XQuery in the Browser

Ghislain Fourny
Systems Group
ETH Zürich, Switzerland
ghislain.fourny@inf.ethz.ch

Markus Pilman
Systems Group
ETH Zürich, Switzerland
mpilman@student.ethz.ch

Daniela Florescu
Oracle
Redwood City, CA, USA
dana.florescu@oracle.com

Donald Kossmann
Systems Group
ETH Zürich, Switzerland
donaldk@inf.ethz.ch

Tim Kraska
Systems Group
ETH Zürich, Switzerland
tim.kraska@inf.ethz.ch

Darin McBeath
Elsevier
West Chester, OH, USA
D.McBeath@elsevier.com

ABSTRACT

Since the invention of the Web, the browser has become more and more powerful. By now, it is a programming and execution environment in itself. The predominant language to program applications in the browser today is JavaScript. With browsers becoming more powerful, JavaScript has been extended and new layers have been added (e.g., DOM-Support and XPath). Today, JavaScript is very successful and applications and GUI features implemented in the browser have become increasingly complex. The purpose of this paper is to improve the programmability of Web browsers by enabling the execution of XQuery programs in the browser. Although it has the potential to ideally replace JavaScript, it is possible to run it in addition to JavaScript for more flexibility. Furthermore, it allows instant code migration from the server to the client and vice-versa. This enables a significant simplification of the technology stack. The intuition is that programming the browser involves mostly XML (i.e., DOM) navigation and manipulation, and the XQuery family of W3C standards were designed exactly for that purpose. The paper proposes extensions to XQuery for Web browsers and gives a number of examples that demonstrate the usefulness of XQuery for the development of AJAX-style applications. Furthermore, the paper presents the design of an XQuery plug-in for Microsoft's Internet Explorer. The paper also gives examples of applications which were developed with the help of this plug-in.

Categories and Subject Descriptors

D.3.2 [**Software Engineering**]: Language Classifications—XQuery; H.4.0 [**Information Systems Applications**]: General

General Terms

Design, Languages, Performance, Standardization

Keywords

XML, XQuery, browser, script, scripting, JavaScript, DOM, HTML, XHTML, events, stylesheets, CSS, client-side programming, mash-up

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW 2009, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

1. INTRODUCTION

Over the years, the code producing Web pages has kept moving back and forth between the client and the server. Many applications are server-side, many others are client-side, and the borders are blurred. After a period in which almost all the code was on the server (thin-clients), we are now experiencing the AJAX and Web 2.0 trend, in which a great deal of code is executed on the client again. Client-side software today means browser-embedded software, so that there is no installation. The browser is no longer just a rendering engine, it has become a programming platform, more powerful than ever.

The most popular programming language for the browser today is JavaScript. JavaScript was specifically designed to run in a Web browser and, thus, JavaScript is a good match for the development of client-side programs. In particular, JavaScript is well-suited for programming event-based user interfaces in the Web browser. All main-stream Web browsers today support the execution of JavaScript natively.

In addition to JavaScript, there are several alternatives to effect browser-embedded application programming. First, JavaScript has been extended in order to embed XPath code into a JavaScript program. XPath is useful in order to declaratively navigate the DOM that represents the Web page. Second, the Google Web Toolkit (GWT) is a popular tool that allows cross-compilation from Java to JavaScript. GWT helps Java programmers to get started with programming the Web browser without the need to learn a new programming language. Furthermore, GWT enables the development of both server-side and client-side application code in a uniform way. Another extension provided by Google is the Gears framework which, among others, supports persistent data (i.e., database access with SQL) and threading as part of JavaScript programs. Finally, Adobe Flash and Flex are popular ways to program powerful user interfaces in the Web browser.

The purpose of this paper is to provide another alternative to program applications inside the Web browser: XQuery. The goal is to combine the advantages of the existing alternatives (e.g., JavaScript, XPath, GWT, and Gears) into a single, uniform, and powerful offering. Because it is Turing-complete, XQuery is powerful enough to implement anything that can be implemented by JavaScript or GWT. However, XQuery can also co-exist with existing technologies in the Web browser, thereby re-using existing JavaScript and Flex application code and using XQuery to implement func-

tionality that is not well supported by the existing technologies. This also provides a graceful evolution of existing systems and to compensate for features that are not well supported by XQuery today.

XQuery has several compelling reasons why it should be used in the browser. First, XQuery is a super-set of XPath which is already heavily used inside Web browsers in order to navigate the DOM. XQuery is not only useful to navigate the DOM; it can also manipulate the DOM in a declarative way, it supports declarative access to persistent data (like SQL in Gears), it has a powerful function and operator library (e.g., for dates and times), and it supports scripting (like JavaScript). In a nutshell, XQuery was designed to process and correlate data on the Web and that is exactly what AJAX-style programs and *Web mash-ups* need to do. Section 6 gives an example application that show-cases this property. Furthermore, XQuery is carefully designed to be highly optimisable.

Second, XQuery runs on all application tiers (database, middleware, and Web browser) and is thus highly portable. For instance, all major database vendors (e.g., IBM, Microsoft, and Oracle) have implemented XQuery as part of their database product and many middleware products (e.g., BEA's workflow engine, information integration product, and enterprise service bus) support XQuery. Hence, XQuery code can be shipped between different tiers which can be exploited in order to reduce cost. Section 6 presents an example application that show-cases this advantage.

Third, XQuery is a family of standards endorsed by the W3C so that it is standardised and the whole XQuery familiy interoperates well with other W3C recommendations; e.g., XML, XML Schema, REST/Web Services. The XQuery family is a complete and powerful programming model.

Finally, there is a significant industry push for XQuery. A great deal of tools are developed and products are maturing. XQuery is taught in the curricula of the Computer Science programs of many universities. As a consequence, it can be expected that more and more XQuery programmers will soon appear on the job market.

