JEE 5 & 6 – CDI

Einführung in Contexts and Dependency Injection

.consulting .solutions .partnership

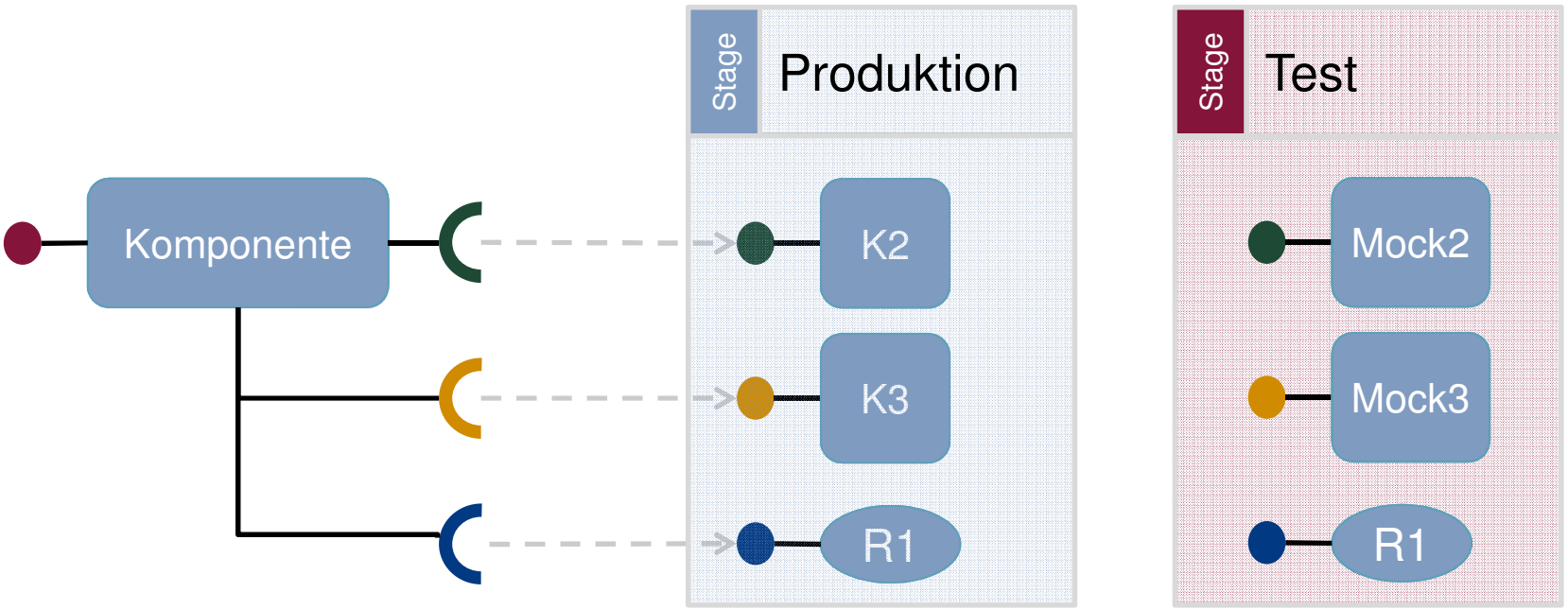
.msg systems

@Inject Agenda

1. **Dependency Injection – Woher kommen wir?** 10
   J2EE, Spring, JEE5, JSRs,…
2. CDI Basics 20
   Annotationen, Factories, Qualifiers, Contexts, …
3. CDI Advanced 20
   @Observes, @Interceptor, @Decorator,…
4. CDI für Profis 10
   Portable Extension SPI

Was ist eigentlich das Problem?

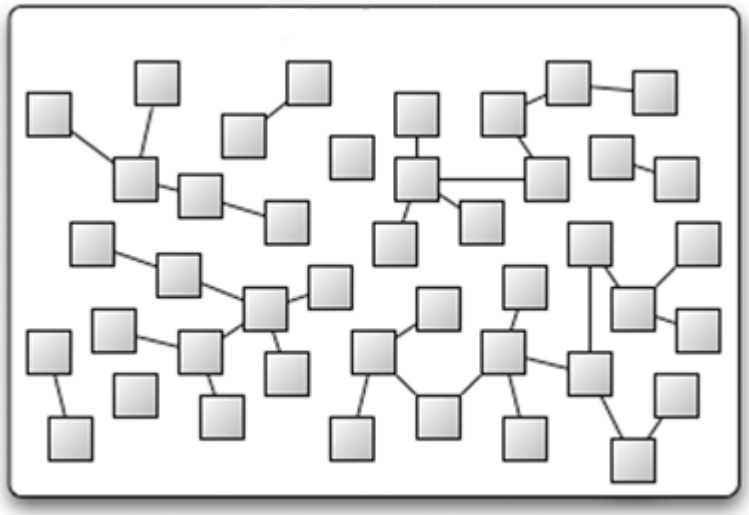
…das Wiring von Komponenten, oder die Antwort auf die Frage: wie kommt man zu seinen Mitspielern?



Was ist eigentlich das Problem?

…das Wiring von Komponenten, oder die Antwort auf die Frage: wie kommt man zu seinen Mitspielern?

Dramatisch wird es bei einer Vielzahl von Verbindungen



Was ist eigentlich das Problem?

…das Wiring von Komponenten, oder die Antwort auf die Frage: wie kommt man zu seinen Mitspielern?

Das kann man naiv so bauen

```
class CampaignServiceImpl implements CampaignService {
    Campaign  updateCampaign(Campaign campaign) throws CampaignException{
    try{
         CampaignDAO campaign = new CampaignDAOImpl();
         ..
         OpportunityDAO oppDAO = new OpportunityDAOImpl();
         // Alternatively, initialize thru factory
         // OpportunityDAO oppDAO = OppFactory.getOpp();
         oppDAO.save(campaign);
      ..
    }catch(Exception e){
    }
}
```

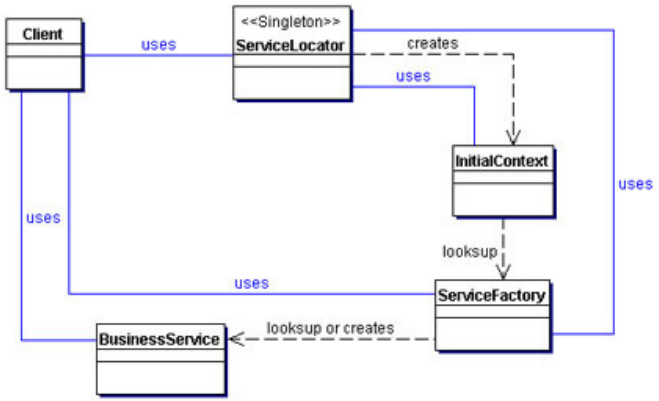
Der Code ist eng gekoppelt und die Business-Logik ist verseucht mit nichtfachlichem Factory-Code!

