Introduction to Web Beans

Gavin King
gavin@hibernate.org
http://in.relation.to/Bloggers/Gavin

Java

Sun microsystems

Goals

- Web Beans provides a unifying component model for Java™ EE 6 platform, by defining:
  - A programming model for stateful, contextual components compatible with EJB™ 3.0 software and JavaBeans™ architecture
  - An extensible context model
  - Component lookup, injection and EL resolution
  - Conversations
  - Lifecycle and method interception
  - An event notification model
  - Persistence context management for optimistic transactions
  - Deployment-time component overriding and configuration
  - Integration with JavaServer™ Faces platform, Java Servlet API, Java Persistence API (JPA) and Common Annotations for the Java Platform

Target environment

- Should Web Beans be compatible with Java SE platform?
- Java EE platform now has “profiles”
  - what profile should we target?
- We won’t target a specific platform
  - instead, we will explicitly define which features depend upon the availability of other specifications in the runtime environment

Migration

- Any existing EJB3 software session bean may be made into a Web Bean by adding annotations
- Any existing JavaServer Faces platform managed bean may be made into a Web Bean by adding annotations
- New Web Beans may interoperate with existing EJB3 software session beans
  - via `@EJB` or Java Naming and Directory Interface (JNDI)
- New EJB software may interoperate with existing Web Beans
  - Web Beans injection and interception supported for *all* EJB software

What's different about Web Beans?

- The theme of Web Beans: *loose coupling with strong typing!*
  - decouple server and client via well-defined APIs and "binding types"
    - server implementation may be overridden at deployment time
  - decouple lifecycle of collaborating components
    - components are contextual, with automatic lifecycle management
    - allows stateful components to interact like services
  - decouple orthogonal concerns
    - via interceptors
  - completely decouple message producer from consumer
    - via events
- Web Beans unifies the "web tier" with the "enterprise tier"
  - a single component may access state associated with the web request, and state held by transactional resources

What is a Web Bean?

- Kinds of components:
  - Any Java platform class
  - EJB software session and singleton beans
  - Resolver methods
  - Java Message Service (JMS) components
  - Remote components
- Essential Ingredients:
  - Deployment type
  - API types
  - Binding types
  - Name
  - Implementation

Simple Example

- A simple component:

```
public
@Component
class Hello {

   public String hello(String name) {
      return “hello “ + name;
   }

}
```

Simple Example

- A simple client

```
public
@Component
class Printer {

   @Current Hello hello;

   public void hello() {
      System.out.println( hello.hello("world") );
   }

}
```

Simple Example

- Or, using constructor injection

```
public
@Component
class Printer {

   private Hello hello;

   public Printer(Hello hello) { this.hello=hello; }

   public void hello() {
      System.out.println( hello.hello("world") );
   }

}
```

Simple Example

- Or, using initializer injection

```
public
@Component
class Printer {

   private Hello hello;

   @Initializer
   initPrinter(Hello hello) { this.hello=hello; }

   public void hello() {
      System.out.println( hello.hello("world") );
   }

}
```

Component names

- A named component:

```
public
@Component
@Named(“hello”)
class Hello {

   public String hello(String name) {
      return “hello “ + name;
   }

}
```

Simple Example

- Unified EL client

```
<h:commandButton value="Say Hello"
                 action="#{hello.hello}"/>
```

Binding types

- A binding type is an annotation that lets a client choose between multiple implementations of an API
  - Binding types replace lookup via string-based names
  - `@Current` is the default binding type

Binding types

- Define a binding type:

```
public
@BindingType
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
@interface Casual {}
```

Binding types

- Same API, different implementation

```
public
@Casual
@Component
class Hi extends Hello {

   public String hello(String name) {
      return “hi ” + name;
   }

}
```

Binding types

- A client of the new implementation

```
public
@Component
class Printer {

   @Casual Hello hello;

   public void hello() {
      System.out.println( hello.hello("JBoss Compass") );
   }

}
```

