Distributing XSLT Processing between Client and Server

O'Neil Delpratt

Saxonica

<oneil@saxonica.com>

Debbie Lockett

Saxonica

<debbie@saxonica.com>

Abstract

In this paper we present work on improving an existing in-house License Tool application. The current tool is a server-side web application, using XForms in the front end. The tool generates licenses for the Saxon commercial products using server-side XSLT processing. Our main focus is to move parts of the tool's architecture client-side, by using "interactive" XSLT 3.0 with Saxon-JS. A beneficial outcome of this redesign is that we have produced a truly XML end-to-end application.

Keywords: XSLT, Client, Server

1. Introduction

For a long time now browsers have only supported XSLT 1.0, whereas on the server-side there are a number of implementations for XSLT 2.0 and 3.0 available. For applications using XSLT processing, the client/server distribution of this processsing is governed by the implementations available in these environments. As a result, many applications, including our in-house "License Tool" web application, rely heavily on server-side processing for XSLT 2.0/3.0 components.

The current License Tool web application is built using the Servlex framework [1] [2], and principally consists of a number of XSLT stylesheets. The HTML front end uses XForms, and the form submission creates HTTP requests which are handled by Servlex. We use XSLTForms [3] to handle the form processing in the browser, an implementation of XForms in XSLT 1.0 and JavaScript.

The main motivation for this project is to improve our License Tool webapp by moving parts of the server-side XSLT processing into the client-side. This can only be made possible by a client-side implementation of XSLT 2.0/3.0. We would like to see which components of the application's architecture can now be done using client-side interactive XSLT.

Interactive XSLT is a set of extension elements, functions and modes, to allow rich interactive client-side applications to be written directly in XSLT, without the need to write any JavaScript. (For information on the beginnings of interactive XSLT, see [4], and for the current list of `ixsl` extensions available see [5].) Stylesheets can contain event handling rules to respond to user input (such as clicking on buttons, filling in form fields, or hovering the mouse), where the result may be to read additional data and modify the content of the HTML page. The suggested idea for building such interactive XSLT applications is to use one skeleton HTML page, and dynamically generate page content using XSLT. Event handling template rules are those which match on user interaction events for elements in the HTML DOM. For instance, the template rule

```
<xsl:template match="button[id='submit']"
              mode="ixsl:onclick"/>
```

handles a click event on a specific HTML button element. When the corresponding event occurs, this causes a new transformation to take place, with this as the initial template, and the match element (in the HTML DOM) as the initial context item. The content of the template defines the action. For example, a fragment of HTML can be generated and inserted into a specific target element in the HTML page using a call such as

```
<xsl:result-document select="div[id='target']"
                     mode="ixsl:replace-content"/>
```

doi:10.14337/XMLLondon17.Lockett01

In order to use client-side interactive XSLT 3.0 within our License Tool, we use Saxon-JS [6] - a run-time XSLT 3.0 processor written in pure JavaScript, which runs in the browser and implements interactive XSLT. We still maintain some server-side XSLT processing, as required. But by using XSLT 3.0 [7], with the interactive extensions, we are able to do much more of the tool's processing client-side, which means that we can achieve our objective. The redesign means that the tool is now XML end-to-end, without any environment specific glue, which minimises the need to translate between objects.

One benefit of moving the processing client-side is that more of it is brought directly under our control, so we should then be in a better position to resolve and in places avoid incompatibilities between the technologies and environments. For instance, in the current tool, we are aware of some data encoding issues for non-ASCII characters [8]. In the current License Tool the data is sent by XSLTForms encoded in a certain format, but this encoding is not what Servlex expects. The problem is made more complicated by the multiple layers of technologies in use, and of course the internal XSLTForms and Servlex processing is out of our control.

The reluctance of browser vendors to upgrade XSLT support has meant that tools such as XSLTForms are stuck with using XSLT 1.0. Many mobile browsers do not even support XSLT 1.0. By using Saxon-JS in the browser, we are freed from this restriction, and so can replace our use of XSLTForms in the License Tool. We have worked towards a new implementation of XForms using XSLT 3.0 and interactive XSLT, and produced a prototype partial implementation.

Along with making improvements to the tool, we were also interested to see how the experience gained from this real world example may initiate further developments for Saxon-JS itself. In particular, one major challenge is how to handle the communications between client and server using HTTP, within our interactive XSLT framework.

In the following sections we will introduce what the application actually does, how it originally worked, and the changes we have made. We will focus on how we have used XSLT 3.0 and interactive XSLT in the redesign, the benefits of this change, and how it has improved the application.

