Tutorial: XQuery API for Java (XQJ) - Introduction

y Process as JSR 225, lets programmers use XQuery for XML processing and data integration applications, with full support for the Java SE and Java EE platforms. XQJ allows a Java program to connect to XML data sources, prepare and issue XQueries, and process the results as XML. This functionality is similar to that of JDBC Java API for SQL, but the query language for XQJ is XQuery.

Let's start with a simple XQJ application — the XQuery/XQJ version of "Hello World!":

```
XQDataSource xqjd = new DDXQDataSource();
XQConnection xqjc = xqjd.getConnection();
XQExpression xqje = xqjc.createExpression();
XQSequence xqjs = xqje.executeQuery("'Hello World!'");
xqjs.writeSequence(System.out, null);
xqjc.close();
```

You might recognize some of the JDBC concepts in the code example above. Indeed, the XQJ expert group decided to make XQJ stylistically consistent with JDBC. As a result, XQJ has JDBC-like concepts such as:

- Datasources
- Connections
- Expressions and prepared expressions

But at the same time, XQuery is not SQL, and as such XQJ differs from JDBC in some areas, including:

- Data model
- Typing
- Static context
- Error handling

We'll touch on each of these concepts throughout the remaining chapters in this tutorial. So, let's get started!

Setting up an XQuery API for Java (XQJ) Session

XQJ is designed to accommodate the differences that can exist across XQuery implementations in terms of architecture and the range of supported data sources. For example, depending on the XQJ and XQuery implementation, the parameters and settings needed to configure the implementation might be different. Some implementations are server based and require information to locate the server, for example. Or, if the XQJ and XQuery implementations are co-located, you might need to specify the default location to query files on the local file system. Let's take a closer look at how connections work in XQJ.

The XQDataSource Object

An XQJ application always starts with accessing an XQDataSource object. Such an object encapsulates all the parameters and settings needed to create a session with a specific implementation, and eventually execute XQuery expressions and process the results.

Every XQJ driver has its own XQDataSource implementation. This XQDataSource object supports a number of implementation-specific properties. For each of these properties, "getter" and "setter" methods are provided.

Assume an application wants to query both an Oracle database and some files located in "/usr/joe/data" using the DataDirect XQuery® engine. Two properties need to be specified:

- The *base URI*, which is used to retrieve the files in our folder without explicitly referencing the folder in the XQuery
- , *and*
- The *JDBC URL*, which is used to connect to the Oracle database (feel free to replace Oracle with your favorite database, be it SQL Server, MySQL or any other)

Here's what the DataDirect XQuery implementation of the XQDataSource object, DDXQDataSource, needed to specify these settings would look like:

```
DDXQDataSource xqds = new DDXQDataSource();
xqds.setBaseUri("/usr/joe/data");
xqds.setJdbcUrl("jdbc:xquery:oracle://sales:1521;SID=ORA10");
```

Now think about a different XQuery implementation, perhaps one provided by Oracle, in which the server parameters are specified as individual properties rather than through a JDBC URL. It could be expressed as something like this:

```
OracleDataSource xqds = new OracleDataSource();
xqds.setServerName("sales");
xqds.setPortNumber(1521);
xqds.setSID("ORA10");
```

Establishing a Session

Once we have access to an XQDataSource object, what's next? An XQDataSource is a *factory* for XQConnection objects. The XQConnection object represents a session in which XQuery expressions are executed. Establishing such a session is straightforward:

```
XQConnection xqc = xqds.getConnection();
```

If required, user credentials can be specified as arguments to the getConnection() method:

```
XQConnection xqc = xqds.getConnection("joe", "topsecret");
```

Making Applications Independent

You might have noticed that using the approach outlined so far to create XQDataSource objects makes the application dependent on a specific XQJ implementation — the proprietary classes DDXQDataSource (DataDirect XQuery®) and OracleDataSource, for example.

This is not necessarily wrong — there are scenarios in which hard-coding the underlying XQJ implementation makes sense — but it's not always desirable. After all, one of the benefits of XQJ is the ability to make your applications independent from the underlying XQuery implementation. How can we achieve that? We'll show two approaches:

- Using a Java properties files
- Through the Java Naming and Directory Interface (JNDI)

Using Properties Files

Assume all the XQDataSource properties are stored in a Java properties file, and in addition a property *ClassName* is specified to identify the XQDataSource implementation that needs to be used.

For the DataDirect XQuery® example above, the properties file would look like this:

```
ClassName = com.ddtek.xquery3.xqj.DDXQDataSource
BaseUri = /usr/joe/data
JdbcUrl = jdbc:xquery:oracle://sales:1521;SID=ORA10
```

For the Oracle implementation:

```
ClassName = org.example.xqj.OracleDataSource
ServerName = sales
PortNumber = 1521
SID = ORA10
```

Using such property files, an application can easily abstract out any hard-coded dependencies on the underlying XQJ implementation. The XQDataSource class is loaded through reflection and the next step is to simply provide the property file:

```
// load the property file
Properties p = new Properties();
p.load(new FileInputStream("/tmp/xqjds.prop"));

// create an XQDataSource instance using reflection
String xqdsClassName = properties.getProperty("ClassName");
Class xqdsClass = Class.forName(xqdsClassName);
XQDataSource xqds = (XQDataSource)xqdsClass.newInstance();

// remove the ClassName property
// the XQJ implementation will not recognize it and raise an error
p.remove("ClassName");

// set the remaining properties
xqds.setProperties(tmpProperties);

// create an XQConnection
XQConnection xqc = xqds.getConnection();
```

Of course, this is just an example; it might well be that for some applications different ways to load the datasource properties are better suited.

Using JNDI

Similarly to JDBC, the XQDataSource object can be stored in a JNDI-enabled naming service when running in a Java EE environment. This allows applications to access the XQDataSource by simply specifying a logical name:

```
// get the initial JNDI context
Context ctx = new InitialContext();
// load the XQDataSource instance
XQDataSource xqds = (XQDataSource)ctx.lookup("xqj/sales");
// create an XQConnection
XQConnection xqc = xqds.getConnection();
```

If you have a JDBC background, you are probably familiar with the fact that JDBC has two mechanisms to establish a connection:

- DriverManager
- DataSource

Don't look in XQJ for DriverManager-like functionality; XQJ doesn't offer this legacy functionality.

At this point you know how to create an XQConnection. Now you're ready to do some "real work" and execute queries.

Querying Data from XML Files or Java XML APIs with XQJ

In the previous chapter, XQJ Tutorial Part II: Setting up an XQJ Session, we learned how to create a connection. Now your application is ready to do some real work — executing queries.

The XQExpression Object

In XQJ, XQExpression objects allow you to execute XQuery expressions. XQExpression objects are created in the context of an XQConnection. The example described on this page creates an XQExpression and uses it to execute an XQuery expression. Let's take a look:

```
...
// assume an XQConnection xqc
XQExpression xqe = xqc.createExpression();
xqe.executeQuery("doc('orders.xml')//order[id='174']");
...
```

The result of an XQuery evaluation is a sequence, which is modeled as an XQSequence object in XQJ. Hence, the result of the executeQuery() method is an XQSequence. In typical scenarios the code example of above will actually look like this:

```
...
XQExpression xqe;
XQSequence xqs;
xqe = xqc.createExpression();
xqs = xqe.executeQuery("doc('orders.xml')//order[id='174']");
// process the query results
...
```

You can find details about the XQSequence functionality in XQJ Tutorial Part IV: Processing Query Results.

Reusing XQExpression Objects

An XQExpression object can be reused — a different XQuery expression can be executed each time. The next example first retrieves all orders with id 174, and next all orders with id 267:

```
...
XQExpression xqe;
XQSequence xqs;
xqe = xqc.createExpression();
// execute a first query
xqs = xqe.executeQuery("doc('orders.xml')//order[id='174']");
// process the query results
...
// execute a second query
xqs = xqe.executeQuery("doc('orders.xml')//order[id='267']");
// process the query results
...
```

Prepared Queries

XQJ supports the concept of *prepared* queries. The underlying rationale is to “prepare” (“compile”, if you prefer) the query only once, and subsequently “execute” it several times. During the preparation phase, the query is parsed, statically validated, and an optimized execution plan is generated. This sequence of operations can be a relatively expensive; using XQPreparedExpression

objects can improve performance if the same query is executed multiple times.

