ScriptTypeConfiguration

Various custom configuration elements support a common content model used to supply JSR-223 scripts inside other object configurations. Where applicable, the specific configuration elements that share this content model will link to this page. These include the following documentation pages:

- ConditionScript
- AttributeFilterScript
- ResponseMapping

This feature is new in V3.4 and is a supplement to the older supported approach of defining scripted beans in native Spring files and referring to the beans by reference.

Namespace and Schema

Configuration elements that contain scripts are of type `ScriptType`, which is a type used across a number of namespaces. The specific namespace will depend on where the element shows up in your configuration.

The following sections describe the attributes and child elements of an element of type `ScriptType`.

Attributes

An element of type `ScriptType` has the following XML attributes:

Name	Type	Use	Default	Description
`language`	string	optional	`"javascript"`	Defines the JSR-223 language to use. The default is ECMAScript using either the Rhino (Java 7) or Nashorn (Java 8) engines.
`customObjectRef` 3.2	string	optional		The ID of a Spring bean defined elsewhere in the configuration.

If the `customObjectRef` attribute is present, the result of the referenced Spring bean is made available to the script in a variable named `custom`. This is in addition to the normal script context discussed below.

Child Elements

An element of type `ScriptType` has the following child elements:

Name	Cardinality	Description
`<Script>`	Exactly One	An inline script
`<ScriptFile>`		Path to a local file or classpath resource containing the script

The script may be stored in a local file (with `<ScriptFile>`) or written inline (with `<Script>`). An inline script should be wrapped with a CDATA section to prevent interpretation of any special XML characters that may be included in the script.

Always wrap inline scripts with a CDATA section

Always wrap inline scripts with a CDATA section, even if the script contains no special XML characters. This will future-proof your script.

Script Context

Each element of type `ScriptType` provides relevant script context, that is, one or more input objects (in the general sense) to be utilized by the script. For specific details, consult the individual configuration element pages listed above.

Examples

The following example illustrates the use of a CDATA section

A script wrapped with a CDATA section

```
<Script>
<![CDATA[
    // script goes here
]]>
</Script>
```

For additional examples of scripts, consult the individual configuration element pages listed above.

Scripting Language

The default scripting language is JavaScript (`language="javascript"`). Therefore all of the sample scripts are written in JavaScript, which is based on the ECMAScript standard. The following table illustrates the relationship between JRE version and ECMAScript version:

JRE Version	Default Script Engine	ECMAScript Version
Java 7	Rhino	ECMAScript for XML (E4X)
Java 8	Nashorn	ECMAScript 5.1 (June 2011)
Java 9	Nashorn	ECMAScript 5.1 (plus some features of ECMAScript 2015 aka ECMAScript 6)

Since Java 7 reached end-of-life on April 2015, we assume Java 8 or above, which implements Nashorn.

Nashorn documentation from Oracle

An introduction to Nashorn from Oracle:

Part 1: Introducing JavaScript, ECMAScript, and Nashorn
Part 2: The Java in JavaScript
Part 3: Database Scripting

If you're still using Rhino, the sooner you migrate to Nashorn, the better. That said, many of the sample scripts (which are quite simple) will run under both Rhino and Nashorn.

Still using Rhino?

Consult the Rhino Migration Guide for helpful advice.

Since Nashorn is included with Java 8 (and later), the sample scripts aim to conform to ECMAScript 5.1. In particular, the scripts avoid features introduced in ECMAScript 6 (also known as ECMAScript 2015) for compatibility.

ECMAScript 2015 is not supported

Language features introduced in ECMAScript 2015 (aka ECMAScript 6) are intentionally not used in the sample scripts for compatibility reasons. This includes `let`, `const`, and the so-called *fat arrow function* notation.

Many of the sample scripts are written in Strict Mode. Such a script will include an explicit "use strict" directive, which intentionally precludes the use of certain (error-prone) JavaScript features. Although Strict Mode was introduced in ECMAScript 5, the "use strict" directive is a harmless addition to any script. In particular, it has no effect (positive or negative) when used in scripts under Rhino. That said, you may disable Strict Mode at any time simply by removing (or commenting out) the "use strict" directive.

Follow best coding practices

Write all your scripts in Strict Mode. Moreover, check all your scripts against JSLint. Both practices will help you write better JavaScript code that is more easily debugged and maintained.

Using the JSLint tool

The JSLint tool cannot tell that the JavaScript is being run within the environment of the IdP, with the implied inputs and outputs that that infers. This manifests itself in two ways:

- JSLint does not like the `input` object and `custom`objects that seem to magically appear from nowhere. From a Shibboleth perspective, this is a feature, not a bug. You can safely ignore this warning issued by the JSLint tool.
- Equally it does not like the "last value is the implied return" paradigm. Should this grate you can fool JSLint by using a closure as the last line, for instance replacing

```
var retVal;
retVal = false;
//
// arbitrary code to set up retVal, consulting input and customref
//
retVal;
```

with

```
var retVal;
retVal = false;
//
// arbitrary code to set up retVal, consulting input and customref
//
(function (val) {
    return val;
}(retVal));
```

Only you can decide whether this will help maintainabilty. For clarity and didactic purpose the examples omit this paradigm.