Multi-user interaction using client-side XSLT

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

Abstract

We describe two use-case applications to illustrate the capabilities of the first XSLT 2.0 processor designed to run within web browsers. The first is a technical documentation application, which permits browsing and searching in a intuitive way. We then present a multi-player chess game application; using the same XSLT 2.0 processor as the first application, it is in fact very different in purpose and design in that we provide multi-user interaction on the GUI and implement communication via a social media network: namely Twitter.

Table of Contents

Introduction ... 1
Browsing and searching technical documentation ... 2
  XML on the Server ... 3
  Implementing the User Interface ... 4
Chess Application ... 8
  Architecture ... 9
  GUI interaction and Twitter Communication ... 10
  Chess game logic ... 13
Acknowledgement ... 15
References ... 15

Introduction

One of the original aims of XSLT was that it should be possible to use the language to convert XML documents to HTML for rendering on the browser. This aim has largely been achieved, but it took a long time before XSLT processors with a sufficient level of conformance and performance were available across all common browsers; and while that was happening, the landscape changed. It changed in several ways:

- XSLT 2.0 came along, raising the level of capability of the language and making the limitations of XSLT 1.0 even more obvious
- XML processing in the browser went out of fashion, in good measure because of the absence of universal XSLT support (and because, for users writing in Javascript, handling JSON was a lot easier)
- Microsoft's near-monopoly on browser installations was broken, with the result that web applications could only adopt new technologies when there was consensus to implement them across all the browsers; this added further to the disinclination of browser vendors to improve the level of XML support
- Web 2.0 came along: the web was no longer about producing pretty renditions of static documents, but about generating interactive applications. So XSLT as originally conceived was only capable of doing half the job. Why would anyone want to use a mix of two languages (XSLT and Javascript) when Javascript could tackle it all?
- Mobile devices became as common a client platform as the traditional desktop, increasing the need for content repurposing to meet different device capabilities

Javascript has become a mature and powerful language, and there are good reasons for its popularity. However, in the areas where XSLT is strong, Javascript is at its weakest. Simple document transformation tasks, like sorting or grouping the rows of a table, or generating a table of contents, are painfully tortuous. So there are good reasons for feeling that the user community has a right to expect something better; developers writing web applications where document manipulation plays a significant role are currently using tools that deliver very poor productivity.

But if XSLT is to be viable in this space, it needs to be able to do far more than simple XML to HTML conversion; it also needs to handle the user interaction, and the other tasks that a modern web application is expected to perform, such as “behind-the-scenes” interaction with the server (still sometimes known by the inappropriate name of AJAX).

Although the rule-based processing model of XSLT was designed primarily with document rendition in mind (it lends itself well to handling documents with unpredictable or variable structure), it turns out that the same model is well suited to handling the other asynchronous and unpredictable tasks that arise in a web application. The way in which XSLT “push-mode” stylesheets are written to handle events from the XML parser is not at all dissimilar to the kind of event-based programming used in many graphical user interface toolkits.

This talk will examine how the first implementation of XSLT 2.0 on the browser, Saxon-CE, addresses this opportunity. This will be done by demonstrating example applications in which it is deployed.

We will look in particular at two applications.

The first is an application for 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 my moving the XSLT processing to the client side and taking advantage of some of the new facilities to handle user interaction.

The second application is a classic Web 2.0 use case, an interactive multi-player game, where the communication between the players takes place over an underlying Twitter feed. Although the presentation of this application will no doubt be entertaining to the conference audience, the purpose is a serious educational one: the design of the application will be studied to show how real benefits are obtained by coding it in a high-level declarative language like XSLT, with a push-based event-driven approach at the heart of its processing model. It also gives an opportunity to examine some of the security issues associated with client-side processing, proxy authentication, cross-site scripting constraints, and the like.

XSLT 2.0 on the browser has been demonstrated at XML Prague in previous years, but only with very simple applications. Since last year's conference, Saxon-CE has become a fully-released product, and its capabilities have considerably increased. In particular, the interaction with the rest of the browser environment has been greatly strengthened, and these demonstrations will illustrate what can be achieved by taking advantage of these new capabilities.

