ORDIX AG

Kontext und Dependency Management in Java EE - Einführung in JSR-299

DOAG Konferenz
16. – 18.11.2010, Nürnberg

Björn Konrad
info@ordix.de
www.ordix.de

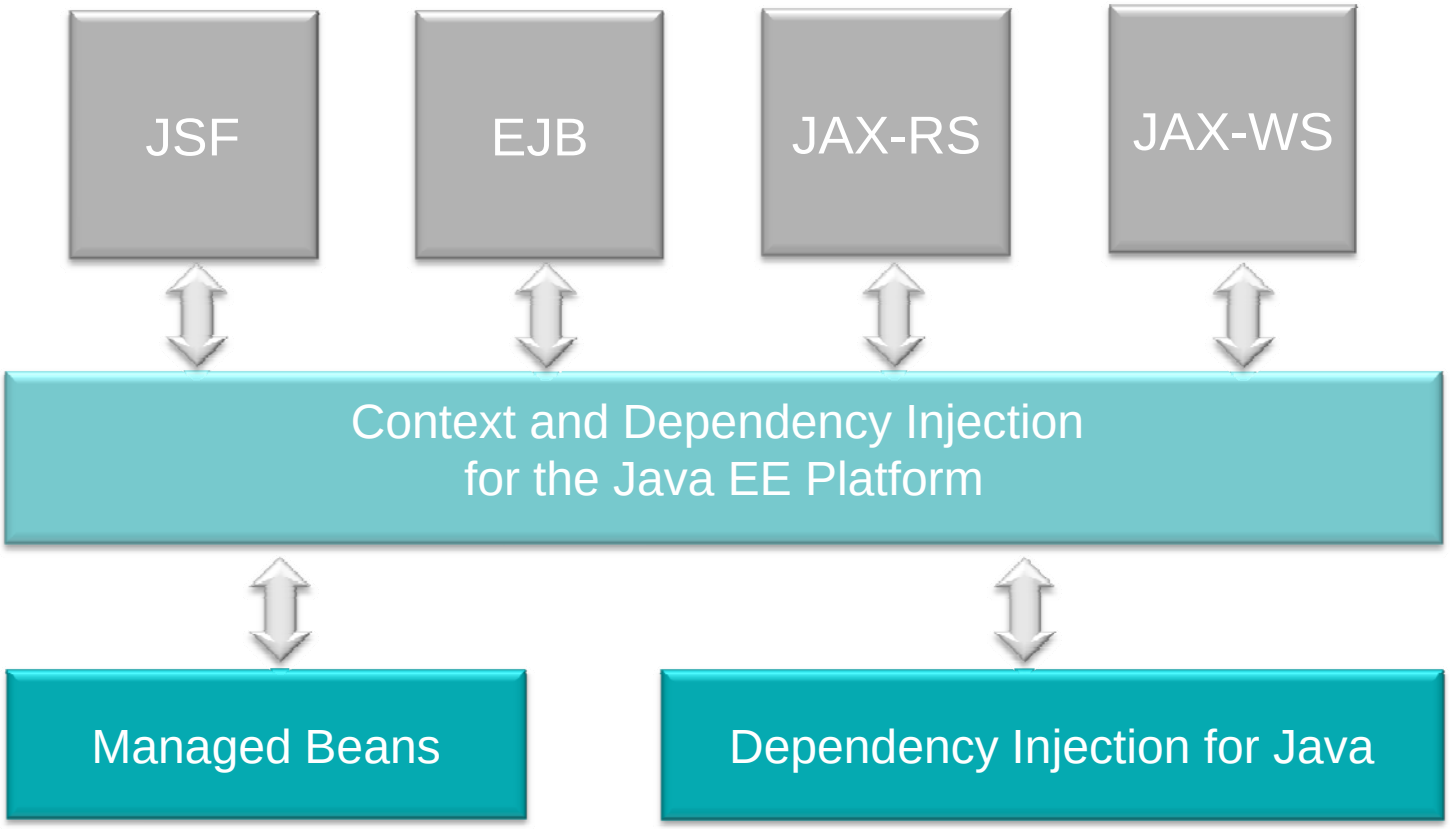
Agenda

ORDIX AG

- Einführung Managed Beans
- Dependency Injection mit CDI
- Interceptoren, Dekoratoren und Events
- Scopes und Kontext
- Portable Extensions