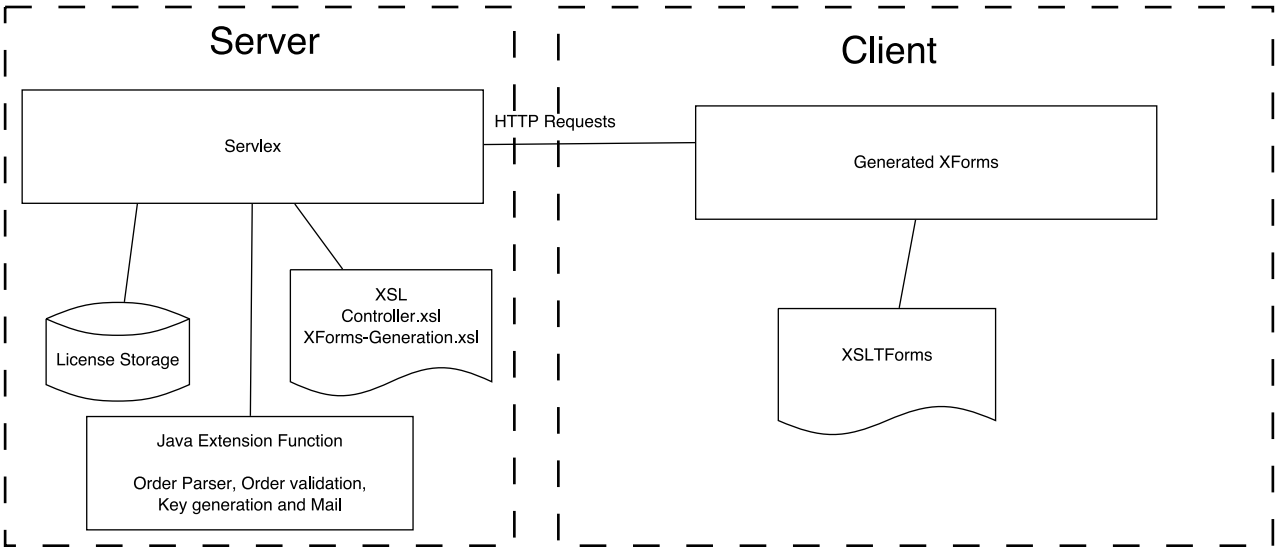
2. License Tool application: what it does, and how it currently works

The License Tool processes license orders (i.e. purchase or registration infomation) for the Saxon commercial products, and then generates and issues license files (which contain an encrypted key used to authorize the commercial features of the product) to the customer. The License Tool also maintains a history of orders and licenses (as a set of XML log files) and provides simple reporting functions based on this history.

The application is built using the Servlex framework. Servlex is a container for EXPath Web Applications [9], using the EXPath Packaging System [10], whose functionality comes from XML technologies. Servlex handles all the networking and server-side XML processing, and connects the XML components to the HTTP layer.

For our current tool, this means using Servlex on the server-side to parse the license order and convert to a custom XML format, process the XML instance data received from the XForms form, and send feedback to the user in the browser. This is all driven by XSLT stylesheets on the server. On the client-side, we use XSLTForms to handle the XForms processing. An overview of the architecture of the License Tool is shown in [Figure 1]. This architecture diagram shows the main components, and technologies used, client and server side.

Figure 1. Old License Tool architecture diagram



In more detail, the tool works as follows:

1. When purchasing or registering for a license, a customer completes a web form to provide certain order information: contact details, the name of the purchased product, etc.
2. This license order information is sent to us by email, as structured text (it would be nice if it were XML or JSON, but this is the real world). See [Figure 2] and [Figure 3] for examples.
3. We input the license order text from the email into the License Tool via an XForms form in the "Main Form" HTML page of the webapp, and use the form submit to send this data to the server.
4. All communication with the server-side of the License Tool is done using HTTP requests, which are picked up within the Servlex webapp by an XSLT controller stylesheet which then processes the license order. The first steps are to parse the text and convert it into a custom XML format, which is then validated. See [Figure 4] for an example of the order XML.
5. The application then returns the license order to the user as the XML instance data of another XForms form, the "Edit Form". At this point the order may be manually edited. (This page can also be used to edit existing licenses before reissuing, for instance for upgrades and renewals.)
6. Next, when the "Edit Form" is submitted, the customer's license file is created. This processing is done using reflexive extension functions written in Java within the XSLT on the server.
7. The application then reports to its user the outcome of generating the license, for final confirmation. If there has been a problem with the license generation, there is again the option to manually modify the license order. Otherwise, when the user confims the order, the license is issued (again using a server-side Java extension function).
8. The data model of the application is XML driven, with the exception of the email text for a license order used as the initial input.