Again, the goal of this work is to study XQuery as an alternative and complementary technology in order to implement client-side applications inside the Web browser. That is, the purpose is to study the potential of this technology and see, with the help of examples, how it co-exists with existing technology such as JavaScript. To this end, this paper reports on the following contributions:

- Show that XQuery is a viable option for client-side, browser-embedded applications. We present examples which show the advantages and expressive power of XQuery for this purpose.
- Extend XQuery for the browser, thereby providing a syntax for event-based programming, CSS, and access to the Browser Object Model.
- Design a plug-in for running XQuery in the Web browser. A first release of this plug-in for Microsoft's Internet Explorer is available for free download and open sourced at `http://www.xqib.org`. An implementation of the plug-in for Firefox is currently on the way and should be ready in 2009.
- Give two application examples, a Web mashup and a publishing application, which demonstrate the flexibility of XQuery to integrate horizontally and vertically into an application stack. Additional examples that demonstrate the power of XQuery and the plug-in are available at `http://www.xqib.org`.
- Show that building an XQuery-on-all-tiers application from scratch significantly simplifies the technology stack.

The remainder of this paper is organised as follows: Section 2 describes the state of the art and gives the main features of JavaScript, XPath embedded in JavaScript, GWT, Gears, and Flex. The goal of our work is to combine these features into a more powerful programming framework based on XQuery. Section 3 gives a brief overview of the XQuery family. Section 4 shows the proposed extensions for XQuery so that it becomes a candidate for Web browser-embedded programming. Section 5 gives the design of an XQuery plug-in for the Internet Explorer. Section 6 describes our experience in using this plug-in for two example applications as well as notes on XQuery-only web applications. Section 7 contains conclusions and shows avenues for future research.

2. STATE OF THE ART

Given the growing importance of the Web browser, a number of alternative techniques to develop Web browser-based applications have been developed. This section gives an overview of the most prominent languages and tools: JavaScript, XPath embedded into JavaScript, GWT, Gears, and Flex. The goal of this work is to combine the advantages of these languages and tools into an XQuery application development framework.

2.1 JavaScript

JavaScript was developed in 1995. The initial motivation was to validate forms at the client-side without the need to exchange data with the server (which was much slower than today). Today, JavaScript has become a popular language and several extensions have been added by vendors to make it a powerful programming tool for the Web browser. AJAX (Asynchronous JavaScript And XML) is probably the best example of how JavaScript can help build powerful client-side applications. JavaScript is a great programming language for the browser because JavaScript was specifically designed for this purpose.

An important feature of JavaScript is that it supports event-based programming which is needed for modern user interfaces. A second feature of JavaScript is that it supports DOM, which is the API used in order to navigate and manipulate Web pages in a Web browser. A third feature of JavaScript is its availability in all browsers.

XQuery naturally supports the navigation and manipulation of Web pages because XQuery supports the navigation and manipulation of XML and, thus, any kind of data stored behind a DOM API. The support for event-based programming in XQuery will be detailed in Section 4. As a result, XQuery is a viable candidate to replace JavaScript. As will become clear in the remainder of this paper, however, both XQuery and JavaScript can also co-exist in a single application.

2.2 Embedded XPath in JavaScript

Since it is quite cumbersome to navigate Web pages using DOM, Web browsers have supported XPath for a long time. In particular, it is possible to embed XPath expressions into JavaScript programs. Many browsers support this feature. The following example demonstrates this feature for Firefox:

```
var allDivs, newElement;
```

```
allDivs = document.evaluate(
    "//div[contains(., 'love')]",
    document, null, XPathResult.
    UNORDERED_NODE_SNAPSHOT_TYPE, null);
if (allDivs.snapshotLength > 0) {
    newElement = document.createElement('img');
    newElement.src = 'http://.../heart.gif';
    document.body.insertBefore(newElement,
           document.body.firstChild);
}
```

This JavaScript code uses XPath as part of the document. evaluate function. This JavaScript function is called with an XPath expression which looks for all divs containing the word "love". If such an occurence is found, an image with a heart (i.e., "heart.gif") is inserted into the Web page.

Obviously, XQuery naturally supports navigation of Web pages with XPath because XPath is a sub-set of the XQuery programming language. That is, all XPath expressions can be executed by an XQuery processor.

2.3 Google Web Tools (GWT)

As mentioned in the introduction, GWT enables Java programmers to implement client-side applications which run in the Web browser. With GWT, programmers can program in Java and compile their code to an AJAX application. Hence, programmers no longer need to worry about browser incompatibilities, and a lot of JavaScript inconveniences such as errors (type mismatch, etc) are caught by the compiler at compile-time instead of been caught at runtime. Furthermore, GWT programmers can be supported by the same IDE tools as regular Java programmers, e.g. Eclipse.

One big advantage of GWT is that GWT blurs the distinction between the presentation layer and the "middle-tier" of an application: in principle, GWT enables the movement of code between these two tiers in both directions based on technology trends. As described in Section 6, this is an important feature. Our work on an XQuery plug-in facilitates the same feature because XQuery also runs in the middle-tier and (with the help of the plug-in) in the presentation layer. In fact, XQuery runs on all three application tiers (including the database), facilitating even more code movement.

2.4 Gears

Gears is an offering from Google in order to enable the development of full-fledged applications inside the Web browser. For instance, Google Apps (i.e., Google's competition to Microsoft's Office products) were developed with the help of Gears. Among the features provided by Gears are support for databases inside the browser. With the help of this feature, browser-based applications can run even if the client is not connected to the Internet. Furthermore, such a client-side database improves performance if fine-grained changes are made, e.g., to a text document.

Again, our work on enabling XQuery in Web browsers targets in exactly the same direction as Gears. XQuery can also be used to facilitate client-side database access. Furthermore, our XQuery plug-in naturally co-exists with the Gears browser extensions.

