
Advancing with XQuery: Develop application idioms

Work with extension functions, unit tests and assertions, recursion and sorting, and higher-order functions

Skill Level: Intermediate

[James R. Fuller \(jim.fuller@webcomposite.com\)](mailto:jim.fuller@webcomposite.com)
Technical Director
FlameDigital Limited & Webcomposite s.r.o.

30 Sep 2008

The XQuery specification is well over a year old now. A surfeit of solid implementations combined with (if developer chatter is anything to go by) marked new interest, seems to indicate that XQuery is finally experiencing higher adoption rates. Possibly this is due to developers starting to figure out how to utilize XQuery within a rich mixture of XML technologies (such as XML databases, XSLT, XML Schema). Learn how to use XQuery beyond its original role as an XML query language and apply it toward the development of middleware and Web applications.

Section 1. Before you start

Before you examine XQuery code samples, here's how to get the most of this tutorial, and instructions on how to install and use the included source code (see [Downloads](#)).

About this tutorial

This tutorial is about using XQuery to develop applications and middleware. It outlines some of XQuery's limitations while you develop applications, gives you

practical advice along the way on how to manage these limitations as well as highlights where XQuery makes it easy or difficult. The bulk of the tutorial then builds on these principles, as it presents a series of programmatic idioms commonly found in application development:

- Use of extension functions
- Unit testing and assertions
- Recursion and sorting
- Higher-order functions

Each section comes with accompanying source code examples (see [Downloads](#)).

Objectives

The goal of this tutorial is to give you a practical grounding in how to develop applications with XQuery. I try to provide cut-and-paste code that you can re-use in your own application-development efforts, though it is my hope that these code examples will inspire you to reconsider using XQuery for more than querying XML.

Prerequisites

This tutorial is written for developers who have a general familiarity with XML technologies as well as some cursory XSLT or Query experience. The programmatic idioms presented can be found in one form or another in a lot of computer programming languages, and I make no claim of invention on them. This re-use should mean that for most readers, you'll see familiar constructions—albeit set in the context of XQuery programming.

System requirements

You must install Michael Kay's [Saxon XSLT and XQuery processor SA aware version](#) to execute code examples. As you need the Saxon-SA version, you must register for a 30-day trial (for example, at the time of this writing version 9.1SA was used in testing). Many of the code examples use higher-order functions to take advantage of specific Saxon-SA extension functions (specifically, the `saxon:function()`).

Place all the Java™ Archive (JAR) files and license file in Saxon under the `/lib` directory.

To run examples you can invoke Saxon from the command line or use the included

Ant build file (which I used to test the code examples). If you do use the build file, then you should also install the latest Apache Ant, and take care to amend the `saxon.lib.dir` property to point to the `/lib` directory containing Saxon JARs. To test whether Saxon is working properly, run the Ant target `checkSaxon`, which will successfully process when Saxon is installed properly. All Ant targets generate result output into the `/result` directory.

Section 2. Today's XQuery

Now let's discuss some of the past concerns with XQuery and where it is today.

The XQuery specification

By *XQuery specification*, I mean a group of interrelated documents:

- **XQuery 1.0 (an XML query language):** Defines core language
- **XQuery 1.0 and XPath 2.0 data model:** Defines the data model that XPath and XQuery share
- **XQuery 1.0 and XPath 2.0 formal semantics:** Provides mathematical foundation
- **XQuery 1.0 and XPath 2.0 functions and operators:** Define functions that XPath and XQuery share
- **XSLT 2.0 and XQuery 1.0 serialization:** Defines how XML is constructed and outputted from XQuery

Some people also find the XML Query use case document useful, as it presents each usage scenario with an accompanying XQuery solution (see [Resources](#) for links to all specifications). Even more specifications and drafts deal with other capabilities, such as updating XML or full text search.

Frequently used acronyms

- API: Application programming interface
- EXSLT: Extensions for XSLT.
- URI: Uniform Resource Identifier
- W3C: World Wide Web Consortium
- XML: Extensible Markup Language

- XSLT: XSL Transformations

You are not alone if you ask "Why so many specifications?" It's difficult to understand why the W3C Query Working Group generated so many interrelated documents. I'm not against complexity if it's warranted, but I do find myself wondering how XSLT 1.0 (see [Resources](#)) was able to do all it did effectively in one (arguably two specifications—as there is the related XPath 1.0 specification, as well).

During the creation of the XQuery 1.0 specification, a few permatopics persistently trailed XQuery. Many of these topics represented (to me at the time) extreme limitations that I was convinced would hobble XQuery (most of which, after a year of heavy-duty XQuery development, have yet to materialize into anything more than irrelevancy). Acknowledging one's own failures is always a learning experience, so I think it would be useful to revisit some of these issues.

Permatopic 1: XSLT versus XQuery

As an active XSLT programmer, I was concerned about overlap between the two technologies. XSLT and XQuery are syntactically different but share a lot in common—for example, the languages both:

- Take XML as their input and produce XML as their output
- Have the same data model and type system
- Are declarative and meant to be free from side effects
- Share XPath, having the same library of built-in functions

XQuery syntax

Here are a few examples of where XQuery syntax can be a bit bothersome:

- XQuery uses curly braces (`{ }`) in many situations, whereas XSLT uses them solely for Attribute Value Template (AVT).
- A minus sign (`-`) needs spaces. For example, `$t-1` is a valid variable name and should be `$t - 1`.
- Code comments in XQuery (`((: :))`) make me smile, but I see little need in defining yet another new way to annotate source code.

Differences between XSLT and XQuery plainly exist, and (to score a point for the XSLT crowd) it's also accurate to say that XQuery represents a subset of XSLT 2.0 functionality. For example, XQuery has no concept of dynamic binding, whereas

XSLT provides dynamic matching through template matching rules. XSLT also supplies a form of polymorphism, with the ability to override template matching rules with `xsl:import`. XQuery has no such facility.

After using XQuery for a bit, I found that some jobs are better suited to one technology or the other:

- XQuery is unsurprisingly good at querying XML. I find it useful to get the smallest slice of data, then hand off to XSLT for formatting and presentation.
- XQuery is good at controlling processes (for example, XSLT), especially in the context of XML databases.
- XSLT wins out whenever you need to format numbers.
- XQuery is easier to explain and has a shallower learning curve.
- Walking trees in XQuery is cumbersome. Using XSLT template matches can be more appropriate.

Teaching XSLT versus teaching XQuery

In a [blog entry](#) by Dan McCreary (Opinion, June 2008), read why an experienced SQL developer might find it easier to learn XQuery than XSLT.

It is arguably a magnitude [easier to impose](#) XQuery onto those database administrators (DBAs) who have grumbled for the past decade about the current XML infection in their systems or why they have to shred XML into their relational tables.

