Implementing XForms using interactive XSLT 3.0

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

Debbie Lockett
Saxonica
<debbie@saxonica.com>

Abstract

In this paper, we discuss our experiences in developing Saxon-Forms, a new partial XForms implementation for browsers using "interactive" XSLT 3.0, and suggest some benefits of this implementation over others. Firstly we describe the mechanics of the implementation - how XForms features such as actions are implemented using the interactive XSLT extensions available with Saxon-JS, to update form data in the (X)HTML page, and handle user input using event handling templates. Secondly we discuss how Saxon-Forms can be used, namely by integrating it into the client-side XSLT of a web application, and examples of the advantages of this architecture. As a motivation and use case we use Saxon-Forms in our in-house license tool application.

Keywords: XML, XSLT, XPath, XForms, Saxon, Saxon-JS

1. Introduction

1.1. Use-case: License Tool application

The motivation for developing Saxon-Forms was a specific use case - namely a project to improve our in-house license tool application (a form-based application for managing and generating licenses). The application used XForms [1] in the browser (using XSLTForms [2]) in the front-end, with server-side XSLT (and Java) processing in the back-end. The project was motivated first, by business needs to improve functionality in an in-house application that has slowly become unmaintainable, and secondly, by the fact that we wanted to improve the capability of Saxon-JS [3] to handle real-world applications with both front-end and back-end processing. We felt that using the technology for an in-house application would be the best way to discover what product enhancements were needed.

The license tool architecture redesign is discussed in detail in [4], where the focus is on the redistribution of XSLT processing, by using Saxon-JS in the browser for client-side XSLT. In this paper, our focus is another part of the project: the use of XForms. Rather than using existing implementations of XForms which run in the browser (such as XSLTForms), alongside the client-side of the application which is written in interactive XSLT [5] [6] and runs in Saxon-JS, we set out to work towards a new implementation of XForms 1.1 which would also run in Saxon-JS. This would allow us to better integrate the use of XForms into the client-side application, as well as being a further exercise in (and demonstration of) using interactive XSLT and Saxon-JS.

The screenshot in figure [fig.1] shows the edit page form of new our license tool application, rendered by Saxon-Forms.

