MarkLogic Server

XCC Developer's Guide

MarkLogic 9
May, 2017

Last Revised: 9.0-2, July, 2017

Copyright © 2019 MarkLogic Corporation. All rights reserved.

Table of Contents

XCC Developer's Guide

1.0 Introduction to XCC 5
1.1 Overview of XCC 5
1.1.1 XCC Client Libraries Communicate With an XDBC Server 5
1.1.2 Client-Server Architecture 6
1.1.3 Automatically Pools Connections 6
1.2 API and Other Documentation 6
1.3 XCC Requirements 7
1.3.1 XCC MarkLogic Server Requirements 7
1.3.2 XML Contentbase Connector for Java (XCC/J) Requirements 7

2.0 Programming in XCC 9
2.1 Configuring an XDBC Server 9
2.2 XCC Sessions 9
2.3 Point-In-Time Queries 10
2.4 Automatically Retries Exceptions 10
2.5 Coding Basics 10
2.6 Evaluating JavaScript Queries 11
2.7 Working With JSON Content 12
2.7.1 Required Libraries 12
2.7.2 Inserting JSON Documents 12
2.7.3 Processing JSON Query Results 14
2.8 Accessing SSL-Enabled XDBC App Servers 15
2.8.1 Creating a Trust Manager 15
2.8.2 Accessing a Keystore 17
2.8.3 Managing Client Side Authentication 18
2.9 Understanding Result Caching 19
2.10 Multi-Statement Transactions 19
2.10.1 Overview 19
2.10.2 Example: Using Multi-Statement Transactions in Java 21
2.10.3 Terminating a Transaction in an Exception Handler 22
2.10.4 Retrying Multi-statement Transactions 22
2.10.5 Using Multi-Statement Transactions With Older MarkLogic Versions 24
2.11 Participating in XA Transactions 24
2.11.1 Overview 24
2.11.2 Predefined Security Roles for XA Participation 25
2.11.3 Enlisting MarkLogic Server in an XA Transaction 26
2.11.4 Heuristically Completing a Stalled Transaction 27
2.11.4.1 Understanding Heuristic Completion 27

2.11.4.2 Heuristically Completing a MarkLogic Server Transaction 28
2.11.4.3 Cleaning Up After Heuristic Completion 30
2.11.5 Reducing Blocking Caused by Slow XA Transactions 30
2.12 Using a Load Balancer or Proxy Server with an XCC Application 30
2.12.1 Enable HTTP 1.1 Compliance 31
2.12.2 Configure the Load Balancer 31

3.0 Downloading and Using the XCC API 33

4.0 Using the Sample Applications 35
4.1 Setting Up Your Environment 35
4.1.1 Setting Up Your MarkLogic Server Environment 35
4.1.2 Setting Up Your Java Environment 35
4.2 Sample Applications 36
4.2.1 ContentFetcher 37
4.2.2 ContentLoader 37
4.2.3 DynamicContentStream 37
4.2.4 HelloSecureWorld 37
4.2.5 HelloWorld 38
4.2.6 ModuleRunner 38
4.2.7 OutputStreamInserter 39
4.2.8 SimpleQueryRunner 39
4.2.9 XA 39

5.0 Technical Support 43

6.0 Copyright 45
6.0 COPYRIGHT 45

1.0 Introduction to XCC

The XML Contentbase Connector (XCC) is an interface to communicate with MarkLogic Server from a Java middleware application layer. This chapter provides background on XCC and includes the following sections:

- Overview of XCC
- API and Other Documentation
- XCC Requirements

1.1 Overview of XCC

The XML Contentbase Connector (XCC) is used to communicate between a Java application layer and MarkLogic Server using the XDBC protocol.

This section provides an overview of XCC and includes the following parts:

- XCC Client Libraries Communicate With an XDBC Server
- Client-Server Architecture
- Automatically Pools Connections

1.1.1 XCC Client Libraries Communicate With an XDBC Server

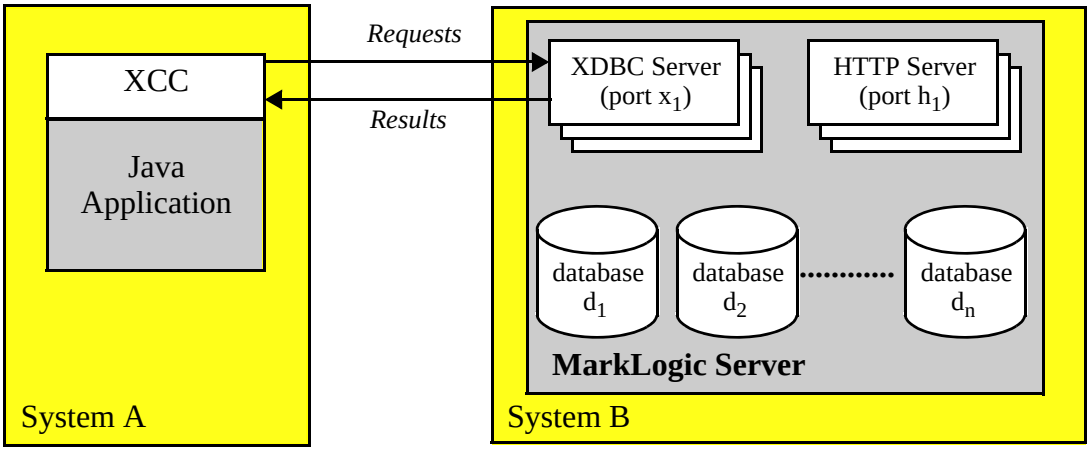
XCC has a set of client libraries that you use to build applications that communicate with MarkLogic Server. XCC requires that an XDBC server is configured in MarkLogic Server.

An XDBC server responds to XDBC and XCC requests. XDBC and XCC use the same wire protocol to communicate with MarkLogic Server. You can write applications either as standalone applications or ones that run in an application server environment. Your XCC-enabled application connects to a specified port on a system that is running MarkLogic Server, and communicates with MarkLogic Server by submitting requests (for example, XQuery statements) and processing the results returned by those programs. These XQuery programs can incorporate calls to XQuery functions stored and accessible by MarkLogic Server, and accessible from any XDBC-enabled application. The XQuery programs can perform the full suite of XQuery functionality, including loading, querying, updating and deleting content.

XQuery requests submitted via XCC return results as specified by the XQuery code. These results can include XML and a variety of other datatypes. It is the XCC application's responsibility to parse, process and interpret these results in a manner appropriate to the variety of datatypes available. There are a number of publicly available libraries for assisting with this task, or you may write your own code. In order to accept connections from XCC-enabled applications, MarkLogic Server must be configured with an XDBC Server listening on the designated port. Each XDBC Server connects by defaut to a specific database within MarkLogic Server, but XCC provides the ability to communicate with any database in the MarkLogic Server cluster to which your application connects (and for which you have the necessary permissions and privileges).

1.1.2 Client-Server Architecture

XCC communicates with MarkLogic Server with a client-server architecture, where the XCC application is the client and MarkLogic Server is the server. The following figure illustrates the high-level architecture:



As shown in the diagram above, the XCC-enabled application can run on the same system as an instance of MarkLogic Server (a host), or it can run on a completely different system, as long as the two systems are networked together.

In the diagram, the XCC application running on System A has opened an XDBC connection to port x_1 on System B. On System B, MarkLogic Server is configured with an XDBC Server listening to port x_1, and that XDBC Server connects to database$_{d1}$. Consequently, the configuration shown in the diagram above allows the XCC application on System A to submit XQuery requests (including query, load, update and delete) for evaluation against database$_{d1}$.

1.1.3 Automatically Pools Connections

XCC automatically does connection pooling, so you do not need to write any connection pooling logic in your application. The XCC `Session` object automatically obtains and releases connections for XCC applications as needed.