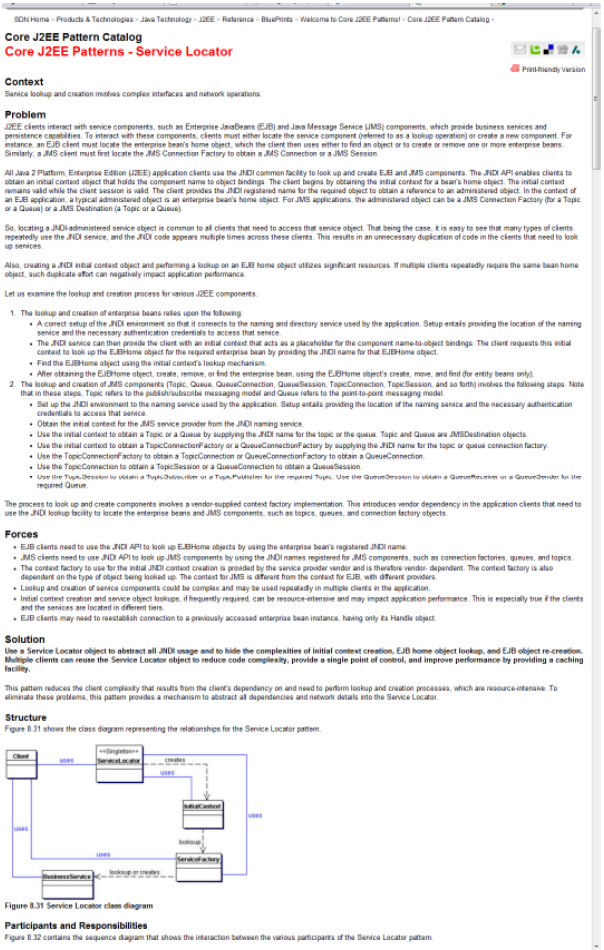
Dependency Injection – Woher kommen wir?

J2EE-Welt – ServiceLocator-Pattern

\-
- Ce/Ca – Metrik
- Lose Kopplung
- Testbarkeit

\+
- JNDI gekapselt





SDN Home > Products & Technologies > Java Technology > J2EE > Reference > BluePrints > Welcome to Core J2EE Patterns! > Core J2EE Pattern Catalog >

Core J2EE Pattern Catalog

Core J2EE Patterns - Service Locator

Print-friendly Version

Context

Service lookup and creation involves complex interfaces and network operations.

Problem

J2EE clients interact with service components, such as Enterprise JavaBeans (EJB) and Java Message Service (JMS) components, which provide business services and persistence capabilities. To interact with these components, clients must either locate the service component (referred to as a lookup operation) or create a new component. For instance, an EJB client must locate the enterprise bean's home object, which the client then uses either to find an object or to create or remove one or more enterprise beans. Similarly, a JMS client must first locate the JMS Connection Factory to obtain a JMS Connection or a JMS Session.

All Java 2 Platform, Enterprise Edition (J2EE) application clients use the JNDI common facility to look up and create EJB and JMS components. The JNDI API enables clients to obtain an initial context object that holds the component name to object bindings. The client begins by obtaining the initial context for a bean's home object. The initial context remains valid while the client session is valid. The client provides the JNDI registered name for the required object to obtain a reference to an administered object. In the context of an EJB application, a typical administered object is an enterprise bean's home object. For JMS applications, the administered object can be a JMS Connection Factory (for a Topic or a Queue) or a JMS Destination (a Topic or a Queue).

So, locating a JNDI-administered service object is common to all clients that need to access that service object. That being the case, it is easy to see that many types of clients repeatedly use the JNDI service, and the JNDI code appears multiple times across these clients. This results in an unnecessary duplication of code in the clients that need to look up services.

Also, creating a JNDI initial context object and performing a lookup on an EJB home object utilizes significant resources. If multiple clients repeatedly require the same bean home object, such duplicate effort can negatively impact application performance.

Let us examine the lookup and creation process for various J2EE components.

1. The lookup and creation of enterprise beans relies upon the following:
   - A correct setup of the JNDI environment so that it connects to the naming and directory service used by the application. Setup entails providing the location of the naming service and the necessary authentication credentials to access that service.
   - The JNDI service can then provide the client with an initial context that acts as a placeholder for the component name-to-object bindings. The client requests this initial context to look up the EJBHome object for the required enterprise bean by providing the JNDI name for that EJBHome object.
   - Find the EJBHome object using the initial context's lookup mechanism.
   - After obtaining the EJBHome object, create, remove, or find the enterprise bean, using the EJBHome object's create, move, and find (for entity beans only).
2. The lookup and creation of JMS components (Topic, Queue, QueueConnection, QueueSession, TopicConnection, TopicSession, and so forth) involves the following steps. Note that in these steps, Topic refers to the publish/subscribe messaging model and Queue refers to the point-to-point messaging model.
   - Set up the JNDI environment to the naming service used by the application. Setup entails providing the location of the naming service and the necessary authentication credentials to access that service.
   - Obtain the initial context for the JMS service provider from the JNDI naming service.
   - Use the initial context to obtain a Topic or a Queue by supplying the JNDI name for the topic or the queue. Topic and Queue are JMSDestination objects.
   - Use the initial context to obtain a TopicConnectionFactory or a QueueConnectionFactory by supplying the JNDI name for the topic or queue connection factory.
   - Use the TopicConnectionFactory to obtain a TopicConnection or QueueConnectionFactory to obtain a QueueConnection.
   - Use the TopicConnection to obtain a TopicSession or a QueueConnection to obtain a QueueSession.
   - Use the TopicSession to obtain a TopicSubscriber or a TopicPublisher for the required Topic. Use the QueueSession to obtain a QueueReceiver or a QueueSender for the required Queue.

The process to look up and create components involves a vendor-supplied context factory implementation. This introduces vendor dependency in the application clients that need to use the JNDI lookup facility to locate the enterprise beans and JMS components, such as topics, queues, and connection factory objects.

Forces

- EJB clients need to use the JNDI API to look up EJBHome objects by using the enterprise bean's registered JNDI name.
- JMS clients need to use JNDI API to look up JMS components by using the JNDI names registered for JMS components, such as connection factories, queues, and topics.
- The context factory to use for the initial JNDI context creation is provided by the service provider vendor and is therefore vendor- dependent. The context factory is also dependent on the type of object being looked up. The context for JMS is different from the context for EJB, with different providers.
- Lookup and creation of service components could be complex and may be used repeatedly in multiple clients in the application.
- Initial context creation and service object lookups, if frequently required, can be resource-intensive and may impact application performance. This is especially true if the clients and the services are located in different tiers.
- EJB clients may need to reestablish connection to a previously accessed enterprise bean instance, having only its Handle object.