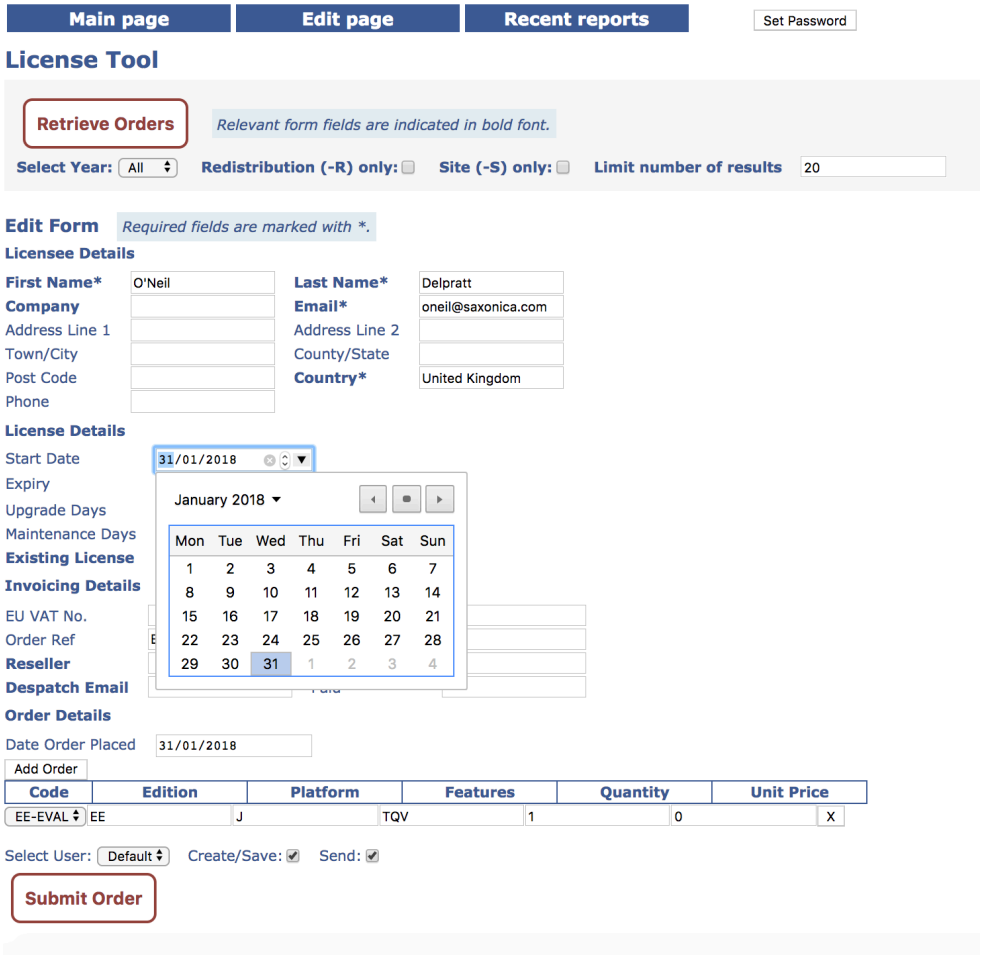

Figure 1. The edit page of the license tool application

1.2. XForms

Forms are a common feature of interactive web applications, allowing users to enter data for submission. HTML forms can be generated in many ways: some

sites serve up form pages from servers using languages such as PHP, JSP, ASP, etc. where the form submission and validation is handled on the server or via Ajax techniques. One of the greatest shortcomings of HTML forms is that the combination of presentation and the content is cumbersome and chaotic to manage. XForms was designed as a direct replacement for HTML forms to address these problems and to do much more. In XForms the presentation and content are separate, and more complicated forms can be authored using form model and controller logic.

Using XForms, a form consists of a section that describes what the form does, called the XForms *model* (contained in an `xforms:model` element, where the `xforms` prefix is used for elements in the XForms `http://www.w3.org/2002/xforms` namespace), and another section that describes how the form is to be presented. The model contains the *instance* (in an `xforms:instance` element) holding the underlying data of the form in an XML structure, *model item properties* describing declarative validation information for constraining values (in `xforms:bind` elements), and details for form data submission (in `xforms:submission` elements). The presentational part of a form contains XForms form controls (such as `input`, `select`, and `textarea` elements) which act as a direct point of user interaction, and can provide read/write access to the instance data. Typically form controls will bind to instance data nodes (as defined by an XPath expression in the `ref` attribute). Note that instance data is only presented to the user when such a binding to a form control exists; and individual form controls are only included in the user interface if the instance data node is relevant (as defined using a `relevant` attribute on an `xforms:bind` element). Actions defining event responses are specified on form controls using action elements, such as `xforms:action` and `xforms:setvalue`.

For a form-based application such as the license tool, XForms is the right choice. As described, it allows for data processing and validation in the form, and of course we want to use XML technologies and maintain our data in XML!

We decided to write a new implementation of XForms to use in our license tool, rather than using existing implementations which run in the browser, because we could see the potential for better integration of XForms into a web application which uses Saxon-JS technologies. As well as being able to use new XSLT 3.0 features, the use of Saxon-JS technologies for our new XForms implementation provides the opportunity to do more at the boundary between the XForms form and the containing application. For example in our license tool, the application logic allows parsing of structured input pasted into a text field. That's beyond the capability of XForms itself, but it can be done in XSLT, and can be integrated into what is predominantly a form-based application. So it's not just XForms; it's XForms integrated into declarative client-side applications.

