Introduction to JBoss Seam

Gavin King
gavin.king@jboss.com
gavin@hibernate.org

Java EE 5 programming model

- JSF 1.2
  - ✓ Template language
    - extensible component model for widgets
  - ✓ "Managed bean" component model
    - JavaBeans with dependency injection
    - XML-based declaration
    - session/request/application contexts
  - ✓ Defines interactions between the page and managed beans
    - Fine-grained event model (true MVC)
    - Phased request lifecycle
    - EL for binding controls to managed beans
  - ✓ XML-based "Navigation rules"
    - Ad hoc mapping from logical outcomes to URL

Java EE 5 programming model

- EJB 3.0
  - ✓ Component model for transactional components
    - dependency injection
    - declarative transaction and persistence context demarcation
    - sophisticated state management
  - ✓ ORM for persistence
  - ✓ Annotation-based programming model

Let's suppose we have some data

```
create table Document (
    id bigint not null primary key,
    title varchar(100) not null unique,
    summary varchar(1000) not null,
    content clob not null
)
```


We'll use an entity bean

```
@Entity
public Document {
    @Id @GeneratedValue private Long id;
    private String title;
    private String summary
    private String content;

    //getters and setters...
}
```

Surrogate key identifier attribute

Search page

```
<h:form>
   <table>
      <tr>
         <td>Document Id</td>
         <td><h:inputText value="#{documentEditor.id}"/></td>
      </tr>
   </table>

   <h:commandButton type="submit" value="Find"
                    action="#{documentEditor.get}"/>
</h:form>
```

A JSF-EL *value binding*

A JSF control

A JSF-EL *method binding*

Edit page

```
<f:form>
   <table>
      <tr>
         <td>Title</td>
         <td>
            <h:inputText value="#{documentEditor.title}">
               <f:validateLength maximum="100"/>
            </h:inputText>
         </td>
      </tr>
      <tr>
         <td>Real Name</td>
         <td>
            <h:inputText value="#{documentEditor.summary}">
               <f:validateLength maximum="1000"/>
            </h:inputText>
         </td>
      </tr>
      <tr>
         <td>Password</td>
         <td><h:inputText value="#{documentEditor.content}"/></td>
      </tr>
   </table>

   <h:messages/>

   <h:commandButton type="submit" value="Save" action="#{documentEditor.save}"/>
</f:form>
```

A JSF validator

Should we use a SLSB?

```
@Stateless
public EditDocumentBean implements EditDocument {
   @PersistenceContext
   private EntityManager em;

   public Document get(Long id) {
      return em.find(Document.class, id);
   }

   public Document save(Document doc) {
      return em.merge(doc);
   }
}
```


And a "backing bean"?

```
public class DocumentEditor {
     private Long id;
     private Document document;

     public String getId() { return id; }
     public void setId(Long id) { this.id = id; }

     public String getTitle() { return document.getTitle(); }
     public void setTitle(String title) { document.setTitle(title); }

     //etc...

     @EJB private EditDocument editDocument;

     public String get() {
          document = editDocument.get(id);
          return document==null ? "notFound" : "success";
     }

     public String save() {
          document = editDocument.save(document);
          return "success";
     }

}
```

Properties bound to controls via the value bindings

Action listener methods bound to controls via the method bindings

JSF outcome

Declare the managed bean

```
<managed-bean>
     <managed-bean-name>documentEditor</managed-bean-name>
     <managed-bean-class>
          com.jboss.docs.DocumentEditor
     </managed-bean-class>
     <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

The name of a *contextual variable* we can refer to in the EL

This is a session-scoped component!

JSF Navigation rules

```
<navigation-rule>
     <from-view-id>/getDocument.jsp</from-view-id>
     <navigation-case>
          <from-outcome>success</from-outcome>
          <to-view-id>editDocument.jsp</to-view-id>
     </navigation-case>
</navigation-rule>

<navigation-rule>
     <from-view-id>/editDocument.jsp</from-view-id>
     <navigation-case>
          <from-outcome>success</from-outcome>
          <to-view-id>findDocument.jsp</to-view-id>
     </navigation-case>
</navigation-rule>