Deployment types

- A deployment type is an annotation that identifies a class as a deployed Web Bean
  - Deployment types may be enabled or disabled, allowing whole sets of components to be easily enabled or disabled at deployment time
  - Deployment types have a precedence, allowing the container to choose between different implementations of an API
  - Deployment types replace verbose XML configuration documents
- Default deployment type: `Production`

Deployment types

- Define a custom deployment type:

```
public
@DeploymentType
@Retention(RUNTIME)
@Target({TYPE, METHOD})
@interface Espanol {}
```

Deployment types

- Same API, once again:

```
public
@Espanol
@Component
class Hola extends Hello {

   public String hello(String name) {
      return “hola ” + name;
   }

}
```

Component types

- Implementation depends upon which component types are enabled:

```
<web-beans>
   <component-types>
      <component-type>javax.webbeans.Standard</component-type>
      <component-type>javax.webbeans.Production</component-type>
      <component-type>org.jboss.i18n.Espanol</component-type>
   </component-types>
</web-beans>
```

Scopes and contexts

- Extensible context model
  - A scope type is an annotation
  - A context implementation can be associated with the scope type
- Dependent scope, `@Dependent`
- Built-in scopes:
  - Any servlet
    - `@ApplicationScoped`, `@RequestScoped`, `@SessionScoped`
  - JavaServer Faces platform requests
    - `@ConversationScoped`
  - Web service request, Java RMI calls...
- Custom scopes

Scopes

- A session-scoped component

```
public
@SessionScoped
@Component
class Login {

   private User user;

   public void login() {
      user = ...;
   }

   public User getUser() { return user; }

}
```

Scopes

- The client does not need to know the lifecycle of the session-scoped component

```
public
@Component
class Printer {

   @Current Hello hello;
   @Current Login login;

   public void hello() {
      System.out.println(
         hello.hello( login.getUser().getName() ) );
   }

}
```

Conversation context

- Spans multiple requests
- “Smaller” than session
- Allows multi-window / multi-tab operation
- Corresponds to an optimistic transaction
  - conversation-scoped managed persistence context
  - solves problems with optimistic locking and lazy fetching

Conversation context

- The conversation context is demarcated by the application

```
public
@ConversationScoped
@Component
class ChangePassword {

   @UserDatabase EntityManager em;
   @Current Conversation conversation;
   private User user;

   public User getUser(String userName) {
      conversation.begin();
      user = em.find(User.class, userName);
   }

   public User setPassword(String password) {
      user.setPassword(password);
      conversation.end();
   }
}
```

Custom scopes

- After this, the hard work begins!

```
public
@ScopeType
@Retention(RUNTIME)
@Target({TYPE, METHOD})
@interface BusinessProcessScoped {}
```

EJB software in the web tier

- JavaServer Faces platform form

```
<h:form>
   Old password: <h:inputText value="#{changePassword.old}"/>
   New password: <h:inputText value="#{changePassword.new}"/>
   <h:commandButton value="Change Password"
                    action="#{changePassword.update}"/>
</h:form>
```

EJB software in the web tier

```
public
@RequestScoped
@Stateful
@Named
@Component
class ChangePassword {

   @UserDatabase EntityManager em;
   @Current User user;

   private String old;
   private String new;

   public void setOld(String old) { this.old=old; }
   public void setNew(String new) { this.new=new; }

   public void update() {
      if ( user.getPassword().equals(old) ) {
         user.setPassword(new);
         em.merge(user);
      }
   }
}
```

Producer methods

- Producer methods allow control over the production of a component instance
  - For runtime polymorphism
  - For control over initialization
  - For Web-Bean-ification of classes we don't control
  - For further decoupling of a "producer" of state from the "consumer"

Producer methods

- Simple producer method

```
public
@SessionScoped
@Component
class Login {

   private User user;

   public void login() {
      user = ...;
   }

   @Produces
   User getUser() { return user; }

}
```

Producer methods

- Producer method components may have a scope

```
public
@RequestScoped
@Component
class Login {

   private User user;

   public void login() {
      user = ...;
   }

   @Produces @SessionScoped
   User getUser() { return user; }

}
```