1.3. XSLT 3.0 and interactive XSLT in the browser with Saxon-JS

Saxon-JS is an XSLT run-time which executes an SEF (stylesheet export file), the compiled form of an XSLT stylesheet generated using Saxon-EE. The Saxon-Forms XSLT stylesheet module is designed to be imported into the client-side XSLT stylesheet of a web application, which is exported to SEF for use with Saxon-JS. Details of how the use of XForms (via Saxon-Forms) can be integrated into the application stylesheet will be covered later. In this section, we briefly highlight the features of Saxon-JS which make it a good fit for implementing XForms:

1. XSLT 3.0 [7] (including XPath maps and dynamic evaluation)
2. Interactive XSLT - for browser event handling
3. Using global JavaScript variables and functions

In Saxon-Forms, we use a number of new XSLT 3.0 features, such as XPath maps and arrays (e.g. for actions and bindings), and dynamic evaluation of XPath with `xsl:evaluate` [8] (e.g. for XForms binding references for form controls). For further details see Section 2. Another feature of Saxon-JS which is crucial to Saxon-Forms is interactive XSLT, used to implement the dynamic interactive functionality of XForms. The interactive XSLT extensions available with Saxon-JS allow rich interactive web applications to be written directly in XSLT. Event handling templates can respond to user input; and trigger template rules to modify the content of the HTML page.

Furthermore, using the `ixsl:schedule-action` instruction with the `http-request` attribute, HTTP requests can be made, and the responses handled. See the submission example in Section 3.2 for further information on how this can be used in the integration of XForms in an application.

A few parts of the XForms implementation are done using JavaScript rather than XSLT. Using Saxon-JS, global JavaScript variables and functions are accessible within the XSLT stylesheet as functions in the `http://saxonica.com/ns/globalJS` namespace, or using the `ixsl:call()` function. Script elements can be inserted into the HTML page using interactive XSLT, providing global JavaScript functions to be used later. Global JavaScript variables are very useful as mutable objects, for example we use a JavaScript variable to hold the XForms instance as a node, this can then be easily accessed and changed to process the form interaction.

2. XForms implementation

The main work of our XForms implementation can be split into two parts, that we will refer to as *initialization* and *interaction handling*. Initialization consists of transforming the presentational part of an XForms form, to render this using HTML to

correctly display the form in a browser; as well as setting up various structures which hold the details of the form (the model item properties, etc.), to be used internally by the implementation. Interaction handling involves acting on user interaction with form controls, to update the XForms instance and form display accordingly, and handling user submission which means submitting the instance to a server. In Section 2.1 we describe how we implement these two areas, using interactive XSLT 3.0. In the early stages of development, we referred to the XSLTForms implementation (which is based on XSLT 1.0 to compile XForms to (X)HTML and JavaScript) for ideas on how to get started, but using XSLT 3.0 and interactive XSLT provides many new ways of doing things and so our implementation is really written from scratch.

Following this, we briefly discuss the XForms coverage of the Saxon-Forms implementation, to give an idea of how much of the XForms specification is implemented.

2.1. Overview of how Saxon-Forms works

Initialization

XForms is designed to be integrated into other markup languages, e.g. (X)HTML. For use with Saxon-Forms, a form is supplied as an XML document, containing the XForms model and presentational part. This XForms form document is supplied via the main entry template rule of the stylesheet, named "xformsjs-main", as a template parameter. Further template parameters can be used to also supply XML instance data, and details of where the form is to appear in the HTML page (by giving the `id` of an HTML `div` element into which the rendered form will be inserted).

The result of Saxon-Forms initialization should be that the form is rendered using HTML, and inserted into the HTML page as directed. Behind the scenes, various variables have also been initialized for internal use, and these are held in the JavaScript global space, using a `script` element (with `id="xforms-cache"`) which is inserted into the HTML head. The script also includes corresponding JavaScript set/get functions for these variables. (When such functions are called from the Saxon-Forms XSLT stylesheet, e.g. using `ixsl:call()`, Saxon-JS will convert the XML items supplied as parameters into JavaScript, and convert the results back the other way, as described in [6]. Below we generally just refer to the XML types.) We cache the following variables relating to the current XForms document:

- the XForms document itself, as a node, required if we need to reset the form
- the instance in its initial state, as a node
- the instance, a node which is updated as a user interacts with the form