1.2 API and Other Documentation

This document provides an introduction to the XCC developer libraries. For detailed API documentation for XCC and for MarkLogic Server, or to learn how to configure XDBC servers in MarkLogic Server, see the appropriate documents:

- Java API documentation (XCC Javadoc, available on developer.marklogic.com)
- MarkLogic Server *Application Developer's Guide*
- MarkLogic Server *Administrator's Guide*

- *MarkLogic XQuery and XSLT Function Reference*

1.3 XCC Requirements

This section lists the requirements for XCC and has the following parts:

- XCC MarkLogic Server Requirements
- XML Contentbase Connector for Java (XCC/J) Requirements

1.3.1 XCC MarkLogic Server Requirements

XCC requires MarkLogic Server 7.0-1 or later.

Note that not all XCC features are usable with all versions of MarkLogic Server. For example, you must have MarkLogic 8 or later to use the Server-Side JavaScript and JSON features with XCC.

1.3.2 XML Contentbase Connector for Java (XCC/J) Requirements

XCC has the following requirements:

- Java 8 or later
- MarkLogic Server 7.0-1 or later (on any platform)

Note that not all XCC features are available with all versions of MarkLogic Server.

You must have MarkLogic 8 or later to use the Server-Side JavaScript and JSON features of XCC, and your classpath must include Jackson JAR files for Jackson version 2.5 or later. For more information about Jackson, see http://github.com/FasterXML/jackson.

To use XCC in an environment that includes a load balancer between MarkLogic and your XCC application, some configuration is required. For details, see “Using a Load Balancer or Proxy Server with an XCC Application” on page 30.

2.0 Programming in XCC

XCC allows you to create multi-tier applications that communicate with MarkLogic Server as the underlying content repository. This chapter describes some of the basic programming concepts used in XCC. It includes the following sections:

- Configuring an XDBC Server
- XCC Sessions
- Point-In-Time Queries
- Automatically Retries Exceptions
- Coding Basics
- Evaluating JavaScript Queries
- Working With JSON Content
- Accessing SSL-Enabled XDBC App Servers
- Understanding Result Caching
- Multi-Statement Transactions
- Participating in XA Transactions
- Using a Load Balancer or Proxy Server with an XCC Application

2.1 Configuring an XDBC Server

Use the Admin Interface to set up an XDBC server, specifying a name, port, a database to access, and other configuration parameters. For detailed instructions how to configure an XDBC Server, see the *Administrator's Guide*. You need an XDBC Server for an XCC program to communicate with MarkLogic Server. Alternately, you can use set up a REST instance to accept XDBC requests by setting the `xdbc-enabled` option to `true` in your REST instance; for details, see Administering REST Client API Instances in the *REST Application Developer's Guide*.

2.2 XCC Sessions

XCC programs use the `Session` interface to set up and control communication with MarkLogic Server. XCC automatically creates and releases connections to MarkLogic Server as needed, and automatically pools the connections so that multiple requests are handled efficiently.

A `Session` handles authentication with MarkLogic Server and holds a dynamic state, but it is a lightweight object. It is OK to create and release Session objects as needed and as makes logical sense for your program. Do not expend effort to pool and reuse them, however, because they are not expensive to create. For example, if your program is doing multiple requests one after another, create a `Session` object at the beginning and close it when the last request is complete.

You set up the connection details with the `ContentSource` object. You can submit the connection details when you invoke the XCC program with a URL that has the following form:

```
xcc://username:password@host:port/database
```

Also, there are discrete arguments to the constructors in the API to set up any or all portions of the connection details.

2.3 Point-In-Time Queries

Point-in-time queries allow you to query older versions of content in a database. In an XCC application, you set up the options for any requests submitted to MarkLogic Server with the `RequestOptions` class. One of the options you can set is the effective point-in-time option. Therefore, to set up a query to run at a different point in time, you just set that option (the `setEffectivePointInTime` method in Java) on the `RequestOptions`. The query will then run at the specified point in time.

There are several things you must set up on MarkLogic Server in order to perform point-in-time queries. For details, see the "Point-In-Time Queries" chapter of the *Application Developer's Guide*.

2.4 Automatically Retries Exceptions

Certain exceptions that MarkLogic Server throws are *retryable*; that is, the exception is thrown because of a condition that is transitory, and applications can try the request again after getting the exception. XCC will automatically retry retryable exceptions in single-statement transactions. You can control the maximum number of retryable exceptions with the `RequestOptions` interface.

Multi-statement transactions cannot automatically be retried by the server. Your application must handle retries explicitly when using multi-statement transactions. For details, see "Retrying Multi-statement Transactions" on page 22.

2.5 Coding Basics

To use XCC, there are several basic things you need to do in your Java code:

- Import the needed libraries.
- Set up the `ContentSource` object to authenticate against MarkLogic Server.
- Create a new `Session` object.
- Add a `Request` to the session object.
- Submit the request and get back a `ResultSequence` object from MarkLogic Server.
- Do something with the results (print them out, for example).
- Close the session.

The following are Java code samples that illustrate these basic design patterns:

```
package com.marklogic.xcc.examples;

import com.marklogic.xcc.ContentSource;
import com.marklogic.xcc.ContentSourceFactory;
import com.marklogic.xcc.Session;
import com.marklogic.xcc.Request;
import com.marklogic.xcc.ResultSequence;

URI uri = new URI("xcc://user:pass@localhost:8000/mycontent");
ContentSource contentSource =
          ContentSourceFactory.newContentSource (uri);

Session session = contentSource.newSession();

Request request = session.newAdhocQuery ("\"Hello World\"");

ResultSequence rs = session.submitRequest (request);

System.out.println (rs.asString());

session.close();
```

Note: `Session` objects are not thread safe. A `Session` object should not be used concurrently by multiple threads.

2.6 Evaluating JavaScript Queries

You can use AdhocQuery and Session.submitRequest to evaluate either XQuery or Server-Side JavaScript queries on MarkLogic Server. By default, XCC assumes the query language is XQuery. Use `RequestOptions.setQueryLanguage` to specify JavaScript instead. For example:

```
// Create a query that is Server-Side JavaScript
AdhocQuery request =
    session.newAdhocQuery("cts.doc('/my/uri')");

// Set the query language to JavaScript
RequestOptions options = new RequestOptions();
options.setQueryLanguage("javascript");

// Submit the query
request.setOptions(options);
ResultSequence rs = session.submitRequest(request);
```

You can the results of your query in the usual way. When a ResultItem in the result sequence is a JsonItem, you can extract the item as Jackson `JsonNode` and use the Jackson library functions to traverse and access the structure.

Note that there is a difference between returning native JavaScript objects and arrays returning JSON nodes from the database:

- A JavaScript object or array corresponds to an atomic result item type. That is, the underlying value type is `JS_OBJECT` or `JS_ARRAY`, and `ItemType.isAtomic` returns true.
- A JSON node, such as the result of calling `cts.doc` or the output from calling a `NodeBuilder` method, has a node value type. That is, a value type such as `OBJECT_NODE`, `ARRAY_NODE`, `BOOLEAN_NODE`, `NUMBER_NODE`, or `NULL_NODE`. Also, `ItemType.isNode` returns true.

In most cases, your code can ignore this distinction because you can use Jackson to manipulate both kinds of results transparently through the `JsonItem` interface. For details, see "Working With JSON Content" on page 12.

2.7 Working With JSON Content

This section covers the following topics related to using XCC to read and write JSON data:

- Required Libraries
- Inserting JSON Documents
- Processing JSON Query Results

2.7.1 Required Libraries

The XCC interfaces include an integration with Jackson for manipulating JSON data in Java. To use XCC methods such as `JsonItem.asJsonNode` or the `ContentFactory.newJsonContent` overload that accepts a `JsonNode`, you must have an installation of Jackson and put the Jackson jar files on your classpath.