Note: I recommend that you check out Benoit Marchal's article, "Comparing XSLT 2.0 and XQuery" (see [Resources](#)), for a fuller overview of the differences between XQuery and XSLT.

Permatopic 2: XML Schema and datatypes

The W3C XQuery Working Group's provocative linking of XML Schema datatypes (see [Resources](#)) within XQuery was seen (by me at least) as a cynical attempt to create a dependency on a technology that had failed to convince the entire XML community that it was the best approach to constrain and validate XML. It was a bit of a shock for me, then, to find that XQuery implementations, on the whole, do a pretty good job at letting datatypes stay well out of your way if you don't want to use them.

I won't rehash the arguments or reasons why XML Schema (see [Resources](#)) or any schema technology is bad or good. Maintenance and an ability to refactor quickly will help you decide how you use schemas or datatypes in your XQuery applications.

Here are a few informal rules I follow when I consider whether to use them:

- If you do not use schemas, you might consider primitive datatypes in functional signatures.
- If you use XML Schema and datatypes, consider auto-generating code (stubs and so on) to ease maintenance.
- You can typically ignore schemas and datatypes and add them later, although you lose some of the benefits (catching errors). I would recommend this approach for your first forays into XQuery.

Past that, if you are uncomfortable with the idea of being locked into a particular schema technology, feel free to ignore their use in XQuery.

Permatopic 3: Strong versus weak typing

The question of whether or not to statically or dynamically type data is a permatopic that runs throughout the larger programmer community. XQuery just becomes another (un)willing victim in this long-running sometimes irrelevant debate.

As I just alluded in the previous permatopic, XML Schema datatypes are opt in. It's possible to completely ignore their use in XQuery, which means that the strong-versus-weak or static-versus-dynamic typing argument really never needs to be debated: Both approaches are effectively and mutually enabled for you to use or not use.

Perhaps the question now becomes, "Should you use datatyping?" Datatyping changes the way you program, so I think it's useful to outline how I use it and why I came to the conclusion that using datatypes sparingly can be a good thing.

I find that my just enough level is to datatype the input arguments and return values of a function, as in [Listing 1](#).

Listing 1. local:add function

```
xquery version "1.0";

declare function local:add($a as xs:integer, $b as xs:integer) as xs:integer {
  xs:integer($a+$b)
};

document {
  <result>{
    local:add(1,2)
  }
}</result>
}
```

If you run the `exampleSimple` Ant target, you get the result printed out to the

console as well as saved to a file under the /result directory, as in [Listing 2](#).

Listing 2. Result of processing example.xq

```
Buildfile: src/build.xml
example:
[echo] Source document ignored - query does not access the context item
[echo] <?xml version="1.0" encoding="UTF-8"?>
[echo] <result>3</result>

BUILD SUCCESSFUL
Total time: 1 second
```

Apart from the comfort of knowing that addition works in XQuery, you can see that I constrained the inputs and return values of the `local:add` function to be an XML Schema datatype of `xs:integer`. If I supply the function with the wrong inputs—for example, `local:add(1, "2")` (with the 2 supplied as a string)—I get useful type error information. Datatypes aside, XQuery will still generate an error, but this is related to XQuery being smart enough to know that the XQuery plus (+) operator works with numbers.

Static type analysis means that all possible type errors (I'm mainly talking about mismatches in datatypes) will occur at the time code is analyzed versus evaluation, which means that any external XQuery code that invokes the `local:add` function will need to pass static analysis before it runs. Through this kind of pessimistic static checking, datotyping can enhance the overall quality of applications, as you must ensure that datatypes match before you consider deploying your code.

To summarize, even if your XML has no defined datatypes or related schemas, you gain some benefit to using primitive datatypes. Also, you position your application to take advantage of XML Schema datatypes in the future.

Permatopic 4: No update mechanisms

When XQuery was initially conceived, no features were planned to specifically address the updating of XML documents (in a file system, database, or otherwise). As of March 2008, the Candidate Recommendation of XQuery Update Facility (XQF—see [Resources](#)) remedies this weakness.

The W3C XQuery Working Group probably had little time to consider adding this set of features to the core of XQuery. Its omission at that time meant that the XQuery Update Facility specification could be changed through feedback from using live implementations of XQuery processors. This kind of real-world experience is invaluable when creating specifications, and I am sure XQF is better-specified for it.

XQuery impediments and drawbacks

Even with most of the permatopics dispelled, I still found plenty of issues to discuss with the current XQuery language.

Lacking a system-property() function

XSLT has a useful function that queries the processor about itself, returning properties like processor vendor or which version of XSLT is in effect. This kind of introspection is valuable if one is to create XQuery applications that perform across XQuery processors. All I can say is that XQuery does not have one, or—perhaps more accurately—XPath 2.0 does not have anything which performs this function and is usable by XQuery.

Warning: Using extension functions can cause lock-in

Most XQuery implementations have added their own third-party functions, providing all manner of additional capabilities. The obvious issue with using such extension functions is that you make your XQuery code potentially incompatible if you depend on a specific, non-standard functionality exposed by a specific implementation.

I fought this fight against vendor lock-in before in assisting with the [EXSLT](#) effort. I am not surprised that it comes up again in XQuery.

The best way to illustrate the deleterious effects on development that extension functions can have is with an elementary case. I provide a survey of the innocuous `random()` function's implementation across three XQuery implementations (see [Resources](#)) as no small proof of how complicated things can become:

- eXist XML database:
 - `util:random()` `xs:double`
 - `util:random($a as xs:integer)` `xs:integer`
 - `math:random()` `xs:double`
- MarkLogic XML database:
 - `xdmp:random([$max as xs:unsignedLong]) as xs:unsignedLong`
- Saxon XQuery and XSLT processor:
 - `math:random()`: Returns a somewhat random number between 0 and 1
 - `random:random-sequence(1, $seed)`: Returns a set of random

numbers based on an initial seed

With six potentially different `random()` functions, you begin to see how difficult it is to write compatible XQuery code when this much variability exists in extension function libraries to generate a random number.

The three eXist XML database functions offer perhaps the least-optimal situation, as the relationship is unclear between `util:random` and `math:random`. I presume that this is concomitant with some refactoring or (common in open source development) to two functions created by two individuals. Worst, these two functions might represent some kind of versioning between old and new.

To cap it all, eXist's `util:random` function adds more diversity by accepting an `xs:integer` seed argument and returning an `xs:integer` (rather than the `xs:double` that the other two functions return).

MarkLogic defines a single `random($max)` function that returns a random, unsigned integer between 0 and a number up to 64 bits long. Its return type, being an `xs:unsignedLong`, represents an unsigned integer between 0 and 18446744073709551615 (inclusive). The function optionally defines a maximum value by supplying a `$max` argument of the same type.

