Qizx Programmer's Guide

Qizx/open v0.4

Table of Contents

Qizx/open is basically a *class library* implementing a XQuery engine embeddable in different kinds of applications. It has therefore a Java API which allows to compile and execute queries, define execution environments, serialize the results or pass them to a SAX output. The API also provides access to the *Data Model*, allowing to manipulate *Nodes* that constitute XML documents. This API is presented in Section 5 (Java API) ; see also the Javadoc.

For *XQuery* programming, Qizx provides different kinds of extensions:

XQuery functions
: Additional predefined functions which implement serialization, XSLT transformations, dynamic evaluation, error handling, text searching and highlighting...

  They belong to a private namespace referenced by the **x:** or **qizx:** prefixes.

Extensions to standard XQuery functions
: Some standard functions (manipulation of date, time, duration) have been extended or modified to become more powerful or more convenient.

Binding Java methods as supplementary functions
: This mechanism (similar to those found in other XSLT and XQuery engines, for example Saxon) provides an easy and very powerful way to extend XQuery by binding methods of any Java class, making this methods appear as XQuery functions. Arguments and results are automatically converted if possible (number, string, boolean) or can be manipulated as opaque wrapped objects with type `xdt:object`. Qizx also converts Java arrays, vectors and enumerations to XQuery sequences and conversely.

SQL Connection
: An extension which allows to query data from relational databases (using SQL) and transform it on-the-fly into XML, providing an easy way to merge relational data into XML documents.

Web applications

This extension uses XQuery as a powerful and convenient Web page template language: the results of an expression evaluation are serialized to the HTTP output stream, or alternately can be piped to a XSLT transformation. The whole Java Servlet API is available through the Java extension mechanism mentioned above, or through convenience functions. This feature is described in a separate document: XML Query Server Pages.

1. Extension functions

These functions belong to the namespace "qizx.extensions" for which the prefix `x:` is predefined.(qizx: is deprecated).

1.1. Serialization

Though Serialization -- the process of converting XML nodes evaluated by XQuery into a stream of characters-- is well-defined in the W3C specifications, there is no specific function for this purpose.

```
function x:serialize( $tree as element(), $options as element(option) )
         as xs:string?
```

Serializes the tree element into marked-up text. The output can be a file, or the *default output* of the execution context.

Parameter $tree: a XML tree to be serialized to text.

Parameter $options: an element holding options in the form of attributes: see below.

Returned value: The path of the output file if specified, otherwise the empty sequence.

The options argument (which may be absent) has the form of a element of name "options" whose attributes are used to specify different options. For example:

```
x:serialize( $doc,
             <options output="out\doc.xml"
                      encoding="ISO-8859-1" indent="yes"/>)
```

This mechanism reminds XSLT's xsl:output specification and is very convenient since the options can be computed or extracted from a XML document.

Table 1. Implemented options

option name	values	description
method	XML (default) XHTML, HTML, or TEXT	output method
output / file	a file path	output file. If this option is not specified, the generated text is written to *default output*, which can be specified through the Java control API.
version	default "1.0"	version generated in the XML declaration. No validity check.
standalone	"yes" or "no".	No check is performed.
encoding	must be the name of an encoding supported by the JRE.	The name supplied is generated in the XML declaration. If different than UTF-8, it forces the output of the XML declaration.
indent	"yes" or "no". (default no)	output indented.
indent-value	integer value	(extension) specifies the number of space characters used for indentation.
omit-xml-declaration	"yes" or "no". (default no)	controls the output of a XML declaration.
include-content-type	"yes" or "no". (default no)	for XHTML and HTML methods, if the value is "yes", a META element specifying the content type is added at the beginning of element HEAD.
escape-uri-attributes	"yes" or "no".	for XHTML and HTML methods, escapes URI attributes.
doctype-public	the public ID in the DOCTYPE declaration.	Triggers the output of the DOCTYPE declaration.
doctype-system	the system ID in the DOCTYPE declaration.	Triggers the output of the DOCTYPE declaration.

1.2. XSL Transformation

New in Qizx/open 0.4, this function invokes a XSLT stylesheet and can retrieve the results of the transformation as a tree, or let the stylesheet output the results.

An example is given below.

```
function x:transform( $source as node(),
                      $stylesheet-URI as xs:string,
                      $xslt-parameters as element(parameters)
                      [, $options as element(options)] )
    as node()?
```

Transforms the source tree through a XSLT stylesheet. If no output file is explicitly specified in the options, the function returns a new tree.

Parameter $source: a XML tree to be transformed. It does not need to be a complete document.

Parameter $stylesheet-URI: the URI of a XSLT stylesheet. Stylesheets are cached and reused for consecutive transformations.