Solution

Use a Service Locator object to abstract all JNDI usage and to hide the complexities of initial context creation, EJB home object lookup, and EJB object re-creation. Multiple clients can reuse the Service Locator object to reduce code complexity, provide a single point of control, and improve performance by providing a caching facility.

This pattern reduces the client complexity that results from the client's dependency on and need to perform lookup and creation processes, which are resource-intensive. To eliminate these problems, this pattern provides a mechanism to abstract all dependencies and network details into the Service Locator.

Structure

Figure 8.31 shows the class diagram representing the relationships for the Service Locator pattern.

Figure 8.31 Service Locator class diagram

Participants and Responsibilities

Figure 8.32 contains the sequence diagram that shows the interaction between the various participants of the Service Locator pattern.

Dependency Injection – Woher kommen wir?

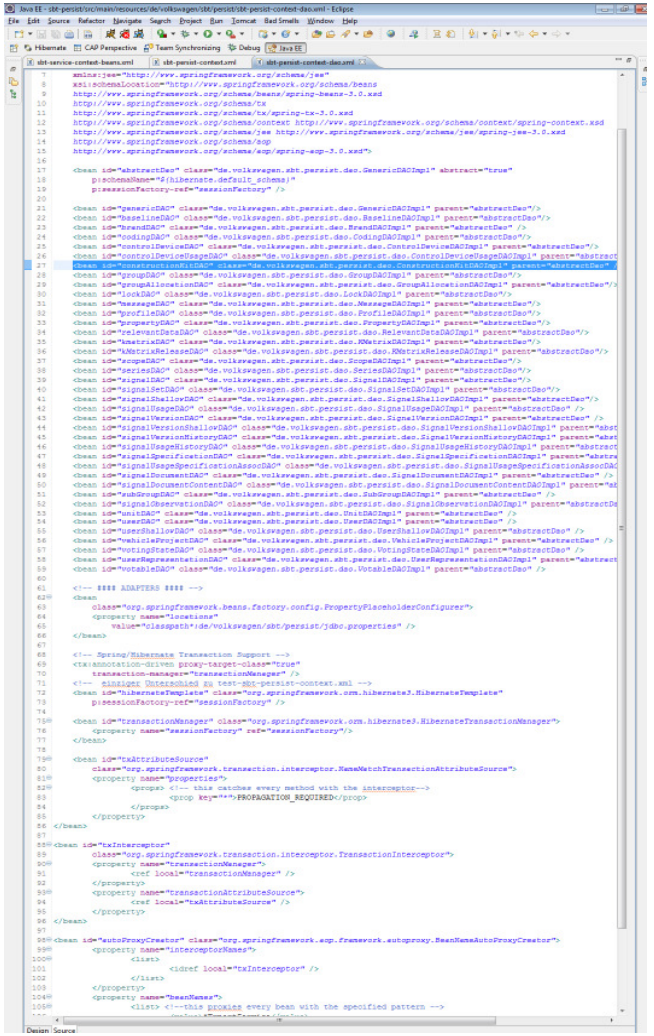
J2EE-Welt – Spring…

-

- J2EE ohne J2EE
- XML Konfig-Wüste

+

- Hollywood - DI
- Unterstützt AOP
- Vorlagen

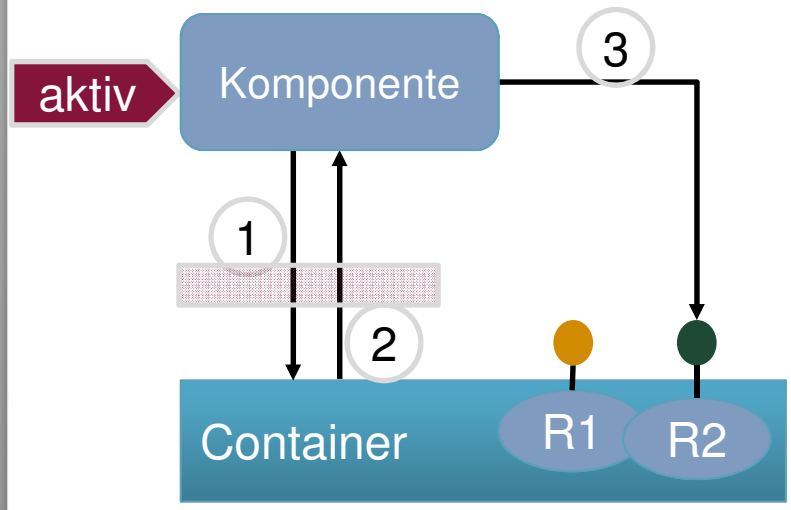


…und andere

- Google Guice
- PicoContainer

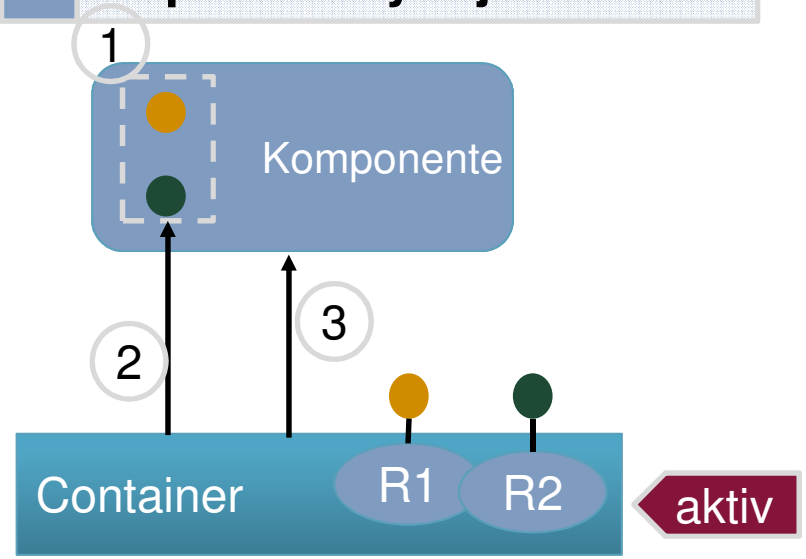
DI – Prinzip: Don‘t call us, we call you…“I“ für Inversion

- Service Locator



1. Die Komponente teilt dem Container mit, dass sie die Ressourcen R1 und R2 benötigt
2. Der Container stellt beide bereit
3. Die Komponente verwendet R1 und R2

+ Dependency Injection

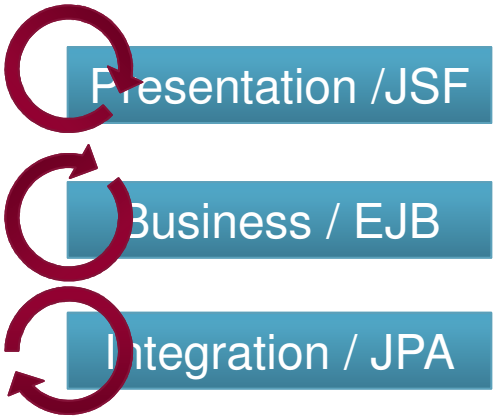


1. Die Komponente **deklariert**, dass sie etwas, das wie R1 und R2 aussieht benötigt.
2. Der Container analysiert die Komponente und ermittelt deren Bedarf
3. Der Container stellt die richtigen Implementierungen zu.

