

Vorlesung Software aus Komponenten

3. Komponentenmodelle

Prof. Dr. Hans-Gert Gräbe
Wintersemester 2013/14

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 von 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**

Weitere Konzepte

- Transaktionskontrolle: entweder via JTA oder über EntityTransaction
- Persistenz-Kontexte: Innerhalb eines solchen Kontexts wird die Eindeutigkeit der Zuordnung Entity – Persistenzobjekt garantiert
- Query API mit eigener, an SQL angelehnter EJB Query Language
- Entity Packaging: Im Deployment-Deskriptor können genauere Informationen über EntityManager gespeichert werden.

Zusammenfassung

- Aufgaben des EJB-Container werden an den Kontext delegiert
- Spezielle Eigenschaften des Ausführungskontexts werden durch Annotationen direkt im Quelltext festgelegt statt als Deployment-Informationen
- Persistenz folgt dem Konzept der Java Data Objects JDO
- Standard fixiert in JSR 220

Server-Provider (Server-Anbieter)

- stellt Plattform zur Verfügung
- Netzwerkanbindung, Skalierungsfunktion
- Prozess- und Ressourcenmanagement

EJB-Kontext-Provider (Container-Anbieter)

- setzt auf Plattform des Server-Providers auf
- Benutzerverwaltung, Transaktionsmanagement, Persistenz
- Herstellung der Installationswerkzeuge
- Implementierung der EJB-Kontextfunktionalität
- Container- und Server-Provider oft identisch
 - bessere Performance und Wartbarkeit

3.6. Java und Komponenten

Komponenten und Rollen

Bean-Provider (Komponenten-Entwickler)

- realisiert geforderte Anwendungslogik (Geschäftslogik)
- keine grundlegenden Funktionalitäten (Persistenz, Netzwerk, etc.)
- benötigt Fachwissen über Anwendungsbereich
- Konzentration auf inhaltliche Problemstellung

Application-Assembler (Monteur)

- verbindet Komponenten zu einer Anwendung
- Clients
- Verwendung von Basiskomponenten
- Nutzung wiederverwendbarer Komponenten von Drittanbietern
- oftmals Bean-Provider und Application-Assembler in einem Haus

Bean-Deployer (Installation)

- installiert Komponenten der Anwendung auf Zielsystem
- verwendet Werkzeuge des Container-Providers
- nutzt Deployment-Deskriptor
- konfiguriert EJBs
- generiert Stummel- und Skelettklassen
- legt Benutzerdatenbank an
- benötigt detailliertes Wissen über Server und Container

Systemadministrator

- verantwortlich für reibungslosen Ablauf
- Benutzerverwaltung