Exactly which libraries you need to add to your classpath depends on the Jackson features you use, but you will probably need at least the Jackson core libraries, available from http://github.com/FasterXML/jackson.

For example, you might need to add the following libraries to your classpath:

- `jackson-core-`*`version`*`.jar`
- `jackson-annotations-`*`version`*`.jar`
- `jackson-databind-`*`version`*`.jar`

For information on version restrictions, see "XML Contentbase Connector for Java (XCC/J) Requirements" on page 7.

2.7.2 Inserting JSON Documents

You can JSON data into the database the same way you insert other data. If the document URI extension is mapped to the JSON document format in the MarkLogic Server MIME type mappings, then a JSON document is automatically created.

For example, the following code snippet reads JSON data from a file and inserts it into the database as a JSON document.

```
ContentSource cs = ContentSourceFactory.newContentSource(SERVER_URI);
Session session = cs.newSession();

File inputFile = new File("data.json");
String uri = "/xcc/fromFile.json";
Content content = ContentFactory.newContent(uri, inputFile, null);
session.insertContent(content);
```

If there is no URI extension or you use an extension that is not mapped to JSON, you can explicitly specify JSON using ContentCreateOptions.setFormat. For example:

```
ContentCreateOptions options = new ContentCreateOptions();
options.setFormat(DocumentFormat.JSON);
Content content = ContentFactory.newContent(uri, inputFile, options);
```

The following code snippet inserts a JSON document into the database using an in-memory String representation of the contents:

```
ContentSource cs = ContentSourceFactory.newContentSource(SERVER_URI);
Session session = cs.newSession();

String uri = "/xcc/fromString.json";
String data = "{\"num\":1, \"arr\":[0, 1], \"str\":\"value\"}";

ContentCreateOptions options = ContentCreateOptions.newJsonInstance();
Content content = ContentFactory.newContent(uri, data, options);

session.insertContent(content);
```

You can also use Jackson to build up JSON content, and then pass a Jackson `JsonNode` in to `ContentFactory.newJsonContent`. To learn more about Jackson, see http://github.com/FasterXML/jackson-docs.

2.7.3 Processing JSON Query Results

If you run an ad hoc query that returns JSON (or a JavaScript object or array), you can use Jackson to traverse and manipulate the data in your Java application.

For example, the following code snippet evaluates an ad hoc Server-Side JavaScript query that retrieves a JSON document from the database, and then accesses the value in the document's "num" property as an integer:

```
Session session = cs.newSession();
AdhocQuery request =
    session.newAdhocQuery("cts.doc('/xcc/fromString.json')");
RequestOptions options = new RequestOptions();
options.setQueryLanguage("javascript");
request.setOptions(options);

ResultSequence rs = session.submitRequest(request);
while (rs.hasNext()) {
    XdmItem item = rs.next().getItem();
    if (item instanceof JsonItem) {
        JsonItem jsonItem = (JsonItem) item;
        JsonNode node = jsonItem.asJsonNode();
        // process the value...
    }
}
```

You can use `JsonItem.asJsonNode` to convert a JSON result item into a Jackson `JsonNode` (`com.fasterxml.jackson.databind.JsonNode`). For example:

```
JsonItem jsonItem = (JsonItem) item;
JsonNode node = jsonItem.asJsonNode();
```

You can also use the Jackson interfaces to manipulate native JavaScript objects and arrays returned by ad hoc Server-Side JavaScript queries. That is, the above conversion to a JsonNode works whether the item is a JSON node or an atomic `JS_OBJECT` result.

Then you can use any of the Jackson interfaces to manipulate the contents. For example, the following code snippet accesses the value of the "num" JSON property as an integer:

```
node.get("num").asInt()
```

To learn more about Jackson, see http://github.com/FasterXML/jackson-docs.

2.8 Accessing SSL-Enabled XDBC App Servers

There are three basic approaches for an XCC application to create a secure connection to an SSL-enabled XDBC App Server, which include:

- Creating a Trust Manager
- Accessing a Keystore
- Managing Client Side Authentication

These approaches are described in this section and demonstrated in the `HelloSecureWorld.java` example distributed with your MarkLogic XCC software distribution.

2.8.1 Creating a Trust Manager

This section describes how to use a simple Trust Manager for X.509-based authentication. The Trust Manager shown here does not validate certificate chains and is therefore unsafe and should not be used for production code. See your Java documentation for details on how to create a more robust Trust Manager for your specific application or how to obtain a Certificate Authority from a keystore.

To enable SSL access using a trust manager, import the following classes in addition to those described in “Coding Basics” on page 10:

```
import javax.net.ssl.SSLContext;
import com.marklogic.xcc.SecurityOptions;
import javax.net.ssl.TrustManager;
import javax.net.ssl.X509TrustManager;

import java.security.KeyManagementException;
import java.security.cert.X509Certificate;
import java.security.cert.CertificateException;
```

Create a trust manager and pass it to the `SSLContext.init()` method:

```
protected SecurityOptions newTrustOptions()
  throws Exception
{
   TrustManager[] trust = new TrustManager[] { new X509TrustManager() {
    public void checkClientTrusted(
       X509Certificate[] x509Certificates,
       String s)
          throws CertificateException {
     // nothing to do
    }

    public void checkServerTrusted(
       X509Certificate[] x509Certificates,
       String s)
          throws CertificateException {
     // nothing to do
    }

    public X509Certificate[] getAcceptedIssuers() {
     return null;
    }
  }
};

  SSLContext sslContext = SSLContext.getInstance("SSLv3");
  sslContext.init(null, trust, null);
  return new SecurityOptions(sslContext);
}
```

Call `ContentSourceFactory.newContentSource()` with a host name, port, user name, password, and SSL security options defined by `newTrustOptions()`:

```
ContentSource cs =
    ContentSourceFactory.newContentSource (host,
                                           port,
                                           username,
                                           password,
                                           null,
                                           newTrustOptions());
```

Note: If you are passing a URI to `ContentSourceFactory.newContentSource()`, specify a connection scheme of `xccs`, rather than `xcc`., as shown in "Accessing a Keystore" on page 17.

2.8.2 Accessing a Keystore

You can use the Java `keytool` utility to import a MarkLogic certificate into a keystore. See the Java JSSE documentation for details on the use of the `keytool` and your keystore options.

You can explicitly specify a keystore, as shown in this example, or you can specify a null keystore. Specifying a null keystore causes the `TrustManagerFactory` to locate your default keystore, as described in the *Java Secure Socket Extension (JSSE) Reference Guide*.

To enable SSL by accessing certificates in a keystore, import the following classes in addition to those described in "Coding Basics" on page 10:

```
import com.marklogic.xcc.SecurityOptions;
import com.marklogic.xcc.ContentSource;
import com.marklogic.xcc.ContentSourceFactory;

import java.io.FileInputStream;
import java.net.URI;
import javax.net.ssl.KeyManager;
import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.TrustManager;
import javax.net.ssl.TrustManagerFactory;
import javax.net.ssl.X509TrustManager;
import javax.net.ssl.SSLContext;

import java.security.KeyStore;
import java.security.cert.X509Certificate;
```

Get the signed certificate from a keystore and pass it to the `SSLContext.init()` method:

```
protected SecurityOptions newTrustOptions()
  throws Exception
{

// Load key store with trusted signing authorities.
  KeyStore trustedKeyStore = KeyStore.getInstance("JKS");
  trustedKeyStore.load(
    new FileInputStream("C:/users/myname/.keystore"),
    null);

// Build trust manager to validate server certificates using the
   specified key store.
  TrustManagerFactory trustManagerFactory =
    TrustManagerFactory.getInstance("SunX509");
  trustManagerFactory.init(trustedKeyStore);
  TrustManager[] trust = trustManagerFactory.getTrustManagers();

  SSLContext sslContext = SSLContext.getInstance("SSLv3");
  sslContext.init(null, trust, null);
  return new SecurityOptions(sslContext);
}
```

