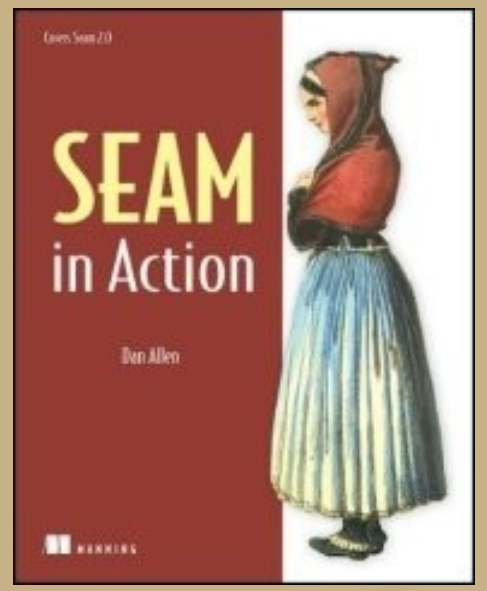
JBoss Seam

Integration with intent to use

Dan Allen
Software Consultant
Author, Seam in Action

Who am I?

- Author of Seam in Action
- Author of the *Seamless JSF* series
- Committer on the JBoss Seam Project
- Software consultant
- Linux and open source advocate
- http://www.mojavelinux.com



Who am I *really*?

A software developer looking for solutions to deliver projects on time and on budget

...and who digs

What is JBoss Seam?

- I've overheard some say...
  - "Seam, that sounds too technical for me."
  - "Isn't SEAM an acronym for something?"
  - "I wish Seam could finish my curtains."

What is JBoss Seam?

Let's not start with this question

Let's **start** with the *problem*

...then **look** for a *solution*

Dan's Top 5 List: Development challenges

#5 You have a lot of requirements, but not a lot of time

#4 You burn too much time setting up the project

#3 Glue code must be written to integrate disparate technologies

#2 Managing state in a web application is a *pain*

#1 You never get to the fun stuff (Ajax, PDFs, email)

Seam-gen: Seam's project generator

- Assembles Ant-based project in minutes
  - Supports EAR and WAR archive formats
- Ready to be imported into an IDE
  - Officially supports Eclipse and NetBeans
  - Any IDE that can drive Ant tasks
- Can handle deployment to JBoss AS
  - Incremental hot redeployment for "instant change"
- Build includes three profiles (dev, test, prod)

Why use seam-gen?

- Let's you get to work quickly
- Let's you use Seam in its *natural* environment
- Gets you doing the fun stuff sooner

You always have the option to go your own way once you are comfortable using Seam

Seam-gen: Seam's code generator

- Seam's answer to Ruby on Rails (and Grails)
- Can build an entire CRUD application
  - From a database
  - From existing Java classes
- Can generate code templates
  - View templates (Facelets)
  - JPA entity classes
  - Action components
  - Integration test cases

Seam-gen: setup

- Requirements to setup new project
  - JBoss AS >= 4.2.2 installed
  - Database server (unless using embedded)
  - Database JDBC driver JAR
  - Existing tables or generated schema
  - Project name
  - Projects directory

Seam-gen's package

- JCA DataSource – database connection pool
- JPA Persistence Unit – object/relational mapping
- RichFaces (or ICEFaces) – UI component library
- Facelets view templates – basic layout
  - Facelets purely XML-based, looks like clean JSP
- Supplemental stylesheet
- Internationalization (i18n) message bundles
- Embedded JBoss for integration testing

Dealing in tiers

- Persistence layer
- Business layer
- Presentation layer

Every web application's got 'em

It's your job to *integrate* them

The Seam component model

- Unifies components across tiers and libraries
  - Java Persistence API (JPA) or Hibernate
  - Enterprise JavaBeans (EJB 3) or plain JavaBeans
  - JavaServer Faces (JSF)
- Unifies contexts across and entire application
  - Java Servlet API (request, session, application)
  - JSF component tree (page)
  - Seam (conversation)
  - jBPM (business process)

Contextual components

- Component
  - Definition of how to create an object
  - Managed class instantiated by Seam
  - Instances represent state of system; typically scoped to the use-case
- Context
  - Buckets where the component instances are stored
  - Accessible from anywhere
  - One way to access contexts

Contextual means that **state** matters!

