ASYNCHRONY WITH PROMISES IN SAXONJS

Declarative Amsterdam 2023

Debbie Lockett
debbie@saxonica.com

SAXONICA.COM
XSLT AND XQUERY PROCESSING

OUTLINE

Aim: introduce IXSL promises which will be available in the next major SaxonJS release

- Motivation for the development
- Some background on asynchronous processing and JavaScript promises
- Introduce the new IXSL syntax
- Show some examples

INTRODUCTION

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.

ASYNCHRONOUS IXSL

Asynchronous processing from XSLT currently available in SaxonJS using the extension instruction `ixsl:schedule-action`.

IXSL:SCHEDULE-ACTION EXAMPLE

Fetch an XML document and copy into the HTML page.

```
<xsl:template match="button" mode="ixsl:onclick">
   <ixsl:schedule-action document="{$href}">
      <xsl:call-template name="action">
         <xsl:with-param name="href" select="$href"/>
      </xsl:call-template>
   </ixsl:schedule-action>
</xsl:template>

<xsl:template name="action">
   <xsl:param name="href"/>
   <xsl:result-document href="#target">
      <xsl:copy-of select="doc($href)"/>
   </xsl:result-document>
</xsl:template>
```

EXAMPLE DESCRIPTION

- `document` attribute specifies resource to fetch.
- Contained `xsl:call-template` provides callback action for when the asynchronous fetch returns.
- If the document fetch is successful, it is added to the document pool cache.
- The `fn:doc()` call in the callback action template gets the document from the cache.
- Content from the document is added to the HTML page using `xsl:result-document`.

IXSL:SCHEDULE-ACTION

`ixsl:schedule-action` instruction has a choice of attributes for different asynchronous operations:

- `wait` to delay an action
- `document` for loading documents asynchronously
- `http-request` for HTTP requests

But this set of asynchronous operations is restricted and not easily extensible.

IXSL:PROMISE

New instruction `ixsl:promise` designed to replace `ixsl:schedule-action` for asynchronous processing in XSLT with SaxonJS 3.

- Can call any asynchronous function, e.g. to fetch any kind of resource.
- Design and implementation closely aligned with JavaScript Promises - making it easier for those familiar with the JavaScript processing model.

ASYNCHRONY IN SAXONJS 3

- Updated internal implementation to now use the promise-based Fetch API for loading documents and the HTTP client in the browser and on Node.js.

  (SaxonJS 2 uses XMLHttpRequest in the browser, and does not provide HTTP access in Node.js.)

- New interactive XSLT extensions for promise-based asynchronous processing.

FURTHER IDEAS

- The new IXSL extensions are based on syntax extensions proposed by M. Kay at Balisage 2020 in Asynchronous XSLT.
- That paper also contains more ambitious proposals to further address other limitations, not covered here.

ASYNCHRONOUS PROCESSING AND PROMISES IN JAVASCRIPT

ASYNCHRONOUS PROCESSING

- Making an HTTP request (e.g. to load a document) is a slow process.
- This can be done synchronously from a browser, but that is not a good idea because while the request is happening, the browser will become inactive.
- Instead, processes like this should be done asynchronously, allowing other processing to happen at the same time.
- On Node.js, HTTP requests are **only** available asynchronously.

CALLBACKS

How do you write asynchronous code?

One approach is to make functions that perform an asynchronous operation take an extra argument, a *callback function*. The operation is started, and when it finishes, the callback function is called with the result.

CALLBACK EXAMPLE

Use the Node.js `fs.readFile()` method to asynchronously read a file:

```
function callback(err, data) {
  if (err) throw err;
  console.log(data);
}

fs.readFile('myfile.txt', callback);
```

PROBLEMS WITH CALLBACKS

Callbacks make asynchronous processing possible.

But coding problems arise with nested callbacks:

- callback pyramid of doom
- error handling can get messy

PROMISES

- Foundation of asynchronous processing in modern JavaScript.
- Rather than arranging for a function to be called at some point in the future, return an object that represents this future event.
- A `Promise` is an object representing the eventual completion or failure of an asynchronous operation.
- Handler functions are attached to a promise object, instead of passing callbacks into a function.

