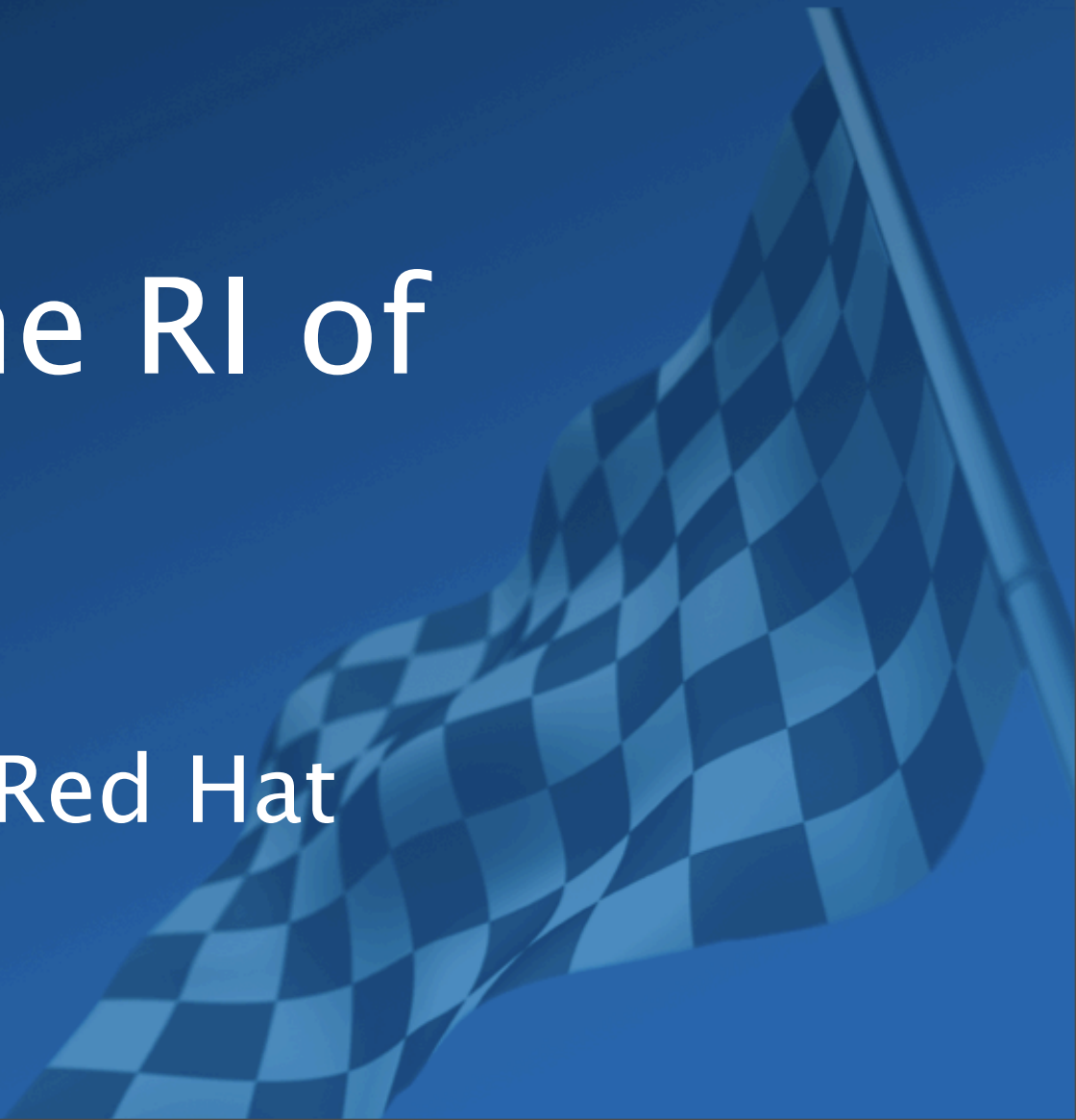
redhat

JBoss
a division of Red Hat

Web Beans, the RI of JSR-299

Pete Muir
JBoss, a Division of Red Hat



Road Map

- Background
- Concepts
- Demo
- Status

Java EE 6

- The EE 6 web profile removes most of the “cruft” that has developed over the years
  - mainly the totally useless stuff like web services, EJB 2 entity beans, etc
  - some useful stuff like JMS is also missing, but vendors can include it if they like
- EJB 3.1 – a whole bunch of cool new functionality!
- JPA 2.0 – typesafe criteria query API, many more O/R mapping options
- JSF 2.0 – don’t need to say much more!
- Bean Validation 1.0 – annotation-based validation API
- Servlet 3.0 – async support, better support for frameworks
- Standard global JNDI names

Goals

- JSR-299 defines a unifying dependency injection and contextual lifecycle model for Java EE 6
  - a completely new, richer dependency management model
  - designed for use with stateful objects
  - integrates the “web” and “transactional” tiers
  - makes it much easier to build applications using JSF and EJB together
  - includes a complete SPI allowing third-party frameworks to integrate cleanly in the EE 6 environment

What can be injected?

- Pre-defined by the specification:
  - (Almost) any Java class
  - EJB session beans
  - Objects returned by producer methods
  - Java EE resources (Datasources, JMS topics/queues, etc)
  - Persistence contexts (JPA EntityManager)
  - Web service references
  - Remote EJBs references
- Plus anything else you can think of!