<navigation-rule>
     <from-view-id>/editDocument.jsp</from-view-id>
     <navigation-case>
          <from-outcome>notFound</from-outcome>
          <to-view-id>notFound.jsp</to-view-id>
     </navigation-case>
</navigation-rule>
```

Navigation rules map logical, named "outcomes" to URL of the resulting view

The *outcome* returned by the action listener method

Compared to J2EE

- *Much* simpler code
  - ✓ Fewer artifacts (no DTO, for example)
  - ✓ Less noise (EJB boilerplate, Struts boilerplate)
  - ✓ More transparent (no direct calls to HttpSession, HttpRequest)
  - ✓ Much simpler ORM (even compared to Hibernate)
  - ✓ Finer grained components

Compared to J2EE

- *Much* simpler code
  - ✓ Fewer artifacts (no DTO, for example)
  - ✓ Less noise (EJB boilerplate, Struts boilerplate)
  - ✓ More transparent (no direct calls to HttpSession, HttpRequest)
  - ✓ Much simpler ORM (even compared to Hibernate)
  - ✓ Finer grained components
- Also more powerful for complex problems
  - ✓ JSF is amazingly flexible and extensible
  - ✓ EJB interceptors support a kind of "AOP lite"
  - ✓ Powerful ORM engine

Compared to J2EE

- *Much* simpler code
  - ✓ Fewer artifacts (no DTO, for example)
  - ✓ Less noise (EJB boilerplate, Struts boilerplate)
  - ✓ More transparent (no direct calls to HttpSession, HttpRequest)
  - ✓ Much simpler ORM (even compared to Hibernate)
  - ✓ Finer grained components
- Also more powerful for complex problems
  - ✓ JSF is amazingly flexible and extensible
  - ✓ EJB interceptors support a kind of "AOP lite"
  - ✓ Powerful ORM engine
- Unit testable
  - ✓ All these components (except the JSP pages) may be unit tested using JUnit or TestNG

Room for improvement

- The managed bean is just noise – its concern is pure "glue"
  - ✓ and it accounts for more LOC than any other component!
  - ✓ it doesn't really decouple layers, in fact the code is more coupled than it would otherwise be

Room for improvement

- The managed bean is just noise – its concern is pure "glue"
  - ✓ and it accounts for more LOC than any other component!
  - ✓ it doesn't really decouple layers, in fact the code is more coupled than it would otherwise be
- This code does not work in a multi-window application
  - ✓ and to make it work is a major architecture change!

Room for improvement

- The managed bean is just noise – its concern is pure "glue"
  - ✓ and it accounts for more LOC than any other component!
  - ✓ it doesn't really decouple layers, in fact the code is more coupled than it would otherwise be
- This code does not work in a multi-window application
  - ✓ and to make it work is a major architecture change!
- The application leaks memory
  - ✓ the backing bean sits in the session until the user logs out
  - ✓ in more complex apps, this is often a source of bugs!

Room for improvement

- The managed bean is just noise – its concern is pure "glue"
  - ✓ and it accounts for more LOC than any other component!
  - ✓ it doesn't really decouple layers, in fact the code is more coupled than it would otherwise be
- This code does not work in a multi-window application
  - ✓ and to make it work is a major architecture change!
- The application leaks memory
  - ✓ the backing bean sits in the session until the user logs out
  - ✓ in more complex apps, this is often a source of bugs!
- "Flow" is weakly defined
  - ✓ navigation rules are totally ad hoc and difficult to visualize
  - ✓ How can this code be aware of the long-running business process?

Room for improvement

- The managed bean is just noise – its concern is pure "glue"
  - ✓ and it accounts for more LOC than any other component!
  - ✓ it doesn't really decouple layers, in fact the code is more coupled than it would otherwise be
- This code does not work in a multi-window application
  - ✓ and to make it work is a major architecture change!
- The application leaks memory
  - ✓ the backing bean sits in the session until the user logs out
  - ✓ in more complex apps, this is often a source of bugs!
- "Flow" is weakly defined
  - ✓ navigation rules are totally ad hoc and difficult to visualize
  - ✓ How can this code be aware of the long-running business process?
- JSF is still stuck using XML for things that are better done in annotations!

The case for SFSB

- "stateful session beans are unscalable" … why?
  - ✓ replicating conversational state in a clustered environment (needed for transparent failover) is somewhat expensive

The case for SFSB

- "stateful session beans are unscalable" … why?
  - ✓ replicating conversational state in a clustered environment (needed for transparent failover) is somewhat expensive
- solution 1: keep all state in the database

The case for SFSB

- "stateful session beans are unscalable" … why?
  - ✓ replicating conversational state in a clustered environment (needed for transparent failover) is somewhat expensive
- solution 1: keep all state in the database
  - ✓ traffic to/from database is even more expensive (database is the *least* scalable tier)