Browsing and searching technical documentation

The first application we examine is an application for browsing technical documentation: specifically, it is used for display the Saxon 9.4 documentation on the Saxonica web site. (It is described as a demonstration, but it has been running successfully for several months, and we intend to cut over to this as the primary means of displaying the documentation at the next release.)

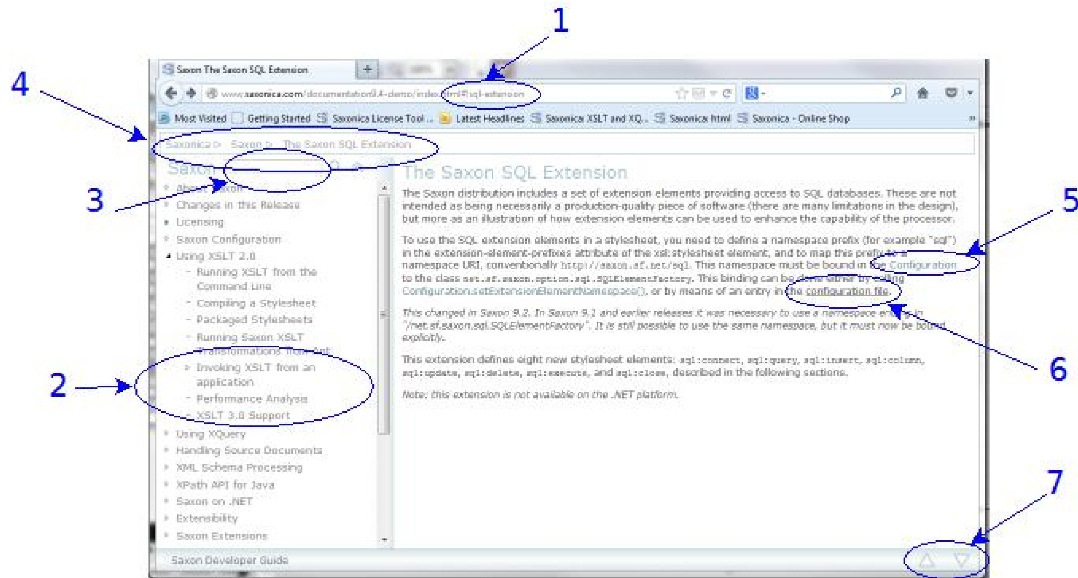
The documentation for Saxon 9.4 can be found at:

- http://www.saxonica.com/documentation9.4-demo/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 (800Kb of Javascript source code generated from around 100K lines of Java), compiling this, and then executing it before it can even start compiling and executing the XSLT code.

The documentation is presented in the form of a single-page web site. The screenshot in Figure 1 shows its appearance.

Figure 1. Technical documentation application in the browser



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 index 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 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 XHTML5. 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 5.0 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.

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.

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.

Implementing the User Interface

This section discusses how the various aspects of the user interface are implemented. The implementation is done almost entirely in XSLT 2.0, with a few 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/documentation9.4-demo/index.html#!configuration*

For some background on the hashbang convention, and an analysis of its benefits and drawbacks, see Jeni Tennison's article at [11]. 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.

