JBoss®

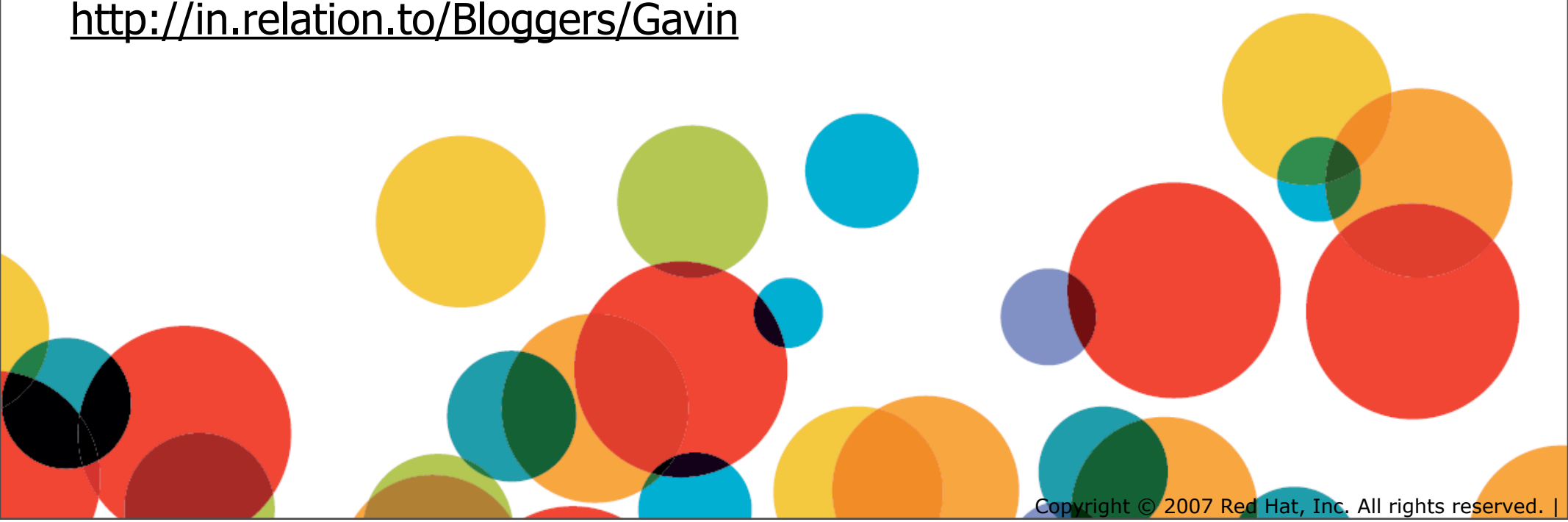
Contexts and Dependency Injection for Java EE

An introduction to JSR-299

Gavin King

gavin@hibernate.org

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



Why do you care about 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 - finally fixes the problems!
- Bean Validation 1.0 - annotation-based validation API
- Servlet 3.0 - async support, better support for frameworks
- Finally, standard global JNDI names
- Contexts and Dependency Injection for Java EE
  - JSR-299, the spec formally known as "Web Beans"
  - finally, a complete, standard DI mechanism

What is JSR-299?

- 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

The theme

- *Loose coupling...*
  - decouple server and client via well-defined types and "binding types"
    - so that the server implementation may vary
  - decouple lifecycle of collaborating components
    - components are contextual, with automatic lifecycle management
    - allows stateful components to interact like services, purely by message-passing
  - decouple orthogonal concerns
    - via interceptors
  - completely decouple message producer from consumer
    - via events

The theme

- *..with strong typing!*
  - eliminate lookup using string-based names
    - the compiler will detect typing errors
    - you don't need special authoring tools to get autocompletion, etc

What's unique?

- Implementations of a types may vary at deployment time - without the need for a central list of available implementations!
  - No need to explicitly list beans in XML (Spring, HiveMind, etc)
  - Nor even using a Java-based DSL (Guice)

What kinds of things can be injected?