Loose coupling

- Events, interceptors and decorators enhance the loose-coupling that is inherent in this model:
  - event notifications decouple event producers from event consumers
  - interceptors decouple technical concerns from business logic
  - decorators allow business concerns to be compartmentalized

Going beyond the spec

- Web Beans will provide extra integrations
  - Tomcat/Jetty support
  - Wicket support
  - ???
- and features which can be used in any JSR-299 environment
  - jBPM integration
  - log injection (choose between log4j and jlr, parameter interpolation
  - Seam2 bridge
  - Spring bridge

Seam 3?

- Use the JSR-299 core
- Provide a development environment
  - JBoss Tools
  - Seam-gen (command line tool)
- include a set of modules for any container which includes JSR-299
  - Seam Security
  - Reporting (Excel/PDF)
  - Mail

Road Map

- Background
- Concepts
- Demo
- Status

Essential ingrediants

- API types
- Binding annotations
- Scope
- Deployment type
- A name (optional)
- Interceptor bindings
- The implementation

Simple Example

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

Any Java Bean can use these services

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

So can EJBs

Simple Example

```
public class Printer {

    @Current Hello hello;

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

`@Current` is the default (built in) binding type

Constructor injection

```
public class Printer {
    private Hello hello;

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

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

Mark the constructor to be called by the container `@Initializer`

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

Web Bean Names

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

By default not available through EL.

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

If no name is specified, then a default name is used. Both these beans have the same name

JSF Page

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

Calling an action on a 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

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

We also specify the `@Casual` binding type. If no binding type is specified on a bean, `@Current` is assumed

Using a binding type

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

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

Deployment Types

- A deployment type is an annotation that identifies a deployment scenario
  - Deployment types may be enabled or disabled, allowing whole sets of beans to be easily enabled or disabled at deployment time
  - Deployment types have a precedence, allowing different implementations of an API to be chosen
  - 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

```
@Espanol
public class Hola extends Hello {

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

}
```

Same API, different implementation

Enabling deployment types

```
<Beans>
  <Deploy>
    <Standard />
    <Production>
    <i8ln:Espanol>
  </Deploy>
</Beans>
```

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

Only Web Bean implementations which have enabled deployment types will be 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

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

Session scoped

Scopes

```
public 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

```
@ConversationScoped
public 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 Web Bean where:
  - the objects to be injected are not required to be instances of Web Beans
  - the concrete type of the objects to be injected may vary at runtime
  - the objects require some custom initialization that is not performed by the Web Bean constructor

Producer methods

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

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

Producer methods

```
@SessionScoped
public class Login {
   private User user;

   @Produces @RequestScoped @LoggedIn
   User getUser() { return user; }

   @Produces
   String getWelcomeMessage(@Current Hello hello) {
      return hello.hello(user);
   }
}
```

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

Producer method can have a binding type

You can inject parameters

Producer methods

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

Much better, no dependency on Login!

Producer Fields

- Simpler alternative to Producer methods

```
@SessionScoped
public class Login {

   @Produces @LoggedIn @RequestScoped
   private User user;

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

Similar to outjection in Seam

Disposal Method

- Clean up after a producer method

```
public class UserDatabaseEntityManager {

   @Produces @UserDatabase
   EntityManager create(EntityManagerFactory emf) {
      return emf.createEntityManager();
   }

   void close(@Disposes @UserDatabase EntityManager em) {
   em.close();
   }
}
```

Same API type

Same binding types

Java EE Resources

- To inject Java EE resources, persistence contexts, web service references, remote EJB references, etc, we use a special kind of producer field declaration:

```
public class PricesTopic {
    @Produces @Prices
    @Resource(name="java:global/env/jms/Prices")
    Topic pricesTopic;
}
```

```
public class UserDatabasePersistenceContext {
    @Produces @UserDatabase
    @PersistenceContext
    EntityManager userDatabase;
}
```

Events

Event producers raise events that are then delivered to event observers by the Web Bean manager.

- not only are event producers decoupled from observers; observers are completely decoupled from producers
- observers can specify a combination of "selectors" to narrow the set of event notifications they will receive
- observers can be notified immediately, or can specify that delivery of the event should be delayed until the end of the current transaction

Event producer

```
public 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`

"Fire" an event, the producer will be notified

Event consumer

```
public 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

Specialization

- Allows a bean with a higher precedence deployment to completely replace a bean with a lower precedence
  - even producer methods, observer methods etc.

```
@Mock
@Specializes
public class MockLogin extends Login {

   @Produces
   User getUser() { return new DummyUser(); }
}
```

A @Mock deployment type for testing

Road Map

- Background
- Concepts
- Demo
- Status

Road Map

- Background
- Concepts
- Demo
- Status

JSR-299

- Public Review Draft 2 published
- Currently working on EE6 integration
- Web Beans “Book” (a less formal guide to JSR299)
  - http://www.seamframework.org/WebBeans
- Send feedback to jsr-299-comments@jcp.org

Web Beans

- The Reference implementation
  - Feature complete preview released in next few days
- Download it, try it out, give feedback!
  - http://seamframework.org/Download
- Supported in upcoming release:
  - JBoss 5.1.CR1
  - GlassFish V3 build 46
  - Tomcat 6.0.x
  - Jetty 6.1.x

Q & A

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

http://www.seamframework.org/WebBeans

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