- *actions map*, a JSON object whose keys are unique identifiers for each action defined in the form, and the corresponding value is an XPath map which holds the details of the actions
- *relevant attributes map*, an XPath map which maps instance nodes to XPath expressions, taken from the `ref` and `relevant` attributes on `xforms:bind` elements, for example:

```
map{"Document/Options/MaintenanceDate": "../MaintenanceDateSelected='true'",
   "Document/Options/UpgradeDate": "../UpgradeDateSelected='true'", ...}
```

- *pending updates list*, an XPath map which keeps a record of updates for instance nodes which are not bound to form controls

Meanwhile, Saxon-Forms converts XForms form controls to equivalent (X)HTML form control elements (inputs, drop-down lists, textareas, etc.), populated with any bound data from the instance, and which are embellished with additional attributes containing references for use internally. For example:

```
<xforms:input incremental="true"
   ref="Document/Shipment/Order/MaintenanceDays">
   <xforms:action ev:event="xforms-value-changed">
      <xforms:setvalue ref="../../../Options/MaintenanceDate"
         value="if(xs:integer(.) &gt; 0) then
            xs:date(../../../Options/StartDate) +
            xs:dayTimeDuration(concat('P',.,'D'))
            else xs:date(../../../Options/StartDate) +
            xs:dayTimeDuration(concat('-','P',abs(xs:integer(.)),'D'))"/>
      <xforms:setvalue
        ref="../../../Options/Updated">true</xforms:setvalue>
   </xforms:action>
</xforms:input>
```

Will be converted to:

```
<input data-element="MaintenanceDays" data-constraint="number(.) ge 0"
   data-action="d26aApDhDa"
   type="text" value="30"
   data-ref="Document/Shipment/Order/MaintenanceDays"/>
```

Here, in the Saxon-Forms template rule which matches the `xforms:input` control we get the string value from the `ref` attribute, which defines the binding to an instance node, and use this XPath expression in two ways. Firstly, we call the XSLT 3.0 `xsl:evaluate` instruction to dynamically evaluate the XPath expression, to obtain the relevant data value from the instance. This will be used to populate the corresponding HTML form `input` element. Secondly, the `ref` attribute XPath expression is copied into a `data-ref` attribute added to the `input` element, to preserve the binding to the instance node. For each group of action elements in an XForms form control we add an entry to the actions map in the "xforms-cache"

`script` element. For this actions map entry, the key will be a unique identifier, and the value is an XPath map containing all the details of the actions (e.g. from the `xforms:setvalue` elements, etc.) In this example, we add an entry to the actions map object with key "d26aApDhDa", and value:

```
map{"@ref": "Document/Shipment/Order/MaintenanceDays",
   "@event": "xforms-value-changed",
   "setvalue": [map{"@value": "if(xs:integer(.) > 0) then ...",
         "ref": "../../../Options/MaintenanceDate"},
      map{"value": "true",
         "ref":"../../../Options/Updated"}]}
```

Then, as in the example above, we use the `data-action` attribute to link the input element to its relevant entry in the actions map. The conversion, and binding preservation, of other XForms form control elements is achieved in a similar way.

Interaction handling

Interactive XSLT event handling templates are used to handle user interactions with the form, such as data input in a form field or the click of a button. The event handling templates correspond to onchange and onclick events. In figure [fig.2] we illustrate the general pipeline of the processes involved when a user interacts with the form. In this example the template rule with `mode="ixsl:onchange"` and `match="input"` is triggered when a user makes a change in an input box. Here the trigger of the template rule can only happen if the input form control has one or more actions associated with it.

Firstly, we fetch the instance XML for the form and update it with any changes made in the form controls which are not already in the instance. Secondly, we use the value in the `data-action` attribute on the input element to get the associated actions from the actions map. Recall that these associated actions are represented in an XPath map. So we use XPath map functions to extract the details for these actions (e.g. details for setvalue, add or delete) which are then executed. For actions which update instance nodes that are bound to form controls we first update the associated form control. Otherwise, for actions which update instance nodes which are not bound to a form control, we add the changes to the pending updates list.