- Certain kinds of things 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!
  - An SPI allows third-party frameworks to introduce new kinds of things

Simple example

- A really simple Java class:

```
public class Greeting {

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

}
```

EJB example

- The class could be an EJB:

```
@Stateless
public class Greeting {

    @RolesAllowed("friend")
    public String greet(String name) {
        return "hello " + name;
    }

}
```

Field injection

- A simple client:

```
public class Printer {

    @Current Greeting greeting;

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

}
```

Constructor injection

- Or, using constructor injection:

```
public class Printer {

    private Greeting greeting;

    public Printer(Greeting greeting) {
        this.greeting = greeting;
    }

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

}
```

Initializer method injection

- Or, using initializer method injection:

```
public class Printer {

   private Greeting greeting;

   @Initializer
   void init(Greeting greeting) {
      this.greeting = greeting;
   }

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

}
```

Binding types

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

Defining binding types

- Define a new binding type:

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

Declaring bindings

- Same type, different implementation:

```
public
@Informal
class InformalGreeting extends Greeting {

   public String greet(String name) {
      return "hi " + name;
   }

}
```

Declaring injection point bindings

- A client of the new implementation:

```
public class Printer {

    @Informal Greeting greeting;

    public void greet() {
        System.out.println( greeting.greet("SDC") );
    }

}
```

EL names

- To use our class in Unified EL expressions, give it a name:

```
public
@Named("printer")
class Printer {

    @Current Greeting greeting;

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

}
```

EL name defaulting

- Well, actually, that name can be defaulted:

```
public
@Named
class Printer {

    @Current Greeting greeting;

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

}
```

Unified EL

- Now we can use the object in a JSF or JSP page:

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

A stateful class

- If we want our object to hold state, we need to declare the scope of that state:

```
public
@RequestScoped
@Named
class Printer {

   @Current Greeting greeting;
   private String name;

   public void setName(String name) { this.name=name; }
   public String getName() { return name; }

   public void greet() {
      System.out.println( greeting.greet(name) );
   }

}
```

Unified EL

- And now we can use it to process a JSF form:

```
<h:form>
   <h:inputText value="#{printer.name}"/>
   <h:commandButton value="Say Hello"
                    action="#{printer.greet}"/>
</h:form>
```

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`
  - this is the default
  - it means that an object exists to serve exactly one client, and has the same lifecycle as that client
- Built-in scopes:
  - Any web request, web service request, RMI call, EJB timeout:
    - `@ApplicationScoped`, `@RequestScoped`
  - Any servlet:
    - `@SessionScoped`
  - JSF requests:
    - `@ConversationScoped`
- Custom scopes
  - provided by third-party frameworks via an SPI

Scoped objects

- A session-scoped object

```
public
@SessionScoped
class Login {

   private User user;

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

   public User getUser() { return user; }

}
```

Injecting a scoped object

- The client doesn't know anything about the lifecycle of the session-scoped object:

```
public
@Named
class Printer {

   @Current Greeting greeting;
   @Current Login login;

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

}
```

Custom scopes

- It's easy to create the annotation for a custom scope:

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

- After this, the hard work begins!
  - implement the `Context` SPI

Producer methods

- Producer methods allow control over the production of the injected instance
  - For runtime polymorphism
  - For control over initialization
  - Allow injection of classes we don't control, that don't satisfy the normal requirements of a class that may be injected
  - For further decoupling of a "producer" of state from the "consumer"

Declaring producer methods

- Simple producer method:

```
public
@SessionScoped
class Login {

   private User user;

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

   @Produces
   User getUser() { return user; }

}
```

Client of a producer method

- No more dependency to **`Login`**!

```
public class Printer {

    @Current Hello hello;
    @Current User user;

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

}
```

Scoped producer methods

- Producer methods may have a scope:

```
public
@RequestScoped
class Login {

   private User user;

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

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

}
```

- they may even have bindings, names, etc...

Producer fields

- Producer fields are just a shortcut:

```
public
@RequestScoped
class Login {

   @Produces @SessionScoped User user;

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

}
```

Declaring 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 UserDatabasePersistenceContext {

    @Produces @UserDatabase
    @PersistenceContext
    EntityManager userDatabase;

}

public class PricesTopic {

    @Produces @Prices
    @Resource(name="java:global/env/jms/Prices")
    Topic pricesTopic;

}
```

Injecting Java EE resources

- Now we've eliminated the use of string-based names:

```
public class UserDatabasePersistenceContext {

    @UserDatabase EntityManager userDatabase;

}
```

```
public class PricesTopic {

    @Prices TopicSession topicSession;
    @Prices TopicPublisher topicPublisher;