  - ✓ So, inevitably, end up needing a second-level cache
  - ✓ second-level cache must be kept transactionally consistent between the database and every node on the cluster – even more expensive!

The case for SFSB

- "stateful session beans are unscalable" … why?
  - ✓ replicating conversational state in a clustered environment (needed for transparent failover) is somewhat expensive
- solution 1: keep all state in the database
  - ✓ traffic to/from database is even more expensive (database is the *least* scalable tier)
  - ✓ So, inevitably, end up needing a second-level cache
  - ✓ second-level cache must be kept transactionally consistent between the database and every node on the cluster – even more expensive!
- solution 2: keep state in the **`HttpSession`**

The case for SFSB

- "stateful session beans are unscalable" … why?
  - ✓ replicating conversational state in a clustered environment (needed for transparent failover) is somewhat expensive
- solution 1: keep all state in the database
  - ✓ traffic to/from database is even more expensive (database is the *least* scalable tier)
  - ✓ So, inevitably, end up needing a second-level cache
  - ✓ second-level cache must be kept transactionally consistent between the database and every node on the cluster – even more expensive!
- solution 2: keep state in the **`HttpSession`**
  - ✓ totally nuts, since **`HttpSession`** is exactly the same as a SFSB
  - ✓ but it does not have dirty-checking,
  - ✓ and methods of a JavaBean in the session can't be transactional

JBoss Seam

- Unify the two component models
  - ✓ Simplify Java EE 5, filling a gap
  - ✓ Improve usability of JSF

JBoss Seam

- Unify the two component models
  - ✓ Simplify Java EE 5, filling a gap
  - ✓ Improve usability of JSF
- Integrate jBPM
  - ✓ BPM technology for the masses

JBoss Seam

- Unify the two component models
  - ✓ Simplify Java EE 5, filling a gap
  - ✓ Improve usability of JSF
- Integrate jBPM
  - ✓ BPM technology for the masses
- Deprecate so-called stateless architecture
  - ✓ Managed application state -> more robust, more performant, richer user experience
  - ✓ Take advantage of recent advances in clustering technology

JBoss Seam

- Unify the two component models
  - ✓ Simplify Java EE 5, filling a gap
  - ✓ Improve usability of JSF
- Integrate jBPM
  - ✓ BPM technology for the masses
- Deprecate so-called stateless architecture
  - ✓ Managed application state -> more robust, more performant, richer user experience
  - ✓ Take advantage of recent advances in clustering technology
- Decouple the technology from the execution environment
  - ✓ Run EJB3 apps in Tomcat
  - ✓ Or in TestNG
  - ✓ Use Seam with JavaBeans and Hibernate

JBoss Seam

- Unify the two component models
  - ✓ Simplify Java EE 5, filling a gap
  - ✓ Improve usability of JSF
- Integrate jBPM
  - ✓ BPM technology for the masses
- Deprecate so-called stateless architecture
  - ✓ Managed application state -> more robust, more performant, richer user experience
  - ✓ Take advantage of recent advances in clustering technology
- Decouple the technology from the execution environment
  - ✓ Run EJB3 apps in Tomcat
  - ✓ Or in TestNG
  - ✓ Use Seam with JavaBeans and Hibernate
- Enable richer user experience

Contextual components

- Most of the problems relate directly or indirectly to *state management*
  - ✓ The contexts defined by the servlet spec are not meaningful in terms of the application
  - ✓ EJB itself has no strong model of state management
  - ✓ We need a richer context model that includes “logical” contexts that are meaningful to the application

Contextual components

- Most of the problems relate directly or indirectly to *state management*
  - ✓ The contexts defined by the servlet spec are not meaningful in terms of the application
  - ✓ EJB itself has no strong model of state management
  - ✓ We need a richer context model that includes "logical" contexts that are meaningful to the application
- We also need to fix the mismatch between the JSF and EJB 3.0 component models
  - ✓ We should be able to use annotations everywhere
  - ✓ An EJB should be able to be a JSF managed bean (and vice versa)

Contextual components

- Most of the problems relate directly or indirectly to *state management*
  - ✓ The contexts defined by the servlet spec are not meaningful in terms of the application
  - ✓ EJB itself has no strong model of state management
  - ✓ We need a richer context model that includes "logical" contexts that are meaningful to the application
- We also need to fix the mismatch between the JSF and EJB 3.0 component models
  - ✓ We should be able to use annotations everywhere
  - ✓ An EJB should be able to be a JSF managed bean (and vice versa)
- It makes sense to think of binding EJB components directly to the JSF view
  - ✓ A session bean acts just like a backing bean, providing event listener methods, etc
  - ✓ The entity bean provides data to the form, and accepts user input

Slight change to the edit page

