Configuration values to the server, do:

```
configMgr.writeConfiguration();
```

13.3 REST Server Properties

`com.marklogic.client.admin.ServerConfigurationManager` objects have `get` and `set` methods for the following server properties:

- `ContentVersionRequests`: Deprecated. Use `UpdatePolicy` instead.
- `DefaultDocumentReadTransform`: Name of the default transform applied to documents as they are read from the server. For information about document transforms, see "Content Transformations" on page 282.
- `QueryOptionsValidation`: Boolean specifying whether the server validates query options before storing them in configurations. For information about query options, see "Query Options" on page 190.
- `ServerRequestLogging`: Boolean specifying whether the REST Server logs requests to the MarkLogic Server error log (`ErrorLog.txt`). For performance reasons, you should only enable this when debugging your application. For information about logging, see "Logging" on page 274.
- `UpdatePolicy`: Value from the `ServerConfigurationManager.UpdatePolicy` enum specifying whether the system tries to detect if a document is "fresh" or not via use of an opaque numeric identifier and whether to merge or overwrite metadata on update. For more information, see "Optimistic Locking" on page 269.

13.4 Creating New Server-Related Manager Objects

Most manager objects described so far handle access to the database and its content, and accordingly are created via a method on a `DatabaseClient` object. The following managers handle listing, reading, writing, and deleting REST Server data and settings, rather than those of the database. Therefore, these managers are created by factory methods on a `ServerConfigurationManager` instead of a `DatabaseClient`.

The `ServerConfigurationManager` associated managers are:

- `NamespaceManager`: Namespace bindings. For details about namespaces, see "Namespaces" on page 277.
- `QueryOptionsManager`: Query options. For details, about query options, see "Query Options" on page 190.
- `ResourceExtensionsManager`: Resource service extensions. For details about resource service extensions, see "Extending the Java API" on page 288.
- `TransformExtensionManager`: Transform extensions. For details, about transform extensions, see "Content Transformations" on page 282.

13.5 Namespaces

Namespaces are similar to Java packages in that they differentiate between potentially ambiguous XML elements. With the Java API, you can define namespace bindings on the REST Server.

In XML and XQuery, element and attribute nodes are always in a namespace, even if it is the empty namespace (sometimes called no namespace) which has the name of the empty string (""). Each non-empty namespace has an associated URI, which is essentially a unique string that identifies the namespace. That string can be bound to a namespace prefix, which is a shorthand string used as an alias for the namespace in path expressions, element QNames, and variable declarations. Namespace operations in the Java Client API are used to define namespace prefixes on the REST Server so the client and server can share identical namespace bindings on XML elements and attributes for use in queries.

Note that a namespace URI can be bound to multiple prefixes, but a prefix can only be bound to one URI.

If you need to use a namespace prefix in a context in which you cannot declare it, use the REST Management API to define the binding in the App Server. For details, see `PUT:/manage/v2/servers/[id-or-name]/properties`.

For more information about namespaces, see Understanding XML Namespaces in XQuery in the *XQuery and XSLT Reference Guide*, which provides a detailed description of XML namespaces and their use.

This section includes the following parts:

- Namespaces Manager
- Getting Server Defined Namespaces
- Adding And Updating A Namespace Prefix
- Reading Prefixes
- Deleting Prefixes

13.5.1 Namespaces Manager

> **Note:** The `NamespacesManager` interface is deprecated. Use the REST Management API instead. For details, see `PUT:/manage/v2/servers/[id-or-name]/properties` or `GET:/manage/v2/servers/[id-or-name]/properties`.

The `com.marklogic.admin.NamespacesManager` class provides editing for namespaces defined on the REST Server. To use `NamespacesManager`, the application must authenticate as `rest-admin`. Since namespaces are based on the REST Server, a new `NamespacesManager` is defined via `com.marklogic.client.admin.ServerConfigManager`.

```
NamespacesManager nsManager =
    client.newServerConfigManager().newNamespacesManager();
```

13.5.2 Getting Server Defined Namespaces

Note: The `NamespacesManager` interface is deprecated. Use the REST Management API instead. For details, see `PUT:/manage/v2/servers/[id-or-name]/properties` or `GET:/manage/v2/servers/[id-or-name]/properties`.

Use `com.marklogic.client.admin.NamespacesManager` to get all of the namespaces defined on the REST Server. For example:

```
nsManager.readAll();
```

This returns a `javax.xml.namespace.NamespaceContext` interface that includes all of the REST Server defined namespaces. You can run the following on the `NamespaceContext` object.

```
nsContext.getNamespaceURI(prefix-string);
nsContext.getPrefix(URI-string);
nsContext.getPrefixes(URI-string);
```

`getNamespaceURI()` returns the URI associated with the given prefix. `getPrefix()` returns one of the prefixes associated with the given URI. `getPrefixes()` returns an iterator of all the prefixes associated with the given URI.

In addition, by casting the `NamespaceContext` to `EditableNamespaceContext`, you can iterate over the complete set of prefixes and URIs:

```
EditableNamespaceContext c =(EditableNamespaceContext)nsMgr.readAll();
for (Entry e:c.entrySet()){
  prefix = e.getKey();
  nsURI = e.getValue();
...
}
```

13.5.3 Adding And Updating A Namespace Prefix

Note: The `NamespacesManager` interface is deprecated. Use the REST Management API instead. For details, see `PUT:/manage/v2/servers/[id-or-name]/properties` or `GET:/manage/v2/servers/[id-or-name]/properties`.

Use `com.marklogic.client.admin.NamespacesManager` to add a new namespace prefix. For example:

```
nsManager.addPrefix("ml", "http://marklogic.com/exercises");
```

The first argument is the prefix, and the second argument is the URI being associated with the prefix.

To update the value of an existing prefix, do the following:

```
nsManager.updatePrefix("ml", "http://marklogic.com/new_exercises");
```

Where the first argument is the prefix, and the second argument is the new URI bound to it.

13.5.4 Reading Prefixes

Note: The `NamespacesManager` interface is deprecated. Use the REST Management API instead. For details, see `PUT:/manage/v2/servers/[id-or-name]/properties` or `GET:/manage/v2/servers/[id-or-name]/properties`.

Use `com.marklogic.client.admin.NamespacesManager` to read, or get, the associated URI value, of a single prefix. For example:

```
nsManager.readPrefix("ml");
```

It returns the prefix's associated URI as a string.

In order to read, or get, all of the prefixes associated with a Namespace Manager, do the following:

```
NamespaceContext context = nsManager.readAll();
```

`NamespaceContext` is a standard `javax.xml` Interface for storing a set of namespace declarations on the client. With a `NamespaceContext` object, you can:

- Get the prefix for any URI for which a prefix-URI binding has been created in this `NamespaceServer`. The below would return its prefix, say, `"ml"`.

  ```
  context.getPrefix("http://marklogic.com/new_exercises");
  ```

- Get the URI for any prefix for which a prefix-URI binding has been created in this `NamespaceServer`. The below returns the URI `"http://marklogic.com/new_exercises"`

  ```
  context.getNamespaceURI("ml");
  ```

- Get all of the prefixes for any URI for which prefix-URI bindings have been created in this `NamespaceServer`. The below returns all the associated prefixes in an Iterator.

  ```
  context.getPrefixes(“http://marklogic.com/new_exercises);
  ```

13.5.5 Deleting Prefixes

Note: The `NamespacesManager` interface is deprecated. Use the REST Management API instead. For details, see `PUT:/manage/v2/servers/[id-or-name]/properties` or `GET:/manage/v2/servers/[id-or-name]/properties`.

To delete a single prefix from the namespaces manager, do:

```
nsManager.deletePrefix("ml");
```

To delete all of the prefixes defined under a `NamespaceManager`, do:

```
nsManager.deleteAll();
```

13.6 Logging Namespace Operations

As with all manager objects, you can start and stop logging operations on a `NamespacesManager` via the `startLogging()` and `stopLogging()` methods. For details on how to use the logging facility, see “Logging” on page 274.

14.0 Content Transformations

The MarkLogic Java API allows you to create custom content transformations and apply them during operations such as document ingestion and retrieval. You can also apply transformations to search results. Transforms can be implemented using server-side JavaScript, XQuery, and XSLT. A transform can accept transform-specific parameters.

You can specify default transformations as well as operation-specific transformations. For example, setting the `DefaultDocumentReadTransform` property of `ServerConfigurationManager` to the name of a content transformation automatically applies the transformation to every document as it is read from the database. By default, there is no default read transform. Setting up default transforms requires `rest-admin` privileges.

This chapter contains the following sections:

14.1 Installing Transforms

To install a transform on your server, do the following steps:

1. If you have not already done so, create a `DatabaseClient` for connecting to the database. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

2. Create a manager for transform extensions. Since transforms are installed on the REST API instance, use a `ServerConfigManager` to create the manager.

```
TransformExtensionsManager transMgr =
    client.newServerConfigManager().newTransformExtensionsManager();
```

3. Optionally, specify the metadata for the transform, using an `ExtensionMetadata` object.

```
ExtensionMetadata metadata = new ExtensionMetadata();
metadata.setTitle("XML-TO-HTML XSLT Transform");
metadata.setDescription("This plugin transforms an XML document with a
                         known vocabulary to HTML");
metadata.setProvider("MarkLogic");
metadata.setVersion("0.1");
```

4. Create a handle to the transform implementation. For example, the following code creates a handle that streams the implementation from a file.

```
FileInputStream transStream = new FileInputStream(
              "scripts"+File.separator+TRANSFORM_NAME+".xsl");
InputStreamHandle handle = new InputStreamHandle(transStream);
```

5. Install the transform and its metadata on MarkLogic Server.

```
transMgr.writeXSLTransform(TRANSFORM_NAME, handle, metadata);
```

6. Release the client if you no longer need the database connection.

```
client.release();
```

14.2 Using Transforms

Once you install a transform, you can apply it under the following circumstances:

- inserting a document into the database
- reading a document from the database
- retrieving search results
- testing for alerting rule matches

This section describes how to use transforms and includes the following topics:

14.2.1 Transforming a Document When Reading It

A read transform receives the document from the database as input and produces the document to be returned to the client application as output. Specify a read transform by including a `ServerTransform` object in your call to `DocumentManager.read`.

