CDI

Context and Dependency Injection

https://www.bookstack.cn/read/quarkus-v1.0-en/dd95a4ecd244e671.md

https://docs.jboss.org/cdi/spec/2.0/cdi-spec.html

Geschichte/Hintergrund

- Aufgabe von CDI:
  - Objekte und ihre Abhängigkeiten verwalten
  - Objekte und Abhängigkeiten mit Lebenszyklus versehen

Geschichte/Hintergrund

- CDI 1.0 beschränkte sich stark auf die JSF-Welt
- Ursprünglicher Ansatz für Java EE kommt vom Seam-Framework
- In JSR-299 formalisiert, aktuelle Version ist im JSR-346 beschrieben
- Nicht monolithisch, kann durch eigenen Kontext-Provider erweitert werden

Geschichte/Hintergrund

- Grundproblem: Abhängigkeiten verwalten
- In der Vergangenheit:
  - Feste Verdrahtungen von Komponenten
  - Unflexible und starre Applikationsstruktur
- SPRING-Framework:
  - Abhängigkeitsmanagement leicht gemacht
  - Konfiguration per XML und später per Annotation
  - Weite Verbreitung und starke Nutzung im Nicht-EE-Umfeld

Fixe „Verdrahtung“

```java
public class KontakteProvider {

    public List<String> getKontaktNamen() {
        return new ArrayList<>(
                Arrays.asList(
                        new String[] {"Susi", "Hansi", "Maxi"}));
    }
}
```

```java
public class KontakteManager {

    private final KontakteProvider kontakteProvider;

    public KontakteManager() {
        kontakteProvider = new KontakteProvider();
    }

    public List<String> getKontakteNamen() {
        return kontakteProvider.getKontaktNamen();
    }
}
```

CDI aktivieren

- CDI wird in allen Java EE 7-fähigen Servlet-Containern unterstützt
- ~~CDI ist erst dann aktiviert, wenn es eine Datei *beans.xml*~~ gibt
- @*Inject*-Annotation definiert, wo per CDI eine Komponente injiziert werden soll

If you specify Java EE 7 Web as the Java EE version, CDI 1.1 is enabled by default and the beans.xml file is not required. In Java EE 7, when no beans.xml is present the archive that is deployed is an implicit bean archive.