Using prepared queries often implies the use of external variables in your query. The application can bind different values to each of the external variables for different executions. Here is an example that shows how the previous code snippet can be modified to take advantage of an XQPreparedExpression object. Note that the XQuery expression is specified when the XQPreparedExpression object is created, not at execution time:

```
...
XQPreparedExpression xqp;
XQSequence xqs;
xqp = xqc.prepareExpression(
            "declare variable $id as xs:string external; " +
            "doc('orders.xml')//order[id=$id]");
// execute a first query and process the query results
xqp.bindString(new QName("id"),"174", null);
xqs = xqp.executeQuery();
...
// execute a second query and process the query results
xqp.bindString(new QName("id"), "267", null);
xqs = xqp.executeQuery();
...
```

Context Items

XQuery has a concept of *context item*. The context item is represented by a "dot" in your queries. The previous examples show how values can be bound to XQExpression and XQPreparedExpression objects by name. In XQuery, the context item has no name; as such, XQJ defines a constant to bind the context item: XQConstants.CONTEXT_ITEM. The next example is similar to the previous one, but it binds the DOM document to the initial context item, rather than to an external variable:

```
...
Document domDocument = ...;
XQExpression xqe = xqc.createExpression();
xqe.bindNode(XQConstants.CONTEXT_ITEM, domDocument, null);
XQSequence xqs = xqe.executeQuery(".//order[id='174']");
// process the query results
...
```

Later in this tutorial, you'll see how to bind values to external variables or the context item in greater detail.

Using Input Streams

Note that in all the above examples, the XQuery expressions are specified as Java character strings. XQJ also allows specifying an InputStream, as shown in the next example, where the query in the getorders.xquery file is executed:

```
...
InputStream query;
query = new FileInputStream("home/joe/getorders.xquery")
XQExpression xqe;
XQSequence xqs;
xqe = xqc.createExpression();
xqs = xqe.executeQuery(query);
// process the query results
...
```

Of interest here is that an XQuery can optionally specify the encoding in the query prolog. Good XQJ/XQuery implementations will use that information and properly parse the InputStream. If no encoding is specified, the assumed encoding depends on the implementation; for DataDirect

XQuery®, this is UTF-8.

Processing Query Results with XQuery API for Java (XQJ)

XQJ Tutorial Part III: Querying Data from XML Files or Java XML APIs describes how to execute queries. In XQuery, query evaluation results in a sequence. In XQJ, executing a query through XQExpression or XQPreparedExpression returns an XQSequence object. An XQSequence represents an XQuery sequence and a cursor to iterate over the sequence.

Applications can navigate through an XQSequence using the next() method. Initially the current position of the XQSequence is before the first item. next() moves the current position forward and returns true if there is another item to be consumed. Once all items in the sequence have been read, next() returns false.

Let's iterate through a sequence:

```
...
XQExpression xqe;
XQSequence xqs;
xqe = xqc.createExpression();
xqs = xqe.executeQuery("doc('orders.xml')//order[id='174']");
while (xqs.next()) {
  ...
}
...
```

Retrieving Data Using Get Methods

Once positioned on an item, applications can retrieve the data using one of the get*XXX*() methods. Let's take a look at some of these methods.

An application can use getObject() to retrieve the current item of an XQSequence as a Java object. XQJ defines a mapping for each of the XQuery item types to a Java object value.

Working with Elements

One of the most common scenarios is probably a query returning a sequence of elements. Using getObject(), XQJ defines a mapping to Java DOM elements:

```
...
org.w3c.dom.Element employee;
XQExpression xqe;
XQSequence xqs;
xqe = xqc.createExpression();
xqs = xqe.executeQuery("doc(employees.xml)//employee");
while (xqs.next()) {
  employee = (org.w3c.dom.Element)xqs.getObject();
  ...
}
...
```

Working with Atomic Values

Actually, XQJ defines a mapping for every XQuery type to Java objects, including all the atomic types. Assume, for example, a query returning xs:decimal values; using getObject() your Java application retrieves the items as java.math.BigDecimal objects:

```
...
java.math.BigDecimal price;
```

```
XQExpression xqe;
XQSequence xqs;
xqe = xqc.createExpression();
xqs = xqe.executeQuery(
  "doc('orders.xml')/orders/order/xs:decimal(total_price)");
while (xqs.next()) {
  price = (java.math.BigDecimal)xqs.getObject();
  ...
}
...
```

Suppose you have a query returning atomic values, and you want to retrieve a textual (character string) representation to output to System.out. The getAtomicValue() method returns a character string representation of an atomic value according to the XQuery xs:string casting rules, and throws an exception if the item is not an atomic value (that is, if it's a node).

In the next example the query returns a sequence of atomic values. Note that the items are not all of the same type:

```
...
XQExpression xqe = xqc.createExpression();
XQSequence xqs = xqe.executeQuery(
  "'Hello world!', 123, 1E1, xs:QName('abc')");
while (xqs.next()) {
  System.out.println(xqs.getAtomicValue());
}
...
```

Working with SAX and StAX

Beside the DOM, XQJ also provides native support for two other popular XML APIs — SAX and StAX. In the next example, each of the items is returned to the application through SAX:

```
...
ContentHandler ch = ...
XQExpression xqe = xqc.createExpression();
XQSequence xqs = xqe.executeQuery(
  "doc(employees.xml)//employee");
while (xqs.next()) {
  xqs.writeItemToSAX(ch);
}
...
```

Reading a Sequence as a Stream

So far we have seen a number of examples in which the application iterates over all the items in the sequence, and retrieves them one-by-one. The XQSequence interface also offers functionality to retrieve the complete sequence within a single call. The next example executes a query and serializes the complete result into a SAX event stream.

```
...
ContentHandler ch = ...
XQExpression xqe;
XQSequence xqs;
xqe = xqc.createExpression();
xqs = xqe.executeQuery("doc('employees.xml')//employee");
xqs.writeSequenceToSAX(ch);
...
```

Similarly, the following example reads the complete sequence as a StAX event stream:

```
...
XQExpression xqe = xqc.createExpression();
```

```
XQSequence xqs = xqe.executeQuery("doc('employees.xml')");
XMLStreamReader xmlReader = xqs.getSequenceAsStream();
while (xmlReader.next() != XMLStreamConstants.END_DOCUMENT) {
  ...
}
...
```

Besides exposing the sequence through a SAX or StAX event stream, XQSequence also provides the ability to serialize into a binary or character stream. Here we're entering the arena of XSLT 2.0 and XQuery 1.0 Serialization, which is discussed in XQJ Tutorial Part V: Serializing Query Requests.

Scrollable Sequences

The above examples all iterate forward through the XQSequence objects. XQJ has also the notion of *scrollable sequences*, allowing you to move both forward and backward through a sequence, set the cursor to an absolute position, and iterate through the XQSequence more than once.

Serializing Query Results with XQuery API for Java (XQJ)

The XQuery 1.0 specification consists of multiple books; one is XSLT 2.0 and XQuery 1.0 Serialization. Given a data model instance, the specification defines how to serialize that instance into a sequence of octets. The XQuery 1.0 specification defines a number of parameters that influence the serialization process:

- byte-order-mark
- cdata-section-elements
- doctype-public
- doctype-public
- encoding
- escape-uri-attributes
- include-content-type
- indent
- media-type
- method
- normalization-form
- omit-xml-declaration
- standalone
- undeclare-prefixes
- use-character-maps
- version

You'll learn more about some of these serialization parameters later in this chapter.

Note that serialization is an Optional Feature in XQuery. However, XQJ is stricter and requires that every implementation support serialization. XQJ does not require that every parameter defined in the XQuery Serialization specification be supported to its full extent, but it does require that at least a default value for each of the parameters be documented and behave according to the specification. (For the DataDirect XQuery® implementation, all parameters are documented here.)

Serializing Results to a File

The XQuery 1.0 specification provides guidelines on, among other topics, writing query results using XML syntax into a file (a typical use case for query result processing). Let's use a simple example to illustrate the process of serializing your query results in a file:

```
...
XQExpression xqe;
XQSequence xqs;
xqe = xqc.createExpression();
xqs = xqe.executeQuery(
          "doc('orders.xml')/*/ORDERS[O_ORDERKEY = '39']");
xqs.writeSequence(
          new FileOutputStream("/home/jimmy/result.xml"),
          new Properties());
...
```

Note that the second argument of writeSequence() is an empty Properties object. You can also specify null. Both an empty Properties object and null imply that the XQJ driver uses the default values for each of the serialization parameters.

