ActivationConditions

- Concepts
- Use Cases
- Attaching Conditions
    - Native Spring Syntax
    - Custom Sytax
- Examples
    - Specific Relying Parties
    - Relying Parties By Group
    - Relying Parties By Tag
    - Client Address Ranges
    - Attribute Checking
    - Compound Boolean Conditions

Concepts

A widespread pattern throughout much of the IdP configuration is the concept of an "activation condition". This is a mechanism by which functions and settings can be selectively enabled or disabled at runtime based on an arbitrary condition that is evaluated for each request. This addresses the ability to easily extend the behavior of components and handle new use cases. For consistency, we've tried to use the property name "activationCondition" across many of the components with this feature.

An activation condition, formally, is a Java bean of type `Predicate<ProfileRequestContext>`. A Predicate is a very simple generic interface that implements a single method:

```
public interface Predicate<T> {
        boolean apply(T input)
}
```

That is, it lets the IdP run a method against an input object to get a yes/no answer. The input in the majority of cases is the "root" object of the context state tree that tracks a request from start to finish, of type org.opensaml.profile.context.ProfileRequestContext. By consistently basing everything off of that abstraction, it's possible to build up a library of condition objects that can be used in many different places to implement some kind of check.

The IdP includes a fairly extensive library of such conditions, and there are a number of pre-defined beans provided to help make configuration simpler for a lot of common use cases and we can extend that set over time (and it's easy to submit more examples here that people can cut and paste).

We also supply public implementations of this interface as of V3.2.0 such that the logic can be supplied via a script or via a Spring Expression, avoiding the overhead of creating a new Java class in a separate jar.

Use Cases

Some of the advanced uses you might have for this feature include:

- Creating custom relying party overrides beyond the ones supplied
- Limiting when authentication flows may be used
- Limiting when NameID generation or consumption plugins may be used
- Controlling the use of attribute resolver connectors, definitions, or encoders
- Determining when profile intercept flows run, such as attribute consent
- Extending the attribute filter policy language

In most cases, using this feature amounts to setting a bean implementing the condition you want into a component bean's `activationCondition` property. With the older configuration files that are not Spring bean files, there will usually be some kind of XML attribute used to reference to condition (e.g. `activationConditionRef`) and you will have to define the condition bean in a separate file. This isn't ideal, but it allows for essentially any condition to be attached without regard for how complex its own configuration might be.

Defining the condition bean

The location where the condition bean would be defined depends on the use case. It's generally safe to define them in *global.xml,* as long as the naming is unique. For the most isolation, and to support reloadability of their definitions, it's better to define them inside specific service contexts when possible, generally by adding a bean resource file to the set of resources defined in *services.xml* for a given service.

Attaching Conditions

The various examples below demonstrate the creation of stand-alone beans of the appropiate type. To actually use them in specific cases typically requires injecting the bean ID into a property named `activationCondition-ref` in another bean, or using a custom XML attribute.

Native Spring Syntax

For instance, attaching a condition bean named "MyCondition" to control the execution of a NameID generator bean in *saml-nameid.xml* looks like this:

```
<bean parent="shibboleth.SAML2AttributeSourcedGenerator"
        p:format="urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress"
        p:attributeSourceIds="#{ {'mail'} }"
        p:activationCondition-ref="MyCondition" />
```

Custom Sytax

In contrast, attaching a condition to an attribute defintion looks like this:

```
<resolver:AttributeDefinition xsi:type="Simple" xmlns="urn:mace:shibboleth:2.0:resolver:ad" id="attr1"
        sourceAttributeID="attr1source" activationConditionRef="MyCondition">
   <resolver:Dependency ref="staticAttributes" />
</resolver:AttributeDefinition>
```

In this case the bean definition for the activation condition (`MyCondition` above) should be included in a suitable native spring file.

- Simple conditions, and those requiring to be reloadable, should be places in a stand alone file which is referenced by the `conf\services.xml` file.