JEE 5 – DI endlich angekommen

\-
- Schlecht integriert
- Beschränkt (nur JNDI Resourcen)
- Kein AOP

\+
- Teil des Standards
- @-basiert

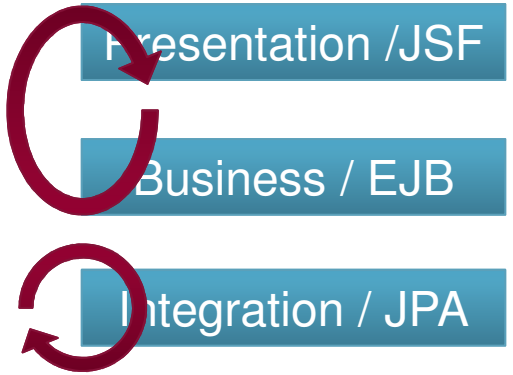



```
@Stateless
@Local(TxTransactionService.class)
class TxTransactionServiceBean {
      @EJB
      CorebankingPaymentUC corebankingPaymentUC;

      @EJB
      TxCrud txCrud;
```

JEE 6 – CDI

-	▪ Keine 100%-Durchdringung
+	▪ Teil des Standards ▪ Gute Integration über Schichten ▪ Unterstützt AOP ▪ Auch in J2SE verwendbar ▪ Erweiterbar ▪ Contexts!



JEE 6 – CDI – warum 2 JSRs?

JSR-330 Dependency Injection for Java

- Rod Johnson (SpringSource), Bob Lee (Google Inc.)
- Definition der DI Basis in Java
- Neu: @Named

JSR-299 Java Contexts and Dependency Injection

- Gavin King (Red Hat Inc)
- Nimmt 330 als Basis und erweitert es grundlegend…
- ~~@Named~~

@Observes Agenda

1. Dependency Injection – Woher kommen wir?
   J2EE, Spring, JEE5, JSRs,…
2. **CDI Basics**
   Annotationen, Factories, Qualifiers, Contexts, …
3. CDI Advanced
   @Observes, @Interceptor, @Decorator,…
4. CDI für Profis
   Portable Extension SPI

CDI ist Annotations-basiert

- Annotationen sind typsicher
- Annotationen sind ähnlich wie Code und damit Tool-freundlich

CDI erlaubt einen XML-Fallback (beans.xml)

- Damit erlaubt CDI, die Annotationen zu überschreiben/ergänzen
- Erlaubt es Klassen/Features „einzuschalten“
- Es gibt bereits Plug-in Support (Seam 3 XML) für erweiterten XML-Support
- Manche Entwickler bevorzugen XML

CDI Basics – Let's @Inject

JEE6 - Beliebige Klassen

```
interface IPaymentProcessor {
            //[...]
}


class DefaultPaymentProcessor    implements
IPaymentProcessor {
            //[...]
}




class ShoppingCart {
      @Inject
      IPaymentProcessor
            paymentProcessor;
}
```

Produkt

Ziel

JEE5 – nur JNDI Ressourcen

```
@Stateless
@Local
class DefaultPaymentProcessor
            implements IPaymentProcessor {
            //[...]
}




@Stateless
@Remote
class ShoppingCart {
      @EJB
      IPaymentProcessor
            paymentProcessor;
}
```

JEE6 – CDI Factory

Produkt

Ziel

```
interface IPaymentProcessor {
}

class DefaultPaymentProcessorFactory {
      @Produces
      static final Logger LOGGER;
      @Produces
      IPaymentProcessor create(VAT pVat) {
            //[...]
            return aPaymentProcessor;
      }
      void remove(@Disposes IPaymentProcessor pPP) {…}
}


class ShoppingCart {
      @Inject
      static Logger theLogger; //field

      @Inject
      ShoppingCart(IPaymentProcessor paymentProcessor) {
            //CTOR with Injection
      }
}
```

Ziel

JEE6 – CDI Qualifier

Qualifier Was?

```
@Qualifier
@Target({TYPE, METHOD, PARAMETER, FIELD})
@interface Asynchronous {}
```

Produkt

```
class DefaultPaymentProcessorFactory {
     @Produces @Synchronous @Reliable
     IPaymentProcessor create(VAT pVat) {
          //[...]
          return aPaymentProcessor;
     }
}
```

```
@Asynchronous
class EnhancedPaymentProcessor
     implements IPaymentProcessor {
     //[...]
}
```

Ziel

```
class ShoppingCart {
     @Inject @Asynchronous
     ShoppingCart(IPaymentProcessor paymentProcessor) {
          //CTOR with Injection
     }
}
```

JEE6 – CDI Qualifier.2

Qualifier

```
@Qualifier
@Target({TYPE, METHOD, PARAMETER, FIELD})
@interface Concurrency {
     enum ConcurrencyType { SYNC, ASYNC };
     ConcurrencyType value();
}
```

Der Einsatz eines parametrisierten Qualifiers kann helfen eine Annotations-Explosion zu verhindern

Annotations-Parameter, die nicht Qualifizieren? → @NonBinding!

Produkt

```
class DefaultPaymentProcessorFactory {
     @Produces @Concurrency(SYNC) @Reliable
     IPaymentProcessor create(VAT pVat) {
          //[...]
          return aPaymentProcessor;
     }
}
```

```
@Concurrency(ASYNC)
class EnhancedPaymentProcessor
     implements IPaymentProcessor {
     //[...]
}
```

Ziel

```
class ShoppingCart {
     @Inject @Concurrency(SYNC)
     ShoppingCart(IPaymentProcessor paymentProcessor) {
          //CTOR with Injection
     }
}
```

CDI Basics – Lebenszyklus Callbacks

Entstehen und vergehen

Enstehen

Vergehen

```
class HttpAcceptor {
      Socket listening;
      Thread acceptThread;

      @PostConstruct
      void init() {
            listening = new …
            acceptThread = new AcceptThread(…);
      }

      @PreDestroy
      void destroy() {
            acceptThread.interrupt();
            listening.close();
            listening = null;
      }
}
```

@PostConstruct

Wird vom Container aufgerufen **nachdem** die Bohne erstellt wurde und noch **bevor** eine Business-Methode aufgerufen wurde.

@PreDestroy

Wird vom Container aufgerufen **bevor** die Bohne entfernt wird.