You might get a result like this (assume this to be one line; we used new lines here for formatting considerations):

```
<ORDERS><O_ORDERKEY>39</O_ORDERKEY><O_CUSTKEY>
8177</O_CUSTKEY><O_ORDERSTATUS>O</O_ORDERSTATUS>
<O_TOTALPRICE>307811.89</O_TOTALPRICE><O_ORDERDATE>
1996-09-20T00:00:00</O_ORDERDATE><O_ORDERPRIORITY>3-MEDIUM
</O_ORDERPRIORITY><O_CLERK>Clerk#000000659</O_CLERK>
<O_SHIPPRIORITY>0</O_SHIPPRIORITY><O_COMMENT>furiously
unusual pinto beans above the furiously ironic asymptot
</O_COMMENT> </ORDERS>
```

Specifying Indenting and Encoding

That's not really readable, is it? Some indentation would help. It's also good practice to add the XML declaration and an encoding. Let's assume we want to encode the XML file as UTF-16:

```
...
Properties serializationProps = new java.util.Properties();
// make sure we output xml
serializationProps.setProperty("method", "xml");
// pretty printing
serializationProps.setProperty("indent", "yes");
// serialize as UTF-16
serializationProps.setProperty("encoding", "UTF-16");
// want an XML declaration
serializationProps.setProperty("omit-xml-declaration", "no");
XQExpression xqe;
XQSequence xqs;
xqe = xqc.createExpression();
xqs = xqe.executeQuery(
         "doc('orders.xml')/*/ORDERS[O_ORDERKEY = '39']");
xqs.writeSequence(
         new FileOutputStream("/home/jimmy/result.xml"),
         serializationProps);
...
```

The result looks much better now:

```
<?xml version="1.0" encoding="UTF-16"?>
<ORDERS>
        <O_ORDERKEY>39</O_ORDERKEY>
        <O_CUSTKEY>8177</O_CUSTKEY>
        <O_ORDERSTATUS>O</O_ORDERSTATUS>
        <O_TOTALPRICE>307811.89</O_TOTALPRICE>
        <O_ORDERDATE>1996-09-20T00:00:00</O_ORDERDATE>
        <O_ORDERPRIORITY>3-MEDIUM</O_ORDERPRIORITY>
        <O_CLERK>Clerk#000000659</O_CLERK>
        <O_SHIPPRIORITY>0</O_SHIPPRIORITY>
        <O_COMMENT>furiously unusual pinto beans above the furiously ironic
asymptot</O_COMMENT>
</ORDERS>
```

Handling Characters That Require Escaping

During serialization, characters are escaped as needed for the specified encoding. Suppose a query returns a document with a registered trademark character (®), with the specified encoding US-ASCII:

```
...
Properties serializationProps = new java.util.Properties();
serializationProps.setProperty("method", "xml");
serializationProps.setProperty("encoding", "ASCII");
```

```
XQExpression xqe;
XQSequence xqs;
xqe = xqc.createExpression();
xqs = xqe.executeQuery(
          "<product>DataDirect XQuery®</product>");
xqs.writeSequence(
          new FileOutputStream("/home/jimmy/result.xml"),
          serializationProps);
...
```

You'll get the following result (note that the ® character is serialized as a character reference because it is not defined in the ASCII character set):

```
<product>DataDirect XQuery&#xae</product>
```

Using CDATA

In some use cases, the cdata-section-elements parameter is useful. Imagine that you're serializing some XML elements including ampersand characters. By default the "&" characters are escaped; using CDATA sections may be preferable to make the XML file more human readable:

```
...
Properties serializationProps = new java.util.Properties();
serializationProps.setProperty("method", "xml");
serializationProps.setProperty("cdata-section-elements", "product");
XQExpression xqe;
XQSequence xqs;
xqe = xqc.createExpression();
xqs = xqe.executeQuery(
          "<product>DataDirect XQuery &amp; XML Converters</product>");
xqs.writeSequence(
          new FileOutputStream("/home/jimmy/result.xml"),
          null);
...
```

The result is serialized as follows:

```
<product><![CDATA[DataDirect XQuery & XML Converters]]></product>
```

Note that multiple elements can be specified through the cdata-section-elements parameter, separating each one with a semi-colon character. And if the element is in a namespace, you can add the namespace URI using the James Clark notation, "{"+namespace URI+"}"localname:

```
...
Properties serializationProps = new java.util.Properties();
serializationProps.setProperty("method", "xml");
serializationProps.setProperty("encoding", "UTF-8");
serializationProps.setProperty("omit-xml-declaration", "no");
serializationProps.setProperty("cdata-section-elements",
                               "product;{uri}product");
XQExpression xqe;
XQSequence xqs;
xqe = xqc.createExpression();
xqs = xqe.executeQuery(
  "<e xmlns:p='uri'> " +
  " <product>DataDirect XQuery &amp; XML Converters</product>" +
  " <p:product>DataDirect XQuery &amp; XML Converters</p:product>" +
  "</e>");
xqs.writeSequence(
          new FileOutputStream("/home/jimmy/result.xml"),
          null);
...
```

The result is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<e xmlns:p="uri">
        <product><![CDATA[DataDirect XQuery & XML Converters]]></product>
        <p:product><![CDATA[DataDirect XQuery & XML Converters]]></p:product>
</e>
```

Specifying HTML and XHTML Output

In addition to the XML output method, the XQuery serialization specification defines other output methods like HTML and XHTML. Note that these serialization methods will not "magically" produce (X)HTML — it is still the query's responsibility to generate results that conform to the (X)HTML specifications. But the serializer will consider the (X)HTML rules outputting the results. For example, when choosing HTML,
 elements will be serialized without a closing </br>.

Note, for example, the difference between the result.xml and result.html for the following code:

```
...
Properties serializationProps = new java.util.Properties();
XQPreparedExpression xqpe = xqc.createPreparedExpression(
         "<html>line1<br/>line2</html>");
XQSequence xqs = xqpe.executeQuery();
serializationProps.setProperty("method", "xml");
xqs.writeSequence(
         new FileOutputStream("/home/jimmy/result.xml"),
         serializationProps);
XQSequence xqs = xqpe.executeQuery();
serializationProps.setProperty("method", "html");
xqs.writeSequence(
         new FileOutputStream("/home/jimmy/result.html"),
         serializationProps);
...
```

result.xml is as follows:

<html>line1
line2</html>

... while result.html looks like this:

<html>line1
line2</html>

Alternatives to Streaming Results

In all previous examples, we've serialized the query results in a FileOutputStream. An XQSequence can also be serialized into a java.io.Writer using the writeSequence() method. And getSequenceAsString() serializes to a java.lang.String.

Similar to serializing the complete XQSequence, there are methods to serialize the current individual item in the XQSequence. In the following example, the items in the query result are saved into distinct files — result1.xml, result2.xml, and so on.

```
...
Properties serializationProps = new java.util.Properties();
serializationProps.setProperty("method", "xml");
serializationProps.setProperty("indent", "yes");
serializationProps.setProperty("encoding", "UTF-8");
serializationProps.setProperty("omit-xml-declaration", "no");
XQExpression xqe;
XQSequence xqs;
xqe = xqc.createExpression();
xqs = xqe.executeQuery("doc('orders.xml')/*/ORDERS");
int i = 1;
while (xqs.next()) {
  FileOutputStream file;
```

```
  file = new FileOutputStream("/home/jimmy/result" +
                              i + ".xml");
  xqs.writeItem(file, serializationProps);
  file.close();
}
...
```

Note that XML serialization doesn't always result in a well-formed XML document. More precisely, it is either a well-formed XML document or a well-formed XML external general parsed entity. See the serialization specification for more information on this topic.

Manipulating the Static Context with XQuery API for Java (XQJ)

This chapter explains how to access and manipulate the static context through the XQJ API.

XQuery defines the Static Context as follows:

The static context of an expression is the information that is available during static analysis of the expression, prior to its evaluation.

The static context includes information like the following (refer to the XQuery specification for the complete list):

- Default element namespace
- Statically known namespaces
- Context item static type
- Default order for empty sequences
- Boundary-space policy
- Base uri

Initializing the Static Context

Most of the components in the static context can be initialized or augmented in the query prolog. In the following example, the boundary-space policy is explicitly specified:

```
declare boundary-space preserve;
<e>  </e>
```

If a static context component is not initialized in the query prolog, an implementation default is used. Indeed, although XQuery defines default values for each of the components in the static context, as outlined in Appendix C of the XQuery specification, implementations are free to override and/or extend those defaults. In theory this means that the same query can behave in substantially different ways between two "conformant" XQuery implementations.

Setting the boundary-space Through the XQJ API

Applications often need to change the defaults for some of the static context components. If preserving boundary spaces in all queries is a requirement, applications have the option to add the boundary-space declaration to the queries, as shown above. But in many cases it is preferable to override the implementation's default through the API, so that the same settings are applied to all queries.

How can an application set boundary-space policy to preserve through the XQJ API?

