

Vorlesung Software aus Komponenten

3. Komponentenmodelle

Prof. Dr. Hans-Gert Gräbe
Wintersemester 2015/16

3.5. Enterprise Java Beans

EJB 2 - Nachteile

Probleme des Standards

- Viele Schnittstellen zu implementieren (EJBHome, EJBObject, callback zum Kontext)
- Nur eine Geschäftsmethoden-Schnittstelle pro EJB Bean
- Weitere Konventionen sind zu beachten (RMI, spezielle Basis-Schnittstellen)
- Zusätzliche und andere Konventionen für EJB-Klassen im Vergleich zu normalen (plain old) Java-Klassen
- Konfiguration von Beans und Applikationen für Beans erfolgt durch riesige XML-basierte Deployment-Deskriptoren
- EJB-Laufzeit zu rudimentär spezifiziert
- Komplexität der Interaktion mit der Persistenzschicht
- Keine Schnittstellen für Logging, Tracing und andere Basisdienste, um Komponenten zu testen und in der Laufzeit zu verfolgen.
- Beans müssen vom Client manuell gesucht und angesprochen werden.

Ziele von EJB 3

- Genauere Spezifikation des Bean-Lebenszyklus durch mehr Erweiterungspunkte im Kontext
- Dependency Injection: Einzelne Attributwerte, Methoden- oder Klassendefinitionen werden zur Laufzeit aus dem Kontext importiert
- Verwendung von Meta-Daten
- Interzeptoren und aspektorientierte Ansätze für Infrastrukturdienste
- Vereinfachung der Persistenzbehandlung
- Reduktion der Zahl der Artefakte, die ein Bean-Entwickler bereitstellen muss
- Vereinfachung der EJB Typen durch Reduktion der Zahl der Schnittstellen
- Verbesserung der Testmöglichkeiten außerhalb des Containers

Wie wird das umgesetzt?

- Einsatz von POJO's (plain old java objects) und POJI's (plain old java interfaces)
 - Dienstschnittstelle als Java-Schnittstelle
 - kein Home-Interface mehr
 - Annotation durch Metadaten für die Konfiguration der EJB-Typen, Local/Remote, Transaktionen, Sicherheit
- Dependency Injection
 - für Attribute, Eigenschaften, Ressourcennutzung
 - werden aus der Umgebung durch Annotationen in die Laufzeit der Beaninstanz „injiziert“
- Erweiterter Lebenszyklus-Unterstützung
 - nutzerdefinierte Callback-Methoden
- Interzeptoren
 - Unterbrechung der Geschäftsmethode in AOP-Manier

Programmiermodell: Die Bean-Klasse

- Bean-Klasse als zentrales Artefakt, das bereitgestellt wird
- keine Home-Schnittstelle mehr, Laufzeitunterstützung kommt direkt aus dem Kontext
- Beantyp wird durch Annotation festgelegt
- Beanklasse kann Ausnahmen über Annotation vom Typ **@ApplicationException** werfen
- Beanklasse wirft keine **java.rmi.RemoteException** mehr

```
public interface Buchung {  
    public void buchen(Kunde k, Seminartyp s) ;  
}
```

```
@Stateful public class BuchungBean implements Buchung {  
    public void buchen(Kunde k, Seminartyp s) {  
        //Code zur Ausführung der Buchung  
    }  
}
```

Programmiermodell: Ereignisse im Lebenszyklus

- Ereignisse im Lebenszyklus werden durch den Container über annotierte Callbacks an die Bean-Klasse weitergegeben
- Callbacks können **RuntimeException** auslösen, die Rollback von Transaktionen nach sich ziehen, aber keine **ApplicationException**.
- Letzteres kann durch annotierte **CallbackListener** implementiert werden

```
@Stateful public class BuchungBean implements Buchung {  
    @PreDestroy dispose() { ... }  
}
```

Programmiermodell: Interzeptoren

- Interzeptoren unterbrechen die Abarbeitung einer Geschäftsmethode
- Interzeptormethoden können in der Beanklasse oder in eigenen Interzeptorklassen definiert sein
 - können **RuntimeException** oder **ApplicationException** werfen, wie in der Schnittstelle der Geschäftsmethode vereinbart
 - Abarbeitung im selben Transaktions- und Sicherheitskontext wie die Geschäftsmethode
 - Können JNDI, JDBC, JMS, andere Beans und den **EntityManager** (Schnittstelle zum Persistenzkontext) nutzen
 - verwendet Dependency Injection
- Für dieselbe Geschäftsmethode können mehrere Interzeptoren definiert sein
- Signatur einer Interzeptormethode:
public Object methodName(invocationContext) throws Exception

Programmiermodell: Interzeptoren (Fortsetzung)