Figure 2. Example license order text for an evaluation license.

```
First Name: Tom

Last Name: Bloggs

Company: Bloggs XML

Country: United Kingdom

Email Address: tom@bloggs.com

Phone:

Agree to Terms: checked
```

Figure 3. Example license order text for a purchased license.

```
Order #9999 has just been placed

Email: tom@bloggs.com

Comments: ZZZ-9999

==== Items ====

item_name: Saxon-EE (Enterprise Edition),
           initial license (ref: EE001)
item_ID: EE001
item_options:
item_quantity: 1
item_price: £360.00

item_name: Saxon-EE (Enterprise Edition),
           additional licenses (ref: EE002)
item_ID: EE002
item_options:
item_quantity: 2
item_price: £180.00


==== Order Totals ====

Items: £720.00
Shipping: £0.00
Tax: £0.00
TOTAL: £720.00

-- Billing address --

company: Bloggs XML

billing_name: Tom Bloggs
billing_street: 123 Fake St
billing_city: Somewhere
billing_state: Nowhere
billing_postalCode: A1 1XY
billing_countryName: United Kingdom
billing_phone:
```

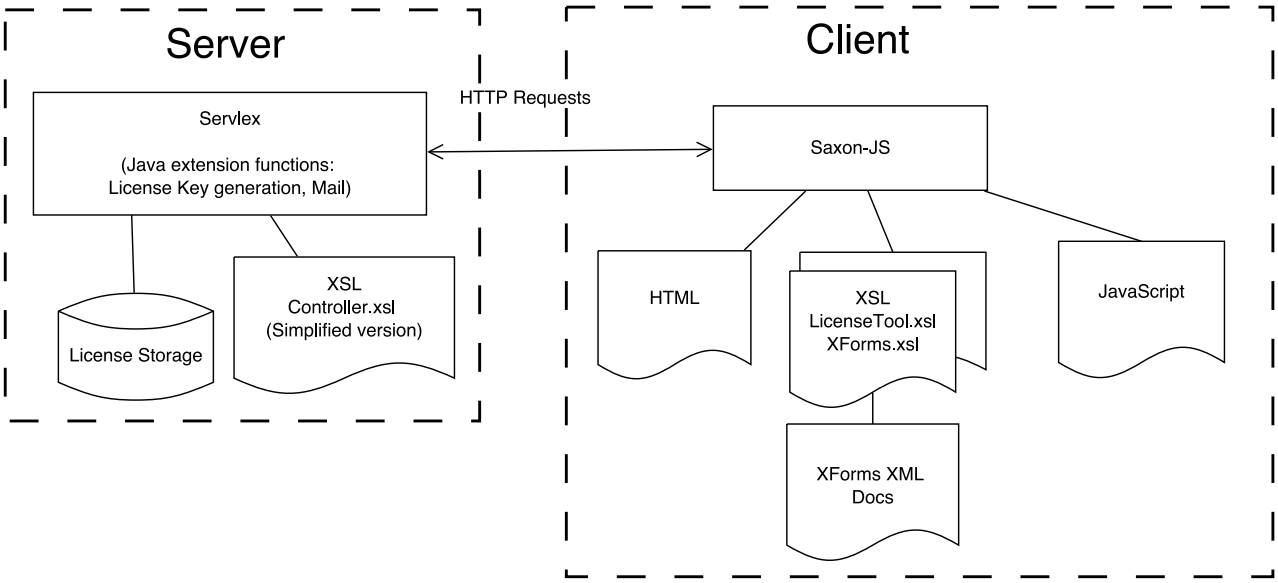
Figure 4. Example of an order in XML format (the result of converting the example license order text in [Figure 3])

```
<Order>
   <OrderRef>#9999</OrderRef>
   <DatePlaced>2017-05-05</DatePlaced>
   <DateOfExpiry>never</DateOfExpiry>
   <First>Tom</First>
   <Last>Bloggs</Last>
   <Company>Bloggs XML</Company>
   <Address1>123 Fake St</Address1>
   <Address2/>
   <Town>Somewhere</Town>
   <County>Nowhere</County>
   <Postcode>A1 1XY</Postcode>
   <Country>United Kingdom</Country>
   <Email>tom@bloggs.com</Email>
   <Phone/>
   <UpgradeDays>366</UpgradeDays>
   <MaintenanceDays>366</MaintenanceDays>
   <Online>true</Online>
   <OrderPart>
      <ProductCode>EE001</ProductCode>
      <Edition>EE</Edition>
      <Platform>J</Platform>
      <Features>TQV</Features>
      <Quantity>1</Quantity>
      <Value>360</Value>
      <Domain/>
   </OrderPart>
   <OrderPart>
      <ProductCode>EE002</ProductCode>
      <Edition>EE</Edition>
      <Platform>J</Platform>
      <Features>TQV</Features>
      <Quantity>2</Quantity>
      <Value>360</Value>
      <Domain/>
   </OrderPart>
</Order>
```

3. Application redesign

The main aim of this project is to move more components of the License Tool's processing architecture client-side, by using interactive XSLT 3.0. We have achieved this by building a new interactive front end for our tool, written in interactive XSLT 3.0. This stylesheet is compiled using Saxon-EE to produce a stylesheet export file (SEF) which the Saxon-JS run-time executes in the browser.