Call `ContentSourceFactory.newContentSource()` with a URI:

```
ContentSource cs =
    ContentSourceFactory.newContentSource (uri,
                                           newTrustOptions());
```

The URI is passed from the command line in the form of:

```
xccs://username:password@hostname:port
```

2.8.3 Managing Client Side Authentication

You can define a `KeyManager`, if your client application is required to send authentication credentials to the server. The following example adds client authentication to the `newTrustOptions` method shown in “Accessing a Keystore” on page 17:

```
protected SecurityOptions newTrustOptions()
  throws Exception
{

// Load key store with trusted signing authorities.
  KeyStore trustedKeyStore = KeyStore.getInstance("JKS");
  trustedKeyStore.load(
    new FileInputStream("C:/users/myname/.keystore"),
    null);

// Build trust manager to validate server certificates using the
   specified key store.
  TrustManagerFactory trustManagerFactory =
    TrustManagerFactory.getInstance("SunX509");
  trustManagerFactory.init(trustedKeyStore);
  TrustManager[] trust = trustManagerFactory.getTrustManagers();

// Load key store with client certificates.
  KeyStore clientKeyStore = KeyStore.getInstance("JKS");
  clientKeyStore.load(
    new FileInputStream("C:/users/myname/.keystore"),
    null);

// Get key manager to provide client credentials.
  KeyManagerFactory keyManagerFactory =
    KeyManagerFactory.getInstance("SunX509");
  keyManagerFactory.init(clientKeyStore, "passphrase");
  KeyManager[] key = keyManagerFactory.getKeyManagers();

// Initialize the SSL context with key and trust managers.
  SSLContext sslContext = SSLContext.getInstance("SSLv3");
  sslContext.init(key, trust, null);
  return new SecurityOptions(sslContext);
}
```

2.9 Understanding Result Caching

When you submit a request to MarkLogic Server, the results are returned to your application in a `ResultSequence`. By default the `XdmItem` objects in the sequence are cached. That is, all the result items are read and buffered in memory. Cached results do not tie up any connection resources, and are usually preferred.

A non-cached, or streaming, `ResultSequence` may only be accessed sequentially and hold the connection to MarkLogic Server open. Individual results may only be read once and on demand, so the result set consumes less memory, at the cost of less efficient access.

If you are retrieving large results, such as a large binary document, you may disable result caching to conserve memory. You may disable result caching per request by creating a `RequestOption` object with the setting disabled, and associating it with a request, either directly with `Request.setOptions` or passing it as a parameter to a Request creation method such as `Session.newAdhocQuery`. You may disable result caching per session by setting the default request options for the session using `Session.setDefaultRequestOptions`.

For details, see the `ResultSequence` XCC javadoc.

2.10 Multi-Statement Transactions

By default, all transactions run as single-statement, auto-commit transactions. MarkLogic Server also supports multi-statement, explicitly commited transactions in XQuery, Server-Side JavaScript, and XCC. For more details on transaction concepts, see Understanding Transactions in MarkLogic Server in the *Application Developer's Guide*. This section covers only related behaviors unique to XCC:

- Overview
- Example: Using Multi-Statement Transactions in Java
- Terminating a Transaction in an Exception Handler
- Retrying Multi-statement Transactions
- Using Multi-Statement Transactions With Older MarkLogic Versions

2.10.1 Overview

Use the following procedure to use multi-statement, explicitly committed transactions with XCC:

1. Create a `Session` object in the usual way.

2. Call `Session.setAutoCommit` with a value of `false`. The next transaction created in the session will run as a multi-statement, explicit commit transaction.

3. Optionally, call `Session.setUpdate` to specify an explicit transaction type. By default, MarkLogic determines the transaction type through static analysis of the first statement in

a request, but you can explicitly set the transaction type to update or query using `Session.setUpdate`.

4. Call `Session.submitRequest` as usual to operate on your data. All requests run in the same transaction until the transaction is commited or rolled back.

5. Call `Session.commit` or `Session.rollback` to commit or rollback the transaction. If the session ends or times out without explicitly commit or rolling back, the transaction is rolled back.

6. To restore a session to the default, single-statement transaction model, call `Session.setAutoCommit` with a value of `true` and `Session.setUpdate` with a value of `AUTO`.

Note that the transaction configuration defined by `setAutoCommit` and `setUpdate` remain in effect for all transactions created by a session until explicitly changed. If you override the transaction configuration in an ad hoc query, the override applies only to the current transaction.

Multi-statement query transactions allow all the statements in a transaction to share the same point-in-time view of the database, as discussed in “Point-In-Time Queries” on page 10.

In a multi-statement update transaction, updates performed by one statement (or request) are visible to subsequent statements in the same transaction, without being visible to other transactions.

A multi-statement transaction remains open until it is committed or rolled back. Use `Session.commit`. to commit a multi-statement transaction and make the changes visible in the database. Use `Session.rollback` to roll back a multi-statement transaction, discarding any updates. Multi-statement transactions are implicitly rolled back when the containing session ends or the transaction times out. Failure to explicitly commit or rollback a multi-statement update transaction can tie up resources, hold locks unnecessarily, and increase the chances of deadlock.

Note: You may receive a `java.lang.IllegalStateException` if you call `Session.commit` from an exception handler when there are no pending updates in the current transaction. Committing from a handler is not recommended.

For a detailed discussion of multi-statement transactions, see the “Understanding Transactions” chapter of the *Application Developer’s Guide.*

Multi-statement transactions impose special re-try semantics on XCC applications. For details, see “Retrying Multi-statement Transactions” on page 22.

2.10.2 Example: Using Multi-Statement Transactions in Java

The following example demonstrates using multi-statement transactions in Java. The first multi-statement transaction in the session inserts two documents into the database, calling `Session.commit` to complete the transaction and commit the updates. The second transaction demonstrates the use of `Session.rollback`. The third transaction demonstrates implicitly rolling back updates by closing the session.

```
import java.net.URI;

import com.marklogic.xcc.ContentSource;
import com.marklogic.xcc.ContentSourceFactory;
import com.marklogic.xcc.Session;

public class SimpleMST {
  public static void main(String[] args) throws Exception {
    if (args.length != 1) {
      System.err.println("usage: xcc://user:password@host:port/
contentbase");
      return;
    }

    // Obtain a ContentSource object for the server at the URI.
    URI uri = new URI(args[0]);
    ContentSource contentSource =
        ContentSourceFactory.newContentSource(uri);

    // Create a Session and set the transaction mode to trigger
    // multi-statement transaction use.
    Session updateSession = contentSource.newSession();
    updateSession.setAutoCommit(false);
    updateSession.setUpdate(Session.Update.TRUE);

    // The request starts a new, multi-statement transaction.
    updateSession.submitRequest(updateSession.newAdhocQuery(
    "xdmp:document-insert('/docs/mst1.xml', <data/>)"));

    // This request executes in the same transaction as the previous
    // request and sees the results of the previous update.
    updateSession.submitRequest(updateSession.newAdhocQuery(
    "xdmp:document-insert('/docs/mst2.xml', fn:doc('/docs/
mst1.xml'));"));

    // After commit, updates are visible to other transactions.
    // Commit ends the transaction after current stmt completes.
    updateSession.commit();                  // txn ends, updates kept

    // Rollback discards changes and ends the transaction.
    updateSession.submitRequest(updateSession.newAdhocQuery(
    "xdmp:document-delete('/docs/mst1.xml')"));
    updateSession.rollback();                // txn ends, updates lost

    // Closing session without calling commit causes a rollback.
```

```
            updateSession.submitRequest(updateSession.newAdhocQuery(
            "xdmp:document-delete('/docs/mst1.xml')"));
            updateSession.close();                    // txn ends, updates lost
          }
        }
```

2.10.3 Terminating a Transaction in an Exception Handler

Calling `Session.commit` from an exception handler that wraps a request participating in a multi-statement transaction may raise `java.lang.IllegalStateException`. You may always safely call `Session.rollback` from such a handler.