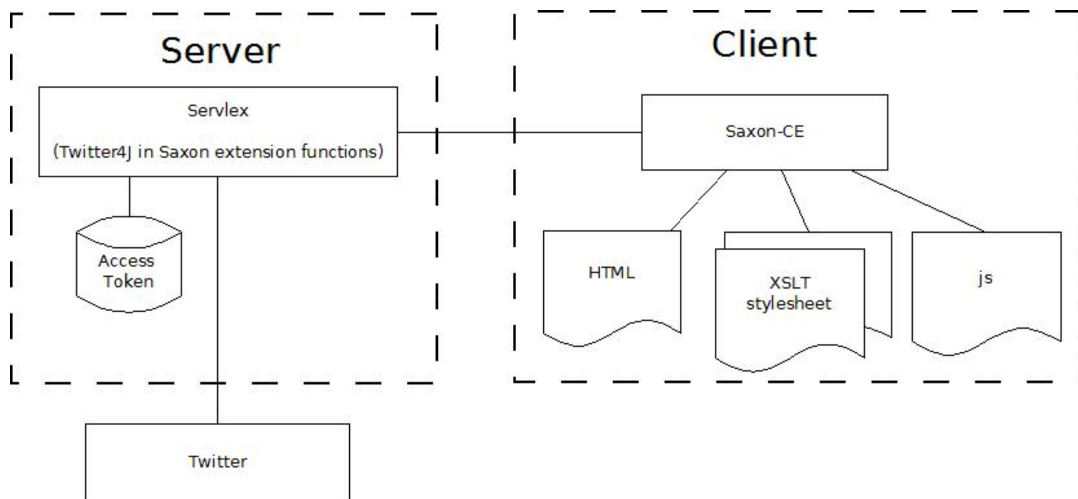
Chess Application

In this section we discuss the second of the two Saxon-CE applications: namely the multi-player chess game, which has been written primarily in XSLT and makes use of a proxy server to communicate via Twitter. There are three interesting and reasonable self-contained parts to the application, which we will examine in turn: the communication mechanism, the interactive user interface, and the chess game logic. But we will start by describing the overall application architecture.

Architecture

The multi-player game over Twitter consists of two components as shown in Figure 2: The server-side component containing tooling to handle Twitter communication and a client-side component containing the Saxon-CE application.

Figure 2. Architecture of Chess Game application



The architecture of the chess game application on Saxon-CE

Client-side Component

The client-side component is core. It consists of the Saxon-CE XSLT 2.0 processor for the browser, a single HTML file, two stylesheets and a Javascript file. The HTML file contains a skeleton 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. The first stylesheet handles the rendering of the chess board, response to user input, and calls to the Javascript to perform Twitter support. The second stylesheet handles the chess game validation and general game play moves. 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 we find it useful to abstract the Twitter http GET requests to a Javascript layer.

Server-side Component

The purpose of the server-side component is to proxy communication with Twitter both to receive and send tweets. The original aim of this project was to develop an application where all functionality including Twitter communication was done client-side. However there are security and configuration concerns with sending tweets directly from the client-side. As discussed by [7], Javascript is a powerful language with good browser support, but the fundamental problem with using Javascript is that the source code is viewable. This means the consumer keys required in the Twitter authentication mechanism (OAuth) are exposed, which would compromise the application; and hence a server-side approach is appropriate.

We use Twitter's OAuth protocol [10] in favour of the Basic Auth facility, which is now deprecated and not supported. The OAuth authentication protocol allows a user to approve an application to act on their behalf without disclosing password credentials; security involves the use of consumer keys held by the application.

The API calls of OAuth are achieved using the Twitter4J API [8]. Twitter4J is an unoffical Java library for the Twitter API, which we integrete in the Servlex webapp [9] as Saxon extension functions called from within XSL. Servlex provides a way to write web applications directly in XSLT, XQuery and/or XProc. The request URIs are mapped to XSLT functions alongside variables which allow passing of data between the server and the client.

It should be emphasized that the server-side components are used only to proxy communication between the client and Twitter. The only data retained on the server is the Twitter authentication credentials. It is therefore not necessary for the two players to use the same server. Indeed, since the protocol for exchanging moves in Twitter messages is very simple, there is no real need for both players to be using the same client-side application. One could even envisage one of the players entering moves (as tweets) directly from the keyboard.

GUI interaction and Twitter Communication

Figure 3. Chess board application in browser

Screen-shot of the chess board application in a browser.

The chess game application in multi-player mode can played over the internet in the browser. The appearance of the graphical user interface (GUI) illustrated in Figure 3. The main components are:

- The display of the current state of the board
- The display of the history of the game
- Input fields and buttons allowing moves to be made (or other actions, such as resigning). The normal way of making a move is by drag-and-drop in the intuitive way. Support for this varies a little bit by platform, and on some devices; users may find it easier to enter moves from the keyboard in traditional chess notation.

