ENTERPRISE JAVABEANS

2621 Web Applikationen II

ENTERPRISE JAVABEANS

Chapter 01 - Introduction

DEFINITION

- Die Enterprise JavaBeans Architektur wird für die Entwicklung von komponentenbasierter Unternehmensanwendungen eingesetzt
- Enterprise JavaBeans Applikationen sind Sklaierbar, Transaktions- und Multi-User-Sicher
- Diese Anwendungen können einmal geschrieben werden und dann auf einer beliebigen EE-Server-Plattform eingesetzt werden

EINFACHER GESAGT…

Enterprise JavaBeans ist ein Standard für ein
Server-Seitiges-Komponenten-Modell
für verteilte Geschäftsanwendungen!

EINLEITUNG

- Anwendungsentwicklung kann ein komplexes Unterfangen sein…
- Applikationen müssen Ihre „Hauptaufgabe“ gut erledigen….
- Was macht eine gute Applikation aus?
  - Korrektheit (Funktioniert wie spezifiziert)
  - Sicherheit
  - Systemintegrität
  - Skalierbarkeit
  - Interoperabilität (Zusammenarbeit von/mit verschiedenen Systemen)
  - Robust / Widerstandsfähig

- Dies sind Voraussetzungen für eine „gute“ Applikation, keine einzige ist allerdings spezifisch für eine bestimmte Applikation oder Geschäftsfall

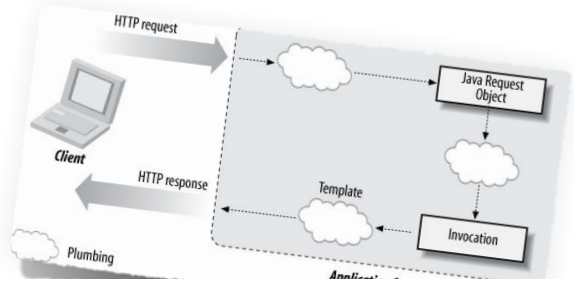
VERANTWORTLICHKEITEN AUFBRECHEN

Aus Gründen der Einfachheit können wir den gesamten Code in einem System, in eine von drei Kategorien einordnen:

- Kernaufgaben
- Querschnittaufgaben
- Systemaufgaben

SYSTEMAUFGABEN

- Module wurden gebaut, um die Kernaufgaben zu lösen! Aber wie gelangen Daten von Punkt A nach B?
- Dies und mehr sind Systemaufgaben:
  - Steuerung einer HTTP-Anfrage an einen Aktion Handler
  - Unterhaltung einer JDBC Verbindung oder eine JavaMail Session
  - Übertragen von Übergabeparametern aus einem JSON Request in ein Java Objekt
  - …



CODE SMART, NOT HARD

mach weniger…. führe eine Abstraktion ein, welche die notwendigen Funktionen als entkoppelte Schicht bietet

The Container

(Ein Container ist ein Host, der diese Dienstleistungen für Gastanwendungen bietet)

REVIEW

- Es gibt Anforderungen welche in einer Vielzahl von Anwendungen eine Rolle spielen
- Es ist wichtig Kernaufgaben (Business-Logik) von Querschnittaufgaben zu Trennen
- „roll-your-own-approach“ generiert unnötigen Code

SCHLUSSFOLGERUNG

- Die EJB Specification ist eine Lösung für….
  - Adressierung allgemeiner Probleme bei der Programm Entwicklung
  - code less
  - Programm/Architektur Standardisierung
  - Interaktion mit anderen Technologien unter dem Dach der Java Enterprise Edition

ENTERPRISE JAVABEANS

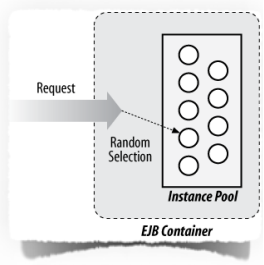
Chapter 01.02 - Component Types

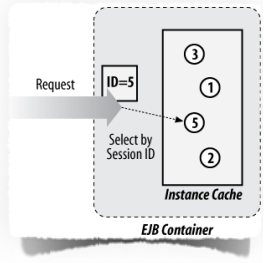
SERVERSEITIGE KOMPONENTEN

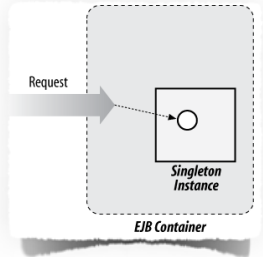
- Typen vom „Servierseitigen Komponenten“ befinden sich ausschließlich auf dem Server
- Der Kunde muss mit ihnen über vorgegebene Kanäle interagieren
- Es gibt zwei wichtige serverseitige Komponenten Typen
  - session beans (bietet die Sicht für den Client)
  - message-driven beans (wirkt als „event listeners“)

SESSION BEANS

- Stateless session beans (SLSBs)
  - sind nicht auf einen bestimmten Client gebunden (service)
  - der Zustand wird nicht unterhalten (gespeichert)
- Stateful session beans (SFSBs)
  - ist an einen client gebunden (für die ganze „bean life time“)
  - unterhält den Zustand für die Lebenszeit
- Singleton beans
  - von allen Clients gemeinsam genutzt
  - speichert Programmweite-Daten

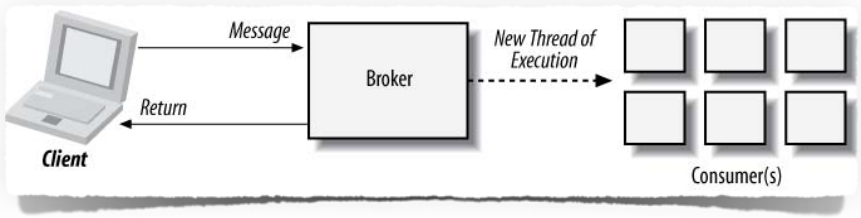






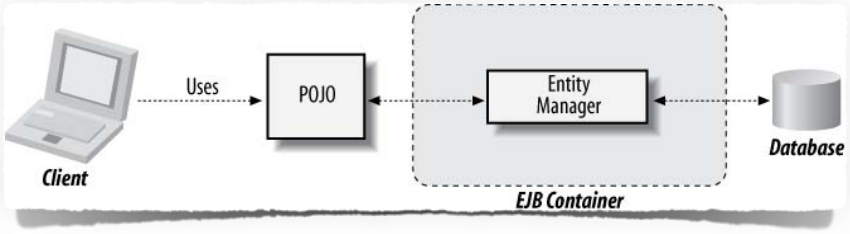
MESSAGE-DRIVEN BEANS

- Message-Driven beans (MDBs)
  - Zustandslos
  - Serverseitig
  - Transactions bewusst
  - Asynchron



ENTITY BEANS

- Entities beans (Entities)
  - sind Java Objekte welche mit O/R Mapper Metadaten annotiert werden
  - haben Eigenschaften welche den persistierten Zustand repräsentieren
  - müssen einen Primären Schlüssel (primary key) haben



ENTERPRISE JAVABEANS

Chapter 01.03 - Container Services

DIE SPEZIFIKATION BIETET

- Dependency injection
- Concurrency
- Instance pooling/caching
- Transactions
- Security
- Timers
- Naming and object stores
- Interoperability
- Lifecycle callbacks
- Interceptors
- Java Enterprise Platform integration

DEPENDENCY INJECTION

EJB ist eine komponentenbasierte Architektur. Dementsprechend bietet es die Möglichkeit Module, welche von einander abhängig sind, auf eine entkoppelte Art zu referenzierten

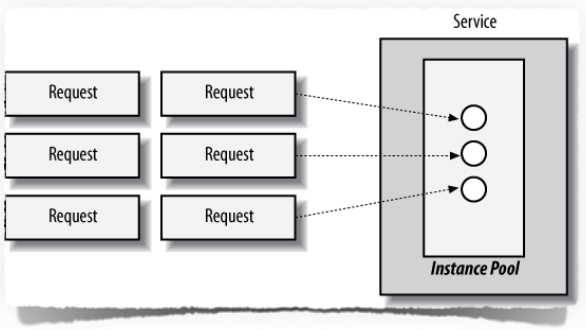
Sie müssen sich lediglich an einen „Vertrag“ halten. Der Container übernimmt die Umsetzung und Zeitpunkt der Bereitstellung selber (zur Laufzeit).

CONCURRENCY

Durch eine Reihe von Richtlinien kann der Anwendungsentwickler das Problem mehrheitlich dem Container übergeben.

INSTANCE POOLING/CACHING

Aufgrund der strengen concurrency Richtlinien welche vom Container durchgesetzt werden, kann eine nicht verfügbare Ressource einen möglichen Engpass bilden (bis die anderen Request's abgearbeitet sind).
EJB geht dieses Problem durch die Pooling Technik an, dabei werden von jedem Modul mehrere Instanzen erstellt um die eingehende Anfragen abzuarbeiten.

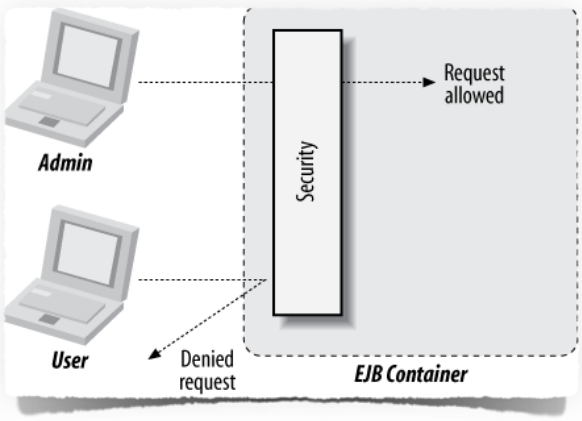


TRANSACTION

- **A**tomicity
  - Jeder Befehl wird vollständig abgearbeitet oder garnicht. Gibt es einen Fehler auf halbem Weg, wird der Zustand vor diesem Befehl wiederhergestellt
- **C**onsistency
  - Das System ist vor sowie nach dem Aufruf konsistent und hält alle Richtlinien ein
- **I**solation
  - Transaktionen werden isoliert und sind von aussen nicht einsehbar bis der Prozess erfolgreich abgeschlossen ist. Gemeinsame Ressourcen dürfen nicht von zwei Transaktionen auf einmal geändert werden.
- **D**urability
  - Sobald eine Transaktion erfolgreich abgeschlossen ist, müssen die Änderungen dauerhaft übernommen sein (persistiert).

SECURITY

- Multiuser Programme werden wahrscheinlich eine Vielzahl von unterschiedlichen Operationen bereitstellen, welche je nach Benutzer unterschiedlich sind
- EJB bietet für diesen Fall rollenbasierte Sicherheitsmechanismen an welche durch den Container geregelt werden



TIMERS

- geplante Aufgaben wie…
  - ein Ticket Reservationssystem darf nicht beanspruchte Tickets nach einer Zeitüberschreitung oder Inaktivität freigeben
  - Auktionen schliessen zu einer bestimmten Zeit
  - Die Rechnungsläufe laufen einmal im Monat mit anschliessendem E-Mail Versand
- Der EJB Timer Service kann eingesetzt werden um solche Ereignisse auszulösen

NAMING AND OBJECT STORES

- Enterprise JavaBeans verwendet das Java Naming and Directory Interface (JNDI) als lookup API für Java Clients
- JNDI unterstützt fast jede Art von Namens- und Verzeichnisdiensten
- Die Art und Weise JNDI ist in Java Enterprise Edition Anwendungen verwendet wird, ist in der Regel recht einfach
- Java-Client-Anwendungen können JNDI verwenden um eine Verbindung zu einem EJB-Server zu initiieren und einen bestimmtes EJB zu suchen

INTEROPERABILITY

- Interoperabilität ist ein wichtiger Bestandteil von EJB
- Die Spezifikation beinhaltet Unterstützung für
  - Java RMI-IIOP für remote method invocation, transaction, naming und security
  - JAX-WS
  - JAX-RPC
  - Web Services

LIFECYCLE CALLBACKS

- Einige Services benötigen eine Initialisierung oder Bereinigung für dessen einwandfreie Funktion
- EJB Komponenten haben einen Lebenszyklus welcher es erlaubt „callback“ Methoden aufzurufen um solche Aufgaben abzuarbeiten

INTERCEPTORS

- EJB nimmt sich vielen Querschnittaufgaben an, allerdings kann es natürlich auch Sonderfälle geben
- Aus diesem Grund bietet EJB sogenannte interceptors an
- So können eigene Querschnittaufgaben einfach und zentral implementiert werden und mittels „invocation chains“, ohne den Kernaufgaben Code zu „verunreinigen“, angewendet werden

PLATFORM INTEGRATION

- Als Schlüsseltechnologie der Java Enterprise Edition, fasst EJB viele der anderen Plattform-Frameworks und API zusammen
  - Java Transaction Service
  - Java Persistence API
  - Java Naming and Directory Interface (JNDI)
  - Security Services
  - Web Services

ENTERPRISE JAVABEANS

Chapter 01.04 - Your First EJB

CALCULATOR

1. INTERFACE

```
public interface Calculator {

    public double add(double x, double y);

    public double subtract(double x, double y);

    public double multiply(double x, double y);

    public double divide(double x, double y) throws DivisionByZeroException;

}
```


1. INTERFACE EJB

```
@Remote
public interface Calculator {

    public double add(double x, double y);

    public double subtract(double x, double y);

    public double multiply(double x, double y);

    public double divide(double x, double y) throws DivisionByZeroException;

}
```

2. BEAN

```
public class CalculatorBean implements Calculator {

    @Override
    public double add(double x, double y) {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public double subtract(double x, double y) {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public double multiply(double x, double y) {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public double divide(double x, double y) throws DivisionByZeroException {
        // TODO Auto-generated method stub
        return 0;
    }

}
```


2. SESSION BEAN EJB

```
@Stateless
public class CalculatorBean implements Calculator {

    @Override
    public double add(double x, double y) {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public double subtract(double x, double y) {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public double multiply(double x, double y) {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public double divide(double x, double y) throws DivisionByZeroException {
        // TODO Auto-generated method stub
        return 0;
    }

}
```

3.TEST (JUNIT)

```
public class CalculatorIT {

    private Calculator calculator;
    private double x, y;

    @BeforeClass
    public static void setup() throws NamingException {
    }

    @Before
    public void init() throws NamingException {
    calculator = new CalculatorBean();
        x = (double) (new Random().nextInt(200000) - 100000) / 100;
        y = (double) (new Random().nextInt(200000) - 100000) / 100;
    }

    @Test
    public void testAdd() {
        double result = calculator.add(x, y);
        assertEquals(x + y, result, 0.01);
    }
```


3.TEST (JUNIT) EJB

```
public class CalculatorIT {

    private Calculator calculator;
    private double x, y;

    @BeforeClass
    public static void setup() throws NamingException {
    }

    @Before
    public void init() throws NamingException {
    calculator = (Calculator) new InitialContext().lookup("java:global/calculator01/CalculatorBean");
        x = (double) (new Random().nextInt(200000) - 100000) / 100;
        y = (double) (new Random().nextInt(200000) - 100000) / 100;
    }

    @Test
    public void testAdd() {
        double result = calculator.add(x, y);
        assertEquals(x + y, result, 0.01);
    }
```

3.TEST (JUNIT)

```
public class CalculatorIT {

    private static final String JNDI_NAME = "java:global/calculator01/CalculatorBean";
    private static Context jndiContext;
    private Calculator calculator;
    private double x, y;

    @BeforeClass
    public static void setup() throws NamingException {
        jndiContext = new InitialContext();
    }

    @Before
    public void init() throws NamingException {
        calculator = (Calculator) jndiContext.lookup(JNDI_NAME);
        x = (double) (new Random().nextInt(200000) - 100000) / 100;
        y = (double) (new Random().nextInt(200000) - 100000) / 100;
    }

    @Test
    public void testAdd() {
        double result = calculator.add(x, y);
        assertEquals(x + y, result, 0.01);
    }
```


JNDI.PROPERTIES

```
java.naming.factory.initial=com.sun.enterprise.naming.SerialInitContextFactory
java.naming.factory.url.pkgs=com.sun.enterprise.naming
java.naming.factory.state=com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl

org.omg.CORBA.ORBInitialHost=localhost
org.omg.CORBA.ORBInitialPort=3700
```

Exercise 00.1: FirstEJB

Beschreibung

Erstelle dein erstes EJB 🙂
Es soll ein Rechner mit den vier Grundoprationen implementiert werden.

