Introduction to JBoss Seam

Christian Bauer
christian@hibernate.org

Road Map

- **The standards: JSF and EJB 3.0**
- A Java EE web application example
- Analyzing the application
- Improving the application with Seam
- Seam feature highlights

JSF: The Big Picture

JavaServer Faces standardizes an event-driven framework for web applications:

- Extensible component model for UI widgets
- "Managed beans" for application logic
- Expression language for value and action listener binding
- Standardized event processing
- Navigation rules for coordination of page flow

EJB 3.0: The Big Picture

EJB 3.0 standardizes a programming model for transactional components:

- POJO-based - any class is an EJB component
- Stateless, stateful, and message-driven components
- Configuration by exception, metadata in annotations (preferred) or XML
- Dependency injection of managed components and other resources
- Declarative handling of cross-cutting concerns, e.g. transaction demarcation and security requirements
- Full object/relational mapping with the new Java Persistence API (JPA)

Road Map

- The standards: JSF and EJB 3.0
- **A Java EE web application example**
- Analyzing the application
- Improving the application with Seam
- Seam feature highlights

Let’s suppose we have some data...

A table in a SQL DBMS:

```
create table ITEM (
   ID           bigint not null primary key,
   NAME         varchar(100) not null unique,
   DESCRIPTION  varchar(1000),
   PRICE        decimal(10,2) not null
)
```

Our web application allows searching for items and modifying their values...

Map the table to an entity class

Using JPA annotations:

```
@Entity
public class Item {

  @Id @GeneratedValue
  private Long id;
  private String name;
  private String description;
  private BigDecimal price;

  // Constructor
  // Optional: Getter/setter method pairs
}
```

Surrogate key identifier attribute

If we put @Id on a field, no getter and setter methods are required, add them dependent on value mutability

The search page

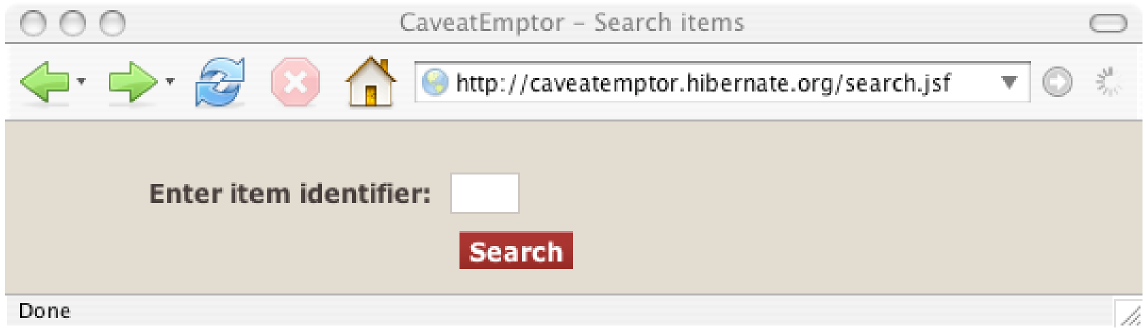
JSF widgets are bound to a backing bean with value- and action-binding expressions, referencing the backing bean by name:

A JSF-EL *value binding*

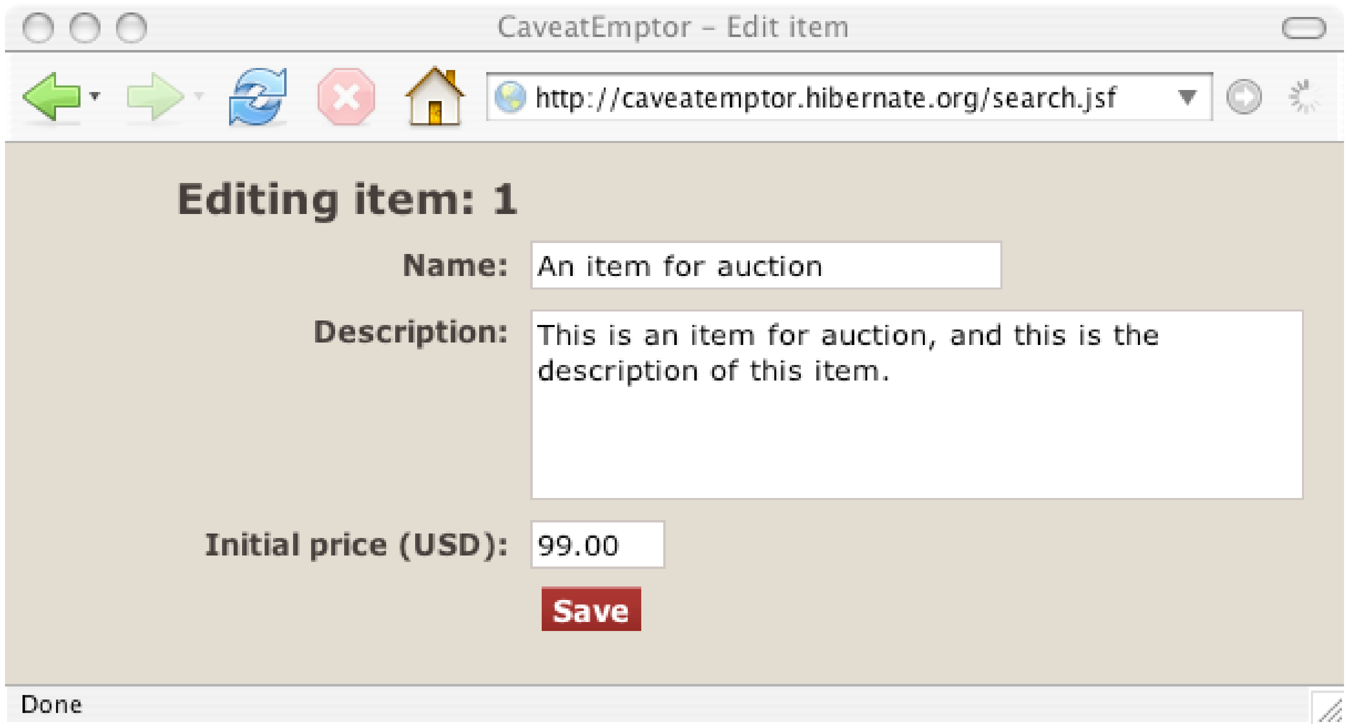
```
<h:form>
  Enter item identifier: <h:inputText value="#{itemEditor.id}"/>
  <h:commandButton value="Search" action="#{itemEditor.doGet}"/>
</h:form>
```

A JSF control

A JSF-EL *method binding*


CaveatEmptor – Search items
http://caveatemptor.hibernate.org/search.jsf
Enter item identifier:
Search
Done

The rendered edit page



CaveatEmptor - Edit item

http://caveatemptor.hibernate.org/search.jsf

Editing item: 1

Name:	An item for auction
Description:	This is an item for auction, and this is the description of this item.
Initial price (USD):	99.00

The edit page source

JSF supports validation and conversion, validation errors render messages:

```
<h:form>
  Editing Item:  <h:outputText value="#{itemEditor.id}/>
  Name:          <h:inputText value="#{itemEditor.title}">
                   <f:validateLength maximum="255"/>
                 </h:inputText>
  Description:   <h:inputText value="#{itemEditor.description}">
                   <f:validateLength maximum="4000"/>
                 </h:inputText>
  Initial Price (USD):
                 <h:inputText value="#{itemEditor.price}"/>
                   <f:convertNumber type="currency"
                                    pattern="$### ###.## USD"/>
                 </h:inputText>

  <h:messages/>
  <h:commandButton value="Save" action="#{itemEditor.doSave}"/>
</h:form>
```

JSF Validator

JSF Converter

Should we use a SLSB?

```
@Stateless
public class EditItemBean implements EditItem {

  @PersistenceContext
  EntityManager em;

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

  public Item save(Item item) {
    return em.merge(item);
  }
}
```

Container will prepare a pool of instances

Container will inject the EntityManager at call time

All method calls are wrapped in a system transaction, the persistence context is scoped to that transaction automatically

And a JSF backing bean?

```
public class ItemEditor {

  private Long id;
  private Item item;
  public Long getId() { return id; }
  public void setId(Long id) { this.id = id; }
  public String getTitle() { return item.getTitle(); }
  public void setTitle(String title) { item.setTitle(title); }
  // etc..

  @EJB EditItem editItemEJB;

  public String doGet() {
    item = editItemEJB.find(id);
    item == null ? return "notFound" : "success";
  }
  public String doSave() {
    item = editItemEJB.save(item);
    return "success";
  }
}
```