Figure 5. New License Tool architecture diagram



This redesign to the application means that the client-side processing can now handle the initial parsing of the license order text, and convert to the order XML format; before the need for any server-side processing. We have also produced a new partial prototype implementation for XForms using interactive XSLT 3.0, as an improvement to using XSLTForms, which is included in the front end process. An overview of how the main processing components and technologies are now distributed, client and server side, is shown in the architecture diagram for the new License Tool in [Figure 5].

The redesign has also introduced some changes to the tool's processing pipeline. The flow diagram in [Figure 6] illustrates the design for the new tool. It shows the steps of the process - user interactions with the application, the processing actions client and server side - and the flow between all of these steps. As can be seen, we have indeed moved much of the processing client-side: parsing license order text; converting to XML; validating; generating and rendering the XForms "Edit Form"; and handling the submit button click event. We currently still rely on server-side processing for some final stage components of the pipeline - namely generating the license (which includes the encrypted key), issuing it via email, and storing the license order.

In the following sections, we will describe in more detail the three main areas of development in the License Tool's redesign:

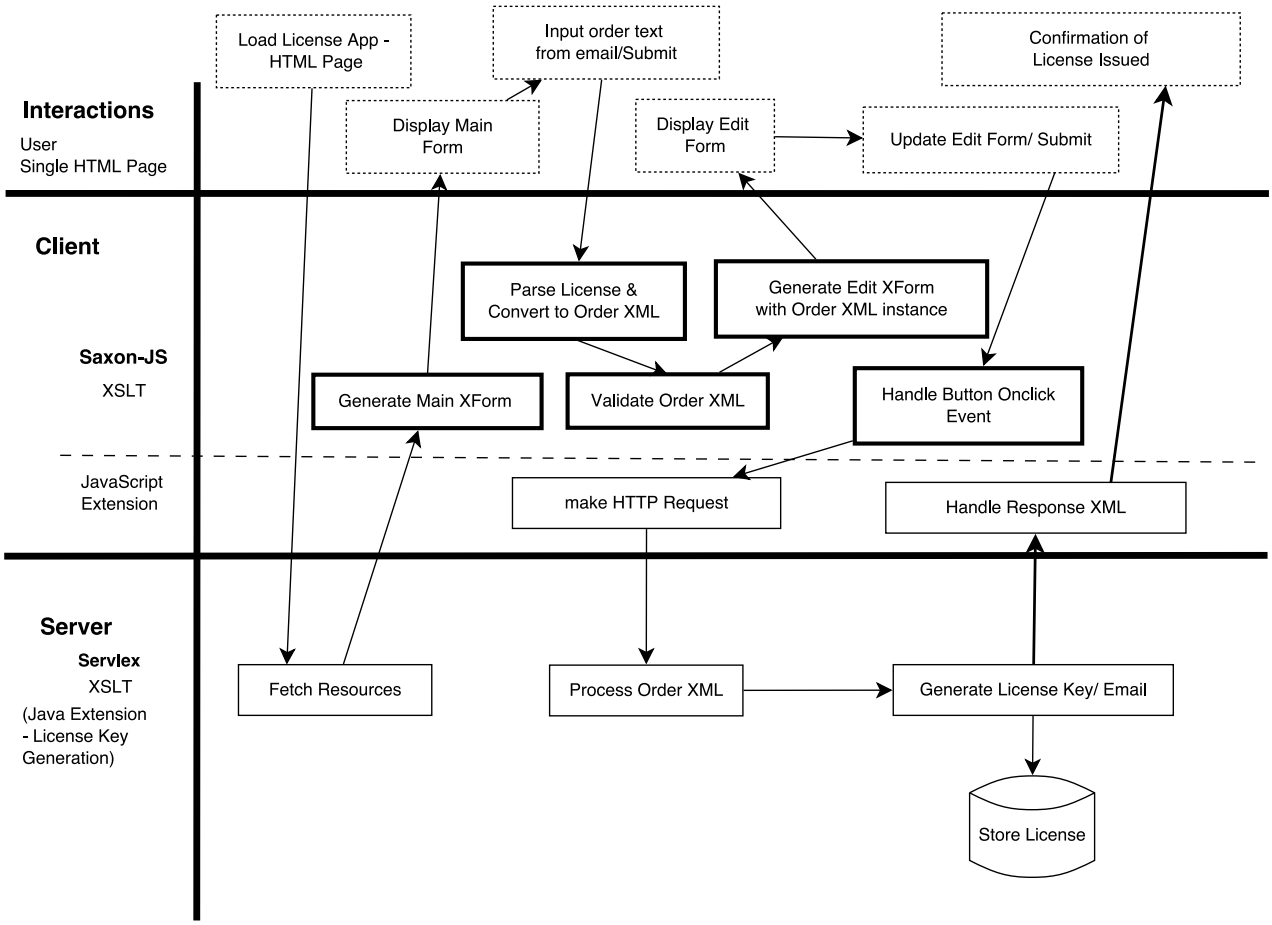
1. Client-side XSLT processing, to replace the use of Java extension functions.
2. A prototype for a new XForms implementation using XSLT 3.0 and interactive XSLT, to run client-side.
3. Handling HTTP communication between client and server.