2.5 Flex

Flex is a popular tool in order to develop graphical user interfaces for Web browsers. It is based on a programming language called ActiveScript which is similar to JavaScript. In order to run ActiveScript, a Web browser must install a Flex plug-in, in the same way as proposed in this work for running XQuery in a Web browser. Like JavaScript, ActiveScript and XQuery can co-exist in the Web browser - in fact, all three programming languages can be embedded into a Web page. This way, it is possible to design fancy user interfaces with Flex and program more advanced application logic and Web page manipulation with, say, XQuery. Such an approach has been taken already in several projects at ETH Zurich.

3. XQUERY OVERVIEW

This section gives an overview of XQuery, the XQuery Update Facility, the XQuery Scripting Facility as well as Web services and REST support in XQuery. At the end of this section, we explain why XQuery is suitable for browser-embedded programming and enables the implementation of whole applications in a single programming language, thereby simplifying the technology stack of modern Web-based applications.

3.1 XQuery

XQuery is a shorthand for XML Query Language. It has been a W3C recommendation since January 2007 [4]. Originally, it was designed as a query language to query and transform XML data natively. By now, it has evolved to a general purpose programming language that can read, update, and transform any kind of data (including, of course, XML). XQuery is a functional programming language, and it is Turing complete.

XQuery is a super-set of XPath. That is, any valid XPath expression is also a valid XQuery expression. Hence, there is no need to embed XPath into XQuery in the way it is done for JavaScript. Like XPath 2.0, XQuery uses an extended XML Data Model [7]: The XQuery 1.0 and XPath 2.0 Data Model. XQuery is strongly typed, yet flexible because data can be typed as "untyped". In particular, XQuery can natively process (untyped) Web pages.

XQuery has a broad functionality, covering simple expressions such as constants, variables, and comparisons to complex expressions for database queries, transformations, and information retrieval. For example, the FLWOR (pronounced "flower") expression corresponds to the "SELECT FROM WHERE" statement in SQL, for example:

```
for $x at $i in
   doc("bill.xml")/paymentorder/paymentorders
let $price := $x/price
where $x/name ftcontains "computer"
return <li>
         {$x/name}
         <eur>{data($price)}</eur>
       </li>
```

In order to support information retrieval, XQuery also involves full-text search [3] as shown below.

```
for $b in /books/book
where $b/title ftcontains
     ("dog" with stemming) ftand "cat"
return $b/author
```

This example finds all authors of all books the title of which contains the word "cat", as well as the word "dog" or a word with the same stem.

Just like any other functional programming language, an XQuery expression is evaluated in a context. The context contains functions, namespaces, schemas, and variable bindings. For instance, the expression $x will be evaluated using the context; if the variable $x is not defined in the context, then an error is raised as part of the evaluation of this expression. Otherwise, this expression is evaluated to the value of $x as defined in the context. Likewise, function invocations are evaluated according to the definition of the functions in the context. The XQuery recommendation already defines the "http://www.w3c.org/xquery-functions" namespace and a whole function library in this namespace (e.g., sum, distinct-values, etc.) [9]. Extending the context with new browser-specific namespace, schema, and function definitions is an important part of integrating XQuery into the Web browser (Section 4.1).

3.2 XQuery Update Facility

XQuery 1.0 is side-effect free, i.e., an XQuery expression cannot alter an XML node. Obviously, side effects are needed for AJAX-style programming in order to change the Web page based on the actions of a user. The XQuery Update Facility [6] has been designed to extend XQuery in order to facilitate updates to XML nodes. For instance, the XQuery Update Facility enables to insert new elements, delete elements, rename elements and replace the content of an element. Since DOM is an API for XML data, the XQuery Update Facility can be used naturally in order to update Web pages; i.e., data stored behind a DOM interface. The XQuery Update Facility has been a *candidate recommendation* since August 2008.

The following contains two example updates, an insert of a new book into a library and the update of the price of a book:

```
insert node <book title="Starwars"/>
    into doc("library.xml")/books,
replace value of node
   doc("bill.xml")/bill/items[@id="computer"]/price
   with 1500
```

All modifications are performed once the expression is entirely evaluated: there are no side effects until the end and instructions do not see the side effects of former instructions.

3.3 XQuery Scripting Extension

The XQuery Scripting Extension [5] is an extension of XQuery and the XQuery Update Facility. With this extension, XQuery becomes a full-fledged programming language which can be used in order to program any kind of application. The main difference as compared to XQuery and the XQuery Update Facility is that updates become visible during the execution of a program. That is, XQuery expressions can be ordered for sequential execution and the effects of the execution of one expression (e.g., an assignment) become visible for the execution of other, sub-sequent expressions. Furthermore, the XQuery Scripting Extensions introduces several new expressions which are commonly found in modern programming languages; e.g., blocks, variable declaration and assignment, while loops, continue, break, returning values as a result of executing a function.

The following gives a simple example program written using the XQuery Scripting Extension:

```
{ declare variable $b;
set $b := //book[title="starwars"];
insert node $b into doc("lib.xml")/books;
set $b := doc("lib.xml")//book[title="starwars"];
insert node <comment>6 movies</comment> into $b; }
```

This program is a block, the expressions of which are executed sequentially. First, a variable $b is declared. $b is initialized to a book which is copied and inserted into lib.xml. Then $b is set to the newly inserted node in lib.xml (it can be read because this instruction sees the side effect of the insertion). Finally, a comment is inserted into the book.

The XQuery Scripting Extension is still in its early stages of standardisation but first implementations already exist.