Value binding methods

Action binding methods

Action method outcome

Declare the bean in faces-config.xml

```
<faces-config>
  <managed-bean>
    <managed-bean-name>itemEditor</managed-bean-name>
    <managed-bean-class>caveatemptor.ItemEditor</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
</faces-config>
```

When an instance of *itemEditor* needs to be resolved, it either will be in the *session* context or a new instance is created and held in the session context

Hence, *itemEditor* is a **contextual variable** with a value managed by JSF

Declare navigation rules

```
<faces-config>
  <managed-bean>...</managed-bean>

  <navigation-rule>
    <from-view-id>/getItem.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/editItem.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>

  <navigation-rule>
    <from-view-id>/editItem.xhtml</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/getItem.xhtml</to-view-id>
    </navigation-case>
  </navigation-rule>

</faces-config>
```

Navigation rules map logical "outcomes" to view URLs

Road Map

- The standards: JSF and EJB 3.0
- A Java EE web application example
- **Analyzing the application**
- Improving the application with Seam
- Seam feature highlights

Much simpler code

- **Fewer artifacts:** No DTOs required
- **Less noise:** No Struts/EJB 2.x boilerplate code
- **More transparent:** No direct calls to HttpSession or HttpRequest
- **Much simpler ORM:** Even compared to Hibernate API!
- **Finer grained components:** Clean MVC

More powerful for complex problems

- JSF is amazingly flexible and extensible
- Custom sets of UI widgets are available, even with AJAX support
- EJB 3.0 supports interceptors for "AOP lite"
- Powerful object/relational mapping, far beyond EJB 2.x CMP entity beans
- All components (except the views) are easily testable with TestNG or JUnit

The JSF backing bean is just noise

- The component with the most lines of code
- Brittle, every change of view or the application logic requires change of backing bean code
- Looks like it decouples layers but in fact the layers are more coupled together than they should be

No multi-window support

1. Open the search view in one browser window, look up an item and start editing it
2. Open the search view in a second browser window, the item you look up will overwrite the state in the session
3. If you now save the item in the first window, you are actually saving the item you loaded in the second window - without realizing it!

Fixing this simple bug (today this is considered a bug) is a major architectural change!

Application leaks memory

1. Open the search view in one browser window, look up an item and start editing it
2. Save your changes - back on the search page

The *ItemEditor* instance in the session variable *itemEditor* will only be cleaned up when the session is destroyed

Now imagine an application with many session-scoped backing beans and many forms... extremely difficult to fix category of bugs

Flow is weakly defined

- If you want to know where the click on "Find" will take you, how many files do you have to look at?
- No tight control over user navigation, totally ad-hoc (back button, bookmarking)
- How do you tie this flow into the overall long-running business process? Maybe editing an item was just a small task in a larger review process...

Too much and bad XML metadata

- The faces-config.xml is clunky - Sun still doesn't know how to use XML attributes
- This metadata is much better defined in annotations - after all, how often does the role of a component change without any code change?

Road Map

- The standards: JSF and EJB 3.0
- A Java EE web application example
- Analyzing the application
- **Improving the application with Seam**
- Seam feature highlights

JBoss Seam Goals

- Unify EJB and JSF component models
- Deprecate so-called stateless architecture
- Integrate BPM - workflow and business process management for the masses
- Decouple technology from the execution environment - rely on standard runtimes
- Enable richer user experience - AJAX, multi-window web applications

Adding Seam to the application

Minor change to the edit page:

Now referencing the entity properties!

```
<h:form>
  Editing Item:  <h:outputText value="#{itemEditor.id}/>
  Title:         <h:inputText value="#{itemEditor.item.title}">
                   <f:validateLength maximum="255"/>
                 </h:inputText>
  Description:   <h:inputText value="#{itemEditor.item.description}">
                   <f:validateLength maximum="4000"/>
                 </h:inputText>
  Price:         <h:inputText value="#{itemEditor.item.price}"/>
                   <f:convertNumber type="currency"
                                    pattern="$### ###.## USD"/>
                 </h:inputText>

  <h:messages/>

  <h:commandButton value="Save" action="#{itemEditor.doSave}"/>
</h:form>
```

Our first Seam component!

```
@Name("itemEditor")
@Stateful
public class EditItemBean implements EditItem {

  @PersistenceContext EntityManager em;

  private Long id;
  private Item item;
  // Getter and setter pairs

  @Begin
  public void doGet(Long id) {
    item = em.find(Item.class, id);
    item == null ? return "notFound" : "success";
  }

  @End @Destroy @Remove
  public void doSave(Item item) {
    item = em.merge(item);
    return "success";
  }
}
```