4. Client-side XSLT processing

There is great potential to simplify the tool's architecture by using XML end-to-end. Throughout the pipeline, the main object we are dealing with is an "order" - which contains information about the customer, the products ordered, the date of the order, etc. Ideally we would be handling this order in our custom XML format throughout the processing pipeline. So the order information is captured directly into XML format, which can be modified, passed between client and server, and stored server-side, without the need to be converted to any other different formats along the way.

The original legacy version of the License Tool was a Java application. The current tool makes much use of Java extension functions in the webapp's XSLT stylesheets in order to reuse the original code. For instance, the process of parsing the license order text and converting it into XML format is done by a Java extension function. So there is a Java class with methods to parse the input license order text, and produce a Java `Order` object. This `Order` object is then converted into XML using Java methods. In fact, many such Java components of the webapp could be rewritten in XSLT,

Figure 6. New License Tool flow diagram



and this is clearly more straightforward, since we then avoid converting between XML and Java objects and back. Making the change to do such processing directly in XSLT could have been done already within the current server-side webapp, but there was perhaps little incentive - "if it ain't broke, don't fix it". However, now we are looking to bring the processing client-side, it does make sense to write XSLT solutions to replace the Java code. As well as being able to move such components to the client-side, the tool's architecture is generally simplified by replacing the Java code.

So, in the new tool, the first step of parsing the license order text, and converting to the custom order XML format, is done directly in XSLT. In fact, rather than parsing the structured text and creating the order XML directly, as an intermediate stage it is convenient to use a representation of the order as an XPath 3.0 map item. Having split the input license order text by line, if a line looks like an order category/value pair, then it is added to the order map as a key/value pair. This order map is then used to add text content to an order XML skeleton. For example,

```
<First>
  <xsl:value-of select="$orderMap?first"/>
</First>
```

There may be other stages where it is more convenient to use the XPath map representation for a license order, rather than the XML format. It actually makes a lot of sense to handle the order as an XPath map item, which is easier to modify, and use XSLT functions to convert these to the custom XML format and back. However currently we generally stick to the order XML format.

There may still be times when we need to serialize to a string, and reparse to XML. Infact also, in the new tool, at certain stages we convert to JSON and back, as well as to XPath map and back. But at least these can all now be handled directly within XSLT 3.0, and there is no need to use other objects outside of the XDM model.

5. XForms implementation in interactive XSLT 3.0

XSLTForms is based on XSLT 1.0 to compile XForms to (X)HTML and JavaScript in the browser. As previously discussed, support for XSLT in browsers is limited to XSLT 1.0, and vendors are not inclined to move this forward. Saxon-JS now provides us with XSLT 3.0 processing in the browser, and so we can write a new XForms implementation using XSLT 3.0 and interactive XSLT, to replace the use of XSLTForms. We now describe our proof-of-concept XForms implementation exploring the possibilities in (interactive) XSLT 3.0.

The XForms model, instance data and form controls which provide the user interactions are written as XML in accordance with the XForms specification. The XForms processor is entirely written using interactive XSLT 3.0 using the XSLTForms implementation as a starting point. In the main entry template rule we use `xsl:params` to supply the XForms form (as an XML document), and optionally corresponding XML instance data, to the stylesheet. The main process of the implementation stylesheet is to convert the XForms form controls elements into equivalent (X)HTML form controls elements (inputs, drop-down lists, textareas, etc.). At the same time the forms controls are populated with any bound data from the XML instance data.

For the XForms controls we specify the binding references to the XML instance data as an XPath expression. For example

```
<xforms:input incremental="true"
              ref="Shipment/Order/DatePlaced" />
```

In the template rule which matches the `xforms:input` control we get the string value from the `ref` attribute, and use this XPath in two ways. Firstly, we call the XSLT 3.0 instruction `<xsl:evaluate>` to dynamically evaluate the XPath expression, which obtains the relevant data value from the XML instance data [10b]. This will be used to populate the HTML form `input` element which we convert to. Secondly, the XPath from the `ref` attribute is copied into the `id` attribute of the `input` element, to preserve the binding to the XML instance data (so that upon form submission, we will be able to copy the changes made within the HTML form back into the instance data, as described later). The result in this example is the following:

```
<input type="text" id="Shipment/Order/First/text()"
       value="Tom">
```

The conversion, and binding preservation, of other XForms controls elements are achieved in a similar way. The full code example is shown in [Figure 7].

