Seam Remoting

Seam Remoting - Basic Features

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>
```

Note

If your application is running within a Servlet 3.0 (or greater) environment, then the servlet configuration listed above is not necessary as the Seam Remoting JAR library bundles a `web-fragment.xml` that configures the Remoting servlet automatically.

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>
```

By default, the client-side JavaScript is served in compressed form, with white space compacted and JavaScript comments removed. For a development environment, you may wish to use the uncompressed version of `remote.js` for debugging and testing purposes. To do this, simply add the `compress=false` parameter to the end of the url:

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

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.

Note

If you are performing a persistence operation in the method marked `@WebRemote` you will also need to add a `@Transactional` annotation to the method. Otherwise, your method would execute outside of a transaction without this extra hint.That's because unlike a JSF request, Seam does not wrap the remoting request in a transaction automatically.

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. Messages

The Seam International module provides a Messages API that allows generation of view-independent messages. This is useful if you want to convey additional information to a user that is not returned directly from the result of a method invocation.

Using the Messages API is extremely easy. Simply add the Seam International libraries to your application (see the Seam International configuration chapter to learn how to do this), then inject the `Messages` object into your bean. The `Messages` object provides several methods for adding messages, see the Seam International documentation for more information. Here's a simple example showing how to create an `info` message (messages generally follow the same DEBUG, INFO, WARN, ERROR levels that a typical logging framework would provide):

```
import javax.inject.Inject;
```

```
import org.jboss.seam.international.status.Messages;
import org.jboss.seam.remoting.annotations.WebRemote;

public class HelloAction {
   @Inject Messages messages;

   @WebRemote
   public String sayHello(String name) {
      messages.info("Invoked HelloAction.sayHello()");
      return "Hello, " + name;
   }
}
```

After creating the message in your server-side code, you still need to write some client-side code to handle any messages that are returned by your remote invocations. Thankfully this is also simple, you just need to write a JavaScript handler function and assign it to `Seam.messageHandler`.

If any messages are returned from a remote method invocation, the message handler function will be invoked and passed a list of Message objects. These objects declare three methods for retrieving various properties of the message - `getLevel()` returns the message level (such as DEBUG, INFO, etc). The `getTargets()` method returns the targets of the message - these may be the ID's for specific user interface controls, which is helpful for conveying validation failures for certain field values. The `getTargets()` method may return null, if the message is not specific to any field value. Lastly, the `getText()` method returns the actual text of the message.

Here's a really simple example showing how you would display an alert box for any messages returned:

```
function handleMessages(msgs) {
  for (var i = 0; i < msgs.length; i++) {
    alert("Received message - Level: " + msgs[i].getLevel() + " Text: " + msgs[i].getText();
  }
}

Seam.messageHandler = handleMessages;
```

You can see the Messages API in action in the HelloWorld example. Simply choose the "Formal" option for the Formality, and "Localized (English)" for the Localization. Invoking this combination will cause a server-side message to be created, which you will then see in the Messages list at the top of the screen.

1.7. 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 two methods, `getExceptionClass()` which returns the name of the exception class that was thrown, and `getMessage()`, which returns the exception message which is produced by the exception thrown by the `@WebRemote` method.

It is also possible to register a global exception handler, which will be invoked if there is no exception handler defined for an individual invocation. By default, the global exception handler will display an alert message notifying the user that there was an exception - here's what the default exception handler looks like:

```
Seam.defaultExceptionHandler = function(exception) {
      alert("An exception has occurred while executing a remote request: " +
 exception.getExceptionClass() + ":" + exception.getMessage());
};
```

If you would like to provide your own global exception handler, then simply override the value of `Seam.exceptionHandler` with your own custom exception handler, as in the following example:

```
 function customExceptionHandler(exception) {
     alert("Uh oh, something bad has happened! [" + exception.getExceptionClass() + ":" +
 exception.getMessage() + "]");
 }

 Seam.exceptionHandler = customExceptionHandler;
```

1.8. 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.