Use the following procedure to transform a document when reading it:

1. If you have not already done so, create a `DatabaseClient` for connecting to the database. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

2. Create an appropriate Document Manager for the to be transformed document.

   a. In this case, we use a `XMLDocumentManager`.

   ```
   XMLDocumentManager XMLDocMgr = client.newXMLDocumentManager();
   ```

   b. In this case, we use `JSONDocumentManager`.

   ```
   JSONDocumentManager JSONDocMgr = client.newJSONDocumentManager();
   ```

3. Create an appropriate read handle for the document's content.

   a. This example uses a `com.marklogic.client.io.DOMHandle` object.

   ```
   DOMHandle handleXML = new DOMHandle();
   ```

   b. This example uses a `com.marklogic.client.io.JacksonHandle` object.

   ```
   JacksonHandle handleJSON = new JacksonHandle();
   ```

4. Optionally, specify the expected MIME type for the content. This is only needed if the transform supports content negotiation and changes the content from one MIME type to another.

   ```
   handleXML.setMimetype("text/xml");
   //OR
   handleJSON.setMimetype("text/json");
   ```

5. Create a transform descriptor by creating a `ServerTransform` object. Specify the transform name and any parameter values expected by the transform.

   ```
   ServerTransform transform = new ServerTransform(TRANSFORM_NAME);
   transform.put("some-param", "value");
   ```

6. Read the document from the database, supplying the `ServerTransform` object. The read handle will contain the transformed content.

   ```
   XMLDocMgr.read(theDocURI, handleXML, transform);
   //OR
   JSONDocMgr.read(theDocURI, handleJSON, transform);
   ```

7. Release the database client if you no longer need the database connection.

   ```
   client.release();
   ```

14.2.2 Transforming a Document When Writing It

A write transform receives the document from the client application as input, and should produce the document to be written to the database as output. Specify a write transform by including a `ServerTransform` object in your call to `DocumentManager.write`.

Use the following procedure to transform a document when writing it:

1. If you have not already done so, create a `DatabaseClient` for connecting to the database. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

1. Create an appropriate Document Manager for the document. In this case, we use a `TextDocumentManager`.

```
TextDocumentManager writeMgr = client.newTextDocumentManager();
```

2. Create a handle to input data. For example, the following code streams the content from the file system.

```
FileInputStream docStream = new FileInputStream("/path/to/my.txt");
InputStreamHandle writeHandle = new InputStreamHandle(docStream);
```

3. Optionally, specify the MIME type for the content. This is only needed if the transform supports content negotiation and changes the content from one MIME type to another.

```
writeHandle.setMimetype("text/xml");
```

4. Create a transform descriptor by creating a ServerTransform object. Specify the transform name and any parameter values expected by the transform.

```
ServerTransform transform = new ServerTransform(TRANSFORM_NAME);
transform.put("drop-font-tags", "yes");
```

5. Write the content to the database. The transform is applied to the content on MarkLogic Server before inserting the document into the database.

```
String theDocURI = "/examples/mydoc.xml";
writeMgr.write(docId, writeHandle, transform);
```

6. Release the database client if you no longer need the database connection.

```
client.release();
```

14.2.3 Transforming Search Results

When you apply a transform to search results, the transform receives the search response data prepared by MarkLogic Server as input, and should produce the output to be returned to the client application. For example, if the response is in XML, the input is a document with a `<search:response/>` root element.

For details, see "Transforming Search Results" on page 173.

14.2.4 Transforming Alert Match Results

When you apply a transform to the results of an alerting match, the transform receives the match results prepared by MarkLogic Server as input, and should produce the output to be returned to the client application. For example, if the response is in XML, the input is a document with a `<rapi:rules>` root element.

For details, see "Transforming Alert Match Results" on page 260.

14.2.5 Overall Transform Administration

You can list all currently installed transform extensions by doing the following:

```
String result = transMgr.listTransforms(
   new StringHandle().withFormat(Format.XML)).get();
// format can be JSON as well
   new StringHandle().withFormat(Format.JSON)).get();
```

By default, calling `listTransforms()` rebuilds the transform metadata to ensure the metadata is up to date. If you find this refresh makes discovery take too long, you can disable the refresh by setting the `refresh` parameter to false:

```
String result = transMgr.listTransforms(
   new StringHandle().withFormat(Format.XML), false).get();
//Or
   new StringHandle().withFormat(Format.JSON), false).get();
```

Disabling the refresh can result in this request returning inaccurate information, but it does not affect the "freshness" or availability of the implementation of any transforms.

To delete a transform, effectively uninstalling it from the server do the following:

```
transMgr.deleteTransform(TRANSFORM_NAME);
```

14.2.6 Reading Transforms

To read the source code of an XQuery implemented transform into your application, do:

```
StringHandle textHandle = transMgr.readXQueryTransform(TRANSFORM_NAME,
   new StringHandle()); // can be any text handle
```

To read the source code of an XSLT implemented transform into your application, do:

```
XMLReadHandle xHandle = transMgr.readXSLTransform(TRANSFORM_NAME,
   new XMLReadhandle());
```

To read the source code of an Javascript implemented transform into your application, do:

```
JSONReadHandle jHandle =
transMgr.readJavascriptTransform(TRANSFORM_NAME,
   new JSONReadHandle());
```

14.2.7 Logging

Since it is a manager, you can define a `RequestLogger` object and start and stop logging client requests to the `TransformExtensionsManager`. For more information, see “Logging” on page 274

```
RequestLogger logger = client.newLogger(stream);
transformsMgr.startLogging(logger);
transformsMgr.stopLogging();
```

14.3 Writing Transformations

You can write transforms using server-side JavaScript, XQuery, or XSLT. The transform interface is shared across multiple MarkLogic client APIs, so you can use the same transforms with the Java Client API, Node.js Client API, and the REST Client API. For the interface definition, authoring guidelines, and example implementations, see Writing Transformations in the *REST Application Developer’s Guide*.

Warning Resource service extensions, transforms, row mappers and reducers, and other hooks cannot be implemented as JavaScript MJS modules.

15.0 Extending the Java API

You can extend the Java API in a variety of ways, including resource service extensions and evaluation of ad-hoc queries and server-side modules. This chapter covers the following topics:

15.1 Available Extension Points

The Java API offers several ways to extend and customize the capabilities using user-defined code that is either pre-installed on MarkLogic Server or supplied at request time. The following extension points are available:

- Content transformations: A user-defined transform function can be applied when documents are written to the database or read from the database; for details, see “Content Transformations” on page 282. You can also define custom replacement content generators for the patch feature; for details, see “Construct Replacement Data on the Server” on page 67.
- Search result customization: Customization opportunities include constraint parsers for string queries, search result snippet generation, and search result customization. For details, see “Searching” on page 144 and the *Search Developer’s Guide*.
- Resource service extensions: Define your own REST endpoints, accessible from Java using the `ResourceExtensionsManager` interface. Resource service extensions are covered in detail in this chapter. To get started, see “Introduction to Resource Service Extensions” on page 289.
- Ad-hoc query execution: Send an arbitrary block of XQuery or JavaScript code to MarkLogic Server for evaluation. For details, see “Evaluating an Ad-Hoc Query or Server-Side Module” on page 298.
- Server-side module evaluation: Evaluate user-defined XQuery or JavaScript modules after installing them on MarkLogic Server. For details, see “Evaluating an Ad-Hoc Query or Server-Side Module” on page 298.

15.2 Introduction to Resource Service Extensions

Resource service extensions extend the MarkLogic Java API by making XQuery and server-side JavaScript modules available for use from Java. A resource extension implements services for a server-side resource. For example, you can create a dictionary program resource extension that looks up words, checks spelling, and makes suggestions for not found words. The individual operations an application programmer may call, for example, `lookUpWords()`, `spellCheck()`, and so on, are the services that make up the resource extension.

The following are the basic steps to create and use a resource extension using the Java API:

1. Create an XQuery or JavaScript module that implements the services for the resource.

2. Install the resource service extension implementation in the modules database associated with the REST API instance using `com.marklogic.client.admin.ResourceExtensionsManager`.

3. Make your resource extension available to Java applications by creating a wrapper class that is a subclass of `com.marklogic.client.extensions.ResourceManager`. Inside this class, access the resource extension methods using a `com.marklogic.client.extensions.ResourceServices` object obtained through the `ResourceManager.getServices()` method.

4. Use the methods of your `ResourceManager` subclass to access the services provided by the extension from the rest of your application.

The key classes for resource extensions in the Java API are:

- `ResourceExtensionsManager`, which manages creation, modification, and deletion of resource service service extension implementations on the REST Server. You must connect to MarkLogic as a user with the `rest-admin` role to create and work with `ResourceExtensionsManager`.
- `ResourceManager`, the base class for classes that you write to provide client interfaces to resource services.
- `ResourceServices`, which supports calling the services for a resource. The resource services extension implementation must already be installed on the server via the `ResourceExtensionsManager` before `ResourceServices` can access it.

These objects are created via a `ServerConfigManager`, since resource services are associated with the server, not the database.

For a complete example of implementing and using a resource service extension, see `com.marklogic.client.example.cookbook.ResourceExtension` in the `example/` directory of your Java API installation.