Figure 7. Example template from the new XForms implementation, for converting an `xforms:input` element

```
<xsl:template match="xforms:input">
  <xsl:param name="instance1" as="node()?"
             select="()"/>
  <xsl:param name="bindings"
             as="map(xs:string, xs:QName)"
             select="map{}"/>
  <xsl:variable name="in-node" as="node()?">
    <xsl:evaluate xpath="@ref"
      context-item="$instance1/*:document"/>
  </xsl:variable>

  <input>
    <xsl:choose>
      <xsl:when test="
        map:get($bindings, generate-id($in-node)) =
                          xs:QName('xs:date')">
        <xsl:attribute name="type"
                       select="'date'"/>
      </xsl:when>
      <xsl:otherwise>
        <xsl:attribute name="type"
                       select="'text'"/>
      </xsl:otherwise>
    </xsl:choose>
    <xsl:attribute name="id" select="@ref"/>
    <xsl:attribute name="value">
      <xsl:if test="exists($instance1) and
                    exists(@ref)">
        <xsl:evaluate xpath="@ref"
          context-item="$instance1/*:document"/>
      </xsl:if>
    </xsl:attribute>
  </input>
</xsl:template>
```

As well as the conversion from XForms form elements to HTML form elements, the second main purpose of the XForms implementation stylesheet is to handle form interaction. We now describe the process of capturing HTML form data, updating the data in the XForms instance data, and making a form submit.

In order to ensure that the complete XML instance data structure is sent when a user submits a form, we

hold the XML instance data as a JSON object on the page. We use JSON rather than XML for two reasons:

1. Browser inconsistencies and complications in embedding XML islands within HTML, see section 2.4 of [11].
2. The XSLT representation of maps and arrays allows for efficient modifications of immutable data structures (typically `map:put()` does not require the whole tree to be physically copied; the Saxon implementation uses an immutable trie structure to achieve this). It is much harder to achieve efficient small local changes to an XML tree, because an XML tree has node identity and parent pointers which a JSON data structure typically doesn't. Making a small change to an XML document typically involves copying the whole tree. (See [12].)

The purpose of holding the instance data is to ensure that its structure is maintained, so that when a user submits a form, the complete XML representation of the instance data is sent. When the form is submitted (e.g by HTTP post request), the JSON instance data is converted back to its XML format, and then updated with any changes that have been made in the HTML form, before being sent. (Note that it is not necessarily possible to rebuild the XML instance data from the HTML form from scratch - for example, using the XPath paths in the `id` attributes - since the HTML form may of course not be a direct mapping to the instance data. So we need to hold the instance data structure somewhere in the page.) This is achieved by a number of steps.

Firstly, the JSON object is created using the XPath 3.1 function `xml-to-json()` and added to the page in a script element. The code below shows how this is done, to add to the script element with `id="{$xforms-instance-id}"` on the HTML page. Since there is no direct mapping of the XML instance data format to JSON we first have to convert the instance data to an intermediate form - i.e. the XML representation of JSON which is accepted by the `xml-to-json()` function [13] - using our stylesheet function `convert-xml-to-intermediate()`.

```
<xsl:result-document href="{$xforms-instance-id}"
                     method="ixsl:replace-content">
  <xsl:value-of select="xml-to-json(
      local:convert-xml-to-intermediate(
        $instance-doc
      ))"/>
</xsl:result-document>
```

Secondly, the submission process is implemented using an interactive XSLT event handling template. The submit control element has been converted to an HTML button which includes generated `data-*` attributes which match the XForms submission specific attributes, such as `action`. The click event is handled by an event handling template for a button with a `data-action` attribute. At the current stage of development of the license tool, we actually override this event handling template with one which is specific to our tool (as will be described in Section 6).

Within the event handling template rule we convert the instance data held as JSON back to the XML format (going via the intermediate XML format using the XPath 3.1 function `json-to-xml()`), and from this build new updated XML instance data. As we build the new XML instance data we update with any new data from the form, by using the `id` attributes with the XPath paths. This is achieved by using an XSLT apply-templates (with `mode="form-check"`) on the XML instance data. The matching template rules keep track of the path to the matched node within the XML instance data. The template which matches text nodes then uses its path to look within the HTML form for an element whose `id` attribute value is this path. If such an element is found, and a change has been made to the form data, then the new XML instance data is updated correspondingly. See below for the full template:

```
<xsl:template match="text()" mode="form-check">
  <xsl:param name="curPath" select="''"/>
  <xsl:variable name="updatedPath"
    select="concat($curPath,
                   local-name(parent::node()),
                   '/text()')"/>

  <xsl:variable name="control">
    <xsl:apply-templates
      select="ixsl:page()//*[@id=$updatedPath]"
      mode="get-control"/>
  </xsl:variable>

  <xsl:choose>
    <xsl:when test="$control=.">
      <xsl:copy-of select="."/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:copy-of select="$control"/>
    </xsl:otherwise>
  </xsl:choose>

</xsl:template>
```

Some basic validation is included in our partial XForms implementation (for instance, at the loading of the page,

and the submission of the form). There is much more that needs to be implemented for full validation.