Falls doch eine beans.xml gewünscht ist

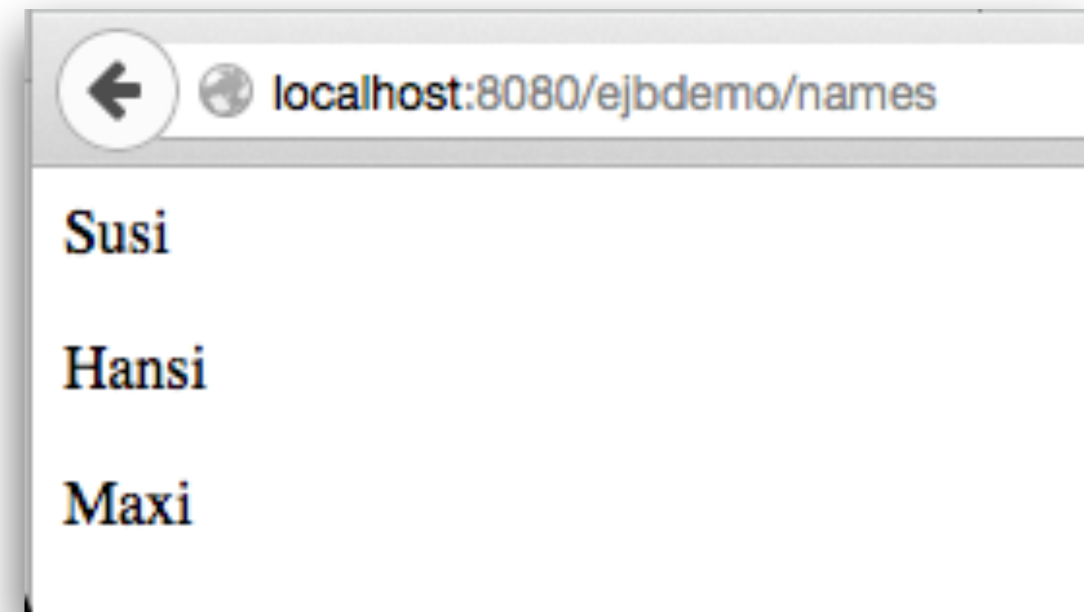
beans.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
                        http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
    bean-discovery-mode="all">
</beans>
```

Verwendung von CDI

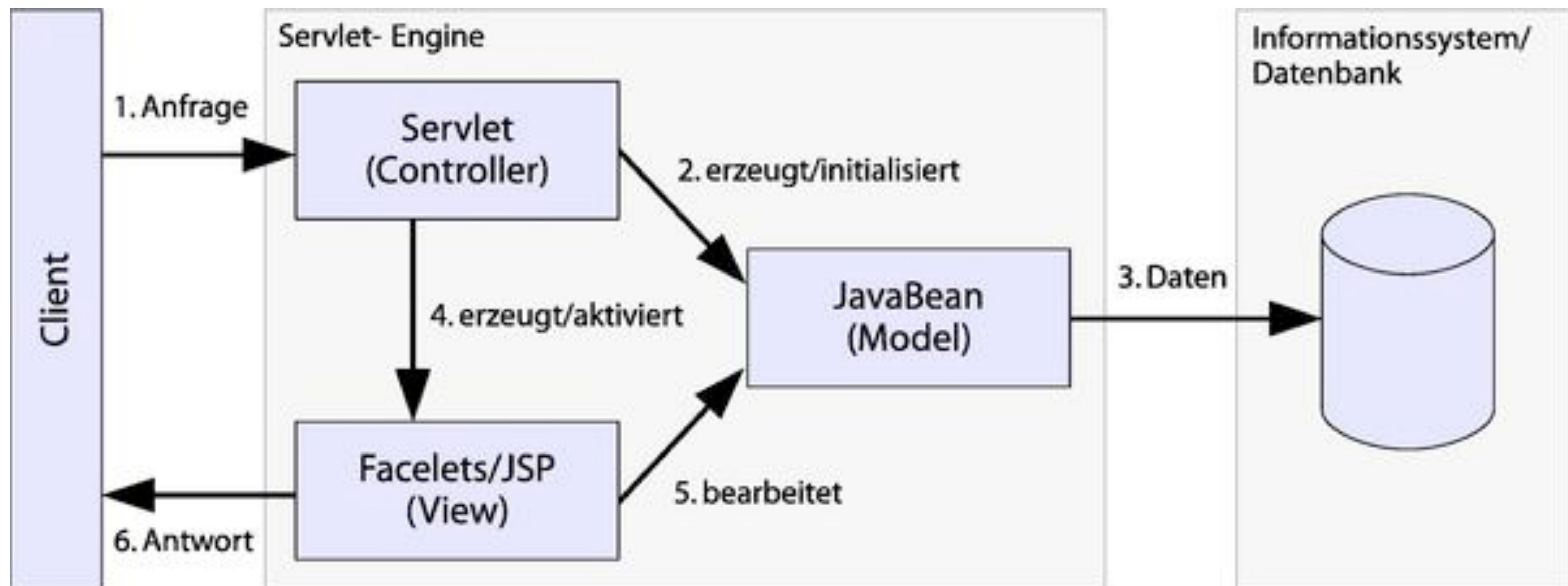
```java
public class KontakteManager {

    //private final KontakteProvider kontakteProvider;
    @Inject
    private KontakteProvider kontakteProvider;

    public KontakteManager() {

    }

    public List<String> getKontakteNamen() {
        return kontakteProvider.getKontaktNamen();
    }
}
```

Ausgabe mit Servlet




```
import javax.inject.Inject;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

@WebServlet("/names")
public class KontakteServlet extends HttpServlet {

    @Inject
    private KontakteProvider kontakteProvider;

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        for (String s : kontakteProvider.getKontaktNamen()) {
            resp.getWriter().write("<p>"+s+"</p>");
        }
    }
}
```

Exkurs: Servlets, JSP, JSF



Injizieren von Abhängigkeiten

- Wie kann injiziert werden:
  - @Inject-Annotation gewünschter Ebene notieren
- Wenn *Bean Discovery Mode* den Wert *annotated* hat:
  - Jede einzelne zu injizierende Bean muss mit einer CDI-Annotation versehen sein!

Injizieren von Abhängigkeiten

- Wichtigste Regel:
  - Zu injizierende Bean muss eindeutig bestimmbar sein
  - Gibt eine Exception, wenn dies nicht möglich ist

```
Ambiguous dependencies for type KontakteProvider with qualifiers @Default
```

Qualifier

- Qualifier erlauben es, Beans eindeutig unterscheidbar zu machen
- Qualifier sind Annotationen
- Qualifier sind annotiert mit der @*Qualifier*-Annotation

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ ElementType.TYPE, ElementType.CONSTRUCTOR, ElementType.FIELD, ElementType.METHOD })
public @interface Test {

}
```

