

Seam Remoting

An AJAX framework for JBoss Seam

Shane Bryzak

shane.bryzak@jboss.com

A brief introduction to Seam



- JBoss Seam is an Enterprise Java framework
 - ✓ Provides a rich, contextual component model
 - ✓ Simplified integration with various useful technologies
 - ✓ Runs in many different environments
 - ✓ Standards based, submitted as JSR-299 (Web Beans)
 - ✓ Supports bijection (inversion of control) for wiring of components
 - ✓ Seam components can be session beans, entity beans or POJOs (Plain Old Java Objects)
 - ✓ Excellent tooling available
 - ✓ Community web site at www.seamframework.org



A simple example



- Hello World – we start with this simple JAVA class:

```
public class HelloAction {
    public String sayHello(String name) {
        return "Hello, " + name;
    }
}
```

A simple example



- We need a single annotation to turn our class into a Seam component:

```
@Name("helloAction")
```

```
public class HelloAction {  
    public String sayHello(String name) {  
        return "Hello, " + name;  
    }  
}
```



A simple example



- We also need to annotate any methods that are to be called via AJAX:

```
@Name("helloAction")
public class HelloAction {
    @WebRemote
    public String sayHello(String name) {
        return "Hello, " + name;
    }
}
```



A simple example



- Next we need to create our web page. We start by importing the required JavaScript
- This is made very easy for us with a custom facelets tag:

```
<s:remote include="helloAction" />
```



A simple example



- We need to write some JavaScript to invoke the component:

```
<script type="text/javascript">
  function sayHello() {
    var name = prompt("What is your name?");
    var callback = function(result){alert(result);};
    Seam.Component.getInstance("helloAction")
      .sayHello(name, sayHelloCallback);
  }
</script>
```



A simple example



- Finally, we add a button to invoke our JavaScript function:

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



A simple example



- All together now:

```
<s:remote include="helloAction"/>
<script type="text/javascript">
  function sayHello() {
    var name = prompt("What is your name?");
    var callback = function(result) { alert(result); };
    Seam.Component.getInstance("helloAction").sayHello(name,
callback);
  }
</script>
<button onclick="javascript:sayHello()">Say Hello</button>
```



A simple example



- Let's see it in action

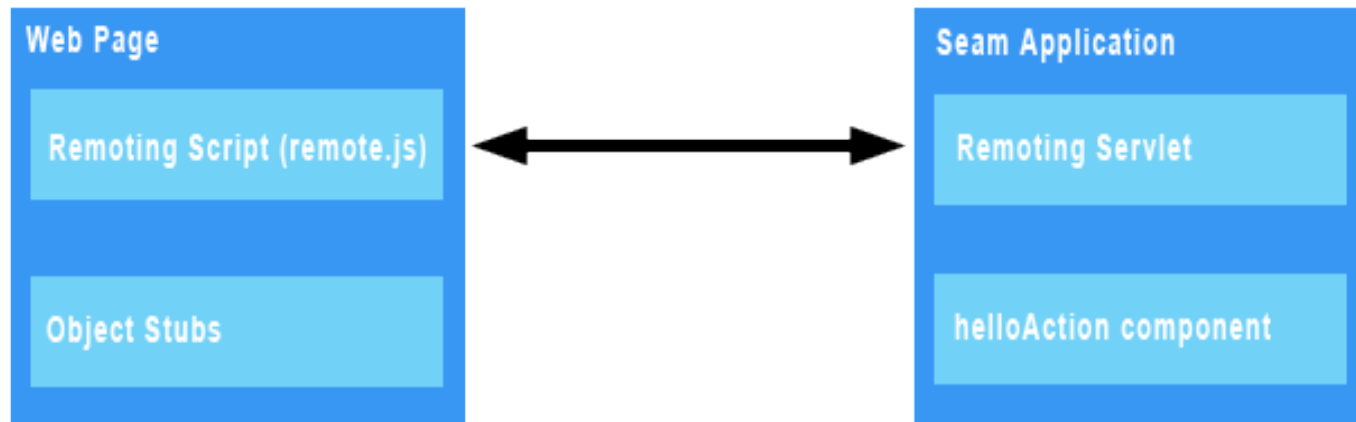
(DEMO)



An overview



- So how does it all work?



