XQuery Java API (tutorial)

Qizx/open 1.1

Table of Contents

Warning

In the next version of Qizx/db, this API will be unified with the XML Library API, and many interface, class and method names will change.

We recommend therefore to avoid significant developments using this API.

1. Overview

This API provides the methods for compiling and executing XQuery/XPath scripts from Java and exploiting the results. This is very similar to using the SQL language through a Java interface like JDBC.

Using XQuest's implementation of XQuery provides a high-level query language and extended processing capabilities. It is therefore advisable to implement the largest possible part of an application with XQuery, and use Java only for retrieving final results. This is especially true when connecting to a remote server, because such an approach will likely minimize the network traffic.

The API is actually used in the GUI and Command-line Interface applications provided with XQuest, as well as in the "Server Pages" extension, which embeds the XQuery engine in a Servlet.

The API allows:

- Creation and setup of a compilation and execution environment (which implements both the *static context* and *dynamic context* defined in the XQuery Specifications).

  These environments are basically provided by an interface named `XQueryConnection`.

- Creation of *Expressions* from a XQueryConnection (interfaces `XQueryExpression` and `XQueryPreparedExpression`). Such expressions are similar to *Statements* found in database connectivity interfaces like JDBC.

  Expressions receive a Query (i.e. a XQuery script) which is compiled and then executed once or several times.

An Expression is itself a context which inherits the environment provided by the connection, and can then be set up individually before execution. In particular, it is possible to bind global variables of the XQuery expressions to initial values.

- Exploitation of results of Expression evaluations.

  The result sets are iterators, since the XQuery/XPath2 language can generally return sequences of *Items*.

  Items are either atomic values (such as numbers or strings), or *Nodes*, which describe the structure of XML documents and data. The properties and relationships of Nodes are described in a section of the XQuery/XPath2/XSLT2 specifications which is called the *Data Model*. A section of the present document is dedicated to the Data Model interfaces, which handle Nodes.

For calling Java methods from within XQuery expressions, see the *Java Binding* section of the *XQuery Extensions* documentation.

For dealing specifically with XML databases (*XML Libraries*), see the *XML Library API*.

1.1. Relationship with XQJ (XQuery API for Java)

The **XQuery API for Java** (JSR 225) is a specification in progress in the Java Community Process (JCP). Simply speaking, XQJ is to XQuery what JDBC is to SQL. Therefore the API described in the present document and XQJ have a very similar purpose.

Since XQJ is still a draft (as of Fall 2006), compatibility with this specification is not the question of the moment. XQuest will definitely support XQJ when the actual specifications will be published.

The API described in the present document is conceptually close to XQJ, with most interface and class names identical or similar to those of XQJ. However, XQJ deals only with queries (read-only access) while this API also provides means for managing databases (XML Libraries) and perform updates.

This API will be kept as stable as possible. Compatibility with XQJ will be provided through wrapper classes.

2. Packages and classes

2.1. Packages

To use the API, interfaces and classes from the following packages have to be imported:

Table 1. packages

`net.axyana.qizxopen.xquery`	This is the root package for XQuery, it contains in particular `XQueryConnection`, `XQueryExpression`, `XQValue`, `XQItem`, `XQType`.
`net.axyana.qizxopen.xquery.dm`	(XQuery Data Model) Can be used for lower-level operations: contains principally the XQuery `XQNode` interface.
`net.axyana.qizxopen.dm`	Data Model, independent of XQuery: contains support for serialization (`XMLSerializer`) and a super-interface `Node`.
`net.axyana.qizxopen.util`	Utilities.

2.2. Overview of Interfaces and Classes

XQueryConnection [interface]

This is the fundamental interface for applications. A `XQueryConnection` provides an environment for creating Expressions, which can then be executed. Notice that the word *Connection* does not necessarily imply a remote connection through a network. XQuest currently uses the RMI technology, so the application and the "server" can work in a distributed object environment, but of course the client code can also run in the same Java Virtual Machine as the XQuery engine.

A `XQueryConnection` is obtained from a `XQueryServer`, or from `XQueryDataSource`, which is an abstract connection factory.

Each application should have its own XQueryConnection. Access to the same connection by several threads must be explicitly synchronized.

XQueryServer [interface]
Abstract view of a XQuery Engine (embedded or remotely accessible). It is mainly a provider of connections through a method named getConnection() which can specify a particular XML Library (database).

