NativeSPApplicationOverride

Before getting into the gory details here, please be sure to read the NativeSPApplicationModel topic that explains the underlying concept of an "application" and how resources and applications relate. This topic is essentially a How-To guide with examples for implementing the configuration options introduced there.

Having digested that, what is an override? An override is a way to assign non-default sets of configuration rules to content by grouping the content into units called "applications" and then creating settings for those applications inside of an `<ApplicationOverride>` element.

The rule of thumb is that any settings you don't override inside the element will be inherited from the `<ApplicationDefaults>` element that surrounds the override. This is **generally** true, but there are exceptions and some of the rules get a little complex, so the specifics are covered down below and should address any questions about how the inheritance works.

Valid and Invalid Reasons for Additional Applications

Here are some use cases where an additional application has to be defined:

- overriding the entityID (i.e., exposing an additional logical SP from a single physical installation)
- different metadata or security policy
- different attribute mapping or filtering rules
- different rules for session behavior, such as timeouts

Here are some use cases that usually do **NOT** require an additional application be defined:

- use of a particular IdP or discovery service based on the resource
- use of different credentials (e.g., certificates) with different IdPs
- customized error handling

Some of those use cases did require additional application definitions in older versions, but now there are a variety of enhanced options available via content settings. Instead of creating a bunch of extra XML configuration, you can simply set properties using Apache or the `<RequestMapper>` based on the virtual host or path, and change which IdP gets used (`entityID`) or change many other behaviors.

Establishing the New Application

Assuming you're unlucky and actually need to create an additional application in the configuration, it's a two-part process:

Step 1: Create and name the application via `<ApplicationOverride>` and the `id` property.

Each application has a simple identifier associated with it. The "default" application that corresponds to everything initially is simply called "default" (duh). When you create new applications, you need to give them simple names to refer to them in the second step below. Using short, simple names is best, but any string is fine. Once you decide on a name, you insert an element like so just inside the `</ApplicationDefaults>` closing tag:

```
...
    <ApplicationOverride id="myappname" />
</ApplicationDefaults>
...
```

Initially the application doesn't **do** anything; there are no overridden settings and it may not even work properly, but we'll fix that later.

Step 2: Map the applicable resources using a matching `applicationId` property.

The second step is to associate the resources that make up the new application (and the associated handler path) to an `applicationId` content setting that matches the `id` you chose in the previous step. There are two ways to do this (the simple examples apply to an entire virtual host being mapped to the new application, other examples will be shown later).

Mapping via Apache Commands (httpd.conf or equiv.)

```
<VirtualHost>
...
ServerName myapphost.example.org:443
...
## Note, if you can't apply this globally, you need to make sure
## you apply the setting to both the content/resources *and* to
## the appropriate handler path (generally /Shibboleth.sso).
<Location />
ShibRequestSetting applicationId myappname
</Location>
...
</VirtualHost>
```

Mapping via the <RequestMap> (shibboleth2.xml)

```
<RequestMap applicationId="default">
...
    <Host name="myapphost.example.org" applicationId="myappname"/>
...
</RequestMap>
```

Obviously the former only works on Apache, IIS requires the latter approach. Apache commands are usually safer on Apache, because using the `<RequestMap>` will only work safely if you turn `UseCanonicalName On`.

As the comment above in the example notes, you MUST make sure that the SP handlers that are used with the application override are appropriately mapped to the same application ID as the resources. The handlers and resources are a unit.

Host vs. Path

A key issue to address when establishing additional applications is whether to organize applications by virtual host or by path. The reason for using paths is basically all about the overhead of virtual hosting, primarily in terms of having the ability to register them in your DNS and because of the need for additional certificates and IP addresses in most cases involving SSL.

Nevertheless, there are a couple of major reasons why the best thing to do is use virtual hosts: security and simplicity.

The security issue depends somewhat on your reasons for defining additional applications, but essentially any web resources that share a virtual host should be treated as being part of a single security "zone" because that's how browsers treat them. You cannot isolate applications that share a virtual host and keep them from seeing cookies restricted based on paths, and any scripts on a virtual host can manipulate the browser in order to leak information from one application to another. So if your reasons for the new application involve creating any real separation between resources, you have no choice but to use a different virtual host. Otherwise just don't bother.

The simplicity argument is related to how the SP software deals with separate applications, and the requirement (outlined here) that every application have a unique `handlerURL` prefix for its handlers.

When you use virtual hosting, there is no special handler configuration neeeded; you can simply inherit the default settings because the handlers will automatically be specific to each virtual host. When a request comes into `https://myapphost.example.org/Shibboleth.sso/SAML2/POST`, it will automatically be recognized as bound for the new application on that virtual host. All you have to do is ensure that the metadata you provide for the SP includes the necessary endpoints on that virtual host.

In contrast, when you divide things by path, you have to explicitly add XML to the configuration to establish a special `handlerURL` for the new application, as well as providing the necessary metadata. Typically it will look something like this:

Example of Path-Based Application Override (shibboleth2.xml)

```
<RequestMap applicationId="default">
    <Host name="www.example.org">
        <Path name="myappfolder" applicationId="myappname"/>
    </Host>
</RequestMap>
...
<ApplicationDefaults ...>
...
    <ApplicationOverride id="myappname">
        <Sessions lifetime="28800" timeout="3600" checkAddress="false" handlerURL="/myappfolder/Shibboleth.sso"
/>
    </ApplicationOverride>
</ApplicationDefaults>
```

The big thing to understand is that because of limitations on inheriting settings, you have to supply all the settings needed in the `<Sessions>` element because of the need to override the `handlerURL`. Since one setting is overridden, you have to supply all of them; none of the default properties from the outer `<Sessions>` element will be visible to the new application.

