



**Contexts Dependency Injection for the Java EE Platform (JSR-299)  
aka WebBeans  
von Rolf Kulemann**

- 1. „dependency injection“ allgemein**
- 2. Exkurs: IoC – Inversion of Control**
- 3. „dependency injection“ und Java**
- 4. Überblick CDI Architektur**
- 5. CDI Grundlagen anhand CDI Beispiel ohne Java EE**
  1. @Inject und @Produce
  2. Interceptors
  3. Decorators
  4. Events
- 6. Weiterführende Themen**
  1. Stereotypes
  2. Qualifiers
  3. Scopes and Contexts
  4. Portable Extensions
- 7. Vergleich CDI und Springframework**
- 8. Lektüre zu CDI**
- 9. Komplexes JSF CDI Beispiel mit Java EE**

- **Dependency Injection (DI) ist ein Entwurfsmuster und dient in einem objektorientierten System dazu, die Abhängigkeiten zwischen Komponenten oder Objekten zu minimieren.**
- **Dependency Injection ist eine Anwendung des Prinzips der Inversion of Control (IoC), bezieht sich aber nur auf die Erzeugung und Initialisierung von Objekten. Sie kann als Verallgemeinerung der Fabrikmethoden verstanden werden.**
- **In einem klassisch aufgebauten OO-System ist jedes Objekt selbst dafür zuständig, seine Abhängigkeiten, also benötigte Objekte und Ressourcen, zu erzeugen und zu verwalten. Dafür muss jedes Objekt einige Kenntnisse seiner Umgebung mitbringen, die es zur Erfüllung seiner eigentlichen Aufgabe normalerweise nicht benötigen würde. Insbesondere muss es, um die entsprechenden Objekte erzeugen zu können, ihre konkrete Implementierung kennen.**

- **Dependency Injection überträgt die Verantwortung für das Erzeugen und die Verknüpfung von Objekten an ein extern konfigurierbares Framework, entsprechend einem Komponentenmodell. Dadurch wird der Code des Objektes unabhängig von seiner Umgebung und von der konkreten Umsetzung der Klassen, die es benötigt. Das vermeidet unnötige Abhängigkeiten beim Kompilieren und erleichtert besonders die Erstellung von Unit-Tests.**
- **Nachteilig wirkt sich hingegen aus, dass so Programmlogik in Konfigurationsdateien ausgelagert wird, was die Übersichtlichkeit vermindern und die Wartung erschweren kann: die Entwickler müssen nun zum Verstehen des Codes noch die Konfiguration berücksichtigen, welche sich zudem manchen Hilfsmitteln der Codeanalyse (z.B. IDE-unterstütztes Finden von Abhängigkeiten oder Refactoring) entzieht.**

```
public class ServerFacade {  
    public Object respondToRequest(Object pRequest) {  
        if(businessLayer.validateRequest(pRequest)) {  
            DAO.getData(pRequest);  
            return Aspect.convertData(pRequest);  
        }  
        return null;  
    }  
}
```

```
public class ServerFacade {  
    public Object respondToRequest(Object pRequest) {  
        return DAO.getData(pRequest);  
    }  
}
```

- **Im Java Umfeld werden Anwendungen, welche DI nutzen i.d.R. über XML Dateien und/oder Annotations konfiguriert**
- **Siehe auch JSR 330 - Dependency Injection for Java**
- **und JSR 299 - Common Dependency Injection**

- This specification defines a powerful set of complementary services that help improve the structure of application code.
- A well-defined lifecycle for stateful objects bound to *lifecycle contexts*, where *the set of contexts is extensible*
- A sophisticated, typesafe *dependency injection mechanism*, including the *ability to select dependencies at either development*
- *or deployment time, without verbose configuration*
- Integration with the Unified Expression Language (EL), allowing any contextual object to be used directly within a JSF or JSP page
- The ability to *decorate injected objects*
- The ability to associate *interceptors* to objects via typesafe *interceptor bindings*
- An *event notification model*
- A *web conversation context in addition to the three standard web contexts defined by the Java Servlets specification*
- An SPI allowing *portable extensions to integrate cleanly with the container*

- **Problem: Es gibt nicht nur Webanwendungen. Auch Normale Java Anwendungen sollten CDI nutzen können**
- **Lösung: WELD ist die Referenzimplementierung von CDI. Diese kann auch ohne Java EE genutzt werden**
- **Damit Beans/Komponenten über CDI genutzt werden können, muss folgende Datei existieren:**
- **META-INF/beans.xml**

# CDI Grundlagen am Beispiel @Inject und @Produce



- **Jede Javaklasse ist eine Komponente, deren Instanzen durch den CDI Container verwaltet werden**
- **Grundsätzlich sollte jede Klasse die JavaBean Konventionen erfüllen -> EL**
- **Annotation @Inject: Markiert Instanzvariable, Konstruktor oder Setter zur Injektion durch den CDI Container**
- **Annotation @Produce: Manche zu injizierenden Abhängigkeiten müssen kontextabhängig erzeugt werden**

# CDI Grundlagen am Beispiel Interceptors



- **Interceptoren sind klassische leichtgewichtige Aspekte, die per Konfiguration auf Java Objekte angewendet werden, ohne dass das Ziel etwas davon weiß**
- **In CDI weiß das Ziel leider vom Interceptor(binding), da Annotationen verwendet werden müssen! -> Nachteil, vergl. Springframework**