Abgeleitete Klasse mit eigenem Qualifier

```java
@Test
public class TestKontakteProvider extends KontakteProvider {

    @Override
    public List<String> getKontaktNamen() {

        ArrayList<String> namen = new ArrayList<String>();
        namen.add("Peter");
        namen.add("Tester");
        namen.add("Sonnenschein");

        return namen;
    }
```

Nun funktioniert's

```
public class KontakteManager {

    private KontakteProvider kontakteProvider;

    @Inject @Test
    public KontakteManager(KontakteProvider kontakteProvider) {
        this.kontakteProvider = kontakteProvider;
    }

    public List<String> getKontakteNamen() {
        return kontakteProvider.getKontaktNamen();
    }

}
```

Vordefinierte Qualifier

- **@Default**
  Ist implizit auf jeder Bean vorhanden, Ausnahme: @*Named*

- **@Any**
  Ist implizit auf jeder Bean vorhanden

- **@Named**
  Definiert, dass die Bean per EL verwendbar ist

- **@New**
  Veraltet, sollte nicht mehr verwendet werden, erzeugt bei jeder
  Abrufen eine neue Instanz

Gültigkeitsbereiche und Kontexte

Gültigkeitsbereiche und Kontexte

- Eine Bean befindet sich in einem Gültigkeitsbereich und ist assoziiert mit einem Kontext
  - Der Kontext ist verantwortlich für Lebenszyklus und Sichtbarkeit
  - Eine Bean wird immer einmal für einen Gültigkeitsbereich erzeugt und dann wiederverwendet
  - Nutzende Komponenten müssen sich um den Gültigkeitsbereich nicht kümmern
  - Gültigkeitsbereiche sind auf Bean- und Produzentenebene definierbar

Gültigkeitsbereiche

- **@RequestScoped**
  Bean ist an einen Web-Request gebunden
- **@SessionScoped**
  Bean ist an eine Benutzersitzung gebunden
- **@ApplicationScoped**
  Bean ist an die Applikation gebunden
- **@ConversationScoped**
  Mischung zwischen Request- und Session-Scopes
- **@Dependent**
  Standard, Bean existiert in keinem anderen Gültigkeitsbereich
- **@Singleton**
  Bean existiert genau einmal
- **@New**
  Veraltet, @Dependent verwenden

@ViewScoped

Gültigkeitsbereiche

- Beans in einem Gültigkeitsbereich werden nie direkt referenziert
  - Proxy wird von der Umgebung erzeugt und zugewiesen
  - Proxy kümmert sich um Interna
- @Dependent/@Singleton
  - Client hält die Referenz direkt
  - Bean wird niemals mehrfach verwendet
- @Dependent
  - Bei EL-Ausdrücken wird bei **jedem** Zugriff eine neue Instanz erstellt!
- @Singleton
  - Sollte nicht verwendet werden, @ApplicationScoped nutzen

Produzenten

Produzenten

- Ohne Produzent: Statische Erzeugung
  - Keine direkte Konfiguration der erzeugten Instanz möglich
- Oftmals werden dynamische Ansätze benötigt:
  - Konkreter Typ einer Bean erst zur Laufzeit bekannt
  - Objekt ist gar keine Bean (String)
  - Objekt muss "von Hand" erzeugt werden
- Umsetzung des Factory-Pattern
- Polymorphie

Produzenten

- Produzenten sind stets Methoden oder Felder
- Werden mit der @*Produces*-Annotation versehen

Produzenten

- Injizierung dann wie gehabt:

```
@Inject List<String> list;
```

- Achtung: Nicht eindeutige Produzenten-Kennzeichnungen können zu Fehlern führen – auch Produzenten müssen eindeutig unterscheidbar sein

Disposer

- @Disposes-Annotation kennzeichnet eine Methode zum Aufräumen
  - Wird mit einem Produzenten verwendet
  - Wird **automatisch** aufgerufen, wenn der Lebenszyklus der erzeugten Bean endet