Einführung

ORDIX AG



Einführendes Beispiel

ORDIX AG

JSF und EJB mit CDI

```
@Stateful @SessionScoped
public class BidManagerBean implements BidManager {

          @Inject
          private EntityManager entityManager;

          public Long addBid() {
                    entityManager.persist(bid);
          }
}
```

```
<f:view>
  <h:form>
    <h:panelGrid columns="2">
      <h:outputLabel for="amount">Betrag:</h:outputLabel>
      <h:inputText id="amount" value="#{bid.amount}"/>
    </h:panelGrid>
    <h:commandButton value="Login" action="#{bidManagerBean.addBid}" />
  </h:form>
</f:view>
```

Der Begriff Bean

ORDIX AG

- Vor Java EE 6: Keine einheitliche Definition
  - JSF Managed Beans, Enterprise Java Beans, Web Beans, …
  - Dependency Injection beschränkt auf bestimmte Java EE Resourcen
  - Namensbasierte Dependency Injection fehleranfällig
- Mit Java EE 6
  - Standardisierung des Begriffs *Managed Bean*
  - Managed Bean = ein vom Container verwaltetes Objekt
  - Standard ermöglicht typsicheres Resource Injection und Lifecycle Services

CDI Grundlagen

ORDIX AG

- Lebenszyklus und Sichtbarkeit einer Bean wird vom Container bestimmt
  - Client muss den Lebenszyklus nicht verwalten
  - Container sorgt dafür, wann eine Bean Instanz erzeugt/zerstört wird
  - Objekte mit Zuständen werden automatisch den richtigen Clients zur Verfügung gestellt
- Der Container verwaltet Abhängigkeiten zwischen Beans
  - Lose Kopplung zwischen Komponenten des Applikationscodes
  - Dependency Injection, Interceptoren und Events
- Der Container unterstützt
  - Managed Beans
  - EJBs
  - Producer Methoden
  - Java EE Resourcen

Aktivierung von CDI

ORDIX AG

- beans.xml muss bereitgestellt werden
- muss für ein WAR Archiv im Root Classpath liegen
- muss für ein JAR Archiv in META-INF liegen
- darf leer sein

```
<beans
   xmlns="http://java.sun.com/xml/ns/javaee"
   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/beans_1_0.xsd">

</beans>
```

Definition einer Managed Bean

ORDIX AG

- einfache Java Klasse
- Plain Old Java Object (POJO)
- Standard Konstruktor
- muss eine konkrete Klasse sein
- Unterstützung der Lifecycle Callbacks *@PostConstruct* und *@PreDestroy*

Anatomie einer CDI Bean

ORDIX AG

- Menge an Bean Types
- Menge an Bean Qualifier
- Scope
- EL Name
- Interceptor Bindings
- Bean Implementierungsklasse

```
@Typed(ShoppingCart.class)
@RequestScoped
@Named("shoppingCart")
public class ShoppingCartBean implements ShoppingCart {

          @Inject
          @HibernateOrderDao
          private OrderDao orderDao;

          @Transactional
          public void order() {

          }
}
```

CDI mit EJB

ORDIX AG

- EJBs haben ihren eigenen Lifeycle, Zustandsmanagement und Nebenläufigkeitsmodell
- EJBs werden zu CDI Beans, haben jedoch weiterhin ihr eigenen Lifecycle und Zustandsmanagement
- CDI Services lassen sich auf Stateful und Stateless Session Beans voll anwenden
- Managed Beans können in EJBs injiziert werden und umgekehrt

