XML on the Web: is it still relevant?

O'Neil Delpratt, Saxonica <oneil@saxonica.com>

Table of Contents

In this paper we discuss what it means by the term *XML on the Web* and how this relates to the browser. The success of XSLT in the browser has so far been underwhelming, and we examine the reasons for this and consider whether the situation might change. We describe the capabilities of the first XSLT 2.0 processor designed to run within web browsers, bringing not just the extra capabilities of a new version of XSLT, but also a new way of thinking about how XSLT can be used to create interactive client-side applications. Using this processor we demonstrate as a use-case a technical documentation application, which permits browsing and searching in a intuitive way. We show its internals to illustrate how it works.

Introduction

The W3C introduced Extensible Markup Language (XML) as a multi-purpose and platform-neutral text-based format, used for storage, transmission and manipulation of data. Fifteen years later, it has matured and developers and users use it to represent their complex and hierarchically structured data in a variety of technologies. Its usage has reached much further than its creators may have anticipated.

In popular parlance 'XML on the Web' means 'XML in the browser'. There's a great deal of XML on the web, but most of it never reaches a browser: it's converted server-side to HTML using a variety of technologies ranging from XSLT and XQuery to languages such Java, C#, PHP and Perl. But since the early days, XML has been seen as a powerful complement to HTML and as a replacement in the form of XHTML. But why did this not take off and revolutionise the web? And could this yet happen?

XML has been very successful, and it's useful to remind ourselves why:

- XML can handle both data and documents.
- XML is human-readable (which makes it easy to develop applications).
- XML handles Unicode.
- XML was supported by all the major industry players and available on all platforms.
- XML was cheap to implement: lots of free software, fast learning curve.
- There was a rich selection of complementary technologies.
- The standard acquired critical mass very quickly, and once this happens, any technical deficiencies become unimportant.

However, this success has not been gained in the browser. Again, it's a good idea to look at the reasons:

- HTML already established as a defacto standard for web development
- The combination of HTML, CSS, and Javascript was becoming ever more powerful.
- It took a long while before XSLT 1.0 was available on a sufficient range of browsers.
- When XSLT 1.0 did eventually become sufficiently ubiquitous, the web had moved on ("Web 2.0").
- XML rejected the "be liberal in what you accept" culture of the browser.

One could look for more specific technical reasons, but they aren't convincing. Some programmers find the XSLT learning curve a burden, for example, but there are plenty of technologies with an equally daunting learning curve that prove successful, provided developers have the incentive and motivation to put the effort in. Or one could cite the number of people who encounter problems with ill-formed or mis-encoded XML, but that problem is hardly unique to XML. Debugging Javascript in the browser, after all, is hardly child's play.

XSLT 1.0 was published in 1999 [1]. The original aim was that it should be possible to use the language to convert XML documents to HTML for rendering on the browser 'client-side'. This aim has largely been achieved. Before the specification was finished Microsoft implemented XSLT 1.0 as an add-on to Internet Explorer (IE) 4, which became an integral part of IE5. (Microsoft made a false start by implementing a draft of the W3C spec that proved significantly different from the final Recommendation, which didn't help.) It then took a long time before XSLT processors with a sufficient level of conformance and performance were available across all common browsers. In the first couple of years the problem was old browsers that didn't have XSLT support; then the problem became new browsers that didn't have XSLT support. In the heady days while Firefox market share was growing exponentially, its XSLT support was very weak. More recently, some mobile browsers have appeared on the scene with similar problems.

By the time XSLT 1.0 conformance across browsers was substiantially achieved (say around 2009), other technologies had changed the goals for browser vendors. The emergence of XSLT 2.0 [2], which made big strides over XSLT 1.0 in terms of developer productivity, never attracted any enthusiasm from the browser vendors - and the browser platforms were sufficiently closed that there appeared to be little scope for third-party implementations.

The "Web 2.0" movement was all about changing the web from supporting read-only documents to supporting interactive applications. The key component was AJAX: the X stood for "XML", but Javascript and XML never worked all that well together. DOM programming is tedious. AJAX suffers from "Impedence mismatch" - it's a bad idea to use programming languages whose type system doesn't match your data.

That led to what we might call AJAJ - Javascript programs processing JSON data. Which is fine if your data fits the JSON model. But not all data does, especially documents. JSON has made enormous headway in making it easier for Javascript programmers to handle structured data, simply because the data doesn't need to be converted from one data model to another. But for many of the things where XML has had most success - for example, authoring scientific papers like this one, or capturing narrative and semi-structured information about people, places, projects, plants, or poisons - JSON is simply a non-starter.

So the alternative is AXAX - instead of replacing XML with JSON, replace Javascript with XSLT or XQuery. The acronym that has caught on is XRX, but AXAX better captures the relationship with its alternatives. The key principle of XRX is to use the XML data model and XML-based processing languages end-to-end, and the key benefit is the same as the "AJAJ" or Javascript-JSON model - the data never needs to be converted from one data model to another. The difference is that this time, we are dealing with a data model that can handle narrative text.

A few years ago it seemed likely that XML would go no further in the browser. The browser vendors had no interest in developing it further, and the browser platform was so tightly closed that it wasn't feasible for a third party to tackle. Plug-ins and applets as extension technologies were largely discredited. But paradoxically, the browser vendors' investment in Javascript provided the platform that could change this. Javascript was never designed as a system programming language, or as a target language for compilers to translate into, but that is what it has become, and it does the job surprisingly well. Above all else, it is astoundingly fast.

Google were one of the first to realise this, and responded by developing Google Web Toolkit (GWT) [3] as a Java-to-Javascript bridge technology. GWT allows web applications to be developed in Java (a language which in many ways is much better suited for the task than Javascript) and then cross-compiled to Javascript for execution on the browser. It provides most of the APIs familiar to Java programmers in other environments, and supplements these with APIs offering access to the more specific services available in the browser world, for example access to the HTML DOM, the Window object, and user interface events.