? Warum nicht CTOR und finalize verwenden

CTOR und finalize sind “Lebenszyklus-Methoden” bzgl. der JVM und nicht im Kontext des CDI-Containers. Das ist ein kleiner aber feiner Unterschied…

CDI Basics – Scopes

?

Scopes bestimmen…

- …den Lebenszyklus von Beans (zeitlich)
- …die Sichtbarkeit von Beans (contextual)

Typen

- @RequestScoped
- @SessionScoped
- @ApplicationScoped
- @ConversationScoped

```
Klassisch:
Von Web-Anwendungen bekannt
…aber erweiterbar
```

Web

- Jeder Servlet-Request hat Zugriff auf einen aktiven Request-, Session-, Application-Scope
- Jede JSF-Anwendung hat Zugriff auf einen Conversation-Scope

JEE6

Request- und Applikation-Scope sind aktiv

- Beim Aufruf von EJBs
- Bei der Zustellung von Nachrichten zu MDBs
- Beim Aufruf von Web-Services
- …

CDI Basics – JSF Integration - Beispiel

JSF-Fragment

```
<f:view>
    <h:form>
        <h:panelGrid columns="2" rendered="#{!login.loggedIn}">
            <h:outputLabel for="username">Username:</h:outputLabel>
            <h:inputText id="username" value="#{credentials.username}"/>
            <h:outputLabel for="password">Password:</h:outputLabel>
            <h:inputText id="password" value="#{credentials.password}"/>
        </h:panelGrid>
        <h:commandButton value="Login" action="#{login.login}" rendered="#{!login.loggedIn}"/>
        <h:commandButton value="Logout" action="#{login.logout}" rendered="#{login.loggedIn}"/>
    </h:form>
</f:view>
```

Integration von Beans über ihren Unified Expression Language Name

The built-in stereotype @javax.enterprise.inject.Model is intended for use with beans that define the *model* layer of an MVC web applic

CDI Basics – JSF Integration - Beispiel

Es ist ein Model und es sieht gut aus…

```
@Model //StandardScope = @RequestScope
class Credentials {
    String username;
    String password;

    String getUsername() { return username; }
    void setUsername(String username) { this.username = username; }

    String getPassword() { return password; }
    void setPassword(String password) { this.password = pass
}
```

@Model ist ein CDI Stereotype

…dazu später mehr

CDI ist nicht nur JSF-centric:

The built-in stereotype @javax.enterprise.inject.Model is intended for use with beans that define the model layer of an MVC web application architecture such as JSF

CDI Basics – JSF Integration - Beispiel

Ein SessionScopedBean

```
@SessionScoped @Model //optional @Stateful
class Login implements Serializable {

    @Inject Credentials credentials;
    @Inject @Users EntityManager userDatabase;
    […]
    User user;

    @Inject void initQuery(@Users EntityManagerFactory emf) {
        CriteriaBuilder cb = emf.getCriteriaBuilder();
        usernameParam = cb.parameter(String.class);
        […]
    }

    void login() {
        List<User> results = userDatabase.createQuery(query)
            .setParameter(usernameParam,
             credentials.getUsername())
        […]
        if ( !results.isEmpty() ) { user = results.get(0); }
    }
```

→

```
    void logout() { user = null; }

    boolean isLoggedIn() {
        return user!=null;
    }

    @Produces @LoggedIn User getCurrentUser() {
        if (user==null) {
            throw new NotLoggedInException();
        }
        else {
            return user;
        }
    }
}
```

CDI Basics – JSF Integration - Beispiel

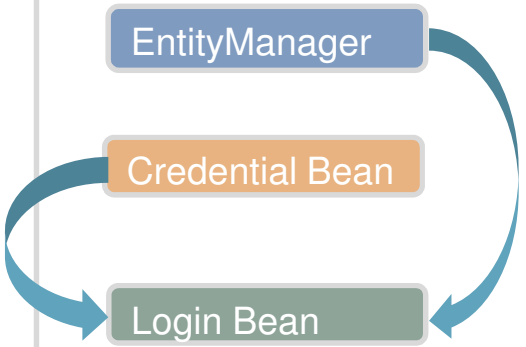
? Was passiert wenn jemand den Login Button drückt?

1. CDI erzeugt ein Credential Bean im Scope Request und füllt es mit den Parametern des JSF Requests.

2. Gibt es bereits ein Login Bean im Scope Session?
   1. JA! Dann weiter bei 3.
   2. NEIN! Dann erzeugt CDI ein Login Bean und injiziert alle Ressourcen u.a. das Credential Bean

3. JSF ruft die Methode „login“ des Login Beans auf

4. CDI kann andere Beans mit dem logged in User versorgen (@Produces)



CDI Basics – JSF Integration - Beispiel

? Eins fehlt noch. Wie funktioniert eigentlich…

```
@SessionScoped @Model
class Login implements Serializable {

    […]
    @Inject @Users EntityManager userDatabase;
```

CDI-Qualifizierte JPA-Ressource!

! Über einen Qualify-Produce-Proxy

```
class Databases {
    @Produces @PersistenceContext(unitName="UserData")
    @Users EntityManager userDatabaseEntityManager;

    @Produces @PersistenceUnit(unitName="UserData")
    @Users EntityManagerFactory userDatabaseEntityManagerFactory;
}
```

Vorteil:

- Typsicherheit im Vergleich zu Zeichenketten
- Scoping möglich

JEE6 – CDI Scopes

Ziel

```
@ApplicationScoped
class LoginProcessor {
     @Inject
     LoginData data;

     void login() {
          //[...]
          return aPaymentProcessor;
     }
}
```