Usually, an exception raised during multi-statement transaction processing leaves the `Session` open, allowing you to continue working in the transaction after handling the exception. However, in order to preserve consistency, exceptions occurring under the following circumstances always roll back the transaction:

- After an XQuery statement has finished but before the XCC request is completed
- In the middle of an explicit commit or rollback

If such a rollback occurs, the current transaction is terminated before control reaches your exception handler. Calling `Session.commit` when there is no active transaction raises a `java.lang.IllegalStateException`. Calling `Session.rollback` when there is no active transaction does not raise an exception, so rollback from a handler is always safe.

Therefore, it is usually only safe to call `Session.commit` from an exception handler for specific errors you expect to receive and for which you can predict the state of the transaction.

2.10.4 Retrying Multi-statement Transactions

MarkLogic Server sometimes detects the need to retry a transaction. For example, if the server detects a deadlock, it may cancel one of the deadlocked transactions, allowing the other to complete; the cancelled transaction should be re-tried.

With single-statement, auto-commit transactions, the server can usually retry automatically because it has the entire transaction available at the point of detection. However, the statements in a multi-statement transactions from XCC clients may be interleaved with arbitrary application-specific code of which the server has no knowledge.

In such cases, instead of automatically retrying, the server throws a `RetryableXQueryException`. The calling application is then responsible for re-trying all the requests in the transaction up to that point. This exception is more likely to occur when using multi-statement transactions.

The following example demonstrates logic for re-trying a multi-statement transaction. The multi-statement transaction code is wrapped in a retry loop with an exception handler that waits between retry attempts. The number of retries and the time between attempts is up to the application.

```
import java.net.URI;

import com.marklogic.xcc.ContentSource;
import com.marklogic.xcc.ContentSourceFactory;
import com.marklogic.xcc.Session;
import com.marklogic.xcc.exceptions.RetryableXQueryException;

public class TransactionRetry {
  public static final int MAX_RETRY_ATTEMPTS = 5;
  public static final int RETRY_WAIT_TIME = 1;

  public static void main(String[] args) throws Exception {
    if (args.length != 1) {
      System.err.println("usage: xcc://user:password@host:port/
contentbase");
      return;
    }

    // Obtain a ContentSource object for the server at the URI.
    URI uri = new URI(args[0]);
    ContentSource contentSource =
        ContentSourceFactory.newContentSource(uri);

    // Create a Session and set the transaction mode to trigger
    // multi-statement transaction use.
    Session session = contentSource.newSession();
    Session.setAutoCommit(false);
    Session.setUpdate(Session.Update.TRUE);

    // Re-try logic for a multi-statement transaction
    for (int i = 0; i < MAX_RETRY_ATTEMPTS; i++) {
      try {
        session.submitRequest(session.newAdhocQuery(
          "xdmp:document-insert('/docs/mst1.xml', <data/>)"));
        session.submitRequest(session.newAdhocQuery(
          "xdmp:document-insert('/docs/mst2.xml', fn:doc('/docs/
mst1.xml'));"));
        session.commit();
        break;
      } catch (RetryableXQueryException e) {
        Thread.sleep(RETRY_WAIT_TIME);
      }
    }
    session.close();
  }
}
```

2.10.5 Using Multi-Statement Transactions With Older MarkLogic Versions

If you use multi-statement transactions or set the transaction time limit when using XCC version 8.0-2 or later with versions of MarkLogic Server older than 8.0-2, you should set the system property `xcc.txn.compatible` to true. If you do not set this, then you will get an exception when trying to set the transaction mode or transaction time limit.

You can set the property on the java command line with an argument of the following form:

```
java -Dxcc.txn.compatible=true
```

You can also set the property programmatically by calling `System.setProperty`.

You do not need to set the property if your XCC application does not use multi-statement transactions or if your application communicates with MarkLogic Server version 8.0-2 or later.

2.11 Participating in XA Transactions

MarkLogic Server can participate in distributed transactions by acting as a Resource Manager in an XA/JTA transaction. This section covers the following related topics:

- Overview
- Enlisting MarkLogic Server in an XA Transaction
- Heuristically Completing a Stalled Transaction
- Reducing Blocking Caused by Slow XA Transactions

Note: To use XA, a license that includes XA is required.

2.11.1 Overview

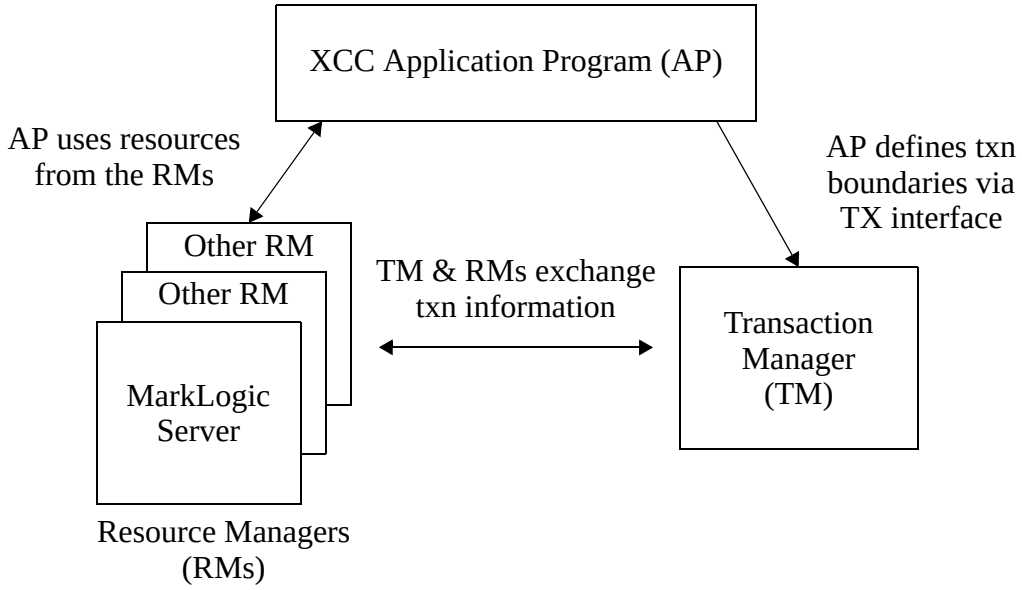
XA is a standard for distributed transaction processing defined by The Open Group. For details about XA, see:

https://www2.opengroup.org/ogsys/jsp/publications/PublicationDetails.jsp?catalogno=c193

JTA is the Java Transaction API, a Java standard which supports distributed transactions across XA resources. For details about JTA, see http://java.sun.com/products/jta/.

The XCC API includes support for registering MarkLogic Server as a resource with an XA Transaction Manager. For details, see “Enlisting MarkLogic Server in an XA Transaction” on page 26.

The basic XA architecture as it applies to MarkLogic Server is shown in the following diagram:



2.11.2 Predefined Security Roles for XA Participation

The following security roles are predefined for participating in and administering XA transactions:

- The `xa` user role allows creation and management of one's own XA transaction branches in MarkLogic Server.
- The `xa-admin` role allows creation and management of any user's XA transaction branches in MarkLogic Server.

The `xa` role is required to participate in XA transactions. The `xa-admin` role is intended primarily for Administrators who need to complete or forget XA transactions; see "Heuristically Completing a Stalled Transaction" on page 27.

2.11.3 Enlisting MarkLogic Server in an XA Transaction

To use MarkLogic Server in an XA transaction, use the `Session.getXAResource` method to register your XCC session as a `javax.transaction.xa.XAResource` with the XA Transaction Manager.