Agenda

ORDIX AG

- Einführung Managed Beans
- Dependency Injection mit CDI
- Interceptoren, Dekoratoren und Events
- Scopes und Kontext
- Portable Extensions

Dependency Injection - Grundlagen

ORDIX AG

- Objekte werden von außen d. h. unabhängig vom Code „zusammengesteckt“
- ermöglicht mehr Flexibilität bei der Wartbarkeit: Komponenten können einfacher ausgetauscht werden
- gilt insbesondere bei der Verwendung von Interfaces
- alternative Implementierungen sind auch zur Deployment-Zeit injizierbar
- Vereinfacht die Testbarkeit des Applikationscodes. Mock Implementierungen sind leichter injizierbar
- Verschlankung: Code wird von Erzeugung des Objektnetzes befreit (`new()` Deklarationen)

Dependency Injection mit CDI

ORDIX AG

- Injection Points - Wo kann injiziert werden?
  - Felder
  - Konstruktor
  - Initialisierer Methoden
- Injection Sources - Was kann injiziert werden?
  - Managed Beans
  - Session Beans
  - Java EE Resources
  - Objekte, die durch Producer-Methoden erzeugt werden
  - weiterer Resourcen, die per SPI angebunden werden

Dependency Injection – Qualifier (1)

ORDIX AG

- Annotation Deklaration

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

}
```

- Qualifier Deklaration

```
@HibernateOrderDao
public class HibernateOrderDao implements OrderDao {

          @Inject @OrderDatabase
          private EntityManager entityManager;
}
```

- Field Injection

```
@RequestScoped
public class ShoppingCartBean implements ShoppingCart {

          @Inject @HibernateOrderDao
          private OrderDao orderDao;
}
```

Dependency Injection – Qualifier (2)

ORDIX AG

- Konstruktor Injection

```
@RequestScoped
public class ShoppingCartBean implements ShoppingCart {

          private OrderDao orderDao;

          @Inject
          public ShoppingCartBean(@HibernateOrderDao OrderDao orderDao) {
             this.orderDao = orderDao;
          }
}
```

- Initialisierer Method Injection

```
@RequestScoped
public class ShoppingCartBean implements ShoppingCart {

          private OrderDao orderDao;

          @Inject @HibernateOrderDao
          public setOrderDao(OrderDao orderDao) {
             this.orderDao = orderDao;
          }
}
```

Dependency Injection – Qualifier (3)

ORDIX AG

- Build-in Qualifier
  - @Default – Standard Qualifier
  - @Any – Nützlich für polymorphe Injizierung
- Qualifier mit Member Variablen
  - @HibernateDao("myHibernateDao")
  - vermeidet die Explosion von Annotationsdeklarationen
- Mehrere Qualifier
  - @HibernateDao @Synchronous
  - injiziert wird nur die Bean, die *beide* Qualifier deklariert

Dependency Injection – Alternativen

ORDIX AG

- Ermöglicht die Injizierung von Beans für ein bestimmtes Deployment-Szenario

```
@Alternative
public class MockOrderDao implements OrderDao {

          public void order() {
               // Some Mock Implementation
          }
}
```

- Alternatives müssen explizit aktiviert werden

```
<beans
   xmlns="http://java.sun.com/xml/ns/javaee"
   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/beans_1_0.xsd">
   <alternatives>
         <class>org.mycompany.mock.MockOrderDao</class>
   </alternatives>
</beans>
```

Agenda

ORDIX AG

- Einführung Managed Beans
- Dependency Injection mit CDI
- Interceptoren, Dekoratoren und Events
- Scopes und Kontext
- Portable Extensions

Interceptoren

ORDIX AG

- bekannt aus der EJB Spezifikation
- entkoppeln technische Belange von Geschäftslogik
- CDI erweitert Interceptoren durch einen annotationsbasierten Mechanismus

Beispiel: Interceptoren (1)

ORDIX AG

- Interceptor Binding

