183.223 Web Application Engineering & Content Management

Leitung: Univ.Prof. DI Dr. Thomas Grechenig

CDI - DI hard

Mark Struberg

Kontakt: teaching@inso.tuwien.ac.at

About me

- Mark Struberg, INSO TU Vienna
- struberg@apache.org
- Apache Software Foundation member
- Committer / PMC for Apache OpenWebBeans, MyFaces, Maven, OpenJPA, BVal, Isis, DeltaSpike, commons, Geronimo, JBoss Arquillian, ...
- Java JCP Expert Group member
- Twitter: @struberg
- github.com/struberg

Agenda

- JSR-299 Overview
- Basic CDI Concepts
- Producers
- Alternatives
- Interceptors
- Events
- Extensions
- JSR-346 (CDI-1.1) preview

How the ASF works

- Non profit Organization
- Contributors
- Committers
- PMC member
  (Project Management Committee)
- ASF member
- ASF board of directors

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
- Based on JSR-330

CDI Features

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

Available Implementations

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

Basic CDI Concepts

What is Dependency Injection?

- Uses Inversion Of Control pattern for object creation
- Hollywood Principle
  "Don't call us, we call you"
- Macho Principle
  "Gimme that damn thing!"
- No more hardcoded dependencies when working with Interfaces

```
MailService ms = new VerySpecialMailService();
```

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/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 String sendMail() {
    mailSvc.send(usr.getEmail(),
                 „other“, „sometext“);
    return “successPage”;
  }
}
```

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.sendMail}"/>
</h:form>
```

Injection

- Defined in JSR-330
- field injection:

```
private @Inject MailService mailService;
```

- constructor injection

```
public class X {
  @Inject
  public X(User u) {
    this.credential = user.getCredential(); ...
```

- @Inject methods

```
public @Inject setUser(User u) {...
```

- parameter injection is available for methods invoked by the container. Parameters dont need '@Inject'

How DI containers work

- Contextual Instance **factory pattern**
  - recursively fill all InjectionPoints
- 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
- @SessionScoped and @ConversationScoped are marked as **passivating**="true"
  - Beans stored in those Contexts must be **Serializable**!

Special Scopes

- @Dependent
  - Pseudo scope
  - Each InjectionPoint gets a new instance
  - If injected, the Contextual Instance will get/use the Lifecycle of it's parent
- @Dependent is assumed if no other Scope is explicitly assigned to a class!
- Qualifier @New (no scope!)
  - Annotation only valid at injection points!
  - For each injection, a new instance will be created
  - barely usable...

Terms – „Bean“

- The term Bean means a Java Class and all it's **rules** to **create contextual instances** of that bean.
- 'Managed Beans' in JSR-299 and JSR-346 doesn't mean JavaBeans!
  - It does **NOT** mean @ManagedBean from **EE-6**
  - it does **NOT** mean @ManagedBean from **JSF-2**
- Interface `Bean<T> extends Contextual<T>`

The Bean<T> interface

```
public interface Contextual<T> {
  T create(CreationalContext<T> creationalContext);
  void destroy(T instance, CreationalContext<T> cc);
}

public interface Bean<T> extends Contextual<T> {
  Set<Type> getTypes();
  Set<Annotation> getQualifiers();
  Class<? extends Annotation> getScope();
  String getName();
  Set<Class<? extends Annotation>> getStereotypes();
  Class<?> getBeanClass();
  boolean isAlternative();
  boolean isNullable();
  Set<InjectionPoint> getInjectionPoints();
}
```

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.

How Proxies work

- 'Interface Proxies' vs 'Class Proxies'
- Generated subclasses which overload all non-private, non-static methods

```
public class User$$ extends User {
  public String getName() {
    return getInstance().getName();
  }
  private T getInstance() {
    beanManager.getContext().get(...);
  }
}
```

- Can contain Interceptor logic, Decorators,
- Usually in a 'MethodHandler' using reflection

Why use Proxies

- scope-differences - this happens e.g. if a @SessionScoped User gets injected into a @ApplicationScoped MailService.
- Passivation of non-serializable contextual instances (via it's contextual reference)
- Interceptors, Decorators
- Circular Injection A->B->A
- Non-normalscoped beans only get a proxy if they have an Interceptor or Decorator

Unproxyable Bean Types

- The following classes cannot be proxied
  - classes which don't have a non-private default ct
  - classes which are declared final or have non-static final methods,
  - primitive types,
  - array types.