3.4 REST and Web Services Support

The W3C is also currently standardising extensions to XQuery in order to effect REST and Web Service calls and in order to make sure that XQuery modules and functions can be called directly with the help of REST or Web Service calls. With the current proposal discussed inside the W3C working group, for instance, a Web service could be defined using the following XQuery code (i.e., a Web service corresponds to an XQuery module):

```
module namespace ex="www.example.ch" port:2001;
declare option fn:webservice "true";
declare function ex:mul($a,$b) {$a * $b};
```

The following XQuery code could be used in order to call the Web service defined above:

```
import module namespace ab="www.example.ch"
    at "http://localhost:2001/wsdl";
replace value of node
    html//input[@name="textbox"]/value
with ab:mul(2,5)
```

Support for REST, which seems to be more popular on the open Web, goes along the same lines.

4. XQUERY IN THE BROWSER

The previous section showed that XQuery has all the ingredients needed to develop Web-based applications. It processes XML data natively, it is declarative, yet powerful enough for complex applications, and it supports remote calls via REST and Web Services. This section presents extensions to XQuery so that XQuery becomes a viable option for AJAX-style applications in the Web browser. Furthermore, it shows how XQuery can be embedded into HTML pages for execution inside the Web browser.

4.1 Overview

Like JavaScript, we propose to embed XQuery scripts into HTML documents with a **<script/>** tag.

XQuery expressions in the browser are executed in a browser-specific context. This context contains all the namespaces and function libraries (Section 3) as well as browser-specific extensions such as the predefined functions.

The processing model is the following: XQuery (and JavaScript) code is executed at the beginning, thereby registering events according to the DOM level 3 standard [8]. Thus, the browser listens for these events and executes the XQuery functions (or JavaScript or VBScript code) accordingly. Currently, JavaScript is executed first, then XQuery (this is the way browsers do it because JavaScript is supported natively). There can be other models (e.g., execution in order of the **<script/>** tags) in the future.

We will follow the tradition and provide the Hello World version of XQuery in the Browser.

```
<html><head>
  <title>Hello World Page</title>
  <script type="text/xquery">
     browser:alert("Hello, World!")
  </script>
</head><body/></html>
```

Again, the XQuery code is embedded into a <**script**/> tag of the HTML webpage. In the header, the script pops up a "Hello, World!" message. This code is called when the page is loaded.

The examples in the next subsection show that XQuery is as expressive as JavaScript; the main difference is that it takes much less code to process and manipulate a webpage using XQuery than using JavaScript because XQuery was designed for that purpose. In the rest of this section, we will also detail browser-specific extensions to XQuery. The most critical ones are event handling and CSS handling.

4.2 Browser Context

As mentioned in Section 3.1, each XQuery expression is evaluated in a context. By default from the XQuery recommendation, this context already comes with a rich function library [9]. Embedding XQuery into the browser, extends the context with one additional namespace (i.e., "http://www.example.com/browser" which is bound to the prefix "browser"). Furthermore, the context includes browser-specific functions. For instance, the browser:self() function returns all the information of the browser's current window; a call to this function is equivalent to accessing the window object in JavaScript. This way, browser-specific components can be accessed using XQuery just as using JavaScript. The remainder of this subsection gives examples for some of the pre-defined functions and types in the browser-specific XQuery context.

4.2.1 Window

Information about the window (or the frame within which the current document is shown) can be accessed with JavaScript through the window object. Retrieving the browser's window objects can be implemented in XQuery using the browser:top() function. The browser:top() function returns an XML element representing the topmost window (the equivalent of top in JavaScript). It has the following form (for the sake of simplicity, we omit namespaces):

```
<window name="top_window">
  <status>Welcome</status>
  <location>
    <href>http://www.dbis.ethz.ch"</href>...
  </location>...
  <frames>
    <window name="child1">
      <status>First child</status>...
    </window>
    <window name="child2">
      <status>Second child</status>...
    </window>
  </frames>
</window>
```

Note that this function call might return different results within the same query and is therefore marked as non-deterministic. A window node contains properties similar to those of window objects in JavaScript (status, name, location, ...). The same is true for location nodes (href, host, port, ...).

In addition, the browser:self() function returns a window node which is a descendant of the XML element above and corresponds to the window in which the XQuery code is being executed.

In order to retrieve certain properties (e.g., a frame, a location), it is possible to navigate through these elements using XQuery. (Recall that XQuery is a superset of XPath.) Furthermore, the window element can be manipulated using the XQuery Update Facility; thereby changing the browser's window. The following lists some examples (the equivalent JavaScript is listed as a comment, below).

This code looks for a child of the top window which is called "leftframe":

```
browser:top()//window[@name="leftframe"]
(: JavaScript: top.frame["leftframe"] :)
```

The following example changes the status of the current window to "Welcome":

```
replace value of node browser:self()/status
    with "Welcome";
(: JavaScript: self.status = "..." :)
```

Here a new variable $win initialised to the second child of the current window is declared:

```
declare variable $win :=
  browser:self()/frames/window[2];
(: JavaScript: var win = self.frames[2] :)
```

To change the location of the variable $win (as above) and cause a new webpage to be displayed, one can write:

```
replace value of node $win/location/href
    with "http://www.dbis.ethz.ch"
(: JavaScript: win.location = "..." :)
```

The following code raises an alert giving the date and time at which the variable $win bound above was last modified:

```
browser:alert($win/lastModified)
(: JavaScript: alert(win.lastModified) :)
```

All these examples can be implemented easily in JavaScript too, as shown in the comments below. But XQuery can do more.

This code looks for any frame (children, grandchildren... of top()) which is named "myframe":

```
browser:top()//window[@name="myframe"]
```

This FLWOR expression writes a big red warning on every frame not pointing to an https location:

```
for $x in browser:top()//window
let $d := browser:document($x)
where not ($x/location/href
  ftcontains "https://")
return
insert node <h1>
  <font color="red">Warning: this page
      is not secure</font></h1>
  into $d/html/body as first
```

(the browser:document function is explained in 4.2.3).

The last example requires significantly more JavaScript code to accomplish the same job.