Thirdly, after all actions have been executed we again update the instance XML (applying the updates in the pending updates list, and picking up changes within form controls) to maintain consistency between the data currently held in the form controls and the instance itself. The final stage is to execute the relevant properties tests for instance nodes (as defined in the relevant attributes map), to determine whether the form controls that they bind to should be included in the rendered form. The corresponding HTML form controls are hidden and revealed by setting the display style property (using the `ixsl:set-property` interactive XSLT extension instruction) to "none" or "inline" respectively.

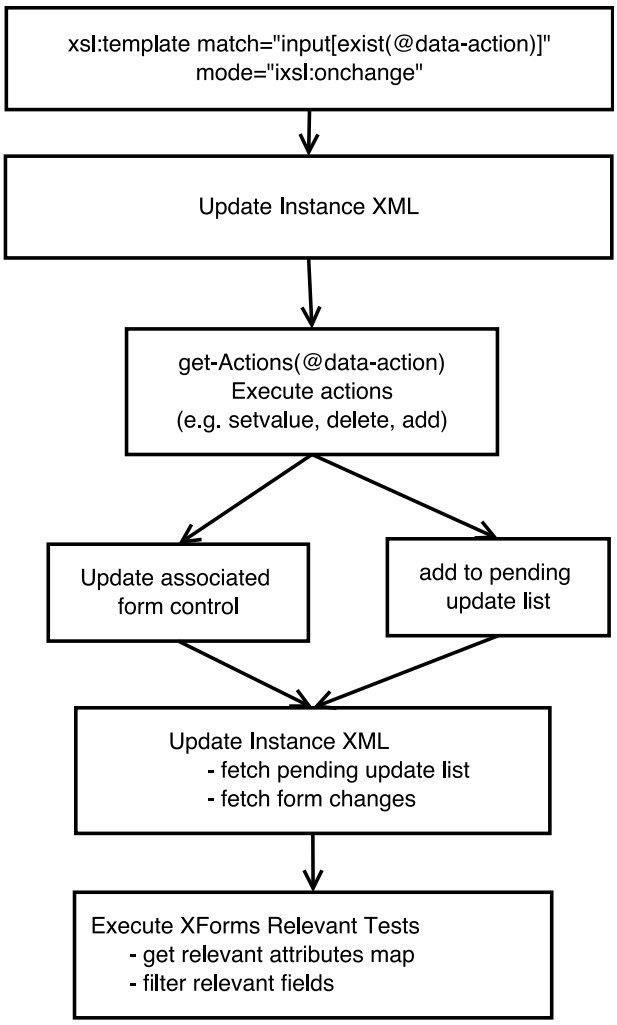

Figure 2. Action handling pipeline diagram

As well as handling changes to form data, the other key user interaction that needs to be handled is submission. However, the XForms submission element is not yet fully implemented in Saxon-Forms. One reason for this is that submission is one of the features where it is desirable, and possible, for more to be done from the application stylesheet, than could be done by a direct implementation of XForms submission. For instance, in our license tool, we use event handling templates (for onclick events on submit buttons) to override the XForms implementation for submission, in order to handle this processing and integrate handling of the server response. Further details follow in Section 3.

2.2. Coverage of the XForms Specification

Saxon-Forms is a partial implementation of the XForms 1.1 specification. The focus was on implementing the parts required to get the license tool application working. But of course it is our intention that the implementation is general

enough for wider use (either used in a standalone way or as a component in an application), and has the potential to be extended for full XForms conformance. Here we summarise the main parts of the XForms specification that are implemented in Saxon-Forms, but note that in all cases (except XPath expressions) there is more which is not implemented:

- *Document structure:* Saxon-Forms currently supports just one model and one instance. In the document structure we represent the `model` element, which consists of the `instance`, `bind` and `submission` elements. This includes the type, required, constraint and relevant model item properties.
- *XPath expressions in XForms:* The specification [1] states "XForms uses XPath to address instance data nodes in binding expressions, to express constraints, and to specify calculations". Saxon-Forms is conformant to the support of XPath since Saxon-JS supports nearly all of XPath 3.1.
- *XForms Function Library:* XForms 1.1 defines a number of functions, of which Saxon-Forms currently only implements `index()` and `avg()`. These are implemented using stylesheet functions, which are then available in the static context for calls on `xsl:evaluate`. Other XForms functions could be implemented in the same way.
- *Core Form Controls:* Saxon-Forms implements the `input`, `textarea` and `select1` form control elements. Of the common support elements (child elements of the form controls), the `label` and `hint` elements are implemented. Of the container form controls (used for combining form controls), only the `repeat` element is implemented.
- *XForms Actions:* Saxon-Forms implements the `action`, `setvalue`, `insert` and `delete` elements.

3. Integrating Saxon-Forms into applications

3.1. Standard integration

Saxon-Forms includes a Saxon-JS stylesheet providing generic XSLT 3.0 code to implement the XForms specification. This can be integrated with application-specific XSLT 3.0 code. Thus, the Saxon-Forms stylesheet module can either be imported into a containing XSLT stylesheet (for the client-side of a web application), or used directly. In either case, to run in Saxon-JS, the stylesheet must first be exported to SEF using Saxon-EE. This can then be run from within an HTML page: as with all Saxon-JS applications, first Saxon-JS is loaded in a script element, and then the SEF can be executed using a JavaScript call to `SaxonJS.transform()`. An XForms document is supplied to Saxon-Forms either as a file or as a document node, along with the optional XForms instance data.

If the Saxon-Forms stylesheet is to be used directly, then the XForms document can be supplied as the source to the transform, as in the example below:

```
<script>
   window.onload = function () {
      SaxonJS.transform({
         "stylesheetLocation": "saxon-xforms.sef.xml",
         "sourceLocation": "sampleBookingForm.xml"
      })
   }
</script>
```

Alternatively, when the Saxon-Forms stylesheet module is imported into the client-side XSLT stylesheet of a web application (e.g. sample-app.xsl), this can be run as follows:

```
<script>
   window.onload = function () {
      SaxonJS.transform({
         "stylesheetLocation": "sample-app.sef.xml",
         "initialTemplate": "main"
      })
   }
</script>
```

And in this case, the XForms document can be supplied at the point that the entry template "xformsjs-main" of Saxon-Forms is called in the sample-app stylesheet:

```
<xsl:template name="call-saxon-forms">
   <xsl:call-template name="xformsjs-main" >
      <xsl:with-param name="xforms-doc" select="doc($bookingForm)"/>
      <xsl:with-param name="xFormsId" select="'xForm'"/>
   </xsl:call-template>
</xsl:template>
```

Here the `xFormsId` parameter gives the `id` of a `div` element in the HTML page where the form is to be inserted; the default is "xForm".

3.2. Integration with application logic

Saxon-Forms is more than just another XForms implementation for the browser, because it allows for form enrichment from application logic in the application stylesheet in which it is integrated. In this section we will present some examples of this:

1. Parsing structured text from a form input textarea, to XML.
2. Overriding submission.
3. Using user defined functions in XPath expressions in the XForm.

Example 1. Parsing input from form textareas

This has proved very useful in our license tool. License orders are often received by email using structured text of a standard form (e.g. for purchases from the online shop, and for evaluation license requests). Because the text is structured, it can be processed using XSLT to extract the data and convert it into XML format. So this parsing can be done in the application stylesheet.

So, a user copies the structured text from an email and inputs it into the textarea of a form in the tool. When the "Parse" button is clicked, this is handled by event handling templates in the application stylesheet which capture the text string and process it to produce some XML output. This XML is then supplied as the instance for another XForms form (in fact, the edit page form, as shown in [fig.1]).