When a user has made a move, we rely on the user to wait or watch for the opponent's reply (which might be after a few seconds or after a day or more, depending on the style of play). We don't poll for the move within the browser, or "push" the move from the server to the browser when it arrives. Instead, users have many tools available to alert them to tweets from their opponent, and when a tweet

arrives they can click the "Request Move" button to accept their opponent's reply. Alternatively, if they have closed the browser in the meantime, they can reopen the application by loading the HTML page, and clicking "Restore", which restores the current state of play by reference to the Twitter timeline.

Figure 4 shows a flow diagram of the states and actions in the chess game application. The diagram is split into four four main sections: user interactions; Client which controls the user actions and interfaces with the server-side; Server which interfaces client actions with Twitter using published APIs; and the final section is Twitter itself.

Figure 4. Data flow diagram of the Chess game application

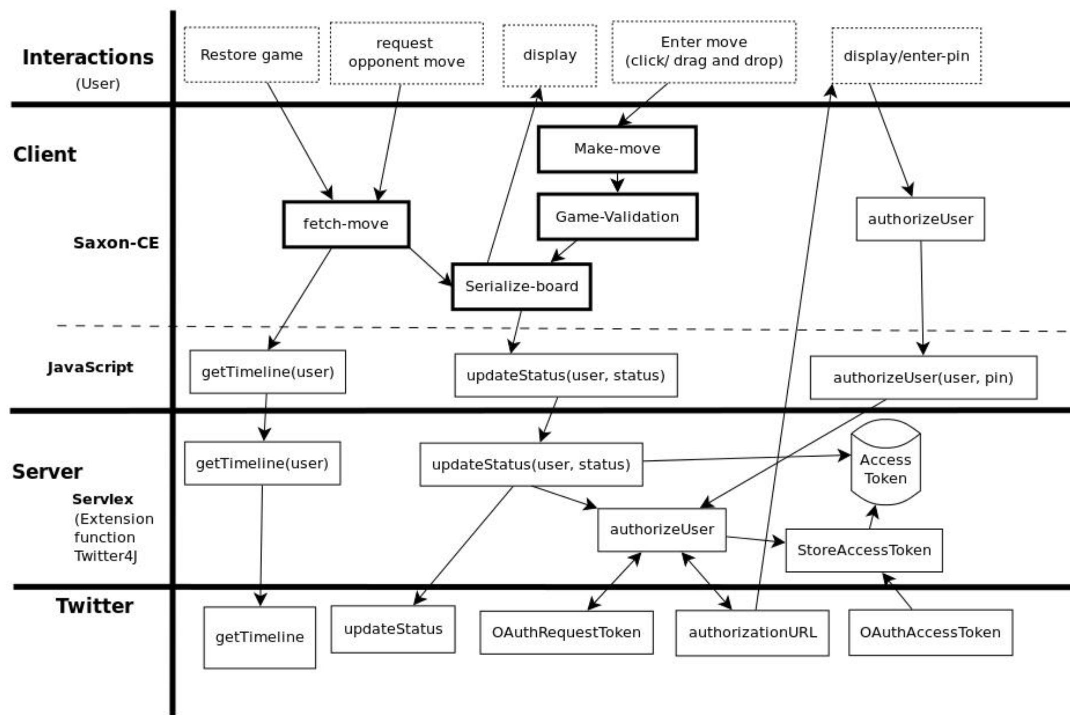


Diagram shows the interaction between user, client, server and Twitter.

The server is required to communicate data between the chess game (on the client) and on Twitter. Servlex works as a dispatching and routing mechanism to components (implemented as XSLT stylesheets), applying a configuration-based mapping between the request URI and the component used to process that URI. The container communicates with the components by means of an XML representation of the HTTP request, and receives in turn an XML representation of the HTTP response to send back to the client. There are three XSLT functions to handle the Twitter operations: update Status, get timeline and authenticate user. These functions are written in Java using the Twitter4J library and made available as Saxon extension functions called from the XSLT.

