Xindice 1.0 Developers Guide

$Revision: 1.2 $

by Kimbro Staken

1. Introduction to Programming Xindice

1.1. Accessing the Server

The Xindice server can be accessed either programmatically through the server's APIs or from the command line using the provided command line tools. This document covers programmatic access, for more information on using the command line interface please refer to the Xindice Users Guide.

1.1.1. API's

Xindice currently offers three layers of APIs that can be used to develop applications.

- XML:DB XML Database API used to develop Xindice applications in Java. This is the primary API used to develop applications and will be the API given the most coverage in this manual. The XML:DB API is built on top of the Xindice CORBA API. Xindice currently implements the May 07, 2001 draft of the XML:DB API. This API will change slightly in the future to track the development of the XML:DB API.
- CORBA API used when accessing Xindice from a language other then Java. The CORBA API is built on top of the Core Server API. It is likely that the CORBA API will be replaced with a new network API in a later version of Xindice.
- Core Server API is the internal Java API of the core database engine. This API is used to build the CORBA API and is also used when writing XMLObject server extensions. This is the lowest level API and is only available to software running in the same Java VM as the database engine itself. the most common use of this API is in developing XMLObject server extensions.

The most common API for end user applications is the XML:DB XML Database API that was developed by the XML:DB Initiative. This API is a vendor neutral API intended to make it possible to build applications that will work with more then one XML database without too much difficulty. This is similar to the capabilities provided by JDBC for relational databases. More information about this API can be found on the XML:DB Initiative web site,

http://www.xmldb.org. Most programming examples in this manual will use the XML:DB API. The Xindice implementation of the API is a Core Level 1 implementation.

The Xindice server also exposes a CORBA API that is used to implement the XML:DB API. The CORBA API will mainly be of interest to those who want to access Xindice from a language other then Java. Any language that supports a CORBA binding should be able to utilize the services of the Xindice server via the CORBA API. This document does not cover development with the CORBA API as the XML:DB API is the preferred mechanism for developing Xindice applications. If you are developing applications in Java you can safely ignore the existence of this API. The CORBA API will be covered in a seperate document to be written at a later time.

The final API for Xindice is the Core Server API.

1.2. Introducing the XML:DB XML Database API

XML:DB API is being developed by the XML:DB Initiative to facilitate the development of applications that function with minimal change on more then one XML database. This is roughly equivalent to the functionality provided by JDBC or ODBC for providing access to relational databases. Xindice provides an implementation of the XML:DB API that also serves as the primary programming API for Xindice.

The XML:DB API is based around the concept of collections that store resources. A resource can be an XML Document, a binary blob or some type that is not currently defined. Collections can be arranged in a hierarchical fashion. This makes the architecture very similar to that of a typical Windows or UNIX file system. What is different however, is that collections also expose services that allow you to do things such as query XML documents using XPath or update resources in a transactionally secure manner.

The XML:DB API defines several levels of interoperability called Core Levels in XML:DB terminology. The Xindice implementation of the API is a Core Level 1 implementation plus implementations of some of the optional services.

The additional supported services include.

- XUpdateQueryService - Enables execution of XUpdate queries against the database.
- CollectionManagementService - Provides basic facilities to create and remove collections.

In addition to Core Level 1 support the Xindice implementation also supports a few added services that are specific to Xindice. These services exist because the functionality is necessary to fully utilize all the capabilities provided by Xindice. However, they are proprietary to Xindice and will not function unchanged on other XML databases.

The following services are currently provided by Xindice and are not part of the common XML:DB API.

- DatabaseInstanceManager - Provides the ability to control the operation of the server programatically.
- CollectionManager - Provides the ability to create and configure collection instances within the server. This is a much more functional version of CollectionManagementService that will only work with Xindice.

While this guide aims to provide some useful examples and to guide you in the process of getting to know how to program Xindice it is also useful to know that there is a good source of example code within the server it self. The Xindice command line tools are built 100% on the XML:DB API and provide a pretty comprehensive set of examples in how to use the API. This is especially true when it comes to the Xindice specific services that are included with the server. The source code for all the command line tools can be found in `Xindice/java/src/org/apache/xindice/tools/command`.