```
<f:form>
   <table>
      <tr>
         <td>Title</td>
         <td>
            <h:inputText value="#{documentEditor.document.title}">
                <f:validateLength maximum="100"/>
            </h:inputText>
         </td>
      </tr>
      <tr>
         <td>Real Name</td>
         <td>
            <h:inputText value="#{documentEditor.document.summary}">
                <f:validateLength maximum="1000"/>
            </h:inputText>
         </td>
      </tr>
      <tr>
         <td>Password</td>
         <td><h:inputText value="#{documentEditor.document.content}"/></td>
      </tr>
   </table>

   <h:messages/>

   <h:commandButton type="submit" value="Save" action="#{documentEditor.save}"/>
</f:form>
```

Bind view to the entity bean *directly*

Our first Seam component!

```
@Stateful
@Name("documentEditor")
public EditDocumentBean implements EditDocument {
    @PersistenceContext
    private EntityManager em;
    private Long id;
    public void setId(Long id) { this.id = id; }

    private Document document;
    public Document getDocument() { return document; }

    @Begin
    public String get() {
        document = em.find(Document.class, id);
        return document==null ? "notFound" : "success";
    }

    @End
    public String save() {
        document = em.merge(document);
        return "success";
    }
}
```

The **@Name** annotation binds the component to a contextual variable – it's just like **<managed-bean-name>** in the JSF XML

The **@Begin** annotation defines the beginning of a logical scope – it starts a *conversation*

The **@End** annotation ends the conversation – a conversation can also end by being timed out

The Seam context model

- Seam defines a rich context model for stateful components, enabling container-management of application state
- The contexts are:
  - ✓ **EVENT**
  - ✓ **PAGE**
  - ✓ **CONVERSATION**
  - ✓ **SESSION**
  - ✓ **BUSINESS_PROCESS**
  - ✓ **APPLICATION**
- The highlighted "logical" contexts are demarcated by the application itself
  - ✓ Using annotations: **@Begin, @End, @BeginTask, @EndTask**

Seam component model

- Components are associated with context variables using **@Name** or **@Role**
  - ✓ this allows Seam to instantiate when the context variable is null
- Components are assigned a scope using the **@Scope** annotation
- Seam recognizes four types of component
  - ✓ Stateful session bean – default to **CONVERSATION** scope
  - ✓ Stateless session bean – **STATELESS** pseudo-scope
  - ✓ Entity bean – default to **CONVERSATION** scope
  - ✓ JavaBean – default to **EVENT** scope

Demo

- Seam Hotel Booking Demo

Conversations

- How is state stored between requests?
  - ✓ Server-side conversations (HttpSession + conversation timeout)
  - ✓ Client-side conversations (serialize into the page)
  - ✓ Business process state is made persistent by jBPM
- Conversations are not that exciting until you really start thinking about them:
  - ✓ multi-window operation
  - ✓ "workspace management"
  - ✓ Back button operation
- What about a conversation that involves multiple distinct "steps"
  - ✓ the steps may be completed in parallel
  - ✓ A nested conversation has write access to its own variables, read access to the outer conversation variables
  - ✓ Seam's nested conversation model represents a stack of continuable states

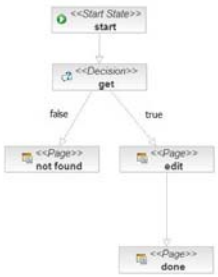
Demo

- Seam Issue Tracker Demo

Pageflow

- Two models for conversational pageflow
  - ✓ The stateless model: JSF navigation rules
    - ad hoc navigation (the app must handle backbutton)
    - actions tied to UI widgets
  - ✓ The stateful model: jBPM pageflow
    - no ad hoc navigation (back button usually bypassed)
    - actions tied to UI widgets *or called directly from pageflow transitions*
- Simple applications only need the stateless model
- Some applications need both models

jBPM pageflow definition



jBPM pageflow definition

