JSR-299
Contexts and Dependency and Dependency Injection for Java EE

Gavin King & Pete Muir
Road Map

- Background
- Concepts
- Status
Java EE 6

• The EE 6 web profile removes most of the “cruft” that has developed over the years
  • mainly useless stuff like web services, EJB 2 entity beans etc.
  • some useful stuff (e.g. JMS) is missing, **but** vendors can include it
• EJB 3.1 - a whole bunch of cool new functionality!
• JPA 2.0 - typesafe criteria API, many more O/R mapping options
• JSF 2.0 - Ajax, easy component creation, bookmarkable URLs
• Bean Validation 1.0 - annotation-based validation API
• Servlet 3.0 - async support, better support for frameworks
• Standard global JNDI names
• Managed Beans
Managed Beans

- Container-managed POJOs with minimal requirements
- Support a set of basic services
  - Resource injection
  - Lifecycle callbacks
  - Interceptors
- The foundation for all other component types in the platform
  - Core services centralized under Managed Beans
- Other specifications will add support for additional services
  - Remoting
  - Instance pooling
  - Web services
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
Loose coupling

- Decouple server and client
  - Using well-defined types and “qualifiers”
  - Allows server implementation to vary
- Decouple life cycle of collaborating components
  - Automatic contextual life cycle management
  - Stateful components interact like services
- Decouple orthogonal concerns (AOP)
  - Interceptors
  - Decorators
- Decouple message producer from message consumer
  - Events
**Strong typing**

- Eliminate reliance on string-based names
- Compiler can detect typing errors
  - No special authoring tools required for code completion
  - Casting virtually eliminated
- Report errors early
  - At deployment
  - Tooling can give prevent ambiguous dependencies
Going beyond the spec

• Web Beans provides extra integrations
  • Tomcat/Jetty support
  • Java SE support
  • OSGi containers
  • ???

• and features which can be used in any JSR-299 environment
  • Seam2 bridge
  • Spring bridge
  • Wicket support
  • ???
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
  • jBPM integration
  • Seam Security
  • Reporting (Excel/PDF)
  • Mail
  • etc.
Road Map

- Background
- Concepts
- Status
Essential ingredients

• API types
• Qualifier annotations
• Scope
• Alternative
• A name (optional)
• Interceptor bindings
• The implementation
Simple Example

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

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

Any Managed Bean can use these services

So can EJBs
Simple Example

```java
public class Printer {
    @Inject Hello hello;

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

@Inject defines an injection point. @Default qualifier is assumed.
**Constructor injection**

```java
public class Printer {
    private Hello hello;

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

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

- Mark the constructor to be called by the container with `@Inject`.
- Constructors are injected by default; `@Default` is the default qualifier.
Web Bean Names

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

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

By default not available through EL.

If no name is specified, then a default name is used. Both these Managed Beans have the same name.
JSF Page

Calling an action on a bean through EL

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

A qualifier is an annotation that lets a client choose between multiple implementations of an API at runtime.
Defining a qualifier

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Casual {}
Using a qualifier

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

We also specify the `@Casual` qualifier. If no qualifier is specified on a bean, `@Default` is assumed.
Using a qualifier

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

Here we inject the Hello bean, and require an implementation which is qualified by @Casual
Alternatives

• An alternative bean is one which must be specifically enabled for a particular deployment
  • It replaces the managed or session bean for which it is an alternative
  • May also completely replace it
    • all producers and observers defined on original bean are disabled for this deployment
  • Alternatives enabled in XML deployment descriptor
Defining an alternative

@Alternative
public class Hola extends Hello {
    public String sayHello(String name) {
        return "hola " + name;
    }
}

Same API, different implementation
Enabling an alternative

```xml
<beans>
  <alternatives>
    <class>com.acme.Hola</class>
    <stereotype>com.acme.SouthernEuropean</stereotype>
  </alternatives>
</beans>
```

Can also define a stereotype as an alternatives. Any stereotyped beans will be an alternative.
Stereotypes

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

• A stereotype encapsulates any combination of:
  • a default scope, and
  • a set of interceptor bindings.

• A stereotype may also specify that:
  • all beans with the stereotype have defaulted bean EL names
  • all beans with the stereotype are alternatives
Creating a stereotype

@RequestScoped
@Named
@Alternative
@Stereotype
@Retention(RUNTIME)
@Target(TYPE)
public @interface
    AlternativeAction{}
Using a stereotype

@AlternativeAction
public class Hello {
    public String sayHello(String name) {
        return "hi " + name;
    }
}

Scopes and Contexts

• Built-in scopes:
  • Any servlet - @ApplicationScoped, @RequestScoped, @SessionScoped
  • JSF requests - @ConversationScoped

• Dependent scope: @Dependent

• Custom scopes
  • A scope type is an annotation, can write your own context implementation and scope type annotation
Scopes

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

```java
public class Printer {

    @Inject Hello hello;
    @Inject Login login;

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

No coupling between scope and use of implementation.
Producer methods

- Producer methods allow control over the production of a bean where:
  - the objects to be injected are not managed instances
  - 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 bean constructor
Producer methods

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

@Produces
User getUser() {
    return user;
}
}
Producer methods

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

Much better, no dependency on Login!
```
Producer Fields

- Simpler alternative to Producer methods

```java
@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

```java
public class UserDatabaseEntityManager {

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

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

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

```java
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

```java
class Hello {
    @Inject @Any Event<Greeting> greeting;

    public void sayHello(String name) {
        greeting.fire(new Greeting("hello " + name));
    }
}
```

Inject an instance of Event. Additional qualifiers 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

```java
public class Printer {

    void onGreeting(@Observes Greeting greeting,
                    User user) {
        System.out.println(user + " says " + greeting);
    }

}
```

- Observer methods take the API type and additional qualifiers.
- Additional parameters can be specified and will be injected by the container.
Road Map

- Background
- Concepts
- Status
JSR-299

• Proposed Final Draft published
• Proposed Final Draft 2 in draft form
• Web Beans “Book” (a less formal guide to JSR299)
  • http://www.seamframework.org/WebBeans
  • look for an update soon!
• Send feedback to jsr-299-comments@jcp.org
Web Beans

• The Reference implementation

• Feature complete preview of second public review draft. Download it, try it out, give feedback!
  • http://seamframework.org/Download

• Working on first release candidate of the final draft (expected mid September)
Web Beans

• Integrated into:
  • JBoss 5.1.0 and above
  • GlassFish V3 build 46 and above

• Available as an addon for:
  • Tomcat 6.0.x
  • Jetty 6.1.x
Q & A

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

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

http://www.seamframework.org/WebBeans