Seam component vs JSF managed bean

- Shares same idea of a container-created object
- JSF can also store object in stateful scopes
  - Scope options are limited
- Seam component is more capable
  - Additional scope options
  - Strung with interceptors
    - manages life cycle of instance
    - decorates with services (transactions, security, etc)

JSF managed bean

```
<managed-bean>
  <managed-bean-name>beerAction</managed-bean-name>
  <managed-bean-class>
    org.connessieur.action.BeerAction
  </managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
</managed-bean>
```

JavaBean-Seam component

```
@Name("beerAction")
public class BeerAction {
    ...
}
```

- Stored in request scope by default
- Request scope == event context

Conversational component

```
@Name("beerAction")
@Scope(ScopeType.CONVERSATION)
public class BeerAction {
    ...
}
```

- Seam introduces the conversation context
- Supports a single use-case, which may span multiple pages and actions

EJB 3-Seam component

```
@Stateful
@Name("beerAction")
public class BeerActionBean implements BeerAction {
    ...
}
```

- Stored in conversation context by default
- Instantiated by EJB 3 container then intercepted by Seam

Annotations instead of XML

No more XM-hell!

More concise XML

But if you must...

```
<component name="beerAction"
  class="org.connessieur.action.BeerAction"/>
```

- Declared in the Seam component descriptor
  - META-INF/components.xml
  - BeerAction.component.xml

Looking up a component

- Every component has a name
- When name is requested, instance is created
  - Some situations in which creation is skipped
- Several ways to perform lookup
  - Value expression - #{beerAction}
  - Annotation - @In("beerAction")
  - Seam API - Component.getInstance("beerAction")

Using a component

- Capture form data
  - Bind properties to input fields
- Respond to action in user interface
  - Click a button
- Model a sequence of interactions
  - Demarcate conversation boundaries
  - Hold state for a conversation

Capturing form input in JSF

- Abstraction over HTTP protocol and POST data
- Input element bound directly to model property
  - Uses value binding expression - `#{beer.name}`
- On submit, JSF will...
  - read value from POST data - name
  - convert and validate value
  - create new entity class instance - Beer
  - assign value to property on model - setName()

Entity class as a component

- Entity plays central role in Seam application
- Object Relational Mapping (ORM) tool handles the task of persistence
  - Java Persistence API (JPA)
  - Hibernate (native)
- Life cycle of entity controlled by ORM
- Seam seeds the new instance
- Instance populated using JSF form

Defining an entity component

```
@Entity
@Name("beer")
public class Beer implements Serializable {
    protected Long id;
    protected String name;
    ...
    @Id @GeneratedValue
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String n) { this.name = n; }
    ...
}
```

Seam-gen: new-form

- Use new-form command to create stub JSF view
  - Component name: beerAction
  - Bean class name: BeerAction
  - Action method name: save
  - Page name: editBeer
- Artifacts generated
  - BeerAction.java (under src/action)
  - editBeer.xhtml (under view)
  - BeerActionTest.java (under src/test)

Binding entity to form inputs

editBeer.xhtml

```
<h:form id="beer">
  <rich:panel>
    <f:facet name="header">Add Beer</f:facet>
    <s:decorate id="nameField"
      template="layout/edit.xhtml">
      <ui:define name="label">Name</ui:define>
      <h:inputText id="name" required="true"
        value="#{beer.name}"/>
    </s:decorate>
    ...
  </rich:panel>
</h:form>
```