Because the Saxon XSLT 2.0 processor is written in Java, this gave us the opportunity to create a browser-based XSLT 2.0 processor by cutting down Saxon to its essentials and cross-compiling using GWT.

We realized early on that simply offering XSLT 2.0 was not enough. Sure, there was a core of people using XSLT 1.0 who would benefit from the extra capability and productivity of the 2.0 version of the language. But it was never going to succeed using the old architectural model: generate an HTML page, display it, and walk away, leaving all the interesting interactive parts of the application to be written in Javascript. XRX (or AXAX, if you prefer) requires XML technologies to be used throughout, and that means replacing Javascript not only for content rendition (much of which can be done with CSS anyway), but more importantly for user interaction. And it just so happens that the processing model for handling user interaction is event-based programming, and XSLT is an event-based programming language, so the opportunities are obvious.

In this paper we examine the first implementation of XSLT 2.0 on the browser, Saxon-CE [4]. We show how Saxon-CE can be used as a complement to Javascript, given its advancements in performance and ease of use. We also show that Saxon-CE can be used as a replacement of JavaScript. This we show with an example of a browsing and searching technical documentation.

This is classic XSLT territory, and the requirement is traditionally met by server-side HTML generation, either in advance at publishing time, or on demand through servlets or equivalent server-side processing that invoke XSLT transformations, perhaps with some caching. While this is good enough for many purposes, it falls short of what users had already learned to expect from desktop help systems, most obviously in the absence of a well-integrated search capability. Even this kind of application can benefit from Web 2.0 thinking, and we will show how the user experience can be improved by moving the XSLT processing to the client side and taking advantage of some of the new facilities to handle user interaction.

In our conference paper and talk we will explain the principles outlined above, and then illustrate how these principles have been achieved in practice by reference to a live application: we will demonstrate the application and show its internals to illustrate how it works.

XSLT 2.0 on the browser

In this section we begin with some discussion on the usability of Saxon-CE before we give an overview of its internals. Saxon-CE has matured significantly since its first production release (1.0) in June 2012, following on from two earlier public beta releases. The current release (1.1) is dated February 2013, and the main change is that the product is now released under an open source license (Mozilla Public License 2.0).

Saxon-CE Key Features

Beyond being a conformant and fast implementation of XSLT 2.0, Saxon-CE has a number of features specially designed for the browser, which we now discuss:

1. *Handling JavaScript Events in XSLT*: Saxon-CE is not simply an XSLT 2.0 processor running in the browser, doing the kind of things that an XSLT 1.0 processor did, but with more language features (though that in itself is a great step forward). It also takes XSLT into the world of

interactive programming. With Saxon-CE it's not just a question of translating XML into HTML-plus-JavaScript and then doing all the interesting user interaction in the JavaScript; instead, user input and interaction is handled directly within the XSLT code. The XSLT code snippet illustrates the use of event handling:

```
<xsl:template match="p[@class eq 'arrowNone']" mode="ixsl:onclick">
    <xsl:if test="$usesclick">
        <xsl:for-each select="$navlist/ul/li">
        <ixsl:set-attribute name="class" select="'closed'"/>
        </xsl:for-each>
    </xsl:if>
</xsl:template>
```

XSLT is ideally suited for handling events. It's a language whose basic approach is to define rules that respond to events by constructing XML or HTML content. It's a natural extension of the language to make template rules respond to input events rather than only to parsing events. The functional and declarative nature of the language makes it ideally suited to this role, eliminating many of the bugs that plague JavaScript development.

2. *Working with JavaScript Functions*: The code snippets below illustrates a JavaScript function, which gets data from an external feed:

   ```
   var getTwitterTimeline = function(userName)
   {
       try {
           return makeRequest(timelineUri + userName);
       }
       catch(e) {
           console.log("Error in getTwitterTimeline: " + e);
           return "";
       }
   };
   ```

   Here is some XSLT code showing how the JavaScript function can be used; this is a call to the `getTwitterTimeline` function in XSLT 2.0 using Saxon-CE. The XML document returned is then passed as a parameter to the a JavaScript API function `ixsl:parse-xml`:

   ```
   <xsl:variable name="tw-response" as="document-node()"
      select="ixsl:parse-xml(js:getTwitterTimeline($username))"/>
   ```

3. *Animation*: The extension instruction ixsl:schedule-action may be used to achieve animation. The body of the instruction must be a single call on `<xsl:call-template/>`, which is done asynchronously. If an action is to take place repeatedly, then each action should trigger the next by making another call on `<ixsl:schedule-action />`

4. *Interactive XSLT*: There are a number of Saxon-CE defined functions and instructions which are available. One indispensable useful function is the `ixsl:page()`, which returns the document node of the HTML DOM document. An example of this function's usage is given as follows. Here we retrieve a `div` element with a given predicate and bind it to an XSLT variable:

   ```
   <xsl:variable name="movePiece" as="element(div)"
          select="if (exists($piece)) then $piece
                  else id('board',ixsl:page())/div[$moveFrom]/div"/>
   ```

   In the example below, we show how to set the style property using the extension intruction `ixsl:set-attribute` for a current node in the HTML page. Here we are changing the display property to 'none', which hides an element, causing it not to take up any space on the page:

```
<xsl:if test="position() &gt; $row-size">
 <ixsl:set-attribute name="style:display" select="'none'"/>
 </xsl:if>
```

In the example below we show how we can get the property of a JavaScript object by using the ixsl:get function:

```
<xsl:variable name="piece" as="element(div)"
  select="ixsl:get(ixsl:window(), 'dragController.piece')"/>
```

The full list of the extension functions and extension instructions in Saxon-CE can be found at the following location: *http://www.saxonica.com/ce/user-doc/1.1/index.html#!coding/extensions* [http://www.saxonica.com/ce/user-doc/1.1/index.html#!coding/extensions] and *http://www.saxonica.com/ce/user-doc/1.1/index.html#!coding/extension-instructions* [http://www.saxonica.com/ce/user-doc/1.1/index.html#!coding/extension-instructions]

Saxon-CE Internals

In this section we discuss how we build the client-side XSLT 2.0 processor and how we can invoke it from JavaScript, XML or HTML. The Java code base was inherited from Saxon-HE, the successful XSLT 2.0 processor for Java. The product was produced by cross-compiling the Java into optimized, stand-alone JavaScript files using the GWT 5.2. Although no detailed performance data is available here, all deliver a responsiveness which feels perfectly adequate for production use. The JavaScript runs on all major browsers, as well as on mobile browsers, where JavaScript can run.