Aufgabe

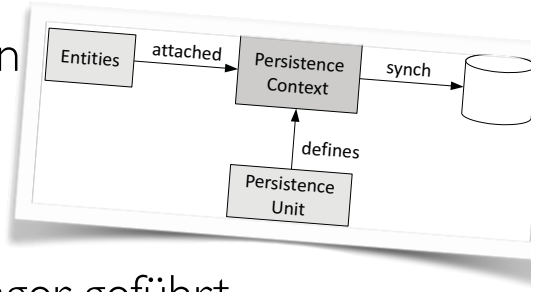
1. Lade den Sourcecode als Maven Projekt in deine IDE
2. Erstelle ein Remote Interface unter
   src/main/java/ch/hftm/example/Calculator.java
3. Implementiere das eben erstellte Interface
   src/main/java/ch/hftm/example/CalculatorBean.java
   (achte auf die DivisionByZeroException)
4. Deploye das EJB auf dem Applikationsserver
5. Erweitere die JUnit Klasse mit dem JNDI Namen und instanziere ein Remote Objekt
   src/test/java/ch/hftm/example/CalculatorIT.java
6. Führe den JUnit Test aus

ENTERPRISE JAVABEANS

Chapter 02 - Persistence

OVERVIEW

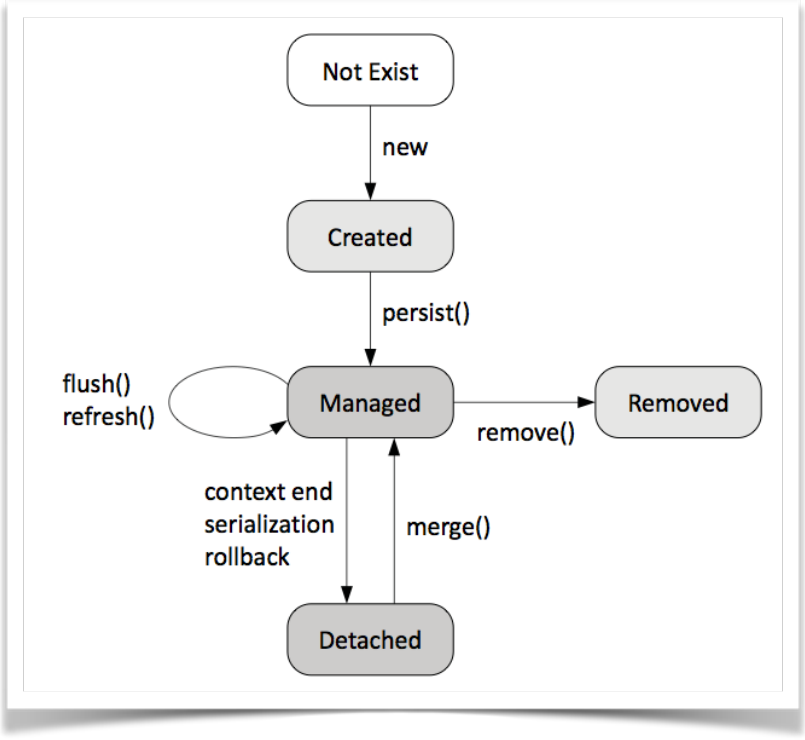
- Entities
  - Java Objekte mit *O/R mapping* annotation
  - Leichtgewichtige *Persistence Objects*
  - haben Eigenschaften, die ihren Zustand darstellen
  - müssen einen Primärschlüssel haben
- Persistence Context
  - Sammlung von Entities
  - Lebenszyklus wird vom Entity Manager geführt
- Persistence Unit
  - Ordnet eine Sammlung von Entities einer Datenbank zu



MANAGED / UNMANAGED ENTITIES

- Managed (attached) Entities:
  - Haben eine *persistence identity*
  - Gehören zu einem *persistence context*
  - Sind mit einer Datenbank synchronisiert
- Unmanaged (detached) Entities:
  - Haben eine *persistence identity*
  - Sind an kein *persistence context* gebunden
  - Sind nicht mit einer Datenbank synchronisiert

LIFE CYCLE OF A ENTITY



ENTITY MANAGER

- Führt die *persistence operations* aus (CRUD)
  - Create => entityManager.persist(obj);
  - Read => entityManager.find(Obj.class, name);
  - Update => entityManager.merge(obj);
  - Delete => entityManager.remove(obj);
- Sorgt für die automatische Synchronisierung und das Caching
- Ist für das *O/R mapping* zuständig

APPLICATION VS. CONTAINER

- Application-Managed Entity Manager
  - EntityManagerFactory wird zum Erstellen benötigt
  - *Persistence class* ist für das Laden zuständig

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("contactbook");
EntityManager entityManager = factory.createEntityManager();
```

- Container-Managed Entity Manager
  - EntityManager kann *injected* (@PersistenceContext) werden
  - EntityManager muss nicht geschlossen werden

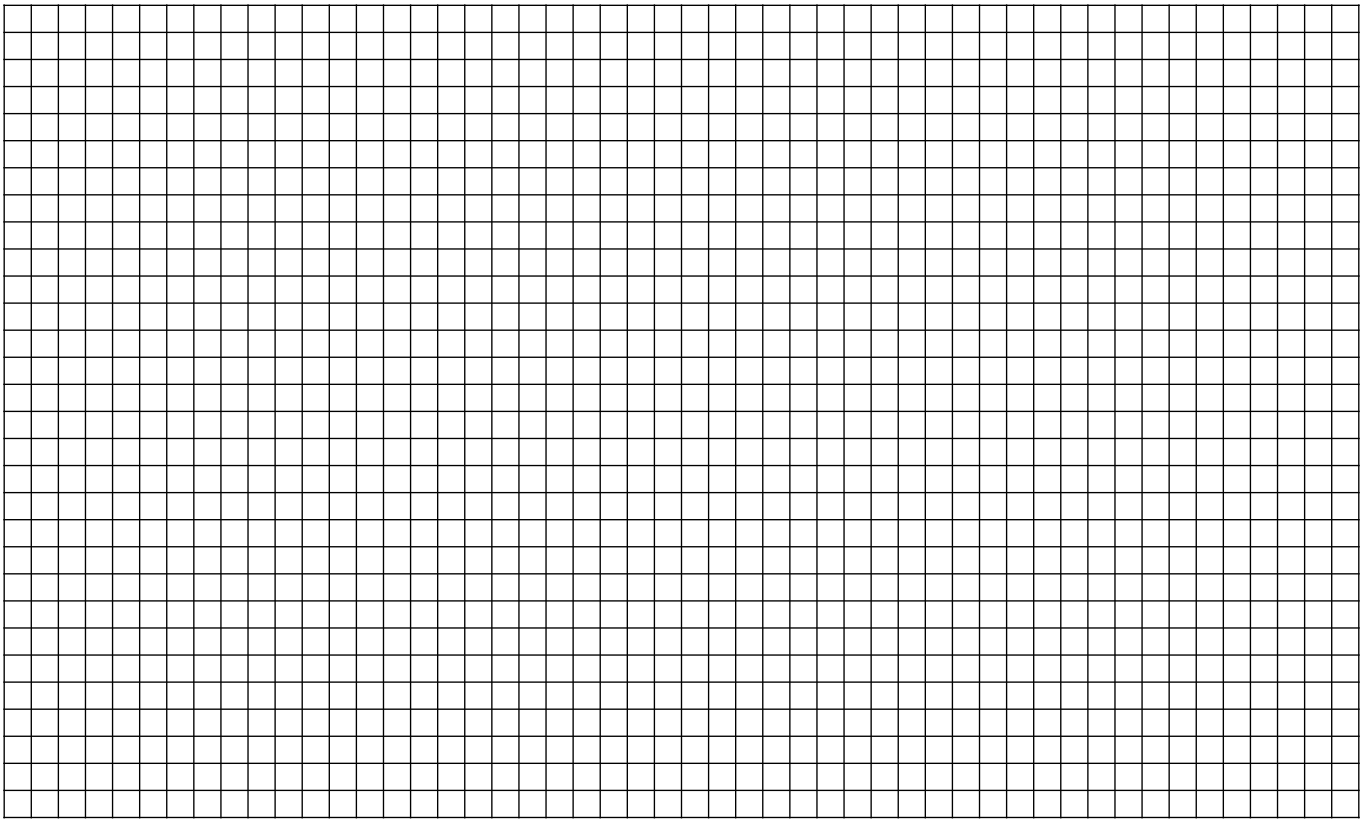
```
@PersistenceContext(unitName="contactbook")
private EntityManager entityManager;
```

EXTENDED / TRANSACTION

- Extended @PersistenceContext(type=EXTENDED)
  - hält die *entities* nach der abgeschlossenen Transaktion *managed*
  - dadurch kann auf lange Transaktionen verzichtet werden
  - können bei *stateful session beans* verwendet werden

- Transaction-Scoped @PersistenceContext(type=Transaction)
  - „Lebt“ solange wie die Transaktion
  - Wird automatisch geschlossen sobald die Transaktion ended
  - Kann bei *session* und *message-driven beans* eingesetzt werden
  - Aufrufen müssen sich um *exceptions* kümmern

EXTENDED / TRANSACTION



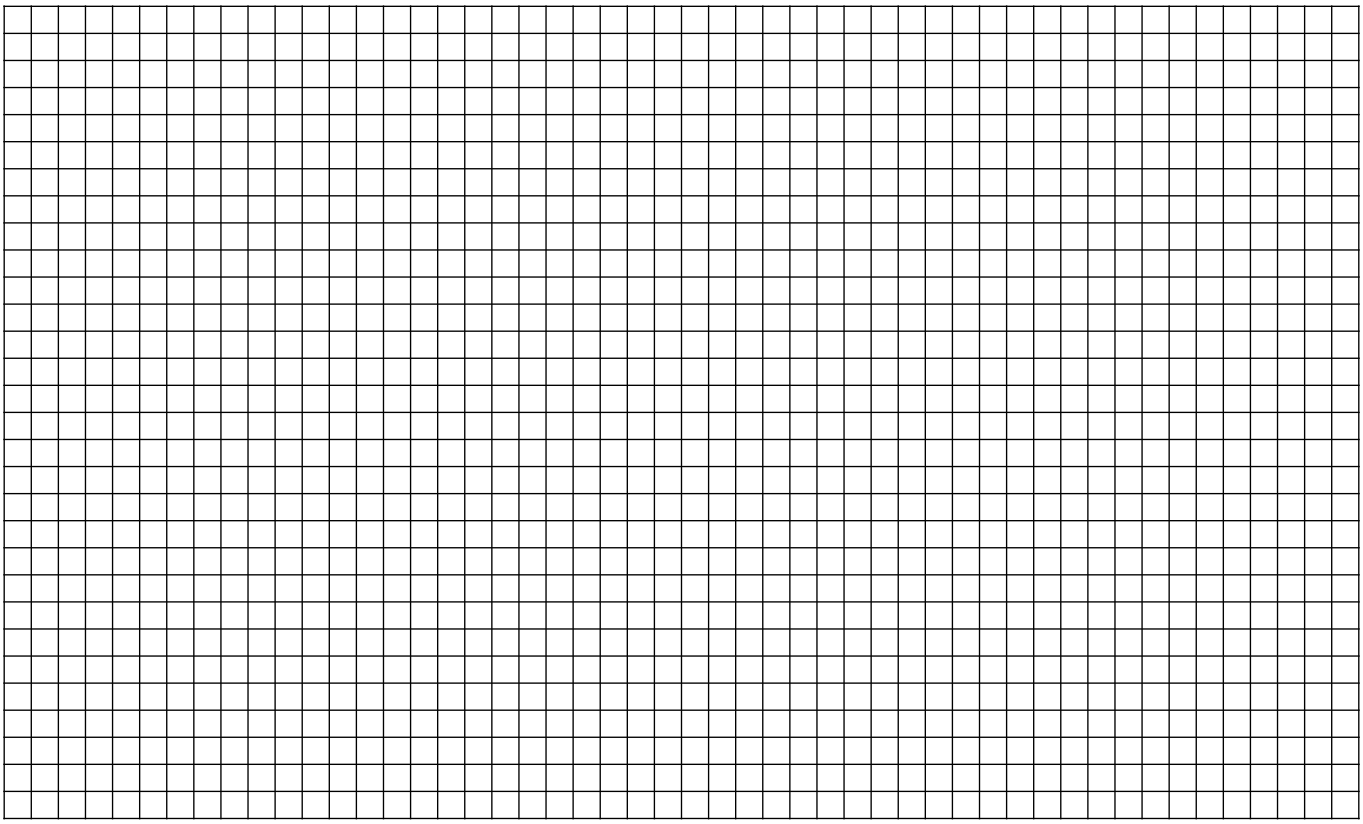
PERSISTENCE DEPLOYMENT DESCRIPTOR

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence version="2.0"
           xmlns="http://java.sun.com/xml/ns/persistence"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation=„http://java.sun.com/xml/ns/persistence
                          http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">

   <persistence-unit name="contactbook" transaction-type="JTA">
      <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
      <jta-data-source>jdbc/contactbook</jta-data-source>
      <properties>
         <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
         <property name="eclipselink.logging.level" value="FINE"/>
         <property name="eclipselink.logging.parameters" value="true"/>
      </properties>
   </persistence-unit>
</persistence>
```


MAP



Exercise 02: Persistence

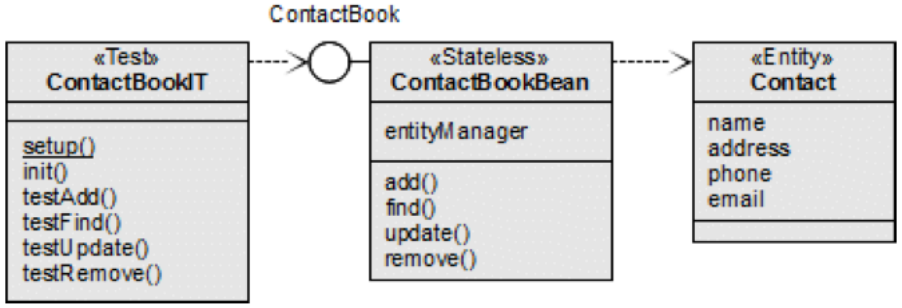
Beschreibung

Das Ziel dieser Übung ist es, die Umsetzung von CRUD-Operationen mit Hilfe eines Entity-Manager zu realisieren.