PROMISE TERMINOLOGY

A *promise* is in one of 3 states:

- *pending* - initial state
- *fulfilled* - if the operation completed successfully
- *rejected* - if the operation failed

A promise is *settled* when it is fulfilled or rejected.

HANDLERS

Handler functions are attached to a promise using the `then()` and `catch()` methods:

- `then()` handler is called when the promise is fulfilled (i.e. when the asynchronous operation succeeds). The handler function acts on the fulfillment value.
- `catch()` handler is called when the promise is rejected (i.e. when the asynchronous operation fails). The handler function acts on the error object.

PROMISE CHAINING

- A key feature of promises is that the `then()` and `catch()` methods themselves return a promise. This new promise will be settled with the result of the corresponding handler function.
- This enables *promise chaining*, which makes it possible to avoid the problems with nested callbacks.

EXAMPLE

Use the Node.js `fsPromises.readFile()` method to asynchronously read a file:

```
fsPromises.readFile('myfile.txt')
  .then(data => {
    console.log(data);
  })
  .catch(err => {
    throw err;
  });
```

CHAINS

```
doSomething(A)
  .then(B => { ... })
  .then(C => { ... })
  .then(D => { ... })
  .catch(err => { ... })
    ...
  .then(X => { ... })
    ...
  .catch(err => { .. });
```

PROMISES IN INTERACTIVE XSLT

USING PROMISES FROM XSLT

The new `ixsl:promise` instruction, available in SaxonJS 3, has one required attribute:

- `select` specifies the asynchronous process - this must be an expression that returns a *promise*

and two optional attributes:

- `on-completion` handler when promise is fulfilled
- `on-failure` handler when promise is rejected

No body content, and immediately returns an empty sequence.

IXSL:PROMISE EXAMPLE

Fetch an XML document and copy into the HTML page.

```
<xsl:template match="button" mode="ixsl:onclick">
   <ixsl:promise select="ixsl:doc($href)"
    on-completion="f:go#1"/>
</xsl:template>

<xsl:function name="f:go" ixsl:updating="true">
   <xsl:param name="doc" as="document-node()"/>
   <xsl:result-document href="#target">
      <xsl:copy-of select="$doc"/>
   </xsl:result-document>
</xsl:function>
```

HANDLER FUNCTIONS

- The mechanism takes advantage of XDM higher-order functions.
- The `on-completion` and `on-failure` attributes of `ixsl:promise` are used to specify the handler functions for when the promise is fulfilled or rejected, respectively.

USING HIGHER-ORDER FUNCTIONS

These attributes accept an expression which returns a single-argument XDM function. For instance:

- a named function reference
  e.g. `f:go#1`
- a partial function application
  e.g. `f:go($input, ?)`
- any other expression that returns a suitable function

IXSL:UPDATING

- A common requirement is for the handler action to have updating side-effects, e.g by calling `xsl:result-document`.
- But usually `xsl:result-document` is not permitted inside a stylesheet function.
- The `ixsl:updating="yes"` attribute is used to break that rule, and allow side-effects for handler functions.

HANDLER FUNCTIONS

- If the promise is fulfilled, the `on-completion` function will be called with the fulfillment value of the promise.
- If the promise is rejected, the `on-failure` function will be called with an XDM map containing the details of the rejection.

THEN AND CATCH?

- Note that `on-completion` and `on-failure` are not completely equivalent to JavaScript `then()` and `catch()`.
- They are used to register handler functions, but `on-completion` and `on-failure` do not return promises.
- As we will see later, promise chaining is instead enabled with a new function `ixsl:then()`.

HOW DO YOU CREATE A PROMISE?

- A number of new core functions are provided which return promises: e.g. `ixsl:doc()` returns a promise that when fulfilled returns a document node.
- Alternatively you can call a user-written function which returns a wrapped JavaScript promise (which when fulfilled, must return an XDM item).