Saxon opts to support EXSLT which should be a good thing (see EXSLT in [Resources](#)). However, because EXSLT currently has no context with XQuery, using the Saxon `random()` functions is somewhat confusing. The Saxon `random()` functions might be the best example of the considerably confusing situation across XQuery processor implementations of the `random()` function.

This combination of dissimilar and multiple namespaces with different functional signatures makes it impossible to create cross-platform run time XQuery applications, as most XQuery processors will choke during static analysis.

Default namespace weirdness

XQuery allows you to define the default namespace, so that all un-prefixed output elements appear in the default namespace. With this facility, you can define the default namespace of all un-prefixed elements in XQuery XML output, as [Listing 3](#) shows.

Listing 3. Declaring a default element namespace

```
xquery version "1.0";
declare default element namespace "http://www.w3.org/1999/xhtml";

<html>
  <body>
    test
  </body>
</html>
```

Run the example for this code, and you get well-formed output (see [Listing 4](#), in which all the elements live in the `http://www.w3.org/1999/xhtml` namespace. Otherwise, you might opt to bind the namespace to a prefix, although this can make life difficult if you are using document type definition (DTD), which requires un-prefixed elements.

Listing 4. Output of no-namespace.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
  <body>test</body>
</html>
```

So, if you do declare a default namespace, you create a few problems. First, how can you have un-prefixed result elements exist with no namespace? Strangely enough, XQuery allows you to bind a prefix to an empty string to indicate placing an element into the default element namespace. [Listing 5](#) shows how to do this.

Listing 5. No namespace binding

```
declare namespace my-no-namespace = ""
```

Now, `my-no-namespace:myelement` has no namespace.

Difficulties creep in when you need elements, within no namespace, to have no prefixes, yet still use the defined default namespace. This is impossible to manage within XQuery, which means that you'll have to resort to some text manipulation after processing with XQuery to remove prefixes and these faux namespace declarations.

Thankfully, all these idiosyncratic concepts are consistently applied and implemented across XQuery processors. It remains to be seen if the use case for it is strong enough to warrant its use and strangeness.

No XQuery evaluation or dynamic XPath

In XSLT, it is impossible to generate an XPath dynamically and then evaluate it without the use of a third-party function. The same limitation applies in XQuery, that is, no evaluation of dynamically generated XPath. Additionally, there is no capacity for dynamically evaluating generated XQuery.

XQuery's lack of an `eval()`-type function with which to execute generated XQuery code makes some idioms difficult to implement. Fortunately, most mature XQuery processors have added their own `eval()` function. In well-architected and written code, you should not have to use this function, although it does come in handy from

time to time.

You might be puzzled at this moment. All I will say is that dynamically evaluating code in any computing context requires the experience to know when to use it and the discipline to ignore it.

Exception-handling mechanisms

XQuery lets you arbitrarily throw an error with the `fn:error()` function but does not provide the facility to detect a static or dynamic error and handle it. I would say that the `fn:error()` method is roughly equivalent to a `throw`. Some XQuery processors provide an extension function like `try/catch`, but, in general, these extension functions are probably best avoided.

Too many optional features

For an XQuery implementation to be compliant, it must provide a core set of features. In addition to the core feature set, an implementor might decide to provide optional features, some of which I have listed below:

- **Schema import:** The schema imports directly through the prolog
- **Schema validation:** Validate XML with schema technologies
- **Static type analysis:** Detection of all static type errors in the analysis phase
- **Module:** Support for library modules and module imports
- **Serialization:** Serialize query results to an XML document
- **Axes:** Support for ancestor, ancestor-or-self, following, following-sibling, preceding, and preceding-sibling axes

Making schema import and processing optional was a clever decision and avoided a lot of potential friction. It also had a beneficial side effect in that no dependency is built into the core of XQuery. Making static analysis optional was a similarly good idea.

Modules in XQuery are the basis to create reusable code libraries, which makes it hard to understand how they can be optional. Another similarly difficult decision to comprehend is why developers might not want control over how XQuery outputs XML; there is a precedent, as XSLT defines this serialization as standard (see [Resources](#)).

There are inconsistencies with serialization—for example, the result of a query is typically a sequence of items, not a single XML document. But most processors use XML output by default. Take the example in [Listing 6](#), which is known as a *quine*. In

programming, a quine is a bit of code that when executed, outputs its own code listing perfectly. Quines are a useful test for highlighting limitations in any programming language and I put XQuery to the test in Listing 6.

Listing 6. Quine with XQuery

```
xquery version "1.0";

declare variable $s:='xquery version "1.0";
declare variable $s:="";fn:substring($s,1,44),$s,fn:substring($s,45)';

fn:substring($s, 1,44),
$s,
fn:substring($s,45)
```

When you execute the `exampleQuine` Ant target, it should output the code in [Listing 7](#).

Listing 7. Result of processing the quine with XQuery

```
<?xml version="1.0" encoding="UTF-8"?>xquery version "1.0";
declare variable $s:=" xquery version "1.0";
declare variable $s:="";fn:substring($s,1,44),$s,fn:substring($s,45)
";fn:substring($s,1,44),$s,fn:substring($s,45)
```

Because of whitespace issues, the only problem with this code is the need to use a processor-specific option to control output of version/encoding—not very nice, that. And, it highlights what I mean by considering serialization part of the XQuery core. The more serious point to make is that most XQuery processes default to using XML output serialization, even though the result of the above query is a sequence of strings!

Moving onto the last listed optional feature, the idea of XQuery optionally supporting *some* XPath axes is near scandalous. Initially, it was thought that these axes were too difficult to implement, but it was subsequently proven that these optional axes can be represented by the other supported axes.

Most XQuery processors implement these optional axes, so it should not pose much of a problem.

Leaving it to the implementor

I just talked about a lot of optional features that you might decide to implement. In addition, some optional features are left undefined, so you have to further define the functionality as you see fit. Allowing implementers to define *how* to implement a feature can lead to lots of variability. I list some of these features here:

- Support either XML 1.0 and Namespaces 1.0 or XML 1.1 and

Namespaces 1.1

- Implement extension functions
- Pre-declare namespaces, including default element and function namespaces
- Built-in schemas, whose type names can be used in queries and whose element and attribute declarations can be used in validation
- Built-in global variables and options along with default values
- Specify a list of supported collations
- Default values for things like the Base-URI of the static context, boundary-space policy, empty-order specification, copy-namespaces mode, construction mode, default collation, and ordering mode
- Default values for serialization options
- What an external variable defines

You might recognize most of the listed optional features. However, giving implementors the freedom to choose how to create software versus ensuring compatibility across processors is difficult with the best of intentions. I think that in the case of XQuery, we may have identified far too many optional features that will just end up promulgating incompatibilities.

Enough of praising and critiquing XQuery. Now that you've an overview of what I consider impediments to development in XQuery, let's move on to a series of programmatic idioms and run code samples to get you started.

Section 3. Using extension functions

Discover an approach to using extension functions in your own code.

If you only had a `system-property()` function

If you had a `system-property()` function, it would be relatively easy to implement XQuery function libraries made up of XQuery processor specific functions. The pseudo-code in [Listing 8](#) shows how you might wrap up the various implementations and switch between all the different vendor implementations of the `random()` function, which you learned about in the [previous section](#).

Listing 8. util:random pseudo-code