Diese Übung zeigt:

- wie ein Entity-Manager in eine Session-Bean injiziert wird
- wie ein Entity-Manager verwendet wird

Design



Vorgehen

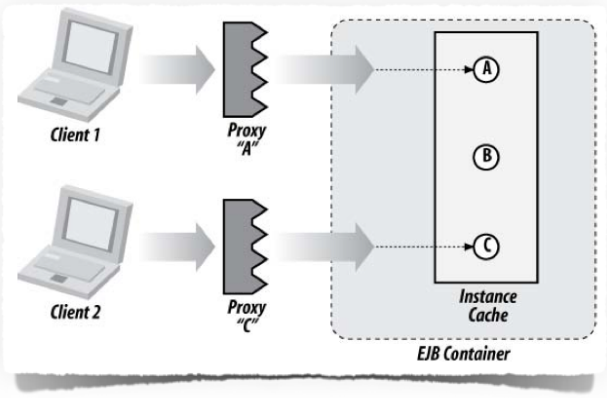
1. Erstellen Sie die Datenbank contactbook und definieren Sie sie als JNDI-Source jdbc/contactbook (sieh FAQ)
2. Implementieren Sie die Entity *Contact* und definieren Sie die Persistence Unit c*ontactbook*welche die Datenquelle *jdbc/contactbook*referenziert
3. Definieren Sie die Remote-Schnittstelle*ContactBookRemote* welche Methoden bietet um Kontakte hinzuzufügen, zu finden, zu aktualisieren und zu Entfernen
4. Erstellen Sie das Stateless Session Bean*ContactBookBean* welches das Interface*ContactBookRemote* implementiert
5. Deployen Sie das EJB Modul auf dem Glassfish Server und testen Sie die Funktionalität mit dem JUnit Test ContactBookIT

ENTERPRISE JAVABEANS

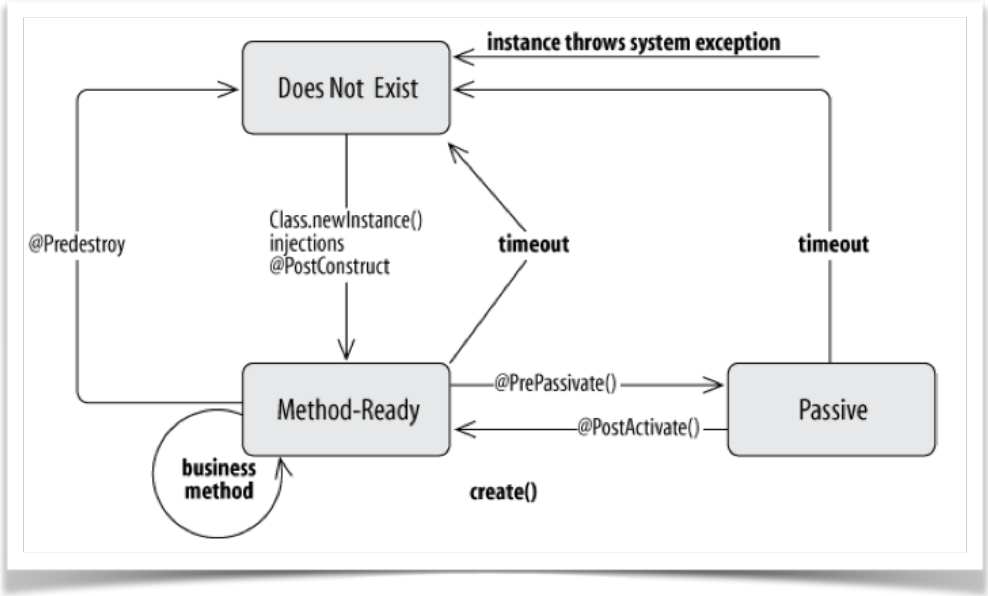
Chapter 04 - Stateful Session Bean

STATEFUL SESSION BEANS

- Sind einem Client für die ganze Bean Lebenszeit zugeordnet
- Unterhalten Zustand zwischen den Methoden Aufrufen



LIFECYCLE



LIFECYCLE

- Does Not Exist State
  - keine Instanz vorhanden
- Method-Ready Pool
  - Bean reagiert auf Client Anfragen
- Passivated State
  - Inaktiver Zustand
  - Nicht im Memory
  - Zustand bleibt erhalten

TRANSITION…

- Does Not Exist ⇨ Method-Ready Pool
  - @PostConstruct (initialize)
- Does Not Exist ⇦ Method-Ready Pool
  - @PreDestroy (cleanup)

- Method-Ready Pool ⇨ Passivated
  - @PrePassivate
- Method-Ready Pool ⇦ Passivated
  - @PostActivate

PASSIVATION/ACTIVATION

- Passivation
  - Der Zustand wird erhalten
    - Primitive Werte
    - Serialisierbare Objekte
    - SessionContext, EntityManager, UserTransaction
    - Referenzen zu anderen Beans
  - alles andere muss…
    - als transient deklariert werden oder
    - in der Methode @PrePassivate auf null gesetzt werden
    - und in @PostActivate neu initialisiert werden

DESTRUCTION

- SFSB werden zerstört…
  - …wenn der Client die @Remove Methode aufruft
  - …aufgrund eines Time Out's
  - … eine „system exception" auftritt
- Der Client erhält beim Versuch auf ein zerstörtes Bean zuzugreifen eine „NoSuchEJBException"

EXTENDED PERSISTENCE CONTEXT

- @PersistenceContext(type=EXTENDED)
  - hält die e*ntities* nach der abgeschlossenen Transaktion *managed*
  - dadurch kann auf lange Transaktionen verzichtet werden
  - *persistence context* wird aktualisiert (clean-up) sobald eine Bean entfernt wird

EJB REFERENCES

- Um die Referenz auf andere Beans zu erhalten, wird die Annotation @EJB verwendet

```
@EJB(beanName="ContactBook")
private ContactBookRemote contactbook;
```


THE INTERFACE

- Remote interfaces @Remote
  - Methoden können von „remote clients“ aufgerufen werden
  - Parameter und Rückgabewerte werden Kopiert und müssen dementsprechend serialisierbar sein
  - Kommunikation läuft über ein Netzwerkprotokoll
- Local interfaces @Local
  - Methoden können nur innerhalb des selben EJB Containers aufgerufen werden
  - Parameter und Rückgabewerte werden „by reference“ übertragen
  - Aufrufe sind effizienter

THE INTERFACE

```
@Remote
public interface AccountManagerRemote {

    public void login(Integer accountNr)
           throws AccountNotFoundException;

    public BigDecimal getBalance();

    public void deposit(BigDecimal amount);

    public void withdraw(BigDecimal amount);

    public void logout();
}
```


THE STATEFUL SESSION BEAN

```
@Stateful(name = "AccountManager")
public class AccountManagerBean implements AccountManagerRemote {

    @PersistenceContext(type = EXTENDED)
    private EntityManager entityManager;
    ……

    @Override
    public BigDecimal getBalance() {
        ……
        ……
    }

    ……
    ……
    ……

    @Override
    @Remove(retainIfException = true)
    public void logout() {
        account = null;
    }
```

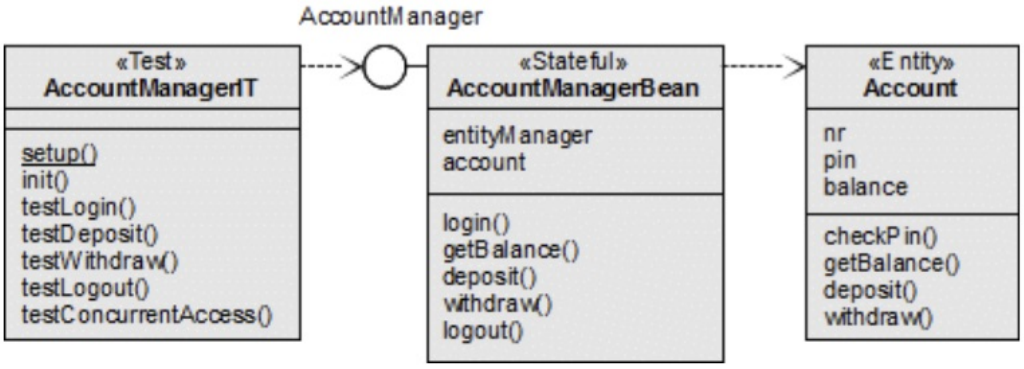
Exercise 04: Stateful Session Bean

Beschreibung

Das Ziel dieser Übung ist es, ein Stateful Session Bean zu implementieren, welches maipulationen auf Bankkonten ausführt.

Diese Übung zeigt:

- wie man eine Staeful Session Bean implementiert
- wie ein extended persistence context eingesetzt wird
- wie der Lebenszyklus eines Stateful Session Bean aussieht



Vorgehen

1. Implementieren Sie das Stateful Session Bean '*AccountManagerBean*' welches '*AccountManagerRemo* implementiert
2. Dabei sind folgende Funktionalitäten zu implementieren:
   - login => Das Konto wird mittels accountNr in das SFSB gespeichert (ohne PIN). Sollte kein Konto r der angegebenen Nr bestehen, wird eine AccountNotFoundException geworfen.
   - getBalance => Gibt den aktuellen Kontostand zurück.
   - deposit => Betrag auf Konto einzahlen.
   - withdraw => Betrag vom Konto abbuchen.
   - logout => setzt das lokal gespeicherte Konto auf null.
3. Setzten Sie ein extended persistence context zum managen der Bankkonten ein
4. Annotieren Sie die logout Methode als '*remove*' Methode
5. Deployen Sie das EJB Modul auf dem Glassfish Server und testen Sie die Funktionalität mit dem JUnit Test '*AccountManagerIT*'
6. Implementieren Sie '*life cycle methodes*' welche die '*life cyle events*' von '*AccountManagerBean*' logger und überprüfen Sie das Verhalten

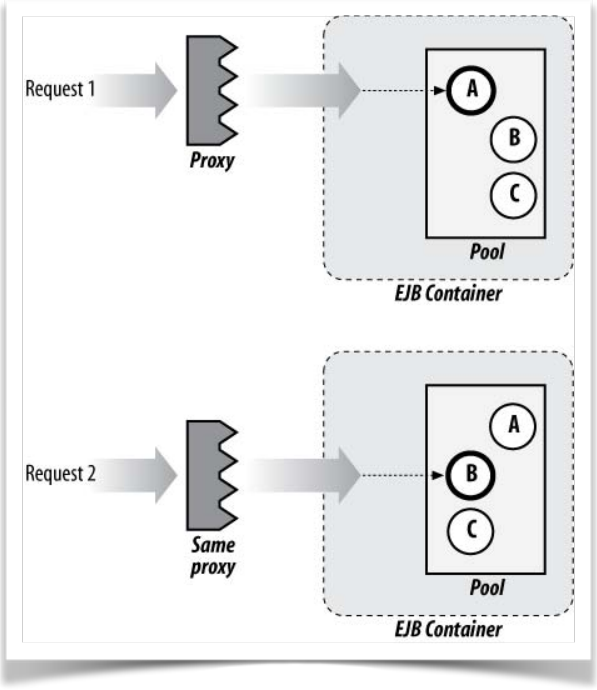
ENTERPRISE JAVABEANS

Chapter 03 - Stateless Session Bean

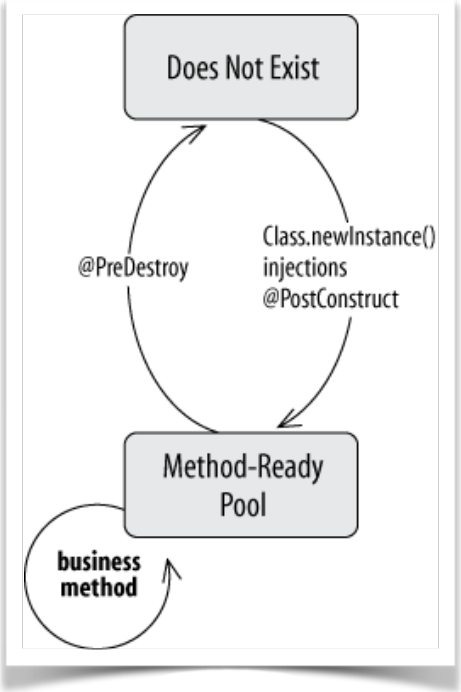
STATELESS SESSION BEANS

- sind Business Objekte welche keinen Zustand unterhalten
- Zugriff auf ein Bean ist auf ein Client pro Zeiteinheit limitiert
- Gleichzeitiger Zugriff ist untersagt
- Instanzen der SLSB sind typischerweise „gepoolt“
- Sind äussert effizient da sie unter den Client getauscht werden könnten

OVERVIEW



LIFECYCLE



LIFECYCLE

- Does Not Exist State
  - keine Instanz im Memory
  - wurde noch nicht instanziert
- Method-Ready Pool
  - sobald der Container sie benötigt
  - sobald der EJB Server restarted wird, werden üblicherweise einige Instanzen der SLSB erstellt
  - sobald die Anzahl der vorhandenen Beans unzureichend ist, können weitere Instanzen erstellt und dem Pool hinzugefügt werden

LIFECYCLE

- Does Not Exist ⇨Method-Ready Pool
  (Übergang zu Method-Ready Pool)
  - no-argument constructor is called
  - @PostConstruct (initialize)
    - return void
    - no parameters

```
@PostConstruct
public void initialize() {
    // Do initialization tasks here
}
```

- Does Not Exist ⇦ Method-Ready Pool
  (Übergang aus dem Method-Ready Pool)
  - @PreDestroy (cleanup)
    - return void
    - no parameters

```
@PreDestroy
public void cleanup() {
    // Do cleanup tasks here
}
```

THE INTERFACE

- Remote interfaces @Remote
  - Methoden können von „remote clients“ aufgerufen werden
  - Parameter und Rückgabewerte werden Kopiert und müssen dementsprechend serialisierbar sein
  - Kommunikation läuft über ein Netzwerkprotokoll
- Local interfaces @Local
  - Methoden können nur innerhalb des selben EJB Containers aufgerufen werden
  - Parameter und Rückgabewerte werden „by reference“ übertragen
  - Aufrufe sind effizienter

THE INTERFACE

```
@Remote
public interface AccountServiceRemote {

   public Integer openAccount(BigDecimal initialBalance);

   public BigDecimal getBalance(Integer accountNr)
         throws AccountNotFoundException;

   public void deposit(Integer accountNr, BigDecimal amount)
         throws AccountNotFoundException;

   public void withdraw(Integer accountNr, BigDecimal amount)
         throws AccountNotFoundException, WithdrawLimitExceededException;

   public void closeAccount(Integer accountNr)
         throws AccountNotFoundException, AccountNotBalancedException;
}
```