```
...
// get a static context object with the implementation's defaults
XQStaticContext xqsc = xqc.getStaticContext();
// make sure boundary-space policy is preserve
xqsc.setBoundarySpacePolicy(XQConstants.BOUNDARY_SPACE_PRESERVE);
// make the changes effective
xqc.setStaticContext(xqsc);
...
```

First, you must retrieve the implementation's default values for the static context components through an XQStaticContext object. XQStaticContext defines setter and getter methods for the various static context components.

As shown in the previous example, an XQStaticContext is a value object. Changing any of the static context components doesn't have any direct effect. Only after calling setStaticContext() on the XQConnection object will the new values in the XQStaticContext become effective. Think about XQStaticContext objects as items passed by value from the XQJ driver to the application and vice-versa. Once the static context is updated, all (and only) subsequently created XQExpression and XQPreparedExpression objects will assume the new values for the static context components.

```
...
// the boundary-space for this first query is implementation defined,
// i.e. depends on the implementation's defaults
XQPreparedExpression xqp1 = xqc.prepareExpression("<e> </e>");
// set the boundary-space policy to preserve
XQStaticContext xqsc = xqc.getStaticContext();
xqsc.setBoundarySpacePolicy(XQConstants.BOUNDARY_SPACE_PRESERVE);
xqc.setStaticContext(xqsc);
// the boundary-space policy for this second query *is* preserve
XQPreparedExpression xqp2 = xqc.prepareExpression("<e> </e>");
...
```

Setting Static Context Defaults

In the previous examples, the static context is updated at the connection level, and as such all subsequently created XQExpression and XQPreparedExpression objects are affected. This is perfect if you want all your XQuery expressions to be based on the same defaults in the static context. But what if the default values need to be different for some XQExpression and XQPreparedExpression objects? The application also has the ability to specify an XQStaticContext during the creation of XQExpression and XQPreparedExpression objects:

```
...
// change the boundary-space policy in the static context object
// but don't apply those change at the connection level
XQStaticContext xqsc = xqc.getStaticContext();
xqsc.setBoundarySpacePolicy(XQConstants.BOUNDARY_SPACE_PRESERVE);
// create a prepared expression using the modified static context
// other expressions subsequently created are not affected
XQPreparedExpression xqp1 = xqc.prepareExpression("<e> </e>", xqsc);
...
```

Again, such an approach is useful if some static context components need to be changed for a specific expression, but you want to keep the default values for (most of) the other expression being executed.

Almost all static context components are accessible through XQStaticContext. Here is the list:

- Statically known namespaces
- Default element/type namespace
- Default function namespace
- Context item static type
- Default collation
- Construction mode
- Ordering mode
- Default order for empty sequences
- Boundary-space policy
- Copy-namespaces mode
- Base URI

In addition, XQStaticContext includes a number of XQJ-specific properties:

- Binding mode

- Holdability of the result sequences
- Scrollability of the result sequences
- Query language
- Query timeout

The most frequently used properties are "Binding mode" and "Scrollability," and they are discussed in other chapters of this XQJ tutorial. The query language is by default XQuery, and can be changed to XQueryX. Supporting query timeout, which sets the number of seconds an implementation will wait for a query to execute, is optional; implementations are free to ignore it.

Tutorial: XQuery API for Java (XQJ) - XQuery Type System

This chapter describes how XQJ interacts with the XQuery type system. XQuery is a strongly typed language; the type system is based on XML Schema. As it is an inherent part of XQuery, you'll need some notions of XML Schema to properly understand the XQuery type system. However, it is out of scope for this XQJ tutorial to go into all the details.

Sequence and Item Types

XQuery defines a sequence type as a type that can be expressed using the SequenceType syntax. It consists of an item type that constrains the type of each item in the sequence, and a cardinality that constrains the number of items in the sequence. Having sequences and items in the XQuery type system, XQJ defines two corresponding interfaces: XQSequenceType and XQItemType.

XQSequenceType is a rather simple interface with only 3 methods:

- getItemType() retrieves the item type of the sequence type.
- getItemOccurrence() retrieves the cardinality that constraints the number of items.
- toString() yields a string representation of the sequence type.

XQItemType encapsulates more information:

- getItemKind() returns whether it is an element, attribute, atomic type, and so on.
- getBaseType() specifies the built-in schema type closest matching this item type. For example, xs:anyType, xs:string, and so on.
- getNodeName() yields the name of the node, which is a QName.
- getPIName() yields the name of a processing instruction, which is a String.
- getTypeName() specifies a QName identifying the XML Schema type of the item type. This can be either a built-in XML Schema type or user defined.
- toString() yields a string representation of the item type.

There are some more attributes defined on XQItemType related to user defined schema types, but that would bring us too far afield in the context of this introductory tutorial.

XQSequenceType and XQItemType objects are used in two different contexts:

- The representation of the static type of an external variable defined in a query and the query result. In this context, the type is possibly abstract, like item(), node()+, or xs:anyAtomicType?.
- The concrete type of an item; here abstract types are not applicable.

XQItemType Usage

Let's have a closer look at XQItemType, which specifies the item kind and base type:

```
...
XQSequenceType xqtype = ...
XQItemType xqitype = xqtype.getItemType();
int itemKind = xqitype.getItemKind();
int schemaType = xqitype.getBaseType();
...
```

XQJ defines constants for each of the item kinds representable in XQuery SequenceType syntax:

Sequence Type	XQJ Definition
QName	XQITEMKIND_ATOMIC

element(...)	XQITEMKIND_ELEMENT
attribute(...)	XQITEMKIND_ATTRIBUTE
comment()	XQITEMKIND_COMMENT
document-node()	XQITEMKIND_DOCUMENT
document-node(element(...))	XQITEMKIND_DOCUMENT_ELEMENT
processing-instruction(...)	XQITEMKIND_PI
text()	XQITEMKIND_TEXT
item()	XQITEMKIND_ITEM
node()	XQITEMKIND_NODE

getBaseType() is used to determine more precisely the type in case of, for example, XQITEMKIND_ATOMIC — when you have an atomic type, is it an xs:string or xs:integer? XQJ defines constants for all the built-in XML Schema and XQuery types. It's a long list; here is a small excerpt:

XML Schema Type	XQJ Definition
xs:string	XQBASETYPE_STRING
xs:integer	XQBASETYPE_INTEGER
xs:untypedAtomic	XQBASETYPE_UNTYPEDATOMIC

Working with Dynamic Types

Iterating over query results, XQJ allows you to request precise type information about each item. Suppose you want to use a different get*XXX*() method, depending on the item type:

```
XQSequence xqs = ...
while (xqs.next()) {
 XQItemType xqtype = xqs.getItemType();
 if (xqtype.getItemKind() == XQItemType.XQITEMKIND_ATOMIC) {
   // We have an atomic type
   switch (xqtype.getBaseType()) {
     case XQItemType.XQBASETYPE_STRING:
     case XQItemType.XQBASETYPE_UNTYPEDATOMIC: {
       String s = (String)xqs.getObject();
       ...
       break;
     }
     case XQItemType.XQBASETYPE_INTEGER: {
       long l = xqs.getLong();
       ...
       break;
     }
     ...
   }
 } else {
   // We have a node, retrieve it as a DOM node
   org.w3c.dom.Node node = xqs.getNode();
   ...
 }
}
```

This can make your code rather complex and long. Sometimes it can't be avoided, but most of the time a number of shortcuts are available to you. As explained in XQJ Tutorial Part IV: Processing Query Results, you can use some of the more the general purpose methods.

Suppose you need a DOM node in case the query returns a node, and the string value for all atomic values. The next simple example shows how to do this:

```
XQSequence xqs = ...
```

```
while (xqs.next()) {
 XQItemType xqtype = xqs.getItemType();
 if (xqtype.getItemKind() == XQItemType.XQITEMKIND_ATOMIC) {
   // We have an atomic type
   String s = xqs.getAtomicValue();
   ...
 } else {
   // We have a node, retrieve it as a DOM node
   org.w3c.dom.Node node = xqs.getNode();
   ...
 }
}
```

Working with Static Types

That's it for the dynamic type of items. The next example shows how to retrieve the static type of a query (for the JDBC, ODBC, and SQL users, this is somewhat similar to "describe information"):

```
...
XQPreparedExpression xqe = xqc.prepareExpression("1+2");
XQSequenceType xqtype = xqe.getStaticResultType();
System.out.println(xqtype.toString());
...
```

With DataDirect XQuery, this example outputs xs:integer to stdout. Similarly, you can use the prepared expression to retrieve information about the external variables. As shown in the next examples, first we determine the external variables declared in the query; next we retrieve the static type of each of the external variables:

```
...
XQPreparedExpression xqe = xqc.prepareExpression(
 "declare variable $i as xs:integer external; $i+1");
QName variables[] = xqe.getAllExternalVariables();
for (int i=0; i<variables.length; i++) {
 XQSequenceType xqtype = xqe.getStaticVariableType(variables[i]);
 System.out.println(variables[i] + ": " + xqtype.toString());
}
...
```

Why is this useful at all? Let's have a quick look at a use case.

The idea of exposing XQueries as web services is not new; see, for example, the paper XQuery at Your Web Service. A fully functional example of such an "XQuery Web Service" is available on www.xquery.com. It is basically a servlet that reads XQueries from a specific directory (or from the classpath), and makes each of the queries available as functions accessible through SOAP or REST. The servlet needs to determine the external variables in each of the queries in order to generate the Web Services Description Language (WSDL), which contains an XML Schema definition describing the parameters for each operation — something like this (assuming an XQuery with two external variables, $employeeName and $hiringDate, declared as xs:string and xs:date):

```
<xs:element name="XXX">
 <xs:complexType>
   <xs:sequence>
     <xs:element name="employeeName" type="xs:string"/>
     <xs:element name="hiringDate" type="xs:date"/>
   </xs:sequence>
 </xs:complexType>
</xs:element>
```

All the information required to generate such an XML schema definition is available in the *sequence type* of each declared variable. And XQJ makes this information immediately accessible. We could write a piece of code translating the relevant item kinds and base types to an XML

Schema definition as shown above — it's only a matter of writing a number of Java switch statements. But is there an easier way?

Using the toSting() Method

XQJ defines toString() on XQItemType as implementation dependent. More precisely, it is a requirement to return a human-readable string. In any case, with DataDirect XQuery the string representation is based on the XQuery sequence type syntax, where the QName prefixes are as follows:

- For QNames representing built-in XML schema types, the xs prefix is always used.
- For QNames representing element or attribute names, the prefixes as defined in the query are used. In case of duplicates, one is chosen in an implementation dependent manner.

Returning to our XQuery Web Service use case, the strategy to map the external variable declaration to the WSDL becomes rather simple using toString():

- If the XQItemType is an atomic type, use the string representation.
- If the XQItemType is anything else, use xs:anyType.

Applications have also the ability to create XQItemType objects. XQJ Tutorial Part VIII: Binding External Variables and XQJ Tutorial Part IX: Creating XDM Instances show how this can be done and describes other use cases.

Binding External Variables with XQuery API for Java (XQJ)

XQJ Tutorial Part III: Querying Data from XML Files or Java XML APIs, shows through some simple examples how to bind a value to an external variable declared in your query. This section contains more details on that subject.

The XQuery Data Model

XQuery operates on the abstract, logical structure of XML, known as the XQuery Data Model (XDM). As such, in XQuery the value bound to an external variable is an XDM instance by definition. So, how do you convert a Java object in your Java application into an XDM instance? XQJ defines this mapping and glues it all together. Let's take a look at a simple example:

```
...
XQPreparedExpression xqp;
XQSequence xqs;
xqp = xqc.prepareExpression(
  "declare variable $id as xs:integer external; " +
  "doc('orders.xml')//order[id=$id]");
xqp.bindObject(new QName("id"),new Integer(174), null);
xqs = xqp.executeQuery();
...
```

The bindObject() Method

The bindObject() method is defined in the XQDynamicContext interface. It provides a number of methods to bind values to external variables. As XQDynamicContext is both the base for XQExpression and XQPreparedExpression, both expression implementations support binding values to external variables.

The first argument to the bindObject() method is a QName, which identifies the external variable in your XQuery. The second argument is the Java object to be bound. XQJ defines a mapping of Java objects to XDM instances; here are a few examples:

Java Type	XQuery Type
java.lang.Integer	xs:int
java.lang.BigInteger	xs:integer
java.lang.BigDecimal	xs:decimal
java.lang.String	xs:untypedAtomic
org.w3c.dom.Document	untyped document node
org.w3c.dom.Element	untyped element node

The third argument, for which null is specified in the example above, allows you to override the default mapping. This is shown next. (The createAtomicType() method defined on XQDataFactory is introduced in XQJ Tutorial Part IX: Creating XDM Instances, but for now it's sufficient to know that it returns an XQItemType object representing the specified atomic type):

```
...
XQItemType xsinteger;
xsinteger = xqc.createAtomicType(XQItemType.XQBASETYPE_INTEGER);

XQPreparedExpression xqp;
XQSequence xqs;
```

```
xqp = xqc.prepareExpression(
  "declare variable $v1 external; " +
  "declare variable $v2 external; " +
  "$v1 instance of xs:integer, "+
  "$v1 instance of xs:int, "+
  "$v2 instance of xs:integer, "+
  "$v2 instance of xs:int");
xqp.bindObject(new QName("v1"),new Integer(174), null);
xqp.bindObject(new QName("v2"),new Integer(174), xsinteger);
...
```

This example yields a sequence of four xs:boolean instances:

true, true, true, false

A Java Integer is by default mapped to xs:int. xs:int extends by restriction xs:integer; as such the first two 'instance of' expressions evaluate to true. The second external variable is bound with an xs:integer instance — the application explicitly specifies to create such an XDM instance. As such, the last 'instance of' evaluates to false, as xs:integer is not extending xs:int.

Note that various error conditions can occur during the binding process:

- The conversion from Java to XDM instance can fail. For example, a java.lang.Integer object with the value 10000 is converted into a xs:byte. As 10000 is out of bounds of the xs:byte value space, an error will be reported.
- Once converted into an XDM instance, the binding can still fail in cases where the external variable declaration includes a declared type. In such a scenario, the XDM instance must match the declared type according to the rules of SequenceType matching. For example, a java.lang.Integer is bound and converted into an xs:integer instance, but the external variable is declared as xs:string.

Binding Atomic Values

We have introduced the bindObject() method through some examples, but XQDynamicContext has many more bind methods. bindAtomicValue() accepts a java.lang.String and will convert it to the specified type according to the casting rules from xs:string; essentially, the specified string must be in the lexical space of the specified atomic type. In the following example, the Java String "123" is converted into xs:string, xs:integer, and xs:double instances and bound to the external variables $v1, $v2, and $v3:

```
...
xqp.bindAtomicValue(new QName("v1"), "123",
  xqc.createAtomicType(XQItemType.XQBASETYPE_STRING));
xqp.bindAtomicValue(new QName("v2"), "123",
  xqc.createAtomicType(XQItemType.XQBASETYPE_INTEGER));
xqp.bindAtomicValue(new QName("v3"), "123",
  xqc.createAtomicType(XQItemType.XQBASETYPE_DOUBLE));
...
```

In contrast, the following two bindAtomicValue() invocations will fail. The first because "abc" is not in the value spaces of xs:integer. The second one because no type has been specified as third parameter; unlike with bindObject(), bindAtomicValue() has no default mapping and a XQItemType must be specified as third argument:

```
...
xqp.bindAtomicValue(new QName("e"), "abc",
  xqc.createAtomicType(XQItemType.XQBASETYPE_INTEGER));
xqp.bindAtomicValue(new QName("e"), "123", null);
...
```

bind() Methods for Java Primitive Types

XQDynamicContext also provides bind*XXX*() methods for each of the Java primitive types:

- boolean
- byte
- double
- float
- int
- long
- short

For example, binding an xs:integer instance 123 to the external variable $v. The default mapping for int is xs:int; as such we specify the type as third parameter:

```
...
xqp.bindInt(new QName("v"), 123,
  xqc.createAtomicType(XQItemType.XQBASETYPE_INTEGER));IntDate
...
```

Binding DOM, SAX, and StAX

Binding a DOM node is also possible; basically it is equivalent to bindObject, with the restriction that the argument must be a DOM node and as such the XDM instance is always a node, never an atomic value. Of course, in addition to DOM, the SAX and StAX APIs are supported through XQDynamicContext.

Let's read an XML document, foo.xml, through DOM, and StAX, and each time bind it to an external variable, $v.

The DOM version:

```
...
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
factory.setNamespaceAware(true);

DocumentBuilder parser = factory.newDocumentBuilder();
Document domDocument = parser.parse("foo.xml");
xqp.bindNode(new QName("e"), domDocument,null);
...
```

The StAX version:

```
...
XMLInputFactory factory = XMLInputFactory.newInstance();
FileInputStream doc = new FileInputStream("foo.xml");
XMLStreamReader reader = factory.createXMLStreamReader(doc);
xqp.bindDocument(new QName("e"), reader, null);
...
```

But of course, this is something you only want to use in specific scenarios. The simple use case of binding an XML file can be easily accomplished in a single line. The XQJ implementation will make sure that the XML file is parsed and queried:

```
...
xqp.bindDocument(new QName("e"), new FileInputStream("foo.xml"));
...
```

An XQItem or a complete XQSequence can also be bound to an external variable. That is described in more details in XQJ Part X: XML Pipelines, along with XQJ support for the JAXP Source and Result interfaces.

Creating XDM Instances with XQuery API for Java (XQJ)

In the previous chapters, we learned how to handle XDM instances that result from query execution; iterating through sequences; and getting access to the items in the sequence. What if we want to *create* an XDM instance, without executing a query?

XQJ offers functionality to create both XQSequence and XQItem objects, not as a result of a query execution, but rather as standalone XDM instances. This functionality is offered through the XQDataFactory interface. An XQDataFactory can create the following types of objects:

- XQItem
- XQSequence
- XQItemType
- XQSequenceType

Every XQConnection must implement the XQDataFactory interface. In XQJ 1.0 these are the only concrete XQDataFactory implementations; future XQJ versions might introduce different mechanisms to get access to an XQDataFactory.

Creating Types

In XQJ Tutorial Part VII: XQuery Type System, we introduced the XQItemType and XQSequenceType interfaces. We also showed how these objects are used to describe the static type of a query result and external variables. How do we create such type objects in our application?

XQJ defines a dozen of XQITEMKIND_*XXX* constants. For each of them, there is a matching create*XXX*Type method:

- createAtomicType
- createAttributeType
- createCommentType
- createDocumentElementType
- createDocumentType
- createElementType
- creatItemType
- createNodeType
- createProcessingInstructionType
- createSchemaAttributeType
- createSchemaElementType
- createTextType

Let's take a closer look at some of the most commonly used methods.

createAtomicType

The method createAtomicType(), creates an XQItemType object representing an XQuery atomic type. It accepts a single argument, an integer which is one of the predefined XQBASETYPE constants. The next example creates three XQItemType instances representing xs:integer, xs:string, and xs:decimal:

```
...
XQItemType xsinteger = xqc.createAtomicType(
                            XQItemType.XQBASETYPE_INTEGER);
XQItemType xsstring = xqc.createAtomicType(
```

```
                              XQItemType.XQBASETYPE_STRING);
XQItemType xsdecimal = xqc.createAtomicType(
                              XQItemType.XQBASETYPE_DECIMAL);
...
```

Every XQConnection is an XQDataFactory; in the example we've used our XQConnection(xqc) to create these XQItemType instances. However, the XQItemType objects are completely independent of the connection.

Where the above example shows how to create XQItemType objects representing one of the built-in atomic XML Schema types, there is a second flavor of createAtomicType() for user-defined atomic types. Assume a *hatsize* user-defined atomic type derived from xs:integer in the *http://www.hatsize.com* schema:

```
...
XQItemType hatsize;
hatsize = xqc.createAtomicType(
                XQItemType.XQBASETYPE_INTEGER,
                new QName("http://www.hatsizes.com", "hatsize"),
                new URI("http://www.hatsizes.com"));
...
```

createElementType

Beside atomic types, element types are frequently used. In the next example we create an XQItemType representing element(person):

```
...
XQItemType type;
type = xqc.createElementType(
                new QName("person"),
                XQItemType.XQBASETYPE_ANYTYPE);
...
```

The first argument to createElementType() is a QName. Where in the previous example a "person" element is created in no namespace, the next example creates an element type person in the namespace http://www.example.com. The second argument can be any of the predefined types; in addition to xs:anyType, xs:untyped is frequently used:

```
...
XQItemType type;
type = xqc.createElementType(
                new QName("person","http://www.example.com"),
                XQItemType.XQBASETYPE_UNTYPED);
...
```

The first argument can also be null, which is assumed to be the wild card. The following code snippet shows the creation of element(*, xs:untyped):

```
...
XQItemType type;
type = xqc.createElementType(
                null,
                XQItemType.XQBASETYPE_UNTYPED);
...
```

createDocumentType

What about document-node() types? In the following example we create two XQItemType instances, a first representing any document, and a second representing a well-formed untyped document:

```
...
XQItemType type1;
```

```
XQItemType type2;
type1 = xqc.createDocumentType();
type2 = xqc.createDocumentElementType(
                 xqc.createElementType(
                           null,
                           XQItemType.XQBASETYPE_UNTYPED));
...
```

In addition to XQItemTypes, XQSequenceType objects can also be created. As explained in XQJ Tutorial Part VII: Typing, an XQSequence consists of:

- An XQItemType
- The cardinality required to constraint the number of items (one of the OCC_*XXX* constants defined on XQSequenceType)

As such creating an XQSequenceType is simple. The next example shows how to create a xs:string* sequence type:

```
...
XQItemType itemType;
XQSequenceType sequenceType;
itemType = xqc.createAtomicType(
                    XQItemType.XQBASETYPE_STRING);
sequenceType = xqc.createSequenceType(
                    itemType,
                    XQSequenceType.OCC_ZERO_OR_MORE);
...
```

Using Types

So, we now know how to create types. But why bother? How can we make use of all these types?

Assume an XQSequence, iterating over the items. If the item is a node retrieve it through the DOM; get atomic values as Strings. This can be accomplished using the instanceOf() method, passing in an XQItemType object:

```
...
XQItemType nodeType = xqc.createNodeType();
XQSequence xqs = ...
...

while (xqs.next()) {
  if (xqs.instanceOf(nodeType)) {
    org.w3c.dom.Node node = xqs.getNode();
    ...
  } else {
    String s = xqs.getAtomicValue();
    ...
  }
}
...
```

Support for Static Types

Some XQuery implementations have support for the Static Typing Feature as defined in XQuery. This requires implementations to detect and report type errors during the static analyses phase. For expressions depending on the context item, the application must specify the static type of the context item. Why? In order to perform static typing, the implementation has to know the static type of the context item. The application has to provide the static type; failing to do so results in an error being reported during the static analysis phase.

As the static type of the context item is a static context component, the XQJ XQStaticContext interface allows it to be manipulated. The next example shows how to set the static type of the initial context item to document-node(element(*, xs:untyped)):

```
...
XQItemType documentType;
documentType = xqc.createDocumentElementType(
                    xqc.createElementType(
                           null,
                           XQItemType.XQBASETYPE_UNTYPED));
XQStaticContext xqsc = xqc.getStaticContext();
xqsc.setContextItemStaticType(documentType);
...
XQPreparedExpression xqp;
xqp = xqc.prepareExpression("//address",
                            xqsc);
...
```

Overriding Default Type Mappings

As a last use case of XQItemType, think about some of the examples described in XQJ Tutorial Part VIII: Binding External Variables. The bind*XXX*() methods defined on XQDynamicContext have all a third parameter which allows you to override the default Java to XQuery data type mapping.

In this example, we bind a Java Integer to the external variable; but rather than using the default mapping to xs:int, we specify to map it to a xs:short:

