XQuery in the Browser

Technical Report, ETH Zürich, November 2007

Ghislain Fourny
ETH Zürich
gfourny@inf.ethz.ch

Donald Kossmann
ETH Zürich
donaldk@inf.ethz.ch

Tim Kraska
ETH Zürich
tim.kraska@inf.ethz.ch

Markus Pilman
ETH Zürich
mpilman@student.ethz.ch

Daniela Florescu
Oracle
dana.florescu@oracle.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). Each browser vendor has implemented these extensions differently and has added its own extensions. Today, JavaScript is already becoming a victim of its own success and is being used way beyond its design space. Furthermore, applications and GUI features implemented in the browser have become increasingly complex. For these reasons, programming client-side Web applications with JavaScript is intricate. The purpose of this paper is to reduce programming complexity by proposing XQuery as a client-side, browser-embedded programming language. 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 the Web browser 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 the implementation of an extension to the Internet Explorer and reports on experiences with the implementation of this extension. Finally, the paper compares XQuery with JavaScript and other approaches (e.g., Flash, GWT) in order to develop complex browser-embedded applications.

Categories and Subject Descriptors

D.3 [**Software**]: Programming languages; H.4 [**Information Systems**]: Information Systems Applications

General Terms

XQuery, JavaScript, Browser

Keywords

XML, XQuery, XQueryP, Browser, Script, JavaScript, DOM, Update, Programming, HTML, Event handling

1. INTRODUCTION

Over the years, the code producing webpages 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: code is moved from server to client and even from client to client (e.g., handover from the laptop to the mobile phone if one needs to leave or runs out of battery). 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 lot 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.

There are several alternatives for browser-embedded software development. The biggest player today is JavaScript. JavaScript is very good at being executed in the browser because it is perfectly geared towards this purpose. However, it also has some drawbacks.

The most important drawback is that JavaScript is a victim of its own success and is used way beyond its design space, leading to increased programming complexity in applications and GUI features implemented in the browser. Hence, it is difficult to build big applications since JavaScript was not designed for big systems. JavaScript has been extended and a lot of layers have been added, in order to support more powerful and declarative DOM navigation and manipulation (i.e., XPath). These extensions have made the language less extensible for the future. Furthermore, not all these extensions are seamless so that much more code than necessary is needed to do simple things. We will give examples and details about this and other JavaScript drawbacks in Section 2.

There are alternatives to JavaScript. For example Flash, and Flex which has been built on top of Flash. Flash is very good at multimedia, solves some problems, but inherits most of the JavaScript problems. It furthermore brings new problems, as it is proprietary (Adobe), requires installation of software, and is not as open as JavaScript.

Because of the problems with JavaScript, there were different attempts to use Java or server-side languages in the browser. There are mainly two different architectures for this: Google Web Toolkit (GWT) is a popular tool that allows cross-compilation from Java to JavaScript, which is of great help to build big client-side applications. Another possibility is to use applets, i.e. to run Java directly in the

browser. This second possibility though tends to become less used for many reasons (e.g., security, load times, etc).

The present paper follows the trend to try to use a server-side language in the browser. Rather than using Java, we propose to use XQuery. Furthermore, we do not intend to cross-compile XQuery to JavaScript because XQuery is much more powerful. Instead, we propose to extend Web Browsers to natively support XQuery in addition to JavaScript.

The power of XQuery is that XQuery is a family of standards which, combined, are a complete programming model. XQuery is a W3C recommendation: the XQuery expression language was standardized in 2007. XQuery Update Facility got a last call in 2007. A working group was chartered and first proposals for an XQuery Scripting Facility (e.g., XQueryP) are on the way.

XQuery was designed to process and manipulate XML. It also supports the creation of libraries. The XQuery family handles web services and is compatible with all W3C standards (XML, XQuery and XPath Data Model...). It is a Turing-complete programming model which works on documents and structured data and allows extensions with user-defined functions. In all, XQuery contains all the ingredients that are needed to implement AJAX applications. Only a few concepts (e.g., events) are missing to make XQuery appropriate to program active user interfaces; as shown in this paper, these concepts can be added seamlessly into the XQuery programming language.

