Writing Java Applications with the XML:DB API

The preferred way to work with eXist-db when developing Java applications is to use the XML:DB API. This API provides a common interface to native or XML-enabled databases and supports the development of portable, reusable applications. eXist's implementation of the XML:DB standards follows the Xindice implementation, and conforms to the latest working drafts put forth by the XML:DB Initiative. For more information, refer to the Javadocs for this API.

The basic components employed by the XML:DB API are *drivers*, *collections*, *resources* and *services*.

Drivers are implementations of the database interface that encapsulate the database access logic for specific XML database products. They are provided by the product vendor and must be registered with the database manager.

A ***collection*** is a hierarchical container for *resources* and further sub-collections. Currently two different resources are defined by the API: XMLResource and BinaryResource. An XMLResource represents an XML document or a document fragment, selected by a previously executed XPath query.

Finally, ***services*** are requested for special tasks such as querying a collection with XPath, or managing a collection.

There are several XML:DB examples provided in eXist's `samples` directory . To start an example, use the `start.jar` jar file and pass the name of the example class as the first parameter, for instance:

```
java -jar start.jar
                org.exist.examples.xmldb.Retrieve [- other options]
```

Programming with the XML:DB API is straightforward. You will find some code examples in the `samples/org/exist/examples/xmldb` directory. In the following simple example, a document can be retrieved from the eXist server and printed to standard output.

```
import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;
import javax.xml.transform.OutputKeys;
import org.exist.xmldb.EXistResource;
public class RetrieveExample {

    private static String URI = "xmldb:exist://localhost:8080/exist/xmlrpc";
    /**
     * args[0] Should be the name of the collection to access
     * args[1] Should be the name of the resource to read from the collection
     */
    public static void main(String args[]) throws Exception {

        final String driver = "org.exist.xmldb.DatabaseImpl";

        // initialize database driver
        Class cl = Class.forName(driver);
        Database database = (Database) cl.newInstance();
        database.setProperty("create-database", "true");
        DatabaseManager.registerDatabase(database);

        Collection col = null;
        XMLResource res = null;
        try {
            // get the collection
            col = DatabaseManager.getCollection(URI + args[0]);
            col.setProperty(OutputKeys.INDENT, "no");
            res = (XMLResource)col.getResource(args[1]);

            if(res == null) {
                System.out.println("document not found!");
            } else {
                System.out.println(res.getContent());
            }
        } finally {
            //dont forget to clean up!

            if(res != null) {
                try { ((EXistResource)res).freeResources(); } catch(XMLDBExcepti
on xe) {xe.printStackTrace();}
            }

            if(col != null) {
                try { col.close(); } catch(XMLDBException xe) {xe.printStackTrac
e();}
            }
        }
    }
}
```

Retrieving a Document with XML:DB

With this example, the database driver class for eXist (org.exist.xmldb.DatabaseImpl) is first registered with the DatabaseManager. Next we obtain a Collection object from the database manager by calling the static method DatabaseManger.getCollection(). The method expects a fully qualified URI for its parameter value, which identifies the desired collection. The format of this URI should look like the following:

```
xmldb:[DATABASE-ID]://[HOST-ADDRESS]/db/collection
```

Because more than one database driver can be registered with the database manager, the first part of the URI (xmldb:exist) is required to determine which driver class to use. The *database-id* is used by the database manager to select the correct driver from its list of available drivers. To use eXist, this ID should always be "exist" (unless you have set up multiple database instances; additional instances may have other names).

The final part of the URI identifies the collection path, and optionally the host address of the database server on the network. Internally, eXist uses two different driver implementations: The first talks to a remote database engine using XML-RPC calls, the second has direct access to a local instance of eXist. The root collection is always identified by /db. For example, the URI

```
xmldb:exist://localhost:8080/exist/xmlrpc/db/shakespeare/plays
```

references the Shakespeare collection on a remote server running the XML-RPC interface as a servlet at localhost:8080/exist/xmlrpc. If we leave out the host address, the XML:DB driver will try to connect to a locally attached database instance, e.g.:

```
xmldb:exist:///db/shakespeare/plays
```

In this case, we have to tell the XML:DB driver that it should create a new database instance if none has been started. This is done by setting the create-database property of class Database to "true" (more information on embedded use of eXist can be found in the deployment guide.

The setProperty calls are used to set database-specific parameters. In this case, pretty-printing of XML output is turned on for the collection. eXist uses the property keys defined in the standard Java package `javax.xml.transform`. Thus, in Java you can simply use class OutputKeys to get the correct keys.

Calling col.getResource() finally retrieves the document, which is returned as an XMLResource. All resources have a method getContent(), which returns the resource's content, depending on it's type. In this case we retrieve the content as type String.

To query the repository, we may either use the standard XPathQueryService or eXist's XQueryService class. The XML:DB API defines different kinds of services, which may or may not be provided by the database. The getService method of class Collection calls a service if it is available. The method expects the service name as the first parameter, and the version (as a string) as the second, which is used to distinguish between different versions of the service defined by the XML:DB API.

