



Business
Technology|Days

BIG
DATA
CON

Mark Struberg / INSO, TU Vienna

CDI-1.1 Deep Dive

About Myself

- struberg@apache.org
- <http://github.com/struberg>
- <http://people.apache.org/~struberg>
- freelancer, programmer since 20 years
- elected Apache Software Foundation member
- Apache OpenWebBeans PMC + PMC and committer in other ASF projects: MyFaces, BVAL, OpenJPA, Maven, DeltaSpike, ...
- CDI EG member

How the ASF works

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

Agenda

- Short CDI Overview
- The Container Lifecycle
- Behaviour Change wo Extensions:
Alternatives, Specializes, Interceptors
- Extension basics
- Sample: Dynamic Interceptor
- Sample: @ViewScoped Context

JSR-346 Overview

- Follow up for JSR-299 CDI-1.0
- Contexts and Dependency Injection for the Java EE platform (CDI)
- Component Model for Java (SE and EE)
- 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

CDI Information

- <http://www.cdi-spec.org/>
- <http://jcp.org/en/jsr/proposalDetails?id=299>
- <http://jcp.org/en/jsr/proposalDetails?id=346>

Available Implementations

- CDI-1.0
 - JBoss Weld (RI)
 - Apache OpenWebBeans
 - Resin CanDI
- CDI-1.1
 - Weld-2.0-alpha preview release
 - OpenWebBeans-2.0.x in preparation
- maybe Spring in the future?

A small Example

- Create a MailService implementation

@ApplicationScoped

```
public class MyMailService
implements MailService {
    private @Inject User usr;
    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>
```

CDI-1.1 Bean Discovery Modes

- META-INF/beans.xml marker file
 - Mandatory in CDI-1.0
 - if empty: discover: all
 - Optional in CDI-1.1! discover: annotated
- `<bean-discovery-mode>` (version="1.1")
 - **all**: like in CDI-1.0
 - **annotated**: no unannotated `@Dependent`
 - **none**: ignore this JAR

Terms – „Bean“

- The term Bean in CDI means a Java Class and all it's rules to create contextual instances of that bean.

- Interface

`Bean<T> extends Contextual<T>`
in CDI-1.1: `BeanAttributes`

- 'Beans' in JSR-299 and JSR-346 doesn't mean JavaBeans!

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 doSthg(parm) {  
    getInstance().doSthg();  
}  
  
private T getInstance() {  
    beanManager.getContext().get(bean, ...);  
}
```

- Can contain Interceptor logic

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

The big picture

- Creating the meta information at startup
 - Classpath Scanning and creating Managed Beans meta-information
- 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

Built-in Qualifiers

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

@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")..`

Own Qualifiers

- Annotation
- `@Qualifier` meta-annotation

`@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();  
}
```

- CDI-1.1: areQualifiersEquivalent, areInterceptorBindingsEquivalent + hashCode

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);}
    }
}
```

Producer Pitfalls

- Each producer method / producer field forms an own Bean<T>!
- 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
- Only Producer for @Dependent can return null

Behaviour Change with Standard CDI

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 {  
}
```

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

- CDI-1.0: Interceptors need to be **enabled manually** in beans.xml
- Only active in the current Bean Archive!
- You can define the **order** in which multiple Interceptors stack

```
<beans>  
  <interceptors>  
    <class>org.mycomp.Secured</class>  
    <class>org.mycomp.Transactionnal</class>  
  </interceptors>  
</beans>
```


Interceptors in CDI-1.1

- new: Constructor interceptors
- Interceptors with lower `@java.annotation.Priority` are called first

```
@Monitored @Interceptor @Priority(100)
```

```
public class MonitoringInterceptor {
```

```
    @AroundInvoke
```

```
    public Object monitorInvocation(InvocationContext ctx)
```

```
        throws Exception { ... }
```

- Order can only be changed via `AfterTypeDiscovery`

@Specializes

- 'Replaces' the Managed Bean of a superclass
- Bean must extend the superclass
- possible to specialize 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)
- This got changed in CDI-1.1
see <https://issues.jboss.org/browse/CDI-18>
- OpenWebBeans always used global enablement
- Alternatives with @Priority globally enabled; highest priority wins.