One reason to propagate XQuery for client-side programming is that there is a big industry push for XQuery. A great deal of tools are developed and products are maturing. Currently, XQuery is pursued mostly in the database tier (e.g., XQuery is integrated in most modern database products such as IBM DB2, Microsoft SQL Server, and Oracle Database) and in the middle-tier (e.g., BEA's workflow engine and many content-based routing products). This paper shows how XQuery can be used at the client-tier; i.e., in the Web browser. As a result, the whole application stack across all layers can be implemented using a single programming language, XQuery, which opens up a great deal of optimization opportunities.

The contributions of our work are threefold:

- To show that XQuery is a viable option for client-side, browser-embedded applications. We present examples that show the advantages of XQuery for this purpose;
- To extend XQuery for the browser, providing a syntax for events, CSS, and the Browser Object Model;
- To extend Microsoft's Internet Explorer to support XQuery and share our experiences in the implementation of XQuery in the Internet Explorer. (An implementation in Firefox is currently on the way.)

The remainder of this paper is organized as follows: Section 2 gives examples that show the issues and discusses in more detail the limitations of JavaScript and GWT. Section 3 gives a brief overview of the XQuery family and gives examples that show why XQuery works for the examples of Section 2. Section 4 shows the proposed extensions for XQuery and lists more application areas for XQuery. Section 5 reports on our experience in integrating an open source XQuery engine into the Microsoft Internet Explorer. Section 6 lists some applications (mostly Web mashups) that we have built with XQuery in the Internet Explorer. Section 7 contains conclusions and shows avenues for future research.

2. STATE OF THE ART

JavaScript is today's biggest player for building powerful client-side applications. Other popular approaches are FLEX/Flash and the Google Web Toolkit.

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 very popular language and a lot of extensions have been added by vendors to make it more powerful than ever. AJAX (Asynchronous JavaScript And XML) is probably the best example of how JavaScript can help build powerful client-side applications. JavaScript is very good at it because it is geared towards the purpose of programming the browser.

JavaScript has some object-oriented features. It allows the manipulation of objects, i.e. field access and method call. It is not strongly typed so that it remains fairly flexible: it can handle any XML elements as objects. Hence, it supports the Document Object Model quite well.

Here is a "Hello World" program in JavaScript.

```
<html>
 <head>
  <title>JavaScript Hello World</title>
  <script type="text/javascript">
   function sayHello()
   { alert("Hello World!"); }
   document.body.addListener("onLoad", sayHello);
  </script>
 </head>
 <body/>
</html>
```

JavaScript code can be either placed between two <**script**> tags, as in this example, or in a .js file which is imported. This example defines a function sayHello(), which makes use of the Browser Object Model to raise a Hello World alert. This function is registered as a listener for the webpage loading.

JavaScript is supported natively by virtually all browser products, today. Nevertheless, there are some issues in using JavaScript as a client-side programming language.

Lack of Modularity: It is extremly difficult to build big applications because JavaScript was not tailored to that purpose: there are no libraries or library support. Extensions are currently under development as part of the AJAX alliance supported by a consortium of vendors such as BEA, Microsoft. etc.

Impedance mismatch: A lot of extensions (e.g., E4X, DOM, XPath Support) have been put on top of JavaScript in order to support more powerful and declarative DOM navigation and manipulation. These extensions have made the language less extensible for the future - not all these extensions are seamless: for example, there is an impedance mismatch between the XML data model and the JavaScript object model. Much more code is used than necessary to do simple things, e.g., navigate to a DOM node. The following example demonstrates the issue:

```
var allDivs, newElement;
allDivs = document.evaluate(
       "//*[contains(., 'Beate')]",
       document, null,
       XPathResult.
          UNORDERED_NODE_SNAPSHOT_TYPE,
       null);

if (allDivs.snapshotLength > 0) {
newElement = document.createElement('img');
newElement.src =
       'http://.../beate.gif';
document.body.insertBefore(newElement,
       document.body.firstChild);
}
```

This code uses an embedded XPath query so as to find out whether the document contains the word "Beate". It then uses the Document Object Model to create a new <**img**> element to insert an image into the body. Hence, a complex function call has to be made to perform a simple full-text search, and three functions have to be called to simply insert a new element into the body. In pure XPath/XQuery, this can be done with three lines of code as will be shown later.