1. *updateStatus*: At the start of a new game we assume the user is already authenicated. The player attempts a move on their chess board, which activates a Twitter submission. A request is made to servlex with the following URI pattern:

   http://192.168.0.2:8080/servlex/chess/updateStatus?user=johnWhite&status=@maryBlack %20RNBQKBNRPPPPPPPP________________________________p____________pppp_ppprnbqkbnr%20e2-e4%20p:1

   We observe the URI pattern *updateStatus* and the parameter data after *?* is used to route the XSLT function and to supply the values required in the function. There are two parameters: the name of the user (in this case the user making the move), and the message to be sent. The message in this case includes the name of the other player, the current state of the board (represented as a simple string of 64 characters), and the move itself in algebraic chess notation (e2-e4 indicating an advance of the Kings Pawn by two squares). The message also includes the ply number (in chess

terminology, a move consists of a ply by white followed by a ply by black). This we consider as the move number or incremented count in the current session.

When an updateStatus request is processed there are two possible outcomes: success or failure. Success means the player's Twitter timeline status has been updated as a result of successful verification of the user's access token, in which case the client receives a "Ok" response, and the tweet is sent, hopefully reaching the other player.

If the updateStatus request fails it means the player has not been granted authorization to the application. In this case we require the player to authenicate the application via Twitter. We achieve Twitter authentication using the PIN-based OAuth flow. Using Twitter4J we request an access token by requesting, from Twitter, a URL for the user to login and grant authorization. This URL is returned back to the client. Upon receipt, the client application allows the user the option to launch the URL in a new window. We discuss this further in step 2.

2. *authenicateUser*: To authenticate a user to play the chess game the client application asks the player to authenicate the application in Twitter. The client receives a URL from Twitter with a generated PIN number. The user will see a PIN code, with instructions to return to the application and enter this value in a form. The value is then sent back from the client as HTTP GET request in the following pattern:

   http://192.168.0.2:8080/servlex/chess/authenticateUser?user=johnWhite&pin=12345

   This is submited to Twitter via the servlex with consumer and user request tokens. If authorization is successful, an access token is sent in a callback by Twitter to the servlex code, and the server retains the details in persistent storage for future verification.

3. *getTimeline*: This function is used to request an opponent's move or restore a game from two players' Twitter feed. The URL pattern is as follows:

   http://192.168.0.2:8080/servlex/chess/timeline?user=maryBlack

To start play, players are required to enter their own Twitter screen-name and that of the opponent whom they will play. (Without this, the game is still playable in stand-alone mode without Twitter functionality, but we will ignore this option). The user has three main forms of action:

1. *Enter-move*: When it's the player's turn to make a move, this is done by a simple drag and drop or click on the piece you want to move and then click on the square you want to move it to. An event is then triggered and fired by the browser. Saxon-CE handles this event and fires an XSLT template rule for the <div> element representing the target square of the move. This is done in the ChessGame XSLT stylesheet. These template rules do basic checks on the syntax of the move and generate an XML view of the board, which is then passed through with the move data to the game-validation template. The game validation we discuss later, but upon successful validation of the move, the logic board is serialized and updated on the DOM view of the HTML page, which leads to it being visually represented on the screen. If the player attempts an invalid move, there is similar visual feedback, and the board state remains unaltered.

   A player's move is communicated to the opponent via Twitter. This is achieved as follows: we make an HTTP get request via a Javascript method called from the client-side XSLT. On the server the Servlex web application manages the HTTP requests.

