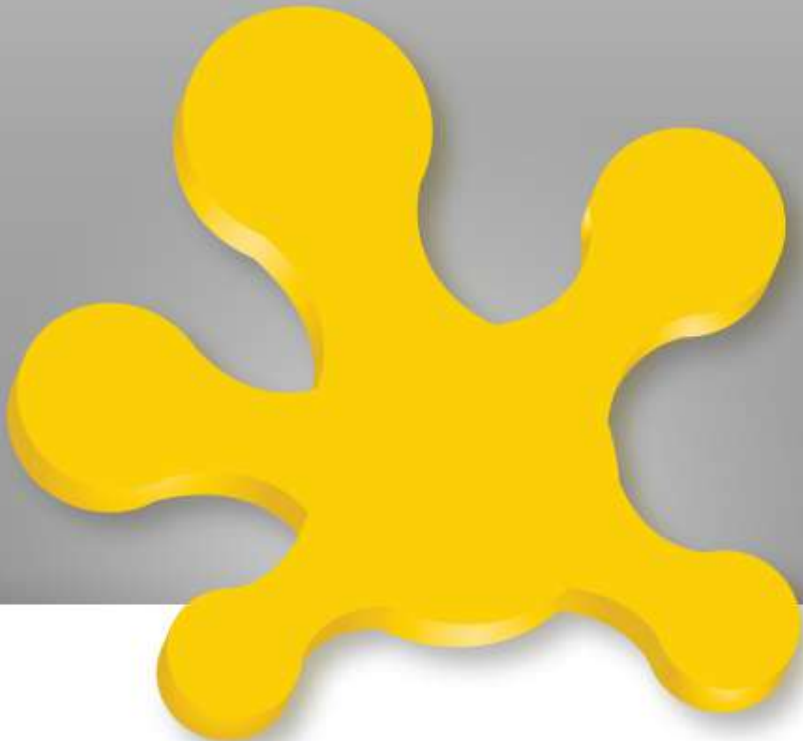


# Mark Struberg

Apache OpenWebBeans PMC

## JSR-299 (DI hard)



Let's move the Java world!



# JSR-299 Overview

- Contexts and Dependency Injection for the Java EE platform (CDI)
- Component Model for Java (SE and EE)
- The spec formerly known as „WebBeans“
- Started ~2007
- Originally designed for J2EE but also usable for standalone applications

# Available Implementations

- JBoss Weld (RI)
- Apache OpenWebBeans
- Resin CanDI
- maybe Spring in the future?

# CDI Features

- Typesafe Dependency Injection
- Interceptors
- Decorators
- Events
- SPI for implementing Portable Extensions
- Unified EL integration

# What is Dependency Injection?

- Uses Inversion Of Control pattern for object creation
- Hollywood Principle: don't call us, we call you
- No more hardcoded dependencies when working with Interfaces

```
> MyDbService dbS =
```

```
> new DbServiceForVerySpecialDatabase();
```

# Why use Dependency Injection

- Easy to change implementations
- Encourage Separation of Concerns
- Minimise dependencies
- Dynamic Object retrieval
- Makes Modularisation easy
- Simplifies Reusability

# Valid Bean Types

- Almost any plain Java class (POJO)
- EJB session beans
- Objects returned by producer methods or fields
- JNDI resources (e.g., DataSource)
- Persistence units and persistence contexts
- Web service references
- Remote EJB references
- Additional Types defined via SPI

# A small Example

- Create a META-INF/beans.xml marker file
- Create a MailService implementation

- > **@ApplicationScoped**

```
> public class MyMailService
> implements MailService {
>     public send(String from, String to,
>                 String body) {
>         .. do something
>     }
> }
```



# A small Example

- We also need a User bean
  - > @SessionScoped
  - > @Named
  - > public class User {
  - > public String getName() {..}
  - > ..
  - > }

# A small Example

- Injecting the Bean

```
> @RequestScoped
> @Named(„mailForm“)
> public class MailFormular {
>     private @Inject MailService mailSvc;
>     private @Inject User usr;
>
>     public sendMail() {
>         mailSvc.send(usr.getName(),
>             „other“, „sometext“);
>     }
> }
```