Parameter $xslt-parameters: an element holding parameter values to pass to the XSLT engine. The parameters are specified in the form of attributes. The name of an attribute matches the name of a xsl:param declaration in the stylesheet (namespaces can be used). The value of the attribute is passed to the XSLT transformer.

Parameter $options: [optional argument] an element holding options in the form of attributes: see below.

Returned value: if the path of an output file is not specified in the options, the function returns a new document tree which is the result of the transformation of the source tree. Otherwise, it returns the empty sequence.

Table 2. available options

option name	values	description
output-file	an absolute file path	output file. If this option is not specified, the generated tree is returned y the function, otherwise the function returns an empty sequence.
XSLT output properties (instruction xsl:output): version, standalone, encoding, indent, omit-xml-declaration etc.		These options are used by the style-sheet for outputting the transformed document. They are ignored if no output-file option is specified.
Specific options of the XSLT engine (Saxon or default XSLT engine)		An invalid option may cause an error.

This example generates the transformed document $doc into a file out\doc.xml:

```
x:transform( $doc, "ssheet1.xsl",
             <parameters param1="one" param2="two"/>,
             <options output-file="out\doc.xml" indent="yes"/>)
```

The next example returns a new document tree. Suppose we have this very simple stylesheet which renames the element "doc" into "newdoc":

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version ="1.0" >
  <xsl:template match="doc">
     <newdoc><xsl:apply-templates/></newdoc>
  </xsl:template>
</xsl:stylesheet>
```

The following XQuery expression:

```
x:transform( <doc>text</doc>, "ssheet1.xsl", <parameters/> )
```

returns:

```
<newdoc>text</newdoc>
```

1.3. (Full) Text Search

In Qizx/open, full-text search functions are *not* index-based and therefore not meant for searching huge volumes of XML. They can be used to query parsed documents or even constructed fragments, with a

decent execution speed (for example the whole works of Shakespeare, about 8 Mb, can be searched in less than one second on a recent machine).

```
function x:words( $query as xs:string [, $context-nodes as node()* ] )
        as xs:boolean
```

```
function x:fulltext( $query as xs:string [,$context-nodes as node()* ] )
         as xs:boolean
```

Note: `x:words` and `x:fulltext` are aliases for the same function.

This function implements context-sensitive full-text search: it can search boolean combinations of words, word patterns and phrases, in the context of specific elements. It is typically used inside a predicate. For example the following expression returns SPEECH elements which contain both words "romeo" and "juliet":

```
//SPEECH [ x:words(" romeo AND juliet ") ]
```

Caution: in the open-source version, the function is implemented in a simple, "brute force" way: though it achieves a decent search speed, it can in no way be compared with the index-based implementation of the commercial query engine. This brute force implementation serves as a fall-back in the (rare) cases where the query optimizer fails to find a query plan using indexes.

The function returns true if the string-value of at least one node of the *`context-nodes`* parameter matches the full-text query. Matching is therefore not affected by element substructure. For example the phrase 'to be or not to be' would be found in `<line>To be <b>or not to be</b> ..</line>`.

When *`context-nodes`* is not specified (it must be inside a predicate), the current context node '.' is used implicitly like in the example above. When *`context-nodes`* parameter is present, it can be relative to the current context node: for example this expression finds SPEECH elements which contain a LINE element which in turn contains both words "romeo" and "Juliet":

```
//SPEECH [ x:words(" romeo AND juliet ", LINE) ]
```

Syntax of full text queries:

Simple term
A word without wildcard characters '*' and '?'. By default case and accents are ignored (i.e. "café is equivalent with "CAFE").

Term with wildcard
Wildcard characters '*' and '?' can match several forms of a word, à la Unix. For example "intern*" would match intern, internal, internals etc.

Approximate term
Notation: *word~*. Uses a generic phonetic distance algorithm (somewhat similar to Soundex).

Term alternative
Notation: *term1 OR term2*. The operator *OR* or the sign '|' can be used. It has precedence over AND (see below).

Term conjunction
Notation: *term1 AND term2*. The operator *AND* or the sign '*&*' can be used or even simple juxtaposition: thus "romeo AND juliet", "romeo & Juliet" and "Roméo Juliet" are equivalent.

Term exclusion
Notation: sign *'-'* or keyword *NOT*. For example "Romeo -Juliet" is equivalent with "Romeo AND NOT Juliet".

Phrase

Ordered sequence of terms (simple words or patterns), surrounded by single or double quotes. By default, terms must appear exactly in the order specified.

It is possible to specify a tolerance or distance, which is the maximum number of words interspersed among the terms of the phrase query. The notation is *phrase~N* where N is a optional count of words (4 by default). The two following examples match the phrase "to be or not to be, that is the question":