```
@InterceptorBinding
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Transactional{

}
```

```
@Named("shoppingCart")
public class ShoppingCartBean implements ShoppingCart {

          @Transactional
          public void order() {

          }
}
```

Beispiel: Interceptoren (2)

ORDIX AG

- Interceptor Implementierung

```
@Interceptor @Transactional
public class TransactionalInterceptor {

          @AroundInvoke
          public void manageTransaction(InvocationContext ctx) {

          }
}
```

- Interceptor Implementierungen können Dependency Injection nutzen
- müssen in beans.xml aktiviert werden

Dekoratoren

ORDIX AG

- entkoppeln Geschäftslogik voneinander
- führen Geschäftslogik aus, die orthogonal zu allen Interface Implementierungen sind
- implementieren ein Business Interface.
  Die Implementierung fängt Aufrufe der regulären Interface-Implementierung ab
- Beispiel: Jede Implementierung des *ShoppingCart* Interfaces muss Warenkörbe bestimmter Größe in die Marketing Datenbank schreiben

Beispiel: Dekoratoren (1)

ORDIX AG

- Business Interface Deklaration

```
public interface ShoppingCart {

          public void order(Order order);

          public List<CartEntry> getShoppingCartEntries();
}
```

- Dekorator Implementierung

```
@Decorator
public abstract MarketingDatabaseShoppingCart implements ShoppingCart {

          @PersistenceContext EntityManager em;

          public void order(Order order) {
                    ...
          }

}
```

Beispiel: Dekoratoren (2)

ORDIX AG

- Injection Point

```
@Decorator
public abstract MarketingDatabaseShoppingCart implements ShoppingCart {

          @Inject @Delegate @Any ShoppingCart shoppingCart;

          public void order(Order order) {
                    if (shoppingCart.getShoppingCartEntries() > 100) {
                              em.persist(order);
                    }
          }

}
```

- Dekorator kann jede Methode des Delegate Objekts aufrufen
- Dekoratoren müssen in beans.xml aktiviert werden

Events

ORDIX AG

- erweitern das Observer/Observable Pattern
- lose Kopplung zwischen Observer und Event Producer:
  Event Producer muss Observer nicht registrieren
- Observer können die Menge an Events bestimmen, über die sie informiert werden wollen
- das CDI Event Modell funktioniert analog zu den annotationsbasierten Dependency Injection Mechanismen

Beispiel: CDI Events

ORDIX AG

- Event Observer

```
public class OrderService {

          public void onAnyOrderEvent(@Observes @Order order) {...}

          public void onCanceledOrderEvent(@Observes @Canceled Order order)
{...}
}
```

- Event Producer

```
public class ShoppingCartBean implements ShoppingCart {

          @Inject @Canceled Event<Order> canceledOrderEvent;

          public void order(Order order) {
                    ...
                    canceledOrderEvent.fire(order);
                    ...
          }
}
```

Producer-Methoden

ORDIX AG

- produzieren Objekte, die zur Laufzeit injiziert werden
- ermöglichen Laufzeit Polymorphismus mit CDI
- Objekte, die keine CDI Beans sind (beispielsweise Java EE Resourcen), werden auf diese Weise injected
- Objekte werden entsprechend der Scope-Deklaration zurückgegeben

Beispiel: Producer-Methoden (1)

ORDIX AG

- Producer Deklaration

```
public class Preferences implements Serializable {

          private PaymentStrategy paymentStrategy;

          @Produces @Preferred @SessionScoped
          public PaymentStrategy getPaymentStrategy() {
                    switch (paymentStrategy) {
                       case CREDIT_CARD: return new CreditCardPaymentStrategy();
                       case PAYPAL : return new PayPalPaymentStrategy();
                    }
          }
}
```

- Producer Injection