The following is an example of using the XML:DB API to execute a database query:

```
import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;
import org.exist.xmldb.EXistResource;
public class XPathExample {

    private static String URI = "xmldb:exist://localhost:8080/exist/xmlrpc";
    /**
     * args[0] Should be the name of the collection to access
     * args[1] Should be the XPath expression to execute
     */
    public static void main(String args[]) throws Exception {

        final String driver = "org.exist.xmldb.DatabaseImpl";

        // initialize database driver
        Class cl = Class.forName(driver);
        Database database = (Database) cl.newInstance();
        database.setProperty("create-database", "true");
        DatabaseManager.registerDatabase(database);

        Collection col = null;
        try {
            col = DatabaseManager.getCollection(URI + args[0]);
            XPathQueryService xpqs = (XPathQueryService)col.getService("XPathQue
ryService", "1.0");
            xpqs.setProperty("indent", "yes");
            ResourceSet result = xpqs.query(args[1]);
            ResourceIterator i = result.getIterator();
            Resource res = null;
            while(i.hasMoreResources()) {
                try {
                    res = i.nextResource();
                    System.out.println(res.getContent());
                } finally {
                    //dont forget to cleanup resources
                    try { ((EXistResource)res).freeResources(); } catch(XMLDBExc
eption xe) {xe.printStackTrace();}
                }
            }
        } finally {
            //dont forget to cleanup
            if(col != null) {
                try { col.close(); } catch(XMLDBException xe) {xe.printStackTrac
e();}
            }
        }
    }
}
```

Querying the Database with XPath(XML:DB API)

To execute the query, method service.query(xpath) is called. This method returns a ResourceSet, containing the Resources found by the query. ResourceSet.getIterator() gives us an iterator over these resources. Every Resource contains a single document fragment or value selected by the XPath expression.

Internally, eXist does not distinguish between XPath and XQuery expressions. XQueryService thus maps to the same implementation class as XPathQueryService. However, it provides a few additional methods. Most important, when talking to an embedded database, XQueryService allows for the XQuery expression to be compiled as an internal representation, which can then be reused. With compilation, the previous example code would look as follows:

```
import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;
import org.exist.xmldb.EXistResource;
public class XQueryExample {

    private static String URI = "xmldb:exist://localhost:8080/exist/xmlrpc";
    /**
     * args[0] Should be the name of the collection to access
     * args[1] Should be the XQuery to execute
     */
    public static void main(String args[]) throws Exception {

        final String driver = "org.exist.xmldb.DatabaseImpl";

        // initialize database driver
        Class cl = Class.forName(driver);
        Database database = (Database) cl.newInstance();
        database.setProperty("create-database", "true");
        DatabaseManager.registerDatabase(database);

        Collection col = null;
        try {
            col = DatabaseManager.getCollection(URI + args[0]);
            XQueryService xqs = (XQueryService) col.getService("XQueryService",
"1.0");
            xqs.setProperty("indent", "yes");

            CompiledExpression compiled = xqs.compile(args[1]);
            ResourceSet result = xqs.execute(compiled);
            ResourceIterator i = result.getIterator();
            Resource res = null;
            while(i.hasMoreResources()) {
                try {
                    res = i.nextResource();
                    System.out.println(res.getContent());
                } finally {
                    //dont forget to cleanup resources
                    try { ((EXistResource)res).freeResources(); } catch(XMLDBExc
eption xe) {xe.printStackTrace();}
                }
            }
        } finally {
            //dont forget to cleanup
            if(col != null) {
                try { col.close(); } catch(XMLDBException xe) {xe.printStackTrac
e();}
            }
        }
    }
}
```

Compiling a Query (XML:DB API)

The XML-RPC server automatically caches compiled expressions, and so calling compile through the remote driver produces no effect if the expression is already cached.

Next, we would like to store a new document into the repository. This is done by creating a new XMLResource, assigning it the content of the new document, and calling the storeResource method of class Collection. First, a new Resource is created by method Collection.createResource(), and expects two parameters: the id and type of resource being created. If the id-parameter is null, a unique resource-id will be automatically generated .