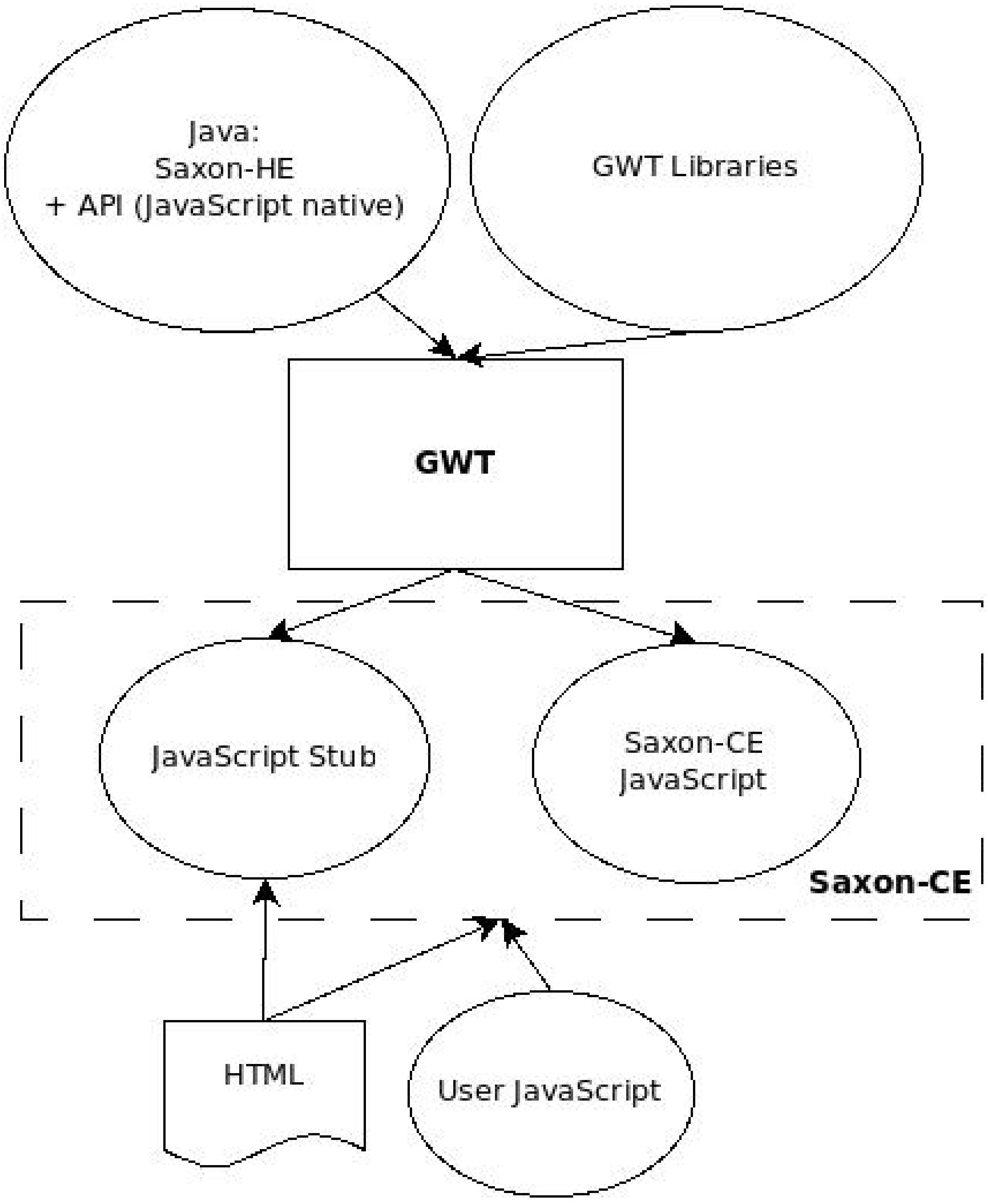
The key achievements in the development of Saxon-CE are given below:

- The size of the Java source was cut down to around 76K lines of Java code. This was mainly achieved by cutting out unwanted functionality such as XQuery, updates, serialization, support for JAXP, support for external object models such as JDOM and DOM4J, Java extension functions, and unnecessary options like the choice between TinyTree and Linked Tree, or the choice (never in practice exercised) of different sorting algorithms. Some internal hanges to the code base were also made to reduce size. Examples include changes to the XPath parser to use a hybrid precedence-parsing approach in place of the pure recursive-descent parser used previously; offloading the data tables used by the `normalize-unicode()` function into an XML data file to be loaded from the server on the rare occasions that this function is actually used.
- GWT creates a slightly different JavaScript file for each major browser, to accommodate browser variations. Only one of these files is downloaded, which is based on the browser that is in use. The size of the JavaScript file is around 900KB.
- The key benefits of the server-side XSLT 2.0 processor were retained and delivered on the browser. Saxon has a reputation for conformance, usability, and performance, and it was important to retain this, as well as delivering the much-needed functionality offered by the language specification. Creating automated test suites suitable for running in the browser environment was a significant challenge.
- Support of JavaScript events. The handling of JavaScript events changes the scope of Saxon-CE greatly, meaning it can be used for interactive application development. Our first attempts to integrate event handling proved the design of the language extensions was sound, but did not perform adequately, and the event handling is the final product was a complete rewrite. The Events arising from the HTML DOM and the client system, which are understood by GWT, are handled via Saxon-CE. This proxying of event handling in the Java code makes it possible for template rules which have a mode matching the event to overide the default behavour of the browser. Events are only collected at the document node (thus there's only one listener for each type of event). As a result, the events are bubbled up from the event target. This mechanism handles the majority of browser events. There are a few specialist events like `onfocus` and `onblur` which do not operate at the document node, and these events are best handled in JavaScript first. GWT provides relatively poor support for these events because their behaviour is not consistent across different browsers.