1.3. Setting up Your Build Environment

Before you can build applications for Xindice you need to make sure you have your build environment properly setup. This mainly consists of making sure that you have the proper VM version and a properly configured `CLASSPATH`.

To build applications for Xindice you can use JDK 1.3 or 1.4. JDK 1.2 and below will not work. If you have more than one Java VM installed make sure that your `JAVA_HOME` environment variable and `PATH` environment variable both include the correct path.

Once you have your Java VM properly configured you need to add a few jar files to your `CLASSPATH`. The following list of jars are required and should be made available on your `CLASSPATH`. All required jars can be found in `Xindice/java/lib`

- xindice.jar - contains the main Xindice classes that are used by the client API.
- xmldb.jar - contains the XML:DB API interfaces.
- openorb-1.2.0.jar - contains the CORBA ORB implementation used by the client API to communicate with the server.
- xerces-1.4.3.jar - contains the Xerces XML parser.
- xalan-2.0.1.jar - contains the Xalan XSL-T engine. This jar isn't absolutely required but you'll probably want it so it's worth adding it.

1.4. Preparing the Server For the Examples

Before we get to some example code, we need to do a little work to setup the server. Don't worry nothing hard.

First we need to make sure the addressbook collection exists. If you followed the install instructions completely you should have already created this, but if not you should do so now. To find out if the collection exists you can run:

```
xindice lc -c /db
```

If you don't see 'addressbook' listed in the result then you need to create the collection. To create it just run:

```
xindice ac -c /db -n addressbook
```

Now that we have the collection, we can add a few example documents so that we have something to play with. You can find the examples in your Xindice installation in the directory `java/examples/guide/xml`. Run these commands to add the documents.

```
cd $XINDICE_HOME/java/examples/guide/xml
xindice ad -c /db/addressbook -f address1.xml -n address1
xindice ad -c /db/addressbook -f address2.xml -n address2
```

If you're on Windows you'll need to adjust the path in the cd command for your platform. Most of the examples in the manual will be written for UNIX but will work fine in Windows if you just replace / with \ and $XINDICE_HOME with %XINDICE_HOME%.

That wasn't so bad and now we're set to look at some example code.

1.5. Diving in With an Example Program

1.5.1. Simple XML:DB Example Program

This example simply executes an XPath query against a collection, retrieves the results as text and prints them out.

You can find the source code for this example in Xindice/java/examples/guide/src/org/apache/xindice/examples/Example1.java

```
package org.apache.xindice.examples;

import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;

public class Example1 {
   public static void main(String[] args) throws Exception {
    Collection col = null;
    try {
       String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
       Class c = Class.forName(driver);
```

```
       Database database = (Database) c.newInstance();
       DatabaseManager.registerDatabase(database);

       col = DatabaseManager.getCollection("xmldb:xindice:///db/addressbook");

       String xpath = "//person[fname='John']";
       XPathQueryService service =
        (XPathQueryService) col.getService("XPathQueryService", "1.0");
       ResourceSet resultSet = service.query(xpath);
       ResourceIterator results = resultSet.getIterator();
       while (results.hasMoreResources()) {
        Resource res = results.nextResource();
        System.out.println((String) res.getContent());
       }
    }
    catch (XMLDBException e) {
       System.err.println("XML:DB Exception occured " + e.errorCode);
    }
    finally {
       if (col != null) {
        col.close();
       }
    }
   }
}
```

Before diving into the gory detail of what this program is doing, let's run it and see what we get back.

If you have a binary build of Xindice the examples are already built and you can run this example by typing.

```
cd $XINDICE_HOME/java/examples/guide
./run org.apache.xindice.examples.Example1
```

If all goes well, you should see a result that looks something like this.

```
<?xml version="1.0"?>
  <person xmlns:src="http://xml.apache.org/xindice/Query"
        src:col="/db/addressbook" src:key="address1">
   <fname>John</fname>
   <lname>Smith</lname>
   <phone type="work">563-456-7890</phone>
   <phone type="home">534-567-8901</phone>
   <email type="home">jsmith@somemail.com</email>
   <email type="work">john@lovesushi.com</email>
   <address type="home">34 S. Colon St.</address>
   <address type="work">9967 W. Shrimp Ave.</address>
  </person>
```

Now that we've seen the result, let's dive in and look at the code in detail. While this isn't the simplest possible example program to start with it does a nice job of showing all the basic techniques used when building applications with the XML:DB API.