Browser portability: In spite of the fact that ECMAScript, on which JavaScript is based, has been standardized, each browser has a slightly different dialect of JavaScript: this means that differences between browsers are not hidden in JavaScript. Developers often need to use tricks to either detect the browser used, or to find out whether a method is supported by the browser. For example:

```
if (document.getElementById)
    // code using this method
else
    // find another way!
```

In this code, it is tested whether the method getElementById in the document object is available. Then two different blocks handle the two possibilities.

When using AJAX, such an issue also comes up with the famous first step, creating the XMLHttpRequest object, as can be seen on the following W3Schools example [1]:

```
function createAJAX() {
  var xmlHttp;
  try {
    xmlHttp=new XMLHttpRequest();
  } catch (ex) {
    try  {
      xmlHttp=new ActiveXObject
                  ("Msxml2.XMLHTTP");
    } catch (ex) {
      try  {
        xmlHttp=new ActiveXObject
                  ("Microsoft.XMLHTTP");
      } catch (ex) {
        alert("Here we really failed.");
        return false;
}}}}
```

In this code, all possible constructs are tried, because each browser has its own one.

Client/Server portability: The code is not portable: somebody who would like to move JavaScript code to the server would have to rewrite the entire program using a server-side programming language, because JavaScript code from the browser cannot be simply copy-pasted into server code. Portability is often a requirement for security reasons or scalability.

Technology jungle: Programmers need to learn several languages to program a website. Queries against the database are made with SQL or XQuery/XPath, the business logic is developed using Java or C#, and the browser executes JavaScript. Hence, JavaScript can be embedded into ASP Code, and XPath is embedded into JavaScript, which produces highly unreadable code, as in the following example:

```
<html><body>
<%
response.write(
    "<script type='text/javascript'>"
);
response.write("var allDivs, newElement;");
response.write("allDivs = document.evaluate(
       '//*[contains(., 'Beate')]',
       document, null,
       XPathResult.
         UNORDERED_NODE_SNAPSHOT_TYPE,
       null);");
response.write("</script");
%>
</body></html>
```

In this case, we have three different programming languages (C#, JavaScript and XQuery/XPath) in the same file, four different syntaxes if we take HTML into account! Another situation could be within a JavaServer Pages file: JavaScript embedded in JSP, XPath embedded in JavaScript, Java embedded in JSP, SQL embedded in Java and XQuery/XPath embedded in SQL. This technology jungle makes the tooling (e.g. debugging) difficult as well as the optimization. Each layer could simply be performed by XQuery, which would solve this problem.

Furthermore, data is converted back and forth between XML and the data models of each language. This is exactly what happens in JavaScript using AJAX: a dynamically generated webpage is available on the server, and the XMLHttpRequest object "calls" this page, transmitting the parameters in the URL and parsing the result page to find its results. Hence, parameter passing and returning the result is done at a low level, which leads to extremly intricate code. Actually, everything could be just written with the XQuery family, using Web Services.

2.2 Flex and Flash

Flash is an integrated programming environment which produces animated vector-based graphics embedded in an SWF file. It is proprietary (earlier Macromedia, now Adobe). To read SWF files in their browsers, users need to download a proprietary plug-in.

The main advantages of Flash is that it can produce extremly elegant animation while using very small files. This is completely independent of the browser used because browser specifics are hidden by the plug-in, thereby avoiding the impedance mismatch issue.

Some drawbacks are that it does not immediately take advantage of new browser features, and it is proprietary (owned by a company). Hence, programmers are dependent

on Adobe's business model. Furthermore, it has the majority of the drawbacks of javascript as well: lack of modularity, impedance mismatch, client/server portability, technology jungle.

