Seam Drools

Seam Drools

Seam provides a convenient method of remotely accessing CDI beans from a web page, using AJAX (Asynchronous Javascript and XML). The framework for this functionality is provided with almost no up-front development effort - your beans only require simple annotating to become accessible via AJAX. This chapter describes the steps required to build an AJAX-enabled web page, then goes on to explain the features of the Seam Remoting framework in more detail.

1.1. Configuration

To use remoting, the Seam Remoting servlet must first be configured in your `web.xml` file:

```
<servlet>
   <servlet-name>Remoting Servlet</servlet-name>
   <servlet-class>org.jboss.seam.remoting.Remoting</servlet-class>
   <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
   <servlet-name>Remoting Servlet</servlet-name>
   <url-pattern>/seam/resource/remoting/*</url-pattern>
</servlet-mapping>
```

The next step is to import the necessary Javascript into your web page. There are a minimum of two scripts that must be imported. The first one contains all the client-side framework code that enables remoting functionality:

```
<script type="text/javascript" src="seam/resource/remoting/resource/remote.js"></script>
```

For a production environment, you may wish to use a compressed version of `remote.js`. To do this, simply add the `compress=true` parameter to the end of the url:

```
<script        type="text/javascript"        src="seam/resource/remoting/resource/remote.js?
compress=true"></script>
```

The compressed version has its white space compacted and JavaScript comments removed. For development and debugging purposes it is recommended that you use the non-compacted version.

The second script that you need contains the stubs and type definitions for the beans you wish to call. It is generated dynamically based on the method signatures of your beans, and includes type definitions for all of the classes that can be used to call its remotable methods. The name of the script reflects the name of your bean. For example, if you have a named bean annotated with `@Named`, then your script tag should look like this (for a bean class called `CustomerAction`):

```
<script type="text/javascript"
         src="seam/resource/remoting/interface.js?customerAction"></script>
```

Otherwise, you can simply specify the fully qualified class name of the bean:

```
<script type="text/javascript"
          src="seam/resource/remoting/interface.js?com.acme.myapp.CustomerAction"></script>
```

If you wish to access more than one bean from the same page, then include them all as parameters of your script tag:

```
<script type="text/javascript"
        src="seam/resource/remoting/interface.js?customerAction&accountAction"></script>
```

1.1.1. Dynamic type loading

If you forget to import a bean or other class that is required by your bean, don't worry. Seam Remoting has a dynamic type loading feature that automatically loads any JavaScript stubs for bean types that it doesn't recognize.

1.2. The "Seam" object

Client-side interaction with your beans is all performed via the `Seam` Javascript object. This object is defined in `remote.js`, and you'll be using it to make asynchronous calls against your bean. It contains methods for creating client-side bean objects and also methods for executing remote requests. The easiest way to become familiar with this object is to start with a simple example.

1.2.1. A Hello World example

Let's step through a simple example to see how the `Seam` object works. First of all, let's create a new bean called `helloAction`:

```
@Named
public class HelloAction implements HelloLocal {
    @WebRemote public String sayHello(String name) {
        return "Hello, " + name;
    }
}
```

Take note of the `@WebRemote` annotation on the `sayHello()` method in the above listing. This annotation makes the method accessible via the Remoting API. Besides this annotation, there's nothing else required on your bean to enable it for remoting.

Now for our web page - create a new JSF page and import the `helloAction` bean:

```
<script type="text/javascript"
        src="seam/resource/remoting/interface.js?helloAction
```

To make this a fully interactive user experience, let's add a button to our page:

```
<button onclick="javascript:sayHello()">Say Hello</button>
```

We'll also need to add some more script to make our button actually do something when it's clicked:

```
<script type="text/javascript">
  //<![CDATA[

  function sayHello() {
    var name = prompt("What is your name?");
    Seam.createBean("helloAction").sayHello(name, sayHelloCallback);
  }

  function sayHelloCallback(result) {
    alert(result);
  }

   // ]]>
</script>
```

We're done! Deploy your application and open the page in a web browser. Click the button, and enter a name when prompted. A message box will display the hello message confirming that the call was successful. If you want to save some time, you'll find the full source code for this Hello World example in the `/examples/helloworld` directory.

So what does the code of our script actually do? Let's break it down into smaller pieces. To start with, you can see from the Javascript code listing that we have implemented two methods - the first method is responsible for prompting the user for their name and then making a remote request. Take a look at the following line:

```
Seam.createBean("helloAction").sayHello(name, sayHelloCallback);
```

The first section of this line, `Seam.createBean("helloAction")` returns a proxy, or "stub" for our `helloAction` bean. We can invoke the methods of our bean against this stub, which is exactly what happens with the remainder of the line: `sayHello(name, sayHelloCallback);`.

What this line of code in its completeness does, is invoke the `sayHello` method of our bean, passing in `name` as a parameter. The second parameter, `sayHelloCallback` isn't a parameter of our bean's `sayHello` method, instead it tells the Seam Remoting framework that once it receives the response to our request, it should pass it to the `sayHelloCallback` Javascript method. This callback parameter is entirely optional, so feel free to leave it out if you're calling a method with a `void` return type or if you don't care about the result.