  Bean defintion for selection by relying party id

  ```
  <?xml version="1.0" encoding="UTF-8"?>
  <beans xmlns="http://www.springframework.org/schema/beans"
      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">

      <bean id="ExampleOrgPredicate" parent="shibboleth.Conditions.RelyingPartyId" c:candidate="https://sp.
  example.com/shibboleth" />
  </beans>
  ```

  This bean file must be supplied to the appropriate resource list in the *services.xml* file. For instance for the Attribute Resolver.

  Including an extra file in services.xml

  ```
  <util:list id ="shibboleth.AttributeResolverResources">
      <value>${idp.home}/conf/attribute-resolver.xml</value>
      <value>${idp.home}/conf/THE_NEW_FILE.xml</value>
  </util:list>
  ```

- Complex beans that will *never* need to be reloaded can be put into the `conf\global.xml` file.

Examples

If you develop useful examples, please consider adding them to this page. The examples below are not fully tested in a few cases, so please report any problems using them to the mailing list or make corrections as you find them. Some of the examples require IdP V3.4 (or later) and do not work with earlier versions of the software.

Specific Relying Parties

This is a common use case, and simple to grasp, so we use it to demonstrate a variety of different approaches that you can apply to other use cases once the syntax makes sense to you.

A built-in bean is provided to make this use case simple. You should provide a list of names, providing a single name may not have the expected result.

Specific Relying Parties by Name

```
<!-- Single SP -->
<bean id="MyCondition" parent="shibboleth.Conditions.RelyingPartyId"
        c:candidate="https://sp.example.com/shibboleth" />

<!-- Multiple SPs, expression -->
<bean id="MyCondition" parent="shibboleth.Conditions.RelyingPartyId"
        c:candidates="#{{'https://sp.example.com/shibboleth', 'https://another.example.com/shibboleth'}}" />

<!-- Multiple SPs, list bean -->
<bean id="MyCondition" parent="shibboleth.Conditions.RelyingPartyId">
        <constructor-arg name="candidates">
                <list>
                        <value>https://sp.example.com/shibboleth</value>
                        <value>https://another.example.com/shibboleth</value>
                </list>
        </constructor-arg>
</bean>
```

A more advanced option supported by the same parent bean is the ability to plug-in an arbitrary condition to run against the relying party name. The input to such a condition is a String rather than the ProfileRequestContext.

Test a Regular Expression

```
<!-- Test a regular expression against the Relying Party name -->
<bean id="MyCondition" parent="shibboleth.Conditions.RelyingPartyId">
        <constructor-arg>
                <bean class="com.google.common.base.Predicates" factory-method="containsPattern"
                        c:pattern="^https://[^/]+\.example\.com/shibboleth$" />
        </constructor-arg>
</bean>
```

You can also write a script (Javascript being the default language):

Javascript Example

```
<!-- A script that checks a Relying Party name -->
<bean id="MyCondition" parent="shibboleth.Conditions.Scripted" factory-method="inlineScript">
        <constructor-arg>
                <value>
        <![CDATA[
                        "use strict";
                        var result = false;

            // an implementation of Predicate<ProfileRequestContext>
            // The IdP environment provides two variables "profileContext" and "custom".
            //     profileContext  is of type org.opensaml.profile.context.ProfileRequestContext
            //     custom          is whatever you injected
            // The value of the last statement in this function is the reurn value
            var id = "https://sp.example.com/shibboleth";  // an entityID

            // specify the child context of the root ProfileRequestContext
            var subcontextClass = "net.shibboleth.idp.profile.context.RelyingPartyContext";
            var subcontext;

            // check the parameter
            if (profileContext!== null) {
               // check the entityID of the relying party
               subcontext = profileContext.getSubcontext(subcontextClass);
               if (subcontext !== null) {
                  result = subcontext.getRelyingPartyId().equals(id);
               }
             }
             result;
        ]]>
                </value>
        </constructor-arg>
</bean>
```

Finally, you can write Spring Expressions, sort of a more concise scripting format:

Spring Expression Example

```
<!-- A Spring Expression that checks a Relying Party name -->
<bean id="MyCondition" parent="shibboleth.Conditions.Expression">
        <constructor-arg>
                <value>
                        #profileContext.getSubcontext(T(net.shibboleth.idp.profile.context.
RelyingPartyContext)).getRelyingPartyId().equals("https://sp.example.com/shibboleth")
                </value>
        </constructor-arg>
</bean>
```

Relying Parties By Group

In this context, grouping RPs is historically done by containing SAML metadata entities in `<md:EntitiesDescriptor>` elements, something common to aggregated files produced by many federations. This is more of a historical use case, because it's dangerous to use with aggregates you don't control. The containment model is very limiting and often doesn't mean what people think it does, so it's better to use the "tag" approach illustrated below.