Example 2. Submission

The XForms implementation for submission can be overridden from the application stylesheet, to allow further logic to be added to specify the exact form of the submitted data, and the way a response is handled. For example in the license tool stylesheet, we have event handling templates for onclick events on submit buttons to handle this processing. The updated instance is obtained from the global JavaScript variable (using the procedure in the Saxon-Forms submission implementation), and this is submitted for server side processing using the interactive XSLT mechanism for asynchronous HTTP messages, i.e. using the `ixsl:schedule-action` instruction with the `http-request` attribute. The value of the `http-request` attribute is an XPath map which defines the HTTP request to be made (e.g. specifying method, URI destination, body and media-type). When it returns, the HTTP response is processed by the template specified within the `ixsl:schedule-action` instruction (it has one `xsl:call-template` child); the HTTP response is also represented as an XPath map, and this is provided as the context item to the named template. For instance, this allows feedback from the response to then be returned to the user within the HTML page.

Example 3. User defined functions

Stylesheet functions defined in the application stylesheet can be used in XPath expressions in the XForms document. The only requirement is that the saxon-xforms.xsl stylesheet must include a namespace declaration binding the prefix used in the form to the namespace of the stylesheet function.

For example, the following stylesheet function is defined in our license tool application stylesheet, to obtain product price data from another XML document:

```
<xsl:function name="f:productCodeToPrice" as="xs:integer">
   <xsl:param name="productCode" as="xs:string"/>
   <xsl:variable name="products" select="doc($productsDoc)//Product"/>
```

```
    <xsl:value-of select="xs:integer($products[@code = $productCode]/▶
@price)"/>
</xsl:function>
```

This function can then be used in the XPath expressions in the `value` attribute of a `xforms:setvalue` instruction in the XForm document, to calculate the order part value from the price and quantity (where parts of an order are grouped by product code).

4. Conclusion

In this paper we have presented a new XForms implementation, Saxon-Forms, which makes use of interactive XSLT 3.0 to realize the initialization and processing model of XForms. This project had three goals:

- Firstly, our aim was to explore how XForms and client-side XSLT could coexist to build applications with rich client-side functionality as well as access to server-side functions.
- Secondly, to develop the beginnings of a new XForms implementation taking advantage of the Saxon-JS technology, and able to integrate with Saxon-JS applications.
- Thirdly, to use this technology platform to re-engineer the in-house Saxon license tool application.

Our achievements so far against these goals are:

1. We have demonstrated that a forms-based application can be usefully augmented with additional functionality implemented in XSLT 3.0, for example parsing and validation of complex input fields, and access to reference datasets.
2. We have shown that many of the technical features of the Saxon-JS technology, such as the ability to handle interactive user input using template rules, the ability to issue asynchronous HTTP requests and process the results, and the ability to dynamically evaluate XPath expressions, can be exploited as underpinnings to a client-side XForms implementation.
3. We have rewritten the Saxon license tool application with many new features, with 90% of the code now being in either client-side or server-side XSLT, reducing the Java to a small number of extension functions handling cryptographic signing of licenses.

Further work taking this technology forwards to a fully compliant XForms implementation will depend on user feedback.

5. Acknowledgements

Many thanks to Michael Kay and Alain Couthures for helpful comments for improving this paper, and Saxon-Forms itself.

Bibliography

[1] *XForms 1.1 Specification*. *W3C Recommendation*. 20 October 2009. John Boyer. W3C. https://www.w3.org/TR/xforms11

[2] *XSLTForms*. Alain Couthures. http://www.agencexml.com/xsltforms

[3] *Saxon-JS: XSLT 3.0 in the Browser*. *Balisage: The Markup Conference 2016*. Debbie Lockett and Michael Kay. http://www.balisage.net/Proceedings/vol17/html/Lockett01/BalisageVol17-Lockett01.html

[4] *Distributing XSLT Processing between Client and Server*. O'Neil Delpratt and Debbie Lockett. XML London. June, 2017. London, UK. http://xmllondon.com/2017/xmllondon-2017-proceedings.pdf#page=8

[5] *Interactive XSLT in the browser*. *Balisage: The Markup Conference 2013*. O'Neil Delpratt and Michael Kay. https://www.balisage.net/Proceedings/vol10/html/Delpratt01/BalisageVol10-Delpratt01.html

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

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

[8] *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.