Like in JavaScript, there are security issues regarding navigation through window nodes: a malicious Web site could tamper with documents in other windows, or learn about

the location of other windows. To avoid this, we suggest to implement window nodes using pull (as opposed to push) and to perform checks in the implementation of all accessors in the window nodes and their descendants (e.g., accessors to get the value of a node, or its children) as well as in routines which tamper with those nodes (when updates are applied). For example, this could be based on a same-origin policy like in JavaScript, or on any other suitable policy. If the check is not successful, an empty sequence is returned.

Imagine, for example, that an application can get a reference to a window node at some point in time (e.g., the pages were on the same domain). If later the policy no longer allows its use (e.g., the user navigated to another domain), then this node becomes useless. That is, all accessors return an empty sequence and thus, it is impossible to learn anything about the new location of this window.

Also for security reasons, we propose to block the fn:doc and fn:put functions in the browser.

4.2.2 Screen and Navigator

Two other important objects in JavaScript are the window.screen object and the window.navigator object which give information on the screen (e.g., size, etc) and the navigator (e.g., vendor, version, etc).

In XQuery, this information can be accessed with two functions browser:screen() and browser:navigator(). Both return an XML node from two XML elements similar to the one shown above (but much simpler). For example, the name of the navigator is accessed with

```
browser:navigator()/appName
(: JavaScript: navigator.appName :)
```

and the height of the screen with

```
browser:screen()/height
(: JavaScript: screen.height :)
```

All of the properties available with the JavaScript screen and navigator objects are accessible as children of the nodes returned by these functions in XQuery.

4.2.3 The Document

Each Web page is a self-contained XML document. If one possesses a window node, say, stored in $w, it is possible to get the corresponding document by calling browser:document ($w).

The document in browser:self() deserves a special treatment: it is the context item ".", meaning that it can be accessed directly instead of browser:document(browser:self()). This means that accessing any node in the document is easy and straightforward with XQuery.

Similarly to window nodes, for security reasons, a check is performed when the browser:document() function is called. If it fails, an empty sequence is returned.

For example, an application could access all HTML div elements in the document containing the XQuery script with:

```
//div
```

and all images in a children window with

```
browser:document(browser:self()/frames/*[2])//img
```

4.2.4 The browser functions

There are some further browser built-in functions available in the browser namespace. We list some of them here. We omit comments as the names are self-explanatory and the names are similar to equivalent JavaScript functions. For more information, see [10], Chapter 5.

- Window-related functions: windowMoveBy, windowMoveTo, windowOpen, windowClose, alert, prompt, confirm
- History-related functions: historyBack, historyForward, historyGo
- Document-related functions (note that with XQuery, best practice would be to modify the XDM and avoid using those functions): write, writeln.

The programmer might also want to access browser information and write browser-specific code in XQuery:

```
if (browser:navigator()/appName
        ftcontains "Mozilla") then
    browser:alert("You are running Mozilla")
else if (browser:navigator()/appName
        ftcontains "Internet Explorer") then
    browser:alert("You are running IE")
```

4.3 Events

One of the most important features of client-side programming is event-handling. The basic principle of event-handling is that some functions (called listeners) are called when an event occurs at a certain location. For this call to occur, the listener is registered in advance.

In JavaScript, there are two ways to register for events. The first, simple way is to use e.g. the onclick, onload properties. The second way is to add listeners manually.

In XQuery, a first approach could be to use a pre-defined function in the browser context which supports the registration of listeners for events. Those functions would be high-order functions as they take functions as an argument. As registering events is a very important scheme, we suggest a second approach: an extension of the XQuery syntax as shown in the following subsection.

4.3.1 Managing event listeners in XQuery

To demonstrate the mechanism, let us assume we have the following function which we would like to be called when the user clicks on a button.

```
declare sequential function local:myEventListener
    ($evt, $obj) as xs:boolean {
  declare variable $message = <message>Event occured:
      {$evt/type}
      at {$obj}
    </message>;
  exit with browser:alert(data($message));
};
```

This very simple function raises a message box with information on the event (this is the information which is passed as parameters: the event itself, and its location).

We would like to register this function as a listener for a click event at the button named “button”.

In JavaScript, we register such a listener with:

```
document.getElementById("button").addEventListener
  ("onclick", myEventListener, false)
```

For XQuery, we propose the following syntax for registering an event:

```
on event "onclick" at //input[@id="button"]
  attach listener local:myEventListener
```

(This means that local:myEventListener shall be called whenever the user clicks at the location specified.)

For deregistering an event:

```
on event "onclick" at //input[@id="button"]
  detach listener local:myEventListener
```

(This means that we cancel the registration, i.e., the listener shall no longer be called if the events occurs.)

For triggering an event:

```
trigger event "onclick" at //input[@id="myButton"]
```

(This simulates a user clicking at the specified location.)

This corresponds to the following XQuery grammar extension:

```
ExprSingle ::= (all existing options)
             | EventAttach | EventDetach
             | EventTrigger
EventAttach ::= "on" "event" ExprSingle
              ("at"|"behind") ExprSingle
              "attach" "listener" QName
EventDetach ::= "on" "event" ExprSingle
              "at" ExprSingle
              "detach" "listener" QName
EventTrigger ::= "trigger" "event" ExprSingle
              "at" ExprSingle
```

The "behind" construct, used for asynchronous calls (e.g., of Web services), is explained later in section 4.4

4.3.2 Event Node

When an event occurs, the listener is called and receives two parameters $evt and $obj. The first parameter is an XML element which contains information about the event, for example whether the alt key was pressed, which mouse button was used... the same information which is available in an Event Object in the DOM [10]. The second element is the DOM node where the event occured.