As an exploration exercise, we have certainly shown the capabilities of an XForms implementation in interactive XSLT 3.0.

6. HTTP client-server communication

Within the License Tool application, communications between client and server are made using HTTP methods. As well as when the application is initially opened in a browser, the other main point of communication (in both the old and new versions of the tool), is when a user clicks the "submit order" button in the "Edit Form" page. Clicking the button sends the form data via HTTP to the server-side Servlex webapp, and a response is then returned in the browser. However, the details of how this works in the old and new versions of the tool is quite different.

1. In the old version of the License Tool, the form is an XForms form in an (X)HTML page. This (X)HTML page is itself the response from a previous HTTP request. The (X)HTML page calls the XSLTForms implementation for XForms (which uses XSLT 1.0 and JavaScript in the browser) to render the form, and handle form interaction - e.g. making changes to the form entries, and the submit click. As defined in the XForms form controls (using an `xforms:submission` element), clicking the submit button sends the form data (held in the `xforms:instance`, in the custom XML order format) to the URI specified (in the `action` attribute) using an HTTP post request. We use the attribute `method="xml-urlencoding-post"`, an XSLTForms-specific `method` attribute option, which means that the HTTP post request contains the form data XML serialized as a string in a parameter called "postdata".

   The server-side webapp receives this request, and the relevant XSLT component picks up the value of the postdata parameter, parses it back into XML, and processes it. If the processing is successful (i.e. the order is allowed and a license is issued), then the HTTP response sent back is a newly generated HTML page (in fact, a new "Main Form" page) which contains some "success" paragraph saying that the license has been sent.
2. In the redesigned tool, the form is again an XForms form in an (X)HTML page. This time, the content of this (X)HTML page has been generated using an interactive XSLT stylesheet processed by Saxon-JS in JavaScript in the browser. The stylesheet includes interactive XSLT to dynamically insert generated fragments of HTML into the page. The XForms form is one of these generated (X)HTML fragments. The form was generated in XSLT using the prototype implementation of XForms discussed in the previous section (this implementation is an XSLT 3.0 stylesheet using interactive XSLT, which is imported into the main client-side XSLT stylesheet).

   Again, clicking the submit button sends the form data using an HTTP post request. However, this time the click event is handled by Saxon-JS. The interactive XSLT stylesheet contains an event handling template rule, which is called on click events for the submit button. The template's action is to call a user-defined JavaScript function, using the instruction

```
<xsl:sequence select="js:makeHTTPrequest(
                        serialize($orderXML))"/>
```

   This JavaScript function creates an asynchronous HTTP request (using an `XMLHttpRequest` object), with the desired URI destination, and with the data (serialized order XML) sent as content of the request, in plain text type (rather than as a parameter which would force URL encoding). (It may seem preferable to send the order XML in the request directly using the content type "application/xml". However, more work is required to find the best way to do this in JavaScript.)

   The redesigned server-side webapp receives this request, picks up the body of the request, parses it back into XML, and processes it. If the processing is successful (i.e. the order is allowed and a license is issued), then the HTTP response sent back is a piece of XML which contains some "success" data. Having used an asynchronous HTTP request, it is not possible to return the response XML directly from the `makeHTTPrequest` JavaScript function to the XSLT stylesheet; but we may produce some output in the HTML page in another way. As defined in the `makeHTTPrequest` JavaScript function, when the response XML is received by the client, it is supplied to a new call on `SaxonJS.transform` as the source XML, using a different initial template. This named template generates a fragment of HTML, containing a "success" paragraph, which is inserted into the original HTML page.

On the surface, it may not be apparent that our new version is actually an improvement. It certainly didn't seem any less complicated to explain; previously we were just using XForms, but now we're using XSLT and

JavaScript as well as XForms. One main benefit is that we now have much more freedom to define the HTTP request ourselves. Previously, because we were using XSLTForms, we were very constrained by having to send the data with a post request using the "postdata" parameter. Using this method, the content is restricted to being URL encoded, and we have not controlled the encoding used. This is one place where potentially our encoding issue arises. Parameter values are URL encoded in the browser before being sent, and we may not be dealing with this correctly on the server-side. We have now eliminated the issue at this point by controlling the HTTP request, and specifically the content (and its type), ourselves.

Actually, our new solution is only a step towards what we would really like to do, so this is one reason why it is still quite complicated. As discussed in the next paragraph, we would like to produce a solution without the need to use a locally defined JavaScript function, by providing this functionality in interactive XSLT implemented in Saxon-JS. Such a solution which only uses XForms and interactive XSLT would clearly be simpler.