```
declare function util:random($seed) as xs:double {
  let $vendor := example:system-property('vendor'),
  if ($vendor='saxon') then
    random:random-sequence(1,$seed)
  else if ($vendor='exist') then
    exist-util:random()
  else if ($vendor='marklogic') then
    marklogic-util:random($seed)
};
```

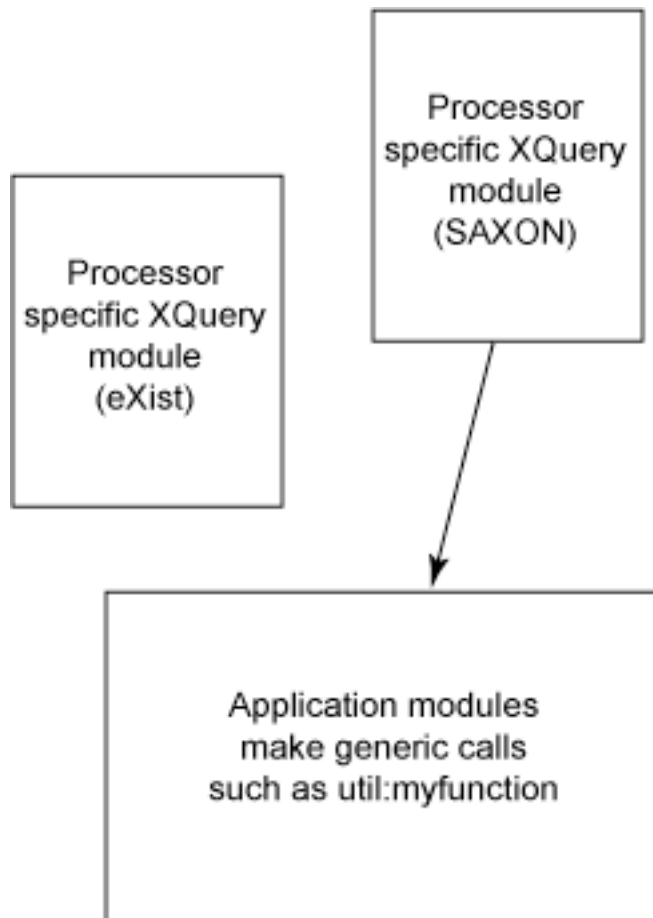
As you saw earlier in the tutorial, if you wrote this kind of XQuery code, the static analysis that most XQuery processors perform invariably throws a static error when the processor discovers that only one of the three `random()` functions actually exists for any single processor executing the code.

Working around the lack of a `system-property()` function

Because you don't have a `example:system-property()` function, you rely on some other awkward and non-standard way to emulate this functionality. You might try to configure namespaces to mock up functions, but this approach quickly becomes unmanageable. The only real way to address the creation of compatible XQuery modules is to take advantage of module imports, segregate your code into modules specific to various XQuery implementations, and have application modules call a generic version of the function.

The diagram in [Figure 1](#) shows how to use one of the processor-specific modules—in this case, the Saxon module.

Figure 1. Module specific importing example



The `util.xqm` module, in [Listing 9](#), implements the `util:random` function.

Listing 9. The `util.xqm` module

```

xquery version "1.0";
module namespace util="http://www.webcomposite.com/util";
declare namespace random="http://exslt.org/random";

declare function util:random($seed) as xs:double {
  random:random-sequence(1,$seed)
};
  
```

Application-specific code then imports the processor-specific `util` module. This just means that you have to change the referenced file in the `import module` statement, as in [Listing 10](#).

Listing 10. `random-example.xq`

```

xquery version "1.0";

(: Module Imports :)
import module namespace util = "http://www.webcomposite.com/util"
  
```

```
    at "util.xqm";

document {
  <example>{
    util:random(78991347)
  }
  </example>
}
```

You then need to provide a processor-specific `util.xqm` file for each XQuery processor that you want to support; just which one your application module imports is a matter of configuration. The only rule here is that as long as the application code calls generic functions, all is well.

I am not overly satisfied with this technique, but it is a pragmatic step toward making reasonably compatible XQuery applications. To see the result of this example, run the Ant target `exampleRandom`. [Listing 11](#) shows the output.

Listing 11. random-example.xq output

```
<?xml version="1.0" encoding="UTF-8"?>
<example>0.3479438098899311</example>
```

Section 4. Unit testing and assertions

I feel a bit lost whenever I first start developing in a new programming language. This is mainly attributed to the fact that in other languages, in which I have programmed for several years, I have the equivalent of a comfortable armchair—that is, a development environment that assists me toward my goal of creating high-quality code, facilitating such things as code completion, continuous integration, logging, versioning, assertions, and unit testing.

Implementing unit testing in XQuery

Perhaps the simplest route for implementing unit testing in XQuery would be to integrate an existing Java library—something like [XMLUnit](#). Without trying to sound too parochial, I think today's programmers—myself included—are spoiled by the embarrassment of riches (APIs and libraries) that exist for most programming languages.

However, I am a firm believer that the proof of a good programming language should

be what you can do with it in its vanilla form without adding libraries. With this in mind, I decided to assign myself the task of creating a small suite of unit test functions, wielding just plain old XQuery (POXQ).

At a minimum, the unit test module must implement a basic set of tests:

- Check whether two XML documents are equivalent.
- Compare two strings to see whether they are equivalent.
- Test for negated comparisons (for example, `testStringNotEqual` and `testXMLNotEqual`).

The unit testing module included in the examples in [Download](#) comes with a few more assertions (for example, `assertXPathExists`). As with most unit tests, you'll just return Boolean values.

assertXMLEqual

Compare two XML documents, and return Boolean False or True if they match. [Listing 12](#) illustrates how I implemented the positive and negated form of the function.

Listing 12. test:assertXMLEqual and testLassertXMLNotEqual

```
declare function test:assertXMLEqual($a as item()*, $b as item()*) as xs:boolean {
    fn:deep-equal($a,$b)
};
declare function test:assertXMLNotEqual($a as item()*, $b as item()*) as xs:boolean {
    fn:not(fn:deep-equal($a,$b))
};
```

These test functions take advantage of the built-in `fn:deep-equal()` function in XPath 2.0.

Note: It's been kindly pointed out to me that these functions can check equality of datatypes as well

assertStringEqual

Next, compare strings, and get 0 or 1 if they match. [Listing 13](#) shows this code.

Listing 13. test:assertStringEqual