```
public class ShoppingCartBean implements ShoppingCart {

          @Inject PaymentStrategy paymentStrategy;

          public void order(Order order) {...}
}
```

Beispiel: Producer-Methoden (2)

ORDIX AG

- Producer Deklaration mit Dependency Injection
  - PaymentStrategy Objekte werden durch Applikation erzeugt (*new*)
  - Objekte können so keine Dependency Injection und Intereptoren nutzen
  - Scope Deklarationen müssen stimmig sein –
    werden nicht vom Container entdeckt

```
public class Preferences implements Serializable {

          private PaymentStrategy paymentStrategy;

          @Produces @Preferred @SessionScoped
          public PaymentStrategy getPaymentStrategy(CreditCardPaymentStrategy ccps,
                                                    PayPalPaymentStrategy ppps) {
                    switch (paymentStrategy) {
                       case CREDIT_CARD: return ccps;
                       case PAYPAL : return ppps;
                    }
          }
}
```

Dependency Injection von Java EE Resourcen

ORDIX AG

- Dependency Injection in Java EE 5 ist begrenzt
  - Injection ist nur für bestimmte Resourcen möglich
    (JDBC Datasources, JMS Queue oder Topic, JPA Persistence Context, Remote EJB oder Web Service)
  - Injection ist nicht konsistent
    (Persistence Unit Name, JNDI Name, EJB Link)
  - Injection ist nicht typsicher durch String basierte Namen
    → Fehler werden erst zur Laufzeit entdeckt
- Durch CDI Producer-Methoden können Java EE Resourcen typsicher und konsistent injiziert werden

Beispiele: Java EE Resource Injection

ORDIX AG

```
@Produces @WebServiceRef(lookup="java:app/service/Catalog")
Catalog catalog;

@Produces @Resource(lookup="java:global/env/jdbc/CustomerDatasource")
@CustomerDatabase Datasource customerDatabase;

@Produces @PersistenceContext(unitName="CustomerDatabase")
@CustomerDatabase EntityManager customerDatabasePersistenceContext;

@Produces @PersistenceUnit(unitName="CustomerDatabase")
@CustomerDatabase EntityManagerFactory customerDatabasePersistenceUnit;

@Produces @EJB(ejbLink="../their.jar#PaymentService")
PaymentService paymentService;
```

Agenda

ORDIX AG

- Einführung Managed Beans
- Dependency Injection mit CDI
- Interceptoren, Dekoratoren und Events
- Scopes und Kontext
- Portable Extensions

Scopes und Kontext

ORDIX AG

- CDI Beans können mit ScopeTypes vesehen werden (beispielsweise @SessionScoped)
- jeder ScopeType hat ein Context-Objekt, welches Lifecycle und Client-Sichtbarkeit der Bean regelt
- jede CDI Bean lebt in einem Context-Objekt als *Contextual Instance*
- eigene ScopeTypes können definiert werden indem das *Context* Interface implementiert wird
- mit jedem Thread ist stets ein Kontext Objekt pro ScopeType assoziiert
- Context-Objekte stehen bei jedem Servlet/JSF, Web Service und EJB Aufruf zur Verfügung

Built-in Scopes

ORDIX AG

- Request-Kontext
  - @RequestScoped Annotation
  - ist aktiv während der `service()`-Methode eines Servlets,
    Java EE Web Service-, EJB Remote- oder JMS *MessageListener* Aufrufen
  - Kontext wird zerstört, wenn `service()`-Methode beendet wurde
    oder die JMS Nachricht ausgeliefert wurde
- Session-Kontext
  - @SessionScoped Annotation
  - ist während der `service()`-Methode eines Servlets aktiv
  - Kontext wird zerstört, wenn HTTP Session endet
  - Bean muss serialisierbar sein (Container Passivation)

Built-in Scopes

ORDIX AG

- Application-Kontext
  - @ApplicationScoped Annotation
  - ist aktiv während der `service()`-Methode eines Servlets,
    Java EE Web Service-, EJB Remote- oder JMS *MessageListener* Aufrufen
  - Kontext wird zerstört, wenn die Anwendung beendet wird

- Conversation-Kontext
  - @ConversationScoped Annotation
  - ist aktiv während allen Standard Lifecycle Phasen eines JSF Requests
  - Conversation entspricht bestimmten User-Workflow
    (z. B. *Bestellung ausführen*)
  - hält Zustand für die Dauer eines User-Workflows
    (d. h. über mehrere Requests hinweg)
  - CDI stellt die Built-in Bean *Conversation* bereit, um den Lebensyklus einer Konversation programmatisch kontrollieren zu können

Beispiel: Conversational-Kontext

ORDIX AG