Seam component name is the contextual variable name

Component is stateful - value bindings to component properties

Begin a *Conversation* - hold variables across requests

End a *Conversation* - destroy variables when this method returns

After adding Seam

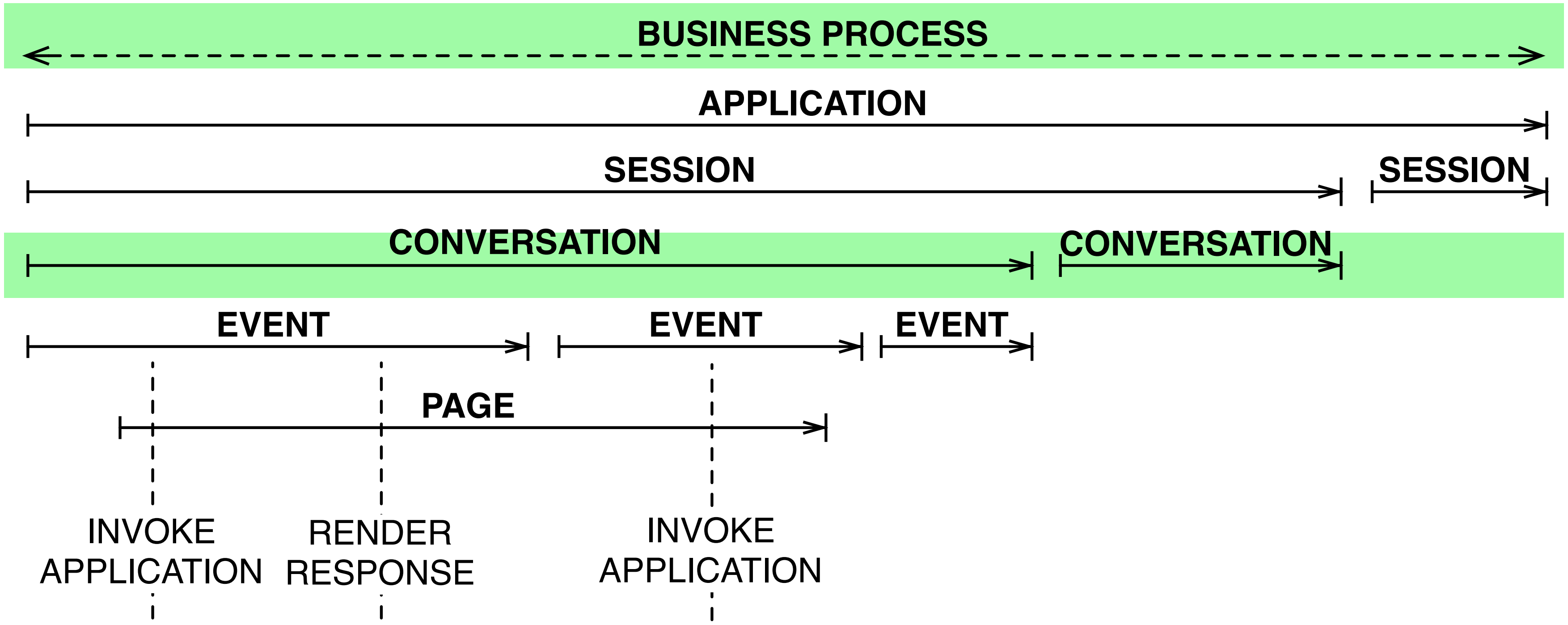
- The JSF backing bean is gone, the Seam component is now referenced with *itemEditor*
- An instance will be created by Seam and held automatically in the new logical conversation context, either for a single or several requests
- The conversation is promoted when a method with a *@Begin* annotation is called and demoted and destroyed when a method with an *@End* annotation returns
- Conversational state is handled properly, application does not leak memory into the HttpSession and now works in several browser windows (parallel conversations)

The Seam context model

- EVENT
- PAGE
- **CONVERSATION**
- SESSION
- **BUSINESS PROCESS**
- APPLICATION

The highlighted “logical” contexts are demarcated by application code or metadata

Hierarchical stateful contexts



State management

- How is state stored between requests?
  - different strategies for each context
- Conversation context
  - Segmented HttpSession + conversation timeout
  - *org.jboss.seam.core.init.conversationTimeout*
- Page context
  - stored in JSF ViewRoot (component tree) of the page
  - might be serialized to client if JSF is configured for client-side state saving, otherwise in HttpSession
- Business Process context
  - must be persistent in the database, handled by jBPM