To begin the program imports several XML:DB packages.

```
import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;
```

These import the basic classes required by the API. `import org.xmldb.api.base.*;` is the base API module and is required for all XML:DB applications. `import org.xmldb.api.*;` imports the all important `DatabaseManager` class which is the entry point into the API. `import org.xmldb.api.modules.*;` brings in the optional modules defined for the API. In this case the module we're interested in is `XPathQueryService`.

Before we can use the API we need to create and register the database driver we want to use. In this case since we're writing for Xindice we use `org.apache.xindice.client.xmldb.DatabaseImpl` for our driver and register it with the `DatabaseManager`

```
String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
Class c = Class.forName(driver);

Database database = (Database) c.newInstance();
DatabaseManager.registerDatabase(database);
```

Now that our driver is registered we're ready to retrieve the collection that we want to work with.

```
Collection col =
   DatabaseManager.getCollection("xmldb:xindice:///db/addressbook");
```

In the XML:DB API collections are retrieved by calling `getCollection` and handing it a URI that specifies the collection we want. The format of this URI will vary depending on the database implementation being used but will always begin with `xmldb:` and be followed by a database specific database name, `xindice:` in the case of Xindice.

The rest of the URI is a path used to locate the collection you want to work with. This path begins with the name of the root collection for the Xindice instance that you are trying to connect with. All Xindice instances must have a unique name for the root collection. The reason for this is that the name of the root collection is also the name of the database instance and that name is what the Xindice server uses to register itself with the naming service. In all examples in this guide the root collection is called `db`. This is the default name used for a newly installed instance of Xindice. If you have more then one instance of Xindice running that you must be sure to change the names on all other instances so that they are unique. Once you do this you can switch between the instances by simply changing the first component of the path.

```
xindice:///db/news
xindice:///db2/news
```

These paths will switch between a Xindice server with a root collection of db and one of db2. These instances could be on the same machine or on two completely different machines and to your application there is no significant difference.

After the root collection name the rest of the URI simply consists of the path to locate the collection you want to work with.

Now that we have a reference to the collection that we want to work with we need to get a reference to the `XPathQueryService` service for that collection.

```
String xpath = "//person[fname='John']";
XPathQueryService service =
   (XPathQueryService) col.getService("XPathQueryService", "1.0");
ResourceSet resultSet = service.query(xpath);
```

Services provide a way to extend the functionality of the XML:DB API as well as enabling the definition of optional functionality. In this case the `XPathQueryService` is an optional part of Core Level 0 but is a required part of Core Level 1. Since Xindice provides a Core Level 1 XML:DB API implementation the `XPathQueryService` is available.

To retrieve a service you must know the name of the service that you want as well as the version. Services define their own custom interfaces so you must cast the result of the getService() call to the appropriate service type before you can call its methods. The `XPathQueryService` defines a method `query()` that takes an XPath string as an argument. Different services will define different sets of methods.

Now that we have an `XPathQueryService` reference and have called the `query()` method we get a `ResourceSet` containing the results. Since we just want to print out the results of the query, we need to get an iterator for our results and then use it to print out the results.

```
ResourceIterator results = resultSet.getIterator();
while (results.hasMoreResources()) {
   Resource res = results.nextResource();
   System.out.println((String) res.getContent());
}
```

Resources are another important concept within the XML:DB API. Since XML can be accessed with multiple APIs and since an XML database could potentialy store more the one type of data, resources provide an abstract way to access the data in the database. The Xindice implementation only supports XMLResource but other vendors may support additional resource types as well.