XQueryBasicEngine [class]
An implementation of the XQuery engine (XQueryServer interface). This is a controller centralizing the management of resources (memory, XML Libraries, compiled XQuery modules, cache of parsed XML documents). The configuration interface belongs to the server side and is not visible to applications, which see this object as a XQueryServer only.

Configuring and starting a XQueryEngine is dependent on the context in which XQuest is used (standalone, J2EE). This topic is explained in a separate section below.

XQueryExpression
A XQueryExpression is the equivalent of a JDBC/SQL Statement. It is created from a Connection and receives a XQuery script.

Its purpose is to execute simple scripts. However values can be bound with XQuery variables, like for XQueryPreparedExpression. It can be reused for several different scripts.

XQueryPreparedExpression
A Prepared Expression is a particular case of XQueryExpression, used to execute repetitively the same XQuery script with different variable settings. Global XQuery variables can be assigned a value through this interface.

XQResultSequence
This is the result of the execution of an Expression. A XQuery expression returns a value which is a sequence of *items* (`XQItem`). An Item is a simple "atomic" value (string, number etc.) or a XML Node (a node of the XML Data Model).

XQResultSequence appears as an iterator which enumerates the items of the sequence. It provides methods to obtain and test the type and value of each item.

XQItem
This is an abstract interface which provides methods to obtain and test the type and value of the item.

XQType and XQItemType
A representation of XQuery types. XQType is the most general: it can describe *Item types* (XQItemType) or *sequence types*. The type of a XQValue is generally XQType, while the type of a XQItem is always a XQItemType.

XQNode
A specialization of XQItem representing XML nodes. It provides access to the XML Data Model. Data Model interfaces are presented in a separate section below.

3. Using the API

This section is an introduction to the use of the API, in the form of a tutorial. Reference material is available as Java documentation (Javadoc).

Warning(1): this section of documentation is currently not sufficiently developed. Developers who really want to use the API should look at the source code of applications: command-line tool (class XQuestCLI) and GUI tool (class XQuestStudio). The license allows you reusing this code in your applications.

Warning(2): this API is likely to change significantly in the forthcoming Beta version. We recommend avoiding large developments using this API.

3.1. Outline

An XQuest application typically performs the following steps:

1. Obtain a XQueryConnection. This is achieved by using the `getConnection` method on a XQueryServer or on a XQueryDataSource.

2. Optionally define settings on the connection. Such settings will be inherited by all Expressions created from this connection. This includes arbitrary named properties, predefined namespaces, collations, global variable values, default XML input (document or collection).

3. Create and execute application-specific Expressions (XQueryExpression or XQueryPreparedExpression). Before executing an expression, it is possible to redefine the settings mentioned above, specifically for the expression. In particular, initial values can be bound to global variables of the expression.

4. Executing an Expression (methods `executeQuery`) can be performed in different ways.

    - The simplest way is to serialize directly the result into XML text (the result must be a well-formed document or a single Node). Serialization options can be specified, in particular a generation method (XML, HTML, XHTML markup, or plain text). Other options are specified in classes XMLSerializer or XMLSerialOptions. See the setOption method of these classes for more details.

    - The most general way is to obtain the results as a `XQResultSequence`, and enumerate the result items. Items can be XML Nodes or atomic values (like string, double, boolean etc.). In the most general case, it is necessary to check the types of items and extract values appropriately through a set of specialized methods.

      A XQResultSequence can also be bound to a global variable of another XQuery Expression.

    - There are also more efficient methods that can be used only in "local" mode (that is when the client application and the XQuery engine run in the same Java Virtual Machine).

3.2. More detailed workflow

1. Obtain a XQueryConnection. For simplicity, we assume we already have a running XQuery engine (XQueryBasicEngine), from which we create a connection:

    ```
    XQueryBasicEngine engine;
    ...
    XQueryConnection connection = engine.getConnection();
    ```

    In practice, it can be a bit more complex: if the engine accesses XML Libraries and these libraries have access control enabled, it is necessary to pass to getConnection() properties that contain login names and passwords for the different Libraries accessed. See the Javadoc for more details.

    Note: the setup of an XQueryBasicEngine is not described here. Please see the Javadoc and the source code of applications.

2. Setting static options: there are quite a few possible settings:

    - Predefine a namespace (prefix + URI) that is visible by compiled queries (method predefineNameSpace).