Seam component examples

```
@Entity
@Name("item")
public class Item { ... }
```

```
@Stateful
@Name("itemEditor")
public class ItemEditorBean implements ItemEditor { ... }
```

```
@Name("itemEditor")
@Scope(ScopeType.CONVERSATION)
public class ItemEditor { ... }
```

```
@Entity
@Name("user")
@Roles({
  @Role(name = "currentUser", scope = ScopeType.SESSION)
})
public class User { ... }
```

Dependency Injection

- Traditional dependency injection is broken for stateful applications
  - 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
  - simple replacement for factories and JNDI lookups

"Bijection"

- Looking for a better name!
- Stateful applications need wiring that is:
  - 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?

The @In and @Out annotations trigger automatic wiring at component call-time:

```
@Name("changePassword")
public class ChangePasswordAction {

  @PersistenceContext
  private EntityManager em;

  @In
  @Out
  private User currentUser;

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

Inject the value of the contextual variable named "currentUser" every time this component is invoked - search all contexts hierarchically

Take the value of the field and set it on the contextual variable "currentUser" every time this component is invoked - the context is the default or defined context for the "currentUser" component

Pageflow

- Two models for conversational pageflow
  - **The stateless model:** JSF navigation rules, either in faces-config.xml or in pages.xml
    - ad hoc navigation (app handles back button)
    - 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 need the stateless model, some applications need both models

What about business process?

- Not the same as 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 a **task**
  - use @BeginTask to begin a conversation that completes work in the business process
    - that means a task is a special kind of conversation!

The Persistence Context

- Java Persistence/EJB3 has a *Persistence Context*
  - think "a HashMap of all objects I loaded and stored through an EntityManager"
  - guarantees integrity and avoids data aliasing problems: at most one in-memory object for each database row while the PC is active
  - is also a natural first-level cache
  - can do dirty checking of objects and write SQL as late as possible (automatic or manual flushing)

- EJB3 Persistence Contexts have a flexible scope
  - default: scope is same as system transaction (JTA)
  - optional: extended PC bound to stateful session bean

Transaction-scoped or extended PC?

- A transaction-scoped persistence context has problems if you re-use detached objects
  - *LazyInitializationException* if you navigate unloaded associations or iterate through unloaded collections
  - *NonUniqueObjectException* if you reattach detached instances into a new PC that already contains an instance with the same identifier (merging helps)
  - Less opportunity for caching (traditional workaround: enable the Hibernate second-level cache...)
- An extended persistence context of a SFSB is
  - not available during view rendering (LIE again)
  - very complicated rules when a PC is propagated into bean calls, depending on the system transaction

Seam: Conversation-scoped PC

Let Seam handle the persistence context scope:

```
@Name("itemEditor")
@Stateful
public class ItemEditorBean implements ItemEditor {

  @In
  private EntityManager em;

  @Begin(flushMode=FlushModeType.MANUAL)
  public Item getItem(Long itemId) {
    return em.find(Item.class, itemId);
  }

  public void processItem(Item item) {
    item.getCategories().iterator().next();
  }