**Um einen Interceptor zu verwenden, benötigt man**

- **Interceptor – Implementierung eines Aspektes**
- **Interceptor Binding – Marker Annotation, verwendet im Interceptor und im Ziel**
- **Ziel – Objekt/oder Methode dessen Aufrufe „intercepted“ werden sollen**
- **Interceptor muss in der beans.xml deklariert sein, damit er wirklich aktiv ist**

- **Im Vergleich zu einem Interceptor, kann ein Dekorierer auf beliebig viele Objekte angewendet werden, auch ohne, dass Ziele vom Dekorierer wissen**

**Um einen Dekorierer zu verwenden, benötigt man**

- **Dekorierer – i.d.R. implementiert per Delegation**
- **Eintrag in der beans.xml**
- **Unterscheid zum Interceptor: Interceptoren agieren im Kontext der Methode, welche aufgerufen wird. Dekorierer agieren auf Interface Ebene!**

- **Dependency injection enables loose-coupling by allowing the implementation of the injected bean type to vary, either at deployment time or runtime. Events go one step further, allowing beans to interact with no compile time dependency at all. Event *producers* raise events that are delivered to event *observers* by the container.**
- **This basic schema might sound like the familiar observer/observable pattern, but there are a couple of twists:**
- **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, and**
- **observers can be notified immediately, or can specify that delivery of the event should be delayed until the end of the current transaction.**

Quelle: <http://docs.jboss.org/weld/reference/1.0.0/en-US/html/events.html>

**Um CDI Events zu verwenden, benötigt man**

- **Event Selector – Dient Observern und Producern zur Qualifizierung bestimmter Events zu einer Klasse**
- **Event Observer – empfängt Events für Objekte bzw. Klassen**
- **Event Producer – erzeugt Events zu einem Object**

**Grundsätzlich erreicht man mit dem Observer Muster einen sehr hohen Grad an Entkopplung ,wie auch hier.**

# CDI weiterführende Konzepte

## Stereotypes



- Ein Stereotyp ist eine Annotation, um CDI Metadaten (Annotationen) zu gruppieren und wiederzuverwenden

***@RequestScoped***

***@Secure***

***@Transactional***

***@Stereotype***

***@Target(TYPE)***

***@Retention(RUNTIME)***

***public @interface Action {}***

- Qualifier dienen zum Selektieren bestimmter Implementierungen zur Injektion

**@Synchronous**

```
class SynchronousPaymentProcessor  
implements PaymentProcessor {
```

```
...  
}
```

**@Asynchronous**

```
class AsynchronousPaymentProcessor  
implements PaymentProcessor {
```

```
...  
}
```

```
class Bean {
```

```
    @Inject @Synchronous PaymentProcessor paymentProcessor;  
}
```

### Built-in scope types

- **Es gibt 5 standard Scope Typen, welche im Package `javax.enterprise.context` definiert sind**
- **@RequestScoped, gültig während eines ServletRequests**
- **@ApplicationScoped, gültig innerhalb der Laufzeit einer Anwendung z.B. Webanwendung**
- **@SessionScoped, gültig während einer Web Session**
- **@ConversationScoped, gültig während einer Abfolge von Client/Server Interaktionen (Servlet Requests)**
- **(@Singleton) ähnelt sehr @ApplicationScoped**
- **Es können eigene Scopes definiert werden. Dazu muss eine "Portable Extension" im Sinne von JSR-299 erstellt werden**
- **Mit jedem Scope ist ein Context verknüpft, welches die Objekte für einen Scope verwaltet**

# CDI weiterführende Konzepte

## Portable Extensions



**„ Portable Extensions“ werden verwendet um einen JSR-299 Container zu erweitern um**

- **Beans, Interceptors und Decorators**
- **Context Implementierungen für eigene Scopes**
- **...**
- **Für die verschiedenen Ansätze müssen Interfaces implementiert werden**

- **Spring ist ein „dependency injection“ basiertes Application Development Framework**
- **CDI beschreibt im wesentlichen den Injektionsmechanismus. Dazu kommen rudimentär die Interceptoren, Events und Dekorierer**
- **Spring integriert heute alle wichtigen 3rd Party Libraries, so dass die „dependency injection“ auch über die Grenzen der eigenen Applikation hinweg funktioniert. Dass auch im nicht J2EE Umfeld!**
- **Spring integriert auch schon JSR-330 und bestimmt auch bald JSR-299**
- **Spring könnte eine der führenden JSR-299 Implementierungen werden**

**Um bei der Anwendungsentwicklung hohe Produktivitätssteigerungen zu erzielen, stehen auf meiner Liste ganz oben**

- **SSD Festplatte**
- **Ein „dependency injection“ basiertes Entwicklungsframework wie Spring oder CDI**

1. [http://de.wikipedia.org/wiki/Dependency\\_Injection](http://de.wikipedia.org/wiki/Dependency_Injection)
2. <http://martinfowler.com/articles/injection.html>
3. JSR-299 -  
*<http://jcp.org/aboutJava/communityprocess/final/jsr299/index.html>*
4. Weld JSR-299 Reference Implementation -  
<http://docs.jboss.org/weld/reference/latest/en-US/html/>