The following code snippet shows how to enlist an XCC session in a global XA transaction. Once you enlist the `Session`, any work performed in the session is part of the global transaction. For complete code, see the sample code in `com.marklogic.xcc.examples.XA`.

```
javax.transaction.TransactionManager tm = ...;
Session session = ...;

try {
    // Begin a distributed transaction
    tm.begin();

    // Add the MarkLogic Session to the distributed transaction
    javax.transaction.xa.XAResource xaRes = session.getXAResource();
    tm.getTransaction().enlistResource(xaRes);

    // Perform MarkLogic Server updates under the global transaction
    session.submitRequest(session.newAdhodquery(
        "xdmp:document-insert('a', <a/>)"));

    // Update other databases here

    //Commit all updates together
    tm.commit();
} catch (Exception e) {
    e.printStackTrace();
    if (tm. getTransaction != null) tm.rollback();
} finally {
    session.close();
}
```

When MarkLogic Server acts as an XA transaction Resource Manager, requests submitted to the server are always part of a multi-statement update transaction, with the following important differences:

- The `Session.setAutoCommit` and `Session.setTransactionMode` settings are ignored. The transaction is always a multi-statement update transaction, even if only a single request is submitted to MarkLogic Server during the global transaction.
- The application should not call `Session.commit` or `xdmp:commit`. The transaction is committed (or rolled back) as part of the global XA transaction. To commit the global transaction, use the Transaction Manager with which the MarkLogic Server `XAResource` is registered.

- The application may call `Session.rollback` or `xdmp:rollback`. Doing so eventually causes rollback of the global XA transaction. Rolling back via the Transaction Manager is usually preferable.

To learn more about multi-statement transactions, see "Multi-Statement Transactions" on page 19.

2.11.4 Heuristically Completing a Stalled Transaction

Under extreme circumstances, a MarkLogic Server administrator may need to heuristically complete (intervene to manually commit or rollback) the MarkLogic Server portion of a prepared XA transaction. This section covers the following topics related to heuristic completion:

- Understanding Heuristic Completion
- Heuristically Completing a MarkLogic Server Transaction
- Cleaning Up After Heuristic Completion

2.11.4.1 Understanding Heuristic Completion

This section provides a brief overview of the concept of heuristically completing XA transactions. For instructions specific to MarkLogic Server, see "Heuristically Completing a MarkLogic Server Transaction" on page 28.

The unit of work managed by a Resource Manager in an XA transaction is a *branch*. Manually intervening to force completion of a prepared XA transaction branch is *making a heuristic decision*, or *heuristically completing* the branch. The branch may be heuristically completed by either committing or rolling back the local transaction.

XA uses Two Phase Commit to commit or rollback global transactions. Normal transaction completion follows the flow:

- The Transaction Manager instructs all participants to prepare to commit.
- Each participant reports responds with whether or not it is ready to commit.
- If all participants report prepared to commit, the Transaction Manager instructs all participants to commit.
- If one or more participants is not prepared to commit, the Transaction Manager instructs all participants to roll back.

If the Transaction Manager goes down due to a failure such as loss of network connectivity or a system crash, the Transaction Manager does not remember the global transaction when it comes back up. In this case, the local transaction times out normally, or may be cancelled with a normal rollback, using `xdmp:transaction-rollback`.

If the Transaction Manager goes down after the transaction is prepared, the Transaction Manager normally recovers and resumes the flow described above. However, it may not always be possible to wait for normal recovery.

For example, if connectivity to the Transaction Manager is lost for a long time, locks may be held unacceptably long on documents in MarkLogic Server. Under such circumstances, the administrator may heuristically complete a branch of the global transaction to release resources.

> **Note:** Heuristic completion bypasses the Transaction Manager and the Two Phase Commit process, so it can lead to data integrity problems. Use heuristic completion only as a last resort.

The XA protocol requires a Resource Manager to remember the outcome of a heuristic decision, allowing the Transaction Manager to determine the status of the branch when it resynchronizes the global transaction.This remembered state is automatically cleaned up if the global transaction eventually completes with the same outcome as the heuristic decision.

Manual intervention may be required after a heuristic decision. If the Transaction Manager recovers and makes a different commit/rollback decision for the global transaction than the local heuristic decision for the branch, then data integrity is lost and must be restored manually.

For example, if the administrator heuristically completes a branch by committing it, but the global transaction later rolls back, then the heuristically completed branch requires manual intervention to roll back the previously committed local transaction and make the resource consistent with the other global transaction participants.

Heuristic completion is not needed in cases where a global transaction stalls prior to being prepared. In this case, the global transaction is lost and the local branches time out or otherwise fall back on normal failure mechanisms.

2.11.4.2 Heuristically Completing a MarkLogic Server Transaction

Use the `xdmp:xa-complete` built-in function to heuristically complete the MarkLogic Server branch of a prepared global XA transaction. When using `xdmp:xa-complete`, you must indicate:

- whether or not to commit the local transaction
- whether or not MarkLogic Server should remember the heuristic decision outcome (commit or rollback)

Usually, you should rollback the local transaction and remember the heuristic decision outcome.

Forgetting the heuristic decision leads to an error and possibly loss of data integrity when the Transaction Manager subsequently attempts to resynchornize the global transaction. If the outcome is remembered, then the Transaction Manager can learn the status of the branch and properly resume the global transaction.

The following examples demonstrate several forms of heuristic completion. The 3rd parameter indicates whether or not to commit. The 4th parameter indicates whether or not to remember the outcome:

```
(: commit and remember the transaction outcome:)
xdmp:xa-complete($forest-id, $txn-id, fn:true(), fn:true())

(: roll back and remember the transaction outcome:)
xdmp:xa-complete($forest-id, $txn-id, fn:false(), fn:true())

(: commit and forget the transaction outcome :)
xdmp:xa-complete($forest-id, $txn-id, fn:true(), fn:false())
```

The forest id parameter of `xdmp:xa-complete` identifies the coordinating forest. Once an XA transaction is prepared, the coordinating forest remembers the state of the MarkLogic Server branch until the global transaction completes. Use the Admin UI or `xdmp:forest-status` to determine the transaction id and coordinating forest id.

For example, the following query retrieves a list of all transaction participants in transactions that are prepared but not yet comitted. The coordinating forest id for each participant is included in the results. For details, see `xdmp:forest-status` in *XQuery and XSLT Reference Guide*.

```
xquery version "1.0-ml";
for $f in xdmp:database-forests(xdmp:database())
return xdmp:forest-status($f)//*:transaction-participants
```

Additional cleanup may be necessary if the Transaction Manager resumes and the global transaction has an outcome that does not match the heuristic decision. For details, see “Cleaning Up After Heuristic Completion” on page 30.

You may also use the Admin Interface to heuristically rollback XA transaction or to forget a heuristic decision. See Rolling Back a Prepared XA Transaction Branch in the *Administrator’s Guide*.

2.11.4.3 Cleaning Up After Heuristic Completion

If a heuristic decision is made for a MarkLogic Server branch of an XA transaction and the Transaction Manager subsequently completes the transaction, there are two possible outcomes:

- The global transaction completes with an outcome that matches the heuristic decision. No further action is required.
- The global transaction completes with an outcome that does not match the heuristic decision. Take the clean up steps listed below.

If the global transaction outcome does not agree with the heuristic decision, you may need to do the following to clean up the heuristic decision:

- Take whatever manual steps are necessary to restore data integrity for the heuristically completed branch.
- If the heuristic decision was remembered by setting the `remember` parameter of `xdmp:xa-complete` to true, call `xdmp:xa-forget` to clean up the remaining transaction state information.

2.11.5 Reducing Blocking Caused by Slow XA Transactions

Since XA transactions may involve multiple participants and non-MarkLogic Server resources, they may take longer than usual. A slow XA transaction may cause other queries on the same App Server to block for an unacceptably long time.