```
...
XQItemType xsshort;
xsshort = xqc.createAtomicType(XQItemType.XQBASETYPE_SHORT);
XQPreparedExpression xqp;
xqp = xqc.prepareExpression(
             "declare variable $v as xs:short external; " +
             "$v + 1");
xqp.bindInt(new QName("v"), 22, xsshort);
...
```

How to Create XDM Instances

In addition to its ability to create XQItemType and XQSequenceType instances, XQDataFactory also provides the ability to create XQItem and XQSequence instances.

If this sounds familiar to you, it's because you understand the way binding to an XQDynamicContext works, as discussed in XQJ Tutorial Part VIII: Binding External Variables. Once you know this, you're more than half the way to understanding how XQItem instances are created.

For every bindXXX() method defined on XQDynamicContext, there is a corresponding createItemFromXXX() method. Let's look at a simple example — binding a java.math.BigDecimal to an external variable $d:

```
...
XQExpression xqe = ...
xqe.bindObject(new QName("d"),new BigDecimal("174"), null);
...
```

The following examples show how to create an XQItem of type xs:decimal from the same java.math.BigDecimal:

```
...
XQItem xqi = xqc.createItemFromObject(new BigDecimal("174"), null);
...
```

XDM Instances and Connections

Note that the XQItem objects created through XQDataFactory are independent of any connection.

Suppose you execute a query returning a single item, and subsequently close the connection but still require access to the XQItem. Closing the XQConnection will invalidate the XQItem object resulting from the query execution. Because of this, XQDataFactory has an XQItem copy method, createItem(). createItem() accepts a single XQItem argument, and returns a (deep) copy of the specified item.

The following example shows how to make a query result available after closing the XQSequence or XQConnection:

```
...
XQConnection xqc = ...
XQExpression xqe = xqc.createExpression();
XQSequence xqs;
xqs = xqe.executeQuery("(doc('book.xml')//paragraph)[1]");
xqs.next();
XQItem xqi = xqc.createItem(xqs.getItem());
xqc.close();
// although the connection is closed, xqi is still valid.
...
```

Reducing Parsing of Repeated Queries

Imagine that you have an XML document that needs to be queried multiple times, but you don't want to incur the XML parsing overhead each time it is queried. In the following example, two queries are executed, and books.xml is parsed twice:

```
...
XQExpression xqe = xqc.createExpression();
xqe.executeQuery("fn:doc('book.xml')//paragraph[contains(.,'XQuery')]");
xqe.executeQuery("fn:doc('book.xml')//paragraph[contains(.,'SQL')]");
...
```

Or suppose you receive a transient XML stream, for example in a servlet environment, and need to query the stream multiple times. Then, one way or the other, the data will need to be buffered in order to query it more than once.

How can we both ensure that an XML document is parsed only once, and, in the event the XML stream is transient, also make it 'queryable' multiple times?

Consider two XQPreparedExpression objects, xqp1 and xqp2. The following example first creates an XQItem representing the XML document; as such it will be parsed only once. Next, it is bound to two different XQPreparedExpression objects:

```
...
InputStream input = ...
XQItem doc = xqc.createItemFromDocument(input, null);
...
xqp1.bindItem(new QName("doc"), doc);
...
xqp2.bindItem(new QName("doc"), doc);
...
```

Such an approach, however, does have a couple of disadvantages, especially when working with a large documents — namely, scalability and memory consumption. For example, in the case of DataDirect XQuery, the streaming capabilities will not be of much use as the complete XML document is instantiated in-memory. As you might expect, however, there are techniques for handling large documents. See XQJ Tutorial Part XI: Processing Large Inputs for more information on this topic.

Finally, XQDynamicContext also allows creating XQSequence objects. The createSequence() copy

operation, with its XQSequence argument, lets you return a copy of an XQSequence object. As with the XQItem example above, it allows the possibility for query results to outlive an XQConnection.

A second flavor of createSequence() accepts a java.util.Iterator, returning a sequence of items based on the objects returned by the Iterator. The objects are converted into XDM instances using the default object mapping defined in XQJ. For example, the following code snippet results in a sequence of xs:decimal instances:

```
...
// assume an ArrayList of BigDecimal objects
ArrayList list = ...
XQSequence s = xqc.createSequence(list.iterator());
...
```

XML Pipelines with XQuery API for Java (XQJ)

In this chapter, we'll show you how to create a pipeline of XQueries and how to integrate XQuery with JAXP-based XML processors.

What Is an XML Pipeline?

An XML pipeline is a sequence of XML processes, also called *transformations*; in an XML pipeline, the result of one transformation is passed as input to the next one. You can find an interesting discussion about XML pipelines in Norman Walsh's essay Why Pipelines?

The transformations that come most easily to mind are XSLT transformations, XML Schema validation, simple XML parsing, and so on. But of course, a transformation can also be an XQuery. This chapter is about integrating XQuery and XQJ in an XML pipeline.

A Simple XML Pipeline

Let's start with an XML pipeline, where the result of a first XQuery is passed to a second. Given that a query execution results in an XQSequence object, and an XQSequence object can be bound to an external variable, pipelining two XQueries is rather simple.

```
...
XQExpression xqe1 = xqc.createExpression();
XQSequence xqs1;
xqs1 = xqe1.executeQuery("doc('orders.xml')//orders");
XQExpression xqe2 = xqc.createExpression
xqe2.bindSequence(xqs1);

XQSequence xqs2 = xqe2.executeQuery(
  "declare variable $orders as element(*,xs:untyped) external; " +
  "for $order in $orders where $order/@status = 'closed' " +
  "return <closed_order id = '{$order/@id}'>{ " +
  " $order/* </closed_order>";
xqs2.writeSequence(System.out, null);
xqe2.close();
xqe1.close();
...
```

In this example, both queries in the pipeline are executed in the context of a single XQConnection object. But this is not required; it is entirely possible to have two different connection objects, even from different XQJ implementations.

The application writer shouldn't be concerned with how the query result from the first XQuery is passed to the second XQuery. The mechanism used to pass the query result — whether DOM, SAX, StAX, serialized XML or some other (proprietary) mechanism, for example — is an implementation detail. The application can assume that the most appropriate mechanism is used.

Using XQuery Results with Other XML Processors

Let's now look at how an XQuery can be integrated in a pipeline with other XML processors. We'll present two scenarios. First, one in which the result of an XQuery is passed to an XSLT transformation; and then, vice versa — an XQuery processes the results of an XSLT transformation.

On the Java platform, XSLT transformations are processed through the JAXP API. JAXP makes use of the Source and Result interfaces to query XML documents and to handle transformation results. As

XQJ has built-in support for these interfaces, it is possible to integrate both JAXP and XQJ in a pipeline.

Passing XSLT Results to XQuery

The following example invokes an XSLT transformation, passing the transformation results as input to an XQuery. Under the covers, SAX events are pushed to the XQuery engine. SAX (or StAX) is preferable to passing a serialized XML stream, in terms of both performance and scalability. (DOM trees, while useful in some cases, should usually be your last choice for this type of query processing.)

First, an XMLFilter for the XLST transformation is created. A SAXSource is created as well, and it serves as the input for the XSLT Transformation. It is this SAXSource that is bound to the external variable of the XQuery. Subsequently the XQuery is executed, and finally the application consumes the results of the XQuery:

```
...
// Build an XMLFilter for the XSLT transformation
SAXTransformerFactory stf;
stf = (SAXTransformerFactory) SAXTransformerFactory.newInstance();
XMLFilter stage1;
stage1 = stf.newXMLFilter(new StreamSource("stage1.xsl"));

// Create a SAX source, the input for the XSLT transformation
SAXSource saxSource;
saxSource = new SAXSource(stage1,new InputSource("input.xml"));

// Create the XQuery expression
XQPreparedExpression stage2;
stage2 = xqc.prepareExpression(new FileInputStream("stage2.xquery"));

// Bind the input document as a Source object to stage2
stage2.bindDocument(new QName("var"),saxSource);

// Execute the query (stage2)
XQSequence result = stage2.executeQuery();

// Process query results
result.writeSequenceToResult(
  new StreamResult(System.out));
...
```

As you might have noticed in this example, the pipeline is activated starting from the end. Under the covers, the XQJ implementation invokes the parse() method on the SAXSource object bound to the external variable. This, in turn, starts invoking SAX callbacks to the XSLT transformation, which ultimately performs callbacks to the XQJ implementation. These callbacks represent the result of the XSLT transformation, allowing the XQuery to yield results back to the application.

Passing XQuery Results to XSLT

Let's now have a closer look at a pipeline in which the results of an XQuery are passed to an XSLT transformation.

First we present a utility class, XQJFilter, an XMLFilter based on an XQJ XQPreparedExpression object:

```
public class XQJFilter extends XMLFilterImpl {
  XQPreparedExpression _expression;

  public XQJFilter(XQPreparedExpression expression) {
    _expression = expression;
  }
```

```
  public void parse(InputSource source) throws SAXException {
    try {
      XQSequence xqs = _expression.executeQuery();
      Result result = new SAXResult(this.getContentHandler());
      xqs.writeSequenceToResult(result);
      xqs.close();
    } catch (XQException e) {
      throw new SAXException(e);
    }
  }
}
```

Next we'll use this XQJFilter implementation to build the pipeline:

```
...
// Create an XQuery expression
XQPreparedExpression xqp;
xqp = xqc.prepareExpression(new FileInputStream ("query.xq"));
// Create the XQJFilter, the first stage in our pipeline
XQJFilter stage1 = new XQJFilter(xqp);
// Create an XMLFilter for the XSLT transformation, the 2nd stage

SAXTransformerFactory stf;
stf = (SAXTransformerFactory) SAXTransformerFactory.newInstance();
XMLFilter stage2 = stf.newXMLFilter(new StreamSource("stage2.xsl"));
stage2.setParent(stage1);
// Make sure to capture the SAX events as result of the pipeline
stage2.setContentHandler(...);
// Activate the pipeline
stage2.parse("");
...
```

As with the previous example, the pipeline here is also activated at the end, by calling the parse() method on the second stage (stage2.parse).

Processing Large Inputs with XQuery API for Java (XQJ)

This chapter explains how to handle and query large XML documents through the XQJ API.

The Roots of DOM

Since XML became a standard in the late 90's, we have been taught that XML is a tree; and the most intuitive (and popular) representation of such a tree has been (still is!) the Document Object Model (DOM).

When you think about querying XML documents, using XQuery, XSLT or XPath, you usually think about a processor that navigates the DOM tree, extracts values, compares the values it needs, and creates another DOM as a result of those operations. This is indeed what happens using typical XML processing implementations. Although today's processors use a more optimal representation than DOM, one problem remains the same — scalability.

What happens if the XML you need to process cannot be represented within the physical constraints of the memory available to your application? That's usually the limit that typical "in-memory" XQuery, XSLT, XPath implementations hit when processing DOM, and short of writing your own SAX- or StAX-based processing, you really have no alternatives. But what if you are able to forget about DOMs, forget about materializing the whole XML tree in memory, and do XML processing in a purely streaming fashion?

The Benefits of XML Streaming

Using an XQuery streaming processor, like DataDirect XQuery, is a good start. But a chain is only as strong as the weakest link. Beside the streaming capabilities of your XQuery implementation, the API must have the provision to handle those large XML fragments.

From an XQuery API perspective, it is crucial that the input to your query can be handled in a streaming fashion. In XQJ Tutorial Part VIII: Binding External Variables, we learned how to bind values to external variables declared in an XQuery. By default, binding a value to an XQExpression or XQPreparedExpression using bind*XXX*(), it is consumed during the binding process, and it stays active and valid for all subsequent execution cycles. We say that XQJ operates in 'immediate binding mode.'

Let's look closely at one of the pipeline examples from XQJ Tutorial Part X: XML Pipelines in this series.

```
...
XQExpression xqe1;
XQSequence xqs1;

xqe1 = xqc.createExpression();
xqs1 = xqe1.executeQuery("doc('orders.xml')//order");

XQExpression xqe2;
xqe2 = xqc.createExpression();
xqe2.bindSequence(xqs1);
xqe1.close();

XQSequence xqs2;
xqs2 = xqe2.executeQuery(
  "declare variable $orders as element(*,xs:untyped) external; " +
  "for $order in $orders " +
```

```
  "where $order/@status = 'closed' " +
  "return " +
  " <closed_order id = '{$order/@id}'>{ " +
  " $order/* " +
  " </closed_order>";
xqs2.writeSequence(System.out, null);
xqe2.close();
...
```

Binding and Buffering

During the bindSequence() call, the complete xqs1 sequence is consumed. Subsequently, we can safely close the xqe1 expression, freeing up any runtime resources it holds. On the other hand, consuming the complete sequence during bindSequence() implies that the XQJ implementation has to buffer the data one way or the other for subsequent query evaluations. All this works perfectly fine as long as we're handling relatively small XML instances. But as the data is buffered, it deprives the underlying XQuery processor of opportunities to take advantage of its streaming capabilities.

The default binding mode in XQJ is 'immediate,' which means the value bound to an external variable is consumed during the bind*XXX*() method. An application also has the ability to set the binding mode to 'deferred.' With deferred binding mode, the application gives a hint to the XQJ implementation and underlying XQuery processor to take advantage of its streaming capabilities. In deferred binding mode, bindings are only active for a single execution cycle. The application is required to explicitly re-bind values to every external variable before each execution.

But what if you know that the data bound to the external variable will be used for only a single XQuery execution. Is there then a way to inform the XQJ/XQuery implementation of possible optimization opportunities, and use its streaming capabilities?

You can change the binding mode through the XQStaticContext interface, as shown in the following example (refer to XQJ Tutorial Part VI: Manipulating the Static Context for more information on how to manipulate the static context):

```
...
XQStaticContext xqsc = xqc.getStaticContext();
// change the binding mode
xqsc.setBindingMode(XQConstants.BINDING_MODE_DEFERRED);
// make the changes effective
xqc.setStaticContext(xqsc);
...
```

In deferred mode, the application cannot assume that the bound value will be consumed during the invocation of the bind*XXX*() method. The XQJ implementation is free to read the bound value either at bind time or during the subsequent evaluation and processing of the query results. This has some consequences for resource clean-up: the first example will not work properly in deferred binding mode, for example. In that example, nNote that xqe1 was closed right after calling bindSequence(). We can address this by modifying the example as follows:

```
...
XQExpression xqe1;
XQSequence xqs1;

xqe1 = xqc.createExpression();
xqs1 = xqe1.executeQuery("doc('orders.xml')//orders");
XQExpression xqe2 = xqc.createExpression();
xqe2.bindSequence(xqs1);

XQSequence xqs2 = xqe2.executeQuery(
  "declare variable $orders as element(*,xs:untyped) external; " +
  "for $order in $orders " +
  "where $order/@status = 'closed' " +
```

```
  "return " +
  " <closed_order id = '{$order/@id}'>{ " +
  " $order/* " +
  " </closed_order>";
xqs2.writeSequence(System.out, null);
xqe2.close();
xqe1.close();
...
```

Binding Streams for Efficient Processing

The previous example shows how to build a pipeline of XQueries. But the deferred binding mode also applies to the other bind*XXX*() methods. In the following example, we show how to bind a StreamSource to the context item. As the binding mode is deferred, the implementation can handle the query in streaming mode, and as such process huge XML documents that don't otherwise fit in available memory.

```
...
XQStaticContext xqsc = xqc.getStaticContext();
// change the binding mode
xqsc.setBindingMode(XQConstants.BINDING_MODE_DEFERRED);
// make the changes effective
xqc.setStaticContext(xqsc);

XQExpression xqe;
XQSequence xqs;

xqe = xqc.createExpression();
xqe.bindDocument(
  XQConstants.CONTEXT_ITEM,
  new StreamSource("large_orders_document.xml"));
xqs = xqe.executeExpression("/orders/order")
...
```

To conclude, using deferred binding mode requires a little more care than is immediately apparent. But the potential improvements when querying large XML documents are enormous. Of course, the API needs to provide the necessary functionality, but the heavy lifting is performed in the underlying XQuery processor — this is especially true with DataDirect XQuery, whose deferred binding mode allows you to take advantage of both XML document projection and its XML streaming capabilities. This allows efficient querying of XML documents in the hundreds of megabytes, even in the gigabytes!

XQuery API for Java (XQJ) References

Many of the following references were cited throughout the *XJQ Tutorial*. They, along with a few others, are presented here for ease of use.

XQuery 1.0: An XML Query Language, W3C Recommendation 23 January 2007

- http://www.w3.org/TR/xquery

JSR 225: XQuery API for Java™ (XQJ)

- http://www.jcp.org/en/jsr/detail?id=225

Java Database Connectivity (JDBC) API

- http://java.sun.com/products/jdbc

Summary of XQuery Standards

- http://www.xquery.com/standards/