```
declare function test:assertStringEqual($a as xs:string, $b as xs:string) as xs:boolean {
    fn:not(
        fn:boolean(
            fn:compare($a, $b)
        )
    )
};
```

```
);
```

Again, you leverage what existed as default in XPath 2.0 by using the `fn:compare()` function, which performs checks of equivalence between string values. The `fn:compare` function somewhat abnormally returns integers to indicate Boolean values, which is why you need to use `fn:boolean` to convert to actual Boolean types.

With these functions defined in a `unit-test.xqm` module, you can use them to create unit tests. How you lay out and develop your own XQuery test scripts depends on a couple of factors. I choose to return an XML document that encapsulates a test suite, as in [Listing 14](#).

Listing 14. unit-test-example.xq

```
xquery version "1.0" encoding "UTF-8";

(: Module Imports :)
import module namespace test = "http://www.webcomposite.com/test"
at "unit-test.xqm";

(: ----- : )
document {
  <testsuite title="some example Unit Tests" desc="">

    <test>
      <name>run successful assertXMLequal unit test</name>
      <result>
        {test:assertXMLequal(<test/>,<test/>)}
      </result>
      <expected>true</expected>
    </test>

    <test>
      <name>run failed assertXMLequal unit test</name>
      <result>
        {test:assertXMLequal(<test/>,<test><b/></test>)}
      </result>
      <expected>false</expected>
    </test>

  </testsuite>
}
```

Running the `exampleUnitTest` Ant target results in the first test returning True and the second test returning False. (This is by design.) [Listing 15](#) shows the output.

Listing 15. Result of processing unit-test-example.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<testsuite title="some example Unit Tests" desc="">
  <test>
    <name>run successful assertXMLequal unit test</name>
    <result>true</result>
```

```

    <expected>true</expected>
  </test>
  <test>
    <name>run failed assertXMLEqual unit test</name>
    <result>>false</result>
    <expected>>false</expected>
  </test>
</testsuite>

```

Transforming unit test output

The advantage of embedding unit tests in an XML document in XQuery is that it's easy to transform into some report. I have included an XSLT stylesheet that does just this. Find the stylesheet named *test.xsl* in the code download /etc directory.

In actual application testing, you would most likely need to import your own XQuery application modules, taking care to declare any application-specific namespaces.

You have probably heard the advice that unit testing your code makes it more robust and easier to maintain. The less-advertised benefit of unit testing is that it is another form of documentation, greatly speeding up the comprehension of the functionality tested. This is good for revisiting code days, weeks, months, or years later, as original issues, concerns, and primary uses are persisted in the test cases themselves.

Debug assertions

A somewhat-related aspect of unit tests, debug assertions are rather like assertions applied in test scripts, but they are inline in your application code and selectively run in debug or production mode. [Listing 16](#) shows how you do it in XQuery.

Listing 16. assertions.xq

```

xquery version "1.0";

declare namespace test="http://www.webcomposite.com/test";

(: set to 1 to enable debugging :)
declare variable $NDEBUG:=1;

declare function test:assert($booleanexp as item(), $why as xs:string) {
  if(fn:not($booleanexp) and fn:boolean($NDEBUG)) then
    fn:error(QName("http://www.webcomposite.com/err", "assertion"),$why)
  else
    ()
};

document
{
  <results>
    <assert>{test:assert(2 eq 2,'testing equality')}</assert>

```

```
</results>
}
```

The `exampleAssertion` Ant target, when executed, should successfully process `if 2 = 2`, which thankfully it does. To test whether the assertions are working, try changing the assert to say `2 eq 3` to see the behavior when XQuery throws an error.

Some people leave assertions running in both debug and production modes. To make it easy to turn assertions on or off in your code, just set the `$NDEBUG` variable to `False` or `0`, which will disable any kind of error output that the assertion throws. Note that this is a naive implementation of assertion (for example, even when disabled, it still calls the `test:assert` function, albeit with almost no processing cost).

Section 5. Recursion and sorting

Recursion is one of the most important tools you have in computing. Learn how to put it to good use within XQuery. One of the most common recursion examples that students see is how to compute the factorial of a given number using recursion.

Applying recursion

[Listing 17](#) shows how to create a recursive function in XQuery.

Listing 17. `local:factorial`

```
xquery version "1.0";

declare function local:factorial($n as xs:integer) as
xs:integer {
  if ($n < 0) then (0)
  else if ($n = 0) then (1)
  else ($n * local:factorial($n - 1))
};
```

In [Listing 18](#), you supply the `local:factorial` function with a value of 4.

Listing 18. Invoking `local:factorial`

```
document
{
  <result>
```

```
{local:factorial(4)}  
</result>  
}
```

This function tries to derive the product (left to right) of $1 * 2 * 3 * 4$, which equals 24. As with unit tests, you elect to invoke and return the value embedded within an XML document.

The interesting part of this function is not the two boundary conditions checked (for example, what to do when it encounters inputs of 0 or less than zero). The working part of the function is where the function applies a multiplication operation against the nested `local:factorial` function. This is a classic example of recursion—that is, a function calling itself. Furthermore, when a recursive function call is at the end of the function, this is known as *tail recursive* and has a series of well-known optimizations associated with it (see [Resources](#)).

In imperative languages, looping requires only a constant amount of memory. If a language lacks a looping logical statement, programmers commonly resort to recursive functions. Note that recursive functions need to allocate a bit of memory every single time they call themselves—minimally, so they know where to return. While recursion is engaged, this allocation has linearly increases memory usage over time when you compare it with imperative looping. Fortunately, most functional language implementations (with which you can group XQuery) typically determine when tail recursion is in effect; because the function is consistently invoked and returns at the end of the function, you can make an optimization with the effect of keeping memory usage constant, which puts it on par with imperative looping.

Walking the tree

[Listing 19](#) provides a more appropriate example of recursion in XQuery—manually walking an HTML document. In XQuery, it's useful to create a standard recursive function that walks the tree in a descent-first recursive. This is an attempt to provide a basic idiom that emulates XSLT template matching capabilities.

Listing 19. treewalker1.xq

```
xquery version "1.0";  
  
declare function local:treewalker ($html) {  
  
  let $children := $html/*  
  return  
    if(empty($children)) then ()  
    else  
      for $c in $children  
      return  
        ( element {name($c)}{  
          $c/@*,  
          $c/text(),  
          local:treewalker($c)}        )  
}
```

```

        local:treewalker($c)
    })
};

<test>
  {local:treewalker(<html info="test">
    <body>
      <a/>
      <b>
        <c info="test1">test</c>
      </b>
      <p>teststs</p>
    </body>
  </html>)}
</test>

```

As with the `local:factorial` example, you have a tail recursive function. The function will recurse through an XML document, outputting what it finds. The `$c/@*` outputs the XML attributes, and the `$c/text()` outputs text. You manually recreate the XML element using XQuery element `{name($c)} { ... }` construction.

