Web Beans

Pete Muir

JBoss, a division of Red Hat

http://in.relation.to/Bloggers/Pete

pete.muir@jboss.org

Road Map

- Background
- Concepts
- Status

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

Goals

- Lifecycle and method interception
- An event notification model
- Persistence context management for optimistic transactions
- Deployment-time component overriding and configuration
- Integration with JSF, Servlets, JPA and Common Annotations

Target Environment

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

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

Theme of Web Beans: Loose Coupling with Strong Typing

- decouple server and client via well-defined APIs and "binding types"
  - implementation may be overridden at deployment time
- decouple lifecycle of collaborating components
  - components are contextual, with lifecycle management
  - allows stateful components to interact like services
- decouple orthogonal concerns
  - via interceptors
- decouple message producer from consumer
  - via events

Seam?

- Seam 3 will be built on the Web Beans core
- Web Beans will provide
  - Contextual programming model and Event Bus
  - Integration with JSF and EJB3
  - Integraton with JPA,Transactions and Bean Validation
- Seam will provide
  - Security
  - BPM & Rule integration
  - PDF and Mail JSF libraries
  - and everything else...

Road Map

- Background
- Concepts
- Status

What is a Web Bean?

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

Simple Example: Component

```
public
@Component
class Hello {
    public String hello(String name) {
        return "hello" + name;
    }
}
```

`@Component` is a built in stereotype

Simple Example: Client

```
public
@Component
class Printer {
    @Current Hello hello;
    public void hello() {
        System.out.println( hello.hello("world") );
    }
}
```

`@Current` is a built in binding type

Simple Example: 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") );
    }
}
```

Constructors are injected by default; `@Current` is the default binding type

Simple Example: Initializer injection

```
public
@Component
class Printer {
    private Hello hello;
    @Initializer
    void initPrinter(Hello hello) { this.hello=hello; }
    public void hello() {
        System.out.println( hello.hello("world") );
    }
}
```

Or you can use a post-creation callback, again with parameter injection

Component Names

```
public
@Component
@Named("hello")
class Hello {
    public String hello(String name) {
        return "hello" + name;
    }
}
```

By default components aren't available through EL. There is a default name used, if none is specified

JSF Page

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

Calling an action on a Web Bean through EL

Binding Types

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

Define a binding type

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

Creating a binding type is really easy!

Using a binding type

```
public
@Casual
@Component
class Hi extends Hello {
    public String hello(String name) {
        return "hi" + name;
    }
}
```

We're still using the `@Component` stereotype. We also specify the `@Casual` binding type (in addition to the implicit `@Current`)

Using a binding type

```
public
@Component
class Printer {
    @Casual Hello hello;
    public void hello() {
        System.out.println( hello.hello("JBoss Compass") );
    }
}
```

Here we inject the `Hello` component, and require an implementation which is bound to `@Casual`

Deployment Types

- A deployment type is an annotation that identifies a class as a 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

Create a deployment type

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

Using a deployment type

```
public
@Espanol
@Component
class Hola extends Hello {
    public String hello(String name) {
        return "hola " + name;
    }
}
```

Same API, different implementation

Enabling deployment types

```
<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>
```

A strongly ordered list of enabled deployment types. Notice how *everything* is an annotation and so typesafe!

Only component implementations which have enabled deployment types will we deployed to the container

Scopes and Contexts

- Extensible context model
  - A scope type is an annotation, can write your own context implementation and scope type annotation
- Dependent scope, @Dependent
- Built-in scopes:
  - Any servlet - @ApplicationScoped, @RequestScoped, @SessionScoped
  - JSF requests - @ConversationScoped
- Custom scopes

Scopes

```
public
@SessionScoped
@Component
class Login {
    private User user;
    public void login() {
        user = ...;
    }
    public User getUser() { return user; }
}
```

Session scoped

Scopes

```
public
@Component
class Printer {
    @Current Hello hello;
    @Current Login login;
    public void hello() {
        System.out.println(
            hello.hello( login.getUser().getName() ) );
    }
}
```

No coupling between scope and use of implementation

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();
    }
}
```

Conversation has the same semantics as in Seam

Conversation is demarcated by the application

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

```
public
@SessionScoped
@Component
class Login {
   private User user;
   public void login() {
      user = ...;
   }

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

Producer methods

```
public
@SessionScoped
@Component
class Login {
   private User user;
   public void login() {
      user = ...;
   }

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

Producer method components can a scope (otherwise inherited from the declaring component)

Producer methods

```
public
@Component
class Printer {
    @Current Hello hello;
    @Current User user;
    public void hello() {
        System.out.println(
            hello.hello( user.getName() ) );
    }
}
```

Much better, no dependency on Login!

Stereotypes

- We have common architectural “patterns” in our application, with recurring component roles
  - Capture the roles using stereotypes

Stereotypes

- A stereotype packages:
  - A *default* deployment type
  - A *default* scope
  - A *set* of interceptor bindings
  - *Restrictions* upon allowed scopes
  - *Restrictions* upon the Java type
  - May specify that components have names *by default*
- Built-in stereotypes: `@Component`, `@Model`

Creating a stereotype

```
public
@RequestScoped
@Named
@Production
@Casual
@Stereotype(
    supportedScopes={RequestScoped.class,
                     SessionScoped.class})
@Retention(RUNTIME)
@Target(TYPE)
@interface CasualAction {}
```

Default scope

Has a defaulted name

Default deployment type

A binding type

The supported scopes; specify another *on the implementation*, bang!

Using a stereotype

```
public
@CasualAction
class Hello {
    public String hello(String name) {
        return "hi " + name;
    }
}
```

Event producer

```
public
@Component
class Hello {
    @Observable @Casual Event<Greeting> casualHello;
    public void hello(String name) {
        casualHello.fire( new Greeting("hello " + name) );
    }
}
```

Inject an instance of `Event` using `@Observable`. Additional binding types can be specified to narrow the event consumers called. API type specified as a parameter on `Event`

Event consumer

```
public
@Component
class Printer {
   void onHello(@Observes @Casual Greeting greeting,
                @Current User user) {
      System.out.println(user + " " + greeting);
   }
}
```

Observer methods, take the API type and additional binding types

Additional parameters can be specified and will be injected by the container

Road Map

- Background
- Concepts
- Status

JSR-299

- Early Draft Review 1 published
  - Binding types
  - Events
  - Deployment types
  - Contexts
  - Components
- Since then
  - Specialization
  - Stereotypes
  - Decorators

Web Beans RI

- Work on implementing the current spec (EDR1+)
  - Components (Biding types, Scopes, Stereotypes
  - Events
  - Contexts
- Todo
  - Specialization
  - Decorators & Interceptors
  - Container initialization
- Beta Release in September

Q & A

http://in.relation.to/Bloggers/Pete

http://www.seamframework.org/WebBeans

http://jcp.org/en/jsr/detail?id=299