As in the DOM, there are several event properties which can be queried: $event/target, $event/type, $event/altKey, $event/button... so that one can adapt the behaviour of the listener:

```
declare function local:listener($evt, $obj) {
    if($evt/button=1) (: do something :)
    else (: do something else :)
};
on event "onclick" at html//input[name="submit"]
    attach listener local:listener
```

In this example, when the user clicks on the submit button, my:listener is called and receives an event node as well as the location node as parameters. It reads the event node and does something if the user clicked with the left button, something else if she clicked with the right button.

4.4 AJAX

In JavaScript, the XMLHttpRequest object allows to make asynchronous calls, at a low-level. In XQuery, we use the event syntax extension to implement asynchronous calls. The following example is the XQuery version of an AJAX example given at [1]:

```
<html><head>
    <script type="text/xquery">
    import module namespace
      ab = "http://example.com"
      at "http://www.example.com/wsdl";
    declare updating function
        local:showHint($str as xs:string) {
      if(fn:length($str)) eq 0)
        do replace value of //*[@id="txtHint"]
          with ()
      else
        on event "stateChanged"
          behind ab:getHint($str)
          attach listener my:onResult };
    declare updating function
        local:onResult($readyState, $result) {
          if($readyState eq 4) then
            do replace value of //*[@id="txtHint"]
              with $result
          else () };
</script></head><body>
    <form>First Name: <input type="text" id="text1"
    onkeyup="local:showHint(value)"></form>
    <p>Suggestions: <span id="txtHint"></span></p>
</body></html>
```

First, the namespace prefix ab is bound to a web service. When the user types in the text box, local:showHint(value) is called (onkeyup attribute of the input element). If the textbox is not empty, then ab:getHint($str) is called asynchronously. In this call, the important new concept is the "behind" construct which binds the event to the evaluation of the expression, rather than to its result. As a consequence, an event is triggered when the computation of the result is completed; i.e., when the remote call returns with a result. Furthermore, the call is non-blocking; i.e., asynchronous. The user keeps control of the user interface. Every response or signal which is returned from ab:getHint, triggers a call to the my:stateChanged function. For instance, if the signal indicates that the response has arrived, then the hint is displayed on the webpage.

4.5 CSS

Another important aspect of browser programming is handling stylesheets. Stylesheets could be actually manipulated directly using XQuery by modifying the style attribute of an element (which is a string containing the list of style properties and their values), with an updating expression.

An alternative way is to introduce additional syntax as this is such a common case. This has the additional advantage of not integrating the style properties in the XML tree as children of the style attribute, which would not be correct XML.

For example, one modifies the style of an element with:

```
set style "border-margin"
    of //table[@id="thistable"] to "2px"
```

And one can query its style with:

```
{ declare variable mystring as xs:string;
set $mystring := get style "border-margin"
    of //table[@id="thistable"]; }
```

This is associated with the following grammar extension:

```
ExprSingle ::= (all existing options)
             | SetStyleExpr | GetStyleExpr
SetStyleExpr ::= "set" "style" ExprSingle
               "of" ExprSingle "to" ExprSingle
```

```
GetStyleExpr ::= "get" "style" ExprSingle
              "of" ExprSingle
```

5. IMPLEMENTATION

For the implementation, we chose to build an extension for Microsoft Internet Explorer. A first release is already available under `http://www.xqib.org`. An implementation for Firefox is under construction as well. Both will use the Zorba XQuery engine to execute the scripts.

5.1 Implementation status

Zorba supports XQuery 1.0, Updates and partly Scripting. Hence, DOM navigation as well as updates of the DOM have been implemented by providing an XDM store wrapping the DOM.

Event-handling and styles have also been implemented. However, as Zorba does not allow to modify in a modular way the XQuery grammar it uses, we use high-order-functions (implemented in C++) to bind events and handle styles instead of the syntax suggested in this paper. Also for technical reasons, the code executed when the page is loaded is put in a function local:main().

As Zorba chose to first support REST, synchronous REST calls (e.g., get) are possible. Web services will follow.

One important issue we encountered with Internet Explorer is that it tranforms all HTML tags to upper-case, so that the XPath expressions have to contain upper-case names. This brings about the risk that XQuery code could be incompatible between browsers. However, we believe that a workaround can be found, or maybe Internet Explorer will fix this in the future.

Code samples demonstrating the various features are available at [2].

5.2 Architecture

The IE plugin, programmed with C++, uses the Zorba XQuery engine which supports XQuery 1.0 as well as the XQuery Update Facility. Zorba is a pluggable open-source XQuery engine which is developed by the FLWOR foundation. Zorba is implemented in C++ and distributed under an Apache 2.0 license.

Our browser extension works as depicted in Figure 1. First, the browser receives an XHTML document and parses it. It generates the DOM, renders the webpage and initialises the extension. Then the plugin obtains the XQuery script, which is an XQuery prolog.

Zorba is called with the XQuery prolog followed by the main query call. This call may register event listeners. As the plugin implements the XDM on top of the DOM, Zorba, by reading and modifying the XDM, will read and modify the DOM accordingly. The plugin then listens for IE events. When an event occurs, Zorba is called with the XQuery prolog followed by the listener call, and the plugin loops between listening for IE events and executing the corresponding listeners.

6. APPLICATION SCENARIOS

Several applications which make use of XQuery in the browser have been written in labs at ETH Zurich. Furthermore, one application has been developed in collaboration with Elsevier. This section gives three examples. The first one, based on the collaboration between ETH and Elsevier, shows how XQuery in the browser helps to migrate computation from servers to client machines; this application, thus, demonstrates the ability of XQuery to integrate different tiers (i.e., vertical integration). The second application is a typical Web mash-up, effected in the browser. It shows the ability of XQuery to co-exist with JavaScript in the brower and the ability of XQuery to integrate different services horizontally. The third example shows how XQuery-only applications simplify the technology stack.