}
```

Deployment types

- A deployment type is an annotation that identifies a deployment scenario
  - Deployment types may be enabled or disabled, allowing whole sets of implementations to be easily enabled or disabled at deployment time
  - Deployment types have a precedence, allowing the container to choose between various implementations of a type
  - Deployment types replace verbose XML configuration documents or Java-based DSLs
- Default deployment type: `Production`

Defining a deployment type

- Define a custom deployment type:

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

- (Actually, we don't really use deployment types for i18n, since the locale depends upon the user, not the deployment!)

Declaring the deployment type

- Same type, different deployment type:

```
public
@Espanol
class Saludo extends Greeting {

   public String greet(String nombre) {
      return "hola " + nombre;
   }

}
```

Enabled deployment types

- Implementation depends upon which deployment types are enabled:

```
<Beans xmlns="urn:java:ee"
       xmlns:myapp="urn:java:com.mydomain.myapp">

   <Deploy>
      <Standard/>
      <Production/>
      <myapp:Espanol/>
   </Deploy>

</Beans>
```

- (The JSR-299 XML is also strongly-typed, but we don't have time to talk about it now.)

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

Controlling the conversation context

- The conversation context is demarcated by the application:

```
public
@ConversationScoped
class NumberGuess {

   @Current Conversation conversation;

   private int number;
   private int min;
   private int max;

   @Initializer
   void start(@Random int random) {
      conversation.begin();
      number = random;
      min = 1;
      max = 100;
   }

...
```

Controlling the conversation context

- The conversation context is demarcated by the application:

```
...

   public boolean guess(int guess) {
      if (guess==number) {
         conversation.end();
         return true;
      }
      else {
         if (guess<number && guess>min) {
            min=guess;
         }
         else if (guess>number && guess<max) {
            max=guess;
         }
         return false;
      }
   }
}
```

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 implementation
  - 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 {}
```

Declaring interceptor bindings of an interceptor

- Interceptor implementation:

```
public
@Secure
@Interceptor
class SecurityInterceptor {

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

}
```

Class-level interceptor bindings

- Class-level interceptor:

```
public
@Secure
class Greeting {

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

}
```

Method-level interceptor bindings

- Method-level interceptor:

```
public class Greeting {

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

}
```

Interceptor binding types

- Multiple interceptors:

```
public
@Transactional
class Greeting {

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

}
```

Enabled interceptors

- Interceptor ordering and enablement:

```
<Beans xmlns="urn:java:ee"
       xmlns:secure="urn:java:org.jboss.secure"
       xmlns:tx="urn:java:org.jboss.tx">

   <Interceptors>
      <secure:SecurityInterceptor/>
      <tx:TransactionInterceptor/>
   </Interceptors>

</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
class Greeting {

   public String greet(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 type
  - May specify that components have names by default

Defining stereotypes

- Defining a new stereotype:

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

Declaring stereotypes

- Using a stereotype:

```
public
@Action
class Greeting {

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

}
```

Event producers

- Event producer:

```
public class Login {

    @Fires Event<LoggedIn> loggedInEvent;

    public void login() {
        User user = ...;
        loggedInEvent.fire( new LoggedIn(user) );
    }

}
```

Event consumers

- Event consumer:

```
public class Printer {

   void onLogin(@Observes LoggedIn loggedIn,
                Greeting greeting) {
      System.out.println(
         greeting.greet( loggedIn.getUser().getName() ) );
   }

}
```

Event producers with binding types

- Events may also use binding types:

```
public class Login {

    @Fires @LoggedIn Event<User> loggedInEvent;

    public void login() {
        User user = ...;
        loggedInEvent.fire(user);
    }

}
```

Event consumers with binding types

- Event consumer:

```
public class Printer {

   void onLogin(@Observes @LoggedIn User user,
                Greeting greeting) {
      System.out.println( greeting.greet( user.getName() ) );
   }

}
```

More information

- Public draft:
  - http://www.jcp.org/en/jsr/detail?id=299
- Reference Implementation:
  - http://seamframework.org/WebBeans
- RI Documentation:
  - http://docs.jboss.org/webbeans/reference/current/en-US/html/
- Blog:
  - http://in.relation.to/Bloggers/Everyone/Tag/Web+Beans