15.3 Creating a Resource Extension

You can implement a resource service Extension using server-side JavaScript or XQuery. The interface is shared across multiple MarkLogic client APIs, so you can use the same extensions with the Java Client API, Node.js Client API, and the REST Client API. For the interface definition, authoring guidelines, and example implementations, see Extending the REST API in the *REST Application Developer's Guide*.

Warning Resource service extensions, transforms, row mappers and reducers, and other hooks cannot be implemented as JavaScript MJS modules.

15.4 Installing Resource Extensions

Before you can use a resource extension, you must install the implementation on MarkLogic Server as follows:

1. If your resource extension depends on additional library modules, install these dependent libraries on MarkLogic Server. For details, see "Managing Dependent Libraries and Other Assets" on page 295.

1. If you have not already done so, create a `DatabaseClient` for connecting to the database. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

2. If you have not already done so, create a `ResourceExtensionsManager` using `ServerConfigManager`.

```
ResourceExtensionsManager resourceMgr =
    client.newServerConfigManager().newResourceExtensionsManager();
```

3. Create a `com.marklogic.client.admin.ExtensionMetadata` object to hold the implementation language of your extension.

```
ExtensionMetadata metadata = new ExtensionMetadata();
metadata.setScriptLanguage(ExtensionMetadata.JAVASCRIPT);
```

4. Optionally, populate the ExtensionMetadataObject with your resource extension's metadata. You can set title, description, provider name, version, and expected parameters. For example:

```
metadata.setTitle("Spelling Dictionary Resource Services");
metadata.setDescription("This plugin supports spelling dictionaries");
metadata.setProvider("MarkLogic");
metadata.setVersion("0.1");
```

5. Optionally, define one or more objects containing method interface metadata using `com.marklogic.client.admin.ResourceExtensionsManager.MethodParameters`. The following example creates metadata for a GET method expecting one string parameter:

```
MethodParameters getParams = new MethodParameters(MethodType.GET);
getParams.add("my-uri", "xs:string?");
```

6. Create a handle (such as an input stream and a handle associated with it) to the extension's source code. For example:

```
FileInputStream myStream = new FileInputStream("sourcefile.sjs");
InputStreamHandle handle = new InputStreamHandle(myStream);
handle.set(myStream);
```

7. Install the extension by calling the `ResourceExtensionManager.writeServices()` method, supplying the extension name, the handle to the implementation, and any metadata objects. For example:

```
resourceMgr.writeServices(DictionaryManager.NAME,handle,metadata,getParams);
```

8. Release the client if you no longer need the database connection.

```
client.release();
```

The following code sample demonstrates the above steps. For a complete example, see `com.marklogic.client.example.cookbook.ResourceExtension` in the `example/` directory of your Java API distribution.

```
// create a manager for resource extensions
ResourceExtensionsManager resourceMgr =
  client.newServerConfigManager().newResourceExtensionsManager();

// specify metadata about the resource extension
ExtensionMetadata metadata = new ExtensionMetadata();
metadata.setScriptLanguage(ExtensionMetadata.XQUERY);
metadata.setTitle("Spelling Dictionary Resource Services");
metadata.setDescription("This plugin supports spelling dictionaries");
metadata.setProvider("MarkLogic");
metadata.setVersion("0.1");

// specify metadata about method interfaces
MethodParameters getParams = new MethodParameters(MethodType.GET);
getParams.add("my-uri", "xs:string?");

// acquire the resource extension source code
InputStream sourceStream = new FileInputStream("dictionary.xqy");

// create a handle on the extension source code
InputStreamHandle handle = new InputStreamHandle();
handle.set(sourceStream);

// write the resource extension to the database
resourceMgr.writeServices(DictionaryManager.NAME, handle,
                          metadata, getParams);
```

15.5 Deleting Resource Extensions

To delete a resource extension, call the `deleteServices()` method of `com.marklogic.client.admin.ResourceExtensionManager`. For example, assuming you have already obtained a `ResourceExtensionsManager` object, do the following:

```
resourceMgr.deleteServices(resourceName);
```

15.6 Listing Resource Extensions

To list all the installed extensions, use a handle as in the following example, which gets the extensions list in XML or JSON format:

```
String result = resourceMgr.listServices(
   new StringHandle().withFormat(Format.XML)).get();
//Or
String result = resourceMgr.listServices(
   new StringHandle().withFormat(Format.JSON)).get();
```

By default, calling `listServices()` rebuilds the extension metadata to ensure the metadata is up to date. If you find this refresh makes discovery take too long, you can disable the refresh by setting the `refresh` parameter to false:

```
String result = resourceMgr.listServices(
   new StringHandle().withFormat(Format.XML), false).get();
//Or
String result = resourceMgr.listServices(
   new StringHandle().withFormat(Format.JSON), false).get();
```

Disabling the refresh can result in this request returning inaccurate information, but it does not affect the "freshness" or availability of the implementation of any extensions.

15.7 Using Resource Extensions

After you install the extension as described in "Installing Resource Extensions" on page 290, create a wrapper class that exposes the functionality of the extension to your application. The wrapper class should be a subclass of `com.marklogic.client.extensions.ResourceManager`. In the implementation of your wrapper class, use `com.marklogic.client.extensions.ResourceServices` to invoke the GET, PUT, POST and/or DELETE methods of the resource extension.

Use these guidelines in implementing your wrapper subclass:

1. Before using any services, initialize your `ResourceManager` subclass by passing it to `com.marklogic.client.DatabaseClient.init()`. For example:

```
public class DictionaryManager extends ResourceManager {
    static final public String NAME = "dictionary";
    ...
```

```
    public DictionaryManager(DatabaseClient client) {
         super();

        // Initialize the Resource Manager via the Database Client
       client.init(NAME, this);
    }
    ...
}
```

2. To pass parameters to a resource extension method, create a `com.marklogic.client.util.RequestParameters` object and add parameters to it. Each parameter is represented by a parameter name and value. Use the parameter names defined by the resource extension. For example:

```
//Build up the set of parameters for the service call
RequestParameters params = new RequestParameters();
params.add("service", "dictionary");
params.add("uris", uris);
```

3. Obtain a `com.marklogic.com.extensions.ResourceServices` object through the inherited protected method `getServices()`. For example:

```
public class DictionaryManager extends ResourceManager {
    ...
    public Document[] checkDictionaries(String. . . uris) {

        ...
        // get the initialized service object from the base class
        ResourceServices services = getServices();
        ...
    }
}
```

4. Use the `get()`, `put()`, `post()`, and `delete()` methods of `ResourceServices` to invoke methods of the resource extension on the server. For example:

```
ResourceServices services = getServices();
ServiceResultIterator resultItr = services.get(params, mimetypes);
ServiceResultIterator resultItr = services.post(params, mimetypes);
ServiceResultIterator resultItr = services.put(params, mimetypes);
ServiceResultIterator resultItr = services.delete(params, mimetypes);
```

The results from calling a resource extension method are returned as either a `com.marklogic.client.extensions.ResourceServices.ServiceResultIterator` or a handle on the appropriate content type. Use a `ServiceResultIterator` when a method can return multiple items; use a handle when it returns only one. Resources associated with the results are not released until the associated handle is discarded or the iterator is closed or discarded.

The following code combines all the guidelines together in a sample application that exposes dictionary operations. For the complete example, see the Cookbook example `com.marklogic.client.example.cookbook.ResourceExtension` in the `example/` directory of your Java API distribution.

```
public class DictionaryManager extends ResourceManager {
    static final public String NAME = "dictionary";
    private XMLDocumentManager docMgr;

    public DictionaryManager(DatabaseClient client) {
         super();

        // Initialize the Resource Manager via the Database Client
       client.init(NAME, this);
   }

    // Our first Java implementation of a specific service from
    // the extension
    public Document[] checkDictionaries(String. . . uris) {
        //Build up the set of parameters for the service call
        RequestParameters params = new RequestParameters();
        // Add the dictionary service parameter
        params.add("service", "dictionary");
        params.add("uris", uris);

        String[] mimetypes = new String[uris.length];
        for (int i=0; i < uris.length; i++) {
            mimetypes[i] = "application/xml";
        }

        // get the initialized service object from the base class
        ResourceServices services = getServices();

        // call the service implementation on the REST Server,
        // returning a ResourceServices object
        ServiceResultIterator resultItr =
            services.get(params, mimetypes);

        //iterate over results, get content
        ...
        // release resources
        resultItr.close();

    }
    ...
}
```

15.8 Managing Dependent Libraries and Other Assets

This section covers installation and maintenance of XQuery libraries and other server-side assets used by your application. This includes dependent libraries needed by resource extensions and transformations, and replacement content generation functions usable for partially updates to documents and metadata.

The following topics are covered:

- Maintenance of Dependent Libraries and Other Assets
- Installing or Updating Assets
- Removing an Asset
- Retrieving an Asset List
- Retrieving an Asset

You can also manage assets using the MarkLogic REST API. For details, see Managing Dependent Libraries and Other Assets in the *REST Application Developer's Guide*.

15.8.1 Maintenance of Dependent Libraries and Other Assets

When you install or update a dependent library module or other asset as described in this section, the asset is replicated across your cluster automatically. There can be a delay of up to one minute between updating and availability.

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

Since dependent assets are installed in the modules database, they are removed when you remove the REST API instance if you include the modules database in the instance teardown.

If you installed assets in a REST API instance using MarkLogic 6, they cannot be managed using the `/ext` service unless you re-install them using `/ext`. Reinstalling the assets may require additional changes because the asset URIs will change. If you choose not to migrate such assets, continue to maintain them according to the documentation for MarkLogic 6.