The big picture

- Creating the meta information at startup
  - Classpath Scanning and creating Managed Beans meta-information -> List<Bean<?>>
- 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

Using Qualifiers

- Qualifiers can be used to distinct between beans of the same Type

  private @Inject **@Favourite** Track;

  private @Inject **@Recommended** Track;

- Special Qualifier @Named for defining EL names.

Built-in Qualifiers

- **@Named** – used to supply an EL name
- **@Default** - Assumed, if no other Qualifier (beside @Named) is specified
- **@Any** – matches all Qualifiers

The @Named Qualifier

- Names are mainly used for EL
- @Named(name=”myName”)
- @Named default name
  - unqualified class name in camelCase
  - camelCase producerMethod name (without 'get' or 'is')
  - camelCase producerField name
- The use of @Named as an string injection point qualifier is not recommended but possible:

```
» private @Inject @Named SpringMailService
  mailService;
» // equivalent to @Named(“mailService”)..
```

Define your own Qualifiers

- Creating own annotations will get your daily bread and butter

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

@Nonbinding

- @Nonbinding excludes annotation parameters from the comparison
- Works for Qualifiers and Interceptorbindings

```
@Qualifier
public @interface ConfigProperty {
  @Nonbinding String name();
}
```

Producers

- Write producer methods or producer fields if more logic is needed at instance creation time

```
@ConversationScoped
public class Playlist {
  @Produces @Favourite @RequestScoped
  public Track getFavTrack() {
    return new Track(favTitle)
  }
  public void drop(@Disposes @Favourite Track t) {
    dosomecleanup(t);}
  }
}
```

- Each producer method / producer field forms an own ManagedBean Bean<T>!

Producer Method pitfalls

- producer method and disposal method must be in the same class
- no @Specializes behaviour defined for disposal methods in CDI-1.0 -> fixed in CDI-1.1
- @Dependent instances of the beans containing the producer method will only exist for a single invocation of the producer method/disposal method

Producer Field pitfalls

- Cannot be destroyed in CDI-1.0
  - @Disposal method for producer fields introduced in CDI-1.1
- Take care about the Scope of the containing Class!

@Specializes

- 'Replaces' the Managed Bean of a superclass
- Bean must extends the superclass
- 'Disables' the original Bean
- possible to specialise Producer Methods
- or even EJBs:

```
@Stateless
public class UserSvcFacade implements UserSvc
 { ... }
@Stateless @Mock @Specializes
public class MockUserSvcFacade extends UserSvcFacade
 { ... }
```

@Alternative

- 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>
 ...
```

Pitfalls with Alternatives and Interceptors

- Per CDI-1.0 only active for the BDA (Bean Definition Archive)
- That will officially be changed in CDI-1.1
  see https://issues.jboss.org/browse/CDI-18
- OpenWebBeans always used global enablement
- DeltaSpike provides a 'globalAlternatives' system for Weld

@Stereotype

- Stereotypes are used to combine various attributes and roles
  - A default scope
  - Enable default naming
  - Set of InterceptorBindings and @Alternative behaviour
  - can be meta-annotated with other Stereotypes!
- A Stereotype should not define Qualifiers other than @Named (without parameter!)

@Stereotype example

```
@Stereotype
@ApplicationScoped
@Transactional
@Secured
@Target(TYPE)
@Retention(RUNTIME)
public @interface Service {
   // in CDI-1.1 see CDI-229
   @Propagage(annotation=Secured.class, name = “roles”);
   String[] roles();
}
```

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
public class UserService {
  @Transactional
  public storeUser(User u) {
    ...
  }
}
```

InterceptorBinding Type

- InterceptorBindings 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.Transactional</class>
  </interceptors>
</beans>
```

Decorators

- Decorators uses the delegation pattern by proxying complete functions
- A Decorator may implement business logic

```
@Decorator
class abstract TimestampLogger implements Logger {
  @Inject @Delegate @Any Logger logger;

  public void log(String text) {
    logger.log(timestamp() + text);
  }
  ...
}
```

Enabling Decorators

- Decorators need to be enabled manually in beans.xml

```
<beans>
  <decorators>
    <class>org.mycomp.TimeStampLogger</class>
  </decorators>
</beans>
```

- Decorators are always called after Interceptors

Events

- Observer/Observable pattern
- Inject a Event producer

```
private @Inject Event<UserLoggedIn> loginEvent;
loginEvent.fire( new UserLoggedIn(“hans”) );
```

- Event Consumer

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

- Attention: don't use private observer methods!
- CDI Events are Synchronous!
- CDI Events are no Messages!

Cache cleaning via Events

- If user changes language or logs in, then we need to change the menu