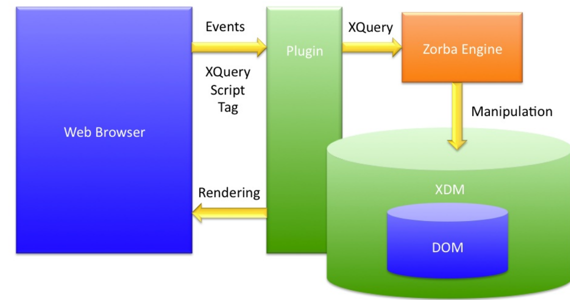

Figure 1: Interaction with the XDM and the DOM

6.1 Elsevier Reference 2.0

The publishing industry has a long tradition of working with markup languages such as SGML and XML. Hence, the publishing industry has been using XQuery for a while in order to manage their XML data in the database and in the middle-tier. Consequently, the project Reference 2.0 was implemented from the beginning using XQuery in the middle-tier. With Reference 2.0, the user can browse through journals, volumes, issues and articles and then, for a given article, study the references (statistics, years...).

In the original architecture, an XQuery application server produces Web pages with data from an XML database available via REST calls. The XML database contains the article hierarchy and their contents. The generated Web pages contain client-side JavaScript code which reacts on events and allows interactivity.

In order to off-load Elsevier's servers, we are using our XQuery plug-in and migrating the XQuery code from the server to the client. As a result, the served Web pages contain both JavaScript and XQuery (see Figure 2). The JavaScript code improves the usability of the interface. The XQuery code takes care of the page layout and directly queries the XML from the database using REST calls. In order to improve performance, whole XML documents can be cached in the browser so that most user requests can be processed without any interaction with the Elsevier server.

The beauty of this migration project is that the JavaScript code is completely unchanged and that the XQuery code which runs in the client is almost the same as the XQuery code that previously ran in the server. In other words, this migration can be done with very little effort and shows how XQuery in the browser can help to make the software architecture of a Web-based application more flexible. In this particular case, reducing cost by off-loading servers was the main motivation for this project. Likewise, other business factors or trends could motivate to move more code back from the client to the server.

In more technical detail, we are carrying out the following

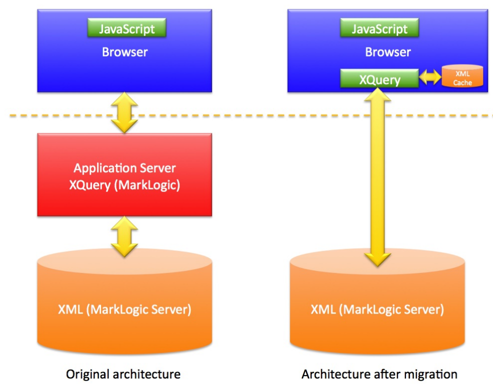

Figure 2: Elsevier Reference 2.0: Server-to-client Migration

changes in order to migrate the existing Reference 2.0 application from server-side to client-side software. The XQuery modules defined in the Reference 2.0 application code are directly published and made available and visible to clients. Furthermore, the REST interface of the Reference 2.0 modules has to be adjusted so that they serve whole documents rather than individual queries to documents (to better enable caching). Furthermore, some specifics of the XQuery product used as an XMLDB (i.e., Marklogic) have to be adjusted. Finally and most importantly, XQuery code is injected into the Web pages served by the Reference 2.0 server. For each dynamically generated webpage, the server-side expressions are moved to the script tag (which is created if it does not exist) as follows: the prolog is directly inserted into the script tag, whereas the contents enclosed in the outermost element constructors (formerly computed by the server) are removed and put into insert expressions in the main function (they will be inserted by the client).

It would also be possible to translate the JavaScript code to XQuery, but it would need more time and it is not necessary for this project.

6.2 Google Maps-Weather Mash-up

The goal of this project was to develop a Mash-up between Google Maps and a Weather service (actually, a selection of different weather services is used, depending on the used language and the region of interest). Furthermore, this application integrates Web cams for the locations of interest. This application gives weather information for every location shown in Google Maps. Likewise, the service displays maps for all locations given and queried as part of one of the weather services.

In this project, the Web application uses both JavaScript and XQuery. JavaScript is used to run Google Maps and communicate with the Google server using AJAX. XQuery in the browser (with our plugin) is used in order to initiate REST calls to the diverse weather services and integrate the results returned by the weather services. Furthermore, XQuery is used in order to search for Web cams at the location of interest. The interesting aspect of this co-existence of JavaScript and XQuery in the browser is that, in this application, code written in both languages listens to the same events. For instance, if the search button in Google Maps is clicked, then naturally, Google is called in order to serve the right map. At the same time, the XQuery code

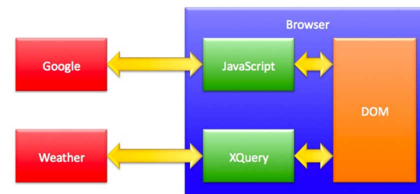

Figure 3: Google Maps-Weather Mash-up: how XQuery and JavaScript can interact

also handles this event (i.e., the click on the search button), extracts the location from the search box and initiates REST calls to weather services and Web cams. Also, code in both languages is used in order to query and manipulate the Web page (in the browser's DOM), as shown in Figure 3. In this figure, the Web page serves like a database and both JavaScript and XQuery code can be used in order to access and update that "database". The browser determines the order in which events are processed (by JavaScript and XQuery functions) in the same way as the browser serialises the order of event processing in the case that only JavaScript is used (and several different JavaScript functions are used to handle the same event).

6.3 XQuery Only

The previous two applications showed how XQuery and JavaScript could co-exist in modern Web-based applications. JavaScript was mostly used because it was already there (i.e., legacy code) and it was too much effort to rewrite the JavaScript code. This section shows how XQuery-only can significantly simplify an application and result in less code, if a new application is designed from scratch.