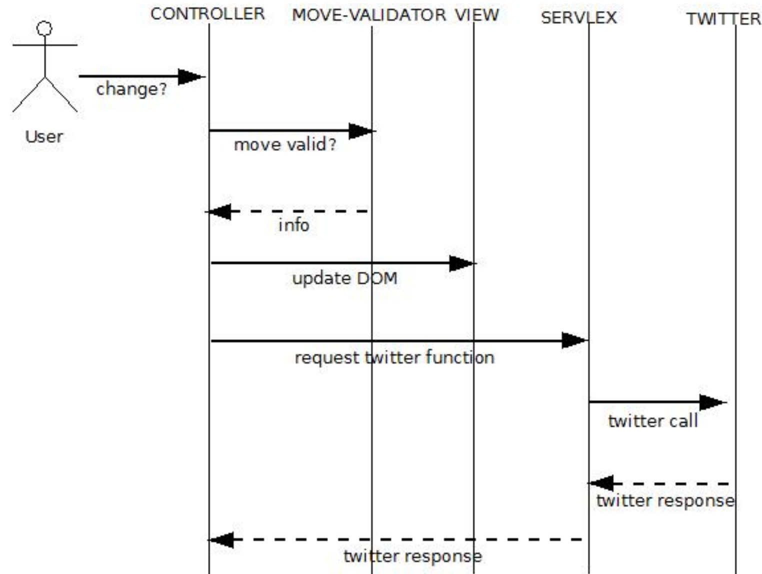
2. *Request opponent move*: When an opponent has made their move, a Twitter message is sent which appears on their Twitter timeline, it mentions the opposing player's screen-name allowing the player to pick up the move when they press request-move.

3. *Restore game*: Sometimes players will interact in real time, but on other occasions they may prefer to play a game slowly, with hours or days elapsing between moves. It is therefore not necessary to keep the browser open between moves. The state of the game can be restored at any time by reference to the Twitter timeline; it does not need to be retained in the client application, and it is not retained in the server part of the application. Each tweet contains a representation of the state of

the board, and if necessary (though we don't do it today) we could reconstruct the full game history by searching back through the time-line.

Figure 5 shows the sequence of operations in the chess game from end-to end.

Figure 5. Sequence Diagram of the Chess game application



Sequence Diagram which shows interaction between a player and various processes in the application.

Chess game logic

The final part of the application is concerned with what programmers often call "business logic", though in this case "game logic" would be more appropriate. The application, of course, does not include any chess strategy (though we could envisage one of the players being replaced with a robot); all it does is to verify that moves are legal, and update the state of the game in response to each move. This requires only enough look-ahead to ensure that a move does not leave a player in check, since such a move is illegal. In fact, our application does not currently have 100% coverage of all the laws of chess; we don't handle all the subtleties of pawn promotion, en-passent captures, or the rules about when castling is and is not permitted. Some of these rules in fact require additional information about the state of the game that cannot be inferred from knowing only the position of the pieces (castling is only allowed, for example, when the king has never moved in the history of the game; and a draw may be claimed if the same position occurs three times in the history of the game or if no pawn move or capture is made for 50 moves).

The XSLT logic of this part of the application is not especially noteworthy, and the same code could be used for a conventional server-side application. Keeping it compact, however, is important, because the size of a stylesheet has a material influence on the perceived responsiveness of Saxon-CE applications. The logic is encapsulated as a set of template rules, one matching each kind of chess piece, taking the state of the board and proposed move as parameters, and returning an indication of whether the move is valid. Here is the relevant rule for testing a knight's move:

```
<xsl:template match="div[@data-piece='knight']"
              mode="is-valid-move"
              as="element(move-test)">
    <xsl:param name="moveFrom" as="xs:integer"/>
    <xsl:param name="moveTo" as="xs:integer"/>
    <xsl:param name="board" as="element(div)+"/>
    <xsl:variable name="destinationAvailable"
                  select="not($board[$moveTo]/@data-colour = @data-colour)"/>
```

```
    <xsl:variable name="rowDistance" as="xs:integer"
                  select="f:row($moveTo) - f:row($moveFrom)"/>
    <xsl:variable name="columnDistance" as="xs:integer"
                  select="f:column($moveTo) - f:column($moveFrom)"/>
    <xsl:variable name="is-valid" as="xs:boolean"
                  select="$destinationAvailable and abs($rowDistance) *
                          abs($columnDistance) = 2"/>
    <move-test is-valid="{if ($is-valid) then 'yes' else 'no'}"/>
</xsl:template>
```

It relies on the simple principle that a knight's move is valid if and only if the product of the number of rows moved and the number of columns moved is 2. It omits the general rule that the target square must be either vacant or occupied by the opponent, because that rule is true for all pieces and can therefore be factored out.