PROMISE TYPE

- In the following, type `promise(T)` means a promise which when fulfilled delivers an XDM value of type T.
- Adding this new primitive item type for a promise was proposed in Asynchronous XSLT.
- *This new type is not actually implemented and available in XPath, it is just used for documentation purposes.*

NEW IXSL FUNCTIONS FOR ASYNCHRONOUS PROCESSING WITH PROMISES

NEW INTERACTIVE XSLT FUNCTIONS

A number of new core functions are provided for working with promises from XSLT.

CREATING PROMISES

Asynchronous versions of document fetching XPath functions, which return a promise:

- `ixsl:doc($href)`
  returns `promise(document-node()?)`
- `ixsl:json-doc($href, $options)`
  returns `promise(item()?)`
- `ixsl:unparsed-text($href, $encoding)`
  returns `promise(xs:string?)`

CREATING PROMISES

Other asynchronous functions which return a promise:

- `ixsl:sleep($wait)` - set a delay, returns `promise(empty-sequence())`
- `ixsl:http-request($request)` - issue an HTTP request, returns `promise(map(*))`

CHAINING PROMISES

- `ixsl:then($promise, $function)` - schedule a handler function for the fulfillment of a promise, and return a new promise

Note that the handler function may explicitly return a promise, or it may not; but the `ixsl:then()` function always returns a promise.

CONCURRENCY METHODS

Analogous to JavaScript Promise methods - used to run multiple asynchronous processes together:

- `ixsl:all($promises)` - fulfilled when all input promises fulfill, rejects if any reject
- `ixsl:all-settled($promises)` - fulfilled when all input promises settle
- `ixsl:any($promises)` - fulfilled when the first input promise fulfills, rejects if all reject
- `ixsl:race($promises)` - settles with first promise to settle (may be fulfilled or rejected)

OTHER ASSOCIATED FUNCTIONS

- `ixsl:abort-controller($timeout)` - create an abort controller to be used with `ixsl:doc()`, `ixsl:http-request()`, etc.

Used to enable timeouts or user-triggered aborts for the corresponding requests.

EXAMPLES

EXAMPLE 1

Fetch a JSON document, extract a URI, fetch an XML document, and copy into the HTML page.

1. Use `ixsl:json-doc()` to fetch the JSON document
2. Extract a new URI
3. Use `ixsl:doc()` to fetch the XML document
4. Copy into the HTML page

Let's look at different ways of doing this...

Using nested `ixsl:promise`

```
 1 <xsl:template match="button" mode="ixsl:onclick">
 2   <ixsl:promise select="ixsl:json-doc($href)"
 3     on-completion="f:get-data#1"/>
 4 </xsl:template>
 5
 6 <xsl:function name="f:get-data" ixsl:updating="true">
 7   <xsl:param name="json" as="map(*)"/>
 8   <xsl:variable name="uri" select="string($json?uri)"/>
 9   <ixsl:promise select="ixsl:doc($uri)"
10     on-completion="f:go#1"/>
11 </xsl:function>
12
13 <xsl:function name="f:go" ixsl:updating="true">
14   <xsl:param name="doc" as="document-node()"/>
15   <xsl:result-document href="#target">
```

Using nested `ixsl:promise`

```
1  <xsl:template match="button" mode="ixsl:onclick">
2    <ixsl:promise select="ixsl:json-doc($href)"
3      on-completion="f:get-data#1"/>
4  </xsl:template>
5
6  <xsl:function name="f:get-data" ixsl:updating="true">
7    <xsl:param name="json" as="map(*)"/>
8    <xsl:variable name="uri" select="string($json?uri)"/>
9    <ixsl:promise select="ixsl:doc($uri)"
10     on-completion="f:go#1"/>
11 </xsl:function>
12
13 <xsl:function name="f:go" ixsl:updating="true">
14   <xsl:param name="doc" as="document-node()"/>
15   <xsl:result-document href="#target">
```