1.8.1. Changing the message

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

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

1.8.2. 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() {};
```

1.8.3. 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
};
```

1.9. 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
}
```

1.9.1. 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();
```

1.9.2. 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();
```

1.9.3. 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();
```

1.9.4. 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();
```

Seam Remoting - Bean Validation

Seam Remoting provides integrated support for JSR-303 Bean Validation, which defines a standard approach for validating Java Beans no matter where they are used; web tier or persistence tier, server or client. Bean validation for remoting delivers JSR-303's vision by making all of the validation constraints declared by the server-side beans available on the client side, and allows developers to perform client-side bean validation in an easy to use, consistent fashion.

Client-side validation by its very nature is an asynchronous operation, as it is possible that the client may encounter a custom validation constraint for which it has no knowledge of the corresponding validation logic. Under these circumstances, the client will make a request to the server for the validation to be performed server-side, after which it receives the result will forward it to the client-side callback method. All built-in validation types defined by the JSR-303 specification are executed client-side without requiring a round-trip to the server. It is also possible to provide the client-side validation API with custom JavaScript to allow client-side execution of custom validations.

2.1. Validating a single object

The `Seam.validateBean()` method may be used to validate a single object. It accepts the following parameter values:

```
Seam.validateBean(bean, callback, groups);
```

The `bean` parameter is the object to validate.

The `callback` parameter should contain a reference to the callback method to invoke once validation is complete.

The `groups` parameter is optional, however may be specified if only certain validation groups should be validated. The `groups` parameter may be a `String` or an array of `String` values for when multiple groups are to be validated.

Here's an example showing how a bean called `customer` is validated:

```
function test() {
  var customer = Seam.createBean("com.acme.model.Customer");
  customer.setFirstName("John");
  customer.setLastName("Smith");
  Seam.validateBean(customer, validationCallback);
}

function validationCallback(violations) {
```

```
  if (violations.length == 0) alert("All validations passed!");
}
```

Tip

By default, when Seam Remoting performs validation for a single bean it will traverse the entire object graph for that bean and validate each unique object that it finds. If you don't wish to validate the entire object graph, then please refer to the section on validating multiple objects later in this chapter for an alternative.

2.2. Validating a single property

Sometimes it might not be desirable to perform validation for all properties of a bean. For example, you might have a dynamic form which displays validation errors as the user tabs between fields. In this situation, you may use the `Seam.validateProperty()` method to validate a single bean property.

```
Seam.validateProperty(bean, property, callback, groups)
```

The `bean` parameter is the object containing the property that is to be validated.

The `property` parameter is the name of the property to validate.

The `callback` parameter is a reference to the callback function to invoke once the property has been validated.

The `groups` parameter is optional, however may be specified if validating the property against a certain validation group. The `groups` parameter may be a `String` or an array of `String` values for multiple groups.

Here's an example showing how to validate the `firstName` property of a bean called `customer`:

```
function test() {
  var customer = Seam.createBean("com.acme.model.Customer");
  customer.setFirstName("John");
  Seam.validateProperty(customer, "firstName", validationCallback);
}

function validationCallback(violations) {
  if (violations.length == 0) alert("All validations passed!");
}
```

2.3. Validating multiple objects and/or properties

It is also possible to perform multiple validations for beans and bean properties in one go. This might be useful for example to perform validation of forms that present data from more than one bean. The `Seam.validate()` method takes the following parameters:

```
Seam.validate(validations, callback, groups);
```

The `validations` parameter should contain a list of the validations to perform. It may either be an associative array (for a single validation), or an array of associative arrays (for multiple validations) which define the validations that should be performed. We'll look at this parameter more closely in just a moment.

The `callback` parameter should contain a reference to the callback function to invoke once validation is complete. The optional `groups` parameter should contain the group name/s for which to perform validation.

The `groups` parameter allows one or more validation groups (specified by providing a `String` or array of `String` values) to be validated. The validation groups specified here will be applied to all bean values contained in the `validations` parameter.

The simplest example, in which we wish to validate a single object would look like this:

```
Seam.validate({bean:customer}, callback);
```

In the above example, validation will be performed for the `customer` object, after which the function named `validationCallback` will be invoked.

Validate multiple beans is done by passing in an array of validations:

```
Seam.validate([{bean:customer}, {bean:order}], callback);
```

Single properties can be validated by specifying a `property` name:

```
Seam.validate({bean:customer, property: "firstName"}, callback);
```

To prevent the entire object graph from being validated, the `traverse` property may be set to `false`:

```
Seam.validate({bean:customer, traverse: false}, callback);
```

Validation groups may also be set for each individual validation, by setting the `groups` property to a `String` or array of `Strings` value:

```
Seam.validate({bean:customer, groups: "default"}, callback);
```

2.4. Validation groups

Validation group names should be the unqualified class name of the group class. For example, for the class `com.acme.InternalRegistration`, the client-side group name should be specified as `InternalRegistration`:

```
Seam.validateBean(user, callback, "InternalRegistration"
```

It is also possible to set the default validation groups against which all validations will be performed, by setting the `Seam.ValidationGroups` property:

```
Seam.ValidationGroups = ["Default", "ExternalRegistration"];
```

If no explicit group is set for the default, and no group is specified when performing validation, then the validation process will be executed against the 'Default' group.

2.5. Handling validation failures

If any validations fail during the validation process, then the callback method specified in the validation function will be invoked with an array of constraint violations. If all validations pass, this array will be empty. Each object in the array represents a single constraint violation, and contains the following property values:

`bean` - the bean object for which the validation failed.

`property` - the name of the property that failed validation

`value` - the value of the property that failed validation

`message` - a message string describing the nature of the validation failure

The callback method should contain business logic that will process the constraint violations and update the user interface accordingly to inform the user that validation has failed. The following

minimalistic example demonstrates how the validation errors can be displayed to the user as popup alerts:

```
function validationCallback(violations) {
for (var i = 0; i < violations.length; i++) {
  alert(violations[i].property + "=" + violations[i].value + " [violation] -> " + violations[i].message);
```

Seam Remoting - Model API

3.1. Introduction

The Model API builds on top of Seam Remoting's object serialization features to provide a *component-based* approach to working with a server-side object model, as opposed to the *RPC-based* approach provided by the standard Remoting API. This allows a client-side representation of a server-side object graph to be modified ad hoc by the client, after which the changes made to the objects in the graph can be *applied* to the corresponding server-side objects. When applying the changes the client determines exactly which objects have been modified by recursively walking the client-side object tree and generating a delta by comparing the original property values of the objects with their new property values.

This approach, when used in conjunction with the extended persistence context provided by Seam elegantly solves a number of problems faced by AJAX developers when working remotely with persistent objects. A persistent, managed object graph can be loaded at the start of a new conversation, and then across multiple requests the client can fetch the objects, make incremental changes to them and apply those changes to the same managed objects after which the transaction can be committed, thereby persisting the changes made.

One other useful feature of the Model API is its ability to *expand* a model. For example, if you are working with entities with lazy-loaded associations it is usually not a good idea to blindly fetch the associated objects (which may in turn themselves contain associations to other entities, ad nauseum), as you may inadvertently end up fetching the bulk of your database. Seam Remoting already knows how to deal with lazy-loaded associations by automatically excluding them when marshalling instances of entity beans, and assigning them a client-side value of `undefined` (which is a special JavaScript value, distinct from `null`). The Model API goes one step further by giving the client the option of manipulating the associated objects also. By providing an *expand* operation, it allows for the initialization of a previously-uninitialized object property (such as a lazy-loaded collection), by dynamically "grafting" the initialized value onto the object graph. By *expanding* the model in this way, we have at our disposal a powerful tool for building dynamic client interfaces.

3.2. Model Operations

For the methods of the Model API that accept action parameters, an instance of `Seam.Action` should be used. The constructor for `Seam.Action` takes no parameters:

```
var action = new Seam.Action();
```

The following table lists the methods used to define the action. Each of the following methods return a reference to the `Seam.Action` object, so methods can be chained.

Table 3.1. Seam.Action method reference

Method	Description
`setBeanType(beanType)`	Sets the class name of the bean to be invoked. • `beanType` - the fully qualified class name of the bean type to be invoked.
`setQualifiers(qualifiers)`	Sets the qualifiers for the bean to be invoked. • `qualifiers` - a comma-separated list of bean qualifier names. The names may either be the simple or fully qualified names of the qualifier classes.
`setMethod(method)`	Sets the name of the bean method. • `method` - the name of the bean method to invoke.
`addParam(param)`	Adds a parameter value for the action method. This method should be called once for each parameter value to be added, in the correct parameter order. • `param` - the parameter value to add.

The following table describes the methods provided by the `Seam.Model` object. To work with the Model API in JavaScript you must first create a new Model object:

```
var model = new Seam.Model();
```

Table 3.2. Seam.Model method reference

Method	Description
`addBean(alias, bean, qualifiers)`	Adds a bean value to the model. When the model is fetched, the value of the specified bean will be read and placed into the model, where it may be accessed by using the `getValue()` method with the specified alias. Can only be used before the model is fetched. • `alias` - the local alias for the bean value. • `bean` - the name of the bean, either specified by the `@Named` annotation or the fully qualified class name. • `qualifiers` (optional) - a list of bean qualifiers.

Method	Description
`addBeanProperty(alias, bean, property, qualifiers)`	Adds a bean property value to the model. When the model is fetched, the value of the specified property on the specified bean will be read and placed into the model, where it may be accessed by using the `getValue()` method with the specified alias. Can only be used before the model is fetched. Example: `addBeanProperty("account", "AccountAction", "account", "@Qualifier1", "@Qualifier2");` • `alias` - the local alias for the bean value. • `bean` - the name of the bean, either specified by the `@Named` annotation or the fully qualified class name. • `property` - the name of the bean property. • `qualifiers` (optional) - a list of bean qualifiers. This parameter (and any after it) are treated as bean qualifiers.
`fetch(action, callback)`	Fetches the model - this operation causes an asynchronous request to be sent to the server. The request contains a list of the beans and bean properties (set by calling the `addBean()` and `addBeanProperty()` methods) for which values will be returned. Once the response is received, the callback method (if specified) will be invoked, passing in a reference to the model as a parameter. A model should only be fetched once. • `action` (optional) - a `Seam.Action` instance representing the bean action to invoke before the model values are read and stored in the model. • `callback` (optional) - a reference to a JavaScript function that will be invoked after the model has been fetched. A reference to the model instance is passed to the callback method as a parameter.

Method	Description
`getValue(alias)`	This method returns the value of the object with the specified alias. • `alias` - the alias of the value to return.
`expand(value, property, callback)`	Expands the model by initializing a property value that was previously uninitialized. This operation causes an asynchronous request to be sent to the server, where the uninitialized property value (such as a lazy-loaded collection within an entity bean association) is initialized and the resulting value is returned to the client. Once the response is received, the callback method (if specified) will be invoked, passing in a reference to the model as a parameter. • `value` - a reference to the value containing the uninitialized property to fetch. This can be any value within the model, and does not need to be a "root" value (i.e. it doesn't need to be a value specified by `addBean()` or `addBeanProperty()`, it can exist anywhere within the object graph. • `property` - the name of the uninitialized property to be initialized. • `callback` (optional) - a reference to a JavaScript function that will be invoked after the model has been expanded. A reference to the model instance is passed to the callback method as a parameter.
`applyUpdates(action, callback)`	Applies the changes made to the objects contained in the model. This method causes an asynchronous request to be sent to the server containing a delta consisting of a list of the changes made to the client-side objects. • `action` (optional) - a `Seam.Action` instance representing a bean method to be invoked after the client-side model changes have been applied to their corresponding server-side objects. • `callback` (optional) - a reference to a JavaScript function that will be invoked after the updates have been applied. A reference to the model instance is passed to the callback method as a parameter.

3.3. Fetching a model

To fetch a model, one or more values must first be specified using `addBean()` or `addBeanProperty()` before invoking the `fetch()` operation. Let's work through an example - here we have an entity bean called `Customer`:

```
@Entity Customer implements Serializable {
  private Integer customerId;
  private String firstName;
  private String lastName;