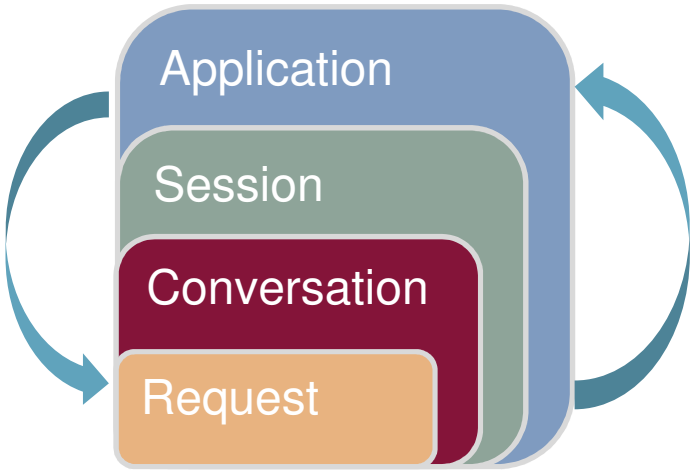
Produkt

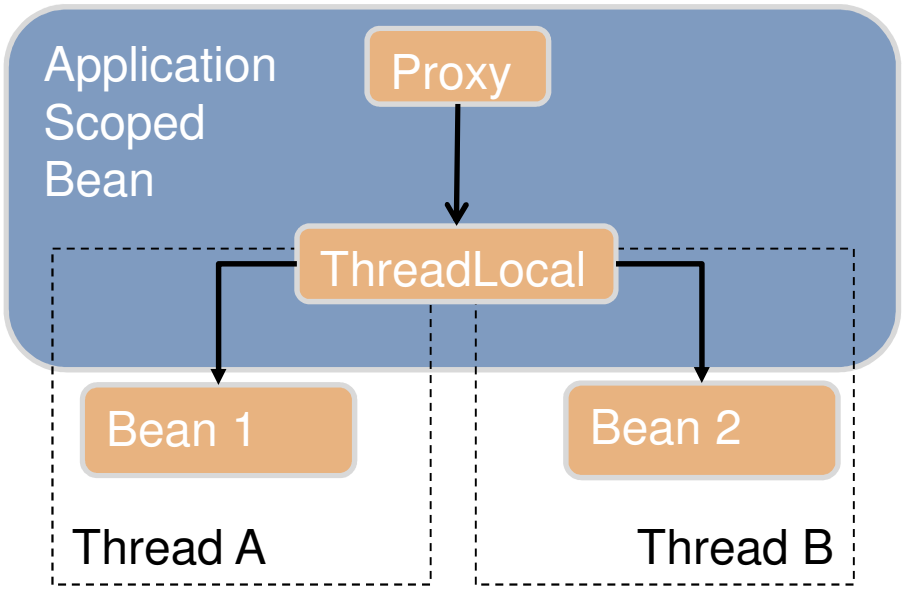
```
@RequestScoped
class LoginData {
     @Inject @HttpParam(„uid“)
     String userID;
     @Inject @HttpParam(„pwd“)
     String userPWD;
}
```

Vormerken:
Wie das mit HttpParam funktioniert kommt später!

CDI Basics – Scopes

! **Injection** von feineren in gröbere Scopes ist auch möglich!





Ein Request ist an einen Thread gebunden (Tx sind auch an den Thread gebunden)!
Damit sind RequestScoped-Beans Thread-spezifische Beans

CDI Basics – Pseudo-Scopes

Normale Scopes

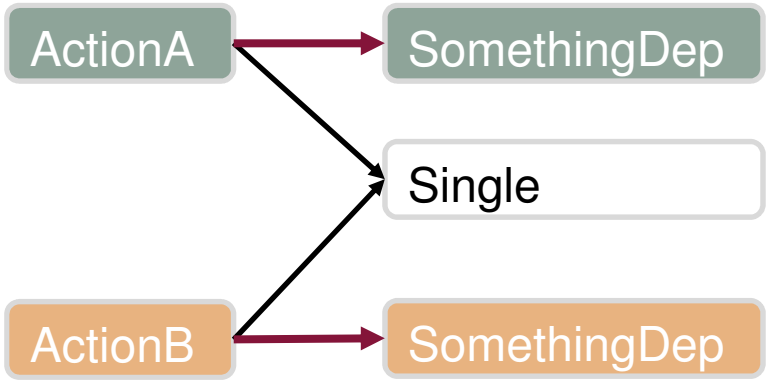
```
@SessionScoped
class ActionA {
     @Inject
     Single singleton;
     @Inject
     SomethingDep dep;
}

@RequestScoped
class ActionB {
     @Inject
     Single singleton;
     @Inject
     SomethingDep dep;
}
```

Pseudo-Scopes

```
@Dependend
class SomethingDep {…}

@Singleton
class Single {…}
```



Dependend Beans

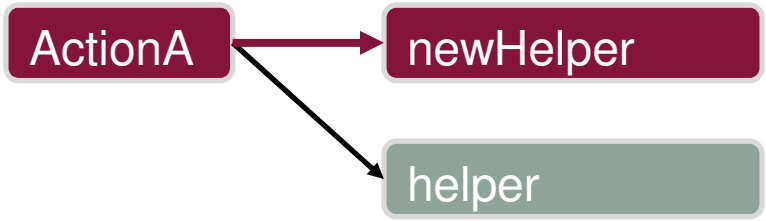
- **erben** den Scope des Owners ihres Injection-Points
- sind an den Lebenszyklus des Owners gebunden.
- Werden nie mehrmals verwendet, sondern pro Injection neu erzeugt!

CDI Basics – New

@New Injection

```
@ConversationScoped
class ActionA {
     @Inject
     Helper helper;
     @Inject @New
     Helper newHelper;
}

@SessionScoped
class Helper {
    …
}
```



New Injection-Points

- **„überschreiben“** den Scope des eigentlichen Beans
- **Vererben** den Scope des Owners auf die neue Instanz
- Binden den Lebenszyklus des Beans an den des Owners

@Typed – Explizite Bean-Types

Produkt

```
@ConversationScoped
@Typed({ActionA.class, IfcA})
class ActionA extends ActionX implements IfcA, IfcB {
    @Inject @New
    Helper newHelper;
}
```

Typed…

…ist anwendbar auf Klassen und Producer-Methoden bzw -Felder

Alternative (z.B. für Test-Stage)

Ziel

~~Ziel~~

```
@SessionScoped
class UseCaseImpl {
    @Inject
    ActionA actionA;

    @Inject
    IfcB ifcB;
}
```

@Observes Agenda

1. Dependency Injection – Woher kommen wir?
   J2EE, Spring, JEE5, JSRs,…
2. CDI Basics
   Annotationen, Factories, Qualifiers, Contexts, …
3. **CDI Advanced**
   @Observes, @Interceptor, @Decorator,...
4. CDI für Profis
   Portable Extension SPI

CDI Advanced – Alternativen und XML

Default-Klasse

```
interface IPaymentProcessor {
            //[...]
}

class DefaultPaymentProcessor
            implements IPaymentProcessor {
            //[...]
}
```

Stage Produktion

```
<beans>
</beans>
```

„Leeres“ beans.xml
→ Markiert eine Deployment Unit als CDI Beans-Archive

Alternative (z.B. für Test-Stage)

```
@Alternative
class MockPaymentProcessor
            implements IPaymentProcessor {
            //[...]
}
```

Stage Test – Alternatives beans.xml

```
<beans>
    <alternatives>
        <class>MockPaymentProcessor</class>
    </alternatives>
</beans>
```

Aktivierte Alternative
Qualifier, Stereotypes, Interceptors – alle können Alternativen haben

CDI Advanced – Interceptoren: Auslagern von Aspekten

Interceptor

```
@Interceptor
@LoggingBindingType
class LoggingInterceptor {
    static Logger LOG = Logger.getLogger("Interceptor");

    @AroundInvoke
    Object doLog(InvocationContext ctx) throws Exception {
        LOG.entering(…);
        Object result = ctx.proceed();
        return result;
    }
}
```