Disposer – Beispiel

```java
@PersistenceContext
private EntityManager em;

@Produces
@UserDatabase
public EntityManager create() {
    return em;
}

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

Der EntityManager wird als @UserDatabase zur Verfügung gestellt

Was wurde nicht besprochen?

- Alternativen @Alternative
- …

Interzeptoren

Interzeptoren

- Interzeptoren werden verwendet, um übergreifende Anforderunge umzusetzen:
  - Logging
  - Transaction-Handling
  - Sicherheit, ...
- Drei Arten von Interzeptoren:
  - Geschäftsmethoden
  - Lebenszyklus
  - Timeout

Interzeptoren

- Interzeptor wird in eigener Klasse definiert
- Am Kopf stehen die @*Interceptor*-Annotation und die Annotation, die als Referenz auf den Interzeptor dient
- Eigentliche Ausführungsart wird mit Annotation angegeben

Interzeptoren

- Mehrere Interzeptoren dürfen das gleiche @Interceptor-Binding benutzen
- Ausführungsreihenfolge ist undefiniert, kann mit *@Priority* gesteuert werden
- Interzeptoren müssen in *beans.xml* deklariert sein ODER die *@Priority*-Annotation besitzen
  - Trifft beides nicht zu, wird der Interzeptor ignoriert
  - Interzeptoren, die in *beans.xml* deklariert sind, werden nur für Klassen im selben Archiv ausgeführt
  - Interzeptoren mit *@Priority*-Annotation sind global aktiv

Arten der Ausführung

- **@AroundInvoke**
  Vor und nach Methode der Zielinstanz
- **@AroundConstruct**
  Vor und nach Konstruktor der Zielinstanz
- **@PostConstruct**
  Nach dem Durchlaufen des Konstruktors
- **@PreDestroy**
  Vor der Zerstörung der Zielinstanz
- **@PrePassivate**
  Bevor eine Session-Bean passiviert wird
- **@PostActivate**
  Nachdem eine Session-Bean passiviert wird

InterceptorBinding

```
import javax.interceptor.InterceptorBinding;
import java.lang.annotation.*;

@Inherited
@InterceptorBinding
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Logging {
}
```

Interceptor

```java
import javax.annotation.Priority;
import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;

@Interceptor
@Logging
@Priority(Interceptor.Priority.APPLICATION + 200)
public class LoggingInterceptor {

    @AroundInvoke
    public Object log(InvocationContext context) throws Exception {
        String name = context.getMethod().getName();
        System.out.println("==> " + name);

        Object result = context.proceed();

        System.out.println("<== " + name);

        return result;
    }
}
```

Anwendung

```java
public class KontakteProvider {

    @Logging
    public List<String> getKontaktNamen() {
        System.out.println("== getKontakteNamen ==");
        List<String> names = new ArrayList<>();
        names.add("Susi");
        names.add("Hansi");
        names.add("Maxi");
        names.add(LocalDateTime.now().format(DateTimeFormatter.ofPattern("hh:mm:ss")));
        return names;
    }
}
```

Output

```
 attribute type REQUIRED on SFSB lifecycle method Method init() of class class at.htl.ejbdemo.StatefulBean,
 valid types are REQUIRES_NEW and NOT_SUPPORTED. Method will be treated as NOT_SUPPORTED.
21:56:53,666 WARN  [org.jboss.as.ejb3] (ServerService Thread Pool -- 164) WFLYEJB0463: Invalid transaction
 attribute type REQUIRED on SFSB lifecycle method Method destroy() of class class at.htl.ejbdemo
 .StatefulBean, valid types are REQUIRES_NEW and NOT_SUPPORTED. Method will be treated as NOT_SUPPORTED.
21:56:53,690 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-5) WFLYUT0021: Registered web
 context: /ejbdemo
21:56:53,696 INFO  [org.jboss.as.server] (management-handler-thread - 43) WFLYSRV0010: Deployed "ejbdemo"
 (runtime-name : "ejbdemo.war")
[2015-05-17 09:56:53,727] Artifact EjbDemo:war exploded: Artifact is deployed successfully
[2015-05-17 09:56:53,727] Artifact EjbDemo:war exploded: Deploy took 356 milliseconds
21:56:57,271 INFO  [stdout] (default task-70) ==> getKontaktNamen
21:56:57,272 INFO  [stdout] (default task-70) == getKontakteNamen ==
21:56:57,272 INFO  [stdout] (default task-70) <== getKontaktNamen
```

Dekoratoren

Dekoratoren

- Interzeptoren sind für verschiedene Klassen zuständig
- Dekoratoren sind für einzelne Klassen zuständig
- Dekoratoren implementieren Interfaces
  - Müssen diese Interfaces jedoch nicht vollständig implementieren
  - Können deshalb auch *abstract* sein

Dekoratoren

- Implementierung eines Dekorators:

```
@Decorator
public abstract class MyDecorator implements Greeting {

    @Inject
    @Delegate
    @Any
    private Greeting greeting;

    public String greet(String name) {
        return "Decorated: " + greeting.greet();
    }
}
```

Dekoratoren

- Dekoratoren sind standardmäßig deaktiviert
  - In *beans.xml* registrieren oder
  - *@Priority*-Annotation verwenden
- Interzeptoren werden stets vor Dekoratoren ausgeführt

Ausgangsklasse