Producer methods

- No more dependency to `Login`!

```
public
@Component
class Printer {

   @Current Hello hello;
   @Current User user;

   public void hello() {
      System.out.println(
         hello.hello( user.getName() ) );
   }

}
```

Interceptors

- The package `javax.interceptor` defines method and lifecycle interception APIs
  - this is good stuff, except for the use of `@Interceptors(...)` to bind interceptors directly to a component
- Interceptor should be completely decoupled from component
  - via semantic annotations
- Interceptor classes should be deployment-specific
  - disable transaction and security interceptors during testing
- Interceptor ordering should be defined centrally

Interceptor binding types

- Define an interceptor binding type:

```
public
@InterceptorBindingType
@Retention(RUNTIME)
@Target({TYPE, METHOD})
@interface Secure {}
```

Interceptor binding types

- Interceptor implementation

```
public
@Secure
@Interceptor
class SecurityInterceptor {

   @AroundInvoke
   public Object aroundInvoke(InvocationContext ctx) {
      ...
   }

}
```

Interceptor binding types

- Class-level interceptor

```
public
@Secure
@Component
class Hello {

   public String hello(String name) {
      return “hello “ + name;
   }

}
```

Interceptor binding types

- Method-level interceptor

```
public
@Component
class Hello {

   @Secure
   public String hello(String name) {
      return “hello “ + name;
   }

}
```

Interceptor binding types

- Multiple interceptors

```
public
@Transactional
@Component
class Hello {

   @Secure
   public String hello(String name) {
      return “hello “ + name;
   }

}
```

Interceptors

- Interceptor ordering and enablement:

```
<web-beans>
   <interceptors>
      <interceptor>
          org.jboss.secure.SecurityInterceptor
      </interceptor>
      <interceptor>
          org.jboss.tx.TransactionInterceptor
      </interceptor>
   </interceptors>
</web-beans>
```

Reusing interceptor bindings

- Interceptor binding types may be applied to other interceptor binding types

```
public
@Secure
@Transactional
@InterceptorBindingType
@Retention(RUNTIME)
@Target(TYPE)
@interface Action {}
```

Interceptor binding types

- Multiple interceptors

```
public
@Action
@Component
class Hello {

   public String hello(String name) {
      return “hello “ + name;
   }

}
```

Stereotypes

- It is not only interceptor bindings we want to reuse!
- We have common architectural "patterns" in our application, with recurring component roles
  - Capture the roles using stereotypes
- A *stereotype* packages:
  - A default deployment type
  - A default scope
  - A set of interceptor bindings
  - Restrictions upon allowed scopes
  - Restrictions upon the Java platform type
  - May specify that components have names by default
- Built-in stereotypes: `@Component`, `@Model`

Stereotypes

- Defining a new stereotype:

```
public
@Secure
@Transactional
@RequestScoped
@Named
@Production
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
@interface Action {}
```

Stereotypes

- Using a stereotype:

```
public
@Action
class Hello {

   public String hello(String name) {
      return “hello “ + name;
   }

}
```

Events

- Event producer:

```
public
@Component
class Hello {

   @Observable Event<Greeting> helloEvent;

   public String hello(String name) {
      helloEvent.fire( new Greeting("hello " + name) );
   }

}
```

Events

- Event consumer:

```
public
@Component
class Printer {

   void onHello(@Observes Greeting greeting) {
      System.out.println(greeting);
   }

}
```

Events

- Event producer:

```
public
@Component
class Hi {

   @Observable @Casual Event<Greeting> helloEvent;

   public String hello(String name) {
      helloEvent.fire( new Greeting("hi " + name) );
   }

}
```

Events

- Event consumer:

```
public
@Component
class Printer {

   void onHello(@Observes @Causal Greeting greeting) {
      System.out.println(greeting);
   }

}
```

More information

- Web Beans EDR:
  - http://www.jcp.org/en/jsr/detail?id=299
- Blog:
  - http://in.relation.to/Bloggers/Everyone/Tag/Web+Beans
- Seam:
  - http://jboss.com/products/seam
- Guice:
  - http://code.google.com/p/google-guice/