SpringConfiguration

The IdP software makes extensive use of native Spring XML configuration for newer and advanced features.

General

For some helpful reference material on how native Spring syntax works, see:

- http://docs.spring.io/spring/docs/4.3.x/spring-framework-reference/html/beans.html
- http://docs.spring.io/spring/docs/4.3.x/spring-framework-reference/html/xsd-configuration.html

Creating New Files

If you need to create your own Spring bean file to import or load into a service's resource set, the following template should be used. While usable to some degree in XML editors, the use of the "p" and "c" shorthand namespaces is not really schema-compatible and is designed for use inside Spring-aware tools such as the Eclipse STS plugin.

Skeletal Spring File

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:util="http://www.springframework.org/schema/util"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema
/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context http://www.springframework.org/schema
/context/spring-context.xsd
                           http://www.springframework.org/schema/util http://www.springframework.org/schema/util
/spring-util.xsd"

       default-init-method="initialize"
       default-destroy-method="destroy">

</beans>
```

Spring Expressions

A key technology that can greatly simplify more complex Spring syntax is the Spring Expression Language:

- http://docs.spring.io/spring/docs/4.1.x/spring-framework-reference/html/expressions.html

Spring expressions in XML configuration are typically bracketed like so: `#{ expression }`

For just one simple example, populating list-based properties can require multiple lines of XML, or can be shorthanded with SpEL:

```
#{ {'item1', 'item2'} }
```

The comma-delimited list expression is denoted by braces (inside the overall expression braces) and string values are quoted.

Properties

Spring includes a feature called "property replacement" that allows macros in Spring XML to be replaced at runtime with the contents of Java property files, Java system properties, and environment variables.

By default, *idp.home/conf/idp.properties* contains the "master" list of properties, and contains a pointer to additional property files, to which you may add your own. All properties are "global" so it doesn't matter which file a property is in; the use of separate files is merely an organizational tool.

Unfortunately, the default syntax for Spring properties clashes with the Velocity macro syntax, which causes conflicts in the attribute resolver configuration. To avoid this problem, the Spring syntax has been customized to use a '%' character as a prefix. So properties in the files will appear as "%{propertyname}" or "%{propertyname:defaultvalue}". If you want to use your own properties, just keep in mind you need to use a different leading character than the "${propertyname}" syntax documented by Spring.

Combining Properties and Spring Expressions

You can generally nest properties inside of Spring expressions (or use them to actually carry expressions) in a natural fashion:

```
#{ %{my.property} }
```

The use of lists deserves some special consideration. Ordinarily, as described above, lists have to contain quoted values. If a property contains a list of values that are explicitly quoted, you can do something simple to substitute the property into a list:

```
#{ { %{my.list.property} } }
```

Alternatively, you might want to make the property simpler and more robust, avoiding quotes and automatically trimming any trailing spaces you accidentally add in an editor. You can generate a list to populate into a property with a bean like so:

```
<bean parent="shibboleth.CommaDelimStringArray" c:_0="#{ '%{my.list.property}'.trim() }" />
```

The parent bean shown is a utility bean we created to automate the use of a utility function that takes a comma-delimited string and turns it into an array of strings that will automatically turn into a collection inside Spring.

Spring Web Flow

While the use of Spring Web Flow (SWF) for user interaction is a major change from V2 to V3, we have tried to minimize the degree to which a typical deployer will need to interact with, let alone customize, flows. But we will eventually provide some notes on how and where this can be done for those with more ambitious needs.

You can safely skip the rest of this section unless/until you have a need to work with custom webflows or are building your own extensions.

The mapping of functionality to URL is by and large derived from a set of mappings that define the built-in flows that come with the software and the paths at which they should be installed. Starting with V3.3, this is managed with an extended bean factory that is more flexible than that provided by SWF natively, and it introduces a couple of new capabilities useful for plugins that are implementing custom flows or for advanced deployers:

- Overriding built-in system flows at a given location
- Adding flows automatically using a plugin jar

There are now Spring beans containing the mappings of flow definitions to locations and flow "patterns" to search for. The defaults are contained in a pair of beans, **shibboleth.DefaultFlowMap** and **shibboleth.DefaultFlowPatterns**. The latter will load flows from the following two locations automatically:

- *idp.home*/flows/**/*-flow.xml
- classpath*:/META-INF/net/shibboleth/idp/flows/**/*-flow.xml

The first of these loads flows defined in the user-modifiable configuration, as in previous releases. The second is new, and is provided to allow plugins to supply flow definitions at load time without requiring copying or other tricks.

Note that in both cases, the flow's name/location will be derived from the directory structure found in the wildcarded portion of the path, so a flow in */META-INF/net/shibboleth/idp/flows/authn/custom/custom-flow.xml* will be named "authn/custom", just as if it were placed in *idp.home/flows/authn/custom/custom-flow.xml*.

Many of the flows in the system fall into special groups that have to follow an additional naming convention. For example, a login flow name must start with the prefix "authn/", as in the example above.

If you have a need to override or supplement existing flow mappings, you can supply two maps called **shibboleth.FlowMap** and **shibboleth.FlowPatterns** to replace, or more likely merge themselves with, the default beans mentioned above. The main use case for this would be to override the flow installed to one of the built-in locations or to register one of the built-in flows at an additional location. The practical reason you would do this would be to add additional endpoints for various services to fit some kind of unusual requirement.

Pre-Supplied Beans and useful classes

A standard IdP configuration provides you with many "named beans" which are there to simplify configuration. Examples include (but are not limited to) the many ActivationConditions which are available.

The PredefinedBeans topic details them all, as well as providing a list of other "useful class names".