THE INTERFACE

```
@Local
public interface AccountServiceLocal {

   public Integer openAccount(BigDecimal initialBalance);

   public BigDecimal getBalance(Integer accountNr)
         throws AccountNotFoundException;

   public void deposit(Integer accountNr, BigDecimal amount)
         throws AccountNotFoundException;

   public void withdraw(Integer accountNr, BigDecimal amount)
         throws AccountNotFoundException, WithdrawLimitExceededException;

   public void closeAccount(Integer accountNr)
         throws AccountNotFoundException, AccountNotBalancedException;
}
```

ENVIRONMENT PROPERTIES

- Der „enterprise naming context“ (ENC) wird für die Speicherung von Konfigurationsdaten und „Resource Referenzen“ verwendet
- Sobald ein Bean „deployed“ wird…
  wird aus Annotationen und dem Deployment Descriptor (ejb-jar.xml) der ENC gebildet
- Sobald ein Bean instanziert ist…
  initialisiert der EJB Container das Bean mit diesen Werten mittels injection

ENVIRONMENT PROPERTIES

```
<?xml version="1.0" encoding="UTF-8"?>

<ejb-jar version="3.1" 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/ejb-jar_3_1.xsd">

   <enterprise-beans>
      <session>
         <ejb-name>AccountService</ejb-name>
         <env-entry>
            <env-entry-name>withdrawLimit</env-entry-name>
            <env-entry-type>java.lang.String</env-entry-type>
            <env-entry-value>1000</env-entry-value>
         </env-entry>
      </session>
   </enterprise-beans>

</ejb-jar>
```

THE STATELESS SESSION BEAN

```
@Stateless(name = "AccountService")
public class AccountServiceBean extends SessionLogger implements
      AccountServiceRemote, AccountServiceLocal {

   @PersistenceContext
   private EntityManager entityManager;

   @Resource(name = "withdrawLimit")
   private String withdrawLimit;

   @Override
   public Integer openAccount(BigDecimal initialBalance) {
      log("open account");
      Account account = new Account(initialBalance);
      entityManager.persist(account);
      entityManager.flush();
      return account.getNr();
   }

   …
```

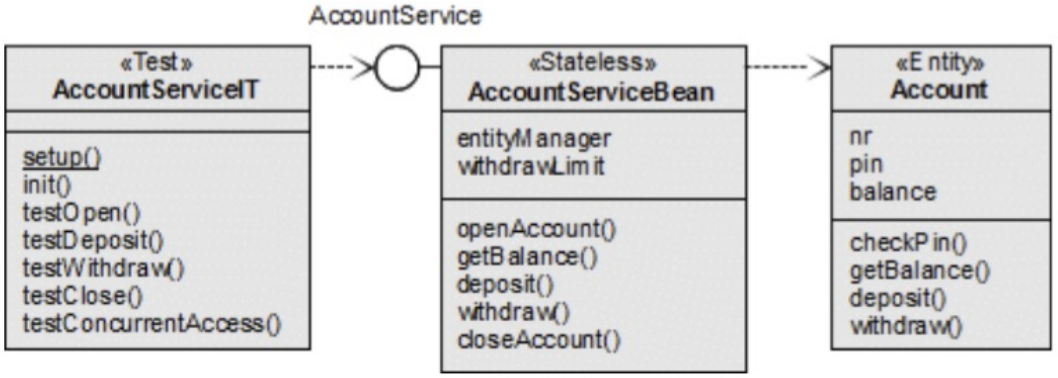
Exercise 03: Stateless Session Bean

Beschreibung

Das Ziel dieser Übung ist es, ein Stateless Session Bean zu implementieren, welches maipulationen auf Bankkonten ausführt.

Diese Übung zeigt:

- wie man eine Stateless Session Bean implementiert
- wie auf "environment properties" zugegriffen weden kann
- wie der Lebenszyklus einer Stateless Session Bean aussieht



Vorgehen

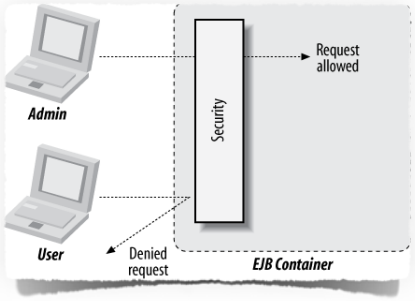
1. Erstellen Sie die Datenbank '*bank*' und definieren Sie sie als JNDI-Source '*jdbc/bank*' (siehe FAQ)
2. Implementieren Sie das Stateless Session Bean '*AccountServiceBean*' welches '*AccountServiceRemot* implementiert
3. Dabei sind folgende Funktionalitäten zu implementieren:
    1. openAccount => Neuen Account erstellen (mit Startkapital) und persistieren, Rückgabewert ist die Account Nr. (generierter Wert von JPA bzw. DBS).
    2. getBalance => Gibt den Kontostand anhand der accountNr zurück.
    3. deposit => Betrag auf Konto (via accountNr) einzahlen.
    4. withdraw => Betrag vom Konto (via accountNr) abbuchen, dabei ist zu Prüfen ob die Summe das withdrawLimit nicht übersteigt. Ist dies der Fall soll eine WithdrawLimitExceededException geworfen werden.
    5. closeAccount => Löscht eine Konto (Account). Dazu muss allerdings das Konto einen Kontostand v 0 aufweisen, andernfalls soll eine AccountNotBalancedException geworfen werden.
4. Erstelle den Deployment Descriptor '*ejb-jar.xml*' und definiere das environmentproperty '*withdrawLimit*' dem Wert 1000
5. Deployen Sie das EJB Modul auf dem Glassfish Server und testen Sie die Funktionalität mit dem JUnit Test '*AccountServiceIT*'
6. Optional: Implementieren Sie '*life cycle methodes*' welche die '*life cyle events*' von '*AccountServiceBea* loggen und überprüfen Sie das Verhalten
    - Erstellen Sie ein Klasse mit dem Namen: *SessionLogger (ch.hftm.util)*

ENTERPRISE JAVABEANS

Chapter 07 - Security

OVERVIEW

- Multiuser Applikationen stellen in der Regel eine Vielzahl an verschiedenen Funktionen zur Verfügung, welche nicht für alle user verwendbar sein sollen.
- EJB stellt zur Bewältigung dieser Anforderung ein rollenbasierter Sicherheitsmechanismus zur Verfügung.
- Dabei muss der bestehende Code in der Regel nicht umfangreich angepasst werden, da sich der Container um die Durchsetzung der Richtlinien kümmert.



TERMINOLOGY

- Realm
  - Ein *realm* ist eine *security policy domain* welche auf dem Web- bzw. Applikations-Server definiert ist
  - Authentifizierungsdienste können Informationen aus unterschiedlichen *realms* Informationen beziehen, nachfolgend einige Beispiele
    - admin-realm (*bei Glastisch für Server Auth. verwendet)
      - *administrator user credentials* lokal in einem File
    - file
      - *credentials* lokal in einem File
    - certificate
      - *credentials* in einer Zertifikat Datenbank
    - database
      - *credentials* in einer Datenbank

TERMINOLOGY

- User
  - Ein *user* ist eine Identität für ein Individuum oder eine Program Identität welche in dem *realm* definiert ist
  - *users* können mit *groups* assoziiert werden
- Group
  - Eine *group* ist eine Sammlung von authentifizierten *users*
  - *group* werden in den *realms* definiert

TERMINOLOGY

- Role
  - Eine *role* ist ein abstrakter Name einer Berechtigung für eine Sammlung von Ressourcen in der Applikation
  - eine *role* kann mit einem Schlüssel welche eine Tür öffnet verglichen werden

ASSIGNING METHOD PERMISSIONS

- @RolesAllowed("……")
  - *role*s welche berechtigt sind auf die Methode zuzugreifen
- @PermitAll
  - Jeder Authentifizierte Benutzer ist berechtigt
- @DenyAll
  - niemand ist berechtigt

Annotation auf dem Bean definiert die „standard" Berechtigung aller Methoden

PROGRAMMATIC SECURITY

Um Informationen zum aktuellen *user* zu erhalten, kann der *session context* benutzt werden. So können auch erweiterte (programatische) Sicherheits-Regeln implementiert werden.

```
// declare all roles which are used in the bean
@DeclareRoles("manager")
public class ……

// Hook to the container to get security information
@Resource
private SessionContext context;

// Get the caller
final String callerName = context.getCallerPrincipal().getName();

// enforce specific logic
if (context.isCallerInRole("manager")){
   ……
}
```


THE RUNAS SECURITY IDENTITY

- @RunAs("……")
  - Definiert die *role* mit welcher ein EJB auf eine Methode eines anderen EJB's zugreift

Exercise 07: Security

Beschreibung

Das Ziel dieser Übung ist es, Methodenzugriff der Bankanwendung zu beschränken.

Diese Übung zeigt:

- wie man Sicherheitsrollen verwendet
- wie man Benutzer und Gruppen erstellt

Vorgehen

1. Erstellen Sie auf dem GlassFish Server die folgenden User:
    1. user: nick / pw: nick / group: employee,manager
    2. user: vicky / pw: vicky / group: employee
2. Implementieren Sie das Singleton Session Bean '*ReportingServiceBean*' welches '*ReportingServiceRemote*' implementiert
3. Implementieren Sie die Methode getNumberOfAccounts welche die Anzahl aller Accounts zurückgibt (Integer)
4. Implementieren Sie die Methode getTotalBalance welche die Summe aller Konten (balance) zurückgibt (BigDecimal)
5. Beschränken Sie den Zugriff auf NumberOfAccounts für die Rolle "employee"
6. Beschränken Sie den Zugriff auf getTotalBalance für die Rolle "manager"
7. Deployen Sie das EJB Modul auf dem Glassfish Server und testen Sie die Funktionalität mit dem JUnit Test ReportingServiceIT

ENTERPRISE JAVABEANS

Chapter 06 - TimerService

OVERVIEW

- Ausführung zu einer spezifizierten Zeit
- kann mit folgenden Beans verwendet werden:
  - stateless / singleton / message-driven
- zwei Wege zur Implementation:
  - programmatically
    - werden durch einen Methodenaufruf (eines clients) gestartet
  - declaratively
    - automatisch während deployment gestartet

TWO WAYS TO DO IT

- programmatically => @Timeout
  - Verwendet „*timer service*“
  - EJB container ruft die „*@Timeout Methode*“ auf
- declaratively => @Schedule
  - erstellt automatisch einen Timer
  - kann auf jeder Methode angewendet werden

TIMER INTERFACE

- ist persistent
- repräsentiert das „*geschedulte*“ Objekt
- kann verwendet werden um Daten aufzurufen
- auslesen wann der Timer abläuft
- Timer abbrechen

TIMER INTERFACE

- Timer Service…
  - single-action
    - …createSingleActionTimer(Date expiration, TimerConfig config)
  - interval
    - …createIntervalTimer(Date expiration, long interval, TimerConfig config)
  - calendar based
    - …createCalendarTimer(ScheduleExpression schedule, TimerConfig config)

SCHEDULE EXPRESSION

- definiert Kalender basierte Timer
- Werte zur Konfiguration:
  - year, month, day of month, day of week, hour, minute, second
- Attribute
  - * / list 0,2,4 / range 0-5 /
  - alle 5 sekunden ⇨ second = "*/5"

```
ScheduleExpression schedule = new ScheduleExpression();
schedule.year("2015");
schedule.month("12");
schedule.dayOfMonth("24");
```

PROGRAMMING A TIMER

```
@Singleton(name = "InterestService")
public class InterestServiceBean implements InterestServiceRemote {

   @Resource
   private TimerService timerService;

   public void schedulePayments(Date expiration) {
      timerService.createSingleActionTimer(expiration, new TimerConfig());
   }

   public void schedulePayments(ScheduleExpression schedule) {
      timerService.createCalendarTimer(schedule);
   }

   public void cancelPayments() {
      ……
   }

   @Timeout
   public void payInterests() {
      ……
   }
}
```


DECLARING A TIMER

```
import javax.ejb.Schedule;
import javax.ejb.Singleton;

@Singleton(name = "Christmas")
public class Christmas {

  @Schedule(year = "*", month = "12", dayOfMonth = "24")
  public void deliferGifts() {

  }

}
```

Exercise 06: Timer Service

Beschreibung

Das Ziel dieser Übung ist es, ein Singleton Session Bean zu implementieren, welches Zins auf Bankkonten zahlt.

Diese Übung zeigt:

- wie man einen Timer programmatisch und deklarativ einsetzt
- wie man eine Singleton Session Bean implementiert

Vorgehen

1. Implementieren Sie das Singleton Session Bean '*InterestServiceBean*' welches '*InterestServiceRemote* implementiert
2. Erweitere den Deployment Descriptor '*ejb-jar.xml*' und definiere das environmentproperty 'interestRate mit dem Wert 0.05
3. Erweitern Sie das Singleton Session Bean '*InterestServiceBean*' mit Timer Services
4. Deployen Sie das EJB Modul auf dem Glassfish Server und testen Sie die Funktionalität mit dem JUnit Test InterestServiceIT
5. Schedule die payInterests Methode dekklarativ (@Shedule) deploye und überprüfe die Ausführung

How EJB3 timer service works?

- You have to specify a callback method in bean class, often called as timeout method, which is annotated with `@Timeout`. This method contains the business logic that handles the timed event.
- When a timer expires (goes off), container automatically invokes this timeout method.

Creating TimerService

To create a timer, we need to create TimerService object and use one of its createTimer() method. TimerService can be accessed in one of the following ways.

Injecting TimerService using @Resource annotation

TimerService is injected by the container into the bean component using @Resource annotation.

```
01  import javax.annotation.Resource;
02  import javax.ejb.Stateless;
03  import javax.ejb.TimerService;
04  import com.theopentutorials.ejb3.business.TimerRemote;
05
06  @Stateless
07  public class TimerBean implements TimerRemote {
08      @Resource
09      TimerService service;
10  }
```

Creating TimerService using EJB context

TimerService can be accessed using EJBContext (or more specifically SessionContext or MessageDrivenContext).

Using EJBContext

```
01  import javax.annotation.Resource;
02  import javax.ejb.EJBContext;
03  import javax.ejb.Stateless;
04  import javax.ejb.TimerService;
05  import com.theopentutorials.ejb3.business.TimerRemote;
06
07  @Stateless
08  public class TimerBean implements TimerRemote {
09
10      @Resource
11      EJBContext context;
12
13      public void startTimer() {
14          TimerService service = context.getTimerService();
15      }
16  }
```

Using SessionContext

```
01  import javax.annotation.Resource;
02  import javax.ejb.SessionContext;
03  import javax.ejb.Stateless;
04  import javax.ejb.TimerService;
05  import com.theopentutorials.ejb3.business.TimerRemote;
06
07  @Stateless
08  public class TimerBean implements TimerRemote {
09
10      @Resource
11      SessionContext context;
12
13      public void startTimer() {
14          TimerService service = context.getTimerService();
15      }
16  }
```

Creating Timer

After getting the TimerService instance using one of the methods discussed above, you can invoke one of the createTimer() methods of the TimerService interface.

