Qizx Programmer's Guide

Table of Contents

Qizx 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 4 (Java API[1]) ; see also the Javadoc[2].

For XQuery programming, Qizx provides different kinds of extensions:

XQuery functions
: Additional predefined functions. They belong to a private namespace "qizx.extensions" referenced by the **x:** or **qizx:** prefixes.

  They implement in particular serialization, error handling, text searching and highlighting.

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 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.

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[3].

1. Extension functions

These functions belong to the namespace "qizx.extensions" for which the prefixes `x:` and `qizx:` are predefined.

`x:serialize( $tree as element(), $options as element(option) ) as xs:string?`
: Serializes the tree element into marked-up text. The value returned is the path of the output file, or the empty sequence if the default output is used.

[1] #javapi
[2] javadoc/index.html
[3] doc_xqsp.html

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.

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

There is 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.

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.

Remark: a better mechanism is desirable in order to retrieve the cause of the error.

`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.

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

This function (also named `x:fulltext`) implements context-sensitive full-text search: it can search boolean combinations of words, word patterns and phrases. 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>
```

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

This function is a companion of 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>`.

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).

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").

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.

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])
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.

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
```

Difference on component extraction functions:

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

The W3C specifications require a UTC return value, but this is rather disconcerting: the expression above would return diverse values when executed in different timezones.

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()
```

Overloading based on number and type of parameters is allowed, with the current limitation that if several methods match the actual argument types, which method is actually called is unpredictable.

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.

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

Extension 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()`.]

The following conversions are performed on arguments and conversely on returned values:

Table 2. 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.ArrayL-ist	xdt:object *

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) }"/>
```

4. Java API

This section explains how to integrate Qizx in applications. Implementing new predefined functions is beyond the scope of this document.

An advanced example application is the provided by the command line tool net.xfra.qizx.app.XQuery (it is recommended to read the source code). The "Server Pages" extension, which embeds the engine in a Servlet, is also an advanced application that provides an example of connection with SAX.

The API allows to:

- setup compilation and execution environments (also known as *static context* and *dynamic context* respectively).
- compile queries
- execute queries
- manipulate results of query evaluations

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

Table 3. 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

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.

Step-by-step illustration:

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 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 and is generally not needed.

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).