Running the `exampleTreewalker1` Ant target reproduces the HTML document, as in [Listing 20](#).

Listing 20. treewalker1.xq output

```

<?xml version="1.0" encoding="UTF-8"?>
<test>
  <body>
    <a/>
    <b>
      <c info="test1">test</c>
    </b>
    <p>teststs</p>
  </body>
</test>

```

Looking at this function, you can now create tests to check for certain elements, text, or attributes and handle them accordingly. This process isn't as elegant as XSLT template matching, but one optimization you can make is to delegate the handling of attributes and text to other functions, as in [Listing 21](#).

Listing 21. treewalker2.xq

```

declare function local:attrHandler ($attr) {
  $attr
};

declare function local:textHandler ($text) {
  $text
};

declare function local:treewalker ($tree) {

```

```

let $children := $tree/*
return
if(empty($children)) then ()
else
for $c in $children
return
( element {name($c)}{
  local:attrHandler($c/@*),
  local:textHandler($c/text()),
  local:treewalker($c)
})
};

```

The change you should see in [Listing 19](#) from the first treewalker implementation is that you have created two functions (`local:textHandler` and `local:attrHandler`) and invoke them in the appropriate places.

Run the `exampleTreewalker2` Ant target which should result in the equivalent output you got for **exampleTreewalker1**. I suggest that you experiment by amending either handler function to vary outputted text and attributes.

Sorting

In XQuery, recursion is useful to implement certain processes that are difficult to implement in other ways. For example, you might want to achieve sorting that is impossible or difficult to conceive of if you used the `order by` clause in a typical FLWR expression (see "FLWR expressions" in [Resources](#)).

Here's an example of the kind of sorting that I mean. Consider the XML markup in [Listing 22](#), which describes dependencies among the parts of a house under construction. For example, you can't build the roof of a house until you build the highest floor; alternately, you can't start to build a house at all if you haven't constructed the basement, and so on.

Listing 22. sort1.xml

```

<build>
  <task id="house">
    <depends id="roof"/>
    <depends id="second-floor"/>
    <depends id="first-floor"/>
    <depends id="basement"/>
  </task>
  <task id="first-floor">
    <depends id="basement"/>
  </task>
  <task id="roof">
    <depends id="second-floor"/>
    <depends id="basement"/>
    <depends id="first-floor"/>
  </task>
  <task id="basement"/>
  <task id="second-floor">
    <depends id="basement"/>
  </task>

```

```

        <depends id="first-floor"/>
    </task>
</build>

```

How do you sort the task elements in an order that accounts for their encapsulated dependencies (for example, `depends` elements)? This kind of sorting is common in computing (see [Resources](#) for a definition of topographical sorting). This is a particularly good example of what I mean by *difficult to implement* using the `order-by` clause; I would not know where to start. I propose using a tail recursive function for the solution, as in [Listing 23](#).

Listing 23. `sorting.xq`

```

xquery version "1.0";

declare function local:my-sort($unsorted, $sorted ) {
  if (empty($unsorted)) then $sorted
  else
  let $allnodes := $unsorted [ every $id in depends/@id satisfies $id = $sorted/@id ]
  return
  if ($allnodes) then
  local:my-sort( $unsorted except $allnodes, ($sorted, $allnodes ) )
  else ()
};

document {
  <construction>{
    local:my-sort( //task, ())
  }
</construction>
}

```

The function first tests whether `$unsorted` has any more `<task/>` elements to process. If `$unsorted` is empty, then the function will halt processing. If there are more nodes to process, the function uses a quantified expression, which evaluates to a Boolean value based on a sequence's content. [Listing 24](#) shows the expression that does the heavy lifting for the sorting you want.

Listing 24. XQuery Quantified expression

```

every $id in depends/@id satisfies $id = $sorted/@id

```

The `every` expression is True if all values match. The function then calls itself with the new values for `$sorted` and `$unsorted`. This process continues until `$unsorted` is empty.

The `exampleSorting1` Ant target runs the above XQuery code on the previously presented XML code, ordering `<task>` elements in terms of precedence of dependency. [Listing 25](#) shows the resulting output.

Listing 25. Output of processing `sorting.xq`

```
<?xml version="1.0" encoding="UTF-8"?>
<construction>
  <task id="basement"/>
  <task id="first-floor">
    <depends id="basement"/>
  </task>
  <task id="second-floor">
    <depends id="basement"/>
    <depends id="first-floor"/>
  </task>
  <task id="roof">
    <depends id="basement"/>
    <depends id="first-floor"/>
    <depends id="second-floor"/>
  </task>
  <task id="house">
    <depends id="roof"/>
    <depends id="second-floor"/>
    <depends id="first-floor"/>
    <depends id="basement"/>
  </task>
</construction>
```

Success! The XML code is now ordered in terms of what you would need to build first, which was explicitly defined with `<depends>` elements. To get another idea of where you might use this idiom, I provide an alternate XML source file, which embeds dependencies in a document. Run Ant Target `exampleSorting2` to see that it's just as easy to extract and sort from these kinds of documents.

Section 6. Higher-order functions

Higher-order functions are those that both accept functions as arguments and return them as results. Unfortunately, XQuery 1.0 does not natively support the concept, although I note that it is on the list of requirements for the next version (see "XQuery 1.1 Requirements" in [Resources](#)). At this point, you need to ensure that you have the SAXON-SA version (v9.1) as the following section will make heavy use of the proprietary `saxon:function()` function.

Treewalker revisited

The [previous section](#) presented a function that walked the tree. Take another look at optimizing this function to illustrate how first-order functions can make your code more flexible.

Take the `treewalker2.xq` concept of using functions to output text and attributes one

step further by passing which functions to use to the `local:treewalker()` function itself, as in [Listing 26](#).

Listing 26. treewalker3.xq

```
xquery version "1.0";

declare function local:attrHandler ($attr) {
  $attr
};

declare function local:textHandler ($text) {
  $text
};

declare function local:treewalker
($tree,$attrFunc,$textFunc) {
  let $children := $tree/*
  return
  if(empty($children)) then ()
  else
  for $c in $children
  return
  ( element {name($c)}{
    saxon:call($attrFunc,$c/@*),
    saxon:call($textFunc,$c/text()),
    local:treewalker($c)
  })
};

<test>
  {local:treewalker(<html info="test"><body><a/>
    <b><c info="test1">test</c></b>
    <p>teststs</p></body></html>,
    saxon:function("local:attrHandler", 1),
    saxon:function("local:textHandler", 1))}
</test>
```

The `local:treewalker` function now allows you to define a function with which to handle the outputting of any text or attributes. As I mentioned, XQuery has no native facilities for this, which is why you need to take advantage of a few extension functions that the Saxon SA processor provides:

- **saxon:function:** Creates a first-class function that can be passed as an argument to other functions
- **saxon:call:** Calls a first-class function previously created using `saxon:function()`

The local:map function

The `local:map` function applies the specified function to each item in a sequence, returning a new sequence containing the results of evaluation. [Listing 27](#) provides an example of this function.