- Interoperability with JavaScript. Many simple applications can be developed with no user-written Javascript. Equally, where Javascript skills or components are available, it is possible to make use of them, and when external services are avalable only via Javascript interfaces, Saxon-CE stylesheets can still make use of them.

Figure 1 illustrates the input and output components involved in building the XSLT 2.0 processor, Saxon-CE:

Figure 1. Saxon-CE Development



Static view of the Saxon-CE product and components involved in the build process

As shown in Figure 1 we use GWT to cross-compile the XSLT 2.0 processor. Saxon-HE and GWT libraries are input to this process. In addition, we write the JavaScript API methods in Java using the JavaScript Native Interface (JSNI), which is a GWT feature. This feature proved useful because it provided access to the low-level browser functionality not exposed by the standard GWT APIs. This in effect provides the interface for passing and returning JavaScript objects to and from the XSLT processor.

The output from this process is Saxon-CE, which comprises of the XSLT 2.0 processor and the stub file, both in highly compressed and obfuscated JavaScript. GWT provides separate JavaScript files for each major browser. User JavaScript code can happily run alongside the XSLT processor.

The invoking of Saxon-CE is achieved in several ways. The first method employs a standard `<?xml-stylesheet?>` processing-instruction in the prolog of an XML document. This cannot be used to invoke Saxon-CE directly, because the browser knows nothing of Saxon-CE's existence. Instead, however, it can be used to load an XSLT 1.0 bootstrap stylesheet, which in turn causes Saxon-CE to be loaded. This provides the easiest upgrade from existing XSLT 1.0 applications. The code snippet below illustrates the bootstrap process of the XSLT 2.0 processor:

```
<?xml-stylesheet type="text/xsl" href="sample.boot.xsl"?>
<dt:data-set xmlns:dt="urn:system.logging.data.xml">
  <dt:rows name="test-data">
  ...
</dt:data-set>
```

The XSLT 1.0 bootstrap stylesheet is given below. It generates an HTML page containing instructions to load Saxon-CE and execute the real XSLT 2.0 stylesheet:

```
<xsl:transform
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     version="1.0">

    <xsl:output method="html" indent="no"/>
    <xsl:template match="/">
        <html>
            <head>
                <meta http-equiv="Content-Type" content="text/html" />
                <script type="text/javascript" language="javascript"
                        src="../Saxonce/Saxonce.nocache.js"/>

                <script>
                    var onSaxonLoad = function() {
                        Saxon.run( {
                             source:     location.href,
                             logLevel:   "SEVERE",
                             stylesheet: "sample.xsl"
                        });
                    }
                </script>

            </head>
            <!-- these elements are required also -->
            <body><p></p></body>
        </html>
    </xsl:template>

</xsl:transform>
```

The second method involves use of the script element in HTML. In fact there are two script elements: one with type="text/javascript" which causes the Saxon-CE engine to be loaded, and the other with type="application/xslt+xml" which loads the stylesheet itself, as shown here:

```
<script type="application/xslt+xml" language="xslt2.0" src="books.xsl" data-sou
```

The third method is to use an API from Javascript. The API is modelled on the XSLTProcessor API provided by the major browsers for XSLT 1.0.

We discussed earlier that the JavaScript API provides an API with a rich set of features for interfacing and invoking the XSLT processor when developing Saxon-CE applications. There are three JavaScript API Sections available: The *Command*, which is designed to be used as a JavaScript literal object and effectively wraps the Saxon-CE API with a set of properties so you can run an XSLT transform on an HTML page in a more declarative way; the *Saxon* object, which is a static object, providing a set of utility functions for working with the XSLT processor, initiating a simple XSLT function, and working with XML resources and configuration; and the *XSLT20Processor*, which is modeled on the JavaScript XSLTProcessor API as implemented by the major browsers. It provides a set of methods used to initiate XSLT transforms on XML or direct XSLT-based HTML updates.

The code snippet below shows a Command API call to run a XSLT transform. Here the stylesheet is declared as `ChessGame.xsl` and the initial template is defined as `main`. We observed the `logLevel` as been set to 'SEVERE'. Saxon-CE provides a debug version which ouputs useful log messages to the JavaScript console, accessable in the browser development tools:

```
var onSaxonLoad = function() {

    proc = Saxon.run( {
        stylesheet:   'ChessGame.xsl',
        initialTemplate: 'main',
        logLevel:     'SEVERE'
    } );

};
```

Use Case: Technical documentation application

We now examine a Saxon-CE driven application used for browsing technical documentation in a intuative manner: specifically, it is used for display of the Saxon 9.5 documentation on the Saxonica web site. The application is designed to operate as a desktop application, but on the web.