XMLResource provides access to the underlying XML data as either text, a DOM Node or via SAX ContentHandlers. In our example we're simply working with the content as text but we could just as easily have called `getContentAsDom()` to get the content as a DOM Node. Since we just want to print the XML out to the screen it is easier to just work with text.

The final element about our example program worth noting is the finally clause.

```
finally {
   if (col != null) {
    col.close();
   }
}
```

The finally clause closes the collection that we created earlier. This is vitally important and should never be overlooked. Closing the collection releases all the resources consumed by the collection. In the Xindice implementation this will make sure that the CORBA resources are released properly. Failure to properly call close on the collection will result in a resource leak within the server.

1.6. Accessing Xindice Remotely

By default Xindice assumes that the client and server are running on the same machine. In most configurations this will not be the case so it will be necessary to include the hostname and port of the server where Xindice is running in your URIs. The port you use is the port that the Xindice HTTP server is listening on. The port setting is displayed when you start the server. By default the configuration used is `xindice://localhost:4080`. To access the collection /db/addressbook on host xml.apache.org port 4090 the URI would look something like this `xindice://xml.apache.org:4090/db/addressbook`. If you are running the client and server on the same machine you do not need to specify a host or port. All examples in this document assume the client and server are on the same machine.

If you are having problems accessing Xindice remotely this may be the result of the server publishing an incorrect IP address as part of its CORBA IOR. The IOR is generated by the ORB using the hostname setting of the server. If the hostname is set to a fully qualified domain name that can not be resolved by the client you will not be able to remotely access the server. To fix this you must insure the host name on the server is resolvable by the client. To see the setting of the host name on the server you should be able to type `hostname` at a command prompt. If the hostname is not a fully qualified domain name then the ORB will use the IP address of the server instead of the name and you shouldn't have any problems.

2. Managing Documents

In this chapter we'll look at using the XML:DB API to manage documents within the Xindice

server. As part of this we'll look at some sample code that could be used to manage the data used by the AddressBook example application included with the server and discussed in more detail later.

When looking at managing documents with the XML:DB API the first thing we need to confront is that the API doesn't actually work directly with documents. It works with what the API calls resources that are an abstraction of a document. This abstraction allows you to work with the same document as either text, a DOM tree or SAX events. This is important to understand as the use of resources runs as a common thread throughout the XML:DB API. The XML:DB API actually defines more then one type of resource however Xindice does not implement anything beyond XMLResource.

2.1. Creating a Collection

Before we can work with any data in the database we need to create a collection to hold our data. While we could easily create this collection using the command line tools it will be more fun to see how you might do this from your own program. This will also show you a quick example of using the Xindice specific `CollectionManager` service to manage collections. This guide doesn't go into detail about using this service but you can find lots of examples by looking at the source code to the command line tool commands in the package `org/apache/xindice/tools/commands`.

The collection we want to create will be named `mycollection` and will be a child of the root collection.

2.1.1. Creating a Collection

```
package org.apache.xindice.examples;

import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;

// For the Xindice specific CollectionManager service
import org.apache.xindice.client.xmldb.services.*;

import org.apache.xindice.xml.dom.*;

public class CreateCollection {
   public static void main(String[] args) throws Exception {
    Collection col = null;
    try {
       String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
       Class c = Class.forName(driver);

       Database database = (Database) c.newInstance();
```

```
        DatabaseManager.registerDatabase(database);
        col =
         DatabaseManager.getCollection("xmldb:xindice:///db/");

        String collectionName = "mycollection";
        CollectionManager service =
         (CollectionManager) col.getService("CollectionManager", "1.0");

        // Build up the Collection XML configuration.
        String collectionConfig =
         "<collection compressed=\"true\" name=\"" + collectionName + "\">" +
         "   <filer class=\"org.apache.xindice.core.filer.BTreeFiler\" gzip=\"true\"/>"
         "</collection>";

        service.createCollection(collectionName,
            DOMParser.toDocument(collectionConfig));

        System.out.println("Collection " + collectionName + " created.");
    }
    catch (XMLDBException e) {
        System.err.println("XML:DB Exception occured " + e.errorCode);
    }
    finally {
        if (col != null) {
         col.close();
        }
    }
   }
}
```

With this example you can see a basic example of how to create the `CollectionManager` service and use it to create the collection. This service is proprietary to Xindice so if you use it in your application you will not be able to port it to another server. However, if you have the need to create collections within your programs this is currently the most powerful way to do it.