2.3 Google Web Tools (GWT)

There are many other approaches to program the browser: another prominent approach is the Google Web Toolkit [2]. The Google Web Toolkit is open source and written under an Apache 2.0 license.

With this toolkit, programmers can program in Java and compile their code to an AJAX application. Hence, they no longer need to worry about browser incompatibilities, and a lot of errors (type mismatch, etc) are catched by the compiler instead of been catched on the client at runtime. Furthermore, the GWT programmers can be supported by the same IDE tools as regular Java programmers, e.g. Eclipse.

With GWT, Programmers can define UI components and add them to panels. There are built-in UI components such as trees, menu bars. GWT also provides a simple framework for Remote Procedure Calls (Web Services...).

We believe that Google Web Tools correctly tackled the problem by changing the programming language, but Java is not designed for GUIs. There is neither native event-handling nor a native XML manipulation. We think that XQuery is the right language choice for client-side programming. Furthermore, we do not intend to cross-compile XQuery to JavaScript as in GWT.

3. XQUERY OVERVIEW

This section gives an overview of XQuery, the XQuery Update Facility and the XQuery Scripting Facility. In the following sections, we will simply use XQuery to denote this whole family of languages.

3.1 XQuery

XQuery is a shorthand for XML Query Language. It has been a W3C recommendation since January 2007 [4]. It is a programming language specifically tailored to natively handle XML contents. It is declarative, side-effect free (it does not modify the XML content) and functional. XQuery is Turing-complete.

XQuery is an extension of XPath: any valid XPath expression is also an XQuery expression. Like XPath, XQuery uses an XML Data Model [9] (XQuery 1.0 and XPath 2.0 Data Model). It is strongly typed, yet flexible because data can be typed as "anyType". In particular, XQuery can natively process (untyped) webpages.

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 (pronounce Flower) expression corresponds to the "'SELECT FROM WHERE"' statement in SQL.

```
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]:

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

This example is taken from [3]. It finds all authors of all the books whose title contains the word "cat", as well as the word "dog" or one 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 will be raised during the evaluation of this expression. Otherwise, this expression will be 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) [11]. 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

As explained above, the current XQuery recommendation is side-effect free: an XQuery expression cannot alter a document. The XQuery Update Facility [7] has been designed to extend XQuery torwards this purpose. The XQuery Update Facility is currently in its last call and a recommendation is expected in early 2008.

XQuery Update Facility extends XQuery so that contents can be changed: it becomes possible to insert new elements, delete elements, replace the name or content of elements:

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

Note that 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.

XQuery extended with the XQuery Update Facility is called XQuery With Updates. It provides a declarative way to update the XDM. It can thus also be used to update the DOM because DOM is XML, and thus webpages. This is exactly what is needed by AJAX applications to update a webpage.

3.3 XQuery Scripting Extension

Although XQuery With Updates allows modification of documents, there still remain some features that are not covered. For example, an expression will never see side-effects caused by former updating expressions and there is no control on the order in which instructions are executed. Side-effecting code never returns a value. There is no error handling and web services calls are not supported.

Geared towards the purpose of extending XQuery to a full-featured programming language with imperative constructs, a working group was chartered in 2007 to make a proposal for XQuery Scripting Extension [8]. XQuery Scripting extends XQuery With Updates. First internal drafts and requirements have been produced, based on XQueryP [6].

With XQueryP, instructions can see side effects of former instructions. The XQuery Scripting Extension introduces blocks (which take advantage of sequential execution) as well as variable declaration and assignment (allowing to store intermediate results in a variable which is passed along in the code).

