# SAXONJS 3
# CODING IMPROVEMENTS

Balisage 2025

Debbie Lockett
debbie@saxonica.com

# BUGS, SUPPORT AND FEATURES

Users raise different kinds of issues at saxonica.plan.io

- **Bugs**: code fails with exception or incorrect result.
  - Investigate and fix mistakes in the Saxon source code. (Developers are only human after all!)
- **Support**: user needs help to get code to work.
- **Features**: user cannot achieve something, or has suggestions for specific improvements.

# WHEN AN ISSUE IS RAISED...

Start off by asking:

1. Can we reproduce the problem?
2. What IS the problem?

# QUESTIONS

- What is the user trying to achieve?
- How would I solve this?
- Have I encountered this problem before?
- Assess existing solution:
  - Is it intuitive / straight-forward / reasonable / documented ?
  - Is there scope for improvement?

# USER FEEDBACK

A user presents us with an issue they have encountered:

*"Here's a problem I'm trying to solve. This is what I can do. But what about X? How can I do it with SaxonJS?"*

# QUICK INTRODUCTION TO SAXONJS

# WHAT IS SAXONJS?

- XSLT 3.0 run-time processor
- Written in JavaScript, runs in the browser and on Node.js
- Executes compiled XSLT stylesheets (SEFs) generated by Saxon-EE or SaxonJS on Node.js

*If SaxonJS is new to you, but you're interested in learning how to use it, then you might like to start with the tutorial from Declarative Amsterdam 2021.*

# WHAT IS NEW IN SAXONJS 3?

- Declarative Amsterdam 2023: Debbie Lockett presented details about IXSL promises.
- Balisage 2024: Norm Tovey-Walsh presented an overview of new features.
- December 2024: SaxonJS 3 beta released.
- Balisage 2025: Debbie Lockett presents some coding improvements.

# USER ISSUES AND CODE IMPROVEMENTS IN SAXONJS 3

# DEVELOPMENT OF SOME NEW SAXONJS 3 FEATURES

Here we focus on three specific issues raised by users of SaxonJS 2 and the corresponding new features which have been developed for SaxonJS 3 to provide solutions.

# EXAMPLE 1

Combining the results of processing multiple source documents

*"Is there a way I can get a bunch of document nodes via* `ixsl:schedule-action` *(or some other means), process all of them, and use the result of the processing in a single result document?"*

# HOW IS THIS EXAMPLE INTERESTING?

- This is an example of a problem which is possible to solve with SaxonJS 2, but the solution, and how it works in practice, is not entirely intuitive.
- New language features in SaxonJS 3 allow a cleaner solution.

# SAXONJS 2 SOLUTION

- Fetch multiple documents using `ixsl:schedule-action`:

```
<ixsl:schedule-action document="{string($doc-uris})">
  <xsl:call-template name="action"/>
</ixsl:schedule-action>
```

- The "action" template is called once for each document fetch.
- We want to know when ALL documents have been fetched. So check for this, and only do subsequent processing when all documents are available.

# SAXONJS 2 SOLUTION

```
<xsl:template name="action">
  <xsl:variable name="docsAvailable"
    select="$doc-uris ! doc-available(.)" as="xs:boolean*"/>
  <xsl:variable name="docsAllAvailable"
    select="not($docsAvailable = false())" as="xs:boolean"/>
  <xsl:if test="$docsAllAvailable">
    <xsl:result-document href="$output-file">
      <out>
        <xsl:for-each select="$doc-uris">
          <wrapper for="{.}">
            <xsl:sequence select="doc(.)"/>
          </wrapper>
        </xsl:for-each>
      </out>
    </xsl:result-document>
```

# SAXONJS 3 SOLUTION

- Fetch multiple documents using promises:

```
<ixsl:promise
  select="ixsl:all(array{$doc-uris ! ixsl:doc(.)})"
  on-completion="f:go#1"
  on-failure="f:fail#1"/>
```

- The `f:go` function is called when all document fetch promises are complete. The result of the resolved promise is passed as the first argument.
- The `f:fail` function can also be used to easily handle any fetch failures.

# SAXONJS 3 SOLUTION

```
<xsl:function name="f:go" ixsl:updating="true">
  <xsl:param name="docs" as="array(document-node())"/>
  <xsl:result-document href="{$output-file}">
    <out>
      <xsl:for-each select="$doc-uris">
        <xsl:variable name="index" select="position()"/>
        <wrapper for="{$doc-uris[$index]}">
          <xsl:sequence select="$docs($index)"/>
        </wrapper>
      </xsl:for-each>
    </out>
  </xsl:result-document>
</xsl:function>
```

# EXAMPLE 2

Supplying HTTP request headers when fetching documents

*"How can I supply HTTP request headers for a document fetch, which is then accessed using* `doc()` *?"*

# HOW IS THIS EXAMPLE INTERESTING?

- This example demonstrates a small gap in the capabilities of SaxonJS 2.
- In the end, the solution is a relatively simple update for SaxonJS 3.
- But it provides an opportunity to look at the options available for fetching documents, and the process of developing new features.

# SAXONJS 2 DOCUMENT FETCH

Option 1: Use `SaxonJS.transform()` option `documentPool` with documents preloaded using `SaxonJS.getResource()`

- Headers can be set when asynchronously loading documents with `SaxonJS.getResource()`
- Documents in the `documentPool` can then be accessed using `doc()`
- But this mechanism does not help in the case that the document URIs are only known dynamically.

