Seam XML Configuration

Seam XML General

Seam provides a method for configuring JSR-299 beans using XML. Using XML it is possible to add new beans, override existing beans, and add extra configuration to existing beans. The default is to add a new bean.

1.1. Configuration

No special configuration is required to use seam-xml, all that is required is to include the jar file and the weld extensions jar in your deployment.

1.2. Getting Started

By default XML files are discovered from the classpath. The extension looks for an XML file in the following locations:

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

```
<?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:seam:core"
       xmlns:test="urn:java:org.jboss.seam.xml.test.injection">

    <test:ProducerQualifier>
        <s:Qualifier/>
    </test:ProducerQualifier>

    <test:ProducerBean>
        <test:value>
            <s:Produces/>
            <test:ProducerQualifier/>
            <s:value>hello world</s:value>
        </test:value>
    </test:ProducerBean>

   <test:ReceiverBean>
        <test:value>
            <test:ProducerQualifier/>
            <s:Inject/>
        </test:value>
   </test:ReceiverBean>
```

```
</beans>
```

You will notice that two new namspace declarations have been added to the `beans.xml` file: `urn:java:seam:core` and `urn:java:org.jboss.seam.xml.test.injection`. The `urn:java:seam:core` namespace is the main one used by the XML extension, we will cover exactly what lives in this namespace later. The `urn:java:org.jboss.seam.xml.test.injection` namespace is used to resolve classes in the java package `org.jboss.seam.xml.test.injection`, so in the example above `<test:ProducerBean>` resolves to `org.jboss.seam.xml.test.injection.ProducerBean`.

```
<test:ProducerQualifier>
    <s:Qualifier/>
</test:ProducerQualifier>
```

The first entry in the file defines a new qualifier. `ProducerQualifier` is an annotation in the package `org.jboss.seam.xml.test.injection`.

```
<test:ProducerBean>
    <test:value>
        <s:Produces/>
        <test:ProducerQualifier/>
        <s:value>hello world</s:value>
    </test:value>
</test:ProducerBean>
```

The next entry in the file is a bean declaration. The bean class is `org.jboss.seam.xml.test.injection.ProducerBean`. It is important to note that this declaration does not change the existing declaration of `ProducerBean`, instead it installs a new bean. In this instance there will be two `ProducerBean` CDI beans.

This bean has a field called `value`, this field is configured to be a producer field using XML (it is also possible to configure producer methods, more on this later). The `<test:value/>` declaration has several child elements. The `<s:Produces/>` element tells the container that this is a producer field. `<test:ProducerQualifier/>` element defines a qualifier for the producer field. The `<s:value>` element defines an initial value for the field.

Child elements of fields, methods and classes that resolve to Annotation types are considered to be annotations on the corresponding element, so the corresponding Java declaration for the XML above would be:

```
public class ProducerBean {
   @Produces
```

```
   @ProducerQualifier
   public String value = "hello world";
}
```

```
<test:ReceiverBean>
    <test:value>
         <test:ProducerQualifier/>
         <s:Inject/>
    </test:value>
</test:ReceiverBean>
```

The XML above declares a new bean that injects the value that was produced above. In this case the `@Inject` annotation is applied instead of `@Produces` and no initial value is set.

1.3. The main namespace

The main namesapce is `urn:java:seam:core` can contain the following elements:

- `Beans`
- `extends`
- `override`
- `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`

So the `<s:Produces>` element above actually resolves to `java.enterprise.inject.Produces` and the `<s:Inject>` element resolves to `javax.inject.Inject`.

1.4. Overriding and extending beans

There may be cases where you want to modify an existing bean rather than adding a new one. The `<s:override>` and `<s:extends>` tags allow you to do this. The `<s:override>` tag prevents the existing bean from being installed, and registers a new one with the given configuration. The `<s:extends>` tag allows you to add extra configuration to an existing bean.

```
<test:MyBean>
 <s:extends>
 <test:NewQualifier/>
</test:MyBean>

<test:OtherBean>
 <s:override>
 <test:NewQualifier/>
</test:OtherBean>
```

The first entry above adds a new qualifier to an existing bean definition. The second prevents the existing bean from being installed, and registers a new bean with a single qualifier.

1.5. Initial Field Values

Inital field values can be set in two different ways, in addition to the `<s:value>` element shown above it can be set as follows:

```
<test:someField>hello world</test:someField>
```

Using this method prevents you from adding any annotations to the field.

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

```
<test:ArrayFieldValue>
    <test:iarray>
        <s:value>1</s:value>
        <s:value>2</s:value>
    </test:iarray>
    <test:carray>
        <s:value>java.lang.Integer</s:value>
        <s:value>java.lang.Long</s:value>
    </test:carray>
    <test:sarray>
        <s:value>hello</s:value>
        <s:value>world</s:value>
    </test:sarray>
</test:ArrayFieldValue>

<test:MapFieldValue>
    <test: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>
    </test:map1>
    <test: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>
    </test:map2>
</test:MapFieldValue>
```

Type conversion is done automatically for all primitives and primitive wrappers, `Date`, `Calendar`,`Enum` and `Class` fields. In this instance `ArrayFieldValue.carray` is actually an array of classes, not an array of Strings.

1.6. Configuring methods

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

```
<?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:seam:core"
       xmlns:test="urn:java:org.jboss.seam.xml.test.method">
    <test:MethodBean>
        <test:method>
            <s:Produces/>
        </test:method>
        <test:method>
            <s:Produces/>
            <test:Qualifier1/>
```

```
            <s:parameters>
                <test:MethodValueBean>
                    <test:Qualifier2/>
                </test:MethodValueBean>
            </s:parameters>
        </test:method>
         <test:method>
            <s:Produces/>
            <test:Qualifier1/>
            <s:parameters>
                <s:array dimensions="2">
                    <test:Qualifier2/>
                    <test:MethodValueBean/>
                </s:array>
            </s:parameters>
        </test:method>
    </test:MethodBean>
</beans>

public class MethodBean {

   public int method() {
      return 1;
   }

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

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

}
```

In this instance `MethodBean` has three methods, all of them rather imaginatively named `method`. The first `<test:method>` 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. The final entry configures a method that takes a two dimensional array of`MethodValueBean`'s as a parameter. 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:

```
@Produces
public int method() {//method body}

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

```
@Produces
@Qualifier1
public int method(@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:

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

1.7. Annotation Members

It is also possible to set the value of annotation members. For example:

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


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

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

The value member can be set using the inner text of the node, as seen in the first example.

1.8. More Information

For further information look at the units tests in the seam-xml 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.