The `sayHelloCallback` method, once receiving the response to our remote request then pops up an alert message displaying the result of our method call.

1.2.2. Seam.createBean

The `Seam.createBean` JavaScript method is used to create client-side instances of both action and "state" beans. For action beans (which are those that contain one or more methods annotated with `@WebRemote`), the stub object provides all of the remotable methods exposed by the bean. For "state" beans (i.e. beans that simply carry state, for example Entity beans) the stub object provides all the same accessible properties as its server-side equivalent. Each property also has a corresponding getter/setter method so you can work with the object in JavaScript in much the same way as you would in Java.

1.3. The Context

The Seam Remoting Context contains additional information which is sent and received as part of a remoting request/response cycle. It currently contains the conversation ID and Call ID, and may be expanded to include other properties in the future.

1.3.1. Setting and reading the Conversation ID

If you intend on using remote calls within the scope of a conversation then you need to be able to read or set the conversation ID in the Seam Remoting Context. To read the conversation ID after making a remote request call `Seam.context.getConversationId()`. To set the conversation ID before making a request, call `Seam.context.setConversationId()`.

If the conversation ID hasn't been explicitly set with `Seam.context.setConversationId()`, then it will be automatically assigned the first valid conversation ID that is returned by any remoting call. If you are working with multiple conversations within your page, then you may need to explicitly set the conversation ID before each call. If you are working with just a single conversation, then you don't need to do anything special.

1.3.2. Remote calls within the current conversation scope

In some circumstances it may be required to make a remote call within the scope of the current view's conversation. To do this, you must explicitly set the conversation ID to that of the view before making the remote call. This small snippet of JavaScript will set the conversation ID that is used for remoting calls to the current view's conversation ID:

```
Seam.context.setConversationId( #{conversation.id} );
```

1.4. Working with Data types

1.4.1. Primitives / Basic Types

This section describes the support for basic data types. On the server side these values as a rule are compatible with either their primitive type or their corresponding wrapper class.

1.4.1.1. String

Simply use Javascript String objects when setting String parameter values.

1.4.1.2. Number

There is support for all number types supported by Java. On the client side, number values are always serialized as their String representation and then on the server side they are converted to the correct destination type. Conversion into either a primitive or wrapper type is supported for `Byte`, `Double`, `Float`, `Integer`, `Long` and `Short` types.

1.4.1.3. Boolean

Booleans are represented client side by Javascript Boolean values, and server side by a Java boolean.

1.4.2. JavaBeans

In general these will be either entity beans or JavaBean classes, or some other non-bean class. Use `Seam.createBean()` to create a new instance of the object.

1.4.3. Dates and Times

Date values are serialized into a String representation that is accurate to the millisecond. On the client side, use a JavaScript `Date` object to work with date values. On the server side, use any `java.util.Date` (or descendent, such as `java.sql.Date` or `java.sql.Timestamp` class.

1.4.4. Enums

On the client side, enums are treated the same as `String`s. When setting the value for an enum parameter, simply use the `String` representation of the enum. Take the following bean as an example:

```
@Named
public class paintAction {
  public enum Color {red, green, blue, yellow, orange, purple};

  public void paint(Color color) {
    // code
  }
}
```

To call the `paint()` method with the color `red`, pass the parameter value as a `String` literal:

```
Seam.createBean("paintAction").paint("red");
```

The inverse is also true - that is, if a bean method returns an enum parameter (or contains an enum field anywhere in the returned object graph) then on the client-side it will be converted to a `String`.

1.4.5. Collections

1.4.5.1. Bags

Bags cover all collection types including arrays, collections, lists, sets, (but excluding Maps - see the next section for those), and are implemented client-side as a JavaScript array. When calling a bean method that accepts one of these types as a parameter, your parameter should be a JavaScript array. If a bean method returns one of these types, then the return value will also be a JavaScript array. The remoting framework is clever enough on the server side to convert the bag to an appropriate type (including sophisticated support for generics) for the bean method call.

1.4.5.2. Maps

As there is no native support for Maps within JavaScript, a simple Map implementation is provided with the Seam Remoting framework. To create a Map which can be used as a parameter to a remote call, create a new `Seam.Map` object:

```
var map = new Seam.Map();
```

This JavaScript implementation provides basic methods for working with Maps: `size()`, `isEmpty()`, `keySet()`, `values()`, `get(key)`, `put(key, value)`, `remove(key)` and `contains(key)`. Each of these methods are equivalent to their Java counterpart. Where the method returns a collection, such as `keySet()` and `values()`, a JavaScript Array object will be returned that contains the key or value objects (respectively).

1.5. Debugging

To aid in tracking down bugs, it is possible to enable a debug mode which will display the contents of all the packets send back and forth between the client and server in a popup window. To enable debug mode, set the `Seam.debug` property to `true` in Javascript:

```
Seam.debug = true;
```

If you want to write your own messages to the debug log, call `Seam.log(message)`.