Using chaining with `ixsl:then()` to avoid nesting

```
<xsl:template match="button" mode="ixsl:onclick">
  <ixsl:promise
    select="ixsl:json-doc($href) => ixsl:then(f:get-data#1)"
    on-completion="f:go#1"/>
</xsl:template>

<xsl:function name="f:get-data">
  <xsl:param name="json" as="map(*)"/>
  <xsl:variable name="uri" select="string($json?uri)"/>
  <xsl:sequence select="ixsl:doc($uri)"/>
</xsl:function>

<xsl:function name="f:go" ixsl:updating="true">
  ...
</xsl:function>
```

Using `ixsl:then()` again:

```
<xsl:template match="button" mode="ixsl:onclick">
  <ixsl:promise
    select="ixsl:json-doc($href)
      => ixsl:then(f:get-uri#1)
      => ixsl:then(ixsl:doc#1)"
    on-completion="f:go#1"/>
</xsl:template>

<xsl:function name="f:get-uri" as="xs:string">
  <xsl:param name="json" as="map(*)"/>
  <xsl:sequence select="string($json?uri)"/>
</xsl:function>

<xsl:function name="f:go" ixsl:updating="true">
  ...
```

Further simplification using an inline function:

```
<xsl:template match="button" mode="ixsl:onclick">
  <ixsl:promise
    select="ixsl:json-doc($href)
      => ixsl:then(function($json){string($json?uri)})
      => ixsl:then(ixsl:doc#1)"
    on-completion="f:go#1"/>
</xsl:template>

<xsl:function name="f:go" ixsl:updating="true">
  ...
</xsl:function>
```

EXAMPLE 2

Asynchronously fetch multiple resources, proceeding when they are all available.

- Use `ixsl:doc()` to fetch an XML document
- Use `ixsl:unparsed-text()` to fetch a text resource
- Use `ixsl:all()` to fetch the resources concurrently

Promise concurrency with ixsl:all():

```
<xsl:template name="xsl:initial-template">
  <ixsl:promise
    select="ixsl:all([ixsl:doc($a),ixsl:unparsed-text($b)])"
    on-completion="f:go#1"
    on-failure="f:fail#1"/>
</xsl:template>

<xsl:function name="f:fail" ixsl:updating="true">
  <xsl:param name="err" as="map(*)"/>
  <xsl:result-document href="{$output-file}"
    expand-text="yes">
    <out>Document not available: {$err?message}</out>
  </xsl:result-document>
</xsl:function>
```

EXAMPLE 3

Document fetch with a timeout.

- Use `ixsl:doc()` to fetch a document
- Use `ixsl:abort-controller()` to add a timeout

Using an abort controller:

```
<xsl:template name="xsl:initial-template">
  <ixsl:promise select="ixsl:doc($href,
      ixsl:abort-controller(1000))"
    on-completion="f:go#1"
    on-failure="f:fail#1"/>
</xsl:template>

<xsl:function name="f:go" ixsl:updating="true">
  <xsl:param name="doc" as="document-node()"/>
  <xsl:result-document href="{$output-file}">
    <wrapper for="{$href}"><xsl:sequence select="$doc"/></wra
  </xsl:result-document>
</xsl:function>

<xsl:function name="f:fail" ixsl:updating="true">
```

SUMMARY

- We've made good progress with the new design for promise-based IXSL for SaxonJS 3, and hopefully this presentation has given you a good idea of what's to come.
- With `ixsl:promise` it is possible to do everything that was previously possible with `ixsl:schedule-action`, and more.
- The new design should be easier to use, especially for those familiar with JavaScript promises.

FURTHER WORK

M. Kay's Asynchronous XSLT paper contained more ideas which address other issues not covered here.

These are not yet implemented but being considered:

- promise item type
- *asynchronous* variables to provide an "await" mechanism

THANK YOU FOR LISTENING

ANY QUESTIONS?

SAXONICA.COM
XSLT AND XQUERY PROCESSING