Luckily, you do **NOT** have to actually redefine all of the handler child elements. Those will still be inherited, and their locations will be adjusted relative to the new `handlerURL` you define.

The other key here is to recognize that the `handlerURL` lives **inside** the path you set aside for this application to use. It **MUST** map to the corresponding `applicationId`, just like any other resources that make up the application. It's literally part of the application. If you fail to make that connection, you will typically experience looping behavior or errors involving invalid audience conditions, because the SAML assertion delivered to the incorrectly specified `handlerURL` will be associated with the wrong application and entityID, usually the "default" one.

Using the Override

Once you have the additional application defined, you can proceed to define any new settings you need, and correcting the SP's metadata. The metadata can be taken care of first (see the following two sections). Below that, you'll find the precise rules for how settings get inherited, and some example configurations for typical use cases involving application overrides.

Metadata for an Existing SP

If the new application is intended to be "part of" an existing SAML SP that you've established, then you will need to modify the metadata for that SP (keyed by its `entityID`) to add the additional endpoints that will serve the new application, principally `<md:AssertionConsumerService>` elements, but possibly others as needed. The location of your new handlers will depend on whether you're using virtual hosts or paths to define the new application (per the previous section).

For a virtual hosting scenario, you should be inheriting a default `handlerURL` of "/Shibboleth.sso", so the ACS endpoints to add should have `Location` attributes like "https://myapphost.example.org/Shibboleth.sso/SAML2/POST" and so forth.

For a path-based scenario, you should be defining a custom `handlerURL` similar to "/myapppath/Shibboleth.sso", so the ACS endpoints to add should have `Location` attributes like "https://www.example.org/myapppath/Shibboleth.sso/SAML2/POST".

Of course, you'll need to add endpoints for any of the protocol bindings you want to support.

Metadata for a New SP

If the application is intended to function as a new logical SP with its own `entityID`, then all of the information in the previous section still applies, but of course the endpoints would be found in a new metadata instance (`<md:EntityDescriptor>`) with the new `entityID`.

In the majority of cases, it's reasonable (and much simpler) to just reuse the same key pair for the new SP. Since the system is physically a single unit, there's no real security advantage to creating separate keys. If one gets compromised, it's likely both will. In fact, a major advantage of using metadata is to indirect the identity of the SP from the keys that it can use, and it's fine for one key to be acceptable for use by more than one SP.

Inheritance Rules

The precise rules for how each top level `<ApplicationDefaults>` element is "inherited" when unspecified by a `<ApplicationOverride>` are specified below. In most cases if an element is **NOT** supplied at the override level, it will be inherited automatically. The exception, noted below, is that `<RelyingParty>` elements are **NOT** inherited unless Version 2.4 and above is used.

- `<Sessions>`
  - If present in the override, the default element's attribute content is ignored. Any child elements/handlers defined at the default level **WILL** apply to the override. If you specify child elements/handlers at the override level, any handlers at a `Location` used by a default handler will replace the default, while others will supplement the default set. In other words, you can add or replace default handlers, but not disable them.
- `<Errors>`
  - If present in the override, the default element's content is ignored.

- `<RelyingParty>`
    - As of Version 2.4, these elements are inherited if none are present in the override. On older versions, this element is **NOT** inherited and must be explicitly specified at the override level.
- `<Notify>`
    - If any are present in the override, the default elements, if any, are ignored.
- `<MetadataProvider>`
- `<TrustEngine>`
- `<AttributeExtractor>`
- `<AttributeResolver>`
- `<AttributeFilter>`
- `<CredentialResolver>`
    - If present in the override, the corresponding default element is ignored.

Common Example Scenarios

These examples show how to use the `<ApplicationOverride>` element to achieve particular goals. The examples don't show the position of the element or the associated changes to achieve the mapping between resources and the new application. You can find examples for that earlier in this topic.

Of course, you can compose these examples as needed.

The most common case is usually wanting to virtualize the SP by making an application act as a new logical SP with a different entityID. This is very simple.

Application Acting as a Distinct SP

```
<ApplicationOverride id="myappname" entityID="https://myapp.example.org/shibboleth"/>
```

If you need to use different lifetime/timeout values for sessions with the new application, or are creating a path-based application that needs its own `handlerURL`, you need to supply a fully populated `<Sessions>` element. Any properties you don't supply will **not** be inherited, but will get their **default** documented values.

As noted earlier, when you supply a custom `<Sessions>` element, you don't need to repeat all of the child elements that define the handlers unless you want to change them in some way also.

Alternate Session Options

```
<ApplicationOverride id="myappname">
    <Sessions lifetime="28800" timeout="3600"
        handlerURL="/myapppath/Shibboleth.sso" handlerSSL="true"
        cookieProps="; path=/myapppath; secure; HttpOnly"/>
</ApplicationOverride>
```

If you want to customize the IdP metadata used by an application, you simply provide the desired `<MetadataProvider>`.

Custom Metadata

```
<ApplicationOverride id="myappname">
    <MetadataProvider type="XML" file="idpformyapp-metadata.xml"/>
</ApplicationOverride>
```

If you want to customize the SAML attribute handling for an application, you simply provide the desired `<AttributeExtractor>`.

Custom Attribute Mappings

```
<ApplicationOverride id="myappname">
    <AttributeExtractor type="XML" file="myapp-attribute-map.xml"/>
</ApplicationOverride>
```

If you want to assign a different private key or certificate to an application, you simply provide the desired `<CredentialResolver>`.

Custom Credentials

```
<ApplicationOverride id="myappname">
    <CredentialResolver type="File" key="myappname-key.pem" certificate="myappname-cert.pem"/>
</ApplicationOverride>
```