You may set the "multi-version concurrency control" App Server configuration parameter to `nonblocking` to minimize blocking, at the cost of less timely results. For details, see Reducing Blocking with Multi-Version Concurrency Control in the *Application Developer's Guide*.

2.12 Using a Load Balancer or Proxy Server with an XCC Application

This section contains important information for environments in which a Layer 3 Load Balancer or a proxy server such as the Amazon Elastic Load Balancer (ELB) sits between your XCC application and MarkLogic Server cluster.

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 queries evaluated in the context 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.*

To enable your load balancer to preserve session affinity, you must do the following:

- Enable HTTP 1.1 Compliance in XCC
- Configure the Load Balancer to use the XCC `SessionID` cookie to associate a client with the MarkLogic host servicing its XCC session.

2.12.1 Enable HTTP 1.1 Compliance

Enabling HTTP compliant mode guarantees the traffic between your XDBC App Server and your XCC client is compliant with the HTTP 1.1 protocol. This enables properly configured load balancers to detect the `SessionID` cookie generated by MarkLogic Server and use it to enforce session affinity.

To enable this mode for a Java application, set the `xcc.httpcompliant` system property to true on the Java command line. For example:

```
java -Dxcc.httpcompliant=true ...
```

Note: Setting `xcc.httpcompliant` to true is incompatible with enabling content entity resolution using `ContentCreateOptions.setResolveEntities`.

If `xcc.httpcompliant` is not set explicitly, then `xcc.httpcompliant` is false.

You must also configure your load balancer to use the value in the `SessionID` cookie for session affinity. Some routers or load balancers may need to have `xcc.httpcompliant` enabled to allow any traffic through, regardless of session affinity issues.

2.12.2 Configure the Load Balancer

In addition to setting `xcc.httpcompliant` to true, you must configure your load balancer to use the `SessionID` cookie generated by MarkLogic Server for session affinity. You might also need to enable session affinity or sticky sessions in your load balancer. The exact configuration steps depend on the load balancer; see your load balancer documentation for details.

Note: Your load balancer must be HTTP 1.1 compliant and support cookie-based session affinity to use this feature of XCC.

A SessionID cookie looks similar to the following:

```
SessionID=25b877c32807aa9f
```

3.0 Downloading and Using the XCC API

The XCC API is available by downloading the XCC package from developer.marklogic.com.

For a description of the sample applications included with XCC, see “Using the Sample Applications” on page 35.

The XCC distribution has the following directory structure:

Document or Directory	Description
`docs/`	Includes the Javadoc for XCC in both expanded HTML and compressed zip format.
`lib/`	Contains the `marklogic-xcc-`*`version`*`.jar` file, which is the XCC libraries, and the `marklogic-xcc-examples-`*`version`*`.jar` file, which has the compiled versions of the sample applications. Note that the name of the XCC jar file has the version number encoded.
`src/`	Includes the source code for the sample applications.
`Readme.txt`	Includes the version number and any last-minute updates not included in the documentation.

4.0 Using the Sample Applications

The XCC packages contain a number of sample applications. Each sample application is provided along with its source code, giving you a starting point for creating your own applications. This chapter describes the sample applications and contains the following sections:

- Setting Up Your Environment
- Sample Applications

4.1 Setting Up Your Environment

Before running the sample applications, be sure to set up the necessary environment to run the application. This section has the following parts:

- Setting Up Your MarkLogic Server Environment
- Setting Up Your Java Environment

4.1.1 Setting Up Your MarkLogic Server Environment

Before you run the sample applications, complete the following steps:

1. Install MarkLogic Server, or have a MarkLogic Server installation to which you can connect. For details on installing MarkLogic Server, see the *Installation Guide*.

2. Create and configure an XDBC Server using the Admin Interface. See the *Administrator's Guide* for details on how to create and configure an XDBC Server.

3. Configure a user for the XDBC Server you created. For example, add a user to the security database with the username as `user` and the password as `pass`. See the *Administrator's Guide* for details on adding a user to the security database.

4.1.2 Setting Up Your Java Environment

If you are using XCC/J, you must have Java installed on your client machine. Additionally, you will need the following set up to run the sample applications:

- Set your `JAVA_HOME` environment variable, if it is not already set. For example, if you are running a Windows machine, set `JAVA_HOME` in a command window as in the following example:

  ```
  set JAVA_HOME=c:\Sun\SDK\jdk
  ```

  Substitute the directory in which Java is installed in your environment.

- Set your `CLASSPATH` environment variable correctly, or use the `-classpath` option to pass the appropriate classpath on the command line. Make sure to use the correct name for the

`marklogic-xcc-N.jar` file in your `CLASSPATH`, where *N* corresponds to the service release version number.

4.2 Sample Applications

The source code and API documentation for the sample applications are includes in the XCC packages.

The Java distribution of XCC includes `marklogic-xcc-jar-N.x.jar` and `marklogic-xcc-examples-N.jar` files. The commands to launch the sample programs in this section assume you have renamed these jar files to `xcc.jar` and `xccexamples.jar`, respectively. The commands to launch the sample programs also assume the XCC Java distrubtion is installed in `XCC_HOME`.

The sample applications are as follows:

Sample	Description
ContentFetcher	This class fetches documents from the conentbase and writes their serialized contents to a provided OutputStream.
ContentLoader	This program accepts a server URI (in the format expected by `ContentSourceFactory.newContentSource(java.net.URI)`) and one or more file pathnames of documents to load.
DynamicContentStream	This program demonstrates inserting unbuffered, chunkable dynamic content into the database without spawning a new thread.
HelloSecureWorld	This simple program prints out the string “Hello World”, using SSL/TLS to connect to MarkLogic Server.
HelloWorld	This simple program prints out the string `"Hello World"`.
ModuleRunner	This simple program invokes a named XQuery module on the server and return the result.
OutputStreamInserter	This program demonstrates inserting unbuffered dynamic content into the database by spawning a new thread to write the data.
SimpleQueryRunner	This is a very simple class that will submit an XQuery string to the server and return the result.
XA	This program demonstrates using MarkLogic Server in a distributed XA transaction, using JBoss as the Transaction Manager.

4.2.1 ContentFetcher

This program fetches a document from MarkLogic Server and serializes its contents. You can serialize the contents to the standard output (display it on the screen) or to a file using the `-o` option. The following is a sample command to run the `ModuleRunner` class:

```
java -classpath "XCC_HOME/lib/xcc.jar:XCC_HOME/lib/xccexamples.jar"
     com.marklogic.xcc.examples.ContentFetcher
     xcc://username:password@localhost:8021
     /mydocs/hello.xml -o myHelloFile.xml
```

This sends the contents of the document at `/mydocs/hello.xml` to the file `myHelloFile.xml` (in the same directory in which the command is run). It connects to the default database of the XDBC Server listening on port 8021 of the local machine, using the credentials `username` and `password` to authenticate the connection.

4.2.2 ContentLoader

This program loads the specified document in the database. It loads the file with a URI equal to the fully-qualified pathname of the file. The following is a sample command to run the `ContentLoader` class:

```
java -classpath "XCC_HOME/lib/xcc.jar:XCC_HOME/lib/xccexamples.jar"
     com.marklogic.xcc.examples.ContentLoader
     xcc://username:password@localhost:8021 hello.xml
```

This loads the file at `hello.xml` to a document with the fully-qualified pathname of `hello.xml` (for example, `c:\xcc\examples\hello.xml`). It loads it into the default database of the XDBC Server listening on port 8021 of the local machine, using the credentials `username` and `password` to authenticate the connection.

4.2.3 DynamicContentStream

This program demonstrates inserting dynamic content without spawning a new thread, by using `ContentFactory.newUnBufferedContent`. The following is a sample command to run the `DynamicContentStream` program:

```
java -classpath "XCC_HOME/lib/xcc.jar:XCC_HOME/lib/xccexamples.jar"
     com.marklogic.xcc.examples.DynamicContentStream
     xcc://username:password@localhost:8021
     /any/valid/docURI
```

This inserts a new document at `/any/valid/docURI` in the contentbase described by the XDBC App Server URI `xcc://username:password@localhost:8021`. For demonstration purposes, the document contents are generated dynamically by the sample application.