- **InvocationContext** definiert als

```
public interface InvocationContext {  
    public Object getBean();  
    public Method getMethod();  
    public Object[] getParameters();  
    public void setParameters(Object[]);  
    public EJBContext get EJBContext();  
    public java.util.Map getContextData();  
    public Object proceed() throws Exception;  
}
```
- Kontext-Daten werden von allen Interzeptoren derselben Routine gemeinsam genutzt.
- Aufruf-Ende von **proceed()** innerhalb des Interzeptors - Rückkehr in die unterbrochene Routine

Wichtige Annotationen und deren Defaultwerte

- **@Local** und **@Remote**: Annotation von Klassen und Schnittstellen
 - Default: @Local
 - ```
@Remote public interface Buchung {
 public void buchen(Kunde k, Seminartyp s) ;
}
```
- **@TransactionManagement**: Annotation der Beanklasse
  - Default `TransactionManagementType.CONTAINER`
- **@TransactionAttribute**: Annotation der Klasse oder einzelner Methoden
  - Beschreibt das Transaktionsverhalten
  - Default: `REQUIRED`
- **@Timeout**: Definition einer Timeout-Methode der Bean
- **@ApplicationException**
- Weitere Annotationen für Sicherheitsaspekte:
  - **@RolesReferences**, **@RolesAllowed**, **@RunAs**, **@SecurityRoles**

### Zustandslose Beans

- Implementieren plain old Java Interfaces
- keine Home-Schnittstelle, Laufzeitkontrolle durch Kontext
- Annotation **@Stateless**
- Definierte Callbacks **@PostConstruct** und **@PreDestroy**
- Interzeptor **@AroundInvoke** um eine einzelne Session
- Unspezifizierte Transaktions- und Sicherheitskontexte
- Dependency Injection muss vor dem ersten Aufruf einer Geschäftsmethode erfolgen

### Bean und Interzeptor. Beispiel

```
@Stateless @Interceptors ({MyInterceptor.class})
public class BuchungBean implements Buchung {
 public void buchen(Kunde k, Seminartyp s) { ... }
}

public class MyInterceptor {
 @AroundInvoke
 public Object zeitVerbrauch(InvocationContext ic) throws Exception {
 long time = System.currentTimeMillis();
 try { return ic.proceed(); }
 finally {
 long total = System.currentTimeMillis() - time;
 System.out.println("Aufruf von " + ic.getMethod().getName()
 + " dauerte "+total+" Millisekunden.");
 }
 }
}
```

## 3.5. Enterprise Java Beans

### EJB 3

#### Zustandsbehaftete Beans

- Implementieren plain old Java Interfaces
- keine Home-Schnittstelle, Laufzeitkontrolle durch Kontext
- Annotation **@Stateful**
- kann Schnittstelle **SessionSynchronization** implementieren
- Definierte Callbacks **@PostConstruct**, **@PreDestroy**, **@PostActivate**, **@PrePassivate**
- Interzeptoren
  - **@afterBegin** – Ausführung zu Beginn jeder Session
  - **@beforeCompletion** – Ausführung vor dem Ende jeder Session
  - **@AroundInvoke** – Einbettung einer Session
- Dependency Injection muss vor dem ersten Aufruf einer Geschäftsmethode erfolgen
- Client bekommt Referenz auf Session direkt über JNDI oder über Dependency Injection
- Annotation **@Remove** für Methode zum Auflösen der Beaninstanz

### Zustandbehaftete Bean. Beispiel

```
@Stateful public class BuchungBean implements Buchung {
 Connection rc;
 @Resource SessionContext sc;

 @PostConstruct @PostActivate public void init() {
 rc = Connection.Open();
 }

 @PreDestroy @PrePassivate public void close() {
 try { rc.close(); }
 catch (CloseException) { /* no op */ }
 }
 // ... weiter
```

```
@Remove public void dispose() { /* was auch immer */ }
```

```
@AroundInvoke public Object monitor(InvocationContext ic) {
 try {
 Object res = ic.proceed();
 if ((OpResult) res == OpResult.SUCCEED) {
 System.out.println("Aufruf war okay");
 }
 return res;
 } catch (Exception ex) { throw ex; }
}
```

```
// Geschäftsmethode
```

```
public void buchen(Kunde k, Seminartyp s) { ... }
}
```

### Zustandbehaftete Bean. Beispiel

Zugriff durch den Client:

```
static public void testBean(Kunde k, Seminartyp s) throws Exception {
 javax.naming.Context ctx = new javax.naming.InitialContext();
```

```
 Buchung test = ctx.lookup(Buchung.class.getName());
```

```
 // oder Injection, wenn innerhalb einer anderen Bean:
```

```
 /* @EJB Buchung test; */
```

```
 test.buchen(k,s); // Test der Geschäftsmethode
```

```
 test.dispose(); // Test der Bean dispose-Methode
```

```
}
```