15.8.2 Installing or Updating Assets

Follow this procedure to install or update a library module or other asset in the modules database associated with your REST Server. If the REST Server is part of a cluster, the asset is automatically propagated throughout the cluster.

> **Note:** The modules database path under which you install an asset must begin with `/ext/`.

1. If you have not already done so, connect to the database, storing the connection in a `com.marklogic.client.DatabaseClient` object. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

1. If you have not already done so, create a `com.marklogic.client.admin.ExtensionLibrariesManager`. Note that the method for doing so is associated with a `ServerConfigManager`.

```
ExtensionLibrariesManager libMgr =
    client.newServerConfigManager().newExtensionLibrariesManager();
```

2. Associate a handle with the asset.

   a. The following example associates a FileHandle with the text file containing an XQuery module.

```
FileHandle handle =
    new FileHandle(new File("module.xqy")).withFormat(Format.TEXT));
```

   b. The following example associates a FileHandle with the text file containing an Javascript module.

```
FileHandle handle =
    new FileHandle(new File("module.sjs")).withFormat(Format.TEXT));
```

3. Install the module in the modules database by calling `ExtensionLibrariesManager.write()`. For example:

```
libMgr.write("/ext/my/path/to/my/module.xqy", handle);
//Or
libMgr.write("/ext/my/path/to/my/module.sjs", handle);
```

You can also specify asset-specific permissions by passing an `ExtensionLibraryDescriptor` instead of a simple path string to `ExtensionLibrariesManager.write()`. The following example uses an descriptor:

```
ExtensionLibraryDescriptor desc = new ExtensionLibraryDescriptor();
desc.setPath("/ext/my/path/to/my/module.xqy");
//Or
desc.setPath("/ext/my/path/to/my/module.sjs");

desc.addPermission("my-role", "my-capability");
...
libMgr.write(desc, handle);
```

To use a dependent library installed with `/ext` in your extension or transform module, use the same URI under which you installed the dependent library. For example, if a dependent library is installed with the URI `/ext/my/domain/my-lib.xqy`, then the extension module using this library should include an import of the form:

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

In Javascript:

```
const dep = require("/ext/my/domain/my-lib.sjs");
```

15.8.3 Removing an Asset

To remove an asset from the modules database associated with the REST Server, call `com.marklogic.client.admin.ExtensionLibrariesManager.delete()`. For example:

```
DatabaseClient client = DatabaseClientFactory.newClient(...);
ExtensionLibrariesManager libMgr =
    client.newServerConfigManager().newExtensionLibrariesManager();

libMgr.delete("/ext/my/path/to/my/module.xqy");
//Or
libMgr.delete("/ext/my/path/to/my/module.sjs");
```

You can also call `delete()` with a `ExtensionLibraryDescriptor`.

If the path passed to `delete()`, whether by `String` or descriptor, is a database directory path, all assets in the directory are deleted. If the path is a single asset, just that asset is deleted.

15.8.4 Retrieving an Asset List

You can retrieve a list of all the assets installed in the modules database associated with the REST Server by calling `com.marklogic.client.admin.ExtensionsLibraryManager.list`(). If you call `list()` with no parameters, you get a list of `ExtensionLibraryDescriptor` objects for all assets. If you call `list()` with a path, you get a similar list of descriptors for all assets installed in that database directory.

The following code snippet retrieves descriptors for all installed assets and prints the path of each one to stdout.

```
DatabaseClient client = DatabaseClientFactory.newClient(...);
ExtensionLibrariesManager libMgr =
    client.newServerConfigManager().newExtensionLibrariesManager();

ExtensionLibraryDescriptor[] descriptors = libMgr.list();
for (ExtensionLibraryDescriptor descriptor : descriptors) {
    System.out.println(descriptor.getPath());
}
```

15.8.5 Retrieving an Asset

To retrieve the contents of an asset installed in the modules database associated with a REST Server, call `com.marklogic.client.admin.LibrariesExtensionManager.read()`. You must first create a handle to receive the contents.

The following code snippet reads the contents of an XQuery library module into a string:

```
DatabaseClient client = DatabaseClientFactory.newClient(...);
ExtensionLibrariesManager libMgr =
    client.newServerConfigManager().newExtensionLibrariesManager();

StringHandle handle =
    libMgr.read("/ext/my/path/to/my/module.xqy", new StringHandle());
```

The following code snippet reads the contents of an Javascript library module into a string:

```
DatabaseClient client = DatabaseClientFactory.newClient(...);
ExtensionLibrariesManager libMgr =
    client.newServerConfigManager().newExtensionLibrariesManager();

StringHandle handle =
    libMgr.read("/ext/my/path/to/my/module.sjs", new StringHandle());
```

15.9 Evaluating an Ad-Hoc Query or Server-Side Module

The `com.marklogic.client.eval.ServerEvaluationCall` enables you to send blocks of JavaScript and XQuery to MarkLogic Server for evaluation or to invoke an XQuery or JavaScript module installed in the modules database. This is equivalent to calling the builtin server functions `xdmp:eval` or `xdmp:invoke` (XQuery), or `xdmp.eval` or `xdmp.invoke` (JavaScript).

This section covers the following related topics:

- Security Requirements
- Basic Step for Ad-Hoc Query Evaluation
- Basic Steps for Module Invocation
- Specifying External Variable Values
- Interpreting the Results of Eval or Invoke

15.9.1 Security Requirements

Evaluating an ad-hoc query on MarkLogic Server requires the following privileges or the equivalent:

- `http://marklogic.com/xdmp/privileges/xdmp-eval`
- `http://marklogic.com/xdmp/privileges/xdmp-eval-in`
- `http://marklogic.com/xdmp/privileges/xdbc-eval`

- `http://marklogic.com/xdmp/privileges/xdbc-eval-in`

Invoking a module on MarkLogic Server requires the following privileges or the equivalent:

- `http://marklogic.com/xdmp/privileges/xdmp-invoke`
- `http://marklogic.com/xdmp/privileges/xdmp-invoke-in`
- `http://marklogic.com/xdmp/privileges/xdbc-invoke`
- `http://marklogic.com/xdmp/privileges/xdbc-invoke-in`

15.9.2 Basic Step for Ad-Hoc Query Evaluation

Follow this procedure to evaluate an Ad-Hoc XQuery or JavaScript query on MarkLogic Server. You must use a user that has the privileges listed in "Security Requirements" on page 298.

1. If you have not already done so, create a `DatabaseClient` for connecting to the database. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

1. Create a `ServerEvaluationCall` object.

```
ServerEvaluationCall theCall = client.newServerEval();
```

2. Associate your ad-hoc query with the call object. You can specify the query using a `String` or a `TextWriteHandle`.

   a. For a JavaScript query, pass in the query text using `ServerEvaluationCall.javascript`:

```
String query = "word1 \" \" + word2";
theCall.javascript(query);
```

   b. For an XQuery query, pass in the query text using `ServerEvaluationCall.xquery`.

```
String query =
  "xquery version '1.0-ml';" +
  "declare variable $word1 as xs:string external;" +
  "declare variable $word2 as xs:string external;" +
  "fn:concat($word1, ' ', $word2)";
theCall.xquery(query);
```

3. If the query expects input variable values, supply them using `ServerEvaluationCall.addVariable`. For details, see "Specifying External Variable Values" on page 301.

```
theCall.addVariable("word1", "hello");
theCall.addVariable("word2", "world");
```

4. Send the query to MarkLogic Server for evaluation by calling `ServerEvaluationCall.eval` or `ServerEvaluationCall.evalAs`. For details, see “Interpreting the Results of Eval or Invoke” on page 302.

```
String response = theCall.evalAs(String.class);
```

5. Release the client if you no longer need the database connection.

```
client.release();
```

The following code puts these steps together into a single block.

```
DatabaseClient client = DatabaseClientFactory.newClient(
   host, port, new DigestAuthContext(username, password));
ServerEvaluationCall theCall = client.newServerEval();
String query = "word1 \" \" + word2";

String result = theCall.javascript(query)
                  .addVariable("word1", "hello")
                  .addVariable("word2", "world")
                  .evalAs(String.class);
```

15.9.3 Basic Steps for Module Invocation

Note: A JavaScript MJS module can be invoked through the `/v1/invoke` endpoint. This is the preferred method.

Note: A data service endpoint can be implemented as a JavaScript MJS module. This is the preferred method.

You can invoke an arbitrary JavaScript or XQuery module installed in the modules database associated with the REST API instance by setting a module path on a ServerEvaluationCall object and then calling `ServerEvaluationCall.eval` or `ServerEvaluationCall.evalAs`. The module path is resolved using the rules described in “Rules for Resolving Import, Invoke, and Spawn Paths” on page 87 in the *Application Developer’s Guide*.

You can install your module is using `com.marklogic.client.admin.ExtensionLibrariesManager`. For details, see “Installing or Updating Assets” on page 295. If you install your module using the `ExtensionLibrariesManager` interface, your module path will always being with “/ext/”.

Follow this procedure to invoke an XQuery or JavaScript module pre-installed on MarkLogic Server. You must use a user that has the privileges listed in “Security Requirements” on page 298.