```
<pageflow-definition name="EditDocument">

  <start-page name="start" view-id="/findDocument.jsp">
    <transition to="get">
      <action expression="#{documentEditor.get}"/>
    </transition>
  </start-page>

  <decision name="get" expression="#{documentEditor.found}">
    <transition name="false" to="not found"/>
    <transition name="true" to="edit"/>
  </decision>

  <page name="not found" view-id="/notFound.jsp">
    <end-conversation/>
  </page>

  <page name="edit" view-id="/editDocument.jsp">
    <transition to="done">
      <action expression="#{documentEditor.save}"/>
    </transition>
  </page>

  <page name="done" view-id="/findDocument.jsp">
    <end-conversation/>
  </page>

</pageflow-definition>
```

A jBPM *state transition action*, instead of a JSF action listener

A jBPM *decision node*, instead of a JSF navigation rule

Each **<page>** node is a jBPM *wait state* – the pageflow "waits" for user input

Search page

```
<h:form>
   <table>
      <tr>
         <td>Document Id</td>
         <td><h:inputText value="#{documentEditor.id}"/></td>
      </tr>
   </table>

   <h:commandButton type="submit" value="Find"/>
</h:form>
```

The method binding is no longer needed

Pure business logic

```
@Stateful
@Name("documentEditor")
public EditDocumentBean implements EditDocument {
     @PersistenceContext private EntityManager em;
     private Long id;
     public void setId(Long id) { this.id = id; }

     private Document document;
     public Document getDocument() { return document; }

     @Create @Begin(pageflow="EditDocument")
     public void start() {}

     public void get() {
          document = em.find(Document.class, id);
     }

     public boolean isFound() {
          return document!=null;
     }

     @End
     public void save(Document doc) {
          document = em.merge(doc);
     }
}
```

When the component is first created, the pageflow execution begins

Notice that the outcomes have disappeared from the component code

Demo

- Seam DVD Store Demo (1)

What about business process?

- Different from a conversation
  - ✓ long-running (persistent)
  - ✓ multi-user
  - ✓ (The lifespan of a business process instance is longer than the process definition!)
- A conversation that is significant in terms of the overarching business process is called a "task"
  - ✓ driven from the jBPM task list screen
- We demarcate work done in a task using **`@BeginTask`** / **`@ResumeTask`** and **`@EndTask`**
- Work done in the scope of a task also has access to the **`PROCESS`** scope
  - ✓ In addition to the task's **`CONVERSATION`** scope

Start a business process

```
@Name("documentSubmission")
@Stateful
public class DocumentSubmissionBean implements DocumentSubmission {
   @PersistenceContext EntityManager entityManager;
   @Out(scope=PROCESS) Long documentId;
   private Document document;

   //some conversation ...