1.6. Handling Exceptions

When invoking a remote bean method, it is possible to specify an exception handler which will process the response in the event of an exception during bean invocation. To specify an exception handler function, include a reference to it after the callback parameter in your JavaScript:

```
var callback = function(result) { alert(result); };
var exceptionHandler = function(ex) { alert("An exception occurred: " + ex.getMessage()); };
Seam.createBean("helloAction").sayHello(name, callback, exceptionHandler);
```

If you do not have a callback handler defined, you must specify `null` in its place:

```
var exceptionHandler = function(ex) { alert("An exception occurred: " + ex.getMessage()); };
Seam.createBean("helloAction").sayHello(name, null, exceptionHandler);
```

The exception object that is passed to the exception handler exposes one method, `getMessage()` that returns the exception message which is produced by the exception thrown by the `@WebRemote` method.

1.7. The Loading Message

The default loading message that appears in the top right corner of the screen can be modified, its rendering customised or even turned off completely.

1.7.1. Changing the message

To change the message from the default "Please Wait..." to something different, set the value of `Seam.loadingMessage`:

```
Seam.loadingMessage = "Loading...";
```

1.7.2. Hiding the loading message

To completely suppress the display of the loading message, override the implementation of `displayLoadingMessage()` and `hideLoadingMessage()` with functions that instead do nothing:

```
// don't display the loading indicator
Seam.displayLoadingMessage = function() {};
Seam.hideLoadingMessage = function() {};
```

1.7.3. A Custom Loading Indicator

It is also possible to override the loading indicator to display an animated icon, or anything else that you want. To do this override the `displayLoadingMessage()` and `hideLoadingMessage()` messages with your own implementation:

```
  Seam.displayLoadingMessage = function() {
    // Write code here to display the indicator
  };

  Seam.hideLoadingMessage = function() {
    // Write code here to hide the indicator
  };
```

1.8. Controlling what data is returned

When a remote method is executed, the result is serialized into an XML response that is returned to the client. This response is then unmarshaled by the client into a JavaScript object. For complex types (i.e. Javabeans) that include references to other objects, all of these referenced objects are also serialized as part of the response. These objects may reference other objects, which may reference other objects, and so forth. If left unchecked, this object "graph" could potentially be enormous, depending on what relationships exist between your objects. And as a side issue (besides the potential verbosity of the response), you might also wish to prevent sensitive information from being exposed to the client.

Seam Remoting provides a simple means to "constrain" the object graph, by specifying the `exclude` field of the remote method's `@WebRemote` annotation. This field accepts a String array containing one or more paths specified using dot notation. When invoking a remote method, the objects in the result's object graph that match these paths are excluded from the serialized result packet.

For all our examples, we'll use the following `Widget` class:

```
public class Widget
{
  private String value;
  private String secret;
  private Widget child;
  private Map<String,Widget> widgetMap;
  private List<Widget> widgetList;

  // getters and setters for all fields
}
```

1.8.1. Constraining normal fields

If your remote method returns an instance of `Widget`, but you don't want to expose the `secret` field because it contains sensitive information, you would constrain it like this:

```
@WebRemote(exclude = {"secret"})
public Widget getWidget();
```

The value "secret" refers to the `secret` field of the returned object. Now, suppose that we don't care about exposing this particular field to the client. Instead, notice that the `Widget` value that is returned has a field `child` that is also a `Widget`. What if we want to hide the `child`'s `secret` value instead? We can do this by using dot notation to specify this field's path within the result's object graph:

```
@WebRemote(exclude = {"child.secret"})
public Widget getWidget();
```

1.8.2. Constraining Maps and Collections

The other place that objects can exist within an object graph are within a `Map` or some kind of collection (`List`, `Set`, `Array`, etc). Collections are easy, and are treated like any other field. For example, if our `Widget` contained a list of other `Widgets` in its `widgetList` field, to constrain the `secret` field of the `Widgets` in this list the annotation would look like this:

```
@WebRemote(exclude = {"widgetList.secret"})
public Widget getWidget();
```

To constrain a `Map`'s key or value, the notation is slightly different. Appending `[key]` after the `Map`'s field name will constrain the `Map`'s key object values, while `[value]` will constrain the value object values. The following example demonstrates how the values of the `widgetMap` field have their `secret` field constrained:

```
@WebRemote(exclude = {"widgetMap[value].secret"})
public Widget getWidget();
```

1.8.3. Constraining objects of a specific type

There is one last notation that can be used to constrain the fields of a type of object no matter where in the result's object graph it appears. This notation uses either the name of the bean (if the object is a named bean) or the fully qualified class name (only if the object is not a named bean) and is expressed using square brackets:

```
@WebRemote(exclude = {"[widget].secret"})
public Widget getWidget();
```

1.8.4. Combining Constraints

Constraints can also be combined, to filter objects from multiple paths within the object graph:

```
@WebRemote(exclude = {"widgetList.secret", "widgetMap[value].secret"})
public Widget getWidget();
```