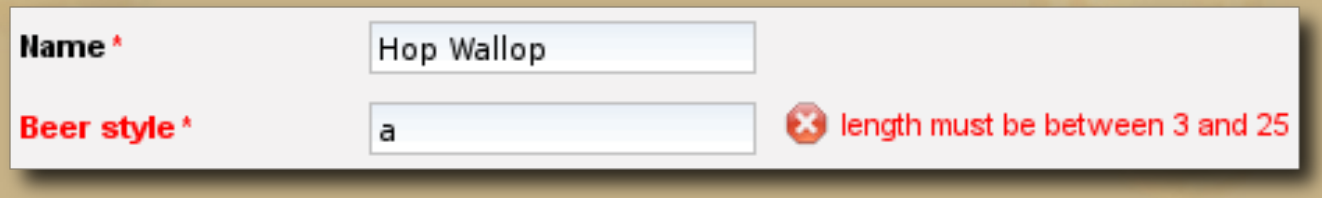
Seam UI decorator

- Builds JSF markup dynamically
  - Composite view pattern
- Adds two implicit context variables
  - required – as defined on input component
  - invalid – indicates if input has validation errors
- Support functionality of `<s:label>` and `<s:message>`
  - akin to `<h:outputLabel>` and `<h:message>`
  - automatically associated with input component

Input field template

layout/edit.xhtml

```
<ui:composition ...>
  <div class="prop">
    <s:label styleClass="name">
      <ui:insert name="label"/>
      <s:span class="required"
        rendered="#{required}">*</s:span>
    </s:label>
    <span class="value">
      <ui:validateAll><ui:insert/></ui:validateAll>
    </span>
    <span class="error"><s:message/></span>
  </div>
</ui:composition>
```



Enforcing validations

- Before submitting form, inputs must be validated
- Validations traditionally defined using UI component tags
  - `<f:validateLength minimum="3" maximum="50"/>`
- Better to define validations in domain model
  - not repeated on every page property is used
  - centralized for other layers to access

Hibernate validator

- Define validation criteria using annotations
  - `@Length(min = 3, max = 50)`
  - `@Email, @CreditCardNumber, etc.`
- Seam adapts Hibernate validator to JSF validation
  - `<s:validate/>` nested within input component
  - `<s:validateAll/>` wrapped around input component
  - apply to inputs automatically using `<s:decorate>`
- Validations take effect in UI immediately
  - Only wort is that required="true" still necessary

Model validations

```
@Entity
@Name("beer")
public class Beer implements Serializable {
    ...
    protected String name;

    @Length(min = 3, max = 50)
    public String getName() { return this.name; }
    ...
}
```

Custom validations

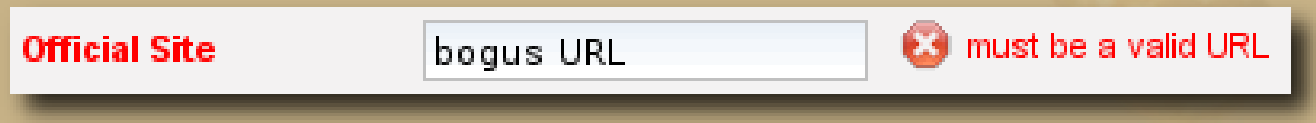
- Validate property using regular expressions
  - `@Pattern(regex="^abc$")`
  - `@Patterns({@Pattern(...)})`
- Define your own validation stereotype annotation
  - Implement `Validator` interface
  - Bind validator to stereotype annotation
  - `@ValidatorClass(UrlValidator.class)`

```
...
public @interface Url {
    String message() default "must be valid URL";
}
```

Applying a custom validator

```
@Entity
@Name("beer")
public class Beer implements Serializable {
    ...
    protected String officialSite;

    @Url
    public String getOfficialSite() {
        return this.officialSite;
    }
    ...
}
```



Action listeners

- JSF is event-driven, listens for actions
- User interacts with UI command component
  - button, link, etc.
- User action triggers method execution on server
- Action listener registered on component using method binding expression - #{beerAction.save}

Binding the action listener

editBeer.xhtml

```
<h:form id="beer">
  ...
  <div class="actionButtons">
    <h:commandButton id="save" value="Save"
      action="#{beerAction.save}"/>
  </div>
</h:form>
```

```
@Name("beerAction")
public class BeerAction {

    public String save() { ... }
}
```

Test first!