Events

- Observer/Observable pattern
- Inject a Event Source

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

- Event Consumer

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

- Attention: private observer methods don't work in all container

Improved Event in CDI-1.1

- **Generics** handling got fixed/clarified
- **EventMetadata** injection point to programmatically inspect information about the sender

@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) {..}  
}
```


CDI-1.1 @Vetoed

- If applied to a Class, this Class will get ignored. No way to remove it programmatically
- If on a package the whole package will get ignored. Not for subpackages though
- NoClassDefFound -> picked up?
<exclude><if-class-not-available>.

CDI-1.1 bean discovery

- Scanning can be tuned in beans.xml
- `<scan>`
- `<exclude>` (**no** `<include>`)
 - unconditionally:
`<exclude name="com.acme.**">`
 - `<if-class-available>`
 - `<if-class-not-available>`
 - `<if-system-property>`

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!

Extension Lifecycle Events

- Extensions picked up and instantiated at Container startup
- Container sends 'system events' which all Extensions can @Observes, e.g:
 - BeforeBeanDiscovery
 - ProcessAnnotatedType
 - ProcessInjectionTarget
 - AfterBeanDiscovery
 - AfterDeploymentValidation
 - BeforeShutdown

New CDI-1.1 Lifecycle Events

- `AfterTypeDiscovery`
 - allows modifying Alternatives, Interceptors and Decorators
- `ProcessBeanAttributes` (1.0: only `ProcessBean`)
- `ProcessInjectionPoint`
- `ProcessAnnotatedType` with `@WithAnnotations(Annotation[])`
- `ProcessSyntheticAnnotatedType`

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

The AnnotatedType

- Provides access to all important reflection based Information
- Can be modified via CDI Extensions in `ProcessAnnotatedType`
- Each scanned Class is one `AnnotatedType`
- `@since CDI-1.1:`
`BBD#addAnnotatedType(Class, String)`
- 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://cwiki.apache.org/confluence/display/DeltaSpike/Index>
<http://incubator.apache.org/deltaspike>

Vetoing Beans

- `ProcessAnnotatedType#veto()`
- Can be used to dynamically disable a Bean
- `DeltaSpike#Exclude`

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

BeanProvider

- #injectFields Used to fill all injection points of a existing instance

Implementing an own Scope

- via Scope annotation:

```
@Target( { TYPE, METHOD, FIELD} )  
@Retention(RUNTIME)  
@NormalScope (passivating=true)  
@Inherited  
public @interface SpecialScoped  
{ }
```


Programmatically add Scope

- add random annotation as Scope via System Event

```
public void addViewScoped(  
    @Observes BeforeBeanDiscovery bbd) {  
    // Scope class, normal, passivating  
    bbd.addScope(ViewScoped.class, true, true);  
}
```

Register the Context

- You need to tell the CDI container which Context should be serve this Scope

```
public void registerViewContext (@Observes  
    AfterBeanDiscovery abd) {  
    abd.addContext (new ViewScopedContext ());  
}
```

Implementing the Context

- A Context stores the 'Contextual Instances'
- Implements

```
javax.enterprise.context.spi.Context
```

The needed Interface

- `getScope()` → defines the Scope the Context serves
- `isActive()` → `ContextNotActiveException`
- `BeanManager#createCreationalContext`
- `get()` with and wo `CreationalContext`
- `Contextual#create(CreationalContext)`
- `Contextual#destroy(instance, cc)`

The CreationalContext

- Stores all Explicit @Dependent Objects:
 - injected @Dependent Objects
 - Instance<T> for @Dependent objects since CDI-1.1
- Also stores all Implicit @Dependents:
 - Interceptors
 - Decorators
- needed to destroy a Contextual Instance

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()`

'BeanBags'

- Contextual Instances will be looked up via it's Bean extends Contextual
- For each Contextual Instance stored in the Context you need:
 - The actually Contextual Instance
 - The Bean
 - The CreationalContext
- See DeltaSpike AbstractContext

Lifecycle Activation

- Store the 'current key' of the Context for a Thread to set `Context#isActive()`
- via `ServletRequestListener`, `HttpSessionListener`, etc
- Needs to take care of async-supported in Servlet-3.0

Lifecycle Ending

- uses requestDestroyed, sessionDestroyed, JSF destroy events etc
- properly destroy all Contextual instances if the container shuts down
- Possible via BeforeShutdown

Legal stuff

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