```
//SPEECH [ x:words(" 'to be that question'~ ", LINE) ]
//SPEECH [ x:words(" 'to be or question'~6 ", LINE) ]
```

Notice that there are some limitations in this syntax: the OR cannot combine AND clauses or phrases, however this problem can be solved by boolean combinations of calls to x:words, for example:

```
doc("r_and_j.xml")//LINE [ x:words("name AND rose") or
                           x:words(" 'smell as sweet' ") ]
```

would yield the two lines (Romeo and Juliet, act II scene 2):

```
<LINE>What's in a name? that which we call a rose</LINE>
<LINE>By any other name would smell as sweet;</LINE>
```

```
function x:phrase( $words as xs:string+ [, $spacing as xs:integer ]
                   [, $context-nodes as node()* ] )
         as xs:boolean
```

Convenience function: A variant of x:fulltext specialized in phrase search, which allows words to be specified as a sequence of strings.

For example:

```
x:phrase( ("to", "be", "or", "not"), 5 )
```

is equivalent to

```
x:fulltext(" 'to be or not'~5 ")
```

```
function x:all-words( $words as xs:string+ [, $context-nodes as node()* ] )
         as xs:boolean
```

Convenience function: a variant of x:fulltext which allows words to be specified as a sequence of strings.

For example:

```
x:all-words( ("romeo", "juliet"), LINE )
```

is equivalent to

```
x:fulltext("romeo AND juliet", LINE)
```

```
function x:any-word( $words as xs:string+ [, $context-nodes as node()* ] )
         as xs:boolean
```

Convenience function: a variant of x:fulltext which allows words to be specified as a sequence of strings.

For example:

```
x:any-word( ("romeo", "juliet"), LINE )
```

is equivalent to

```
x:fulltext("romeo OR juliet", LINE)
```

```
function x:highlighter( $query as xs:string, $fragment as element(),
                        $parts as node()*, $options as element(option) ] )
         as element()
```

This function is a companion of x:words (fulltext search) which "highlights" matched terms, more precisely it returns a copy of a document fragment where matched terms are surrounded by generated elements. By default a generated element has the name 'span' and an attribute 'class' with a value equal to the prefix 'hi' followed by the rank of the term in the query. Applied to a LINE in the example above, this would produce something like:

```
<LINE>What's in a <span class='hi0'>name</span>?
 that which we call a <span class='hi1'>rose</span></LINE>
```

The first argument is a fulltext query. The second argument is the root of the document fragment to process, the optional third argument $parts is a list of sub-elements of the root which must be specifically highlighted (if empty, the whole root fragment is highlighted, otherwise only the specified parts). The 4th argument specifies options: it allows to redefine the generated elements. For example:

```
<options element='el' attribute='at' prefix='pr'/>
```

would surround terms with `<el at="pr0"></el>` instead of `<span class='hi0'></span>`.

1.4. Error handling

This currently no mechanism in XQuery to handle errors. Most errors must not be recovered (for example type errors), however a problem arises for example with the function `doc` which loads a document: if the document is not found or has parsing errors, the desired behavior is generally not that the whole execution fails with a fatal error. Errors can also happen when calling bound Java methods.

There are two similar mechanisms in Qizx for handling errors:

- One is a new syntax extension: try/catch, somewhat similar to constructs of other languages. This is the prefferred way of handling errors, as it gives access to the cause of an error.
- the other is an extension function x:catch-error. It is the older mechanism which is deprecated.

```
try { <expr> } catch($error) { <fallback-expr> }
```

The **try/catch** extended language construct first evaluates the first expression <expr>. If no error occurs, then the result of the try/catch is the value of this expression.

If an error occurs, the local variable $error receives a string value which is the error message, and the < fallback-expr> is evaluated (with possible access to the error message). The resulting value of the try/catch is in this case the value of this fallback expression. An error in the evaluation of the catch expression is of course not caught.

The type of this expression is the type that encompasses the types of both arguments.

Example: tries to open a document, returns an element `'error'` with an attribute containing the error message if the document cannot be opened.

```
try {
    doc("unreachable.xml")
} catch($err) {
    <error msg="{$err}"/>
}
```

```
function x:catch-error( $expression, $fallback )
```

This function catches a possible error in the evaluation of the first argument. If no error occurs, the value of the first argument is returned, else the second argument is evaluated and its value returned. An error in the evaluation of the second argument is not caught.

The type of the function is the type that encompasses the types of both arguments.

1.5. Miscellaneous

```
function x:eval( $expression as xs:string ) as xs:any
```

Dynamic evaluation: compiles and evaluates an expression given as a string.

The expression can use global variables, functions, namespaces of the current static context. However it has no access to the local context (for example if x:eval is invoked inside a function, the arguments or local variables of the function are not visible.)

Parameter $expression: a simple expression (cannot contain prolog declarations).

Returned value: evaluated value of the expression.

Example:

```
declare variable $x := 1;
declare function local:fun($p as xs:integer) { $p * 2 }
let $expr := "1 + $x, fun(3)"
return x:eval($expr)
```

this should return the sequence (2, 6).