1. If you have not already done so, create a `DatabaseClient` for connecting to the database. For example, if using digest authentication:

```
DatabaseClient client = DatabaseClientFactory.newClient(
    host, port, new DigestAuthContext(username, password));
```

2. Create a `ServerEvaluationCall` object.

```
ServerEvaluationCall invoker = client.newServerEval();
```

3. Associate your module with the call object by setting the module path.

```
invoker.modulePath("/my/module/path.sjs");
```

4. If the query expects input variable values, supply them using `ServerEvaluationCall.addVariable`. For details, see "Specifying External Variable Values" on page 301.

```
invoker.addVariable("word1", "hello");
invoker.addVariable("word2", "world");
```

5. Invoke the module on MarkLogic Server by calling `ServerEvaluationCall.eval` or `ServerEvaluationCall.evalAs`. For details, see "Interpreting the Results of Eval or Invoke" on page 302.

```
String response = invoker.evalAs(String.class);
```

6. Release the client if you no longer need the database connection.

```
client.release();
```

The following code puts these steps together into a single block.

```
DatabaseClient client = DatabaseClientFactory.newClient(
   host, port, new DigestAuthContext(username, password));
ServerEvaluationCall invoker = client.newServerEval();

String result = invoker.modulePath("/ext/invoke/example.sjs")
                  .addVariable("word1", "hello")
                  .addVariable("word2", "world")
                  .evalAs(String.class);
```

15.9.4 Specifying External Variable Values

You can pass values to an ad-hoc query or invoked module at runtime using external variables. Specify the variable values using `ServerEvaluationCall.addVariable`. or `ServerEvaluationCall.addVariableAs`.

Use `addVariable` for simple value types, such as `String`, `Number`, and `Boolean` and values with a suitable `AbstractWriteHandle`, such as `DOMHandle` for XML and `JacksonHandle` for JSON. For example:

```
ServerEvaluationCall theCall = client.newServerEval();
...
theCall.addVariable("aString", "hello")
```

```
        .addVariable("aBool", true)
        .addVariable("aNumber", 3.14);
```

Use `addVariableAs` for other complex value types such as objects. For example, the following code uses a Jackson object mapper to set an external variable value to a JSON object that can be used as a JavaScript object by the server-side code:

```
theCall.addVariableAs("anObj",
    new ObjectMapper().createObjectNode().put("key", "value"))
```

If you're evaluating or invoking XQuery code, you must declare the variables explicitly in your ad-hoc query or module. For example, the following XQuery prolog declares two external string-valued variables whose values can be supplied at runtime.

```
xquery version "1.0-ml";
declare variable $word1 as xs:string external;
declare variable $word2 as xs:string external;
...
```

If your XQuery external variables are in a namespace, use `ServerEvaluationCall.addNamespac`e to associate a prefix with the namespace, and then use the namespace prefix in the variable name passed to `ServerEvaluationCall.addVariable`. For example, given the following ad-hoc query:

```
xquery version "1.0-ml";
declare namespace my = "http://example.com";
declare variable $my:who as xs:string external;
fn:concat("hello", " ", $my:who)
```

Set the variable values as follows:

```
theCall.addNamespace("my", "http://example.com")
       .addVariable("my:who", "me")
       ...
```

15.9.5 Interpreting the Results of Eval or Invoke

You can request results in the following ways:

- If you know the ad-hoc query or invoked module returns a single value of a simple known type, use `ServerEvaluationCall.evalAs`. For example, if you know an ad-hoc query returns a single String value, you can evaluate it as follows:

  ```
  String result = theCall.evalAs(String.class);
  ```

- Pass an `AbstractReadHandle` to `ServerEvaluationCall.eval` to process a single result through a handle. For example:

  ```
  DOMHandle result = theCall.eval(new DOMHandle());
  //Or
  ```

```
JacksonHandle result = theCall.eval(new JacksonHandle());
```

- If the query or invoked module can return multiple values or you do not know the return type, use `ServerEvaluationCall.eval` with no parameters to return an `EvalResultIterator`. For example:

```
EvalResultIterator result = theCall.eval();
```

When you use an `EvalResultIterator`, each value is encapsulated in a `com.marklogic.client.eval.EvalResult` that provides type information and accessors for the `value`. The `EvalResult.format` method provides abstract type information, such as text, binary, json, or xml. The `EvalResult.getType` method provides more detailed type information, when available, such as string, integer, decimal, or date. Detailed type information is not always available.

The table below maps common server-side value types to the values you can expect to their corresponding `com.marklogic.client.io.Format` (from `EvalResult.format`) and `EvalResult.Type` (from `EvalResult.getType`).

Value Type	Format	Type
`document-node[object-node()]`	`Format.JSON`	`Type.JSON`
`object-node()`	`Format.JSON`	`Type.JSON`
`document-node[array-node()]`	`Format.JSON`	`Type.JSON`
`array-node()`	`Format.JSON`	`Type.JSON`
`map:map`	`Format.JSON`	`Type.JSON`
`json:array`	`Format.JSON`	`Type.JSON`
`document-node[element()]`	`Format.XML`	`Type.XML`
`element()`	`Format.XML`	`Type.XML`
`document-node[binary()]`	`Format.BINARY`	`Type.BINARY`
`binary()`	`Format.BINARY`	`Type.BINARY`
`document-node[text()]`	`Format.TEXT`	`Format.TEXT`
`text()`	`Format.TEXT`	`Format.TEXT`

Value Type	Format	Type
any atomic value	`Format.TEXT`	corresponding type, such as `Format.BOOLEAN` or `Format.INTEGER`.
`JavaScript string`	`Format.TEXT`	`Format.STRING`
`JavaScript number`	`Format.TEXT`	`Format.DECIMAL`, a derived type such as `FORMAT.INTEGER`, or `Format.STRING` (for infinity)
`JavaScript boolean`	`Format.TEXT`	`Format.BOOLEAN`

16.0 Creating Data Services Using the MarkLogic Java Development Tools

Data Services is a convenient way to integrate MarkLogic into an existing enterprise environment. A data service is a fixed interface over the data managed in MarkLogic expressed in terms of the consuming application. Data services can run queries ("Find eligible insurance plans for an applicant"), updates ("Flag this claim as fraudulent"), or both ("Adjust the rates of plans that haven't made claims in the last year"). A MarkLogic cluster can support dozens or even hundreds of different data services operating over the data and metadata managed in a data hub.

A data service is different from a generic query interface, like JDBC or ODBC, which typically operates at the physical layer of the database. Architecturally, a data service is more like a remote procedure call or a stored procedure. The data service allows the service developer to obscure the physical layout of the data and constrain or enhance queries and updates with business logic.

MarkLogic provides a rich scripting environment as part of the DBMS. The developer implements data services using either JavaScript or XQuery. MarkLogic supports JavaScript and XQuery runtimes. MarkLogic optimizes this code to run close to the data, minimizing data transfer and leveraging cluster-wide indexes and caches.

16.1 Advantages of Data Services

- Avoid unnecessary round-trips by encapsulating the data logic, ensuring that service implementations run close to the data.
- Reduce custom plumbing code by handling network and data marshalling transparently.
- Reduce the potential for API drift as requirements and implementations change by enforcing strongly typed interfaces.

The Java Client API supports physical operations on the database. In particular, the Java Client API provides DocumentManager (and its derivations) and QueryManager to write, read, or query for documents and their metadata at the Uris identifying the documents in the database. Where a transaction must span multiple requests, the client uses a physical Transaction object.

Proxy services complement these physical operations with logical operations. The Java middle-tier invokes endpoints, passing and receiving values. The endpoint is entirely responsible for the implementation of the operation against the database - including the reading and writing of values. Where an operation must interleave middle-tier and e-node tasks, the client uses a logical session represented by a SessionState object (as described later).

The Java Client API and proxy services connect with the database in the same way. Both use the DatabaseClientFactory class to instantiate a DatabaseClient object for use in requests.

A REST server used for the Java Client API can coexist with proxy services, provided the user abides by the following conditions:

1. Do not try to use proxy services on port 8000.

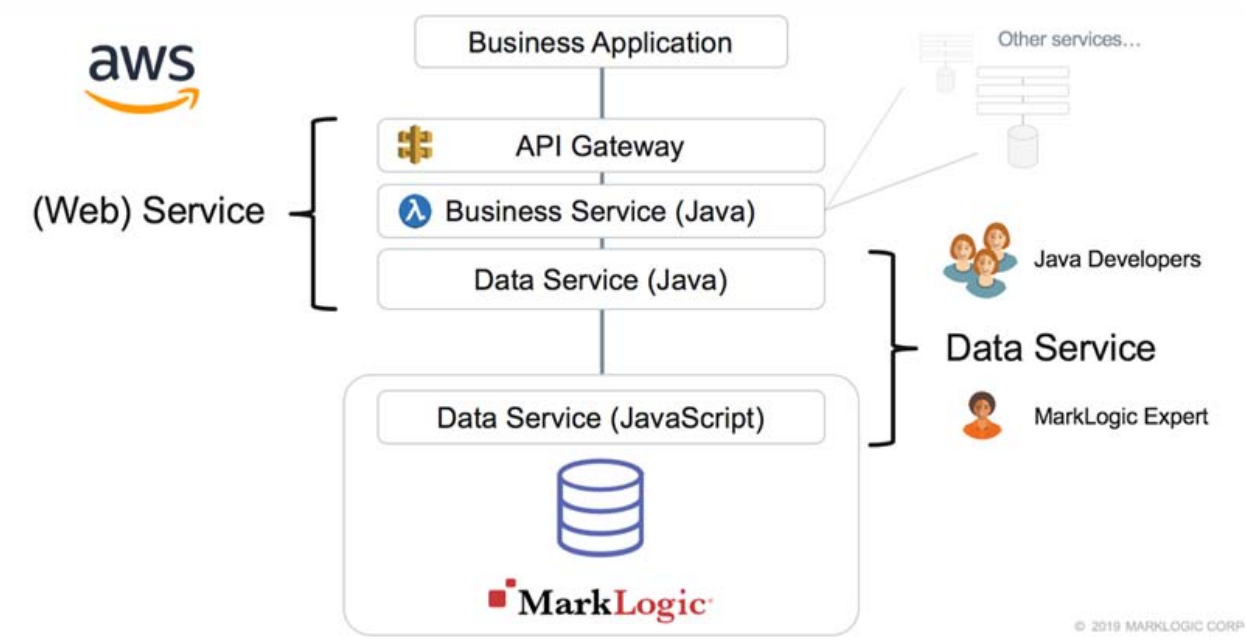
2. You must avoid filename collisions by using a different directory than the one used by the REST API.