Listing 27. map.xq

```
xquery version "1.0";

declare function local:map($func, $seqA as item(*) as item(*) {
  for $a at $i in $seqA
  return
  saxon:call($func, $a)
};

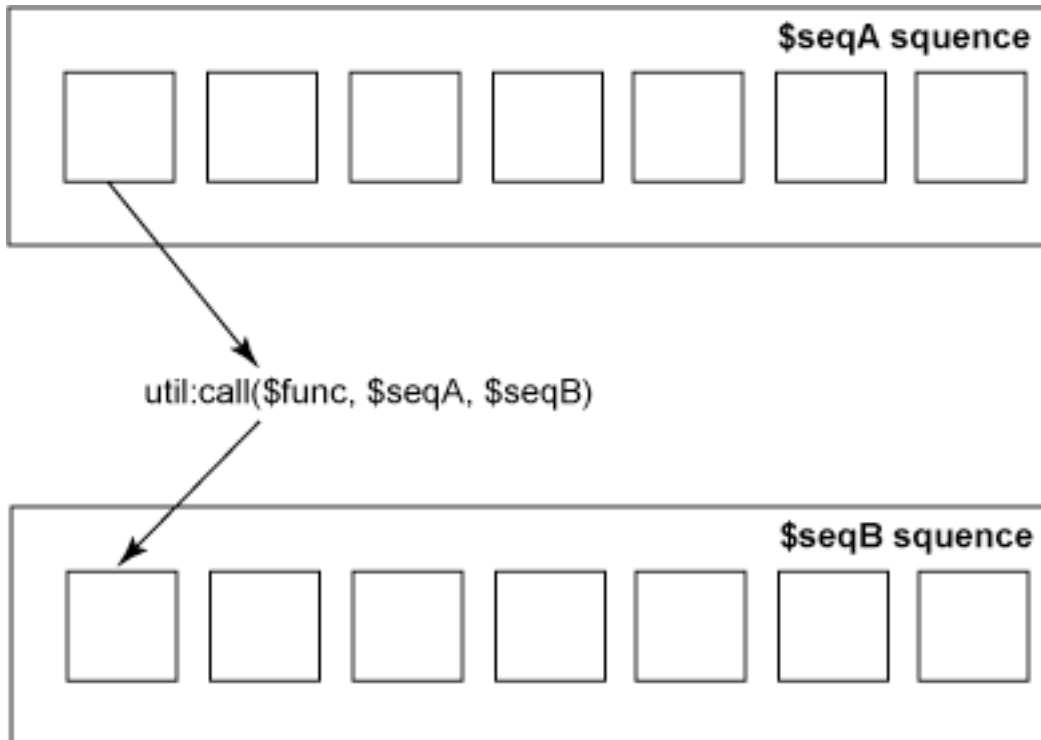
declare function local:doSomething ($a) {
  $a * 2
};

<test>
{
  let $a := (1,2,3,4,5,6,7,8)
  let $b := local:map( saxon:function("local:doSomething", 1),
  $a)
  return
  $b
}
</test>
```

Look at the `local:map` function first. It gets passed two arguments: the function with which you want to process each element and the source sequences. The function iterates through the source sequence, using the `saxon:call` to apply the `local:doSomething` function.

[Figure 2](#) shows how `$func` is applied to each item in the sequence `$seqA`, with the result being placed in an item in the result sequence `$seqB`.

Figure 2. Map processing data from `$seqA`, placing results into `$seqB`



Map is quite suitable for XQuery and its sequences of XML documents; I use it everywhere. The goal of `local:map()` is to apply a function quickly to every item of a sequence and get back the changes in the same order of processing.

Run the Ant target `exampleMap`, and you should get an output containing a sequence of numbers, as in [Listing 28](#).

Listing 28. map.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<test>2 4 6 8 10 12 14 16</test>
```

The output is the result of applying the `local:doSomething()` function to each member of the source sequence, which in this case is just a multiplication operation by 2.

Note: You probably noticed that I elected just to use the return value as the new sequence instead of explicitly passing a result sequence as an argument to the function.

The `local:filter` function

Another variation on processing sequences is the `local:filter` function. This function applies a specified function that tests each item in a source sequence, returning only items from the source sequence that evaluated to Boolean True. [Listing 29](#) provides an example of this function.

Listing 29. filter.xq

```
xquery version "1.0";

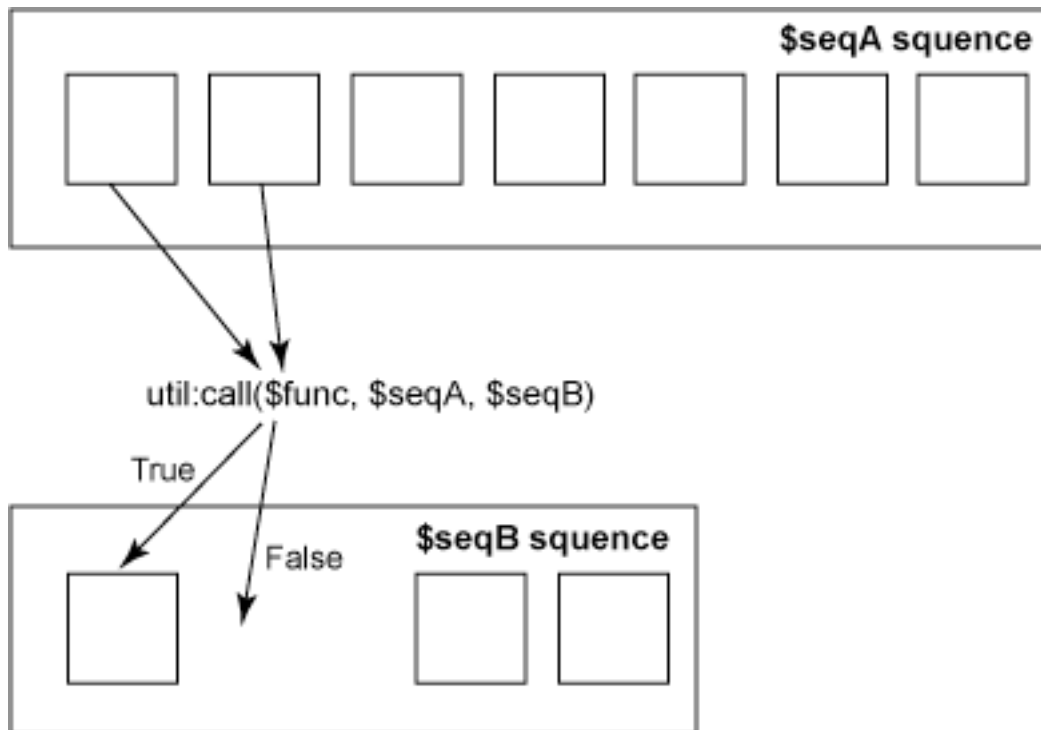
declare function local:filter($func, $seq as item(*) as item()* {
  for $i in $seq
  return
    if(saxon:call($func, $i)) then $i
    else ()
};

declare function local:doSomething ($a) as xs:boolean {
  if ($a = 1) then fn:true()
  else if ($a = 5) then fn:true()
  else fn:false()
};

