## Seam Config Module

# Reference Guide

**Stuart Douglas**

# Seam Config Introduction

Seam provides a method for configuring JSR-299 beans using alternate metadata sources, such as XML configuration. (Currently, the XML provider is the only alternative available, though others are planned). Using a "type-safe" XML syntax, it's possible to add new beans, override existing beans, and add extra configuration to existing beans.

## 1.1. Getting Started

No special configuration is required, all that is required is to include the JAR file and the Seam Solder JAR in your project. For Maven projects, that means adding the following dependencies to your pom.xml:

```
<dependency>
  <groupId>org.jboss.seam.config</groupId>
  <artifactId>seam-config-xml</artifactId>
  <version>${seam.config.version}</version>
  <scope>runtime</scope>
</dependency>

<dependency>
  <groupId>org.jboss.seam.solder</groupId>
  <artifactId>seam-solder</artifactId>
  <version>${weld.extensions.version}</version>
</dependency>
```

To take advantage of Seam Config, the first thing we need is some metadata sources in the form of XML files. By default these are discovered from the classpath in the following locations:

- `/META-INF/beans.xml`
- `/META-INF/seam-beans.xml`

The `beans.xml` file is the preferred way of configuring beans via XML, however it may be possible that some JSR-299 implementations will not allow this, so `seam-beans.xml` is provided as an alternative.

Let's start with a simple example. Say we have the following class that represents a report:

```
class Report {
   String filename;
```

```
    @Inject
    Datasource datasource;

    //getters and setters
}
```

And the following support classes:

```
interface Datasource {
    public Data getData();
}

@SalesQualifier
class SalesDatasource implements Datasource {
  public Data getData()
  {
    //return sales data
  }
}

class BillingDatasource implements Datasource {
  public Data getData()
  {
    //return billing data
  }
}
```

Our `Report` bean is fairly simple. It has a filename that tells the report engine where to load the report definition from, and a datasource that provides the data used to fill the report. We are going to configure up multiple `Report` beans via xml.

### Example 1.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:s="urn:java:ee"                                    (1)
       xmlns:r="urn:java:org.example.reports">                          (2)

  <r:Report>                                                    (3)
```

```
  <s:modifies/>                                                    4
  <r:filename>sales.jrxml<r:filename>                              5
  <r:datasource>
   <r:SalesQualifier/>                                             6
  </r:datasource>
  </r:Report>

 <r:Report filename="billing.jrxml">                               7
  <s:replaces/>                                                    8
  <r:datasource>
   <s:Inject/>                                                     9
   <s:Exact>org.example.reports.BillingDatasource</s:Exact>        10
  </r:datasource>
  </r:Report>
</beans>
```

1. The namespace `urn:java:ee` is Seam Config's root namespace. This is where the built-in elements and CDI annotations live.
2. There are now multiple namespaces in the `beans.xml` file. These namespaces correspond to java package names.

   The namespace `urn:java:org.example.reports` corresponds to the package `org.example.reports`, where our reporting classes live. Multiple java packages can be aggregated into a single namespace declaration by seperating the package names with colons, e.g. `urn:java:org.example.reports:org.example.model`. The namespaces are searched in the order they are specified in the xml document, so if two packages in the namespace have a class with the same name, the first one listed will be resolved. For more information see *Namespaces*.
3. The `<Report>` declaration configures an instance of our `Report` class as a bean.
4. Beans installed using `<s:modifies>` read annotations from the existing class, and merge them with the annotations defined via xml. In addition if a bean is installed with `<s:modifies>` it prevents the original class being installed as a bean. It is also possible to add new beans and replace beans altogether, for more information see *Adding, modifying and replacing beans*.
5. The `<r:filename>` element sets the initial value of the filename field. For more information on how methods and fields are resolved see *Configuring Methods*, and *Configuring Fields*.
6. The `<r:SalesQualifier>` element applies the `@SalesQualifier` to the `datasource` field. As the field already has an `@Inject` on the class definition this will cause the `SalesDatasource` bean to be injected.
7. This is the shorthand syntax for setting a field value.

8. Beans installed using `<s:replaces>` do not read annotations from the existing class. In addition if a bean is installed with `<s:replaces>` it prevents the original class being installed as a bean.
9. The `<s:Inject>` element is needed this bean was installed with `<s:replaces>`, so annotations are not read from the class definition.
10. The `<s:Exact>` annotation restricts the type of bean that is availible for injection without using qualifiers. In this case `BillingDatasource` will be injected. This is provided as part of weld-extensions.

## 1.2. The Princess Rescue Example

The princess rescue example is a sample web app that uses Seam Config. You can run it with the following command:

```
mvn jetty:run
```

And then navigate to `http://localhost:9090/princess-rescue`. The XML configuration for the example is in `src/main/resources/META-INF/seam-beans.xml`.

# Seam Config XML provider

## 2.1. XML Namespaces

The main namespace is `urn:java:ee`. This namespace contains built-in tags and types from core packages. The built-in tags are:

- `Beans`
- `modifies`
- `replaces`
- `parameters`
- `value`
- `key`
- `entry`
- `e` (alias for entry)
- `v` (alias for value)
- `k` (alias for key)
- `array`
- `int`
- `short`
- `long`
- `byte`
- `char`
- `double`
- `float`
- `boolean`

as well as classes from the following packages:

- `java.lang`
- `java.util`
- `javax.annotation`

- `javax.inject`
- `javax.enterprise.inject`
- `javax.enterprise.context`
- `javax.enterprise.event`
- `javax.decorator`
- `javax.interceptor`
- `org.jboss.weld.extensions.core`
- `org.jboss.weld.extensions.unwraps`
- `org.jboss.weld.extensions.resourceLoader`

Other namespaces are specified using the following syntax:

```
xmlns:my="urn:java:com.mydomain.package1:com.mydomain.package2"
```

This maps the namespace `my` to the packages `com.mydomain.package1` and `com.mydomain.package2`. These packages are searched in order to resolve elements in this namespace.

For example, say you had a class `com.mydomain.package2.Report`. To configure a `Report` bean you would use `<my:Report>`. Methods and fields on the bean are resolved from the same namespace as the bean itself. It is possible to distinguish between overloaded methods by specifying the parameter types, for more information see *Configuring Methods*.

## 2.2. Adding, replacing and modifying beans

By default configuring a bean via XML creates a new bean, however there may be cases where you want to modify an existing bean rather than adding a new one. The `<s:replaces>` and `<s:modifies>` tags allow you to do this.

The `<s:replaces>` tag prevents the existing bean from being installed, and registers a new one with the given configuration. The `<s:modifies>` tag does the same, except that it merges the annotations on the bean with the annotations defined in XML. Where the same annotation is specified on both the class and in XML the annotation in XML takes precidence. This has almost the same effect as modifiying an existing bean, except it is possible to install multiple beans that modify the same class.

```
<my:Report>
   <s:modifies>
```

```
    <my:NewQualifier/>
</my:Report>

<my:ReportDatasource>
    <s:replaces>
    <my:NewQualifier/>
</my:ReportDatasource>
```

The first entry above adds a new bean with an extra qualifier, in addition to the qualifiers already present, and prevents the existing `Report` bean from being installed.

The second prevents the existing bean from being installed, and registers a new bean with a single qualifier.

## 2.3. Applying annotations using XML

Annotations are resolved in the same way as normal classes. Conceptually annotations are applied to the object their parent element resolves to. It is possible to set the value of annotation members using the xml attribute that corresponds to the member name. For example:

```
public @interface OtherQualifier {
   String value1();
   int value2();
   QualifierEnum value();
}
```

```
<test:QualifiedBean1>
      <test:OtherQualifier value1="AA" value2="1">A</my:OtherQualifier>
</my:QualifiedBean1>

<test:QualifiedBean2>
      <test:OtherQualifier value1="BB" value2="2" value="B" />
</my:QualifiedBean2>
```

The `value` member can be set using the inner text of the node, as seen in the first example. Type conversion is performed automatically.

**Note**

It is currently not possible set array annotation members.

## 2.4. Configuring Fields

It is possible to both apply qualifiers to and set the initial value of a field. Fields reside in the same namespace as the declaring bean, and the element name must exactly match the field name. For example if we have the following class:

```
class RobotFactory {
  Robot robot;
}
```

The following xml will add the `@Produces` annotation to the `robot` field:

```
<my:RobotFactory>
  <my:robot>
    <s:Produces/>
  </my:robot>
</my:RobotFactory/>
```

### 2.4.1. Initial Field Values

Inital field values can be set three different ways as shown below:

```
<r:MyBean company="Red Hat Inc" />

<r:MyBean>
  <r:company>Red Hat Inc</r:company>
</r:MyBean>

<r:MyBean>
  <r:company>
    <s:value>Red Hat Inc<s:value>
    <r:SomeQualifier/>
  </r:company>
</r:MyBean>
```

The third form is the only one that also allows you to add annotations such as qualifiers to the field.

It is possible to set `Map`,`Array` and `Collection` field values. Some examples:

```
<my:ArrayFieldValue>
```

```
    <my:intArrayField>
      <s:value>1</s:value>
      <s:value>2</s:value>
    </my:intArrayField>

    <my:classArrayField>
      <s:value>java.lang.Integer</s:value>
      <s:value>java.lang.Long</s:value>
    </my:classArrayField>

    <my:stringArrayField>
      <s:value>hello</s:value>
      <s:value>world</s:value>
    </my:stringArrayField>

</my:ArrayFieldValue>

<my:MapFieldValue>

    <my:map1>
      <s:entry><s:key>1</s:key><s:value>hello</s:value></s:entry>
      <s:entry><s:key>2</s:key><s:value>world</s:value></s:entry>
    </my:map1>

    <my:map2>
      <s:e><s:k>1</s:k><s:v>java.lang.Integer</s:v></s:e>
      <s:e><s:k>2</s:k><s:v>java.lang.Long</s:v></s:e>
    </my:map2>

</my:MapFieldValue>
```

Type conversion is done automatically for all primitives and primitive wrappers, `Date`, `Calendar`,`Enum` and `Class` fields.

The use of EL to set field values is also supported:

```
<m:Report>
  <m:name>#{reportName}</m:name>
  <m:parameters>
    <s:key>#{paramName}</s:key>
    <s:value>#{paramValue}</s:key>
  </m:parameters>
```

```
</m:Report>
```

Internally field values are set by wrapping the `InjectionTarget` for a bean. This means that the expressions are evaluated once, at bean creation time.

### 2.4.2. Inline Bean Declarations

Inline beans allow you to set field values to another bean that is declared inline inside the field declaration. This allows for the configuration of complex types with nestled classes. Inline beans can be declared inside both `<s:value>` and `<s:key>` elements, and may be used in both collections and simple field values. Inline beans must not have any qualifier annotations declared on the bean, instead Seam Config assigns them an artificial qualifier. Inline beans may have any scope, however the default `Dependent` scope is recommended.

```
<my:Knight>
  <my:sword>
    <value>
      <my:Sword type="sharp"/>
    </value>
  </my:sword>
  <my:horse>
    <value>
      <my:Horse>
        <my:name>
          <value>billy</value>
        </my:name>
        <my:shoe>
          <Inject/>
        </my:shoe>
      </my:Horse>
    </value>
  </my:horse>
</my:Knight>
```

## 2.5. Configuring methods

It is also possible to configure methods in a similar way to configuring fields:

```
class MethodBean {

  public int doStuff() {
    return 1;
```

```
  }

  public int doStuff(MethodValueBean bean) {
    return bean.value + 1;
  }

  public void doStuff(MethodValueBean[][] beans) {
    /*do stuff */
  }

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:s="urn:java:ee"
      xmlns:my="urn:java:org.jboss.seam.config.xml.test.method">
   <my:MethodBean>

      <my:doStuff>
         <s:Produces/>
      </my:doStuff>

      <my:doStuff>
         <s:Produces/>
         <my:Qualifier1/>
         <s:parameters>
            <my:MethodValueBean>
               <my:Qualifier2/>
            </my:MethodValueBean>
         </s:parameters>
      </my:doStuff>

      <my:doStuff>
         <s:Produces/>
         <my:Qualifier1/>
         <s:parameters>
            <s:array dimensions="2">
               <my:Qualifier2/>
               <my:MethodValueBean/>
            </s:array>
         </s:parameters>
      </my:doStuff>
```

```
   </my:MethodBean>
</beans>
```

In this instance `MethodBean` has three methods, all of them rather imaginatively named `doStuff`.

The first `<test:doStuff>` entry in the XML file configures the method that takes no arguments. The `<s:Produces>` element makes it into a producer method.

The next entry in the file configures the method that takes a `MethodValueBean` as a parameter and the final entry configures a method that takes a two dimensional array of`MethodValueBean`'s as a parameter. For both these methods a qualifier was added to the method parameter and they were made into producer methods.

Method parameters are specified inside the `<s:parameters>` element. If these parameters have annotation children they are taken to be annotations on the parameter.

The corresponding Java declaration for the XML above would be:

```
class MethodBean {

   @Produces
   public int doStuff() {/*method body */}

   @Produces
   @Qualifier1
   public int doStuff(@Qualifier2 MethodValueBean param) {/*method body */}

   @Produces
   @Qualifier1
   public int doStuff(@Qualifier2 MethodValueBean[][] param) {/*method body */}
}
```

Array parameters can be represented using the `<s:array>` element, with a child element to represent the type of the array. E.g. `int method(MethodValueBean[] param);` could be configured via xml using the following:

```
<my:method>
   <s:array>
     <my:MethodValueBean/>
   </s:array>
</my:method>
```

**Note**

If a class has a field and a method of the same name then by default the field will be resolved, unless the element has a child `<parameters>` element, in which case it is resolved as a method.

## 2.6. Configuring the bean constructor

It is also possible to configure the bean constructor in a similar manner. This is done with a `<s:parameters>` element directly on the bean element. The constructor is resolved in the same way methods are resolved. This constructor will automatically have the `@Inject` annotation applied to it. Annotations can be applied to the constructor parameters in the same manner as method parameters.

```
<my:MyBean>
  <s:parameters>
      <s:Integer>
        <my:MyQualifier/>
      </s:Integer>
  </s:parameters>
</my:MyBean>
```

The example above is equivalent to the following java:

```
class MyBean {
  @Inject
  MyBean(@MyQualifier Integer count)
  {
    ...
  }
}
```

## 2.7. Overriding the type of an injection point

It is possible to limit which bean types are availible to inject into a given injection point:

```
class SomeBean
{
   public Object someField;
```

```
}
```

```
<my:SomeBean>
      <my:someField>
         <s:Inject/>
         <s:Exact>com.mydomain.InjectedBean</s:Exact>
      </my:someField>
</my:SomeBean>
```

In the example above only beans that are assignable to InjectedBean will be eligable for injection into the field. This also works for parameter injection points. This functionallity is part of Seam Solder, and the `@Exact` annotation can be used directly in java.

## 2.8. Configuring Meta Annotations

It is possible to make existing annotations into qualifiers, stereotypes or interceptor bindings.

This configures a stereotype annotation `SomeStereotype` that has a single interceptor binding and is named:

```
<my:SomeStereotype>
  <s:Stereotype/>
  <my:InterceptorBinding/>
  <s:Named/>
</my:SomeStereotype>
```

This configures a qualifier annotation:

```
<my:SomeQualifier>
  <s:Qualifier/>
</my:SomeQualifier>
```

This configures an interceptor binding:

```
<my:SomeInterceptorBinding>
  <s:InterceptorBinding/>
</my:SomeInterceptorBinding>
```

## 2.9. Virtual Producer Fields

Seam XML supports configuration of virtual producer fields. These allow for configuration of resource producer fields, Weld Extensions generic bean and constant values directly via XML. First an example:

```
<s:EntityManager>
  <s:Produces/>
  <sPersistenceContext unitName="customerPu" />
</s:EntityManager>

<s:String>
  <s:Produces/>
  <my:VersionQualifier />
  <value>Version 1.23</value>
</s:String>
```

The first example configures a resource producer field. The second configures a bean of type String, with the qualifier `@VersionQualifier` and the value `'Version 1.23'`. The corresponding java for the above XML is:

```
class SomeClass
{

  @Produces
  @PersistenceContext(unitName="customerPu")
  EntityManager field1;

  @Produces
  @VersionQualifier
  String field2 = "Version 1.23";

}
```

Although these look superficially like normal bean declarations, the `<Produces>` declaration means it is treated as a producer field instead of a normal bean.

## 2.10. Notes on Configuring Interceptors

Some versions of weld including `1.1.0.Final` do not support adding the `@AroundInvoke` annotation via the SPI, this will be fixed in future versions.

## 2.11. More Information

For further information look at the units tests in the Seam Config distribution, also the JSR-299 Public Review Draft section on XML Configuration was the base for this extension, so it may also be worthwhile reading.