The documentation for Saxon 9.5 can be found at:

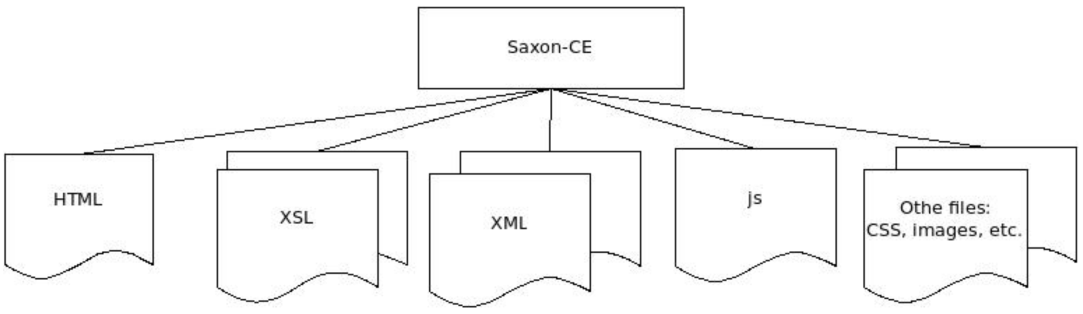
- *http://www.saxonica.com/documentation/index.html* [http://www.saxonica.com/documentation/index.html]

When you click on this link for the first time, there will be a delay of a few seconds, with a comfort message telling you that Saxon is loading the documentation. This is not strictly accurate; what is actually happening is that Saxon-CE itself is being downloaded from the web site. This only happens once; thereafter it will be picked up from the browser cache. However, it is remarkable how fast this happens even the first time, considering that the browser is downloading the entire Saxon-CE product (900Kb of Javascript source code generated from around 76K lines of Java), compiling this, and then executing it before it can even start compiling and executing the XSLT code.

Architecture

The architecture of the technical documentation application is shown in Figure 2:

Figure 2. Architecture of Technical Documentation application



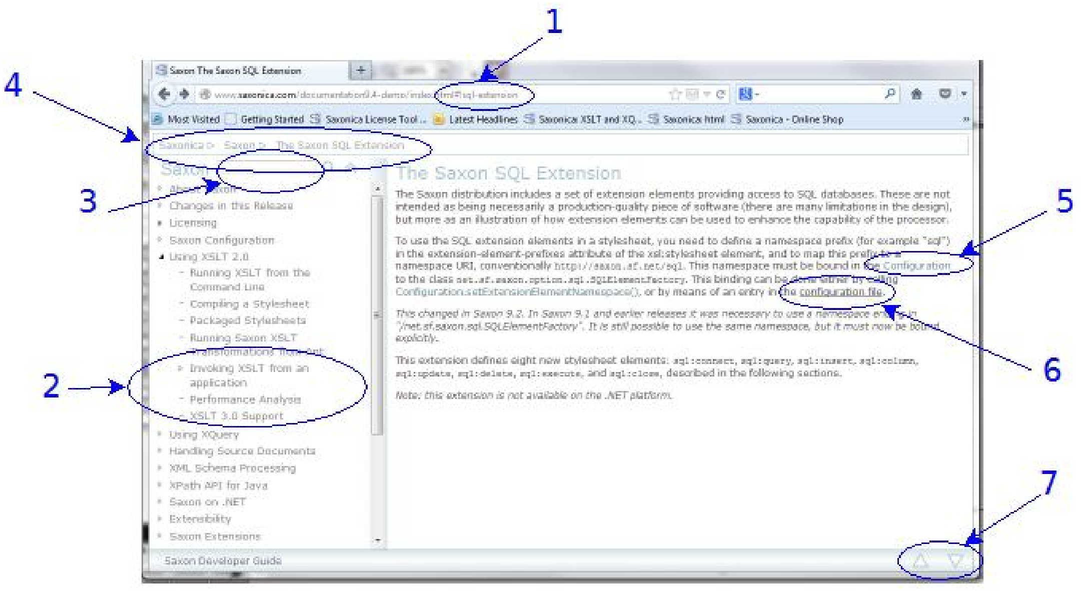
Architectural view of a Saxon-CE application

The application consists of a number of XML documents representing the content data, ten XSLT 2.0 modules, a Javascript file, several other files (CSS file, icon and image files) and a single skeleton HTML webpage; the invariant parts of the display are recorded directly in HTML markup, and the variable parts are marked by empty <div> elements whose content is controlled from the XSLT stylesheets. Development with Saxon-CE often eliminates the need for Javascript, but at the same time it happily can be mixed with calls from XSLT. In this case it was useful to abstract certain JavaScript functions used by the Saxon-CE XSLT transforms.

Key to this application is that the documentation content data are all stored as XML files. Even the linkage of files to the application is achieved by a XML file called catalog.xml: this is a special file used by the XSLT to render the table of contents. The separation of the content data from the user interface means that changes to the design can be done seamlessly without modifiying the content, and vice versa.

The documentation is presented in the form of a single-page web site. The screenshot in Figure 3 shows its appearance.

Figure 3. Technical documentation application in the browser



The Saxon SQL Extension

The Saxon distribution includes a set of extension elements providing access to SQL databases. These are not intended as being necessarily a production-quality piece of software (there are many limitations in the design), but more as an illustration of how extension elements can be used to enhance the capability of the processor.

To use the SQL extension elements in a stylesheet, you need to define a namespace prefix (for example "sql") in the extension-element-prefixes attribute of the xsl:stylesheet element, and to map this prefix to a namespace URI, conventionally `http://saxon.sf.net/sql`. This namespace must be bound in the Configuration to the class `net.sf.saxon.option.sql.SQLElementFactory`. This binding can be done either by calling Configuration.setExtensionElementNamespace(), or by means of an entry in the configuration file.

This changed in Saxon 9.2. In Saxon 9.1 and earlier releases it was necessary to use a namespace ending in "/net.sf.saxon.sql.SQLElementFactory". It is still possible to use the same namespace, but it must now be bound explicitly.

This extension defines eight new stylesheet elements: `sql:connect`, `sql:query`, `sql:insert`, `sql:column`, `sql:update`, `sql:delete`, `sql:execute`, and `sql:close`, described in the following sections.