One way to avoid such collisions would be to establish a convention such as using a "/ds" directory for all data services.

Note: The middle-tier client cannot specify the database explicitly when creating a DatabaseClient but, instead, must use the default database associated with the App Server.

16.2 Where Data Service Fit Within the Enterprise Stack

The diagram below illustrates how MarkLogic Data Services fits within the enterprise development stack.



Enterprise middle-tier business logic generally integrates many services: data services from a MarkLogic cluster as well as services from other providers. This service orchestration and business logic happen at a layer above the data infrastructure, outside of a particular service provider. The flexibility to mix and match services and to decouple providers and consumers is one of the benefits of a service-oriented architecture:

16.2.1 How it Works

You declare a function signature for each endpoint that implements a data service.

From a set of such declarations, the development tools generate a Java proxy service class that encapsulates the execution of the endpoints including the marshalling and transport of the request and response data. The middle-tier business logic can then call the methods of the generated class.

A MarkLogic data service consists of three main components:

- Endpoint Declaration: This is a JSON document used to specify the name of the service as well as the names and data types of the inputs and outputs.
- Endpoint Proxy: Code that exposes the service definition in Java, automatically invoking the services remotely against a MarkLogic cluster for the caller.
- Endpoint Module: This is the implementation of a data service in MarkLogic as a JavaScript or XQuery module.

By declaring the data tier functions needed by the middle-tier business logic, the endpoint declaration establishes a division of responsibility between the Java middle-tier developer and the data service developer. The endpoint declaration acts as a contract for collaboration between the two roles.

It is the responsibility of the end point service developer to limit access to the Data Services assets by adding the necessary security asserts (using xdmp.securityAssert or xdmp:security-assert functions) to test for privileges.

16.2.2 Prerequisites

To create a proxy service, you need a Java JDK environment with Gradle and the following MarkLogic software components:

- Current version of MarkLogic Server
- Current version of MarkLogic Client Java API
- Current version of ml-gradle

The MarkLogic Java development tools are available as a Gradle plugin.

This document assumes that you are familiar with Java and Gradle.

If you are unfamiliar with Gradle, the ml-gradle project lists some resources for getting started:

Installing and learning Gradle

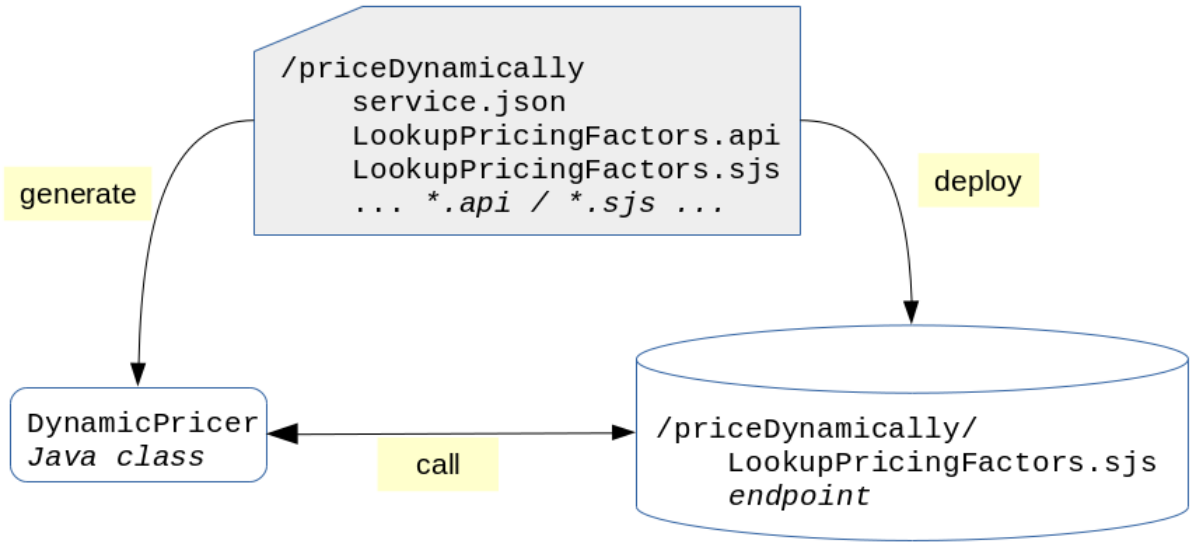
Typically, you create one Gradle project directory for all of the work on proxy services for one content database.

16.2.3 Relation to the Java Client API

The MarkLogic Java Client API includes development tooling and runtime proxies so that a Java application can access custom data services in a MarkLogic cluster. The Java application calls strongly typed services running in the databases as if they were "out of the box" Java methods. The API handles the underlying network protocol and data marshalling.

16.3 Creating a Proxy Service

From the proxy service source files, you generate Java methods that call endpoint modules deployed to the modules database:



The development process consists of the following steps:

1. Set up a MarkLogic App Server
2. Create a proxy service directory within the Gradle project directory
3. Create a file to declare the service
4. Create files to declare one or more endpoint proxies for the service
5. Implement the module for each endpoint proxy
6. Deploy the proxy service directory to the modules database of the App Server
7. Generate the Java Class from the proxy service declaration.

16.3.1 Setting Up an App Server for the Proxy Service

Typically, you set up a single App Server for all of the proxy services for a content database.

The App Server configuration must have the following characteristics:

- Must have a modules database.
- Must have a root of /.

You cannot use the following App Servers, created by default when you install MarkLogic:

- The REST/HTTP/XDBC App Server on port 8000
- The Admin API App Server on port 8001
- The REST Management API App Server on port 8002

As noted above, you are also able to use a REST server (that is, an App Server created for the Client REST API).

> **Note:** Data services can reside on REST and non-REST App Servers.

To make creation and configuration of the App Server and its modules database, you should manage a repeatable operation in a version control system. You can also put resources in the Gradle project directory and use ml-gradle to operate on those resources.

See Getting started for a step-by-step guide to this Gradle procedure.

As an easy expedient when learning about MarkLogic, you can instead configure the App Server and modules database manually. As a long-term practice, however, we recommend a repeatable approach using Gradle.

16.3.2 Creating the Proxy Service Directory

For each proxy service, you create a separate subdirectory under the Gradle project directory.

Each proxy service directory holds all of the resources required to support the proxy service, including:

- The service declaration
- The endpoint proxy declarations
- The module called by each endpoint proxy
- Any server-side libraries to support the endpoint modules

For easier deployment to the modules database using ml-gradle, you should create the proxy service directory under the `src/main/ml-modules/root project` subdirectory. If you are working under a MarkLogic ReST server application, you should use the following proxy service directory: `src/main/ml-modules/root/ds`.

For instance, a project might choose to provide the priceDynamically service in the following proxy service directory:

```
src/main/ml-modules/root/inventory/priceDynamically
```

16.3.3 Declaring the Proxy Service

The proxy service directory must contain exactly one service declaration file. The service declaration file must have the name service.json

The service declaration consists of a JSON object with the following properties:

Table 1: Service Declaration File Properties

Property	Declares
endpointDirectory	The directory path for the installed endpoint modules within the modules database.
$javaClass	The full name of the generated service class including the package qualification.
desc	Optional; plain text documentation for the service (emitted as Java-Doc by the generated class).
comment	Optional; can contain an object, array, or value with developer comments about the declaration.

The following example declares the /inventory/priceDynamically/ directory as the address of the endpoints in the modules database and declares `com.some.business.inventory.DynamicPricer` as the generated Java class:

```
{
   "endpointDirectory" : "/inventory/priceDynamically/",
   "$javaClass"        : "com.some.business.inventory.DynamicPricer"
}
```

Conventionally, the value of the endpointDirectory property should be the same as the path of the proxy service directory under the special ml-gradle src/main/ml-modules/root directory (so, the service directory for this service.json file would conventionally be `src/main/ml-modules/root/inventory/priceDynamically`).

The endpoint directory value should include the leading / and should resemble a Linux path.

After declaring the service, you populate it with endpoint proxy declarations

16.3.4 Declaring the Endpoint

The name, parameters, and return value for each endpoint is declared in a file with the `.api` extension in the service directory. The file contains a JSON data structure with the following properties:

Table 2: Endpoint Properties

Property	Declares
functionName	The name used to call the endpoint, which must match the name (without the .api extension) of the declaration file.
desc	Optional; plain text documentation for the endpoint (emitted as JavaDoc).
params	Optional; an array specifying the parameters of the endpoint; omitted for endpoints with no parameters. Parameter objects have name, desc, datatype, nullable, and multiple properties.
return	Optional; an object specifying the endpoint return value; omitted for endpoints with no return value. The child object has desc, datatype, nullable, and multiple properties.
errorDetail	Optional; specifies a value from the following enumeration to control whether error responses include stack traces: • log: (the default) to log the stack trace on the server but not return the stack trace to the middle-tier. • return: to include the stack trace in the exception on the middle-tier as well as log it on the server.

The endpoint declaration is used both to generate a method in a Java class to call on the middle-tier and to unmarshal the request and marshal the response when the App Server executes the endpoint module.

Note: The `.api` file for proxy endpoint must be loaded into the modules database with the endpoint module.