The state of the art in web application development is to have several languages in the same file. Here is an example of an application which simulates a shopping cart, taking products out of a database (We omitted the code to connect and disconnect). Clicking on the button next to a product adds it to the shopping cart.

```
<html><head><script type='text/javascript'>
function buy(e) {
  newElement = document.createElement("p");
  elementText = document.createTextNode
    (e.target.getAttribute(id));
  newElement.appendChild(elementText);
  var res = document.evaluate(
   "//div[@id='shoppingcart']", document, null,
   XPathResult.UNORDERED_NODE_SNAPSHOT_TYPE, null);
  res.snapshotItem(0).appendChild("newElement");}
</script></head><body>
<div>Shopping cart</div>
<div id="shoppingcart"></div>
<% // Code establishing connection
ResultSet results =
  statement.executeQuery("SELECT * FROM PRODUCTS");
while (results.next()) {
  out.println("<div>");
  String prodName = results.getString(1);
  out.println(prodName);
  out.println("<input type='button' value='Buy'");
  out.println("id='"+prodName+"'");
  out.println("onclick='buy(event)'/></div>"); }
```

```
results.close();
// Code closing connection %></body></html>
```

In this example, the business logic is developed using JSP (Java nested into HTML). JavaScript is executed on the client, with embedded XPath. The server-side code even contains SQL.

If this application had been developed with XQuery only from scratch, it would look like the following XQuery expression:

```
<html><head><script type='text/xqueryp'><![CDATA[
 declare updating function local:buy($evt, $obj) {
  insert node <p>{$obj/@id}</p> as first
   into //div[@id="shoppingcart"]
 };
 on event "onclick" at //input
  attach listener local:buy
]]></script></head><body>
<div>Shopping cart</div>
 <div id="shoppingcart"/>{
 for $p in doc("products.xml")//product
  <div>{$p/name}
  <input type='button' value='Buy' id='{$p/name}'/>
  </div>
}</div>
</body></html>
```

The database is mapped to an XML document with the URI given as parameter to doc(). A FLWOR expression inserts the products. The events are registered for all buttons with a single instruction. The entire code, client-side and server-side (even the HTML tags) is XQuery.

This shows that using XQuery for everything simplifies considerably the Web application. XQuery can be used for database access (instead of SQL), to define the application logic (instead of C# or Java) and inside the browser (instead of JavaScript). Furthermore, this example shows that an XQuery-only implementation can require fewer lines of code and avoid the technology jungle.

Another example that demonstrates how XQuery can reduce the number of lines of code is given on [2]. The multiplication table demoed on that site requires 77 lines of JavaScript code or alternatively only 29 lines of XQuery code.

7. CONCLUSION

The browser today is not only a rendering tool, but has become a programming environment. The most popular client-side programming language, JavaScript, is geared towards this purpose. This paper showed that XQuery is also a viable candidate to program the Web browser. Just like JavaScript, XQuery is a full-fledged programming language with many features for processing and integrating data on the Web. In addition, XQuery has the advantage of being declarative and therefore better optimisable. In particular, XQuery can process XML data declaratively in order to query and manipulate the DOM that represents a Web page in the browser (e.g., HTML tables) or results of REST calls. Another reason to consider XQuery in the browser is to avoid the technology jungle found in many applications today: XQuery runs in all tiers (database, middleware, and Web browser) and therefore whole applications can be implemented using a single programming language, XQuery. All major (relational) database vendors already support XQuery and XQuery is a popular language in several middleware products. With the help of XQuery, it is thus possible to achieve flexible and largely simplified application architectures, thereby possibly eliminating the need for data marshalling and some layers in the application stack.

In order to make XQuery suited for the browser, XQuery had to be slightly extended. Most importantly, support for events had to be integrated in order to be able to implement reactive and asynchronous user interfaces. Other than that, XQuery was sufficient in order to build any kind of AJAX-style application. Furthermore, it turned out to be fairly straightforward to build an XQuery plug-in for the Internet Explorer. This plug-in together with a number of examples are available under a free open-source licence at http://www.xqib.org.

One important avenue for future work is to further explore and quantify the benefits of developing Web applications using XQuery. For instance, we would like to study the performance of XQuery in the browser as compared to JavaScript. A second important avenue for future work is to port the plug-in to Firefox and other Web browsers; we are currently working on this. Furthermore, we are working on tools for XQuery development (http://www.xqdt.org), like a debugger, performance profiler and Eclipse support. In the long run, we expect the availability of tools and libraries for XQuery to surpass those available for JavaScript.

8. REFERENCES

[1] Ajax suggest example. http://www.w3schools.com/ajax/ajax_example_suggest.asp.

[2] XQIB Samples. http://www.xqib.org/samples.

[3] S. Amer-Yahia, C. Botev, S. Buxton, P. Case, J. Doerre, M. Holstege, J. Melton, M. Rys, and J. Shanmugasundaram. XQuery 1.0 and XPath 2.0 Full-Text 1.0. http://www.w3.org/TR/xquery-full-text/.

[4] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robbie, and J. Siméon. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, jan 2007.

[5] D. Chamberlin, D. Engovatov, D. Florescu, G. Ghelli, J. Melton, and J. Siméon. XQuery Scripting Extension 1.0 Working Draft. http://www.w3.org/TR/xquery-sx-10/.

[6] D. Chamberlin, D. Florescu, and J. Robbie. XQuery Update Facility. http://www.w3.org/TR/xquery-update-10/.

[7] M. Fernandez, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). http://www.w3.org/TR/xpath-datamodel/.

[8] P. L. Hégaret and T. Pixley. Document Object Model (DOM) Level 3 Events Specification. http://www.w3.org/TR/2003/NOTE-DOM-Level-3-Events-20031107/.

[9] A. Malhotra, J. Melton, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. http://www.w3.org/TR/xpath-functions/.

[10] N. C. Zakas. JavaScript for Web Developers. Wrox, Wiley Publishing, Inc., 2005.