- Seam provides a test infrastructure for end-to-end testing
  - `Based on TestNG`
  - `Boots an embedded Java EE environment`
- Execute tests using `ant test`
- Instantiate JSF life cycle inside of test case
  - `NonFacesRequest` – emulates initial "GET" request
  - `FacesRequest` – emulates JSF "postback"
- Most seam-gen commands create a test stub

Defining the expected behavior

```
public class BeerActionTest extends SeamTest {
    @Test public void testAddBeer() throws Exception {
        new FacesRequest("/editBeer.xhtml") {
            protected void updateModelValues()
                throws Exception {
                setValue("#{beer.name}", "HopDevil");
            }
            protected void invokeApplication()
                throws Exception {
                invokeMethod("#{beerAction.save}");
            }
        }.run();
    }
}
```

Implementing the behavior

```
@Name("beerAction")
public class BeerAction {
    @In
    protected Beer beer;

    @In
    protected EntityManager;

    public String save() {
        entityManager.persist(beer);
        return null;
    }
}
```

Bijection

- Similar to dependency injection
  - container satisfies the needs of components
- Bijection occurs on *every* method invocation
- Prefix "bi" implies that it occurs twice
  - *injection* is applied before the method is invoked
  - *outjection* is applied after the method is invoked

Bijection = Injection + Outjection

Dynamic injection

- @In annotation applied to property of class
- Simple case
  - Seam looks for a context variable matching the name of the property
  - assigns value of context variable to property
- Additional criteria
  - customize context variable name and scope
  - set required=false to ignore missing value
  - set create=true to create value if not found

Exporting data

- @Out annotation applied to property of class
- Simple case
  - Value of property is assigned to a context variable whose name matches the name of the property
  - Scope of context variable determined as follows:
    - use scope of component with same name, if exists
    - otherwise, use scope of property's component
- Additional criteria
  - customize context variable name and scope
  - set required=false to ignore null value

What bijection buys you

- Simple to wire together objects and tap into managed services
- It's dynamic
  - Can inject a narrower-scoped component into a wider-scoped component
  - Injections are cleared after method call (disinjection)
- No manual assignment to make objects available to the view

Using bijection to implement the behavior

```
@Name("beerAction")
public class BeerAction {
    @In
    protected Beer beer;

    @In
    protected EntityManager;

    public String save() {
        entityManager.persist(beer);
        FacesMessages.instance().add("#{beer.name}");
        return "/home.xhtml";
    }
}
```

inject context variables

issues "redirect after post"

but JSF message is not lost

Where is the EntityManager defined?

- Seam automatically creates a new EntityManager on demand from JPA persistence unit
- Defined in components.xml
  - XML-based equivalent to @Name

```
<persistence:managed-persistence-context
  name="entityManager" auto-create="true"
  entity-manager-factory="#{entityManagerFactory}"/>

<persistence:entity-manager-factory
  name="entityManagerFactory"
  persistence-unit-name="connoisseur"/>
```

What about transactions?