## 3.5. Enterprise Java Beans

### EJB 3

### Nachrichtengesteuerte Beans

- Implementieren Nachrichten-Listener-Schnittstelle des jeweiligen Nachrichtentyps (bspw. **javax.jms.MessageListener**)
- Annotation **@MessageDriven**
- implementiert nicht notwendig **javax.ejb.MessageDrivenBean**
- kann Schnittstelle **SessionSynchronization** implementieren
- Definierte Lebenszyklus-Event Callbacks **@PostConstruct**, **@PreDestroy**
- Interzeptoren für Listener-Aufrufe
- Dependency Injection muss vor dem ersten Aufruf einer Geschäftsmethode erfolgen

```
@MessageDriven(activateConfig = { ... })
```

```
public class ExampleMDB implements javax.jms.MessageListener {
 public void onMessage(Message msg) { ... }
}
```



## 3.5. Enterprise Java Beans

### EJB 3

### Bean-Kontext und Umgebung

- Bean spezifiziert ihre Abhängigkeiten durch **Abhängigkeits-annotationen** für Typ, Name und Charakteristika eines benötigten Objekts oder Ressource

```
@Resource (name = "DB", type = "javax.sql.DataSource.class")
```

- Bean-Entwickler annotiert erforderliche Instanzvariablen, die vom Container automatisch nach Setzen von **EJBContext** und vor dem Aufruf der ersten Geschäftsmethode aus dem Kontext instanziiert werden

```
@Stateless public class MySessionBean implements MySession {
 @Resource(name="DB") public DataSource myDB;
 public void getData() {
 // ohne try-catch! Probleme sind nur auf Kontextseite möglich
 Connection c = myDB.getConnection();
 }
}
```

## Persistenz

- Entity-Klassen sind durch **@Entity** annotiert und müssen einen **public-Konstruktor ohne Argumente** haben.
  - können abstract oder konkret sowie in Vererbungshierarchien eingebunden sein
  - müssen (in der Regel) Schnittstelle **Serializable** implementieren
- Instanzen einer Entity-Klasse sind leichtgewichtige persistente Objekte: Persistenz auf der Ebene einzelner Attribute
  - Kennzeichnung durch Annotation
  - Persistenz-Anbieter greift auf entsprechende Attribute zu; diese müssen nicht nach außen für andere sichtbar sein
  - Zugriff direkt auf das Attribut oder über Getter-Methode (Default)
- Default: Abbildung auf Datenbank-Tabelle mit demselben Namen
  - Primärschlüssel müssen aus einer Klasse sein, die **equals** und **hashCode** korrekt unterstützt.

### Persistenz. Beispiel

```
@Entity @Table(name = "KundenListe")
public class KundenListe implements java.io.Serializable {
 private int id; // Primärschlüssel für die Kundenliste
 private Collection<Kunde> kundenListe;

 @Id(generate = GenerationType.AUTO)
 public int getId() { return id; }
 // spezifiziert, dass dies die Getter-Methode für den Primärschlüssel ist
 public void setId(int id) { this.id = id; }

 @OneToMany public Collection<Kunde> getKunden() { return kundenListe; }
 // Auslesen der Kundenliste

 ...
}
```

### Persistenz (Fortsetzung)

- Persistente Aggregate (Ganzes aus mehreren Teilen) werden über die Annotation **@Embedded** innerhalb einer **@Entity** realisiert
- Eine Entity-Bean kann auf mehrere Datenbank-Tabellen abgebildet werden: **@SecondaryTable**
- Relationen zwischen Entitäten über Annotationen **@OneToOne**, **@OneToMany**, **@ManyToOne**, **@ManyToMany**
  - können unidirektional oder bidirektional sein
- Persistenz kann in Vererbungshierarchien mit vererbt werden: **@Inheritance**

## EntityManager

- **EntityManager** ist für die Verwaltung des Lebenszyklus von Entity-Beans verantwortlich
- Assoziiert mit dem Persistenzkontext, also einer Menge von Entity-Beans

```
@Stateless public class AuftragsListe {
 @PersistenceContext EntityManager em;
 public void neuerAuftrag(int kundenId, Auftrag neuerAuftrag) {
 Kunde k = (Kunde) em.find("Kunde", kundenId);
 k.getAuftraege().add(neuerAuftrag);
 neuerAuftrag.setKunde(k);
 }
}
```

- EntityManager können vom J2EE-Server zur Verfügung gestellt werden, aber auch von der Applikation → **EntityManagerFactory**