The trickiest part of creating a collection is creating the proper XML configuration to hand to the `createCollection` method. This XML is the exact same thing that is placed into the system.xml file. At this time these XML configurations are not documented so to see what they need to be you should look for examples in system.xml and the source code for the command line tools. Future versions of this documentation will cover this area in more detail.

2.2. Working with Documents

Now that we have a collection to store our data, we need to add some data to it. We could use the command line tools to do this but since we want to learn how the XML:DB API works we'll look at how we can do this in a program that we write.

For our examples in this chapter we'll work with some very simple XML files that could be

used to represent a person in an address book. Later in the guide we'll look at an example application that implements the actual address book functionality. Each address book entry is stored in a seperate XML file.

2.2.1. Example Document

```
<person>
   <fname>John</fname>
   <lname>Smith</lname>
   <phone type="work">563-456-7890</phone>
   <phone type="home">534-567-8901</phone>
   <email type="home">jsmith@somemail.com</email>
   <email type="work">john@lovesushi.com</email>
   <address type="home">34 S. Colon St.</address>
   <address type="work">9967 W. Shrimp Ave.</address>
</person>
```

If we store this example XML into a file we can then load it into our addressbook collection using a simple program.

2.2.2. Adding an XML File to the Database

```
package org.apache.xindice.examples;

import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;

import java.io.*;

public class AddDocument {
   public static void main(String[] args) throws Exception {
    Collection col = null;
    try {
       String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
       Class c = Class.forName(driver);

       Database database = (Database) c.newInstance();
       DatabaseManager.registerDatabase(database);
       col =
        DatabaseManager.getCollection("xmldb:xindice:///db/addressbook");

       String data = readFileFromDisk(args[0]);

       XMLResource document = (XMLResource) col.createResource(null,
        "XMLResource");
       document.setContent(data);
       col.storeResource(document);
       System.out.println("Document " + args[0] + " inserted");
    }
    catch (XMLDBException e) {
       System.err.println("XML:DB Exception occured " + e.errorCode);
```

```
    }
    finally {
       if (col != null) {
        col.close();
       }
    }
   }

   public static String readFileFromDisk(String fileName) throws Exception {
    File file = new File(fileName);
    FileInputStream insr = new FileInputStream(file);

    byte[] fileBuffer = new byte[(int)file.length()];

    insr.read(fileBuffer);
    insr.close();

    return new String(fileBuffer);
   }
}
```

Much of this program is similar to what we've already seen in our other XML:DB programs. Really the only difference is the code to add the document.

Documents are added to the server by first creating a new resource implementation from a collection, setting its content and then storing the resource to the collection. The type of resource that is created is an XMLResource this can be used to store XML as either text, a DOM Node or a SAX ContentHandler.

If you had your content already in a DOM tree you could also add the document as a DOM.

```
XMLResource document = (XMLResource) col.createResource(null);
document.setContentAsDOM(doc); // doc is a DOM document
col.storeResource(document);
```

The only difference here is that you must have the document as a DOM Document already and then call `setContentAsDOM()`. From there the resource works the same as always.

One thing to note is that a resource must be stored in the same collection from which it was originally created.

2.2.3. Retrieving an XML Document from the Database

```
package org.apache.xindice.examples;

import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;

import java.io.*;
```

```
public class RetrieveDocument {
   public static void main(String[] args) throws Exception {
    Collection col = null;
    try {
       String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
       Class c = Class.forName(driver);

       Database database = (Database) c.newInstance();
       DatabaseManager.registerDatabase(database);
       col =
        DatabaseManager.getCollection("xmldb:xindice:///db/addressbook");

       XMLResource document = (XMLResource) col.getResource(args[0]);
       if (document != null) {
        System.out.println("Document " + args[0]);
        System.out.println(document.getContent());
       }
       else {
        System.out.println("Document not found");
       }
    }
    catch (XMLDBException e) {
       System.err.println("XML:DB Exception occured " + e.errorCode);
    }
    finally {
       if (col != null) {
        col.close();
       }
    }
   }
}
```