The following sections provide more detail about the params and return declarations

16.3.4.1 Structure of a Parameter Definition

A parameter definition in the params property is an array with the following properties:

Table 3: Parameter Definitions

Property	Declares
name	The name of the parameter
desc	Optional; a description of the parameter to include in JavaDoc.
datatype	The datatype of the parameter (see Server Data Types for Values).
nullable	Optional; whether the parameter can be null (defaulting to false).
multiple	Optional; whether the parameter can have more than one value (defaulting to false).

16.3.4.2 Structure of the Return Type Definition

The return property of an endpoint declaration is an object with the following properties:

Table 4: Return Type Definitions

Property	Declares
desc	Optional; a description of the return to include in JavaDoc.
datatype	The datatype of the return (see Server Data Types for Values).
nullable	Optional; whether the return can be null (defaulting to false).
multiple	Optional; whether the return can have more than one value (defaulting to false).

16.3.4.3 Example of an Endpoint Proxy

The following example declares that the lookupPricingFactors endpoint has two required parameters as well as a required return value:

```
{
  "functionName" : "lookupPricingFactors",
  "params" : [ {
    "name" : "productCode",
    "datatype" : "string"
  }, {
```

```
  "name" : "customerId",
  "datatype" : "unsignedLong"
} ],
"return" : {
  "datatype" : "jsonDocument"
}}
```

16.3.4.4 Server Data Types for Values

You can specify atomic or node server data types for parameters and return values:

Table 5: Server Data Types

Category	Data Types
atomics	boolean, date, dateTime, dayTimeDuration, decimal, double, float, int, long, string, time, unsignedInt, unsignedLong
nodes	array, object, binaryDocument, jsonDocument, textDocument, xmlDocument

The data types with direct equivalents in the Java language atomics are represented with those Java classes by default. These data types `include boolean, double, float, int, long, string, unsignedInt,` and `unsignedLong`. For instance, a Java `Integer` represents an `int`. Likewise, the unsigned methods of the Java `Integer` and `Long` classes can manipulate the `unsignedInt` and `unsignedLong` types.

By default, a Java String represents the other atomic types (including date, `dateTime,` and `dayTimeDuration, decimal` and `time`).

Other server atomic data types can be passed as a string and cast using the appropriate constructor on the server.

A binaryDocument value is represented as an InputStream by default. All other node data types are represented as a Reader by default.

The `array` and `object` data types differ from the `jsonDocument` data type in not having a document node at the root, which can provide a more natural and efficient JSON value for manipulating in SJS (Server-Side JavaScript).

16.3.4.5 Mapping Values to Alternative Java Classes

Instead of the default Java representation, an alternative Java class may represent some server data types. For example, a String can represent a `date` by default, but you can choose to use `java.time.LocalDate` instead.

To specify an alternative Java class, supply the fully qualified class name in the `$javaClass` property of a parameter or return type. You must still specify the server data type in the `datatype` property.

The following table lists server data types with their available alternative representations:

Table 6:

Server Data Type	Mappable Java Classes
date	java.time.LocalDate
dateTime	java.util.Date, java.time.LocalDateTime, java.time.OffsetDateTime
dayTimeDuration	java.time.Duration
decimal	java.math.BigDecimal
time	java.time.LocalTime, java.time.OffsetTime
array	java.io.InputStream, java.io.Reader, java.lang.String, com.fasterxml.jackson.databind.node.ArrayNode, com.fasterxml.jackson.core.JsonParser
object	java.io.InputStream, java.io.Reader, java.lang.String, com.fasterxml.jackson.databind.node.ObjectNode, com.fasterxml.jackson.core.JsonParser
binaryDocument	java.io.InputStream
jsonDocument	java.io.InputStream, java.io.Reader, java.lang.String, com.fasterxml.jackson.databind.JsonNode, com.fasterxml.jackson.core.JsonParser
textDocument	java.io.InputStream, java.io.Reader, java.lang.String

Table 6:

Server Data Type	Mappable Java Classes
xmlDocument	java.io.InputStream, java.io.Reader, java.lang.String, org.w3c.dom.Document, org.xml.sax.InputSource, javax.xml.transform.Source, javax.xml.stream.XMLEventReader, javax.xml.stream.XMLStreamReader

The following example represents the occurred date parameter as a Java LocalDate and represents the returned JSON document as a Jackson JsonNode.

```
{
  "functionName" : "produceReport",
  "params":[ {
    "name":"id", "datatype":"int"
  }, {
    "name":"occurred", "datatype":"date",
      "$javaClass":"java.time.LocalDate"
  } ],
  "return" : {
    "datatype":"jsonDocument",
      "$javaClass":"com.fasterxml.jackson.databind.JsonNode"}
  }
}
```

16.3.4.6 Calling Endpoints in a Session

Ordinarily, the database server does not keep any state associated with a call to an endpoint (with the obvious but important exception of documents persisted in the database). When the middle-tier sends all of the input needed for a data tier operation, the operation can be completed in a single request. This approach typically maximizes performance and minimizes load.

Some operations, however, use sessions that coordinate multiple requests. Examples of such operations include:

- Interleaving middle-tier and data tier operations (such as multi-statement transactions in which the middle-tier logic must be inserted between the initial database change and a subsequent database change)
- Implementing Host affinity with an e-node when working with a load balancer to exploit query caches on the e-node.

You can handle these edge cases by calling the endpoints in a session. If an endpoint needs to participate in a session, its declaration must include exactly one parameter with the session data type. The session parameter may be nullable but not multiple (and may never be a return value).

```
// A simple example of the use of "session" in an .api declaration:
{
  "functionName" : "SessionChecks",
  "params" : [ {
    "name" : "api_session",
    "datatype" : "session",
    "desc" : "Holds the session object"
  },
...
}
```

If at least one endpoint has a session parameter, the generated class provides a `newSessionState()` factory that returns a SessionState object. The expected pattern of use:

- Construct a new session object when needed.
- Pass the same session object on each call that should execute in the same session.

Where endpoint modules need to participate in the same session, you must declare a session parameter for each of the corresponding endpoint proxies and document the expectations for coordination in the middle-tier consumer code. For instance, if one session endpoint starts a multi-statement transaction, another continues work in the same multi-statement transaction, and a third commits the transaction, the documentation should explain that each call would use the same session, as well as the sequence in which to make the calls.

The proxy service does not end the session explicitly. Instead, the session eventually times out (as controlled by the configuration of the App Server). The middle-tier code is responsible for calling an endpoint module to commit a multi-statement transaction before the session expires.

16.3.5 Providing the Module for an Endpoint Proxy

Note: A JavaScript MJS module can be invoked through the `/v1/invoke` endpoint. This is the preferred method.

Note: A data service endpoint can be implemented as a JavaScript MJS module. This is the preferred method.

You implement the data operations for an endpoint proxy in an XQuery or Server-Side JavaScript endpoint module. The proxy service directory of your project must contain exactly one endpoint module for each endpoint declaration in your service.

An endpoint module must have the same base name as the endpoint declaration. In addition, it must have either an `.xqy` (XQuery) or `.sjs` (JavaScript) extension, depending on the implementation language.

The App Server handles marshalling and unmarshalling for the endpoint. That is, the endpoint does not interact directly with the transport layer (which, internally, is currently HTTP).

The endpoint module must define an external variable for each parameter in the endpoint declaration. In an SJS endpoint, use a var statement at the top of the module with no initialization of the variable. In an XQuery endpoint, use an external variable with the server data type corresponding to the parameter data type.

The endpoint module must also return a value with the appropriate data type.

For the `lookupPricingFactors` endpoint whose declaration appears earlier, the SJS endpoint module would resemble the following fragment:

```
'use strict';
var productCode; // an xs.string value
var customerId;  // an xs.unsignedLong value
... /* the code that produces a JSON document as output */
```

The equivalent XQuery endpoint module would resemble the following fragment:

```
xquery version "1.0-ml";
 declare variable $productCode as xs:string       external;
 declare variable $customerId as xs:unsignedLong external;
 declare option xdmp:mapping "false";
 ... (: the code that produces a JSON document as output :)
```

As a convenience, you can use the `initializeModule` Gradle task to create the skeleton for an endpoint module from an endpoint declaration. You specify the path (relative to the project directory) for the endpoint declaration with the `endpointDeclarationFile` property and the module extension (which can be either `sjs` or `xqy`) with the `moduleExtension` property.

Your Gradle build script should apply the `com.marklogic.ml-development-tools` plugin. You can execute the Gradle task using any of the following techniques:

- By setting the properties in the `gradle.properties` file and specifying the `initializeModule` task on the gradle command line
- By specifying the properties with the -P option as well as the `initializeModule` task on the gradle command line
- By supplying a build script with a custom task of the `com.marklogic.client.tools.gradle.ModuleInitTask` type

For the command-line approach, the Gradle build script would resemble the following example:

```
plugins {
    id 'com.marklogic.ml-development-tools' version '4.1.1'
}
```

On Linux, the command-line for initializing the lookupPricingFactors.sjs SJS endpoint module from the lookupPricingFactors.api endpoint declaration might resemble the following example:

```
gradle \
-PendpointDeclarationFile=src/main/ml-modules/root/inventory/priceDyna
mically/lookupPricingFactors.api \
    -PmoduleExtension=sjs \
    initializeModule
```

Once each `.api` endpoint declaration file has an equivalent endpoint module to implement the endpoint, you can load the proxy service directory into the modules database and generate the proxy service Java class. (The Java code generation checks the endpoint module in the service directory to determine how to invoke the endpoint.)

16.3.6 Deploying a Proxy Service

You must load the resources from the proxy service directory into the module database of the App Server. Deploy your resources to the same database directory as the value of the `endpointDirectory` property of the service declaration file (`service.json`).