## Option 2: Asynchronously load documents with `ixsl:schedule-action` and `document` attribute

- Fetched documents can then be accessed in subsequent templates using `doc()`
- But with this mechanism, headers can not be set for the fetch.

## Option 3: Use `ixsl:schedule-action` with `http-request` attribute

- HTTP request for document fetch can be fully defined (e.g. supply request headers).
- The HTTP response body can be accessed in the subsequent templates from the XDM map representation of the HTTP response which is passed as the context item.
- Fetched documents can not be accessed in subsequent templates using `doc()`.

# SAXONJS 2 LIMITATION

- It is not possible to combine these methods in SaxonJS 2.
- It is not possible to specify HTTP request headers, dynamically load documents asynchronously and pass them to templates that use `doc()`.
- But it is reasonable to want to use `doc()` for code reuse.

# SAXONJS 3 SUGGESTED SOLUTION

User suggested adding a new IXSL instruction which would add resources to the local cache.

- But implementing IXSL syntax changes is awkward: it involves implementing changes in two compilers as well as the SaxonJS run-time.
- Simpler alternate solutions are preferable!

# SAXONJS 3 ACTUAL SOLUTION

Add option `'pool'` for HTTP requests, to add the response body content to the local cache.

- `'xml'` to add to the `documentPool` so it can be accessed using `doc()` (or `'text'` to add to the `textResourcePool`)
- Adding entries to the XDM maps which represent HTTP requests and responses does not require any changes in the compile time implementations, so this is a much easier fix!

# SAXONJS 3 SOLUTION

```
<xsl:variable name="request" select="
   map{'method': 'GET', 'href': $docURL,
      'header': map{'Accept': 'text/xml'},
      'pool': 'xml' } "/>

<xsl:template name="fetch-document">
   <ixsl:promise select="ixsl:http-request($request)"
      on-completion="f:go#1"/>
</xsl:template>

<xsl:function name="f:go" ixsl:updating="yes">
   <xsl:param name="response" as="map(*)"/>
   <xsl:result-document href="#target">
      <xsl:apply-templates select="doc($docURL)"/>
   </xsl:result-document>
```

# EXAMPLE 3

Supplying a JSON map as the argument to a JavaScript method

*"How do I supply a JSON map as the argument for a call to a method on a JavaScript object?"*

# HOW IS THIS EXAMPLE INTERESTING?

- This example demonstrates a fundamental challenge for users of SaxonJS: passing objects across the boundary between XDM and JavaScript.
- Many difficulties arise because of the different object models in XML languages versus JavaScript.
- This has also been an on-going headache for implementors!

# WHAT IS THE PROBLEM?

- The user wanted to be able to call

```
element.scrollIntoView({behavior:"smooth",
   block:"start", inline:"nearest"})
```

- Could use `ixsl:call()` for calling methods on JavaScript objects, e.g. `ixsl:call($targetElement, 'scrollIntoView', [])`, but not sure how to pass the `ScrollIntoViewOptions` in the array of arguments.

# SAXONJS 2 LIMITATION

- Users may expect to be able to supply an XDM map, assuming it will be converted to a JSON map by SaxonJS.
- But this will not work as expected, because the XDM to JS conversion in SaxonJS does not automatically convert XDM maps to JSON maps.

# SAXONJS CONVERSIONS BETWEEN XDM AND JS

- In trivial cases, SaxonJS users should not have to think too much about conversions from XDM to JavaScript, and vice versa.
- The conversions for strings, booleans, and numbers are straightforward.
- However, for slightly more complex values, the difference in the data models becomes apparent, and it becomes more necessary to be aware of how SaxonJS converts objects.

# SAXONJS 2 LIMITATION

- With SaxonJS 2, no XDM value is converted to a JSON map.
- Work arounds are possible if you instead construct the JSON map from the JavaScript side
  - e.g. Use a stylesheet parameter whose value is a JSON map. From within the XSLT, this is treated as a `JSValue`-wrapped JavaScript object.

# SAXONJS 3 SOLUTION

- What is missing in SaxonJS 2 is a way to construct JSON objects directly from the XSLT side, which avoids the usual SaxonJS conversion from XDM to JS.
- In SaxonJS 3, the `ixsl:json-parse()` function is provided which plugs this gap.
- This function parses JSON text supplied in an XDM string, and returns the resulting JavaScript object in a `JSValue` wrapper, avoiding any other internal XDM-JS conversion.

# SAXONJS 3 SOLUTION

```
<xsl:variable name="targetElement"
   select="ixsl:page()//div[@id eq 'target']"/>
<xsl:variable name="json-string" as="xs:string">
{"behavior":"smooth", "block":"start", "inline":"nearest"}
</xsl:variable>
<xsl:sequence
   select="ixsl:call($targetElement, 'scrollIntoView',
      [ixsl:json-parse($json-string)])"/>
```

Useful tip: providing the JSON text as a string in a variable means you can avoid using character escapes!

# SUMMARY

- We have looked at some examples of new IXSL features for SaxonJS 3, motivated by feedback from users of SaxonJS 2.
- Designing new language features involves considering usability as well as functionality.

# SUMMARY

- A language should include features that users want to use, providing ways to do things that are both powerful and intuitive.
- New syntax should make sense to the user, by fitting in with the existing syntax, and perhaps by being similar to existing solutions in other languages.
- As well as solving specific problems, designers must consider the wider picture, and provide enough flexibility to enable greatest use.

# THANK YOU FOR LISTENING

## ANY QUESTIONS?