```java
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class KontakteProvider implements KontakteHandler {

    @Logging
    public List<String> getKontaktNamen() {
        System.out.println("== getKontakteNamen ==");
        List<String> names = new ArrayList<>();
        names.add("Susi");
        names.add("Hansi");
        names.add("Maxi");
        names.add(LocalDateTime.now().format(DateTimeFormatter.ofPattern("hh:mm:ss")));
        return names;
    }
}
```

Servlet

```
@WebServlet("/names")
public class KontakteServlet extends HttpServlet {

    @Inject
    private KontakteHandler kontakteProvider;

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        for (String s : kontakteProvider.getKontaktNamen()) {
            resp.getWriter().write("<p>"+s+"</p>");
        }
    }
}
```

Es wird gegen das Interface programmiert.

Decorator

```java
import javax.annotation.Priority;
import javax.decorator.Decorator;
import javax.decorator.Delegate;
import javax.enterprise.inject.Any;
import javax.inject.Inject;
import javax.interceptor.Interceptor;
import java.util.List;

@Decorator
@Priority(Interceptor.Priority.APPLICATION + 100)
public class KontakteDecorator implements KontakteHandler {

    @Inject
    @Any
    @Delegate
    private KontakteProvider provider;


    @Override
    public List<String> getKontaktNamen() {
        List<String> namen = provider.getKontaktNamen();
        namen.add("==> DECORATOR >==");
        return namen;
    }
}
```

Events

(Ereignisse)

Ereignisse

- Ereignisse implementieren das Observer-Entwurfsmuster
  - Annotationsgesteuert, deshalb einfach umsetzbar
  - Produzenten produzieren Ereignisse
  - Konsumenten konsumieren Ereignisse
- Ereignisse werden per Event-Interface definiert

Ereignisse

- Definieren und Werfen eines Ereignisses:
  - Zum Transportieren von Zustandsinformationen muss ein State-Objekt definiert werden (POJO)
  - *fire()*-Methode des Events löst aus und nimmt State-Objekt entgegen

```
@Inject @Any Event<Customer> event;

// Fire the event with a Customer-instance
// as state
event.fire(customer);
```

Ereignisse

- Produzenten können Qualifier definieren
  - Konsumenten müssen alle Qualifier besitzen

```
@Inject @Any @Added Event<Customer> event;

void onCustomer(@Observes @Added Customer event) {
    //. . .
}
```

Ereignisse

- Standardverhalten:
  - Wenn Konsument existiert, wird dieser benachrichtigt
  - Wenn Konsument nicht existiert, wird er erzeugt

- Verhalten kann geändert werden:

```
void onCustomer(
        @Observes(
         notifyObserver= Reception.IF_EXISTS)
         @Added Customer event){
    //. . .
}
```

Event<String> hinzugefügt

```
public class TestKontakteProvider extends KontakteProvider {

    private List<String> namen = new ArrayList<>();

    @Inject  @Any
    private Event<String> event;

    @PostConstruct
    private void init() {
        add("Franzi");
        add("Berti");
        add("Fanny");
    }

    private void add(String name) {
        namen.add(name);
        event.fire(name);
    }

    @Override
    public List<String> getKontaktNamen() {
        return namen;
    }
}
```

Klasse KontakteProvider wurde von CDI ausgeschlossen, da sinst ambigous-Error

```
@Vetoed
public class KontakteProvider implements
```

Consumer

```java
import javax.enterprise.event.Observes;

public class Consument {

    private void onelementAdded(@Observes String added) {
        System.out.println("==> " + added + " wurde hinzugefügt!");
    }

}
```

localhost:8080/ejbdemo/names

Franzi

Berti

Fanny

==> DECORATOR >==

Output

```
22:22:23,383 INFO  [org.wildfly.extension.undertow] (MSC service thread 1-12) WFLYUT0021
 context: /ejbdemo
22:22:23,390 INFO  [org.jboss.as.server] (management-handler-thread - 53) WFLYSRV0010: D
 (runtime-name : "ejbdemo.war")
[2015-05-17 10:22:23,423] Artifact EjbDemo:war exploded: Artifact is deployed successfully
[2015-05-17 10:22:23,424] Artifact EjbDemo:war exploded: Deploy took 202 milliseconds
22:22:34,678 INFO  [stdout] (default task-75) ==> Franzi wurde hinzugefügt!
22:22:34,678 INFO  [stdout] (default task-75) ==> Berti wurde hinzugefügt!
22:22:34,679 INFO  [stdout] (default task-75) ==> Fanny wurde hinzugefügt!
```

Noch Fragen?