But for aggregates you control locally, it's a simple way to apply policy and solve certain problems, such as grouping together SAML 1-only systems or grouping systems that don't support encryption, or grouping together systems by their attribute needs.

As of V3.4, group matching is extended to check for `<AffiliationDescriptor>` content in metadata, which is a way of providing explicit "out of band" groups of entities (analagous to creating an LDAP group that lists its members, rather than embedding the memberships into each member's entry).

Most use cases for this feature tend to be for relying party overrides, which are already supported separately. If you need to use this kind of condition elsewhere, you can reuse the same code with this example:

Relying Parties By Group

```
<!-- One group -->
<bean id="MyCondition" parent="shibboleth.Conditions.EntityDescriptor">
        <constructor-arg name="pred">
                <bean class="org.opensaml.saml.common.profile.logic.EntityGroupNamePredicate"
                        c:_0="nameofgroup" />
        </constructor-arg>
</bean>

<!-- Multiple groups, expression -->
<bean id="MyCondition" parent="shibboleth.Conditions.EntityDescriptor">
        <constructor-arg name="pred">
                <bean class="org.opensaml.saml.common.profile.logic.EntityGroupNamePredicate"
                        c:_0="#{{'group1', 'group2'}}" />
        </constructor-arg>
</bean>

<!-- Multiple groups, list bean -->
<bean id="MyCondition" parent="shibboleth.Conditions.EntityDescriptor">
        <constructor-arg name="pred">
                <bean class="org.opensaml.saml.common.profile.logic.EntityGroupNamePredicate">
                        <constructor-arg>
                                <list>
                                        <value>group1</value>
                                        <value>group2</value>
                                </list>
                        </constructor-arg>
                </bean>
        </constructor-arg>
</bean>
```

Relying Parties By Tag

The "modern" approach to attaching policy to relying parties is to leverage so-called tags, which refers to a metadata extension called an "Entity Attribute", a SAML attribute embedded in a system's metadata that is about the system rather than an individual user. Entity attributes have a variety of use cases, essentially anything one can think of saying about a system, and they're more flexible than groups because they can be attached directly to the entity's metadata regardless of where it's published. If you will, this is the equivalent of an LDAP "memberOf" attribute as opposed to a directory object that contains all the members of a group.

Most use cases for this feature tend to be for relying party overrides, which are already supported separately. If you need to use this kind of condition elsewhere, you can reuse the same code with this example:

Relying Party By Tag

```
<!-- Tag condition -->
<bean id="MyCondition" parent="shibboleth.Conditions.EntityDescriptor">
        <constructor-arg name="pred">
                <bean class="org.opensaml.saml.common.profile.logic.EntityAttributesPredicate">
                    <constructor-arg>
                        <list>
                                        <bean class="org.opensaml.saml.common.profile.logic.
EntityAttributesPredicate.Candidate"
                                                c:name="http://macedir.org/entity-category"
                                       p:values="http://refeds.org/category/research-and-scholarship" />
                                </list>
                    </constructor-arg>
                 </bean>
         </constructor-arg>
</bean>
```

Client Address Ranges

Some components may be sensitive to the address of the client, perhaps to distinguish users running on an enterprise network vs. working remotely. A condition class has been created to make this check simple using CIDR masks. Note that a servlet request bean declared by the Shibboleth software MUST be injected into any bean that needs access to the current request (the same is true of the servlet response in other cases).