To load a directory into the modules database, you can use either of the `mlLoadModules` or `mlReloadModules` tasks provided by ml-gradle. You supply the properties required for deployment including the following:

- mlHost - required
- mlAppServicesUsername - required if not admin and mlPassword not set
- mlAppServicesPassword - required if not admin and mlUsername not set
- ml|AppServicesPort - required if not 8000
- mlModulesDatabaseName - required
- mlModulePermissions - required
- mlNoRestServer - required to be true, so that mlDeploy will not create a REST API server by default.
- mlReplaceTokensInModules - typically false

If you did not create the proxy service directory under the `src/main/ml-modules/root` project subdirectory, you must specify the parent directory for the root directory with the `mlModulePaths` property.

You can supply properties using a gradle.properties file or a task.

After you have configured the properties, the command to load the modules would resemble the following example (or the equivalent with mlReloadModules):

```
gradle mlLoadModules
```

For more information, see How modules are loaded.

16.3.7 Generating the Proxy Service Class

A *proxy service class* is a Java interface for calling the endpoint modules for your service on the MarkLogic e-node. You generate the proxy service class from the resources in the proxy service directory.

The proxy service class has the name specified by the `$javaClass` property of the service declaration file (`service.json`). The class has one method for each endpoint declaration with an associated endpoint module in the proxy service directory.

To generate the class, you use the `generateEndpointProxies` Gradle task. You specify the path (relative to the project directory) of the service declaration file (`service.json`) with the `serviceDeclarationFile` property. You can also specify the output directory with the `javaBaseDirectory` property or omit the property to use the default (which is the `src/main/java` subdirectory of the project directory).

Your Gradle build script should apply the `com.marklogic.ml-development-tools` plugin. You can execute the task using any of the following techniques:

- By setting the properties in the `gradle.properties` file and specifying the `generateEndpointProxies` task on the gradle command line
- By specifying the properties with the -P option as well as the `generateEndpointProxies` task on the gradle command line
- By supplying a build script with custom task of the `com.marklogic.client.tools.gradle.EndpointProxiesGenTask` type
- By supplying a build script with the `endpointProxiesConfig` extension configuration and specifying the `generateEndpointProxies` task on the gradle command line

For the custom task approach, the Gradle build script for generating a class with a method for each endpoint in the priceDynamically service might resemble the following example:

```
plugins {
    id 'com.marklogic.ml-development-tools' version '4.1.1'
}
task generateDynamicPricer(type:
com.marklogic.client.tools.gradle.EndpointProxiesGenTask) {
    serviceDeclarationFile =
'src/main/ml-modules/root/inventory/priceDynamically/service.json'
}
```

The command-line to execute the custom task would resemble the following example:

```
gradle generateDynamicPricer
```

You only need to regenerate the proxy service class when the list of endpoints or the name, parameters, or return value for an endpoint changes. You do not need the regenerate the proxy service class after changing the module that implements the endpoint.

16.3.8 Using a Proxy Service Class

In general, you can work with your generated proxy service Java class in the same way as with manually written Java source files.

The generated class has an `on()` static method that is a factory for constructing the class. The `on()` method requires a `DatabaseClient` for the App Server. You construct the database client by using the `DatabaseClientFactory` class of the Java API.

Note: You cannot specify the database explicitly when creating the DatabaseClient but, instead, must use the default database associated with the App Server.

16.3.8.1 Compiling a Proxy Service Class

After generating the proxy service class, you compile it in the usual way. In particular, by generating the proxy service class in the conventional directory for Gradle (which is `src/main/java`) and declaring a dependency on the MarkLogic Java API in the build script, you can use Gradle to compile the generated class without other configuration.

16.3.8.2 Testing a Proxy Service Class

After deploying your proxy service to the MarkLogic modules database, you can test your proxy service Java class in the same manner as any other Java class.

To write functional tests that confirm the endpoint modules work correctly, you can use any general-purpose test framework (for instance, JUnit). The test framework should:

- Call the `on()` static factory method to construct an instance.
- Call the appropriate method to invoke the endpoint module.
- Inspect the returned value to confirm the operation of the endpoint module.

Because the generated proxy service class is available as a Java interface, you can replace the implementation with a mock implementation of the interface for testing a middle-tier consumer.

16.3.8.3 Documenting a Proxy Service Class

The generated class has JavaDoc comments based on the `desc` properties from the service declaration and endpoint declarations. You can generate JavaDoc for the middle-tier consumer of the proxy service class in the usual way.

16.3.8.4 Packaging a Proxy Service

Finally, you can create a jar file with the compiled executable proxy service class in the usual way.

16.4 Publishing Your Data Service for Use in Other Projects

Users of Data Services need to know how to publish a Data Service for use in another project, and developers that require the end-points provided by a Data Service need to have a way to access them in their own projects.

This section shows you how to use the ml-gradle tool to enable publication of your Data Services.

- Modifying the Source project to Enable Publication
- Using the Maven Bundle in Other Projects

16.4.1 Modifying the Source project to Enable Publication

The procedure is to modify the build.gradle file for the source project to publish the Data Services implementation to a Maven repository, as in:

```
plugins {

    ...

    id 'maven-publish'

    ...

}

configurations {

    ...

    myDataServiceBundle

}

task myDataServiceJar(type: Jar) {

   baseName = 'myDataService'

   description = "..."

   from("src/test/ml-modules/root/ds/myDataService") {

       into("myDataService/ml-modules/root/ds/myDataService")

   }

   destinationDir file("build/libs")

}

publishing {
```

```
        publications {

            ...

            MainMyDataService(MavenPublication) {

                artifactId "myDataService"

                artifact myDataServiceJar

            }

            ...

        }

    }
```

16.4.2 Using the Maven Bundle in Other Projects

After the bundle for the Data Service endpoint implementation has been published to a Maven repository, other projects can use the bundle by configuring build.gradle to use the mlBundle task provided by the ml-gradle tool:

```
    plugins {

        ...

        id "com.marklogic.ml-gradle" version "..."

        ...

    }

    dependencies {

        ...

        mlBundle group: '...', name: 'myDataService', version: '...'

        ...

    }
```

For more information, see the Bundles section of our ml-gradle documentation:

Following the standard approach for Gradle and Maven repositories, the client interface can be published and consumed as a Java jar.

17.0 Troubleshooting

This chapter describes how to troubleshoot errors while programming in the Java API, and contains the following sections:

17.1 Error Detection

As you would expect, the Java API client indicates errors by throwing exceptions. It does not return errors or otherwise indicate problems by any other means. The exceptions are located in `com.marklogic.client` and are:

- `FailedRequestException`: Indicates a problem at the REST API level.
- `ForbiddenUserException`: Indicates credentials used to connect to a REST instance are not sufficient for the requested task. Equivalent to a `403` HTTP status code.
- `MarkLogicBindingException`: Indicates a problem binding a value.
- `MarkLogicInternalException`: Indicates a defect in the API. Call MarkLogic Support.
- `MarkLogicIOException`: `RuntimeException` Thrown when a code block internally throws `java.lang.IOException`.
- `MarkLogicServerException`: The MarkLogic REST Server threw an exception.
- `ResourceNotFoundException`: Thrown when the server responds with an HTTP `404` status.
- `UnauthorizedUserException`: Thrown when a user attempts an operation to which they do not have the rights for.

17.2 General Troubleshooting Techniques

The following are some general guidelines for troubleshooting your program.

- To troubleshoot unexpected search results, pass the query option for `debug`, which returns errors in the query options, and the `return-qtext` option, which returns the pre-parsed query text for the search.
- Remember that documents with no read permission are hidden.
- To troubleshoot exceptions, pay close attention to any messages returned from the server.
- Set the MarkLogic Server error log to `debug` and view the server log (`<marklogic-dir>/Logs/ErrorLog.txt`) for more details.
- To monitor the HTTP requests against the REST Server, look at the access logs under the `<marklogic-dir>/Logs` directory for your REST App Server (for example, `1234_AccessLog.txt` for the server running on port 1234).
- Configure managers with a request logger to confirm requests are correct.

- To troubleshoot extensions, first execute the XQuery code in an XQuery environment. Then look at the requests and server log.
- Check the query options builder output to make sure it is what you expect, either with `QueryOptionsHandle.toString()`, which outputs the XML representation of the query options, or by checking the stored options against what is expected. Errors reported by MarkLogic Server refer to the structure of this document.
- When you have a mismatch between query options and existing indexes, you can look at the `/v1/config/indexes?format=html` endpoint on your REST Server.
- If you want a closer look at the requests against the REST Server, use a network sniffer to watch the HTTP trafic against the REST Server. You can also try to execute an equivalent request for the REST API using cURL or some other HTTP client.

18.0 Technical Support

MarkLogic provides technical support according to the terms detailed in your Software License Agreement or End User License Agreement.

We invite you to visit our support website at http://help.marklogic.com to access information on known and fixed issues, knowledge base articles, and more. For licensed customers with an active maintenance contract, see the Support Handbook for instructions on registering support contacts and on working with the MarkLogic Technical Support team.

Complete product documentation, the latest product release downloads, and other useful information is available for all developers at http://developer.marklogic.com. For technical questions, we encourage you to ask your question on Stack Overflow.

19.0 Copyright

MarkLogic Server 10.0 and supporting products.
Last updated: February, 2022

This technology is protected by U.S. Patent No. 7,127,469B2, U.S. Patent No. 7,171,404B2, U.S. Patent No. 7,756,858 B2, and U.S. Patent No 7,962,474 B2, US 8,892,599, and US 8,935,267.

The MarkLogic software is protected by United States and international copyright laws, and incorporates certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers set forth below.

For all copyright notices, including third-party copyright notices, see the Combined Product Notices for your version of MarkLogic.