# A small Example

- Use the beans via Expression Language
  - > `<h:form>`
  - > `<h:outputLabel value="username"`
  - > `for"username" />`
  - > `<h:outputText id="username"`
  - > `value="#{user.name}" />`
  - > `<h:commandButton value="Send Mail"`
  - > `action="#{mailForm.send}" />`
  - > `</h:form>`

# How DI containers works

- Contextual Instance factory pattern
- Someone has to trigger the instance creation - even in a DI environment
- But the trigger has no control over the Implementation class nor Instance

# 'Singletons', Scopes, Contexts

- Each created Contextual Instance is a 'Singleton' in a well specified Context
- The Context is defined by it's Scope
  - @ApplicationScoped -> ApplicationSingleton
  - @SessionScoped -> SessionSingleton
  - @ConversationScoped -> ConversationSingleton
  - @RequestScoped -> RequestSingleton
  - ... code your own ...

# Built In Scopes

- @ApplicationScoped
- @SessionScoped
- @RequestScoped
- @ConversationScoped
- All those scopes are of type @NormalScope

# Special Scopes

- @Dependent
  - Pseudo scope
  - If injected, the Contextual Instance will get/use the Context of it's parent
- Qualifier @New (no scope!)
  - Annotation only valid at injection points!
  - For each injection, a new instance will be created

# Terms - „Managed Bean“

- The term Managed Bean means a Java Class and all its rules to create contextual instances of that bean.
- Interface Bean<T>
- Managed Beans in the spec don't mean Java Beans!



# Terms - Contextual Instance

- The term 'Contextual Instance' means a Java instance created with the rules of the Managed Bean Bean<T>
- Contextual Instances usually don't get injected directly!

# Terms - Contextual Reference

- The term 'Contextual Reference' means a proxy for a Contextual Instance.
- Proxies will automatically be created for injecting @NormalScope beans.

# Why use Proxies

- Passivation of non-serializable contextual instances (via it's contextual reference)
- scope-differences This happens e.g. if a `@SessionScoped` User gets injected into a `@ApplicationScoped` MailService.
- Interceptors, Decorators

# The big picture

- Creating the meta information at startup
  - Classpath Scanning and creating Managed Beans metainformation
- Contextual Instance creation at runtime
  - Based on the Managed Beans, the Context will maintain the instances
- Well defined contextual instance termination

# Qualifiers

- A Qualifier enables the lookup of *specific* Implementations at runtime
- Qualifier connects 'creation info' with InjectionPoints.
- Kind of a typesafe 'naming' mechanism

# Builtin Qualifiers

- **@Named** – used to supply an EL name
- **@Default** - Assumed, if no other Qualifier (beside @Named) is specified
- **@New** - For each managed bean (and session bean), a second Managed Bean with @New exists.
- **@Any** - Always exists but not for @New beans

# Using Qualifiers

- Qualifiers can be used to distinct between beans of the same Type
  - › `private @Inject @Favorite Track;`
  - › `Private @Inject @Recommended Track;`
- Special Qualifier `@Named` for defining EL names.

# Define your own Qualifiers

- Creating own annotations will get your daily bread and butter
  - > **@Qualifier**
  - > @Retention(RUNTIME)
  - > @Target({TYPE, METHOD, FIELD, PARAMETER})
  - > public **@interface** Favorite {}



# Producers

- Write producer methods or producer fields if more logic is needed at instance creation time
  - > @ConversationScoped
  - > public class Playlist {
  - >     **@Produces** @Favorite @RequestScoped
  - >     public Track getFavTrack() {
  - >         return new Track(favTitle)
  - >     }
  - >     public void dropT(**@Disposes** @Favorite Track t)
  - >         { dosomecleanup(); }
  - >     }

# Alternatives

- Swap Implementations for different installations

- Kind of an optional override of a bean

- > **@Alternative**

- > `public class MockMailService`

- > `implements MailService { ...`

- Disabled by default!