2.2.4. Deleting an XML Document from the Database

```
package org.apache.xindice.examples;

import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;

import java.io.*;

public class DeleteDocument {
   public static void main(String[] args) throws Exception {
    Collection col = null;
    try {
       String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
       Class c = Class.forName(driver);

       Database database = (Database) c.newInstance();
       DatabaseManager.registerDatabase(database);
       col =
        DatabaseManager.getCollection("xmldb:xindice:///db/addressbook");
```

```
        Resource document = col.getResource(args[0]);
        col.removeResource(document);
        System.out.println("Document " + args[0] + " removed");
     }
     catch (XMLDBException e) {
        System.err.println("XML:DB Exception occured " + e.errorCode);
     }
     finally {
        if (col != null) {
         col.close();
        }
     }
    }
}
```

3. Using XPath to Query the Database

3.1. Introduction

Xindice currently supports XPath as a query language. In many applications XPath is only applied at the document level but in Xindice XPath queries are executed at the collection level. This means that a query can be run against multiple documents and the result set will contain all matching nodes from all documents in the collection. The Xindice server also support the creation of indexes on particular XPaths to speed up commonly used XPath queries.

3.2. Using the XML:DB Java API

The XML:DB API defines operations for searching single documents as well as collections of XML documents using XPath. These operations are exposed through the XPathQueryService. In order to query single documents you use the `queryResource()` method and to query an entire collection you use the `query()` method.

3.2.1. Querying with XPath

This example simply executes an XPath query against a collection, retrieves the results as text and prints them out.

You can find the source code for this example in Xindice/java/examples/guide/src/org/apache/xindice/examples/Example1.java

```
package org.apache.xindice.examples;

import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;
```

```
public class Example1 {
   public static void main(String[] args) throws Exception {
    Collection col = null;
    try {
       String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
       Class c = Class.forName(driver);

       Database database = (Database) c.newInstance();
       DatabaseManager.registerDatabase(database);

       col = DatabaseManager.getCollection("xmldb:xindice:///db/addressbook");

       String xpath = "//person[fname='John']";
       XPathQueryService service =
        (XPathQueryService) col.getService("XPathQueryService", "1.0");
       ResourceSet resultSet = service.query(xpath);
       ResourceIterator results = resultSet.getIterator();
       while (results.hasMoreResources()) {
        Resource res = results.nextResource();
        System.out.println((String) res.getContent());
       }
    }
    catch (XMLDBException e) {
       System.err.println("XML:DB Exception occured " + e.errorCode);
    }
    finally {
       if (col != null) {
        col.close();
       }
    }
   }
}
```

3.2.2. Query Results

XPath queries always return anonymous resources. An anonymous resource is one that has no Resource ID. If a resource is anonymous, a call to Resource.getID() returns null. Queries return anonymous resources because a query can return all or a fragment of a document. Documents have a Resource ID and can be retrieved (in their entirety) from the database using that ID. A fragment of a document has no ID. The fragment can only be retrieved using an XPath that selects the same fragment of the document. The burden of knowing whether a query returns a complete document or a fragment is placed on the user. This is simple for the user (users know the structure of their data and syntax of their queries). Requiring the implementation to determine if a query selects a complete Resource or a fragment and to provide either an identified or anonymous Resource depending on that determination places too great a burden on the implementation. Thus, XPath queries always return anonymous Resources. A call to Resource.getDocumentID() will provide the ID of the Resource containing the Resource returned by the query. If a user knows that the Resource is in fact the

complete document, then the ID may be used as a locator of the returned Resource interchangeably with the XPath query.

TODO: cover namespace support

4. Using XUpdate to Modify the Database

4.1. Introduction

XUpdate is a specification under development by the XML:DB Initiative to enable simpler updating of XML documents. It is useful within the context of an XML database as well as in standalone implementations for general XML applications. XUpdate gives you a declarative method to insert nodes, remove nodes, and change nodes within an XML document. The syntax is specified in the XUpdate working draft available on the XML:DB Initiative website.