```
function x:system-property( $name as xs:string )
         as item()?
```

Returns the value of a "system" or application property. Similar to the function with same name in XSLT.

Additional properties can be defined through the Java API.

Parameter $name: name of the system property.

Returned value: value of the property, or empty sequence if unknown property name.

Predefined properties:

- **vendor** : name of the vendor.
- **vendor-url** : URL of the vendor's site, here "http://www.xfra.net/qizxopen/".
- **product-name** : here "Qizx/open".
- **product-version** : the current version in string form.

2. Date, Time, Duration functions

Qizx/open does not currently implement all the functions and operators (some 20) specified in the current XML Query Working Draft for manipulation of duration types. The types xdt:yearMonthDuration and xdt:dayTimeDuration do exist in Qizx but are not really properly handled. We persist in believing that these durations types are of very little utility for real applications (in addition to their peculiar properties that make them difficult to use).

Instead, Qizx provides more useful operators and extends the semantics of date and time constructors.

Additional constructors:

These constructor allow to build date, time, dateTime, and duration objects from numeric values (this useful capability is not provided by the current XQuery specifications).

```
function xs:date( $year as xs:integer, $month as xs:integer,
                  $day as xs:integer )
         as xs:date
```

builds a date from a year, a month, and a day in integer form. The implicit timezone is used.

For example xs:date(1999, 12, 31) returns the same value as xs:date("1999-12-31").

```
function xs:time( $hour as xs:integer, $minute as xs:integer,
                  $second as xs:double )
         as xs:time
```

builds a xs:time from an hour, a minute as integer, and seconds as double. The implicit timezone is used.

```
function xs:dateTime( $year as xs:integer, $month as xs:integer,
                      $day as xs:integer, $hour as xs:integer,
                      $minute as xs:integer, $second as xs:double
                      [, $timezone as xs:double] )
         as xs:dateTime
```

builds a dateTime from the six components that constitute date and time.

A timezone can be specified: it is expressed as a signed number of hours (ranging from -14 to 14), otherwise the implicit timezone is used.

```
function xs:duration( $months as xs:integer, $seconds as xs:double )
         as xs:duration
```

Builds a general duration from a number of months and a duration in seconds. Generally used to convert a duration in seconds to a xs:duration (first argument equal to 0).

Additional arithmetic:

The current XQuery specifications have functions or operators to compute the difference between two dates or two dateTimes, unfortunately the result is a xdt:dayTimeDuration or xdt:yearMonthDuration: when one wants a numeric duration (seconds or days) - we assume that it is the most frequent case -, it is far from easy to convert from these types. Conversely, when one wants to add a numeric duration to a date or dateTime, the current specifications provide a form of the operator + (for example op:add-dayTimeDuration-to-dateTime), but the argument is also a duration, and converting from a number to a duration is even more difficult...

Therefore more convenient operators are provided:

```
operator - ($date1 as xs:date, $date2 as xs:date) as xs:integer
```

returns the difference in days between two dates. Timezones are not taken into account - It seems not to make much sense -, else the result should be decimal or double.

operator - ($date1 as xs:dateTime, $date2 as xs:dateTime) as xs:double

returns the difference in seconds between two dateTimes. Here the timezone are taken into account.

operator - ($time1 as xs:time, $time2 as xs:time) as xs:double

returns the difference in seconds between two times. The timezone are taken into account.

operator + ($date as xs:date, $days as xs:integer) as xs:date

adds days (possibly negative) to a date and returns a new date.

operator + ($dateTime as xs:dateTime, $duration as xs:double) as xs:dateTime

add seconds to a dateTime and returns a new dateTime.

operator + ($time as xs:dateTime, $duration as xs:double) as xs:time

add seconds to a time and returns a new time.

Examples:

```
xs:date("2000-01-01") - xs:date("1999-12-31")  -->  1
xs:dateTime("2000-01-01T00:00:00") - xs:dateTime("1999-12-31T23:59:59")  --> 1
xs:date("1999-12-31Z") + 1                     --> 2000-01-01Z
xs:dateTime("1999-12-31T23:59:59Z") + 1        -->   2000-01-01T00:00:00Z
```

Component extraction functions (no more difference):

The values returned by component extraction functions get-hours-from-xxx are *relative* to the timezone of the date/time object or the local time). For example get-hours-from-dateTime(xs:dateTime("2003-09-23T23:55:00")) returns 23 *whatever the actual implicit timezone*.

Since the W3C specifications have been improved in this respect, Qizx/open is now in compliance without any change...

3. Java binding

This feature allows to call Java methods and to manipulate wrapped Java objects. This is very powerful as it provides access to nearly all the Java APIs.

It is similar to the mechanism provided by XT or Saxon: a call to a function `ns:fun()` where ns is bound to a namespace of the form *java:fullyQualifiedClassName* is treated as a call of the static method `fun` of the class with name *fullyQualifiedClassName*.

Hyphens in method names are removed with the character following the hyphen being upper-cased (aka 'camelCasing').

The following example calls the `getInstance()` method of class `java.util.Calendar`:

```
declare namespace cal = "java:java.util.Calendar"
cal:get-instance()
```

The mechanism is actually a little more flexible: a namespace can also refer to a package instead of a class name. The class name is passed as a prefix of the function name, separated by a dot. For example:

```
declare namespace util = "java:java.util"
util:Calendar.get-instance()
```

The following example invokes a constructor, gets a wrapped File in variable $f, then invokes the non-static method createNewFile():

```
declare namespace file = "java:java.io.File"

let $f := file:new("myfile")
return file:createNewFile($f)     (: or create-new-file() :)
```

This example lists the files of the current directory with their sizes :

```
declare namespace file = "java:java.io.File"

for $f in file:listFiles( file:new(".") )       (: or list-files() :)
return
    <file name="{ $f }" size="{ file:length($f) }"/>
```

Notice that the File array returned by File.listFiles is conveniently converted into a XQuery sequence. This works also with Vector, Enumeration and ArrayList.

Java-bound functions can return objects of arbitrary classes which can then be passed as arguments to other extension functions or stored in variables. The type of such objects is xdt:object (formerly named xs:wrappedObject). It is always possible to get the string value of a Java object [invokes the Java method `toString()`.]

Static and non-static methods: A non-static method is treated like a static method with an additional first argument (this). The additional actual argument must of course match the class of the method.

Constructors: A call to a function named **new** invokes a constructor. Overloading is allowed on constructors in the same way as on regular methods.

3.1. Arguments

Function arguments are automatically converted from XQuery to Java and conversely, when possible. The following section lists the performed conversions.

Calling overloaded Java methods is generally correctly handled, but there is currently limitation if several methods match the actual argument types: in that case which method is actually called is unpredictable. For example, consider the Java method java.lang.Math.abs: it has 4 different instances for int, long, float and double. If we call the bound function math:abs like this:

```
declare namespace math="java:java.lang.Math"
math:abs(-1)
```

then the exact type of the result is not guaranteed: it can be xs:integer (long), xs:float or xs:double, because the type of the argument (long) can match any of the three corresponding Java methods.

3.2. Type conversions

The following conversions apply to arguments passed to Java methods, and conversely to returned values.

Notice that arrays, as well as Vector, Enumeration and ArrayList are handled as XQuery sequences.

Table 3. Types conversions

Java type	XML Query type
void (return type)	empty()
String	xs:string
boolean, Boolean	xs:boolean
double, Double	xs:double
float, Float	xs:float
long, Long	xs:integer
int, Integer	xs:int
short, Short	xs:short
byte, Byte	xs:byte
char, Char	xs:integer
net.xfra.qizxopen.xquery.dm.Node	node()
other class	xdt:object
String[ ]	xs:string *
double[ ], float[ ]	xs:double *
long[ ], int[ ], short[ ], byte[ ], char[ ]	xs:integer *
net.xfra.qizxopen.xquery.dm.Node[ ]	node()*
other array	xdt:object *
java.util.Enumeration, java.util.Vector, java.util.ArrayList	xdt:object *

4. SQL Connection

The "SQL Connection" is an extension which allows to query data from relational databases, using SQL, and transform it on-the-fly into XML, providing an easy way to merge relational data into XML documents.

This is a simple implementation, mainly based on JDBC (Java Database Connectivity) and the Java binding mechanism, which does not attempt to compile XQuery into SQL.

We assume in this section that the reader has some knowledge of SQL and JDBC.

Basically, the SQL Connection provides functions which take a SQL query statement as argument and return the result set as a sequence of "row" XML elements. These elements have sub-elements corresponding with the columns/fields returned by the SQL query.

A simple example: we suppose the existence of a MySQL database "db1" with a table "pets" containing the description of a few animals. Here is a XQuery expression which opens a connection to the database and queries the table:

```
let $conn := sqlx:get-connection("jdbc:mysql://localhost/db1", "user", "password")
return
     sqlx:execute($conn, "SELECT name, species, weight FROM pets")
```

Returning results like:

```
<row>
    <name>Albert</name>
    <species>cat</species>
    <weight>2.5</weight>
```

```
</row>
<row>
    <name>Bertha</name>
    <species>boa constrictor</species>
    <weight>45</weight>
</row>
```

The function has returned two row elements each containing the three sub-elements name, species, and weight for the corresponding fields used in the SQL query.

You may also want to produce a HTML table with this data:

```
let $conn := sqlx:get-connection("jdbc:mysql://localhost/db1", "user", "password")
return
  <table width="80%">
   <tr><th>Name</th><th>Species</th><th>Weight</th></tr>
  {
   for $row in sqlx:execute($conn, "SELECT name, species, weight FROM pets")
   return <tr>
            <td>{ $row/name/text() }</td>
            <td>{ $row/species/text() }</td>
            <td>{ $row/weight/text() }</td>
          </tr>
  }
 </table>
```

Notes:

- To run the examples above it is assumed that your JRE has access to a JDBC driver for MySQL (or any other JDBC enabled database). For this you must have the jar of the driver in your class-path and you must register the driver (either by using the system property jdbc.drivers or by explicitly registering the driver with the helper function sqlx:register-driver() -- see below).
- The namespace prefix **sqlx** is predefined: it is used for the functions of the SQL extension. Some other namespaces are predefined for convenient access to the main JDBC classes or interfaces: **sqlc** for java.sql.Connection, **sqlr** for java.sql.ResultSet, **sqlp** for java.sql.PreparedStatement.
- The object returned by sqlx:get-connection is simply the Connection object of JDBC, therefore all its methods are accessible through the Java binding mechanism. For example, `sqlc:commit($conn)` would call the commit method of interface java.sql.Connection (remember that **sqlc:** is a predefined namespace prefix for java.sql.Connection).

  As a consequence the connection can be obtained by other means, for example using JDBC DataSources, of course still using the Java binding mechanism.

4.1. Prepared and callable statements

JDBC has a notion of Prepared or Callable Statement, which is a precompiled query or update instruction with place-holders for parameterized values.

Qizx's SQL Connection API makes easy to pass actual parameters to such statements:

```
declare variable $QUERY := text {
         SELECT * from pets WHERE name = ?
};

let $conn := sqlx:get-connection("jdbc:mysql://localhost/db1", "user", "password")
    (: prepare a statement for later execution :)
    $prStatement := sqlx:prepare( $conn, $QUERY )
return
     sqlx:execute($prStatement, "Albert")
```

This would return the first row matching the parameter value "Albert" for the field "name":

```
<row>
    <name>Albert</name>
    <species>cat</species>
    <weight>2.5</weight>
</row>
```

More generally, several parameters can be passed this way in the same call to `sqlx:execute` (this function accepts a variable number of arguments). The parameter types must be compatible with the expected field types, otherwise an error is raised.

To illustrate this, let's introduce another function: `sqlx:execUpdate` (the equivalent of the `JDBC executeUpdate` method of PreparedStatement):

```
declare variable $STATEMENT := text {
         INSERT INTO pets VALUES (?, ?, ?)
};
let (: prepare a statement for later execution :)
    $insert := sqlx:prepare( $conn, $STATEMENT )
return (
     sqlx:execUpdate($insert, "Donald", "duck", 4),
     sqlx:execUpdate($insert, "Pooh", "bear", ())
   )
```

This piece of code inserts two new rows into the pets table.

- sqlx:execUpdate returns an integer which is the number of rows affected.
- Each parameter must evaluate as a single item or an empty sequence.
- The empty sequence stands for a NULL field value.

Notice the useful trick of typing SQL inside a text constructor: this saves escaping of quote characters.

4.2. Other query invocation style

The function `sqlx:execute` provides an easy way to perform the fusion of tabular data by manipulating SQL query results as XML elements. However a finer and maybe more efficient style may also be desirable.

This is achieved by the function `sqlx:rawExec` : it allows to deal directly with the ResultSet interface and retrieve field values as typed items, instead of plain text.

The HTML example above could be changed like this, converting a weight from kilograms to pounds:

```
let $conn := sqlx:get-connection("jdbc:mysql://localhost/db1", "user", "password")
return
  <table width="80%">
   <tr><th>Name</th><th>Species</th><th>Weight</th></tr>
  {
   for $r in sqlx:rawExec($conn, "SELECT name, species, weight FROM pets")
   return <tr>
            <td>{ sqlr:getString($r, 1) }</td>
            <td>{ sqlr:getString($r, "species") }</td>
            <td>{ sqlr:getDouble($r, "weight") / 0.463 }</td>
          </tr>
  }
 </table>
```

Notes:

- sqlr:getString for example is a binding of the method ResultSet.getString(int columnIndex). The prefix sqlr is predefined for ResultSet.

- The "get" methods can of course be called either with an integer columnIndex or a String column name.
- The "update" methods can be called.
- Methods which move the cursor of the result set (first, beforeFirst) will most probably have undesirable effects.

4.3. Function Reference