- > `<beans>`

- > `<alternatives>`

- > `<class>org.mycomp.MockMailService</class>`

- > `...`

# Interceptors

- An Interceptor decouples technical concerns from business logic
- Applying cross cutting concerns to beans
- JSR-299 allows you to write your own interceptors
- Interceptors are treated as being `@Dependent` to the intercepted class

# Interceptor usage

- Interceptor annotations can be applied on method or class level
  - > `@ApplicationScoped`
  - > `@Named`
  - > `public class MyUserService {`
  - >     **`@Transactional`**
  - >     `public storeUser(User u) {`
  - >         `...`
  - >     `}`
  - > `}`

# InterceptorBindingType

- InterceptorBindingTypes are used to identify which Interceptors should be applied to a bean
  - > **@InterceptorBinding**
  - > @Retention(RetentionPolicy.RUNTIME)
  - > @Target( { ElementType.TYPE,  
>            ElementType.METHOD } )
  - > public **@interface** Transactional {
  - >     @Nonbinding boolean requiresNew()
  - >                            default false;
  - > }

# The Interceptor Impl

```
> @Interceptor @Transactional
> public class TransactionalInterceptor {
>     private @Inject EntityManager em;
>     @AroundInvoke
>     public Object invoke(InvocationContext context)
>     throws Exception {
>         EntityTransaction t =em.getTransaction();
>         try {
>             if(!t.isActive())
>                 t.begin();
>             return context.proceed();
>         } catch(Exception e) {
>             .. rollback and stuff
>         } finally {
>             if(t != null && t.isActive())
>                 t.commit();
>         }
>     }
> }
```

# Enabling Interceptors

- Interceptors need to be enabled manually in **beans.xml**
- You can define the order in which multiple Interceptors stack
  - › `<beans>`
  - › `<interceptors>`
  - › `<class>org.mycomp.Secured</class>`
  - › `<class>org.mycomp.Transactionnal</class>`
  - › `</interceptors>`
  - › `</beans>`

# Events

- Observer/Observable pattern

- Inject a Event producer

```
> private @Inject Event<UserLoggedIn> loginEvent;  
> private @Inject User usr;  
> ..  
> loginEvent.fire( new UserLoggedIn(user) );
```

- Event Consumer

```
> public void onLogin(@Observes UserLoggedIn ule) {  
>     ule.getUser().getName; ...  
> }
```

- Attention: don't use private observer methods!



# Writing own Extensions

- CDI Extensions are portable
- Easy to write!
- Are activated by simply dropping them into the classpath
- CDI Extensions are based on `java.util.ServiceLoader` Mechanism

# @ProjectStageActivated

- Allows to disable Beans depending on a JSF ProjectStage
- Defining the annotation is easy:
  - > @Retention(RetentionPolicy.RUNTIME)
  - > @Target({ElementType.TYPE})
  - > public **@interface** ProjectStageActivated {
  - > ProjectStage[] value();
  - > }

# @ProjectStageActivated

- As is the Extension implementation

```
> public class ProjectStageActivationExtension
>     implements Extension {
>     protected void vetoAlternativeTypes(
>         @Observes ProcessAnnotatedType<Object> pat) {
>         if (hasProjectStageActivatedAnnotation(pat)) {
>             if (isInProjectStage(activatedIn)) {
>                 return;
>             }
>             // otherwise this alternative shall not get used
>             pat.veto();
>         }
>     }
```

# @ProjectStageActivated

- Extensions need to get registered in the ServiceLoader
- Create a file  
`META-INF/services/javafx.enterprise.inject.spi.Extension`
- Simply write your Extensions classname into it  
> `com.mycomp.ProjectStageActivationExtension`

# Documentation

- JSR-299 spec:

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

- OpenWebBeans:

<http://openwebbeans.apache.org>

svn co <https://svn.apache.org/repos/asf/openwebbeans/trunk>

- Weld: <http://jboss.org>

- Seam3: <http://seamframework.org>

- MFaces EXTCDI

svn co \

<https://svn.apache.org/repos/asf/myfaces/extensions/cdi/trunk>

# Q & A