```
@ConversationScoped @Stateful
public class OrderService {

   private Order order;

   private @Inject Conversation conversation;

   private @PersistenceContext(type = EXTENDED) EntityManager em;

   @Produces public Order getOrder() {
      return order;
   }

   public Order createOrder() {
      order = new Order();
      conversation.begin();
      return order;
   }

   public void addLineItem(OrderItem item, int quantity) {...}

   public void saveOrder(Order order) {
      em.persist(order);
      conversation.end();
   }
}
```

Pseudo-Scopes

ORDIX AG

- Singleton
  - annotiert mit @Singleton (JSR-330)
  - nur eine Instanz
  - Clients haben direkte Referenz auf Singleton Bean (kein Proxy)
- Dependent
  - annotiert mit @Dependent
  - steht niemals für mehrere Clients oder Injection Points zur Verfügung
  - Lebenszyklus richtet sich nach Bean, die eine Dependent Bean deklariert

Lebenszyklus EJBs in CDI

ORDIX AG

- Container erzeugt EJB und gibt Referenz auf internes Proxy-Objekt zurück, welches Methodenaufrufe zur eigentlichen EJB weiterleitet
- eine EJB wird so zu einem von CDI kontrollierten Objekt (Contextual Instance)
- EJB Container kontrolliert dennoch den Lebenszyklus (entsprechend der *@Stateful* oder *@Stateless* Annotationen)
- Stateful Session Beans
  - mit *@Remove* annotierte Methode dürfen nur von einer *Dependent* ScopeType EJB aufgerufen werden
  - darf jede ScopeType haben
- Stateless Session Beans
  - dürfen nur mit Dependent ScopeType deklariert werden

Agenda

ORDIX AG

- Einführung Managed Beans
- Dependency Injection mit CDI
- Interceptoren, Dekoratoren und Events
- Scopes und Kontext
- Portable Extensions

Portable Extensions

ORDIX AG

- Ziel: Integration weiterer Frameworks wie Spring, Seam, GWT oder Wicket
- die CDI API stellt eine Menge von SPIs bereits, so dass programmatisch
  - eigene Beans im Container registriert werden können
  - CDI Beans in eigene Beans injizierbar sind
  - eigene ScopeTypes implementiert werden können
  - auf Container Lifecycle Events reagiert werden kann
    (z. B. vor/nach Bean Discovery)

Portable Extensions

ORDIX AG

- *BeanManager* Interface ist die zentrale Anlaufstelle.
  - hierüber kann auf Beans, Kontexte, Interceptoren und Dekoratoren programmtisch zugegriffen werden
  - ein Bean Manager-Objekt kann einfach als Built-in Bean injiziert werden

```
public class ShoppingCartBean implements ShoppingCart {

          @Inject BeanManager beanManager;
}
```

Fazit

ORDIX AG

- Typsicheres Dependency Injection für Java EE
- gute Integration von JSF und EJB
- Verbesserung von Wartbarkeit und Testbarkeit durch lose Kopplung (Dependency Injection, Intereptoren, Dekoratoren, Events)
- Container kontrolliert Lebenszyklus von CDI Beans
- Weld als Referenzimplementierung (http://www.seamframework.org/Weld)

Vielen Dank
für Ihre Aufmerksamkeit!