```
declare execution sequential;
{
declare variable $myBook, $myCollection;
set $myBook := <book title="starwars"/>;
set $myCollection := <books>{$myBook}</books>;
}
```

This program is a block, whose expressions are executed sequentially. First, two variables $myBook and $myCollection are declared. Then $myBook is initialized to a book, $myCollection to a collection containing $myBook.

Blocks can be used as function bodies. The function returns the value of the block. i.e. the value of its last expression. XQueryP also introduces while loops and levels of atomicity.

The XQuery Scripting Extension also provides syntax for writing the implementation of Web Services as well as use them.

A web service is defined on the server using for example the following code:

```
service namespace ex="www.example.ch" port:2001;
declare execution sequential;
declare function ex:mul($a,$b) {$a * $b};
```

Then the webservice is imported in any program and seamlessly called, as one would call a local function.

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

3.4 Discussion

After this brief introduction of XQuery and its extensions, we would like to show why the XQuery family is a good candidate for the browser. We show that not only can XQuery do what JavaScript can, but it also solves most of the issues described in Section 2.

Big applications with XQuery are possible because of its modularity. With XQuery, namespaces allow library imports with clear separation.

The impedance mismatch issue is solved as well. We gave an example in which, if the word "Beate" is found in the body, then an image is appended to the body. The XQuery code to do the very same operation is:

```
if (/body ftcontains "Beate")
  do insert <img src="http://.../beate.gif"/>
    into /body;
```

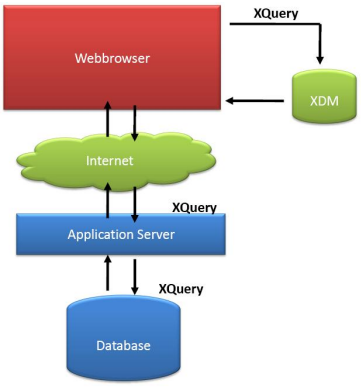

Figure 1: The simplified architecture with XQuery

which is much more concise and human-readable than the JavaScript code in Section 2.

Browser portability is solved by XQuery, as it is a W3C recommendation. In XQuery, there is no need for the method getElementById because XML is natively processed with XPath expressions. As far as AJAX and the XMLHttpRequest object are concerned, XQuery performs much more seamlessly: no XMLHttpRequest is needed, the Browser can import any web service, and call it like a local function. The AJAX paradigma is an integral part of XQuery.

The client/server portability issue as well as the technology jungle issues are solved, since XQuery executes on each layer (Client, Business Logic, against the Database), both of these problems are solved as shown in Figure 4.

4. XQUERY IN THE BROWSER

As shown in the previous section, XQuery is a good candidate for doing client-side programming: it handles XML natively and, thus, is able to process and manipulate HTML webpages. It is declarative and, as a result, many patterns can be programmed in a very concise way. At the same time, XQuery supports an imperative programming style (via its scripting extensions), for more complex applications. Remote calls via web services are supported natively. In all, XQuery is a good starting point for browser-embedded applications. What is missing to implement real AJAX-style applications in XQuery are event handling, asynchronous calls, and CSS handling. Fortunately, XQuery can easily be extended to support these concepts. This section lists the proposed extensions and gives examples.

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 predefined functions.

XQuery can co-exist with JavaScript (just like VBScript and JavaScript can co-exist), which means that a webpage may contain both XQuery and JavaScript.

The processing model is the following: XQuery (and the JavaScript) code is executed at the beginning, thereby registering events according to the DOM level 3 standard [10]. Thus, the browser listens for these events and executes the

XQuery (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. This code has the same structure and effect as the JavaScript "Hello World" example of section 2.1.

```
<html>
 <head>
  <title>Hello World with XQuery
      in the Browser</title>
  <script type="text/xquery">
   declare function SayHello() as () {
     browser:alert("Hello, World!");
   }
   on event "onLoad" at //body
      attach listener SayHello:
  </script>
 </head>
 <body/>
</html>
```

Again, the XQuery code is embed into a <**script**/> tag of the HTML webpage. In the header, an XQuery function is declared, which takes no arguments and returns nothing, and just pops up a "Hello, World!" message. This function is called when the page is loaded (onLoad event).

The examples in the next subsection show that XQuery is as powerful 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 [11]. Embedding XQuery into the browser, extends the context with one additional namespace (i.e., "http://www.example.com/xbrowser" which is bound to the prefix "browser"). Furthermore, the context includes browser-specific schemas and functions. For instance, the "browser:window()" 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 object can be implemented in XQuery using the "browser:window()" function. The "browser:window() function returns an XML element of the following form (for the sake of simplicity, we omit namespaces):