Communication Protocol



- Is an XML-based protocol
- Loosely inspired by XML-RPC
- Supports object graph recursion
- Allows requests to be batched



- A closer look at a request packet

```
<envelope>
  <header>
    <context></context>
  </header>
  <body>
    <call component="helloAction" method="sayHello" id="0">
      <params>
        <param><str>Shane</str></param>
      </params>
      <refs></refs>
    </call>
  </body>
</envelope>
```

- And the response that is returned

```
<envelope>
  <header>
    <context><conversationId>3</conversationId></context>
  </header>
  <body>
    <result id="0">
      <value><str>Hello%2C%20Shane</str></value>
      <refs></refs>
    </result>
  </body>
</envelope>
```

- Supported data types
 - ✓ String (including enums)
 - ✓ Number (short, integer, long, float, double)
 - ✓ Boolean
 - ✓ Date
 - ✓ Bag (collections, lists, arrays)
 - ✓ Map
 - ✓ Null
 - ✓ JavaBean types

A more complex example



- Say we have two classes, Foo and Bar:

```
public class Foo {  
    private String value;  
    public String getValue() { return value; }  
    public void setValue(String value) { this.value = value; }  
}
```

```
public class Bar {  
    private Foo foo;  
    public Foo getFoo() { return foo; }  
    public void setFoo(Foo foo) { this.foo = foo; }  
}
```



A more complex example



- And an action component, fooAction:

```
@Name ("fooAction")
public class FooAction {
    @WebRemote
    public Bar execute(Foo foo) {
        Bar bar = new Bar();
        bar.setFoo(foo);
        return bar;
    }
}
```



A more complex example



- Let's call this component from our page:

```
<s:remote include="fooAction"/>
<script type="text/javascript">
  function createFoo(value) {
    var foo = Seam.Remoting.createType("org.jboss.seam.example.remoting.Foo");
    foo.setValue(value);
    return foo;
  }
  function executeFoo() {
    Seam.Component.getInstance("fooAction").execute(createFoo("bar"), fooCallback);
  }
  function fooCallback(bar) {
    alert(bar.getFoo().getValue());
  }
</script>
<button onclick="javascript:executeFoo()">Execute Foo</button>
```



A more complex example



(DEMO)



Batching multiple requests



- Batching requests allows you to reduce the overall number of calls made to the server
- Start a batch by calling:
`Seam.Remoting.startBatch()`
- Send the batch by calling:
`Seam.Remoting.executeBatch()`



Batching multiple requests



- An example:

```
<script type="text/javascript">
  function executeBatch() {
    Seam.Remoting.startBatch();
    Seam.Component.getInstance("fooAction").execute(
      createFoo("foo1"), fooCallback);
    Seam.Component.getInstance("fooAction").execute(
      createFoo("foo2"), fooCallback);
    Seam.Remoting.executeBatch();
  }
</script>

<button onclick="javascript:executeBatch()">Execute Batch</button>
```



Batching multiple requests



(DEMO)



Restricting what is returned



- Some data is sensitive, or just not required
- Use `@WebRemote(exclude = {""})`
- Can constrain fields, maps, collections and objects of specific type
- Multiple constraints can be combined together
- Excluded fields are `undefined` in JavaScript



Restricting data – an example



- A component method that returns an instance of Foo, with its `value` field restricted

```
@WebRemote(exclude =  
{"[org.jboss.seam.example.remoting.Foo].value"})  
public Foo getFoo() {  
    Foo foo = new Foo();  
    foo.setValue("bar");  
    return foo;  
}
```



Restricting data – an example



- The client code

```
<script type="text/javascript">
  function testRestriction() {
    var callback = function(foo) { alert(foo.getValue()); };
    Seam.Component.getInstance("fooAction").getFoo(callback);
  }
</script>
```

```
<button onclick="javascript:testRestriction()">Get restricted
foo</button>
```



Restricting data – an example



(DEMO)



- Seam introduces a new scope – "conversation"
- Multiple concurrent conversations are possible
- A conversation is started with the `@Begin` annotation, and ended with the `@End` annotation
- Each conversation has a unique ID

A conversational component



- A basic conversational component, called "barAction"

```
@Name ("barAction")
@Scope (CONVERSATION)
public class BarAction {
    private String name;
    @WebRemote @Begin public void start(String name) {
        this.name = name;
    }
    @WebRemote @End public void end() {}
    @WebRemote public String getName() {
        return name;
    }
}
```



Calling the component



(DEMO)



A Chatroom example



- We can use Seam Remoting to write a real-time chat room

Chat Room Example

Enter your name

Users



- The JMS (Java Message Service) API is a messaging standard
- Messaging clients may subscribe or publish to a topic or a queue
- Seam makes working with JMS easy
 - ✓ Factories for creating topic/queue connections and sessions
 - ✓ Simply inject the publisher/session objects into your component using `@In`