Rather than using a locally defined JavaScript function to create the HTTP request (as we have done currently), it would be nice to implement this functionality directly in Saxon-JS. For instance, we could implement the HTTP Client Module [14], which provides a specification for the `http:send-request()` function. This function allows the details of the request to be specified using a custom XML format: the `<http:request>` element defined in the specification. However, the function is defined to return the content of the HTTP response. In order to return the HTTP response, we would need to use a synchronous request; but it is considered better practice and preferable to use asynchronous requests. So we would rather be able to define an extension which takes as input the request information, as well as information specifying what to do when the response is returned. Compare this proposal to the existing `ixsl:schedule-action` instruction, which makes an asynchronous call to a named template, either after waiting a specified time, or after fetching a specified document. We could add a new version which makes the call to the named template once a response from a specified HTTP request has returned. We could use the HTTP Client Module XML format for defining an HTTP request using a `http:request` element, though it may be more natural (and convenient) to use an XPath 3.0 map. The details of how best to do this are still being developed; but working on this License Tool project has been very useful as an exercise to get started, learn about the relevant technologies, and begin getting ideas to work towards a solution.

7. Conclusion

In this paper we have presented a redesign of our License Tool Web application which utilises interactive XSLT 3.0 to allow more of the processing to be done client-side, with minimum server-side processing. The interactive XSLT extensions broaden the benefits of using XSLT in the browser. To use these technologies we use the XSLT run-time engine Saxon-JS. This processor also provides the capability to call global JavaScript functions, which makes it possible to define HTTP requests and handle the responses in the browser.

The main interface of the License Tool is XForms driven. We have implemented a new XForms prototype implementation using interactive XSLT 3.0 for use in the browser. This proof-of-concept shows that it would be possible to implement the full XForms specification using interactive XSLT 3.0. We are no longer reliant on the XSLTForms implementation, which was limiting because it is an XSLT 1.0 implementation. Unfortunately even XSLT 1.0 is not well supported by all browsers - in particular many mobile browsers simply not do implement XSLT. We get around this by using the Saxon-JS XSLT processor that runs within a browser's JavaScript engine.

Can we eradicate the use of XSLT or other processing on the server-side? Possibly not as we still use Servlex to do some XSLT processing on the server. And would it be desirable? No, because for such an application it is paramount to maintain the security of sensitive data and keep data centralised. But we have certainly achieved our aim of improving our tool, so that it now processes and moves around XML data from end-to-end, and does this processing mostly on the client-side, having moved most of the processsing from the server-side environment. Removing the need for translations between so many different third-party tools and languages outside of the XDM model minimises possible failures and incompatibilies, such as encoding issues, which can only be good in the long run.

With the increase of XML data on the web, and continual demands for speed improvements, the option of using client-server distributed XSLT processing is surely attractive, though of course there may be trade-offs. While it may not become a phenomena, certainly we have showcased the innovative possibilities.

Bibliography

[1] *Servlex*. Florent Georges.
http://servlex.net

[2] *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#page=49

[3] *XSLTForms*. Alain Couthures.
http://www.agencexml.com/xsltform

[4] *Interactive XSLT in the browser*. O'Neil Delpratt and Michael Kay. Balisage. 2013.
doi:10.4242/BalisageVol10.Delpratt01

[5] *Interactive XSLT extensions specification*. Saxonica.
http://www.saxonica.com/saxon-js/documentation/index.html#!ixsl-extension

[6] *Saxon-JS: XSLT 3.0 in the Browser*. Debbie Lockett and Michael Kay. Balisage. 2016.
doi:10.4242/BalisageVol17.Lockett01

[7] *XSL Transformations (XSLT) Version 3.0*. Michael Kay. W3C. 7 February 2017.
https://www.w3.org/TR/xslt-30

[8] *Experiences with XSLTForms and Servlex*. O'Neil Delpratt. 8 March 2013.
http://dev.saxonica.com/blog/oneil/2013/03/experiences-with-client-side-xsltforms-and-server-side-servlex.html

[9] *Web Applications*. EXPath Candidate Module. Florent Georges. W3C. 1 April 2013.
http://expath.org/spec/webapp

[10] *Packaging System*. EXPath Candidate Module. Florent Georges. W3C. 9 May 2012.
http://expath.org/spec/pkg

[10b] *XPath 3.1 in the Browser*. John Lumley, Debbie Lockett and Michael Kay. XML Prague. February, 2017. Prague, Czech Republic.
http://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf#page=13

[11] *HTML/XML Task Force Report*. W3C Working Group Note. Norman Walsh. W3C. 9 February 2012.
https://www.w3.org/TR/html-xml-tf-report/

[12] *Transforming JSON using XSLT 3.0*. Michael Kay. XML Prague. February, 2016. Prague, Czech Republic.
http://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf#page=179

[13] *XML Representation of JSON*. XSL Transformations (XSLT) Version 3.0, W3C Recommendation. Michael Kay. W3C. 7 February 2017.
https://www.w3.org/TR/xslt-30/#json-to-xml-mapping

[14] *HTTP Client Module*. EXPath Candidate Module. Florent Georges. EXPath. 9 January 2010.
http://expath.org/spec/http-client