  @Id @GeneratedValue public Integer getCustomerId() { return customerId; }
  public void setCustomerId(Integer customerId) { this.customerId = customerId; }

  public String getFirstName() { return firstName; }
  public void setFirstName(String firstName) { this.firstName = firstName; }

  public String getLastName() { return lastName; }
  public void setLastName(String lastName) { this.lastName = lastName; }
}
```

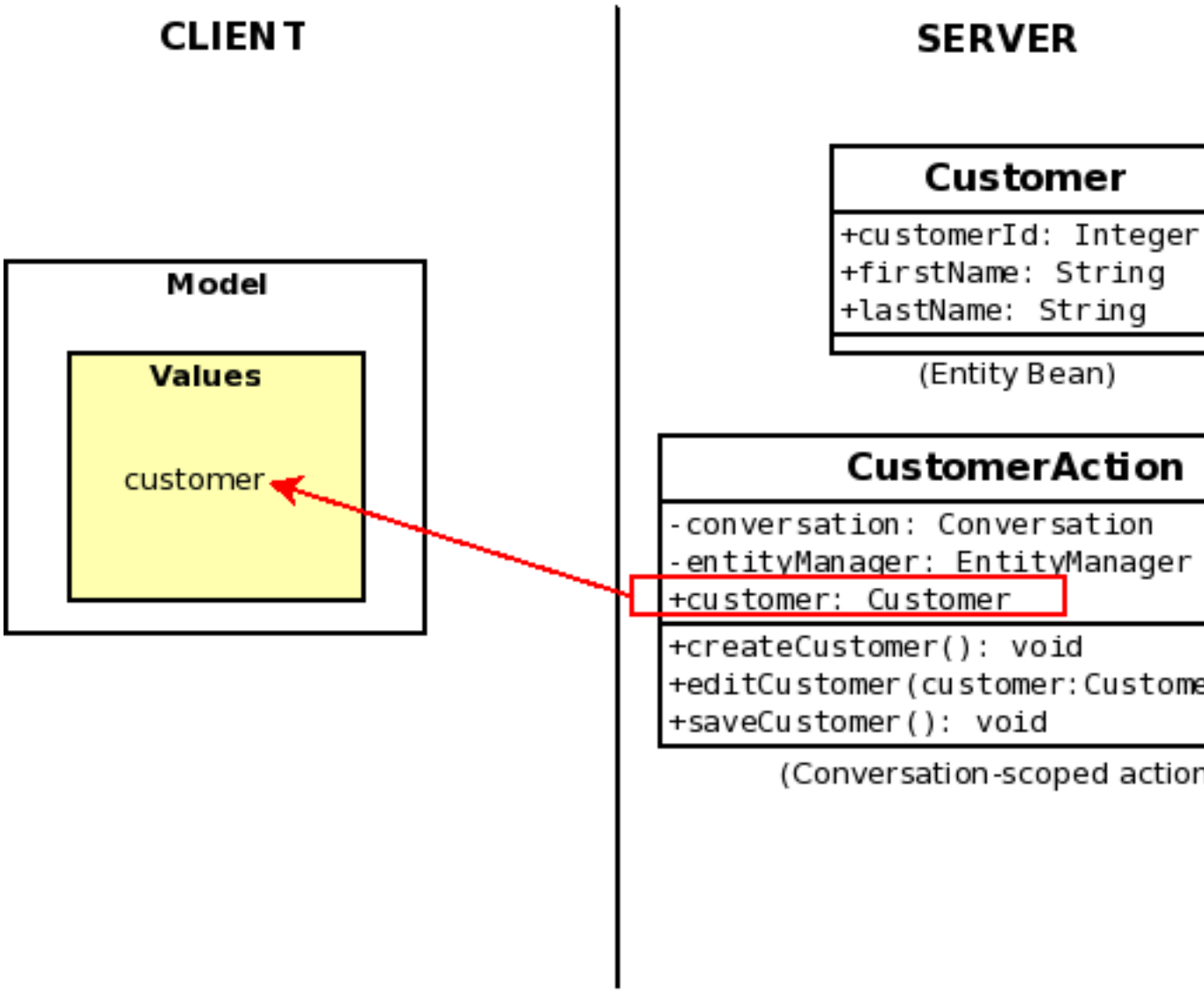
We also have a bean called `CustomerAction`, which is responsible for creating and editing `Customer` instances. Since we're only interested in editing a customer right now, the following code only shows the `editCustomer()` method:

```
@ConversationScoped @Named
public class CustomerAction {
  @Inject Conversation conversation;
  @PersistenceContext EntityManager entityManager;
  public Customer customer;

  public void editCustomer(Integer customerId) {
    conversation.begin();
    customer = entityManager.find(Customer.class, customerId);
  }

  public void saveCustomer() {
    entityManager.merge(customer);
    conversation.end();
  }
}
```

In the client section of this example, we wish to make changes to an existing `Customer` instance, so we need to use the `editCustomer()` method of `CustomerAction` to first load the customer entity, after which we can access it via the public `customer` field. Our model object must therefore be configured to fetch the `CustomerAction.customer` property, and to invoke the `editCustomer()` method when the model is fetched. We start by using the `addBeanProperty()` method to add a bean property to the model:

```
var model = new Seam.Model();
model.addBeanProperty("customer", "CustomerAction", "customer");
```

The first parameter of `addBeanProperty()` is the *alias* (in this case `customer`), which is used to access the value via the `getValue()` method. The `addBeanProperty()` and `addBean()` methods can be called multiple times to bind multiple values to the model. An important thing to note is that the values may come from multiple server-side beans, they aren't all required to come from the same bean.



We also specify the action that we wish to invoke (i.e. the `editCustomer()` method). In this example we know the value of the `customerId` that we wish to edit, so we can specify this value as an action method parameter:

```
var action = new Seam.Action()
  .setBeanType("CustomerAction")
  .setMethod("editCustomer")
  .addParam(123);
```

Once we've specified the bean properties we wish to fetch and the action to invoke, we can then fetch the model. We pass in a reference to the action object as the first parameter of the `fetch()` method. Also, since this is an asynchronous request we need to provide a callback method to deal with the response. The callback method is passed a reference to the model object as a parameter.

```
var callback = function(model) { alert("Fetched customer: "
model.getValue("customer").firstName +
  " " + model.getValue("customer").lastName); };
model.fetch(action, callback);
```

When the server receives a model fetch request, it first invokes the action (if one is specified) before reading the requested property values and returning them to the client.

3.3.1. Fetching a bean value

Alternatively, if you don't wish to fetch a bean *property* but rather a bean itself (such as a value created by a producer method) then the `addBean()` method is used instead. Let's say we have a producer method that returns a qualified `UserSettings` value:

```
@Produces @ConversationScoped @Settings UserSettings getUserSettings() {
  /* snip code */
}
```

We would add this value to our model with the following code:

```
model.addBean("settings", "UserSettings", "@Settings");
```

The first parameter is the local alias for the value, the second parameter is the fully qualified class of the bean, and the third (and subsequent) parameter/s are optional bean qualifiers.

3.4. Modifying model values

Once a model has been fetched its values may be read using the `getValue()` method. Continuing on with the previous example, we would retrieve the `Customer` object via it's local alias (`customer`) like this:

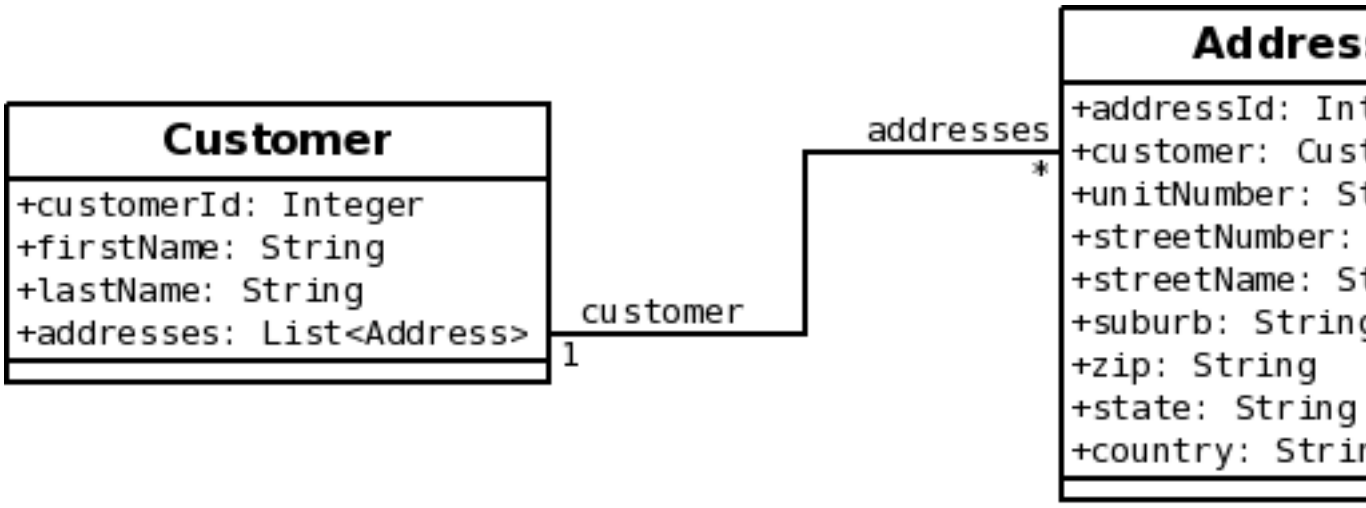
```
var customer = model.getValue("customer");
```

We are then free to read or modify the properties of the value (or any of the other values within its object graph).