The XUpdate implementation in Xindice is based around the Lexus XUpdate implementation that was developed by the Infozone Group.

The general model around XUpdate is to use an `xupdate:modifications` container to batch a series of XUpdate commands. All commands will be performed in series against either a single XML document or an entire collection of XML documents as specified by the developer.

Execution of XUpdate commands is performed in two phases. First selecting a node set within the document or collection and then applying a change to the selected nodes.

4.1.1. Basic XUpdate Insert Command

```
<xupdate:modifications version="1.0"
      xmlns:xupdate="http://www.xmldb.org/xupdate">

   <xupdate:insert-after select="/addresses/address[1]" >

    <xupdate:element name="address">
       <xupdate:attribute name="id">2</xupdate:attribute>
       <fullname>John Smith</fullname>
       <born day='2' month='12' year='1974'/>
       <country>Germany</country>
     </xupdate:element>

   </xupdate:insert-after>
</xupdate:modifications>
```

4.2. XUpdate Commands

- `xupdate:insert-before` - Inserts a new node in document order before the selected node.
- `xupdate:insert-after` - Inserts a new node in document order after the selected node.
- `xupdate:update` - Replaces all child nodes of the selected node with the specified nodes.
- `xupdate:append` - Appends the specified node to the content of the selected node.
- `xupdate:remove` - Remove the selected node
- `xupdate:rename` - Renames the selected node
- `xupdate:variable` - Defines a variable containing a node list that can be reused in later operations.

4.3. XUpdate Node Construction

- `xupdate:element` - Creates a new element in the document.
- `xupdate:attribute` - Creates a new attribute node associated with an `xupdate:element`.
- `xupdate:text` - Creates a text content node in the document.
- `xupdate:processing-instruction` - Creates a processing instruction node in the document.
- `xupdate:comment` - Creates a new comment node in the document.

4.4. Using the XML:DB API for XUpdate

The XML:DB API provides an XUpdateQueryService to enable executing XUpdate commands against single documents or collections of documents. To update a single document you use the `updateResource()` method and to apply the updates to an entire collection you use the `update()` method.

The next example program applies a set of XUpdate modifications to an entire collection of data. It first removes all elements that match the XPath `/person/phone[@type = 'home']` and then adds a new entry after all elements that match the XPath `/person/phone[@type = 'work']`

4.4.1. Using XUpdate to modify the database

```
import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;

/**
 * Simple XML:DB API example to update the database.
 */
```

```
public class XUpdate {
   public static void main(String[] args) throws Exception {
    Collection col = null;
    try {
       String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
       Class c = Class.forName(driver);

       Database database = (Database) c.newInstance();
       DatabaseManager.registerDatabase(database);
       col =
        DatabaseManager.getCollection("xmldb:xindice:///db/addressbook");

       String xupdate = "<xu:modifications version=\"1.0\"" +
        "    xmlns:xu=\"http://www.xmldb.org/xupdate\">" +
        "   <xu:remove select=\"/person/phone[@type = 'home']\"/>" +
        "   <xu:update select=\"/person/phone[@type = 'work']\">" +
        "     480-300-3003" +
        "   </xu:update>" +
        "</xu:modifications>";

       XUpdateQueryService service =
        (XUpdateQueryService) col.getService("XUpdateQueryService", "1.0");
       service.update(xupdate);
    }
    catch (XMLDBException e) {
       System.err.println("XML:DB Exception occured " + e.errorCode + " " +
        e.getMessage());
    }
    finally {
       if (col != null) {
        col.close();
       }
    }
   }
}
```

5. Address Book Example Application

5.1.

The address book example is a simple servlet based application constructued using Xindice. For more information on this example look in the `Xindice/java/examples/addressbook` directory.

TODO: Add more detail about building servlet applications.

6. Experimental Features

There are a couple features in Xindice that are definitely experimental. These features can be interesing to explore to see some things that could be useful in future versions of Xindice but

they should not be considered complete or stable.