      ```
      connection.predefineNameSpace("myns", "my.uri");
      ```

      This allows the use of the myns: prefix to designate the namespace, without declaring it explicitly in queries.

- Predefine a global variable visible by compiled queries (method predefineGlobal): for example the command line application predefines a variable $arguments of type xs:string* that collects the options passed on the command line.

  ```
  connection.predefineGlobal("arguments", Type.STRING.star);
  ```

  Note: this should be used for variables

- Register a collation, define the default collation.
- Define or redefine the ModuleManager: this can be useful if a different implementation is used.
- Define or redefine the DocumentManager: this can be useful if a different implementation is used.
- Explicitly authorize Java classes to be used by the Java binding mechanism: this is a security feature.

3. Compile a Query:

   there are different variants of the method XQueryConnection.compileQuery. Basically it needs a piece of text (a CharSequence, i.e. typically a String) which can also be read from a stream or a File.

   An URI must be specified for use by error message and traces. For a file or URL input this would typically be the string value of the path or the URL.

   ```
   String querySource =
             " for $i in 1 to 3 return element E { attribute A { $i } } ";
   try {
      XQuery query = connection.compileQuery(querySource, "<source>", log);
      ...
   } catch( XQueryException e) {
      ...
   }
   ```

   Exceptions can be raised on a syntax error (prevents further compilation) or by static analysis errors (at end of compilation).

4. Setting run-time options:

   Typically, global variables (declared *external* in queries) can be initialized here. Initial values specified in queries can also be overridden. The method initGlobal has different variants, according to the value passed. An exception is raised if the value does not match the declared type.

   Initial values are part of the execution environment and do not affect compiled Queries which can be shared by several threads.

   Other options: default output for function x:serialize, node or node sequence used for XQuery function input(), implicit timezone, and message log.

5. Executing a compiled query:

   There are several ways to obtain results:

   a. **Direct serialization** (the simplest):

      ```
      XMLSerializer serial = new XMLSerializer();
      serial.setOutput( new FileWriter("out.xml") );
      serial.setOption("method", "xhtml");
      serial.setOption("indent", "yes");
       // ... other options can be set on the serializer...
      connection.executeQuery( query, serial );
      ```

   b. **Tree building**: returns a Node that can be used in further processing.

```
EventDrivenBuilder builder = new EventDrivenBuilder();
connection.executeQuery( query, builder );
Node result = builder.harvest();
```

c. **SAX output**: it is possible this way for example to pipe a XQuery execution with a XSLT transformation.

The class SAXXQueryConnection implements the interface org.xml.sax.XMLReader and can therefore be used to build a SAXSource for use with APIs javax.xml.transform. (See the javadoc for more details).

d. Get a result sequence and **enumerate Items**:

```
XQValue v = connection.executeQuery( query );
while(v.next()) // When next() returns true, an item is available
{
    if(v.isNode()) {
        XQNode n = v.getNode();
        ... // use the XQNode (Data Model) interface to navigate in the
            // subtree, extract element names, attributes, string values...
    }
    else {
        ItemType type = v.getType();  // type of current item
        if (type == Type.DOUBLE) {
            double d = v.getDouble();
        }
        ... // use the different getX() methods, according to the type
    }
}
```

This approach requires a good knowledge of the API. See the next section "Data Model" for an introduction to Node manipulation.

6. Handle errors: execution can raise an EvalException. The message of the exception gives the reason for the error. It is also possible to display the call trace:

```
try {
    Value v = connection.executeQuery( query );
    ...
} catch (EvalException ee) {
    ee.printStack(log, 20);
}
```

The stack trace is printed to a Log object. The second argument gives a depth maximum for the trace (0 means no maximum).

4. Data Model interfaces

This section describes the Java interfaces to the XML/XQuery *Data Model*. The Data Model is defined by a W3C specification: http://www.w3.org/TR/xpath-datamodel/. It is an extension of the XML Infoset which describes precisely the abstract objects (their contents, possible values, and relationship) which constitute XML Documents handled by XPath 2, XQuery and XSLT 2.

This Data Model differs from the W3C DOM in the following respects:

- It supports XML Schema types and the notion of collections.
- It does not keep track of physical features like entity boundaries, marked sections, characters references.
- It does not define updating operations.
- No language bindings are specified.