Note: this extension is not available on the .NET platform.

Screen-shot of the Technical documentation in the browser using Saxon-CE

Note the following features, called out on the diagram. We will discuss below how these are implemented in Saxon-CE.

1. The fragment identifier in the URL

2. Table of contents
3. Search box
4. Breadcrumbs
5. Links to Javadoc definitions
6. Links to other pages in the documentation
7. The up/down buttons

XML on the Server

This application has no server-side logic; everything on the server is static content.

On the server, the content is held as a set of XML files. Because the content is fairly substantial (2Mb of XML, excluding the Javadoc, which is discussed later), it's not held as a single XML document, but as a set of a 20 or so documents, one per chapter. On initial loading, we load only the first chapter, plus a small catalogue document listing the other chapters; subsequent chapters are fetched on demand, when first referenced, or when the user does a search.

Our first idea was to hold the XML in DocBook form, and use a customization of the DocBook stylesheets to present the content in Saxon-CE. This proved infeasible: the DocBook stylesheets are so large that downloading them and compiling them gave unacceptable performance. In fact, when we looked at the XML vocabulary we were actually using for the documentation, it needed only a tiny subset of what DocBook offered. We thought of defining a DocBook subset, but then we realised that all the elements we were using could be easily represented in HTML5 without any serious tag abuse (the content that appears in highlighted boxes, for example, is tagged as an `<aside>`). So the format we are using for the XML is in fact XHTML 5. This has a couple of immediate benefits: it means we can use the HTML DOM in the browser to hold the information (rather than the XML DOM), and it means that every element in our source content has a default rendition in the browser, which in many cases (with a little help from CSS) is quite adequate for our purposes.

Although XHTML 5 is used for the narrative part of the documentation, more specialized formats are used for the parts that have more structure. In particular, there is an XML document containing a catalog of XPath functions (both standard W3C functions, and Saxon-specific extension functions) which is held in a custom XML vocabulary; and the documentation also includes full Javadoc API specifications for the Saxon code base. This was produced from the Java source code using the standard Javadoc utility along with a custom "doclet" (user hook) causing it to generate XML rather than HTML. The Javadoc in XML format is then rendered by the client-side stylesheets in a similar way to the rest of the documentation, allowing functionality such as searching to be fully integrated. For the .NET API, we wrote our own equivalent to Javadoc to produce class and method specifications in the same XML format.

The fact that XHTML is used as the delivered documentation format does not mean, of course, that the client-side stylesheet has no work to do. This will become clear when we look at the implementation of the various features of the user interaction. A typical XML file fragment is shown below:

```
<article id="changes" title="Changes in this Release">
   <h1>Version 9.4 (2011-12-09)</h1>

   <p>Details of changes in Saxon 9.4 are detailed on the following pages:</p>

   <nav>
      <ul/>
   </nav>

   <section id="bytecode-94" title="Bytecode generation">
      <h1>Bytecode generation</h1>
```

```
        <p>Saxon-EE 9.4 selectively compiles stylesheets and queries into Java 
...
```

For the most part, the content of the site is authored directly in the form in which it is held on the site, using an XML editor. The work carried out at publishing time consists largely of validation. There are a couple of exceptions to this: the Javadoc content is generated by a tool from the Java source code, and we also generate an HTML copy of the site as a fallback for use from devices that are not Javascript-enabled. There appears to be little call for this, however: the client-side Saxon-CE version of the site appears to give acceptable results to the vast majority of users, over a wide range of devices. Authoring the site in its final delivered format greatly simplifies the process of making quick corrections when errors are found, something we have generally not attempted to do in the past, when republishing the site was a major undertaking.

The User Interface

In this section we discuss the user interface of the documentation application. The rendition of the webpages is done dynamically, almost entirely in XSLT 2.0. There are a few instances were we rely on helper functions (amounting to about 50 lines) of JavaScript. The XSLT is in 8 modules totalling around 2500 lines of code. The Javascript code is mainly concerned with scrolling a page to a selected position, which in turn is used mainly in support of the search function, discussed in more detail below.

The URI and Fragment Identifier

URIs follow the "hashbang" convention: a page might appear in the browser as:

- *http://www.saxonica.com/documentation/index.html#!configuration*

For some background on the hashbang convention, and an analysis of its benefits and drawbacks, see Jeni Tennison's article at [5]. From our point of view, the main characteristics are:

- Navigation within the site (that is, between pages of the Saxon documentation) doesn't require going back to the server on every click.
- Each sub-page of the site has a distinct URI that can be used externally; for example it can be bookmarked, it can be copied from the browser address bar into an email message, and so on. When a URI containing such a fragment identifier is loaded into the browser address bar, the containing HTML page is loaded, Saxon-CE is activated, and the stylesheet logic then ensures that the requested sub-page is displayed.
- It becomes possible to search within the site, without installing specialized software on the server.
- The hashbang convention is understood by search engines, allowing the content of a sub-page to be indexed and reflected in search results as if it were an ordinary static HTML page.

The XSLT stylesheet supports use of hashbang URIs in two main ways: when a URI is entered in the address bar, the stylesheet navigates to the selected sub-page; and when a sub-page is selected in any other way (for example by following a link or performing a search), the relevant hashbang URI is constructed and displayed in the address bar.

The fragment identifiers used for the Saxon documentation are hierarchic; an example is

- *#!schema-processing/validation-api/schema-jaxp*

The first component is the name of the chapter, and corresponds to the name of one of the XML files on the server, in this case `schema-processing.xml`. The subsequent components are the values of `id` attributes of nested XHTML 5 `<section>` elements within that XML file. Parsing the URI and finding the relevant subsection is therefore a simple task for the stylesheet.