Method	Description
Timer createTimer(long duration, java.io.Serializable info)	Create a single-action timer that expires after a specified duration in milliseconds and not repeated.

Timer createTimer(long initialDuration, long intervalDuration, java.io.Serializable info)	Creates timer at timed intervals with initial timeout and interval duration in milliseconds.
Timer createTimer(java.util.Date expiration, java.io.Serializable info)	Creates a single-action timer that expires at a specific time.
Timer createTimer(java.util.Date initialExpiration, long intervalDuration, java.io.Serializable info)	Creates an interval timer whose first expiration occurs at a given point in time and whose subsequent expirations occur after a specified interval.

These methods throw java.lang.IllegalArgumentException, java.lang.IllegalStateException, javax.ejb.EJBException

Examples:

1.createTimer(long, serializableInstance)

Create a single-action timer that is fired after a specified duration i.e.)1 hour (60*60*1000 milliseconds) and not repeated.

```
1  public void startTimer() {
2      Timer timer = service.createTimer(60*60*1000, null);
3      System.out.println("Timers set");
4  }
```

2.createTimer(long, long, serializableInstance)

This code creates timer at timed intervals with initial timeout (10000 milliseconds) and interval duration of 1 hour (60*60*1000 milliseconds).

```
1  public void startTimer() {
2      Timer timer = service.createTimer(10000, 60*60*1000, null);
3      System.out.println("Timers set");
4  }
```

3.createTimer(date, serializableInstance)

This code creates a single-action timer that fires at a specific time (July 29th at 8:30 PM).

```
01  public void startTimer() {
02      SimpleDateFormat dateFormat  = new SimpleDateFormat("dd/MM/yyyy HI
03      String s = "29/07/2012 20:30:00";
04      try {
05          Date date = dateFormat.parse(s);
06          Timer timer = service.createTimer(date, null);
07          System.out.println("Timers set");
08      } catch (ParseException e) {
09          e.printStackTrace();
10      }
11  }
```

4.createTimer(date, long, serializableInstance)

This code creates an interval timer whose first expiration occurs at a given point in time (July 29th at 8:30 PM) and whose subsequent expirations occur after a specified interval i.e.) every 30 days (30*24*60*60*1000 milliseconds).

```
01  public void startTimer() {
02      SimpleDateFormat dateFormat  = new SimpleDateFormat("dd/MM/yyyy HI
03      String s = "29/07/2012 20:30:00";
04      try {
05          Date date = dateFormat.parse(s);
06          Timer timer = service.createTimer(date, 30*24*60*60*1000, nul
07          System.out.println("Timers set");
08      } catch (ParseException e) {
09          e.printStackTrace();
10      }
11  }
```

Client or any component may invoke a method in bean which creates a timer. For example, client can invoke the above method, startTimer(), on bean to start the timer.

The Timeout Method

Rules for implementing timeout method:

- Bean class can have at most one method annotated with @Timeout.
- This method must return void.
- Take javax.ejb.Timer object as the only parameter.
- May not throw application exceptions.

```
1  @Timeout
2  public void handleTimeout(Timer timer) {
3      System.out.println("Handle timeour event here...");
4  }
```

Refer this link for EJB3 Timer example.

Timers are persistent

- Timers are saved when the server shut downs (or even crashes). They become active again when the server is restarted.
- If a timer expires while the server is down, the container will call the @Timeout method when the server is restarted.

In JBoss AS timers are saved at,

- JBoss AS 7: <JBOSS_HOME>/standalone/data/timer-service-data.
- JBoss AS 6/5: <JBOSS_HOME>/server/<servername>/timer-service-data.
    - ‘servername’ can be default

The timer persistence is a directory and if you do not want to persist the timers, you can manually delete the folder or remove the element “data-store” from the ejb3 subsystem.

```
1  <subsystem xmlns="urn:jboss:domain:ejb3:1.2">
2  <timer-service thread-pool-name="default">
3          <data-store path="timer-service-data" relative-to="jboss.serve:
4  </timer-service>
5  </subsystem>
```

You can also specify at creation time that it does not have to be persisted using TimerConfig and passing it when creating timer using one of the create method.

```
1  @Override
2  public void startTimer() {
3      TimerConfig config = new TimerConfig();
4      config.setPersistent(false);
5
6      Timer timer = service.createSingleActionTimer(60*60*1000, config);
7  }
```

Refer this link for TimerConfig example.

Cancelling Timers

Timers can be cancelled when one of the following events occur;

- When a single-event timer expires, the EJB container calls the @Timeout method and then cancels the timer.
- When the bean invokes the cancel() method of the Timer interface, the container cancels the timer.

```
1  @Timeout
2  public void handleTimeout(Timer timer) {
3      System.out.println("Handle timeour event here...");
4      timer.cancel();
5  }
```

When a method is invoked on a cancelled timer, the container throws the javax.ejb.NoSuchObjectLocalException.

Saving Timers

- Timer object can be saved in a database or file for future reference.
- Invoke getHandle() method and store the TimerHandle object in a database. (A TimerHandle object is serializable.)
- To re-instantiate the Timer object, retrieve the handle from the database and invoke getTimer() on the handle.

```java
@Override
public void startTimer() {
    Timer timer = service.createTimer(10000, 60*60*1000, null);

    TimerHandle handle = timer.getHandle();
    //save handle object in database/file i.e.) serialize

    Timer timer2 = handle.getTimer();
}
```

Getting Timer Information

Timer interface defines methods for obtaining information about timers;

public long getTimeRemaining();

Get the number of milliseconds that will elapse before the next scheduled timer expiration.

public java.util.Date getNextTimeout();

Get the point in time at which the next timer expiration is scheduled to occur.

public java.io.Serializable getInfo();

The getInfo() method returns the object that was the last parameter of the createTimer invocation.

```java
@Override
public void startTimer() {
    Employee employee = new Employee();
    employee.setName("abcd");
    //save employee in database
    Timer timer = service.createTimer(10000, 60 * 60 * 1000, employee
}

@Timeout
public void handleTimeout(Timer timer) {
    Employee emp = (Employee) timer.getInfo();
    // process emp object
}
```

java.util.Collection getTimers()

This method returns a collection of Timer objects.

```java
public String checkStatus() {
    Timer timer = null;
    Collection<Timer> timers = service.getTimers();
    Iterator<Timer> iterator = timers.iterator();
    while (iterator.hasNext()) {
        timer = iterator.next();
        return ("Timer will expire after " + timer.getTimeRemaining()
                milliseconds.");
    }
    return ("No timer found");
}
```

Transactions and Timers

- Timers can survive container restart or crash.
- Timers are transactional.
  - Bean creates a timer within a transaction.
  - Timer creation is rolled back, if a transaction is rolled back.
  - Timer cancellation is rolled back if a bean cancels a timer within a transaction that is rolled back.
  - Transaction failure inside a timeout method rolls back the actions taken by the timer.

SetUp JTA Datasource

1. Start Database Server with GlassFish

- Eclipse -> Preferences -> GlassFish Preferences -> Start the JavaDB database process when Starting GlassFish Server
- restart Glassfish

2. SetUp Database

- Data Source Explorer (Database Development Perspective)
- Database Connections
- New...
  - Derby
  - set name: contactbook
  - Drivers: GlassFishSampleDB
  - Database: contactbook
  - Host: localhost
  - Port number: 1527
  - User name: user
  - Password: user
  - check: Save password
- Test Connection

3. Set Up JDBC Connection Pools

- GlassFish Admin Console (http://localhost:4848)
- Resources -> JDBC -> JDBC Connection Pools
- New...
  - Pool Name: Contactbook_Pool
  - Resource Type: javax.sql.XADataSource
  - Database Driver Vendor: Derby
- Next
- Additional Properties
  - User: user
  - DatabaseName: contactbook

    - Password: user
- Finish

4. Set Up JDBC Resource

- GlassFish Admin Console (http://localhost:4848)
- Resources -> JDBC -> JDBC Resource
- New...
    - JNDI Name: jdbc/contactbook
    - Pool Name: Contactbook_Pool
- OK

Zuletzt geändert: Freitag, 22. November 2013, 19:32

Create user on GlassFish Server

- Open GlassFish Administration Console (http://localhost:4848)
- Configurations -> server-config -> Security:
    - Enable "Default Principal To Role Mapping"
- Configurations -> server-config -> Security -> Realms -> file
    - Manage Users
    - New...
        - User ID: xyz
        - Group List: a,b,c
        - New Password: ***

Zuletzt geändert: Donnerstag, 19. Dezember 2013, 22:22

▼ To Create an LDAP Realm in the GlassFish Server

1. Launch and log in to the GlassFish Admin Console.

   The default URL for the console is http://localhost:4848.

2. In the left navigation panel, expand Configuration, expand Security, and then click Realms.

3. Above the Realms list, click New.

The New Realm page appears.

4. Enter LdapRealm for the name and select com.sun.enterprise.security.auth.realm.ldap.LDAPRealm for the class name.

5. Enter at least the following properties:

   - JAAS context: The type of login to use for this realm. For LDAP, it must be ldapRealm.
   - Directory: The URL of the directory server. For example, ldap://190.111.0.111:389.
   - Base DN: The base Distinguished Name (dn) for the user data.

   You can specify additional optional properties for the realm.

6. Click OK.

7. Continue to To Update web.xml for the Worklist Manager Console (for LDAP)

   https://docs.oracle.com/cd/E19182-01/821-0871/gjepj/index.html

Maven: Generating ejb project

Generating ejb-javaee6 project

- mvn archetype:generate -DarchetypeGroupId=org.codehaus.mojo.archetypes -DarchetypeArtifactId=ejb-javaee6

Convert maven project to eclipse project

- mvn eclipse:eclipse

EJB FAQ

Here are answers to some frequently asked questions about how to use Enterprise Java Beans within GlassFish. Additional resources can be found here. Please send any follow-up questions/comments to ejb@glassfish.java.net or the GlassFish forum.

EJB Clients

- How do I access a Remote EJB component from a stand-alone java client?
- Is a stand-alone Java client portable? What's the difference between a stand-alone Java client and an Application Client component?
- How do I access a Remote EJB (3.0 or 2.x) component from a non-Java EE web container like Tomcat or Resin?
- What if I have multiple instances of the Appserver running and I want to access a Remote EJB component between them?
- Do I need RMI stubs to access EJB components from my java client?
- What if I have an existing stand-alone java client that accesses EJB components through the `CosNaming JNDI` provider ? How do I get static `RMI-IIOP` stubs for it?
- What is the relationship between `@EJB` and `<ejb-ref>`/`<ejb-local-ref>`?
- I have a Remote EJB dependency (`@EJB` or `<ejb-ref>`) in my Java EE component. How do I control which actual target EJB component it refers to?

Local EJB Access

- How do I access a Local EJB component from a POJO or utility class?
- I have an EJB component with a Local interface. Can I access it from an Application Client or a stand-alone java client ?
- I have an EJB component with a Local interface. Can I access it from a web component in a different application?

JNDI names

- Are Global `JNDI` names portable? How do I know if my EJB component has a portable `JNDI` name?
- What is the syntax for portable global `JNDI` names?
- The default portable `JNDI` names are great, but how do I define my own portable global `JNDI` name?
- How are GlassFish-specific (non-portable) Global `JNDI` names assigned to Session / Entity beans?
- How do I specify the Queue or Topic that a Message Driven Bean should consume from?

Testing / Embeddable API

- **How do I "unit test" EJB components?**

How do I access a Remote EJB component from a stand-alone java client?

Step 1. Use the no-arg `InitialContext()` constructor in your code.

The most common problem developers run into is passing specific `JNDI` bootstrapping properties to `InitialContext(args)`. Some other vendors require this step but GlassFish does **not**. Instead, use the no-arg `InitialContext()` constructor.

Step 2. Pass the global JNDI name of the Remote EJB to `InitialContext.lookup()`

Stand-alone java clients do not have access to a component naming environment (`java:comp/env`) or to the `@EJB` annotation, so they must explicitly use the global `JNDI` name to lookup the Remote EJB. (See here for more information on how global `JNDI` names are assigned to EJB components) Assuming the global `JNDI` name of the Remote EJB is `"FooEJB"` :

For Beans with a 3.x Remote Business interface :

```
Foo foo = (Foo) new InitialContext().lookup("FooEJB");
```

Note that in the EJB 3.x case the result of the lookup can be directly cast to the remote business interface type without using `PortableRemoteObject.narrow()`.

For EJB 2.1 and earlier session/entity beans :

```
Object homeObj = new InitialContext().lookup("FooEJB");

FooHome fooHome = (FooHome)
  PortableRemoteObject.narrow(homeObj,FooHome.class);

Foo foo = fooHome.create(...)
```

Step 3. Include the appropriate GlassFish .jars in the java client's classpath.

For GlassFish 3.

Include `$GLASSFISH_HOME/glassfish/lib/gf-client.jar` in the client's classpath.

E.g., assuming the application classes are in /home/user1/myclasses and the main client class is `acme.MyClient` :

```
java -classpath $GLASSFISH_HOME/glassfish/lib/gf-
client.jar:/home/user1/myclasses acme.MyClient
```

Note that the Java EE 6 API classes are automatically included by `gf-client.jar` so there is no need to explicitly add javaee.jar to the classpath. `gf-client.jar` refers to many other .jars from the GlassFish installation directory so it is best to refer to it from within the installation directory itself rather than copying it(and all the other .jars) to another location.

Note: `gf-client.jar` is located in `$GLASSFISH_HOME/modules/gf-client.jar` in GlassFish v3.

For GlassFish v2 and earlier

Include both `$GLASSFISH_HOME/lib/appserv-rt.jar` and `$APS_HOME/lib/javaee.jar` in the client's classpath.

E.g., assuming the application classes are in /home/user1/myclasses and the main client class is `acme.MyClient` :

```
java -classpath $GLASSFISH_HOME/lib/appserv-
rt.jar:$GLASSFISH_HOME/lib/javaee.jar:/home/user1/myclasses acme.MyClient
```

Note : `appserv-rt.jar` uses the JAR MANIFEST-CLASSPATH attribute to include other dependent .jars within the lib directory. When setting your client classpath, it is best to directly refer to the `appserv-rt.jar` within the app server lib directory rather than copying it to another location. If you do copy `appserv-rt.jar`, you'll need to copy additional .jars such as `appserv-deployment-client.jar` and `appserv-ext.jar`. The full set of .jars that might be needed by appserv-rt.jar is located in its META-INF/MANIFEST.MF file.

If your stand-alone client makes use of JMS, you'll also need to add `$GLASSFISH_HOME/lib/install/applications/jmsra/imqjmsra.jar`, `$GLASSFISH_HOME/lib/appserv-admin.jar` and `$GLASSFISH_HOME/lib/appserv-ws.jar`.

If your stand-alone client makes use of the Java Persistence API (e.g. it calls a Remote EJB component that returns a Java Persistence API entity), you'll also need to add `$APS_HOME/lib/toplink-essentials.jar`.

Step 4. Set the server host property, if necessary.

If the stand-alone java client is running on a different host than the server, set the `-Dorg.omg.CORBA.ORBInitialHost` property when starting the client JVM. E.g.
`java -Dorg.omg.CORBA.ORBInitialHost=com.acme.Host1`. This property defaults to `localhost`, so it is not necessary to set it if the java client is running on the same machine as the server.