In XQuest the XML Data Model is seen mainly through the **`Node`** interface (`net.axyana.xquest.dm.Node`). It supports the *accessors* defined in the Data Model specifications plus extensions. See the XML Library Java API for more details.

There is also a XQuery version of Node, which is XQNode: it provides both the Node and the XQItem interfaces.

The `net.axyana.xquest.dm` package also contains a few related interfaces or classes, like `NodeSequence`, `NodeTest`, and service classes like `XMLSerializer` and `FulltextQuery`.

The utility package `net.axyana.xquest.util` contains ancillary classes for handling qualified names (QName and Namespace).

What follows is a short primer. For detailed information, refer to the Java Documentation.

4.1. Basic accessors

These accessors return basic properties of a Node:

`String getNodekind()`
: represents the accessor dm:node-kind() which returns string values like "document", "element", "attribute" etc.

`int getNature()`
: returns the node kinds as integer values, more convenient for programming, like DOCUMENT, ELEMENT, ATTRIBUTE, TEXT, COMMENT, PROCESSING_INSTRUCTION, and NAMESPACE (all constant fields of the Node interface).

`QName getNodeName()`
: represents the accessor dm:node-name() which returns a qualified name if applicable (elements, attributes) or the null value.

`Node parent()`
: Returns the parent node or null.

`String getStringValue()`
: Returns the textual contents of the node, as defined in the DM specifications (The string value of an element is the concatenation of all text fragments encompassed by the element).

`NodeSequence children()`
: For documents and elements, returns a *NodeSequence*, an abstract iterator which can enumerate the children nodes in document order. To iterate on children, the following code pattern is typically used:

```
NodeSequence children = node.children();
while(children.next()) {
    Node child = children.currentNode();
    //...
}
```

For other node kinds, the sequence is always empty.

`NodeSequence attributes()`
: This method returns the sequence of attribute nodes belonging to an element. Example: a crude serialization of an element:

```
if( node.getNature() == Node.ELEMENT ) {
   output.print("<");
   // print element name: needs to convert QName to string
   output.printName(node.getNodeName());

   NodeSequence attributes = node.attributes();
   for( ; attributes.next(); ) {
       Node attr = attributes.next.currentNode();
       output.print(" ");
```

```
        // print attribute name: needs to convert QName to string
        output.printName(attr.getNodeName());
        output.print("='");
        // print attribute value (needs escaping)
        output.printName(attr.getStringValue());
        output.print('");
    }
    output.print(">");
}
```

4.2. Extended accessors

These extended methods return sequences filtered by an abstract `NodeTest`.

BaseNodeTest is a useful implementation of `NodeTest` which can filter nodes according to their kind and their name. It can also perform wildcard name matching. It has convenience subclasses ElementTest and AttributeTest.

`NodeSequence children( NodeTest test )`
Returns the sequence of children which pass the test. For example, this code returns an iterator on children which have the name "section", with a blank namespace:

```
node.children( new ElementTest("section") )
```

which can also be written less simply as:

```
node.children( new BaseNodeTest( Node.ELEMENT,
                                 Namespace.NONE, "section") )
```

`NodeSequence attributes( NodeTest test )`
Returns the sequence of attributes which pass the test. For example, this code returns an iterator on *all* attributes which have a name with namespace **ns**:

```
node.children( new AttributeTest( ns, null ) )
```

`NodeSequence ancestors( NodeTest test )`

`NodeSequence ancestorsOrSelf( NodeTest test )`

`NodeSequence descendants( NodeTest test )`

`NodeSequence descendantsOrSelf( NodeTest test )`

`NodeSequence followingSiblings( NodeTest test )`

`NodeSequence following( NodeTest test )`

`NodeSequence precedingSiblings( NodeTest test )`

`NodeSequence preceding( NodeTest test )`
Similar filtered iterators which implement XPath *axes* like ancestor, descendant etc.

4.3. Extended XQuery accessors

This is the XQuery version of the preceding accessors.

The XQuery node (net.axyana.xquest.xquery.dm.XQNode) has similar but slightly different methods which also return sequences filtered by an abstract `NodeTest`. The returned sequence is of type XQValue, the general XQuery result sequence.

`XQValue getChildren( NodeTest test )`
is equivalent to children(test)

`XQValue getAttributes( NodeTest test )`
etc.

4.4. Node Comparisons

These methods compare the value or the document order of two nodes:

`int orderCompare( Node otherNode )`
Returns -1 if this node is strictly before the other node in the document order, 0 if nodes are identical, or 1 if after the argument node.