Address Range Examples

```
<!-- Example for V3.3+ -->
<bean id="MyCondition" class="org.opensaml.profile.logic.IPRangePredicate"
        p:httpServletRequest-ref="shibboleth.HttpServletRequest"
        p:ranges="#{ '192.168.1.0/24', '192.168.2.0/28' }" />

<!-- Example for older versions. -->
<bean id="MyCondition" class="org.opensaml.profile.logic.IPRangePredicate"
                p:httpServletRequest-ref="shibboleth.HttpServletRequest">
        <property name="addressRanges">
                <list>
           <bean class="net.shibboleth.utilities.java.support.net.IPRange"  factory-method="parseCIDRBlock" c:
cidrBlock="192.168.1.0/24"/>
           <bean class="net.shibboleth.utilities.java.support.net.IPRange"  factory-method="parseCIDRBlock" c:
cidrBlock="192.168.2.0/28"/>
                </list>
        </property>
</bean>
```

Attribute Checking

Some components may need to check for the presence (or absence) of a particular attribute or value for a user. A basic condition is provided for this purpose, or may be a useful code example to follow to implement something more complex.

The class provided supports only simple string-valued attributes, and supports a simple form of wildcarding to indicate that any value is acceptable as long as one exists.

Attribute Checking Examples

```
<!-- Check for a particular entitlement -->
<bean class="net.shibboleth.idp.profile.logic.SimpleAttributePredicate" p:useUnfilteredAttributes="true">
        <property name="attributeValueMap">
                <map>
                        <entry key="entitlement">
                                <list>
                                        <value>urn:mace:dir:entitlement:common-lib-terms</value>
                                </list>
                        </entry>
                </map>
        </property>
</bean>

<!-- Check that an eduPersonPrincipalName exists -->
<bean class="net.shibboleth.idp.profile.logic.SimpleAttributePredicate">
        <property name="attributeValueMap">
                <map>
                        <entry key="eppn">
                                <list>
                                        <value>*</value>
                                </list>
                        </entry>
                </map>
        </property>
</bean>
```

Compound Boolean Conditions

The usual boolean operators (AND/OR/NOT) are available as predicates to allow the construction of complex compound conditions out of inividual parts. It's somewhat similar in structure to using boolean operators in the attribute filter policy language. It is of course possible to arbitrarily nest AND/OR/NOT conditions together to create complex expressions, although the (lack of) readability is such that it may be easier to use a script.

The first example demonstrates an OR operation that is equivalent to earlier examples but illustrates the syntax. The argument to the condition is a collection of condition beans to OR together.

Either of Two Relying Parties

```
<!-- An OR used to check for one of two relying parties -->
<bean id="MyCondition" parent="shibboleth.Conditions.OR">
        <constructor-arg>
                <list>
                        <bean parent="shibboleth.Conditions.RelyingPartyId" c:candidate="https://sp.example.com
/shibboleth" />
                        <bean parent="shibboleth.Conditions.RelyingPartyId" c:candidate="https://another.
example.com/shibboleth" />
                </list>
        </constructor-arg>
</bean>
```

The second example demonstrates a compound condition that checks a specific SP AND for a particular client network range.

Specific Relying Party AND Client Address Range

```
<!-- An AND checking for both an SP and a network address -->
<bean id="MyCondition" parent="shibboleth.Conditions.AND">
        <constructor-arg>
                <bean parent="shibboleth.Conditions.RelyingPartyId" c:candidate="https://sp.example.com
/shibboleth" />
        </constructor-arg>
        <constructor-arg>
                <bean class="org.opensaml.profile.logic.IPRangePredicate"
                                p:httpServletRequest-ref="shibboleth.HttpServletRequest"
                                p:ranges="192.168.1.0/24" />
        </constructor-arg>
</bean>
```

Finally, a simple NOT example checking for any SP except for one:

NOT a Specific Relying Party

```
<!-- Any SP except for one -->
<bean id="MyCondition" parent="shibboleth.Conditions.NOT">
        <constructor-arg>
                <bean parent="shibboleth.Conditions.RelyingPartyId" c:candidate="https://sp.example.com
/shibboleth" />
        </constructor-arg>
</bean>
```