The chatroom methods



- We start with the local interface for our session bean

@Local

```
public interface ChatRoomActionWebRemote {  
    @WebRemote boolean connect(String name);  
    @WebRemote void disconnect();  
    @WebRemote void sendMessage(String message);  
    @WebRemote Set<String> listUsers();  
}
```



Implementing a chatroom



- Next we create our Seam component
- A conversation-scoped, stateful session bean

```
@Stateful
@Name ("chatroomAction")
@Scope (CONVERSATION)
public class ChatRoomAction
    implements ChatRoomActionWebRemote
```



Injecting the JMS bits



- We use Seam's dependency injection to obtain the JMS topic publisher and session

```
@In(create=true)
```

```
private transient TopicPublisher topicPublisher;
```

```
@In(create=true)
```

```
private transient TopicSession topicSession;
```



Implementing the methods



- A conversation is started when a user connects

```
@Begin
```

```
public boolean connect(String username) {  
    this.username = username;  
    boolean added = chatroomUsers.add(username);  
    if (added) {  
        publish(new ChatroomEvent(  
            "connect", username) );  
    }  
    return added;  
}
```



Chatroom events



- A chatroom event is published every time someone joins, leaves or sends a message to the chatroom

```
@Name("chatroomEvent")
public class ChatroomEvent implements Serializable {
    private String action; // "message", "connect" or "disconnect"
    private String user;
    private String data;
    public ChatroomEvent(String action, String user) { this(action, user, null); }
    public ChatroomEvent(String action, String user, String data) {
        this.action = action;
        this.user = user;
        this.data = data;
    }
    public String getAction() { return action; }
    public String getUser() { return user; }
    public String getData() { return data; }
}
```



Publishing chatroom events



- We use the `topicPublisher` to publish the chatroom event to the topic

```
private void publish(ChatroomEvent message) {
    try {
        topicPublisher.publish(
            topicSession.createObjectMessage(message) );
    }
    catch (Exception ex) {
        throw new RuntimeException(ex);
    }
}
```



The other chatroom methods



- We also implement methods for sending messages and disconnecting

```
public void sendMessage(String message) {  
    publish( new ChatroomEvent("message", username, message) );  
}
```

@End

```
public void disconnect() {  
    chatroomUsers.remove(username);  
    publish( new ChatroomEvent("disconnect", username) );  
}
```



The chatroom client



- We start by writing some JavaScript

```
var chatroom = Seam.Component.getInstance("chatroomAction");
function connect() {
    username = getObject("username").value;
    var connectCallback = function(connected, context) {
        setInterfaceState(connected);
        getObject("username").value = username;
        Seam.Remoting.getContext().setConversationId(context.getConversationId());
    };
    var listUsersCallback = function(users) {
        for (var i = 0; i < users.length; i++)
            addUser(users[i]);
    };
    Seam.Remoting.startBatch();
    chatroom.connect(username, connectCallback);
    chatroom.listUsers(listUsersCallback);
    Seam.Remoting.executeBatch();
    Seam.Remoting.subscribe("chatroomTopic", channelMessageCallback);
}
```



Handling JMS messages



- We also write a callback to handle any incoming JMS messages

```
function channelMessageCallback(message) {
    var ctl = getObject("channelDisplay");
    var actionDTO = message.getValue();
    if (actionDTO.action == "message")
        ctl.innerHTML += "<span style='font-weight:bold' + (actionDTO.getUser() == username
            ? ";color:green" : "") + "'>" + actionDTO.getUser() + "></span> " +
            actionDTO.getData() + "<br/>";
    else if (actionDTO.action == "connect") {
        addUser(actionDTO.getUser());
        ctl.innerHTML += "<span style='font-weight:bold;color:red'>" + actionDTO.getUser()
            + " connected.</span><br/>";
    } else if (actionDTO.action == "disconnect") {
        removeUser(actionDTO.getUser());
        ctl.innerHTML += "<span style='font-weight:bold;color:red'>" + actionDTO.getUser()
            + " disconnected.</span><br/>";
    }
    ctl.scrollTop = ctl.scrollHeight;
}
```



Sending messages



- And a method that sends our message to the chatroom

```
function sendMessage() {  
    var ctl = getObject("messageText");  
    chatroom.sendMessage(ctl.value);  
    ctl.value = "";  
    // Force a poll so that we see our  
    // new message straight away  
    Seam.Remoting.poll();  
}
```



Chatroom in action



(DEMO)



In Summary



- Seam Community Site:
www.seamframework.org
- Questions?