  @End public void confirm() { em.flush(); }
}
```

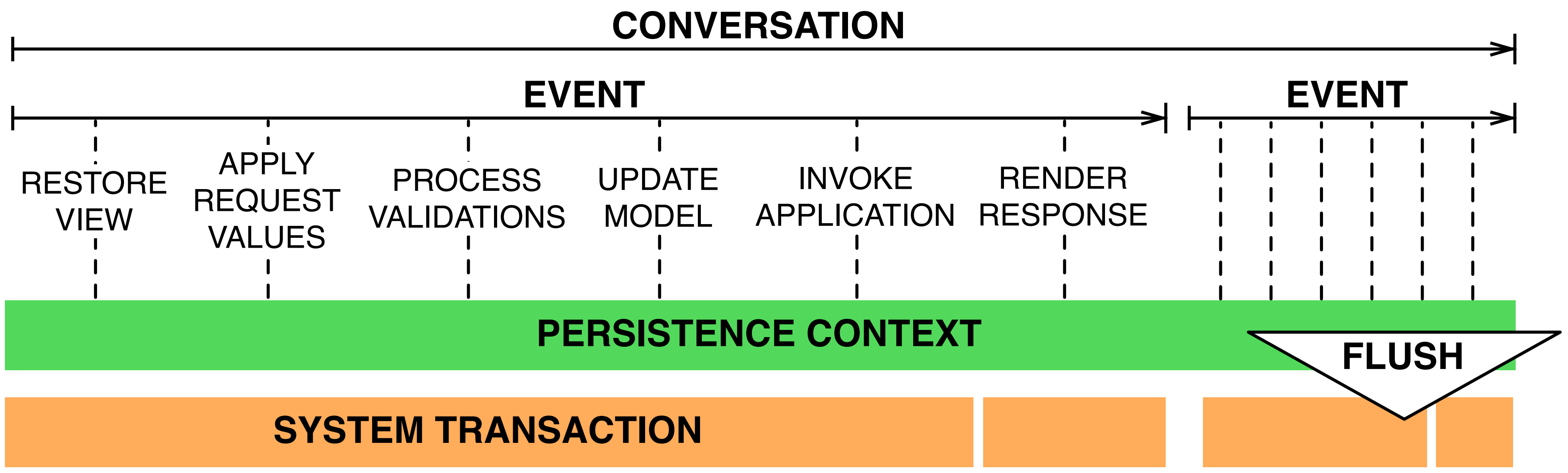
Seam looks up "em" for you

The "item" remains persistent 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

Seam manages PC and TX



Model-based constraints

- Validation belongs in the user interface
  - or does it?
- Most "validations" reflect integrity rules that also appear in the database
  - in fact, most business rules have integrity rules that should be represented with some database constraint
- 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 definition

Hibernate Validator annotations

```
@Entity
public class Item {

  @Id @GeneratedValue
  private Long id;

  @org.hibernate.validator.Length(min=3, max=100)
  @org.hibernate.validator.Pattern(regex="[a-zA-z0-9]")
  private String title;

  @org.hibernate.validator.Length(
    min=3,
    max=1000,
    msg = "The description must be between 3 and 1000 characters!"
  )
  private String description;

  private BigDecimal price;

  // Constructor
  // Optional: Getter/setter method pairs
}
```

Could also reference a message bundle key

Hibernate Validator supports...

- Many built-in validators
  - Max, Min, Length, Range, Size, Email, Future, Past, Pattern, Email, CreditCard, ...
  - Easy to write custom validators
- Validation and message/error display with Seam UI components for JSF
- Validation can be triggered programmatically on objects, throws *InvalidStateException* with an array of invalid properties
- Works with every JPA provider, if used with Hibernate it generates SQL DDL constraints you can use in your database schema

Road Map

- The standards: JSF and EJB 3.0
- A Java EE web application example
- Analyzing the application
- Improving the application with Seam
- **Seam feature highlights**

Seam Security: Authorization

```
<page view-id="/comment.xhtml">
  <restrict/>
  ...
</page>
```

Permission name: *//comment.xhtml*
Permission action: *view*

```
@Entity
class name BlogEntry {
   @PrePersist @Restrict
   public void prePersist() {}
   ...
```

Permission name: *pgk.BlogEntry*
Permission action: *insert*

```
@Name("blog")
class name BlogAction {
   @Begin @Restrict
   public void createComment() {
   ...
```

Permission name: *blog*
Permission action: *createComment*

```
Identity.instance().checkRestriction(
    "#{s:hasPermission('friendComment', 'create', friends)}"
);
```

Added to rules working memory

```
<s:span rendered="#{s:hasPermission('blog','createComment', null)}">
   <s:link view="/comment.seam" value="Add Comment" propagation="none"/>
</s:span>
```

Seam observer/observable pattern

```
@Name("hotelBooking")
class HotelBookingAction {

   @End
   public String confirm() {
      em.persist(booking);
      Events.instance().raiseEvent("bookingConfirmed");
      return "confirmed";
   }
```

Events can carry a payload but it's easier to outject/inject values

```
@Name("bookingList")
class BookingListAction
   @Factory("bookings")
   @Observer("bookingConfirmed")
   public void refreshBookings() {
      bookings = em.createQuery...
   }
```

You can also register listeners in *components.xml* or listen to any of the Seam built-in events

Starting asynchronous procedures

```
@Local
public class PaymentProcessor {

    @Asynchronous
    public void schedulePayment(@Expiration Date when,
                                @IntervalDuration long interval,
                                Payment payment);
}
```

Annotate the interface