The Table of Contents

The table of contents shown in the left-hand column of the browser screen is constructed automatically, and the currently displayed section is automatically expanded and contracted to show its subsections. Clicking on an entry in the table of contents causes the relevant content to appear in the right-hand section of the displayed page, and also causes the subsections of that section (if any) to appear in the table of contents. Further side-effects are that the URI displayed in the address bar changes, and the list of breadcrumbs is updated.

Some of this logic can be seen in the following template rule:

```
<xsl:template match="*" mode="handle-itemclick">
        <xsl:variable name="ids"
                      select="(., ancestor::li)/@id"
                      as="xs:string*"/>
        <xsl:variable name="new-hash"
                      select="string-join($ids, '/')"/>
        <xsl:variable name="isSpan"
                      select="@class eq 'item'"
                      as="xs:boolean"/>
        <xsl:for-each select="if ($isSpan) then .. else .">
            <xsl:choose>
                <xsl:when test="@class eq 'open' and not($isSpan)">
                    <ixsl:set-attribute name="class" select="'closed'"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:sequence select="js:disableScroll()"/>
                    <xsl:choose>
                        <xsl:when test="f:get-hash() eq $new-hash">
                            <xsl:variable name="new-class"
                                          select="f:get-open-class(@class)"/>
                            <ixsl:set-attribute name="class"
                                                select="$new-class"/>
                            <xsl:if test="empty(ul)">
                                <xsl:call-template name="process-hashchange"/>
                            </xsl:if>
                        </xsl:when>
                        <xsl:otherwise>
                            <xsl:sequence select="f:set-hash($new-hash)"/>
                        </xsl:otherwise>
                    </xsl:choose>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:for-each>
    </xsl:template>
```

Most of this code is standard XSLT 2.0. A feature particular to Saxon-CE is the `ixsl:set-attribute` instruction, which modifies the value of an attribute in the HTML DOM. To preserve the functional nature of the XSLT language, this works in the same way as the XQuery Update Facility: changes are written to a pending update list, and updates on this list are applied to the HTML DOM at the end of a transformation phase. Each transformation phase therefore remains, to a degree, side-effect free. Like the `xsl:result-document` instruction, however, `ixsl:set-attribute` delivers no result and is executed only for its external effects; it therefore needs some special attention by the optimizer. In this example, which is not untypical, the instruction is used to change the `class`attribute of an element in the HTML DOM, which has the effect of changing its appearance on the screen.

The code invokes a function `f:set-hash` which looks like this:

```
<xsl:function name="f:set-hash">
   <xsl:param name="hash"/>
   <ixsl:set-property name="location.hash" select="concat('!',$hash)"/>
</xsl:function>
```

This has the side-effect of changing the contents of the `location.hash` property of the browser window, that is, the fragment identifier of the displayed URI. Changing this property also causes the browser to automatically update the browsing history, which means that the back and forward buttons in the browser do the right thing without any special effort by the application.

The Search Box

The search box provides a simple facility to search the entire documentation for keywords. Linguistically it is crude (there is no intelligent stemming or searching for synonyms or related terms), but nevertheless it can be highly effective. Again this is implemented entirely in client-side XSLT.

The initial event handling for a search request is performed by the following XSLT template rules:

```
<xsl:template match="p[@class eq 'search']" mode="ixsl:onclick">
    <xsl:if test="$usesclick">
        <xsl:call-template name="run-search"/>
     </xsl:if>
</xsl:template>

<xsl:template match="p[@class eq 'search']" mode="ixsl:ontouchend">
    <xsl:call-template name="run-search"/>
</xsl:template>

<xsl:template name="run-search">
    <xsl:variable name="text"
                  select="normalize-space(ixsl:get($navlist/div/input, 'value')
    <xsl:if test="string-length($text) gt 0">
        <xsl:for-each select="$navlist/../div[@class eq 'found']">
            <ixsl:set-attribute name="style:display" select="'block'"/>
        </xsl:for-each>
        <xsl:result-document href="#findstatus" method="replace-content">
            searching...
        </xsl:result-document>
        <ixsl:schedule-action wait="16">
            <xsl:call-template name="check-text"/>
        </ixsl:schedule-action>
     </xsl:if>
</xsl:template>
```

The existence of two template rules, one responding to an `onclick` event, and one to `ontouchend`, is due to differences between browsers and devices; the Javascript event model, which Saxon-CE inherits, does not always abstract away all the details, and this is becoming particularly true as the variety of mobile devices increases.

The use of `ixsl:schedule-action` here is not so much to force a delay, as to cause the search to proceed asynchronously. This ensures that the browser remains responsive to user input while the search is in progress.

The template `check-text`, which is called from this code, performs various actions, one of which is to initiate the actual search. This is done by means of a recursive template, shown below, which returns a list of paths to locations containing the search term:

```
<xsl:template match="section|article" mode="check-text">
    <xsl:param name="search"/>
    <xsl:param name="path" as="xs:string" select="''"/>
    <xsl:variable name="newpath" select="concat($path, '/', @id)"/>
    <xsl:variable name="text" select="lower-case(
        string-join(*[not(local-name() = ('section','article'))],'!'))"/>
    <xsl:sequence select="if (contains($text, $search))
            then substring($newpath,2)
            else ()"/>
    <xsl:apply-templates mode="check-text" select="section|article">
        <xsl:with-param name="search" select="$search"/>
        <xsl:with-param name="path" select="$newpath"/>
     </xsl:apply-templates>
</xsl:template>
```

This list of paths is then used in various ways: the sections containing selected terms are highlighted in the table of contents, and a browsable list of hits is available, allowing the user to scroll through all the hits. Within the page text, search terms are highlighted, and the page scrolls automatically to a position where the hits are visible (this part of the logic is performed with the aid of small Javascript functions).