```
alert("Customer name is: " + customer.firstName + " " + customer.lastName);
customer.setLastName("Jones"); // was Smith, but Peggy got married on the weekend
```

3.5. Expanding a model

We can use the Model API's ability to expand a model to load uninitialized branches of the objects in the model's object graph. To understand how this works exactly, let's flesh out our example a little more by adding an `Address` entity class, and creating a one-to-many relationship between `Customer` and `Address`.




```
@Entity Address implements Serializable {
  private Integer addressId;
  private Customer customer;
  private String unitNumber;
  private String streetNumber;
  private String streetName;
  private String suburb;
  private String zip;
  private String state;
  private String country;

  @Id @GeneratedValue public Integer getAddressId() {  return addressId; }
  public void setAddressId(Integer addressId) { this.addressId = addressId; }

  @ManyToOne public Customer getCustomer() { return customer; }
  public void setCustomer(Customer customer) { this.customer = customer; }

  /* Snipped other getter/setter methods */

}
```

Here's the new field and methods that we also need to add to the `Customer` class:

```
private Collection<Address> addresses;

@OneToMany(fetch = FetchType.LAZY, mappedBy = "customer", cascade = CascadeType.ALL)
public Collection<Address> getAddresses() { return addresses; }
public void setAddresses(Collection<Address> addresses) { this.addresses = addresses; }
```

As we can see, the `@OneToMany` annotation on the `getAddresses()` method specifies a `fetch` attribute of `LAZY`, meaning that by default the customer's addresses won't be loaded automatically when the customer is. When reading the *uninitialized* `addresses` property value from a newly-fetched `Customer` object in JavaScript, a value of `undefined` will be returned.

```
getValue("customer").addresses == undefined; // returns true
```

We can *expand* the model by making a special request to initialize this uninitialized property value. The `expand()` operation takes three parameters - the value containing the property to be initialized, the name of the property and an optional callback method. The following example shows us how the customer's `addresses` property can be initialized:

```
model.expand(model.getValue("customer"), "addresses");
```

The `expand()` operation makes an asynchronous request to the server, where the property value is initialized and the value returned to the client. When the client receives the response, it reads the initialized value and appends it to the model.

```
// The addresses property now contains an array of address objects
alert(model.getValue("customer").addresses.length + " addresses loaded");
```

3.6. Applying Changes

Once you have finished making changes to the values in the model, you can apply them with the `applyUpdates()` method. This method scans all of the objects in the model, compares them with their original values and generates a delta which may contain one or more changesets to send to the server. A changeset is simply a list of property value changes for a single object.

Like the `fetch()` command you can also specify an action to invoke when applying updates, although the action is invoked *after* the model updates have been applied. In a typical situation the invoked action would do things like commit a database transaction, end the current conversation, etc.

Since the `applyUpdates()` method sends an asynchronous request like the `fetch()` and `expand()` methods, we also need to specify a callback function if we wish to do something when the operation completes.

```
var action = new Seam.Action();
  .setBeanType("CustomerAction")
  .setMethod("saveCustomer");

var callback = function() { alert("Customer saved."); };

model.applyUpdates(action, callback);
```

The `applyUpdates()` method performs a refresh of the model, retrieving the latest state of the objects contained in the model after all updates have been applied and the action method (if specified) invoked.