4.2.4 HelloSecureWorld

This program runs a query on MarkLogic Server that returns the string `"Hello World"`, using an SSL/TLS connection to MarkLogic Server.

This example can load a specified key store of trusted signing authorities, use the Java default key store, or use a stub that accepts any server certificate. It can also load client certificates from a specified key store, or connect without a certificate.

The server certificate command line parameter may be any of the following:

- the path to a Java Key Store containing trusted signing authorities
- `DEFAULT` - use the Java default cacerts
- `ANY` - accept any server certificate

Optionally, you may also specify the path to a Java Key Store containing client certificates, along with its passphrase.

The following is a sample command to run the `HelloSecureWorld` class, accepting any certificate and no client authentication (the XDBC App Server needs `ssl require client certificate` set to `false` for this configuration).

```
java -classpath "XCC_HOME/lib/xcc.jar:XCC_HOME/lib/xccexamples.jar"
     com.marklogic.xcc.examples.HelloSecureWorld
     xccs://username:password@localhost:8021 ANY
```

Note: When FIPS mode is enabled (which is the default), you need to use strong client ciphers, and you need to download and install the Java Cryptography Extension (JCE) Unlimited Strength package (http://www.oracle.com/technetwork/java/javase/downloads/jce-7-download-432124.html).

4.2.5 HelloWorld

This program runs a query on MarkLogic Server that returns the string `"Hello World"`. The following is a sample command to run the `HelloWorld` class:

```
java -classpath "XCC_HOME/lib/xcc.jar:XCC_HOME/lib/xccexamples.jar"
     com.marklogic.xcc.examples.HelloWorld
     xcc://username:password@localhost:8021
```

4.2.6 ModuleRunner

This program allows you to invoke a module on the server. The module must exist under the XDBC server root, either in the database (when a modules database is configured) or on the filesystem (when the filesystem is configured for modules). The following is a sample command to run the `ModuleRunner` class:

```
java -classpath "XCC_HOME/lib/xcc.jar:XCC_HOME/lib/xccexamples.jar"
     com.marklogic.xcc.examples.ModuleRunner
     xcc://username:password@localhost:8021 hello.xqy
```

This invokes the module named `hello.xqy`. The request is submitted to the XDBC Server running on the local machine at port 8021, using the credentials `username` and `password` to authenticate the connection. The module path is resolved relative to the XDBC Server root.

4.2.7 OutputStreamInserter

This program demonstrates inserting dynamic content by spawning a new thread to write data to the receiving end of an input stream. For an alternative method of inserting dynamically generated streamed content, see the example “DynamicContentStream” on page 37.

The following is a sample command to run the `OutputStreamInserter` program:

```
java -classpath "XCC_HOME/lib/xcc.jar:XCC_HOME/lib/xccexamples.jar"
     com.marklogic.xcc.examples.OutputStreamInserter
     xcc://username:password@localhost:8021
     /any/valid/docURI
```

This inserts a new document at `/any/valid/docURI` in the contentbase described by the XDBC App Server URI `xcc://username:password@localhost:8021`. For demonstration purposes, the document contents are generated dynamically by the sample application.

4.2.8 SimpleQueryRunner

This program allows you to store XQuery in a file and then submit the XQuery to MarkLogic Server. The following is a sample command to run the `SimpleQueryRunner` class:

```
java -classpath "XCC_HOME/lib/xcc.jar:XCC_HOME/lib/xccexamples.jar"
     com.marklogic.xcc.examples.SimpleQueryRunner
     xcc://username:password@localhost:8021 hello.xqy
```

This submits the contents of the `hello.xqy` file to a MarkLogic Server XDBC Server running on the local machine at port 8021, using the credentials `username` and `password` to authenticate the connection.

4.2.9 XA

This program demonstrates using MarkLogic Server in a distributed XA transaction. The sample uses JBoss as a Transaction Manager, with two MarkLogic Server clusters participating in the distributed transaction.

The sample uses libraries from JBossTS. Download and install JBossTS 4.15.0 or later from the following location. The JBoss Application Server is not needed.

http://www.jboss.org/jbosstm

Include the following libraries from the JBossTS package on your classpath. `JBOSS_HOME` is the directory where JBossTS is installed.

- `JBOSSTS_HOME/lib/jbossjta.jar`

- `JBOSSTS_HOME/lib/ext/jboss-transaction-api_1.1_spec.jar` (or other JTA implementation)
- `JBOSSTS_HOME/lib/ext/jboss-logging.jar`

The participating MarkLogic Server clusters may simply be two XDBC App Servers on the same instance, serving different databases.

The following is a sample command to run the `ModuleRunner` class. Change `XCC_HOME`, `JBOSSTS_HOME`, and the two content base URIs to match your installation.

```
java -classpath "XCC_HOME/lib/xcc.jar:XCC_HOME/lib/xccexamples.jar:
     JBOSSTS_HOME/lib/jbossjta.jar:
     JBOSSTS_HOME/lib/ext/jboss-transaction-api_1.1_spec.jar:
     JBOSSTS_HOME/lib/ext/jboss-logging.jar"
     com.marklogic.xcc.examples.XA
     xcc://username1:password1@host1:port1/contentbase1
     xcc://username2:password2@host2:port2/contentbase2
```

The sample program enlists each contentbase as a resource with the JBoss Transaction Manager, and then inserts a document in each contentbase, as part of a single global transaction. The output from xdmp:host-status relevant to each branch's transaction is printed out. This information includes the global transaction id and branch qualifer, as well as the local transaction id:

```
<transaction xmlns="http://marklogic.com/xdmp/status/host">
  <transaction-id>750821115632601886</transaction-id>
  <host-id>8814043795788656336</host-id>
  <server-id>4366002345564888063</server-id>
  <xid format-id="131076" xmlns="http://marklogic.com/xdmp/xa">
    <global-transaction-id>...825D0000000931</global-transaction-id>
    <branch-qualifier>...825D0000000A</branch-qualifier>
  </xid>
  <name/>
  <mode>update</mode>
  <timestamp>0</timestamp>
  <state>active</state>
  <database>2901782035623219290</database>
  <canceled>false</canceled>
  ...
</transaction>
<transaction xmlns="http://marklogic.com/xdmp/status/host">
  <transaction-id>4949312806261581854</transaction-id>
  <host-id>8814043795788656336</host-id>
  <server-id>7579943212553445602</server-id>
  <xid format-id="131076" xmlns="http://marklogic.com/xdmp/xa">
    <global-transaction-id>0...825D0000000931</global-transaction-id>
    <branch-qualifier>...825D0000000D</branch-qualifier>
  </xid>
  <name/>
  <mode>update</mode>
  <timestamp>0</timestamp>
  <state>active</state>
  <database>2852559629722654718</database>
```

```
    <canceled>false</canceled>
    ...
  </transaction>
```

For details on use XA with MarkLogic Server, see “Multi-Statement Transactions” on page 19.

5.0 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.

6.0 Copyright

MarkLogic Server 9.0 and supporting products.
Last updated: November 18, 2019

COPYRIGHT

© 2019 MarkLogic Corporation. All rights reserved.
This technology is protected by U.S. Patent No. 7,127,469B2, U.S. Patent No. 7,171,404B2, U.S. Patent No. 7,756,858 B2, and U.S. Patent No 7,962,474 B2, US 8,892,599, and US 8,935,267.

The MarkLogic software is protected by United States and international copyright laws, and incorporates certain third party libraries and components which are subject to the attributions, terms, conditions and disclaimers set forth below.

For all copyright notices, including third-party copyright notices, see the Combined Product Notices for your version of MarkLogic.