   @CreateProcess(definition="DocumentSubmission")
   public String submitDocument() {
      documentId = document.getId();
      return "submitted";
   }

}
```

Outject **documentId** to the business process context

Create a new *business process instance*

jBPM process definition

jBPM process definition

```
<process-definition name="DocumentSubmission">

    <start-state name="start">
        <transition to="review"/>
    </start-state>

    <task-node name="review">

        <task name="review">
            <assignment actorId="#{user.manager.id}" />
        </task>

        <transition name="approve" to="approved">
            <action expression="#{email.sendApprovalEmail}"/>
        </transition>

        <transition name="reject" to="rejected"/>

    </task-node>

    <end-state name="approved"/>
    <end-state name="rejected"/>

</process-definition>
```

In this case, the wait states are **<task>** nodes, where the process execution waits for the user to begin work on a task

A jBPM *task assignment*, via EL evaluated in the Seam contexts

Perform the task

```
@Name("reviewDocument")
@Stateful
public class ReviewDocumentBean implements ReviewDocument {
   @PersistenceContext EntityManager entityManager;
   @In Long documentId;
   @Out Document document;

   @BeginTask
   public String getDocument() {
      document = entityManger.find(Document.class, documentId);
      return "reviewDocument";
   }

   @EndTask(transition="approve")
   public String approve() { return "documentApproved"; }

   @EndTask(transition="reject")
   public String approve() { return "documentRejected"; }
}
```

documentId *injected* from business process context

document *outjected* to the conversation context

End the task, specifying a *transition name*

Demo

- Seam DVD Store Demo (2)

What about dependency injection?

- Dependency injection is broken for stateful components
  - ✓ A contextual variable can be written to, as well as read!
  - ✓ Its value changes over time
  - ✓ A component in a wider scope must be able to have a reference to a component in a narrower scope
- Dependency injection was designed with J2EE-style *stateless services* in mind – just look at that word "dependency"
  - ✓ it is usually implemented in a static, unidirectional, and non-contextual way
- For stateful components, we need *bijection*
  - ✓ dynamic, contextual, bidirectional
- Don't think of this in terms of "dependency"
  - ✓ Think about this as *aliasing a contextual variable into the namespace of the component*

What does it look like?

```
@Stateless
@Name("changePassword")
public class ChangePasswordBean implements Login {
    @PersistenceContext
    private EntityManager em;

    @In @Out
    private User currentUser;

    public String changePassword() {
        currentUser = em.merge(currentUser);
    }
}
```

The **@In** annotation injects the value of the contextual variable named **currentUser** into the instance variable each time the component is invoked

The **@Out** annotation "outjects" the value of the instance variable back to the **currentUser** contextual variable at the end of the invocation

Conversations and persistence

- The notion of *persistence context* is central to ORM
  - ✓ A canonicalization of pk -> java instance
  - ✓ without it, you lose referential integrity
  - ✓ it is also a natural cache
- A process-scoped persistence context is evil
  - ✓ requires in-memory locking and sophisticated deadlock detection
- A transaction-scoped persistence context has problems if you re-use objects across transactions
  - ✓ LazyInitializationException navigating lazy associations
  - ✓ NonUniqueObjectException reassociating detached instances
  - ✓ Less opportunity for caching (workaround: use a second-level cache, which is quite unscalable)
- EJB3-style component-scoped persistence context is nice *but…*
  - ✓ not held open for entire request (while rendering view)
  - ✓ problems propagating across components
- Solution: *conversation-scoped persistence contexts*

Seam-managed persistence context

```
@Name("documentEditor")
@Stateful
public class DocumentEditorBean implements DocumentEditor {

