Introduction to Web Beans

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

Goals

- Web Beans provides a unifying component model for Java EE 6, by defining:
  - A programming model for stateful, contextual components compatible with EJB 3.0 and JavaBeans
  - 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

Platform integration

- Web Beans may be EJB 3.0 session beans
    - to take advantage of EJB declarative transactions, security, etc.
- Web Beans may be used seamlessly from JSF
    - as a replacement for JSF managed beans
    - request, session, application, conversation contexts
- Web Beans are usable from servlets
    - request, session, application contexts
- Web Beans reuses Common Annotations and **`javax.interceptor`**
- Web Beans will integrate tightly with JPA
    - conversation-scoped extended persistence contexts

Migration

- Any existing EJB3 session bean may be made into a Web Bean by adding annotations
- Any existing JSF managed bean may be made into a Web Bean by adding annotations
- New Web Beans may interoperate with existing EJB3 session beans
  - via **`@EJB`** or JNDI
- New EJBs may interoperate with existing Web Beans
  - Web Beans injection and interception supported for *all* EJBs
- New Web Beans may interoperate with existing JSF managed beans
  - exact annotation still under discussion

SE vs. EE

- The core component model of Web Beans has been architected to have no hard dependency upon EJB or JSF
  - For testing and code reuse outside container
  - Due to pressure from some Google and community to support Java SE usecases...
  - We need further guidance from Sun and the JCP on this!

The theme of Web Beans

- *Loose coupling with strong typing!*
    - Stateful components interact as if they were services
    - Everything built around Java types, no strings hiding under the covers, waiting to bite you when something changes

What’s different about Web Beans?

- How do we achieve loose coupling?
  - 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 class
  - EJB session and singleton beans
  - Resolver methods
  - JMS components
  - Remote components
- Essential Ingredients:
  - Component type
  - API type
  - Binding types (optional)
  - 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”) );
   }

}
```

Simple Example

- Unified EL client

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

Component types and binding types

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

```
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(“SVJUG”) );
    }

}
```

Component types

```
public
@ComponentType
@Retention(RUNTIME)
@Target({TYPE, METHOD})
@interface Spanish {}
```

Component types

- Same API, once again:

```
public
@Spanish
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.Component</component-type>
      <component-type>org.jboss.i18n.Spanish</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`**
  - JSF requests
    - **`@ConversationScoped`**
  - Web service request, RMI calls...
- Custom scopes

Scopes

```
public
@SessionScoped
@Component
class Login {

   private User user;

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

   public User getUser() { return user; }

}
```

Scopes

```
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

```
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 {}
```

EJBs in the web tier

- JSF 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>
```

EJBs in the web tier

```
public
@RequestScoped
@Stateful
@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

```
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

```
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;
   }

}
```

Proposal: enhanced component types

- Still under discussion in the EG!

```
public
@Secure
@Transactional
@RequestScoped
@ComponentType
@Retention(RUNTIME)
@Target({TYPE, METHOD})
@interface Action {}
```

Enhanced component types

- Determines scope and interceptors

```
public
@Action
class Hello {

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

}
```

Enhanced component types

- Rules for multiple component types

```
public
@Mock @Action @Dao
class MockUserManager {

    ...

}
```

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

- EDR out tomorrow!
  - **http://jcp.org/en/jsr/detail?id=299**
- Blog:
  - **http://in.relation.to/Bloggers/GavinsBlog/Tag/Web+Beans**