```
<windows>
 <window name="win1">
  <defaultstatus>Welcome</defaultstatus>
  <frame name="child1">
   <defaultstatus>Welcome to the first
                  child</defaultstatus>
   ...
  </frame>
  <frame name="child2">
   <defaultstatus>Welcome to the second
                  child</defaultstatus>
   ...
  </frame>
  <location>
   <href>http://www.dbis.ethz.ch"</href>
  </location>
  ...
 </window>
 <window name="win2">
  <defaultstatus>I am open in a separate window
              and have no frames</defaultstatus>
  <location>
   <href>http://www.dbis.ethz.ch/other"</href>
  </location>
  ...
 </window>
</windows>
```

In order to retrieve certain properties (e.g., a frame), it is possible to navigate through this element using XQuery. (Recall that XQuery is a superset of XPath.) Furthermore, the "windows" 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):

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

looks for a child of the top window whose name is "leftframe".

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

changes the default status of the current window to "Welcome".

```
do replace value of browser:self()/status
    with "This is a transient message";
(: JavaScript: self.status = "..." :)
```

changes the status of the current window to "This is a transient message".

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

declares a new variable $win initialized to the second child of the current window.

```
$win/location/host
(: JavaScript: win.location.host :)
```

accesses the host of the webpage displayed in the variable $win which is bound as in the previous example.

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

changes the location of the variable $win (as above). This code causes a new webpage to be displayed.

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

raises an alert giving the date and time the variable $win bound above was last modified.

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

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

looks for any frame (children, grandchildren... of top()) whose name is "myframe".

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

This FLWOR expression writes a big red warning on every frame not pointing to an https location. The last two examples are difficult to implement using JavaScript.

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 virtual 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 webpage is a self-contained XML document. To access the document displayed in a window (assuming the windows is in the variable $win), one uses the function call browser:doc($win) which return a document node.

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

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

```
html/body//div
```

and all images in a children window with

```
browser:doc(browser:self()/frames[2])/html//img
```

4.2.4 The browser functions

There are some further browser built-in functions available. We list some of them here and how they can be called using XQuery. We omit comments as the names are self-explanatory and the names are similar to equivalent JavaScript functions. For more information, see [12], Chapter 5.

- Window-related functions

```
- browser:moveBy($win $dx, $dy)
- browser:moveTo($win, $dx, $dy)

- set $newwin := browser:open($url,
                 $name, $features, $bool)
- browser:close($newwin);

- browser:alert("Hello, World")
- browser:prompt("What's your name?", "Mickey")
- browser:confirm("Are you sure?")
```

- History-related functions

```
- browser:historyback();
- browser:historyforward();
- browser:historygo(-1);
```

- Document-related functions (note that with XQuery, best practice would be to modify the XDM and avoid using those functions).

```
- browser:write($doc, $str)
- browser:writeln($doc, $str)
```

Although with XQuery, the code becomes browser-independent, the programmer might 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. In JavaScript, there are two ways to register events. The first, simple way is to use e.g. the onclick, onload properties. The second way is to add listeners manually.

In XQuery, there could be a pre-defined function in the browser context which supports the registration of events, but so as to avoid using higher-order functions (to pass the listener as an argument), we propose to extend the XQuery syntax instead, because event-handling is a very important part of browser programming.

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 register:

```
declare function local:myEventListener
    ($event as element(browser:event))
    as xs:boolean
{
  declare $message = <message>Event occured:
      {$event/ev:type}
      at {$event/ev:target}
    </message>;

  return browser:alert(data($message));
}
```

This very simple function raises a message box with information on the event.

In JavaScript, we register events with:

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

For XQuery, we propose the following syntax to register events:

- For registering an event:

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

- For deregistering an event:

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

- For triggering an event:

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

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
                ["with" "capturing"]

EventDetach ::= "on" "event" ExprSingle
                ("at"|"behind") ExprSingle
                "detach" "listener" QName
                ["with" "capturing"]

EventTrigger ::= "trigger" "event" ExprSingle
                "at" ExprSingle
```

The "behind" construct is explained later in section 4.4

4.3.2 Event Node

In our example, the listener takes a parameter $event of type element(browser:event). As for windows handling, this parameter is actually a virtual XML element. It is a new type from the browser context, which contains informations about the event, for example whether the alt key was pressed, which mouse button was used... the same informations that are available in an Event Object in the DOM [12].