Binding Type (Qualifier)

```
@InterceptorBinding
@Target({METHOD, TYPE})
@Retention(RUNTIME)
@interface LoggingBindingType {

}
```

BindingType

Bestimmt, wo der Interceptor angewendet wird

Anwendung

```
@LoggingBindingType
class HelloImpl implements Hello {
    void sayHello() {
        System.out.println("Hello World!");
    }
    @ExcludeClassInterceptors
    void unintercepted() {…}
}
```

Binding am Einsatzort

Auf Klassen und Methoden-Ebene möglich

Aktiviertes beans.xml

```
<beans xmlns=„...“>
    …
    <interceptors>
        <class>LoggingInterceptor</class>
        <class>OtherInterceptor</class>
    </interceptors>
</beans>
```

Reihenfolge

Die Reihenfolge in beans.xml entspricht der Aufrufreihenfolge

CDI Advanced – Interceptoren: JEE5

Interceptor

```
@Interceptor
class LoggingInterceptor {
    static Logger LOG = Logger.getLogger("Interceptor");

    @AroundInvoke
    Object doLog(InvocationContext ctx) throws Exception {
        LOG.entering(…);
        Object result = ctx.proceed();
        return result;
    }
}
```

Anwendung

```
@Stateless
class HelloImpl implements Hello {
    @Resource EjbContext
    @Interceptors(LoggingInterceptor.class)
    void sayHello() {
        System.out.println("Hello World!");
    }
}
```

Binding am Einsatzort

Auf Klassen und Methoden-Ebene möglich

Aktivieren + Binden: ejb-jar.xml

```
<ejb-jar xmlns = "http://java.sun.com/xml/ns/javaee"
       version = "3.0"
       xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation = "http://java.sun.com/xml/ns/javaee
            http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">
  <interceptors>
    <interceptor>
      <interceptor-class>LoggingInterceptor</interceptor-class>
    </interceptor>
  </interceptors>
  <assembly-descriptor>
    <interceptor-binding>
      <ejb-name>HelloImpl</ejb-name>
      <interceptor-order>
        <interceptor-class>LoggingInterceptor</interceptor-class>
      </interceptor-order>
    </interceptor-binding>
  </assembly-descriptor>
</ejb-jar>
```

CDI Advanced – Decorator: Erweiterung von Business-Methoden

Decorator

```
@Decorator
class SchufaPaymentProcessor
  implements IPaymentProcessor {
    @Delegate @Inject @Any IPaymentProcessor delegate;
    @Inject ISchufaService schufa;

    void process(…) {
        schufa.check();
        delegate.process(…);
    }
}
```

Delegate-Injection

Field oder im CTOR möglich

Aktiviertes beans.xml

```
<beans xmlns=„...“>
    …
    <decorators>
        <class>SchufaPaymentProcessor </class>
        <class>OtherPaymentProcessor</class>
    </decorators>
</beans>
```

Reihenfolge

Die Reihenfolge in beans.xml entspricht der Aufrufreihenfolge

! Regelwerk beachten

8.3.1. Assignability of raw and parameterized types for delegate injection points

Decorator delegate injection points have a special set of rules for determining assignability of raw and parameterized types, as an exception to Section 5.2.3, “Assignability of raw and parameterized types”.
A raw bean type is considered assignable to a parameterized delegate type if the raw types are identical and all type parameters of the delegate type are either unbounded type variables or java.lang.Object.

A parameterized bean type is considered assignable to a parameterized delegate type if they have identical raw type and for each parameter:

- the delegate type parameter and the bean type parameter are actual types with identical raw type, and, if the type is parameterized, the bean type parameter is assignable to the delegate type parameter according to these rules, or
- […]

CDI Advanced – Stereotype

Konzept

Ein ~ bündelt eine Kombination aus

- Default Scope
- Interceptor-Bindings
- Sub-Stereotypes (transitiv)

Zusätzlich kann ein ~ bestimmen, dass

- Ein Default EL-Name angewendet wird
- Die Beans Alternativen sind

Beispiel Stereotype

```
@RequestScoped
@Secure
@Transactional
@Stereotype
@Target(TYPE)
@Retention(RUNTIME)
@Named
@Alternative
@interface MockAction {}
```

Anwendung

```
@MockAction
class HelloImpl implements Hello {
    void sayHello() {
        System.out.println("Hello World!");
    }
}
```

Alternative-Stereotype

Muss erst noch aktiviert werden…

Alternatives beans.xml

```
<beans>
    <alternatives>
        <stereotype>MockAction</stereotype>
    </alternatives>
</beans>
```

Events abfeuern

```
class CustomEventPayload {…}

class DefaultPaymentProcessor implements IPaymentProcessor {
    @Inject @Any Event<CustomEventPayload> event;

    void process(…) {
        event.fire( new CustomEventPayload(…) );
        …
    }
}
```

Events beobachten

```
class SomeListenerClass {
    void listen(@Observes CustomEventPayload pEvt) {…}
}
```

CDI Advanced – Ereignisreiches CDI und Qualifiers.1

Events qualifiziert abfeuern

```
class CRUDEvent {…}

class EntityDAO {
    @Inject @Create Event<CRUDEvent> createEvent;
    @Inject @Retrieve Event<CRUDEvent> retrieveEvent;
    @Inject @Update Event<CRUDEvent> updateEvent;


    void update(…) {
        updateEvent.fire( new CRUDEvent(…) );
        …
    }
}
```

Events beobachten

```
class SomeListenerClass {
    void listen(@Observes @Any CRUDEvent pEvt) {…}
    void hören(@Observes @Create CRUDEvent pEvt) {…}
    void observen(@Observes @Update CRUDEvent pEvt) {…}
}
```

? Welche Observer werden aufgerufen von `updateEvent.fire`

Events programmatisch qualifiziert abfeuern

```
class CRUDEvent {…}

class EntityDAO {
    @Inject @Any Event<CRUDEvent> anyEvent;

    void update(…) {
      anyEvent.select( new AnnotationLiteral<Update>(){} ).fire(new CRUDEvent(…));
    }
}
```

Events beobachten

```
class SomeListenerClass {
    void observen(@Observes @Update CRUDEvent pEvt, @Logger Log pLog) {…}
}
```

! Wichtig auch hier gibt die Spezifikation Regeln vor, wie Fire und Observe zueinander finden!

CDI Advanced – Ereignisreiches CDI und das Abenteuer Nebenläufigkeit

Events feuern

```
class EntityDAO {
    @Inject @Any Event<CRUDEvent> anyEvent;

    void update(…) {
      anyEvent.select( new AnnotationLiteral<Update>(){} ).fire(new CRUDEvent(…));
    }
}
```

Events asynchron beobachten – nur CDI mit JEE

```
@Stateless
class SomeListenerEJB {
    @Asynchronous
    void observe(@Observes @Update CRUDEvent pEvt) {…}
}
```

@Asynchronous
Ist keine CDI Annotation!