This method is generally very efficient.

`int compareStringValues(Node node, java.text.Collator collator)`
compares the string values of two nodes, whatever their kinds, with an optional Collator.

4.5. Parsing

To obtain a Node from a document residing in a file or accessible through an URL, one can use the services of a DocumentParser or a DocumentManager.

- DocumentParser provides basic parsing and tree construction services. It supports XML catalogs.
- DocumentManager is an extension of DocumentParser which supports URI resolution, and caching (so that a document accessed several times does not need to be reparsed). It can be used concurrently by several threads.

The simplest way of parsing a document given its URI (system Identifier in SAX terminology) is to use a static method of DocumentParser:

```
Node root = DocumentParser.parse(new InputSource(uri));
```

To use document caching, a DocumentManager has to be instantiated, then its findDocumentNode method can be used to get the root node of the document from its URI.

Please note that these methods build a different implementation of the Data Model, which is very memory- and processor-efficient, but immutable.

4.6. Serialization

`XMLSerializer` is a class which supports all serialization tasks. It converts any node into a serialized form in XML, XHTML or HTML (if applicable) or plain text (discarding the tags).

After creating a XMLSerializer, options can be set, in particular an output stream:

```
XMLSerializer serial = new XMLSerializer("HTML");
FileOutputStream outputStream = new FileOutputStream("out.html");
serial.setOutput(outputStream, "ISO8859_1");
serial.setOption(XMLSerializer.OMIT_XML_DECLARATION, "yes");
serial.setOption(XMLSerializer.INDENT, "no");
```

A node can be serialized this way:

```
serial.output(node);
```

A Serializer can be reused. The XML or DOCTYPE declarations are output only if the node is a document node. It is also possible to control this at a lower level by using methods reset, terminate, startDocument, endDocument.

Serialization options are described in the Java documentation and in the User's Guide.

4.7. Exporting as trees

Instead of serializing XML results, it is possible to export these results as trees in different implementations.

The first version of Qizx/open supports W3C DOM trees and internal Core Trees, but in fact since the mechanism relies on a general interface (XMLEventHandler), it is quite possible to extend the family of converters to support other XML tree packages, like JDOM or DOM4J. This could be done in future versions, but contributions are welcome.

DOM trees

DOMEventBuilder can build W3C DOM trees.

For example (assuming expr is a XQueryExpression associated with a valid connection) this code snippet would build a DOM tree representing the XML fragment `<E>1 2 3</E>`:

```
DOMEventBuilder builder = new DOMEventBuilder();
expr.executeQuery("element E { 1 to 3 }", builder);
Node result = builder.harvest();
```

The created tree can be retrieved with the method harvest(). The builder can be reused by calling the reset() method.

Core Trees:

Another class CoreEventBuilder, very similar to DOMEventBuilder, can build trees of the internal "Core" Data Model. This is mainly for internal usage.

General interface XMLEventHandler

XMLSerializer, DOMEventBuilder and CoreEventBuilder are in fact particular implementations of a general interface called XMLEventHandler, which is conceptually similar to SAX2.

Like SAX, this interface handles events like StartDocument, StartElement, Attribute, EndElement, Text, Comment... EndDocument, and incrementally builds another representation.

There is a convenience transformation method which traverses any node and its subtree:

```
XMLEventHandler.traverse(node);
```

Though it may not look very intuitive, using this interface can be quite powerful for transforming a tree - or generating large trees with small memory footprint-, by combining export with explicit construction or tree traversal.

```
QName DOC = QName.get("doc");
QName WRAP = QName.get("wrap");
XMLEventHandler eh = new DOMEventBuilder();
eh.evStartElement(DOC);
eh.evText("some text");
eh.evAttribute(QName.get("id"), "x0001");
// wrap the result of a query evaluation in a "wrapper" element:
eh.evStartElement(WRAP);
// include the query result:
expr.executeQuery(eh);
eh.evEndElement(WRAP);
eh.evComment(" a comment ");
eh.evEndElement(DOC);
XQNode result = eh.harvest();
```

This snippet would create the following DOM tree (the fragment `<result>...</result>` stands for whatever is returned by the query execution):

```
<doc id="x0001">some text<wrap>
<result>...</result>
</wrap><!-- a comment --></doc>
```