```
@SessionScoped
public class Menu {
private @Inject MenutItem parentMenu;
protected reloadMenu(
  @Observes UserSettingsChangedEvend uscE) {
    parentMenu = menuSvc.loadMenu();
  }
}
```

Service Provider with Events

- You like to know all free resources:

```
public interface ResourceProvider {...

public class InterrestedIn {
  private List<ResourceProvider> rp = new ArrayList<~>();
}
```

- Collect interrests
  - `InterrestedIn interrested = new InterrestedIn();`
  - `collectResources.fire(interrested);`
  - `for(ResourceProvider p : interrested) ..`
- Declare yourself as interrested
  - `public void declareInterrest(@Observes InterrestedIn in) {`
  - `  in.rp.add(this);`
  - `}`

What's New in CDI-1.1

- Globally enabled Interceptors and Alternatives
- 'implicit bean archive'
- XML configuration -> NO!
- @Disposes for producer fields
- Clarifies some AnnotatedType behaviour
- @Veto + optional beans
- Relax Serialization rules
- @EarApplicationScoped vs @WebApplicationScoped
- https://issues.jboss.org/browse/CDI

Common Pitfalls & New Programming Patterns

AmbiguousResolutionException

- Happens if multiple Bean<T> serve the same InjectionPoint.
- MenuItem example:

```
public class MenuItem implements Serializable {
  private String id;
  private String name;
  private MenuItem parent;
  private MenuItem[] children;
  ...
}
```

The MenuItem Producer

- We use a producer method to create the menu tree

```
@ApplicationScoped
public class MenuProducer {
  private @Inject MenuService menuSvc;
  @Produces @SessionScoped
  public MenuItem createMenuItem(User usr){
    return menuSvc.getMenuOfUser(usr);
  }
}
```

The Bean Resolution Process

- The container first resolves all Beans which implement the Type of this InjectionPoint

```
» private @Inject MailService mailSvc;
```

- The container removes all Beans which do not match the Qualifier of the InjectionPoint. (see @Nonbinding!)
- The container checks if @Specializes or @Alternative Beans are applicable
- If more than 1 Bean remains → AmbiguousResolutionException

@Typed

- Defines the Java Type which should be used for that bean
- Possible to 'disable' a bean with @Typed() to prevent AmbiguousResolutionExceptions

```
@Typed() // basically disables this Bean
public class MenuItem implements Serializable {
```

- handy for subclasses which should NOT conflict with their parent class types

Explicitly @Typed

```
@ApplicationScoped
public class TrackReadOnlyService {
  public Track readTrack(int id) {..}
}

@Typed(TrackReadWriteService.class)
@ApplicationScoped
public class TrackReadWriteService
extends TrackReadOnlyService {
  public void writeTrack(Track t) {..}
}
```

Typesafe Configuration

- Config without properties or XML files
- Just create an Interface or default config class

```
public interface MyConfig{
   String getJdbcUrl();
}
```

- Later just implement, @Specializes, @Alternative, @ProjectStageActivated, @ExpressionActivated your config impl.

Behaviour Mutation

- Difference between cached and uncached version of the same service:

```
@Qualifier
@Target({TYPE, FIELD, METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Cached {
}
```

The Services

```
@ApplicationScoped
public class BusService {
  Bus getBus(int id) {..}
}

@Cached
@ApplicationScoped
public class CachedBusService
extends BusService { .. }
```

Dynamic Resolving