- Seam wraps every JSF request in two transactions
  - one from start of request until after actions are invoked (*Invoke Application* phase)
  - one around the rendering of the response (*Render Response* phase)
- Works with both JTA and entity transactions
- @Transactional marks a method as transactional
  - don't need it as long as using JSF life cycle
  - required if you disable Seam global transactions

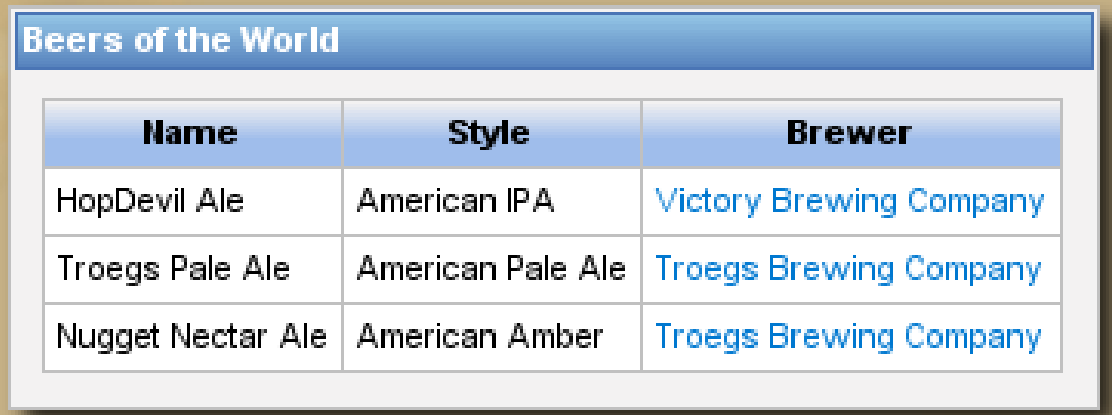
Context variables on demand

- One premise of component model is that data is "pulled" rather than "pushed"
- Instance of component is created when name is requested
- What about arbitrary data?
- Seam factory component
  - Provides a way to initialize *any* context variable when requested

Pulling down a list

beerList.xhtml

```
<rich:panel>
  <f:facet name="header">Beers of the world</f:facet>
  <rich:dataTable var="_beer" value="#{beers}"
    rendered="#{not empty beers}">
    <h:column>
      <f:facet name="header">Name</f:facet>
      #{_beer.name}
    </h:column>
    ...
  </rich:dataTable>
</rich:panel>
```



A context variable factory

```
@Name("beerList")
public class BeerList {
    @In
    protected EntityManager entityManager;

    @Factory("beers")
    public List<Beer> getBeers() {
        return entityManager
            .createQuery("select b from Beer b")
            .getResultList();
    }
}
```

Factory chain reaction

```
@Name("beerList")
public class BeerList {
    @In
    protected EntityManager entityManager;

    @Out
    protected List<Beer> beers;

    @Factory("beers")
    public void loadBeers() {
        beers = entityManager
            .createQuery("select b from Beer b")
            .getResultList();
    }
}
```

exported after factory method is called

Data model selection

- JSF UIData components support notion of row selection
  - `<h:dataTable>`
- Captured when action is triggered within row
- How does it work?
  - JSF `DataModel` wraps collection
  - JSF positions `DataModel#getRowData()` to row of action before calling action listener
  - Action listener must have access to `DataModel`

Transparent data model selection with Seam

```
@Name("beerList")
public class BeerList {
    ...
    @DataModel(scope = ScopeType.PAGE)
    protected List<Beer> beers;

    @DataModelSelection
    protected Beer selectedBeer;

    @Factory("beers")
    public void loadBeers() {
        beers = ...;
    }
}
```

"Clickable" lists

beerList.xhtml

```
<rich:dataTable var="_beer" value="#{_beers}"
  rendered="#{_beers.rowCount gt 0}">
  ...
  <h:column>
    <f:facet name="header">Action</f:facet>
    <s:link view="/editBeer.xhtml"
      action="#{beerAction.edit}" value="Edit">
      <f:param name="id" value="#{_beer.id}"/></s:link>
    <h:commandLink action="#{beerList.delete}"
      value="Delete"/>
  </h:column>
</rich:dataTable>
```

Beers of the World

Name	Style	Brewer	Action
HopDevil Ale	American IPA	Victory Brewing Company	Edit Delete
Troegs Pale Ale	American Pale Ale	Troegs Brewing Company	Edit Delete
Nugget Nectar Ale	American Amber	Troegs Brewing Company	Edit Delete