In some cases, the collection may not yet exist, and so we must create it. To create a new collection, call the createCollection method of the CollectionManagementService service. In the following example, we simply start at the root-collection object to get the CollectionManagementService service.

```
import java.io.File;
import org.exist.xmldb.EXistResource;
import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;
public class StoreExample {

    private static String URI = "xmldb:exist://localhost:8080/exist/xmlrpc";

    /**
     * args[0] Should be the name of the collection to access
     * args[1] Should be the name of the file to read and store in the collectio
n
     */
    public static void main(String args[]) throws Exception {

        if(args.length < 2) {
            System.out.println("usage: StoreExample collection-path document");
            System.exit(1);
        }
        final String driver = "org.exist.xmldb.DatabaseImpl";

        // initialize database driver
        Class cl = Class.forName(driver);
        Database database = (Database) cl.newInstance();
        database.setProperty("create-database", "true");
        DatabaseManager.registerDatabase(database);
        Collection col = null;
        XMLResource res = null;
        try {
            col = getOrCreateCollection(args[0]);

            // create new XMLResource; an id will be assigned to the new
resource
            res = (XMLResource)col.createResource(null, "XMLResource");
            File f = new File(args[1]);
            if(!f.canRead()) {
                System.out.println("cannot read file " + args[1]);
                return;
            }

            res.setContent(f);
            System.out.print("storing document " + res.getId() + "...");
            col.storeResource(res);
            System.out.println("ok.");
        } finally {
            //dont forget to cleanup
            if(res != null) {
                try { ((EXistResource)res).freeResources(); } catch(XMLDBExcepti
on xe) {xe.printStackTrace();}
            }

            if(col != null) {
                try { col.close(); } catch(XMLDBException xe) {xe.printStackTrac
e();}
            }
        }
    }

    private static Collection getOrCreateCollection(String collectionUri) throws
XMLDBException {
        return getOrCreateCollection(collectionUri, 0);
    }

    private static Collection getOrCreateCollection(String collectionUri, int
pathSegmentOffset) throws XMLDBException {

        Collection col = DatabaseManager.getCollection(URI + collectionUri);
        if(col == null) {
            if(collectionUri.startsWith("/")) {
                collectionUri = collectionUri.substring(1);
            }
            String pathSegments[] = collectionUri.split("/");
            if(pathSegments.length > 0) {
                StringBuilder path = new StringBuilder();
                for(int i = 0; i <= pathSegmentOffset; i++) {
                    path.append("/" + pathSegments[i]);
                }
                Collection start = DatabaseManager.getCollection(URI + path);
                if(start == null) {
                    //collection does not exist, so create
                    String parentPath = path.substring(0, path.lastIndexOf("/"
));
                    Collection parent = DatabaseManager.getCollection(URI +
parentPath);
                    CollectionManagementService mgt = (CollectionManagementServi
ce) parent.getService("CollectionManagementService", "1.0");
                    col = mgt.createCollection(pathSegments[pathSegmentOffset]);
                    col.close();
                    parent.close();
                } else {
                    start.close();
                }
            }
            return getOrCreateCollection(collectionUri, ++pathSegmentOffset);
        } else {
            return col;
        }
    }
}
```

Adding a File (XML:DB API)

Please note that the XMLResource.setContent() method takes a Java object as its parameter. The eXist driver checks if the object is a File. Otherwise, the object is transformed into a String by calling the object's toString() method. Passing a File has one big advantage: If the database is running in the embedded mode, the file will be directly passed to the indexer. Thus, the file's content does not have to be loaded into the main memory. This is handy if your files are very large.

Extensions to XML:DB

Additional Services

eXist provides several services in addition to those defined by the XML:DB specification:

The UserManagementService service contains methods to manage users and handle permissions. These methods resemble common Unix commands such as chown or chmod. As with other services, UserManagementService can be retrieved from a collection object, as in:

```
UserManagementService service =
(UserManagementService)collection.getService("UserManagementService", "1.0");
```

Another service called DatabaseInstanceManager, provides a single method to shut down the database instance accessed by the driver. You have to be a member of the dba user group to use this method or an exception will be thrown. See the Deployment Guide for an example.

Finally, interface IndexQueryService supports access to the terms and elements contained in eXist's internal index. Method getIndexedElements() returns a list of element occurrences for the current collection. For each occurring element, the element's name and a frequency count is returned.

Method scanIndexTerms() allows for a retrieval of the list of occurring words for the current collection. This might be useful, for example, to provide users a list of searchable terms together with their frequency.

Multiple Database Instances

As explained above, passing a local XML:DB URI to the DatabaseManager means that the driver will try to start or access an embedded database instance. You can configure more than one database instance by setting the location of the central configuration file. The configuration file is set through the configuration property of the DatabaseImpl driver class. If you would like to use different drivers for different database instances, specify a name for the created instance through the database-id property. You may later use this name in the URI to refer to a database instance. The following fragment sets up two instances:

```
// initialize driver
String driver = "org.exist.xmldb.DatabaseImpl";
Class cl = Class.forName(driver);
Database database1 = (Database)cl.newInstance();
database1.setProperty("create-database", "true");
database1.setProperty("configuration", "/home/exist/test/conf.xml");
database1.setProperty("database-id", "test");
DatabaseManager.registerDatabase(database1);
Database database2 = (Database)cl.newInstance();
database2.setProperty("create-database", "true");
database2.setProperty("configuration", "/home/exist/production/conf.xml");
database2.setProperty("database-id", "exist");
DatabaseManager.registerDatabase(database1);
```

Multiple Database Instances

With the above example, the URI

```
xmldb:test:///db
```

selects the test database instance. Both instances should have their own data and log directory as specified in the configuration files.