<test>
{
  let $a := (1,4,5,1,2,2,1,8)
  let $b := (0,0,0,0,0,0,0,0)
  return
    local:filter( saxon:function("local:doSomething", 1),$a)
}
</test>
```

The `local:filter` function applies a `$func` to each item in the source sequence. If the result of processing the item with `local:doSomething` is True, then the item is copied to the result sequence. [Figure 3](#) shows this process.

Figure 3. The local:filter diagram



Run the `exampleFilter` Ant target to see the output, which is shown in [Listing 30](#).

Listing 30. Output from filter.xq

```
<?xml version="1.0" encoding="UTF-8"?>
<test>1 5 1 1</test>
```

The `local:doSomething` function evaluates to True if it encounters a value of 1 or 5, so this is the output you expect to see with the given input.

That wraps up the treatment on higher-order idioms. It was brief, but I hope it illustrated the basics of how to get started with these functions in XQuery.

Section 7. Summary

Wrap up

Most of the initial anxieties with XQuery seem to go away when you actually start to

use it. You must be aware of a few rough edges, but what language doesn't have its warts (or opinions on syntax)? As for measuring XQuery against XSLT, it's not comparing apples to apples. XSLT clearly wins in a majority of categories, but XQuery is better at some things. Middleware development using both XQuery and XSLT is where you start to see how they can complement each other. These are some examples using XQuery:

- Data Aggregation (mixing, archiving feeds)
- Metrics (dashboards, reporting)
- Data Integration (integrating two systems of XML data definitions)
- Localization
- Data health and validation

I believe that XQuery, when used with the idioms presented in conjunction with other XML technologies such as XML databases and XSLT, can be a serious choice for your server-side application development.

Downloads

Description	Name	Size	Download method
Sample scripts for this tutorial	x-advxquery-tut-code.zip	14 KB	HTTP

[Information about download methods](#)

Resources

Learn

- [XQuery 1.0: An XML Query Language](#): See the core document outlining the language.
- [XML Query Use Cases](#): Get a good reference to problems and their XQuery solutions.
- [XML Query Requirements](#): Browse the list of significant requirements for XQuery to address.
- [XQuery 1.0 and XPath 2.0 Formal Semantics](#): See the algebra formally defining query engine behavior.
- [XML Syntax for XQuery 1.0 \(XQueryX\)](#): Read about the XML syntax for queries—verbose and good for machine parsing.
- [XML Path Language \(XPath\) 2.0](#): Read the related XPath 2.0 specification.
- [XQuery 1.0 and XPath 2.0 Functions and Operators](#): Get a definition of functions and operations that XQuery and XPath share.
- [XSLT 2.0 and XQuery 1.0 Serialization](#): This specification defines rules for construction of XML output from XQuery.
- [XQuery Update Facility 1.0](#): This specification defines an update facility that extends the XML Query language, XQuery.
- [XML Schema specification](#): Read how to define the structure, content, and semantics of XML documents.
- [XML Query Requirements 1.1](#): See the new set of requirements for informing the next version of XQuery.
- [Debunking XQuery myths and misunderstandings](#) (Frank Cohen, developerWorks, July 2005): Read about the various myths and permatopics surrounding XQuery.
- [Comparing XSLT 2.0 and XQuery](#) (Benoit Marchal, developerWorks, April 2006): Learn the specifics of each language, and decide which one will save you more time in your projects.
- [dyn:evaluate function](#): Learn how this function evaluates a string as an XPath expression and returns the resulting value.
- [Tail recursion](#): Find a good definition of this recursion optimization in Wikipedia.
- [Topological sorting](#): Check out a directed acyclic graph (DAG), a topo sort that's a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges.

- [EXSLT](#): Explore EXSLT, a community initiative to provide extensions to XSLT.
- [FLWR expressions](#): See this introduction to FLWR expressions.
- [Michael Kay's Saxon XSLT and XQuery processor](#): Read "XSLT 2.0 and XQuery 1.0 processing in Saxon."
- [IBM XML certification](#): Find out how you can become an IBM-Certified Developer in XML and related technologies.
- [XML technical library](#): See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- [developerWorks technical events and webcasts](#): Stay current with technology in these sessions.
- The [technology bookstore](#): Browse for books on these and other technical topics.
- [developerWorks podcasts](#): Listen to interesting interviews and discussions for software developers.

Get products and technologies

- [List of XQuery implementations](#): See XProc implementations currently undergoing development.
- [Saxon XSLT and XQuery processor](#): Try this mature and full featured XSLT and XQuery processor.
- [eXist-db](#): Try this open source database system that stores XML data according to the XML data model and provides index-based XQuery processing.
- [MarkLogic XML database](#): Discover XQuery 1.0 processing in an XML database.
- [IBM trial software for product evaluation](#): Build your next project with trial software available for download directly from developerWorks, including application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- [XML zone discussion forums](#): Participate in any of several XML-related discussions.
- [developerWorks XML zone: Share your thoughts](#): After you read this article, post your comments and thoughts in this forum. The XML zone editors moderate the forum and welcome your input.
- [developerWorks blogs](#): Check out these blogs and get involved in the [developerWorks community](#).

About the author

James R. Fuller

Jim Fuller has been a professional developer for 15 years, working with several blue-chip software companies in both his native USA and the UK. He has co-written a few technology-related books and regularly speaks and writes articles focusing on XML technologies. He is a founding committee member for [XML Prague](#) and was in the gang responsible for [EXSLT](#). He spends his free time playing with [XML databases](#) and XQuery. Jim is technical director for a few companies ([FlameDigital](#), [Webcomposite s.r.o.](#)) and can be reached at jim.fuller@webcomposite.com.

Trademarks

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

IBM, the IBM logo, ibm.com, DB2, developerWorks, Lotus, Rational, Tivoli, WebSphere, and pureXML are trademarks of IBM Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.