Breadcrumbs

In a horizontal bar above the table of contents and the current page display, the application displays a list of "breadcrumbs", representing the titles of the chapters/sections in the hierarchy of the current page. (The name derives from the story told by Jerome K. Jerome of how the *Three Men in a Boat* laid a trail of breadcrumbs to avoid getting lost in the Hampton Court maze; the idea is to help the user know how to get back to a known place.)

Maintaining this list is a very simple task for the stylesheet; whenever a new page is displayed, the list can be reconstructed by searching the ancestor sections and displaying their titles. Each entry in the breadcrumb list is a clickable link, which although it is displayed differently from other links, is processed in exactly the same way when a click event occurs.

Javadoc Definitions

As mentioned earlier, the Javadoc content is handled a little differently from the rest of the site.

This section actually accounts for the largest part of the content: some 11Mb, compared with under 2Mb for the narrative text. It is organized on the server as one XML document per Java package; within the package the XML vocabulary reflects the contents of a package in terms of classes, which contains constructors and methods, which in turn contain multiple arguments. The XML vocabulary reflects this logical structure rather than being pre-rendered into HTML. The conversion to HTML is all handled by one of the Saxon-CE stylesheet modules.

Links to Java classes from the narrative part of the documentation are marked up with a special class attribute, for example `<a class="javalink" href="net.sf.saxon.Configuration">Configuration</a>`. A special template rule detects the `onclick` event for such links, and constructs the appropriate hashbang fragment identifier from its knowledge of the content hierarchy; the display of the content then largely uses the same logic as the display of any other page.

Links between Sub-Pages in the Documentation

Within the XML content representing narrative text, links are represented using conventional relative URIs in the form `<a class="bodylink" href="../../extensions11/saxon.message">saxon:message</a>`. This "relative URI" applies, of course, to the hierarchic identifiers used in the hashbang fragment identifier used to identify the subpages within the site, and the click events for these links are therefore handled by the Saxon-CE application.

The Saxon-CE stylesheet contains a built-in link checker. There is a variant of the HTML page used to gain access to the site for use by site administrators; this displays a button which activates a check that all internal links have a defined target. The check runs in about 30 seconds, and displays a list of all dangling references.

The Up/Down buttons

These two buttons allow sequential reading of the narrative text: clicking the down arrow navigates to the next page in sequence, regardless of the hierarchic structure, while the up button navigates to the previous page.

Ignoring complications caused when navigating in the sections of the site that handle functions and Javadoc specifications, the logic for these buttons is:

```
<xsl:template name="navpage">
  <xsl:param name="class" as="xs:string"/>
  <xsl:variable name="ids" select="tokenize(f:get-hash(),'/')"/>
  <xsl:variable name="c" as="node()"
              select="f:get-item($ids, f:get-first-item($ids[1]), 1)"/>
  <xsl:variable name="new-li"
              select="if ($class eq 'arrowUp') then
                         ($c/preceding::li[1] union
                         $c/parent::ul/parent::li)[last()]
                      else ($c/ul/li union $c/following::li)[1]"/>
  <xsl:variable name="push" select="string-join(($new-li/ancestor::li union
                                    $new-li)/@id,'/')"/>
  <xsl:sequence select="f:set-hash($push)"/>
</xsl:template>
```

Here, the first step is to tokenize the path contained in the fragment identifier of the current URL (variable `$ids`). Then the variable `$c` is computed, as the relevant entry in the table of contents, which is structured as a nested hierarchy of `ul` and `li` elements. The variable `$new-li` is set to the previous or following `li` element in the table of contents, depending on which button was pressed, and `$push` is set to a path containing the identifier of this item concatenated with the identifiers of its ancestors. Finally `f:set-hash()` is called to reset the browser URL to select the subpage with this fragment identifier.

Conclusion

In this paper we have shown how Saxon-CE was constructed, with the help of Google's GWT technology, as a cross-browser implementation of XSLT 2.0, performing not just XML-to-HTML rendition, but also supporting the development of richly interactive client-side applications. Looking at the example application shown, there are clear benefits to writing this in XSLT rather than Javascript.

Can this transform the fortunes of XML on the Web? It's hard to say. We are in an industry that is surprisingly influenced by fashion, and that is nowhere more true than among the community of so-called "web developers". The culture of this community is in many ways more akin to the culture of television and film production than the culture of software engineering, and the delivered effect is more important than the technology used to achieve it. The costs related to content creation may in some cases swamp the software development costs, and many of the developers may regard themselves as artists rather than engineers. This is not therefore fertile territory for XML and XSLT with their engineering focus.

Nevertheless, there is a vast amount of XML in existence and more being created all the time, and there are projects putting a great deal of effort into rendering that XML for display in browsers. In many cases, the developers on those projects are already enthusiastic about XSLT (and sometimes very negative about learning to write in Javascript). It is perhaps to this community that we should look for leadership. They won't convice everyone, but a few conspicuous successes will go a long way. And perhaps this will also remind people that there is a vast amount of XML on the web; it's just that most of it never finds its way off the web and into the browser.

Acknowledgement

Many thanks to Michael Kay who initiated the Saxon-CE project and reviewed this paper. Also thanks to Philip Fearon for his contribution in the development of the Saxon-CE project, most notably the design of the Javascript event handling and the automated test harness. He also wrote most of the Saxon-CE documentation viewer described in this paper.

References

[1] *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation. 16 November 1999. James Clark. W3C. http://www.w3.org/TR/xslt.

[2] *XSL Transformations (XSLT) Version 2.0*. W3C Recommendation. 23 January 2007. Michael Kay. W3C. http://www.w3.org/TR/xslt20.

[3] *Google Web Toolkit (GWT)*. Google. http://code.google.com/webtoolkit/.

[4] *The Saxon XSLT and XQuery Processor*. Michael Kay. Saxonica. http://www.saxonica.com/.

[5] *Hash URIs*. Jeni Tennison. http://www.jenitennison.com/blog/node/154.