   @In(create=true)
   EntityManager documentDatabase;

   private Document document;

   public void retrieveDocument(Long docId) {
      document = documentDatabase.find(Document.class, docId);
   }

   public Comment[] getImages() {
      return document.getComments();
   }

   public void setDocumentSummary(String summary) {
      document.setSummary(summary);
   }
}
```

injects a Seam-manged persistence context that is scoped to the conversation

The entity remains managed throughout the conversation

Seam transaction management

- When using Seam-managed persistence contexts, it makes more sense to demarcate transactions according to the lifecycle of the web request
- We want as few transactions as possible, but we always want a transaction active
- We want to avoid displaying success messages to the user before the transaction has completed
- Solution: one transaction for read/write operations during the first part of the request, up to and including **INVOKE_APPLICATION**, a second transaction for read-only operations during the **RENDER_RESPONSE** phase

Model-based constraints

- Validation belongs in the user interface
  - ✓ or does it?
- Most "validations" reflect constraints that also appear in the database
  - ✓ If we look closer, the same constraints appear in multiple places: the presentation layer, the persistence layer, the database schema
- It would be better to declare these constraints in just one place: the data model

Hibernate Validator

```
@Entity
public Document {
    @Id @GeneratedValue private Long id;
    @Length(max=100) @NotNull private String title;
    @Length(max=300) private String summary
    @NotNull private String content;

   //getters and setters...
}
```


Hibernate Validator

```
<f:form>
   <table>
      <s:validateAll>
         <tr>
            <td>Title</td>
            <td><h:inputText value="#{documentEditor.title}"/></td>
         </tr>
         <tr>
            <td>Real Name</td>
            <td><h:inputText value="#{documentEditor.summary}"/></td>
         </tr>
         <tr>
            <td>Password</td>
            <td><h:inputText value="#{documentEditor.content}"/></td>
         </tr>
      </s:validateAll>
   </table>

   <h:messages/>

   <h:commandButton type="submit" value="Save"
                    action="#{documentEditor.save}"/>
</f:form>
```

Remove the JSF validation (unless we need JavaScript-level validation)

Validate the submitted data using Hibernate Validator

Seam Remoting

- Call Seam components from JavaScript
  - ✓ JavaScript proxies generated dynamically at runtime, and provided to the client by a servlet
  - ✓ method call and parameters are transmitted asynchronously via XMLHttpRequest
  - ✓ Method return value is passed to a callback function

A stateless Seam component

```
@Stateless
@Name("documentFinder")
public class DocumentFinderBean implements DocumentFinder {

   @PersistenceContext em;

   public List<Document> find(String searchString) {
      return em.createQuery("from Document where title like :search")
               .setParameter("search", searchString)
               .getResultList();
   }

}
```


Local interface

```
@Local
public interface DocumentFinder {
   @WebRemote List<Document> find(String searchString);
}
```

The **@WebRemote** annotation exposes the method of the local interface on the client side, via a JavaScript proxy

HTML page

```
<div>
  <input type="text" id="searchString"/>
  <input type="submit" value="Search"
         onclick="doSearch(); return false;"/>
</div>
<div>
  <table id="results">
     <!– search results display here -->
  </table>
</div>
```


JavaScript

```
function doSearch() {
   var searchString = document.getElementById("searchString").value;
   var documentFinder = Seam.Component.getInstance("documentFinder");
   documentFinder.find(searchString, displayResults);
}

function displayResults(docs) {
   for (doc in docs) {
      var tr = createElement("tr");
      document.getElementById("results").appendChild(tr);
      var tdTitle = document.createElement("td");
      tr.appendChild(tdTitle);
      tdTitle.appendChild( document.createTextNode(doc.title) );
      var tdSummary = document.createElement("td");
      tr.appendChild(tdSummary);
      tdSummary.appendChild( document.createTextNode(doc.summary) );
   }
}
```

Call a method asynchronously

Pass a reference to the callback function

Get an instance of the JavaScript proxy

Return value gets passed as an argument to the callback

Demo

- Seam Remoting

Roadmap

- Seam 1.0 rc 2 out now
  - ✓ JSR-168 Portal integration
  - ✓ Enhanced i18n
  - ✓ Seam Remoting
- Seam 1.0 final for JavaOne
- Seam 1.1 in Q3
  - ✓ Asynchronicity/Calendaring
- Seam 1.5
  - ✓ Seam for SOA / ESB
  - ✓ Drools integration?
- Future
  - ✓ Seam for rich clients?