```
@Inject @Any
private Instance<BusService> busSvcSource;
private BusService busSvc;
..
@PostConstruct
protected init() {
  busSvc = busSvcSource.select(
    user.isAdmin()
      ? new AnnotationLiteral<Default>() {}
      : new AnnotationLiteral<Cached>() {}
          ).get();
}
```

Portable Extensions

Writing own Extensions

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

Extension Basics

- Extensions will get loaded by the BeanManager
- Each BeanManager gets it's own Extension instance
- During the boot, the BeanManager sends Lifecycle Events to the Extensions
- **Attention**:
  strictly distinguish between boot and runtime!

Register the Extension

- Extensions need to get registered for the ServiceLoader
- Create a file

  META-INF/services/javax.enterprise.inject.spi.Extension
- Simply write the Extensions classname into it

  com.mycomp.MyFirstCdiExtension

New CDI-1.1 Lifecycle Events

- ProcessModule
  - allows modifying Alternatives, Interceptors and Decorators
- ProcessSyntheticAnnotatedType
- ProcessInjectionPoint
- ProcessAnnotatedType with @HasAnnotation(Annotation[])
- Maybe @Startup or similar. Even in CDI-1.1 or EE-7

Extension Lifecycle Events

- Extensions picked up and instantiated at Container startup
- Container sends 'system events' which all Extensions can @Observes
  - BeforeBeanDiscovery
  - ProcessAnnotatedType (with `@WithAnnotations` in CDI-1.1)
  - ProcessBean, ProcessProducer
  - ProcessInjectionTarget
  - AfterBeanDiscovery
  - AfterDeploymentValidation
  - BeforeShutdown

Extension Lifecycle Events

- BeforeBeanDiscovery
- ProcessModule (new in CDI-1.1)
- ProcessAnnotatedType
  (ev with `@WithAnnotations` in CDI-1.1)
- ProcessInjectionTarget, ProcessBeanAttributes,
  ProcessBean, ProcessProducer, ProcessInjectionPoint
- ProcessInjectionTarget
- AfterBeanDiscovery
- AfterDeploymentValidation
- BeforeShutdown

The AnnotatedType

- Provides access to all important reflection based Information
- Can be modified via CDI Extensions in ProcessAnnotatedType
- Each scanned Class is one AnnotatedType
- Typically changed in ProcessAnnotatedType

Sample: Dynamic Interceptor

- https://github.com/struberg/InterDyn

```
public class InterDynExtension
implements Extension {
public void processAnnotatedType(
    @Observes ProcessAnnotatedType pat) {
 for (InterceptorRule rule : interceptorRules)
  if (beanClassName.matches(rule.getRule())){
    AnnotatedType at = pat.getAnnotatedType();
    at= addInterceptor(at, rule);
     ...
    pat.setAnnotatedType(at);
  }  ...
```

Popular Extensions

- Apache MyFaces CODI
  http://myfaces.apache.org/extensions/cdi
- JBoss Seam3
  http://seamframework.org/Seam3
- CDISource
  https://github.com/cdisource
- Apache DeltaSpike
  https://incubator.apache.org/deltaspike

Vetoing Beans

- ProcessAnnotatedType#veto()
- Can be used to dynamically disable a Bean
- DeltaSpike @Exclude
- DeltaSpike globalAlternatives also uses veto()
  - META-INF/apache.deltaspike
    org.mycomp.MyInterface=org.mycomp.SomeImpl
  - will veto() all other implementations

DeltaSpike @Exclude

- Used to disable bean depending on a certain situation
- @Exclude()
- @Exclude(ifProjectStage=ProjectStage.Development.class)
- @Exclude(exceptIfProjectStage=ProjectStage.Development.class)
- @Exclude(onExpression=”myproperty==somevalue”)

BeanManagerProvider

- Access to the BeanManager in places where it cannot get injected
  - JPA Entity Listeners
  - Tomcat Valves
  - JMX Beans
  - ServletListener or ServletFilter if not running in EE Server
- CDI-1.1 will contain CDI.current()

BeanProvider

- getContextualReference(..)
- #injectFields Used to fill all injection points of a existing instance
- Do not use for creating @Dependent instances!

Important Considerations

- Only the BeanManager can get injected into an Extension during boot.
- Injection can be used inside Bean<T> implementations, Interceptors, .. **but** only **after** the container started!
  - Use lazy loading instead

DeltaSpike @ConfigProperty

- Technically a CDI Qualifier
- Used to inject configuration in a typesafe way
- Produces @Dependent results and thus can use the InjectionPoint
- → code: ConfigPropertyProducer
- Register own ConfigSources via serviceloader or

```
public class MyConfig implements PropertyFileConfig {
  public String getPropertyFileName() {
    return "myown.properties";
  }
}
```

The Bean lookup

- See BeanProvider#getContextualReference
- BeanManager#getBeans(type, qualifiers) gets all Beans which can
  - serve the type of the Injection Point and
  - have the given qualifiers
  - are not disabled via @Alternatives or @Specializes
- BeanManager#resolve()

Legal stuff

- Apache, OpenWebBeans, MyFaces, OpenEJB, TomEE and OpenJPA are trademarks of the Apache Software Foundation
- Seam3 and Weld are trademarks of JBoss Inc
- Java and Glassfish are trademarks of Oracle Inc
- CanDI is a trademark of Caucho Inc