As a first approximation, we can think of this template simply returning a boolean to indicate whether the move is valid or not. In practice, we also want to say why it's invalid (for example, because the target square is occupied), and for complex moves like castling or en-passent captures we also want to return sufficient information for the calling code to actually make the requested move by applying changes to the state of the board. So instead of a simple boolean, we return a constructed XML element containing this information.

We are primarily using template rules here as a polymorphic despatch mechanism, so that different rules apply to each kind of chess piece. The polymorphic templates are invoked using an <xsl:apply-templates> call contained in the logic of a stylesheet function, which returns the success/failure result to the caller, like this:

```
<xsl:function name="f:isValidMove" as="element(piece-move)">
        <xsl:param name="piece" as="element(div)"/>
        <xsl:param name="moveFrom" as="xs:integer"/>
        <xsl:param name="moveTo" as="xs:integer"/>
        <xsl:param name="board" as="element(div)*"/>

        <piece-move>
            <xsl:choose>
                <xsl:when test="$moveFrom = $moveTo">
                    <xsl:attribute name="is-valid" select="'no'"/>
                    <xsl:attribute name="description" select="'not moved'"/>
                </xsl:when>
                <xsl:when test="$board[$moveFrom][self::empty]">
                    <xsl:attribute name="is-valid" select="'no'"/>
                    <xsl:attribute name="description"
                                   select="'no piece at start position'"/>
                </xsl:when>
                <xsl:when
                    test="not($board[$moveTo][div/@data-piece eq 'empty' or
                          div/@data-colour != $piece/@data-colour])">
                    <xsl:attribute name="is-valid" select="'no'"/>
                    <xsl:attribute name="description"
                        select="'target sqare is occupied by your own colour'"/>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:variable name="move-test" as="element(move-test)">
                        <xsl:apply-templates select="$piece" mode="is-valid-move">
                            <xsl:with-param name="moveFrom" select="$moveFrom"/>
                            <xsl:with-param name="moveTo" select="$moveTo"/>
                            <xsl:with-param name="board" select="$board"/>
                        </xsl:apply-templates>
```

```
                    </xsl:variable>
                    <xsl:copy-of select="$move-test/@*"/>
                </xsl:otherwise>
            </xsl:choose>
        </piece-move>
</xsl:function>
```

For some of the more complex moves, making the move involves more than simply vacating the square where the piece started and occupying the square where it ends. Castling causes two pieces to move; en-passent capture removes a piece that is on neither the starting square nor the ending square; pawn promotion leaves a different kind of piece on the target square (and also involves user input, because the laws allow a pawn to be promoted to something other than the usual queen).

Good design practice suggests a model-view-controller architecture in which the model (the state of the board) is represented by a data structure independent of the view (the visualization of the board), where the controller ensures that the model and the view remain in step with each other. In fact our code holds the model implicitly as part of the HTML page (the view), which in some ways is a useful short-cut, but also reduces flexibility. For example, it makes the logic for deciding whether a player is in check more complicated, because special measures are needed to make trial moves without them being visible on the screen.

The entire logic for verifying and applying chess moves is 400 lines of XSLT coding.

Acknowledgement

Many thanks to Philip Fearon for his contribution in the development of the Saxon-CE project. In particular, he wrote the browser based technical documentation and helped to develop the Chess game application, both driven by Saxon-CE.

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] *CXAN: a case-study for Servlex, an XML web framework*. Florent Georges. XML Prague. March, 2011. Prague, Czech Republic. . http://archive.xmlprague.cz/2011/files/xmlprague-2011-proceedings.pdf.

[6] *Twitter*. Twitter . https://twitter.com/.

[7] *How-to: Secure OAuth in JavaScript*. Derek Gathright. 21 October 2010. Yahoo. http://derek.io/blog/2010/how-to-secure-oauth-in-javascript/.

[8] *Twitter4J*. http://twitter4j.org.

[9] *Servlex*. Florent Georges. http://code.google.com/p/servlex/.

[10] *Twitter OAuth*. Twitter. https://dev.twitter.com/docs/auth/oauth.

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