```
function sqlx:get-connection( $dbURL as xs:string,
                              $user as xs:string, $passwd as xs:string)
    as xdt:object[Connection]
```

A thin wrapper on the DriverManager.getConnection method.

Parameter $dbURI: a database url accepted by JDBC.

Parameter $user: database user name.

Parameter $user: database user password.

Returned value: a wrapped Connection object. If no connection can be opened, an error is raised.

Note: the connection can be closed by `sqlc:close($conn)`. There is no specific function for this purpose.

```
function sqlx:prepare( $connection as xdt:object, $statement as xs:string )
    as xdt:object[PreparedStatement]
```

A thin wrapper on the Connection.prepareStatement(String) method.

Parameter $connection: a valid database connection.

Parameter $statement: a SQL statement.

Returned value: a wrapped PreparedStatement object. An error can be raised on an erroneous statement.

```
function sqlx:execute( $connection as xdt:object, $statement as xs:string )
    as element()*
```

```
function sqlx:execute( $preparedStatement as xdt:object,
                       $param1 as item()?, $param2... )
    as element()*
```

Query execution, accepting simple or prepared statement.

Parameter $connection: a valid database connection.

Parameter $preparedStatement: a prepared SQL statement.

Parameters $param1 ...: values passed as parameters of the SQL statement.

Returned value: a sequence of nodes. Each node has the name "row" (without namespace) and contain as many children elements as there are fields in the query result. Each child element has the name of a queried field.

A field with a NULL value appears as an empty child element.

```
function sqlx:execUpdate( $connection as xdt:object, $statement as xs:string
 )
    as xs:integer
```

```
function sqlx:execUpdate( $preparedStatement as xdt:object,
                          $param1 as item()?, $param2... )
    as xs:integer
```

Update statement execution, accepting simple or prepared statement.

Parameter $connection: a valid database connection.

Parameter $preparedStatement: a prepared SQL statement.

Parameters $param1 ...: values passed as parameters of the SQL statement.

Returned value: an integer which is the number of rows affected.

```
function sqlx:rawExec( $connection as xdt:object, $statement as xs:string )
    as xdt:object*
```

```
function sqlx:rawExec( $preparedStatement as xdt:object,
                       $param1 as item()?, $param2... )
    as xdt:object*
```

Raw statement execution, accepting simple or prepared statement.

This function returns a wrapped ResultSet sequence. In order to iterate on this result set, *it is compulsory to use a **for** loop*: for $rs in sqlx:rawExec(...) return ...

Parameter $connection: a valid database connection.

Parameter $preparedStatement: a prepared SQL statement.

Parameters $param1 ...: values passed as parameters of the SQL statement.

Returned value: a sequence of ResultSet states.

5. Java API

This interface provides the methods for compiling and executing XQuery scripts from Java and exploiting results, thus offering a high-level query interface.

It is used in GUI or 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 to:

- setup compilation and execution environments (also known as *static context* and *dynamic context* respectively). These environments are provided by a class named XQueryProcessor.
- Compile queries
- execute queries
- exploit results of query evaluations. The result sets are iterators similar to iterators of the XML Library API, though more general since the XQuery language can return sequences of any item kinds, not merely Nodes.

Packages: To use the API, classes from the following packages may have to be imported:

Table 4. packages

net.xfra.qizxopen.xquery	This is the root package for XQuery, it contains in particular `XQueryProcessor`, `Query`, `Value`, `Item`, `Type`, `Log`.
net.xfra.qizxopen.xquery.dm	(XQuery Data Model) Can be used for lower-level operations: contains principally the `XQuery Node` interface.
net.xfra.qizxopen.dm	Data Model independent of XQuery: contains support for serialization (`XMLSerializer`) and a super-interface `Node`.
net.xfra.qizxopen.util	Utilities

5.1. Outline

The fundamental API object is **XQueryProcessor**.

XQueryProcessor provides a static environment to compile a query from text source, and a dynamic environment (in particular a Document Manager) to execute this query.

A typical Qizx application will perform the following steps:

1. Instantiate a XQueryProcessor: this can be done from scratch or by cloning a "master" XQueryProcessor that serves as a model.
2. Optionally set options or specify ancillary **Module Manager** or **Document Manager** that can be shared by several XQueryProcessors.

   A Module Manager is in charge of compiling/loading and caching library modules. A Document Manager performs document loading/parsing and optionally caching (Documents are read-only and thread-safe).
3. Compile a query from a file, a URL, a string: this requires a **Log** object which is used for printing messages. A successful compilation returns a **Query** object. The compiled Query can be used several times and in several threads.
4. Before executing a compiled Query, other options can be set in the processor, and global variables can be initialized.
5. Running a Query with the can be performed in different ways (methods `executeQuery` of XQueryProcessor):

- The simplest way is to serialize directly the result into an output stream (this implies that the result is a well-formed document). The serializer (**XMLSerializer**) supports a number of options, notably it can generate XML, HTML, XHTML markup, or plain text.

- With the same method, one can generate a tree using a **EventDrivenBuilder** (package net.xfra.qizxopen.xquery.dm). This tree can then be manipulated through the Node interface. Notice that according to the "functional" approach used in XPath/XQuery/XSLT, the tree cannot be modified once built.

- Another possibility is to generate the result into a SAX interface, for example to pass it to a XSLT processor. This is conveniently achieved by using **SAXXQueryProcessor**, a subclass of XQueryProcessor.

- A third, more general way is to obtain the results as a **Value**, i.e. an **Item** sequence and enumerate the items. Items can be Nodes or atomic values (like string, double, boolean etc.). This implies to check the types of items and extract values appropriately through a set of specialized methods. This is more complicated and generally not necessary.

5.2. Introductory Tutorial

For the details of each class, interface or method, please see the Java documentation.

1. Instantiate a XQueryProcessor, for example like this:

```
import net.xfra.qizxopen.xquery.*;
...
XQueryProcessor processor =
    new XQueryProcessor( moduleBaseURI, documentBaseURI );
```

   The parameters are base URIs (in String form) respectively for resolution of module and document relative URIs. This constructor automatically creates a private ModuleManager and a private DocumentManager.

   A XQueryProcessor can also be created from a master XQueryProcessor, inheriting the Module Manager and the Document Manager and default settings from the master. This can be convenient for server side applications to share the same resources among different clients.

```
XQueryProcessor processor = new XQueryProcessor( masterProcessor );
```

   Note: modules and queries are thread-safe and can be shared. Documents are read-only (this is implied by the XQuery Data Model) and can also be shared without difficulty. However this capability has not yet been tested extensively.

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

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

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

     This allows to use 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.

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

   - 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 method XQueryProcessor.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.

   A third parameter is a Log that is used for receiving messages. By default it writes to System.err and can be redirected to a stream. It has overridable display methods for easier subclassing (for example display in a GUI).

```
String querySource =
          " for $i in 1 to 3 return element E { attribute A { $i } } ";
Log log = new Log(); // Writes on System.err by default
try {
   Query query = processor.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, message log.

5. Executing a compiled query and exploit results:

   a. Direct serialization:

```
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...
processor.executeQuery( query, serial );
```

   b. Tree building:

```
EventDrivenBuilder builder = new EventDrivenBuilder();
processor.executeQuery( query, builder );
Node result = builder.crop();
```

c. SAX output: SAXXQueryProcessor implements the interface org.xml.sax.XMLReader and can therefore be used to build a SAXSource for use with APIS javax.xml.transform: for example pipe a XQuery execution with a XSLT transformation.

d. Get a Value and enumerate Items:

```
Value v = processor.executeQuery( query );
while(v.next()) // When next() returns true, an item is available
{
    if(v.isNode()) {
        Node n = v.asNode();
        ... // use the Node (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.asDouble();
        }
        ... // use the different asX() 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 = processor.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).

5.3. 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 does not keep track of physical features like entity boundaries, marked sections, characters references.
- It defines neither language bindings, nor updating operations,
- It supports XML Schema types and the notion of collections.

In XQuest the XML Data Model is seen mainly through the **`Node`** interface (`net.xfra.qizxopen.dm.Node`). It supports the *accessors* defined in the Data Model specifications plus extensions.

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

The utility package `net.xfra.qizxopen.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.

Basic information access:

We present here some of the basic accessors:

`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 ELEMENT, DOCUMENT, TEXT, COMMENT etc.

`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.

`NodeSequence children()`

For documents and elements, returns a *NodeSequence*, an abstract iiterator 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(">");
}
```

Extended accessors:

XQuest has extended methods which return sequences filtered by an abstract `NodeTest`. A most useful implementation of `NodeTest` is `BaseNodeTest` which can filter nodes according to their kind and their name. It can also perform wildcard name matching.

`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 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 BaseNodeTest( Node.ATTRIBUTE, 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.

Comparisons
There are methods for comparing the value or document order of two nodes:

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

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");
```

Then a node can be serialized:

```
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.

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 needs not 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.