Step 5. Set the naming service port property, if necessary.

The default naming service port in the app server is `3700`. If the naming service is running on a different port, you'll need to set it via the `-Dorg.omg.CORBA.ORBInitialPort` property when starting the client JVM. You can double-check the actual naming service port for a given server instance by looking in the server instace's domain.xml for `"orb-listener-1"`. Alternatively, check the Applications..Configuration..ORB..IIOP Listeners section of the admin GUI for `orb-listener-1`.

Is a stand-alone Java client portable? What's the difference between a stand-alone Java client and an Application Client component?

The Java EE platform defines a component that is specially designed to portably access Java EE services from a JVM running outside of the Application Server. It is called a Java EE Application Client and has been part of the platform since the first release (J2EE 1.2). Like all Java EE components it runs within a container provided by the vendor's implementation. The main advantages of the Application Client are that it's portable and that it allows the developer to use the same programming model for defining and accessing resources as is used within web components and EJB components. It follows the overall philosophy of the Java EE platform that as much of the "plumbing" or system-level work as possible should be performed by a container instead of being part of the Application code. That means a guarantee that the no-arg `InitialContext` constructor will work, that a private component naming context (`java:comp/env`) is available, and in Java EE 5 that platform annotations and injection are available.

One of the most common issues faced by developers is how to initialize the naming context within a client that accesses EJB components. When such a client is *not* written as an Application Client it is referred to as a "stand-alone" java client. By definition, such stand-alone java clients are not portable, so each vendor defines its own way to bootstrap the naming service. This not only makes it more difficult to code the client but causes problems when moving between Java EE implementations.

Like all Java EE components, there are some additional steps required to achieve the portability offerered by the Application Client. E.g., definining a deployment descriptor, packaging the application client .jar and learning how to run the Application Client container.

See the following for more details on using Application Clients :

Chapter 22 (Getting Started with EJBs) of the Java EE 5 tutorial

Simple EJB 3.0 session / message-driven bean code examples

Client chapter of our Developer's Guide

How do I access a Remote EJB (3.0 or 2.x) component from a non-Java EE web container like Tomcat or Resin?

Accessing a Remote EJB component from a non-Java EE web container is similar to the stand-alone java client case. However, the complication is that most Java web servers set the default `JNDI` name provider for the JVM, which prevents our appserver naming provider from being instantiated when the application uses the no-arg `InitialContext()` constructor. The solution is to explicitly instantiate an `InitialContext(Hashtable)` with the properties for our naming provider, as contained in GlassFish's jndi.properties file.

Step 1. Instantiate the InitialContext

```
Properties props = new Properties();

props.setProperty("java.naming.factory.initial",
                  "com.sun.enterprise.naming.SerialInitContextFactory");

props.setProperty("java.naming.factory.url.pkgs",
                  "com.sun.enterprise.naming");

props.setProperty("java.naming.factory.state",
                  "com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl");


// optional.  Defaults to localhost.  Only needed if web server is running
```

```
// on a different host than the appserver
props.setProperty("org.omg.CORBA.ORBInitialHost", "localhost");

// optional.  Defaults to 3700.  Only needed if target orb port is not 3700.
props.setProperty("org.omg.CORBA.ORBInitialPort", "3700");

InitialContext ic = new InitialContext(props);
```

Step 2. Use the global JNDI name of the target Remote EJB in the lookup.

EJB 3.x, assuming a global JNDI name of "com.acme.FooRemoteBusiness" :

```
FooRemoteBusiness foo = (FooRemoteBusiness) ic.lookup("com.acme.FooRemoteBusiness");
```

EJB 2.x, assuming a global JNDI name of "com.acme.FooHome" :

```
Object obj = ic.lookup("com.acme.FooHome");

FooHome fooHome = (FooHome) PortableRemoteObject.narrow(obj, FooHome.class);
```

Step 3. Add the necessary appserver code to the web server's classpath.

See step 3 of stand-alone client access for the list of required .jars.

Step 4. For EJB 3.0 Remote access, use at least Glassfish V2 or Java EE 5 SDK(SJS AS 9) Update 1.

Builds from this point on will contain a required bug fix.
See http://java.net/jira/browse/GLASSFISH-920 for more details.

What if I have multiple instances of the Appserver running and I want to access a Remote EJB component between them?

If the two server instances are part of the same cluster, there is no special mapping needed. By definition, a cluster is homogeneous, meaning the exact same set of applications is deployed to all instances. Each server instance has the same entries in its global JNDI namespace, so the target Remote EJB component can always be resolved intra-server. Therefore, it would never be a good idea to explicitly refer to a Remote EJB component in one server instance in the cluster from a different server instance in the same cluster.

However, if the server instances are either both stand-alone instances or this a cross-cluster access, the only difference is the syntax of the global JNDI name used to resolve the ejb-ref or @EJB. Since the calling code is running within a Java EE component (web or ejb), it should use the standard Java EE programming model for accessing remote EJB components -- @EJB or ejb-ref + java:comp/env lookup. The application code and any ejb-ref .xml should remain location transparent so that the mapping to physical EJB component can be changed without affecting any portable code or descriptors.

Follow the steps outlined here , but instead of using an unqualified global JNDI name, use the CORBA interoperable naming syntax :

```
corbaname:iiop:<host>:<port>#<global_jndi_name>
```

E.g., assume we have a web application running in a non-clustered app server instance on host1 that wants to access a Remote Stateless Session bean in an app server instance on host2. The target Remote EJB component has a global JNDI name of "Foo".

Within the servlet :

```
@EJB(name="fooejbref")
private FooRemote fooRemote;
```

Within sun-web.xml :

```
<ejb-ref>
  <ejb-ref-name>fooejbref</ejb-ref-name>
  <jndi-name>corbaname:iiop:host2:3700#Foo</jndi-name>
</ejb-ref>
```

Do I need RMI stubs to access EJBs from my java client?

No. Our implementation uses a feature called "Dynamic RMI-IIOP" that creates any necessary RMI-IIOP stubs at runtime in a way that is

completely hidden from the application. This makes deployment much faster and avoids many of the configuration problems that result from having to add static stubs to your client. However, Dynamic RMI-IIOP is only enabled when the client is using the naming provider from our server implementation. If the client is a stand-alone java client, that means following the instructions here or here , or using an Application Client component. Any explicit use of the `CosNaming` provider in the client will not be able to use the dynamic RMI-IIOP feature, which is why we don't recommend that approach. If you must use `CosNaming`, see here.

What if I have an existing stand-alone java client that accesses EJB components through the `CosNaming` `JNDI` provider ? How do I get static `RMI-IIOP` stubs for it?

First, read about Portable Java EE clients and "How To Write a Stand-alone Client". The best option would be to change the client to an Application Client or to follow our stand-alone java client recommendations. If that's not possible, you can request that static `RMI-IIOP` stubs be generated during deployment by using the `--generatermistubs` option on the `asadmin deploy` command. In this case, the static `RMI-IIOP` stubs will be placed within the client.jar file. E.g.

```
asadmin deploy --generatermistubs --retrieve /home/user1/clientstubdir  fooapp.ear
```

After the command executes, the client.jar containing the static `RMI-IIOP` stubs will be in `/home/user1/clientstubdir/fooappClient.jar`.

The reason static `RMI-IIOP` stubs are still needed in this case is that if the `CosNaming` provider is instantiated, it is not using the client naming provider from our appserver implementation. It instead uses an ORB from Java SE which does not support dynamic RMI-IIOP.

What is the relationship between `@EJB` and `ejb-ref`/`ejb-local-ref`?

The `@EJB` annotation and the `ejb-ref`/`ejb-local-ref` .xml elements are used to specify the same semantic information. Specifically, that a Java EE component has a dependency on a local or remote EJB component. Every `@EJB` can be translated into an equivalent `ejb-ref`/`ejb-local-ref`. `@EJB` is easier to use but it serves the same purpose as `ejb-ref`/`ejb-local-ref`. Here's a table with more details :

`@EJB` attribute	**Description**	**Default value**	**`ejb-ref`** equivalent	**`ejb-local-ref`** equivalent
`name`	Unique location within the private component namespace(`java:comp/env`).	field-level : `<fully-qualified name of declaring class>/<field-name>` method-level : `<fully-qualified name of declaring class>/<property name>` class level : `name` is required.	`ejb-ref-name`	`ejb-ref-name`
`beanInterface`	For EJB 3.x business interfaces, the Local or Remote business interface of the session bean. For EJB 2.x, the Home/LocalHome interface of the session/entity bean.	field-level : the type of the declared field. method-level : the type of the single setter parameter class level : `beanInterface` is required.	For EJB 3.x : `<remote>` For EJB 2.x : `<home>`	For EJB 3.x : `<local>` For EJB 2.x : `<local-home>`
`beanName`	`ejb-name` (**not** global `JNDI` name) of the target ejb component within the application. This can be used whenever the target ejb component is defined within the same application as the referencing component, regardless of local vs. remote. The only time it can't be used is if the `@EJB` refers to a Remote interface (3.x or 2.x) that is defined outside the application.	Automatically resolved if there is only one EJB component within the application that exposes the value of `beanInterface`	`ejb-link`	`ejb-link`

lookup ***(Added in EJB 3.1)***	Specifies the portable `JNDI` name of the target EJB component to which this `@EJB` dependency refers. This should be used instead of `mappedName` in cases where an `@EJB` dependency needs to be resolved to a Remote EJB component defined in a different application. It can also be used to chain one `@EJB` dependency to another `@EJB` dependency.	n/a	`lookup-name`	`lookup-name`
mappedName	Specifies the product-specific name of the target Remote EJB component. For GlassFish, this refers to the global `JNDI` name of the target Remote EJB component. **Not applicable for local interfaces because `beanName` can always be used instead.** ***(Should not be used in EJB 3.1. See `lookup` instead)**	If the target EJB component is defined within the same application and the `beanName` default applies, no additional mapping is required. Otherwise, the target global `JNDI` name will be set to the value of `beanInterface`	`mapped-name`	n/a

---

I have a Remote EJB dependency (@EJB or <ejb-ref>) in my Java EE component. How do I control which actual target EJB component it refers to?

If the target EJB component is defined within the same application as your referencing component *and* there is only one target EJB component within the application that exposes the remote interface associated with your EJB dependency, then the mapping will happen automatically. In this case, there is no need to specify any additional mapping information.

Otherwise, there are 3 ways to map the remote EJB dependency to the target Remote EJB component. From highest to lowest precedence, they are :

1. Use the sun-specific deployment descriptor

Map the remote EJB dependency to the global `JNDI` name of the target EJB component within the `sun-*.xml` {`sun-web.xml`, `sun-application-client.xml`, `sun-ejb-jar.xml`} file corresponding to the module in which the EJB dependency is defined.

E.g. given a remote EJB dependency defined within a servlet (`com.acme.MyServlet`) :

```
@EJB(name="fooejbref")
private FooRemote fooRemote;
```

To map this remote EJB dependency to an EJB comopnent with global `JNDI` name "`Foo`" :

Within `sun-web.xml` :

```
<ejb-ref>
  <ejb-ref-name>fooejbref</ejb-ref-name>
  <jndi-name>Foo</jndi-name>
</ejb-ref>
```

If the `@EJB` did not use the `name` attribute, you would need to refer to the default name of the EJB dependency when specifying the `sun-*.xml` mapping. E.g. :

```
@EJB
private FooRemote fooRemote;
```

To map this remote EJB dependency to an EJB component with global `JNDI` name "`Foo`" :

Within `sun-web.xml` :

```
<ejb-ref>
  <ejb-ref-name>com.acme.MyServlet/fooRemote</ejb-ref-name>
  <jndi-name>Foo</jndi-name>
</ejb-ref>
```

2. Use `@EJB` `mappedName` or the `<ejb-ref>..<mapped-name>` element.

For `@EJB`, specify the global `JNDI` name of the target EJB component using the `mappedName` attribute. E.g.

```
@EJB(name="fooejbref", mappedName="Foo")
```

```
private FooRemote fooRemote;
```

Note that mappedName is **completely different** than name. name refers to a location within the private component environment namespace (java:comp/env) that represents this EJB dependency. mappedName refers to a location within a product-specific *global* namespace that holds the EJB reference object.

For <ejb-ref>, specify the global JNDI name of the target EJB component using the <mapped-name> element. E.g.

```
<ejb-ref>
  <ejb-ref-name>fooejbref</ejb-ref-name>
  <remote>com.acme.FooRemote</remote>
  <mapped-name>Foo</mapped-name>
</ejb-ref>
```

If an @EJB and an <ejb-ref> are defined for the same EJB dependency(meaning @EJB.name equals <ejb-ref>..<ejb-ref-name>) in the same component environment, the values in <ejb-ref> take precedence.

3. Use @EJB beanName or <ejb-ref>..<ejb-link> element.

beanName and <ejb-link> only apply if the target remote EJB component is defined within the same application as the referencing component. In this case, the value is the ejb-name of the target bean, *not* the global JNDI name. If the target EJB component is defined using ejb-jar.xml, the ejb-name is the value of the <session> or <entity> ejb-name element. If the target EJB component is defined using annotations, the ejb-name is either the value the @Stateless/@Stateful/@Singleton name attribute or if name was not specified, the *unqualified* bean class name.

E.g. , assuming our target EJB component com.acme.FooBean is defined as follows :

```
@Stateless
public class FooBean implements FooRemote { ... }
```

To map the servlet's EJB dependency using beanName :

```
@EJB(name="fooejbref", beanName="FooBean")
private FooRemote fooRemote;
```

To map the servlet's EJB dependency using ejb-link :

```
<ejb-ref>
  <ejb-ref-name>fooejbref</ejb-ref-name>
  <remote>com.acme.FooRemote</remote>
  <ejb-link>FooBean</ejb-link>
</ejb-ref>
```

Note that ejb-name is only guaranteed to be unique within an ejb-jar or .war module. If there are multiple modules within the .ear that define an EJB component with the same ejb-name, the following beanName/<ejb-link> syntax can be used to explicitly refer to the target EJB component :

<relative ejb-jar module uri>#<ejb-name>

E.g., given an application containing ejb1.jar, ejb2.jar, and web1.war, where ejb1.jar and ejb2.jar both define an EJB component with ejb-name FooBean :

To map the servlet's EJB dependency to the EJB component in ejb1.jar using beanName :

```
@EJB(name="fooejbref", beanName="ejb1.jar#FooBean")
private FooRemote fooRemote;
```

To map the servlet's EJB dependency to the EJB component in ejb1.jar using ejb-link :

```
<ejb-ref>
  <ejb-ref-name>fooejbref</ejb-ref-name>
  <remote>com.acme.FooRemote</remote>
  <ejb-link>ejb1.jar#FooBean</ejb-link>
</ejb-ref>
```

If an @EJB and an <ejb-ref> are defined for the same EJB dependency(meaning @EJB.name equals <ejb-ref>..<ejb-ref-name>) in the same component environment, the values in <ejb-ref> take precedence.

I have an EJB component with a Local interface. Can I access it from an Application Client or a stand-alone java client ?