As in the DOM, there are several event properties which can be queried:

```
$event/ev:target, $event/ev:type, $event/ev:altKey,
$event/ev:button...
```

so that one can adapt the behavior of the listener:

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

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 service namespace
      ab = "http://example.com"
      from "http://www.example.com/wsdl";

    declare function my:showHint($str as xs:string)
    {
      if(fn:length($str)) = 0)
        do replace value of //*[@id="txtHint"]
          with ();
      else {
        on event "stateChanged"
          behind ab:getHint($str)
          attach listener my:stateChanged;
      }
    }
    declare function my:stateChanged($event)
     {
```

```
      if($event/readyState = 4)
        do replace value of //*[@id="txtHint"]
            with $event/ev:response;
      }
    </script>
  </head>
  <body>
    <form>
    First Name:
    <input type="text" id="text1"
    onkeyup="my: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, my:showHint(value) is called. 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 that is returned from ab:showHint, 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 using the following code:

```
replace value of \\table[@id="thistable"]/@style
    with "visibility: visible";
```

Actually, this is not very flexible, for example one cannot easily change an existing style property, which would require several lines of code. So we forbid this way of modifying the style of an element and propose a syntax extension because it is very common.

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;
$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
```

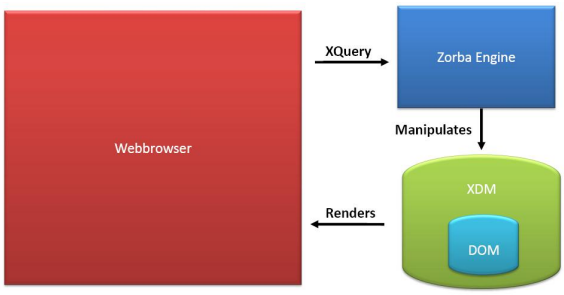

Figure 2: Interaction with the XDM and the DOM

5. IMPLEMENTATION

For the implementation, we chose to build an extension for Microsoft Internet Explorer. An implementation for Firefox is under construction as well, in collaboration with France Telecom.

5.1 Architecture

Our browser extension works as depicted in Figure 2:

1. The browser receives an XHTML document and parses it.

2. The browser generates the DOM, renders the webpage and initializes the extension.

3. The XQuery engine receives the XQuery code from the website as a string. In this work, we used Zorba as a query engine. 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.

4. The XQuery Engine implements the XDM on top of the DOM. It means that modifying the XDM using e.g. XPath will modify the DOM accordingly.

5. Zorba modifies the XDM. Implicitly, this modifies the Microsoft Internet Explorer DOM, because the XDM is implemented on top of that.

5.2 Zorba on IE's DOM

XQuery in the Browser implements some Zorba specific interfaces, that allows Zorba to use the Microsoft implementation of the Document Object Model.

XBrowser uses classes to represent a node (abstract), a document node, an element node, an attribute node, a comment node and a text node. In the XQuery Data Model, there are two other node types defined: one to represent processing instructions and one to represent namespaces. These two are not implemented yet because the Internet Explorer specific DOM implementation does not allow to get any information about processing instructions and because namespaces are represented in Zorba with qualified names and not with namespace nodes. There is ongoing work to implement namespace nodes.

5.3 Experience

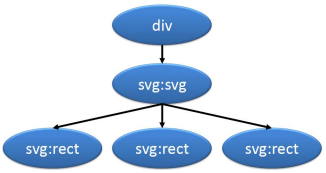

Figure 3: A tree as it should appear

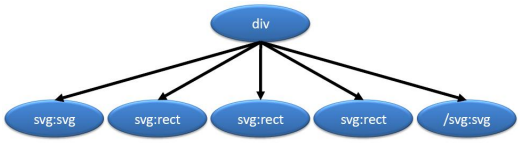

Figure 4: A tree as it actually appears with IE

5.3.1 *Performance*

To get a good performance, the plugin tries to be as lazy as possible. At initialisation, only a document node is created. By getting an iterator for the children or attributes the items are created at every iteration step.