! Bei der Ausführung sind nicht alle Scopes vorhanden

Events feuern – nur CDI mit JEE

```
@DAO
@Stateless
class EntityDAO {
    @Inject @Any Event<CRUDEvent> anyEvent;
    @Inject UserTransaction utx;

    void update(…) {
        utx.begin();
        anyEvent.fire(new CRUDEvent(…));
        …
        utx.commit();
    }
}
```

Events bei bestimmten Transaktionsphasen beobachten

```
class SomeListener {
    void observen(@Observes(during=AFTER_SUCCESS) @Update
      CRUDEvent pEvt) {…}
}
```

TransactionPhases

- IN_PROGRESS,
- BEFORE_COMPLETION
- AFTER_COMPLETION
- AFTER_FAILURE
- AFTER_SUCCESS

! Asynchron und During = Nichts was sich verträgt

@Observes Agenda

1. Dependency Injection – Woher kommen wir?
   J2EE, Spring, JEE5, JSRs,…
2. CDI Basics
   Annotationen, Factories, Qualifiers, Contexts, …
3. CDI Advanced
   @Observes, @Interceptor, @Decorator,…
4. **CDI für Profis**
   Portable Extension SPI

CDI Advanced – Instance

? Wie kann ich eine Instanz programmatisch auswählen

Deklaration des Injection-Points

```
class CheckoutAction {
    @Inject Instance<IPaymentProcessor> processors; //optional hier Qualifier angeben
    …
}
```

Verwendung - Einfach

```
IPaymentProcessor p = processors.get(); //Default Instanz
```

Verwendung - Qualifiziert

```
IPaymentProcessor p = processors.select( new AnnotationLiteral<Synchronous>(){} )
```

Verwendung - Iterierend

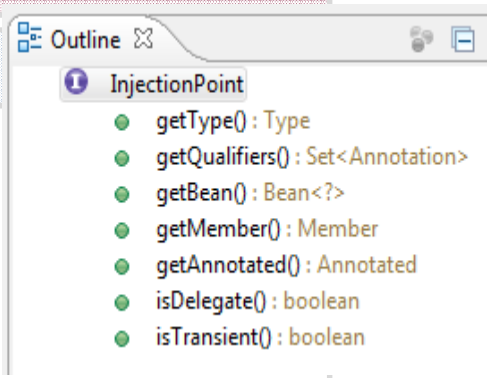
```
for(IPaymentProcessor p : processors) {
   p.process(…); //Iteriere über alle Instanzen
}
```

? Wie kann ich die Produkte programmatisch beeinflussen

Erinnern wir uns an HttpParam

```
class HttpParamProducer {
    @Inject
    FacesContext facesContext;

    @Produces
    @HttpParam
    String getHttpParameter(InjectionPoint ip) {
        String name = ip.getAnnotated().getAnnotation(HttpParam.class).value();
        if ("".equals(name)) name = ip.getMember().getName();
        return facesContext.getExternalContext()
                .getRequestParameterMap()
                .get(name);
    }
}
```



Verwendung

```
@Inject @HttpParam("userId")
String currentUserId;
```

Qualifier mit/ohne Wert - Geht das?

```
@Qualifier
@interface Param {
    @Nonbinding String value() default "";
}
```

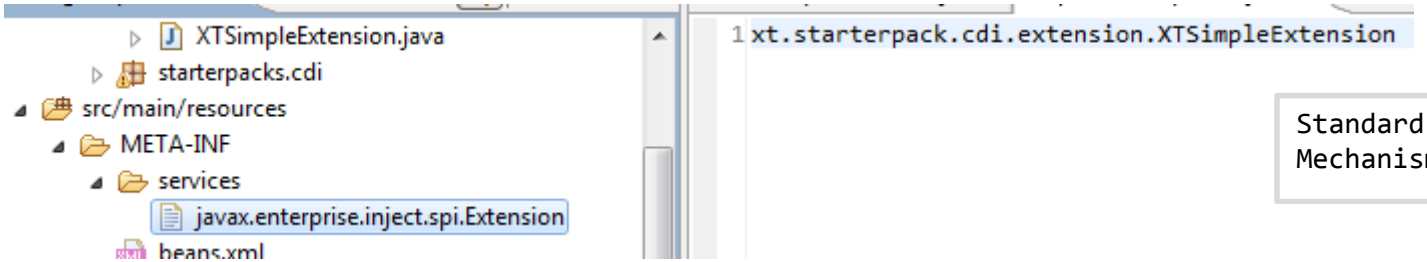
? Was kann ich sonst noch programmatisch machen

Ziemlich viel – das würde aber den Rahmen sprengen

- Es gibt Container-Events - Du kannst dich als @Observes dranhängen.
  - Before/AfterBeanDiscovery
  - Process[AnnotatedType|InjectionTarget|Producer|Bean|ObserverMethod]
- Es gibt einen BeanManager – Dem kannst Du z.B. eigene Beans unterschieben (in JEE ist er über java:comp/BeanManager zu finden, oder man sagt einfach @inject BeanManager bm;)
- Eigene Scope-Typen definieren
- Es gibt schon einige fertige Erweiterungen

CDI für Profis – Eine „dumme“ Extension

Anmelden einer Extension im Beans-Archive



Standard Java Service-Mechanismus

Implementierung

```
class XTSimpleExtension implements Extension {
      void beforeBeanDiscovery(@Observes BeforeBeanDiscovery event) {
            System.out.println("--- "+event);
            AnnotatedType<String> a = new AnnotatedType<String>() {
               …
            };
            event.addAnnotatedType(a); //hier wird von außen was eingefügt!
      }
}
```

BeforeBeanDiscovery
…und viele weitere Container-Events

Interessant!
Der CDI-Container benutzt seine eigenen Mechanismen…

Vielen Dank für Ihre Aufmerksamkeit

Vorname Name

Bereich / Funktion

Telefon: +49 89 96101-xxxx
vorname.name@msg-systems.com

www.msg-systems.com

.consulting .solutions .partnership

Farben

- Primärfarben für Text, Aufzählungspunkte und Hervorhebungen und einfache grafische Darstellungen (Farbwerte RGB)

schwarz 0/0/0	rot 132/20/57	petrol 96/163/188

- Zusatzfarben für komplexe Grafiken, Schaubilder und Diagramme

75% schwarz 63/63/63	grün 30/74/53	gelb 208/139/1	blau 1/57/130
20% schwarz 203/203/203	50% grün 142/164/153	50% gelb 232/179/128	50% blau 127/155/192

- Farbmusterfelder zum Kopieren liegen außerhalb der Folie