If the EJB component is running within the server, no. The EJB Local view is an optimized invocation path that uses call-by-reference semantics. It is only available to web components and EJB components that are part of the *same application* as the target EJB component.

To access EJB components that are running in the server from an Application Client or stand-alone java client, you'll need to use either a Remote 3.x Business interface, a 2.x Home interface, or web services.

One alternative, if using GlassFish v3, is to use the EJB 3.1 Embeddable API. This allows a Java SE program to directly execute EJB components within the same JVM, without using a server process.

I have an EJB component with a Local interface. Can I access it from a web component in a different application?

No. The EJB specification only requires access to an EJB component's local EJB interface from within the same application in the same JVM. One option is to package the `ejb-jar` in the same `.ear` as the `.war`. A second option, if using GlassFish v3, is to package the EJB component directly within the `.war`.

How do I access a Local EJB component from a POJO?

Local EJB component access is only portable from within the same application, so the POJO class must be called from a component running within the same application as the target Local EJB component. Injection is not supported for POJO classes, so the POJO needs to look up the local reference.

If the application is running within GlassFish v3, it can use the portable global `JNDI` names defined by the EJB 3.1 specification. In addition to portable `java:global` names, there are default portable `JNDI` names for two other scopes : `java:module`, and `java:app`. An EJB component's `java:module` `JNDI` name is visible to any code running within the same module. An EJB component's `java:app` `JNDI` name is visible to any code running within the same application.

The syntax for each is :

java:module/<bean-name>

java:app/<module-name>/<bean-name>

`<module-name>` defaults to the unqualified name of the `ejb-jar` file or `.war` file in which the EJB component is defined, minus the file extension. The `<module-name>` can be explicitly specified using the `<module-name>` element of the `ejb-jar.xml` (for ejb-jars) or `web.xml`(for EJB components defined in .wars).

`<bean-name>` corresponds to the session bean's `ejb-name`. It defaults to the *unqualified* name of the session bean class. It can be explicitly specified using the `name` attribute of the `@Stateless`/`@Stateful`/`@Singleton` annotation. Alternatively, if the `ejb-jar.xml` is being used to define the component, `<bean-name>` corresponds to the `<ejb-name>` element of `ejb-jar.xml`.

So, given :

```
@Stateless
public class FooBean { ... }
```

A POJO running within the same module as `FooBean` can lookup an EJB reference to `FooBean` using :

```
FooBean foo = (FooBean) new InitialContext().lookup("java:module/FooBean");
```

An advantage of the `java:module` syntax vs. `java:global` is that the code doing the lookup does not have to know the `<app-name>` or `<module-name>` in which it is executing.

If the application is running in GlassFish v2 or earlier, there are no portable `JNDI` names for the local EJB view. In that case the only way for the POJO to acquire an EJB reference is to define a local EJB dependency using `@EJB` or `ejb-local-ref`, and then look up that dependency within the private namespace (`java:comp/env`) of the component within whose scope it is executing. That can be done using the following steps :

Step 1. Define the local EJB dependency for the component within whose scope the POJO will execute.

Assume the POJO wants to access a local EJB 3.0 business interface `FooLocal` from the following EJB component defined within the same application :

```
@Stateless
public class FooBean implements FooLocal { ... }
```

If the POJO is called from a web application, the `java:comp/env` namespace is shared by the entire `.war`. So, the local EJB dependency can be defined by using an `@EJB` annotation on any managed class within the `.war` :

```
@EJB(name="fooejbref", beanInterface=FooLocal.class)
public class MyServlet extends HttpServlet { ... }
```

Likewise, if the POJO is itself called from an EJB component, the local EJB dependency must be defined within that EJB component. Unlike the `.war` case, the component environment (`java:comp/env`) is private for each EJB component.

```
@EJB(name="fooejbref", beanInterface=FooLocal.class)
@Stateless
public class BarBean implements BarLocal { ... }
```

Alternatively, the local EJB dependency can be declared within the standard deployment descriptor corresponding to the component within whose scope the POJO is executing (`web.xml` / `ejb-jar.xml`)

```
<ejb>
    <ejb-name>BarBean</ejb-name>
    <ejb-class>com.acme.BarBean</ejb-class>

     ...

      <ejb-local-ref>
        <ejb-ref-name>fooejbref</ejb-ref-name>
        <local>com.acme.FooLocal</local>
        <ejb-link>FooBean</ejb-link>
      </ejb-local-ref>

 </ejb>
```

Step 2. In the POJO code, lookup the local EJB dependency within `java:comp/env`

```
InitialContext ic = new InitialContext();

FooLocal foo = (FooLocal) ic.lookup("java:comp/env/fooejbref");
```

Note that if the POJO is called from multiple components, e.g. from both the web application and `BarBean`, you'll need to define the same local EJB dependency for both calling components so that it is always defined in the currently active component environment when the POJO does its lookup.

Are Global `JNDI` names portable? How do I know if my EJB component has a portable `JNDI` name?

If the EJB component is a kind of session bean and it is deployed to any implementation supporting EJB 3.1(E.g. GlassFish v3), it will automatically have one or more portable `JNDI` names defined based on the syntax in the EJB 3.1 specification. Note that this is true of existing EJB 3.0 and 2.x applications that are deployed to an implementation supporting EJB 3.1. *No* code changes are required on the bean class itself in order to have the portable global `JNDI` name automatically assigned when deployed to an EJB 3.1 container. See here for EJB 3.1 portable global `JNDI` name syntax.

Prior to EJB 3.1 (GlassFish v2 and earlier), there were no portable global `JNDI` names defined by the specification. See GlassFish-specific global `JNDI` name syntax.

What is the syntax for portable global `JNDI` names in EJB 3.1?

Per the EJB 3.1 Specification, each session bean is assigned a unique `JNDI` name within the `java:global` namespace. Note that prior to EJB 3.1 / GlassFish v3, there were no portable `JNDI` names defined for session beans, so each vendor defined its own syntax. See here for GlassFish vendor-specific global `JNDI` name syntax .

The syntax for portable global session bean `JNDI` names in EJB 3.1 is :

`java:global[/<app-name>]/<module-name>/<bean-name>`

The `<app-name>` portion is *only* present if the application is deployed as an `.ear` file. In that case, `<app-name>` defaults to the *unqualified* name of the `.ear` file, minus the file extension. The `<app-name>` can be explicitly specified using the `<app-name>` element in `application.xml`.

`<module-name>` defaults to the *unqualified* name of the `ejb-jar` file or `.war` file in which the EJB component is defined, minus the file extension. The `<module-name>` can be explicitly specified using the `<module-name>` element of the `ejb-jar.xml` (for ejb-jars) or `web.xml`(for EJB components defined in .wars).

`<bean-name>` corresponds to the session bean's EJB name. It defaults to the *unqualified* name of the session bean class. It can be explicitly specified using the `name` attribute of the `@Stateless`/`@Stateful`/`@Singleton` annotation. Alternatively, if the `ejb-jar.xml` is being used to define the component, `<bean-name>` corresponds to the `<ejb-name>` element of `ejb-jar.xml`.

Given the session bean :

```
@Stateless
public class FooBean implements Foo { ... }
```

If FooBean is packaged in fooejb.jar and deployed as a stand-alone module, its resulting JNDI name entry is :

java:global/fooejb/FooBean

If FooBean is packaged in fooejb.jar within fooapp.ear, its resulting global JNDI name entry is :

java:global/fooapp/fooejb/FooBean

If FooBean is packaged in a stand-alone web module fooweb.war, its resulting global JNDI name is :

java:global/fooweb/FooBean

If FooBean is packaged in fooweb.war within fooapp.ear, its resulting global JNDI name entry is :

java:global/fooapp/fooweb/FooBean

Also note that the portable java:global syntax is defined for a session bean with a local interface or no-interface view, not just for beans exposing a remote interface. However, *only* the names corresponding to *remote* session beans can be portably looked up from code running outside the application in which the target bean is defined.

Each portable global JNDI name is printed out to the server.log during deployment.

Finally, in the advanced case that a session bean exposes more than one client view, e.g. two local business interfaces, the above syntax is extended to include the fully-qualified name of the interface, with a "!" separator character. The syntax then becomes :

java:global[/<app-name>]/<module-name>/<bean-name>!<fully-qualified-interface-name>

If one of the multiple client views is a no-interface view, the last portion is the *fully-qualified* bean class name.

How do I define my own portable global JNDI name?

The EJB 3.1 portable global JNDI names defined by the specification are always registered by the container during deployment. However, it is possible to define one or more additional user-specified portable JNDI names for the bean using an EJB dependency(@EJB or <ejb-ref>/<ejb-local-ref>). EJB dependencies in Java EE 6 have been extended to allow the name attribute (or <ejb-ref-name> element) to specify a name outside of the component-specific(java:comp/env) scope. That allows a wider selection of code than the component itself to lookup the EJB dependency at a given name. The prefix of the string assigned to name can be ay one of the following :

- **"java:global/"** -- to expose a global name
- **"java:app/"** -- to expose a name only visible to code running within the same application
- **"java:module/"** -- to expose a name only visible to code running within the same module

For example, to define a Singleton with a user-specified global JNDI name of "java:global/MySingleton" the corresponding code would be :

```
@Singleton
@EJB(name="java:global/MySingleton", beanInterface=SharedRemote.class)
public class SharedBean implements SharedRemote { ... }
```

If this Singleton were deployed in a stand-alone ejb-jar called foo.jar, its default portable global JNDI name would be "java:global/foo/SharedBean". However, it would also be available for lookup at "java:global/MySingleton".

Also note that there is no requirement that the @EJB dependency be defined on the same bean class as the bean to which it refers. It could just as easily appear on some other component in the same or different application.

How are GlassFish-specific (non-portable) Global JNDI names assigned to Session beans?

If using GlassFish v3 (and therefore EJB 3.1) it is best to use the portable Global JNDI names defined by the EJB 3.1 specification, so see here instead. If you must use GlassFish-specific JNDI names, the syntax is as follows :

First, global JNDI names only apply to Session/Entity beans that have some kind of Remote interface. If the Session/Entity bean only has some combination of Local interfaces and Web Service interfaces, global JNDI names do **not** apply. In that case, any specified global JNDI name will be ignored.

In our *J2EE 1.4 implementation (Application Server 8.x)* , there is only one way to assign a global `JNDI` name to a session/entity bean's Remote view :

Map the bean's `<ejb-name>` to a `<jndi-name>` within the `<ejb>` element in `sun-ejb-jar.xml`. E.g.

```
<sun-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name>FooBean</ejb-name>
      <jndi-name>FooEJB</jndi-name>
    </ejb>
  </enterprise-beans>
 </sun-ejb-jar>
```

In our *Java EE 5 implementation (GlassFish V2 and earlier, Application Server 9.x)* , there are four ways to specify `JNDI` name for a session/entity bean. In decreasing order of precedence, they are :

1. Use `sun-ejb-jar.xml` (Same as above)

2. Specify the bean's global `JNDI` name using the `<mapped-name>` element in `ejb-jar.xml`

```
<enterprise-beans>
  <session>
    <ejb-name>FooBean</ejb-name>
    <mapped-name>FooEJB</mapped-name>
    ...
  </session>
</enterprise-beans>
```

3. Specify the bean's global `JNDI` name within the `@Stateless`/`@Stateful` `mappedName` attribute. E.g.,

```
@Stateless(mappedName="FooEJB")
public class FooBean implements Foo { ... }
```

4. If no global `JNDI` name has been specified, a default global `JNDI` name will be generated according to the following table :

Bean has 2.x Home/Remote interfaces	Total # of 3.0 Remote Business interfaces	default `JNDI` name	example
No	0	n/a	n/a
Yes	0	fully-qualified name of Home interface	com.acme.FooHome
No	1	fully-qualified name Remote Business interface	com.acme.FooBusiness
No	2 or more	No default -- `JNDI` name must be specified	n/a
Yes	1 or more	No default -- `JNDI` name must be specified	n/a

How do I specify the `Queue` or `Topic` that a Message Driven Bean should consume from?

In our *J2EE 1.4 implementation (Application Server 8.x) :*

Map the bean's `<ejb-name>` to the global `JNDI` name of the `Queue`/`Topic` JMS resource in `sun-ejb-jar.xml` :

```
<sun-ejb-jar>
  <enterprise-beans>
    <ejb>
      <ejb-name>FooMessageBean</ejb-name>
      <jndi-name>jms/NotificationQueue</jndi-name>
    </ejb>
  </enterprise-beans>
</sun-ejb-jar>
```

In our *Java EE 5 / 6 implementation (Application Server 9.x, GlassFish v2/v3)* , there are three ways to do it. In decreasing order of precedence, they are :

1. Use `sun-ejb-jar.xml` (Same as above)

2. Specify the global `JNDI` of the `Queue`/`Topic` JMS Resource using the `<mapped-name>` element in `ejb-jar.xml`

```
  <enterprise-beans>
    <message-driven>
      <ejb-name>FooMessageBean</ejb-name>
      <mapped-name>jms/NotificationQueue</mapped-name>
      ...
    </message-driven>
  </enterprise-beans>
```

3. Specify the global `JNDI` name of the `Queue`/`Topic` JMS Resource using the `@MessageDriven` `mappedName` attribute.

```
  @MessageDriven(mappedName="jms/NotificationQueue")
  public class FooMessageBean implements javax.jms.MessageListener { ... }
```

How do I "unit test" my EJB components?

The EJB 3.1 specification defines a new API called the EJB Embeddable API that allows a Java SE program to execute EJB components within the same JVM. This provides an easy way to invoke EJB components without needing a separate server process or an independent deployment step. To use it, you'll need GlassFish v3.

The Java SE main method is responsible for instantiating the EJB container via the `javax.ejb.embeddable.EJBContainer.createEJBContainer()` method. Once this method returns, all EJB components have been initialized. To lookup EJB references, use a special `JNDI` context returned from `EJBContainer.getContext()`, combined with the portable global `JNDI` names of the target EJB component. When the program is finished with all use of the EJB components, it should call `EJBContainer.close()` to shutdown and cleanup the embeddable EJB container instance. Here's an example :

```
  EJBContainer ejbC = EJBContainer.createEJBContainer();

  FooBean fooRef = (FooBean)
      ejbC.getNamingContext();.lookup("java:global/foo/FooBean");

  fooRef.foo();

  ejbC.close();
```

Executing the Java SE program that uses the EJB Embeddable API is simple. The java classpath needs to contain the following :

- `ejb-jar` containing the EJB components. Alternatively, this can be a directory containing a `META-INF/ejb-jar.xml` file or at least one `.class` with an enterprise bean component-defining annotation.
- Code for the main method and any supporting test classes.
- `$GLASSFISH_HOME/lib/embedded/glassfish-embedded-static-shell.jar`

For example, assume that `FooBean.class` is packaged in `foo.jar` and that the main class `acme.FooTest` is under `$HOME/test`. The corresponding `java` command is :

```
  java -classpath  foo.jar:$HOME/test:$GLASSFISH_HOME/lib/embedded/glassfish-
embedded-static-shell.jar
acme.FooTest
```

EJB - INTERVIEW QUESTIONS

Dear readers, these **EJB 3.0 Interview Questions** have been designed specially to get you acquainted with the nature of questions you may encounter during your interview for the subject of **EJB 3.0**. As per my experience good interviewers hardly plan to ask any particular question during your interview, normally questions start with some basic concept of the subject and later they continue based on further discussion and what you answer:

What is EJB?