Processing the selected row

```
@Name("beerList")
public class BeerList {
    ...
    @DataModelSelection
    protected Beer selectedBeer;
    ...
    public String delete() {
        selectedBeer = entityManager
            .find(Beer.class, selectedBeer.getId());
        entityManager.remove(selectedBeer);
        return "/beerList.xhtml";
    }
}
```

Selecting a row for editing

```
@Name("beerAction")
public class BeerAction {
    ...
    @In(create = true) protected Beer beer;
    @RequestParameter protected Long id;
    ...
    public String edit() {
        beer = entityManager.find(Beer.class, id);
        return "/editBeer.xhtml";
    }
}
```

Checking out a record

- Need to track record's state
  - Create or edit mode
  - Unique id of record being modified, if edit mode
  - Lock status
- Traditionally done using hidden form field
  - Entity must be reloaded from database
  - Form data must be copied onto entity instance

Persistence context

- Persistence manager maintains reference to retrieved entities
  - In memory cache
  - Entity instances are "managed"
  - Guarantees identity of instances
  - Performs automatic dirty checking
- Stops working when closed
  - Entity instances become "detached"

Extended persistence context

- Persistence manager stored in conversation
- Lives across requests
- JSF applies form values to managed entity
- Dirty checking ensures update
- Automatic optimistic locking

Starting the conversation

```
@Name("beerAction")
@Scope(ScopeType.CONVERSATION)
public class BeerAction {
    ...
    @In(create = true) @Out protected Beer beer;
    @RequestParameter protected Long id;
    @Out protected boolean managed;

    @Begin
    public String edit() {
        beer = entityManager.find(Beer.class, id);
        managed = true;
        return "/editBeer.xhtml";
    }
}
```

Save or update?

editBeer.xhtml

```
<h:form id="beer">
  ...
  <div class="actionButtons">
    <h:commandButton action="#{beerAction.save}"
      rendered="#{not managed}"/>
    <h:commandButton action="#{beerAction.update}"
      rendered="#{managed}"/>
  </div>
</h:form>
```

Wrapping things up

```
@Name("beerAction")
@Scope(ScopeType.CONVERSATION)
public class BeerAction {
    ...

    @End
    public String update() {
        managed = false;
        FacesMessages().instance()
            .add("#{beer.name} has been updated.");
        return "/editBeer.xhtml";
    }
}
```

no explicit update instruction

Multi-record editing

```
@Name("beerList")
@Scope(ScopeType.CONVERSATION)
public class BeerList {
    ...
    @Out protected boolean editModeEnabled;

    @Begin
    public void editMode() {
        loadBeers();
        editModeEnabled = true;
    }
    ...
}
```

load records into persistence context

Multi-record editing

```
<rich:dataTable var="_beer" value="#{beers}" ...>
  <h:column>
    <f:facet name="header">Name</f:facet>
    <h:outputText value="#{_beer.name}"
      rendered="#{not editModeEnabled}"/>
    <h:inputText value="#{_beer.name}" required="true"
      rendered="#{editModeEnabled}"/>
  </h:column>
  ...
</rich:dataTable>
```

Multi-record editing

```
@Name("beerList")
@Scope(ScopeType.CONVERSATION)
public class BeerList {
    ...
    @Out protected boolean editModeEnabled;

    @End
    public void saveChanges() {
        editModeEnabled = true;
        FacesMessages.instance().add("Changes saved");
        return "/beerList.xhtml";
    }
    ...
}
```

Questions?
I know you've got 'em

- Now it's your turn to challenge me!

Resources

- Seam 2.0 books
  - Seam in Action, by Dan Allen
- Seam community site
  - http://www.seamframework.org
- Seam links (and lots of them)
  - http://del.icio.us/seaminaction

Thanks for coming!

Dan Allen

dan.allen@mojavelinux.com

http://www.mojavelinux.com