```
@Name("payAction")
public class PayAction {
    @In PaymentProcessor paymentProcessor;
    @In Payment newPayment;

    public void schedule() {
        paymentProcessor.schedulePayment(
          payment.getPaymentDate(),
          payment.getFrequency().interval(),
          payment
        );
    )
}
```

Processing is transparent to controller!

Firing asynchronous Seam events

```
@Name("hotelBooking")
class HotelBookingAction {

   @End
   public String confirm() {
      em.persist(booking);
      Event.instance().raiseAsynchronousEvent("bookingConfirmed");

      // Date when = new Date(...);
      // long interval = ...;
      // Event.instance().raiseTimedEvent(
      //    "bookingConfirmed", when, interval
      // );
      return "confirmed";
   }
```

We will later poll the observers for their state

Remember: Only the BUSINESS PROCESS context is propagated into asynchronous methods, new EVENT and CONVERSATION context

Publish a JMS object message

Use the Seam helper components:

```
@In TopicPublisher stockTickerPublisher;
@In TopicSession topicSession;

public void publish(StockPrice price) {
  try {
    topicPublisher.publish( topicSession.createObjectMessage(price) );
  } catch (Exception ex) {
    throw new RuntimeException(ex);
  }
}
```

Working with JMS queues

Use the Seam helper components:

```
@In QueueSender paymentQueueSender;
@In QueueSession queueSession;

public void publish(Payment payment) {
  try {
    paymentQueueSender.send(
      queueSession.createObjectMessage(payment)
    );
  } catch (Exception ex) { throw new RuntimeException(ex); }
}
```

To receive JMS messages, write a message-driven EJB and turn it into a Seam component, or subscribe with JavaScript and Seam Remoting

Sending e-mails with Seam

Configure the Seam component:

```
<mail:mail-session host="my.smarthost.com" port="25"/>
```

Write a Facelets template:

```
<m:message xmlns:m="http://jboss.com/products/seam/mail">
    <m:from name="Seam" address="do-not-reply@jboss.com" />
    <m:to name="#{person.namae}">#{person.address}</m:to>
    <m:subject>Plain text e-mail sent by Seam</m:subject>
    <m:body type="plain">Dear #{person.firstname},
This is a simple, plain text, e-mail.
    </m:body>
</m:message>
```

Use expressions to access contextual variables!

Send the email by rendering the template:

```
Renderer.instance().render("/mailTemplate.xhtml");
```

Supports HTML, attachments, etc...

Rendering PDFs with Seam/iText

Just write a Facelets page and open it:

```
<p:document xmlns:ui="http://java.sun.com/jsf/facelets"
            xmlns:f="http://java.sun.com/jsf/core"
            xmlns:p="http://jboss.com/products/seam/pdf"
            title="Example PDF" keywords="mykeyword"
            subject="seam" author="Seam Team"
            creator="Seam PDF example app">

    <f:facet name="header">
        <p:font size="12">
            <p:footer>My Footer [<p:pageNumber />]</p:footer>
        </p:font>
    </f:facet>
    <p:paragraph alignment="justify">
       You bought #{shoppingCart.size} items:
    </p:paragraph>
    <p:image value="/jboss.jpg" />
...
```

Use expressions to access contextual variables!

Seam Remoting for JavaScript

- 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

Calling a remote method with JS

Make a method "remotable":

```
@Local
public interface HelloLocal {
   @WebRemote public String sayHello(String name);
}
```

Implementation is nothing special, returns a String

Call it from JavaScript, passing in your handler:

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

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

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

Imports the JavaScript "interface" via the Seam resources servlet

New features in Seam 2.0

- Currently in beta, GA in Summer 2007
- Seam components can be Webservice endpoints
- Seam components can be written in Groovy
- Seam core is now independent of JSF
- Experimental support for GWT
- Integration of Hibernate Search
- Extensions to the unified EL
- Better async processing (Quartz integration)
- Decoupled transaction layer from JTA
- Redesign of JSF components (CDK, Exadel)
- ... much more

Summary

- Seam has way too many features :)
- More features:
  - Many useful JSF components
  - Page fragment caching
  - i18n and message bundle handling
  - Switchable UI theme architecture
  - JBoss Rules integration for business logic handling
  - Wiki text parser and renderer
  - Unit testing support with mock infrastructure
  - Spring integration for migration to stateful apps
- Try the example applications (> 25!) in Seam