But of course, generating a whole branch for every query does not perform well, so it is better to held the created nodes in the memory. The implementation of the namespace standard [5] makes this more difficult than it seems to be in the first second because of the way, the namespaces are implemented:

By stepping across the tree, any node pushes the namespaces and prefixes on a globaly managed stack. With that stack, it is easy for a node to solve the used prefixes. But it is necessary to get a new stack for every query. So it is possible, that one wants to get the children of a node that does not have access to a stack. But because it has a pointer to the parent, the node can rebuild that by recursion.

5.3.2 *Bugs*

While XML has namespace support, Microsoft Internet Explorer does not seem to support them. So the whole W3C-Standard had to be implemented. In XML, to declare a namespace, one uses a xmlns: construct as an attribute of an element node. These attributes can be retrieved from the Internet Explorer DOM. But by mischance, Internet Explorer makes a horrible mistake when it parses a XML document that makes use of namespaces: it flatters the hierachy of the tree - starting at the node a namespace was declared. Figure 4 illustrates a simple tree, which could be part of an XML document and uses namespaces. But Microsoft DOM does not give us the structure in Figure 3 but in Figure 4.

First this seemed to be a very big show stopper. But by looking at Figure 4, one can see another, interesting bug: in this example IE adds another, nonexisting node "/svg:svg". With the help of that bug, it was possible to reconstruct the structure from the IE DOM.

6. APPLICATION SCENARIOS

Some applications of XQueryP (the early drafts used by XQuery Update Facility) have been performed in labs at ETH Zurich and Stanford University: ZVV mashup, celebrity mashup etc, we are partly also able to run e.g. on mobile phones.

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, but appears to have some limitations. The main one is the code overhead because so many layers have been implemented on top of JavaScript to allow it to modifiy the DOM. Because XQuery supports XML manipulation natively, it is a viable candidate to program the browser which solves many problems of JavaScript as well as such alternatives like Google Web Toolkit. Extending the browser (Internet Explorer) can be done with reasonable effort, and there are some interesting applications (mostly Web mash-ups are on the way).

Future work is to make it a product for download (public release March 2008), to port it to firefox (which is going to be hopefully easier because firefox has better DOM standard compliance). More applications are going to be developed so as to validate this work (we started doing this in labs at ETH Zurich). Libraries should be developed, e.g., widget libraries. In fact, we have two years of AJAX libraries to catch up. We also need to integrate database and persistent states (alla Google Gears). Ultimately, we will have to convince browser vendors to support it natively or at least ship it with a built-in extension.

8. REFERENCES

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

[2] Google Web Toolkit - Build AJAX apps in the Java language. `http://code.google.com/webtoolkit/`.

[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] T. Bray, D. Hollander, A. Layman, and R. Tobin. Namespaces in XML 1.1. `http://www.w3.org/TR/2006/REC-xml-names11-20060816/`.

[6] D. Chamberlin, M. Carey, D. Florescu, D. Kossmann, and J. Robbie. XQueryP: Programming with XQuery. In *XML2006*, 2006.

[7] D. Chamberlin, D. Florescu, and J. Robbie. XQuery Update Facility. `http://www.w3.org/TR/xqupdate/`.

[8] D. Engovatov, D. Florescu, and G. Ghelli. XQuery Scripting Extension 1.0 Requirements. `http://www.w3.org/TR/xquery-sx-10-requirements/`.

[9] 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/`.

[10] 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/`.

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

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