EJB stands for Enterprise Java Beans. EJB is an essential part of a J2EE platform. J2EE platform have component based architecture to provide multi-tiered, distributed and highly transactional features to enterprise level applications.

EJB provides an architecture to develop and deploy component based enterprise applications considering robustness, high scalability and high performance. An EJB application can be deployed on any of the application server compliant with J2EE 1.3 standard specification.

What are the benefits of EJB?

Following are the key benefits of EJB.

- Simplified development of large scale enterprise level application.
- Application Server/ EJB container provides most of the system level services like transaction handling, logging, load balancing, persistence mechanism, exception handling and so on. Developer has to focus only on business logic of the application.
- EJB container manages life cycle of ejb instances thus developer needs not to worry about when to create/delete ejb objects.

What is a Session Bean in EJB?

Session bean stores data of a particular user for a single session. It can be stateful or stateless. It is less resource intensive as compared to entity beans. Session bean gets destroyed as soon as user session terminates.

What is Stateful Session Bean in EJB?

A stateful session bean is a type of enterprise bean which preserve the conversational state with client. A stateful session bean as per its name keeps associated client state in its instance variables. EJB Container creates a separate stateful session bean to process client's each request. As soon as request scope is over, statelful session bean is destroyed.

What is Stateless Session Bean in EJB?

A stateless session bean is a type of enterprise bean which is normally used to do independent operations. A stateless session bean as per its name does not have any associated client state, but it may preserve its instance state. EJB Container normally creates a pool of few stateless bean's objects and use these objects to process client's request. Because of pool, instance variable values are not guaranteed to be same across lookups/method calls.

What is Entity Bean in EJB?

Entity beans represents persistent data storage. User data can be saved to database via entity beans and later on can be retrived from the database in the entity bean.

What is Message Driven Bean in EJB?

A message driven bean is a type of enterprise bean which is invoked by EJB container when it receives a message from queue or topic. Message driven bean is a stateless bean and is used to do task asynchronously.

When a local session bean is used in EJB?

If ejb client is in same environment where ejb session bean is to be deployed then we use local session bean.

What a remote session bean is used in EJB?

if ejb client is in different environment where ejb session bean is to be deployed then we use remote session bean.

What are the differences between stateful session bean and stateless session bean?

Following are the differences between stateful session bean and stateless session bean:

- EJB Container creates a separate stateful session bean to process client's each request.whereas EJB Container normally creates a pool of few stateless bean's objects and use these objects to process client's request.
- As soon as request scope is over, statelful session bean is destroyed but stateless bean remains active.
- A stateful session bean is a type of enterprise bean which preserve the conversational state with client. A stateful session bean as per its name keeps associated client state in its instance variables Whereas because of pool of stateless session beans, instance variable values are not guaranteed to be same across lookups/method calls in stateless session beans.

What are the key components of persistence API in EJB?

Following are the key components of persistence API in EJB:

- **Entity** - A persistent object representing the data-store record. It is good to be serializable.
- **EntityManager** - Persistence interface to do data operations like add/delete/update/find on persistent object*entity*. It also helps to execute queries using Query interface.
- **Persistence unit** ***persistence. xml*** - Persistence unit describes the properties of persistence mechanism.
- **Data Source** ***∗ ds. xml*** - Data Source describes the data-store related properties like connection url. user-name,password etc.

Is Message Driven bean a stateless bean?

Message driven bean is a stateless bean and is used to do task asynchronously.

Explain @javax.ejb.Stateless annotation.

@javax.ejb.Stateless annotation specifies that a given ejb class is a stateless session bean.Following are its attributes:

- **name** - Used to specify name of the session bean.
- **mappedName** - Used to specify the JNDI name of the session bean.
- **description** - Used to provide description of the session bean.

Explain @javax.ejb.Stateful annotation.

@javax.ejb.Stateful annotation specifies that a given ejb class is a stateful session bean.Following are its attributes:

- **name** - Used to specify name of the session bean.
- **mappedName** - Used to specify the JNDI name of the session bean.
- **description** - Used to provide description of the session bean.

Explain @javax.ejb.MessageDrivenBean annotation.

@javax.ejb.MessageDrivenBean annotation specifies that a given ejb class is a message driven bean. Following are its attributes: name - Used to specify name of the message driven bean. messageListenerInterface - Used to specify message listener interface for the message driven bean. activationConfig - Used to specify the configuration details of the message-driven bean in operational environment of the message driven bean. mappedName - Used to specify the JNDI name of the message driven bean. description - Used to provide description of the message driven bean.

Explain @javax.ejb.EJB annotation.

@javax.ejb.EJB annotation is used to specify or inject a dependency as ejb instance into another ejb. Following are its attributes:

- **name** - Used to specify name which will be used to locate the referenced bean in environment.
- **beanInterface** - Used to specify the interface type of the referenced bean.
- **beanName** - Used to provide name of the referenced bean.
- **mappedName** - Used to specify the JNDI name of the referenced bean.
- **description** - Used to provide description of the referenced bean.

Explain @javax.ejb.Local annotation.

@javax.ejb.Local annotation is used to specify Local interface*s* of a session bean. This local interface states the business methods of the session bean *whichcanbestatelessorstateful*.

This interface is used to expose the business methods to local clients which are running in same deployment/application as EJB.

Following are its attributes:

- **value** - Used to specify the list of local interfaces as an array of interfaces.

Explain @javax.ejb.Remote annotation.

@javax.ejb.Remote annotation is used to specify Remote interface*s* of a session bean. This remote interface states the business methods of the session bean *whichcanbestatelessorstateful*.

This interface is used to expose the business methods to remote clients which are running in different deployment/application as EJB.

Following are its attributes:

- **value** - Used to specify the list of remote interfaces as an array of interfaces.

Explain @javax.ejb.ActivationConfigProperty annotation.

@javax.ejb.ActivationConfigProperty annotation is used to specify properties required for a message driven bean. For example end point, destination, message selector etc.

This annotation is passed as a parameter to activationConfig attribute of javax.ejb.MessageDrivenBean annotation. Following are its attributes:

- **propertyName** - name of the property.
- **propertyValue** - value of the property.

Explain @javax.ejb.PostActivate annotation.

@javax.ejb.PostActivate annotation is used to specify callback method of ejb lifecycle. This method will be called when EJB container just activated/reactivated the bean instance.

This interface is used to expose the business methods to local clients which are running in same deployment/application as EJB.

What is Callback in EJB?

Callback is a mechanism by which life cycle of an enterprise bean can be intercepted. EJB 3.0 specification has specified callbacks for which callback handler methods are to be created. EJB Container calls these callbacks. We can define callback methods in the ejb class itself or in a separate class. EJB 3.0 has provided many annotations for callbacks.

What are the callback annotations for stateless bean?

Following is the list of callback annotations for stateless bean:

- **@PostConstruct** - method is invoked when a bean is created for the first time.
- **@PreDestroy** - method is invoked when a bean is removed from the bean pool or is destroyed.

What are the callback annotations for stateful bean?

Following is the list of callback annotations for stateful bean:

- **@PostConstruct** - method is invoked when a bean is created for the first time.
- **@PreDestroy** - method is invoked when a bean is removed from the bean pool or is destroyed.
- **@PostActivate** - method is invoked when a bean is loaded to be used.
- **@PrePassivate** - method is invoked when a bean is put back to bean pool.

What are the callback annotations for message driven bean?

Following is the list of callback annotations for message driven bean:

- **@PostConstruct** - method is invoked when a bean is created for the first time.
- **@PreDestroy** - method is invoked when a bean is removed from the bean pool or is destroyed.

What are the callback annotations for entity bean?

Following is the list of callback annotations for entity bean:

- **@PrePersist** - method is invoked when an entity is created in database.
- **@PostPersist** - method is invoked after an entity is created in database.
- **@PreRemove** - method is invoked when an entity is deleted from the database.
- **@PostRemove** - method is invoked after an entity is deleted from the database.
- **@PreUpdate** - method is invoked before an entity is to be updated in the database.
- **@PostLoad** - method is invoked when a record is fetched from database and loaded into the entity.

What is a timer service in EJB?

Timer Service is a mechanism using which scheduled application can be build. For example, salary slip generation on 1st of every month. EJB 3.0 specification has specified @Timeout annotation which helps in programming the ejb service in a stateless or message driven bean. EJB Container calls the method which is annotated by @Timeout.

EJB Timer Service is a service provided by Ejb container which helps to create timer and to schedule callback when timer expires.

Which annoatation is used to inject an ejb into another ejb?

@EJB annotation is used to inject other EJB reference.

Which annoatation is used to inject an datasource into an ejb?

@Resource annotation is used to inject an datasource into an ejb.

Which annoatation is used to inject singleton services like timer service into an ejb?

@Resource annotation is used to inject singleton services like timer service into an ejb.

Which annoatation is used to inject SessionContext into an ejb?

@Resource annotation is used to inject SessionContext into an ejb.

How EJB implements dependency injection?

EJB 3.0 specification provides annotations which can be applied on fields or setter methods to inject dependencies. EJB Container uses the global JNDI registry to locate the dependency.

What is an EJB Interceptor?

EJB 3.0 provides specification to intercept business methods calls using methods annotated with @AroundInvoke annotation. An interceptor method is called by ejbContainer before business method call it is intercepting.

What is class level interceptor in EJB?

Class level interceptor is invoked for every method of the bean. Class level interceptor can be applied both by annotation or via xml$ejb-jar.xml$.

What is default interceptor in EJB?

Default interceptor is invoked for every bean within deployment.Default interceptor can be applied only via xml $ejb-jar.xml$.

What is method level interceptor in EJB?

Method level interceptor is invoked for a particular method of the bean. Method level interceptor can be applied both by annotation of via xml$ejb-jar.xml$.

Explain @Embeddable annotation.

EJB 3.0 provides option to embed JAVA POJO $PlainOldJavaObject$ into an entity bean and allows to map column names with the methods of the embedded POJO class. A java POJO to be embedded must be annotated as @Embeddable.

Explain @Lob annotation.

EJB 3.0 provides support for Blob and Clob types using @Lob annotation.

Which types of java classes can be mapped using @Lob annotation?

Following java types can be mapped using @Lob annotation:

- java.sql.Blob
- java.sql.Clob
- byte[]
- String
- Serializable Object

What is a transaction?

A transaction is a single unit of work items which follows the ACID properties. ACID stands for Atomic, Consistent,Isolated and Durable.

What ACID stands for?

ACID stands for Atomic, Consistent,Isolated and Durable.

Explain ACID in context of transaction management.

- **Atomic** - If any of work item fails, the complete unit is considered failed. Success meant all items executes successfully.
- **Consistent** - A transaction must keep the system in consistent state.
- **Isolated** - Each transaction executes independent of any other transaction.
- **Durable** - Transaction should survive system failure if it has been executed or committed.

What is Container Managed Transactions?

In this type, container manages the transaction states.

What is Bean Managed Transactions?

In this type, developer manages the life cycle of transaction states.

Which are the attributes of Container Managed Transactions?

EJB 3.0 has specified following attributes of transactions which EJB containers implement:

- **REQUIRED** - Indicates that business method has to be executed within transaction otherwise a new transaction will be started for that method.
- **REQUIRES_NEW** - Indicates that a new transaction is to be started for the business method.
- **SUPPORTS** - Indicates that business method will execute as part of transaction.
- **NOT_SUPPORTED** - Indicates that business method should not be executed as part of transaction.
- **MANDATORY** - Indicates that business method will execute as part of transaction otherwise exception will be thrown.
- **NEVER** - Indicates if business method executes as part of transaction then an exception will be thrown.

Which are the attributes of Bean Managed Transactions?

In Bean Managed Transactions, Transactions can be managed by handling exceptions at application level. Following are the key points to be considered:

- **Start** - When to start a transaction in a business method.
- **Sucess** - Identify success scenario when a transaction is to be committed.
- **Failed** - Identify failure scenario when a transaction is to be rollback.

Which are the attributes of Container Managed Security?

EJB 3.0 has specified following attributes/annotations of security which EJB containers implement:

- **DeclareRoles** - Indicates that class will accept those declared roles. Annotations are applied at class level.
- **RolesAllowed** - Indicates that a method can be accessed by user of role specified. Can be applied at class level resulting which all methods of class can be accessed buy user of role specified.
- **PermitAll** - Indicates that business method is accessible to all. Can be applied at class as well as at method level.
- **DenyAll** - Indicates that business method is not accessible to any of user specified at class or at method level.

What is JNDI? Explain its terms in terms of EJB.

JNDI stands for Java Naming and Directory Interface. It is a set of API and service interfaces. Java based applications use JNDI for naming and directory services. In context of EJB, there are two terms:

- **Binding** - This refers to assigning a name to an ejb object which can be used later.
- **Lookup** - This refers to looking up and getting an object of ejb.

Explain annotation in EJB to do the database entity relationships/mappings.

EJB 3.0 provides option to define database entity relationships/mappings like one to one, one to many, many to one and many to many relationships. Following are the relevant annotations:

- **OneToOne** - Objects are having one to one relationship. For example, a passenger can travel using a single ticket at time.
- **OneToMany** - Objects are having one to many relationship. For example, a father can have multiple kids.
- **ManyToOne** - Objects are having many to one relationship. For examples, multiple kids having a single mother.
- **ManyToMany** - Objects are having many to many relationship. For examples, a book can have mutiple authors and a author can write multiple books.

What is EJBQL?

EJB 3.0, ejb query language is quite handy to write custom queries without worrying about underlying database details. It is quite similar to HQL, hibernate query language and is often referred by name EJBQL.

What is Application level Exception in EJB?

If business rule is voilated or exception occurs while executing the business logic. Then EJB container treats it as Application level exception.

What is System level Exception in EJB?

Any exception which is not caused by business logic or business code. RuntimeException, RemoteException are SystemException. For example, error during ejb lookup.Then EJB container treats such exception as System level exception.

How EJB Container handles exceptions?

When Application Exception occurs, ejb container intercepts the exception but returns the same to the client as it is. It does not roll back the transaction unless it is specified in code by EJBContext.setRollBackOnly method. EJB Container does not wrap the exception in case of Application Exception.

When System Exception occurs, ejb container intercepts the exception, rollbacks the transaction and start the clean up tasks. It wraps the exception into RemoteException and throws it to the client.

What is Next ?

Further you can go through your past assignments you have done with the subject and make sure you are able to speak confidently on them. If you are fresher then interviewer does not expect you will answer very complex questions, rather you have to make your basics concepts very strong.

Second it really doesn't matter much if you could not answer few questions but it matters that whatever you answered, you must have answered with confidence. So just feel confident during your interview. We at tutorialspoint wish you best luck to have a good interviewer and all the very best for your future endeavor. Cheers :-)

```
      try{
         userTransaction.begin();

         confirmAccountDetail(fromAccount);
         withdrawAmount(fromAccount,fund);

         confirmAccountDetail(toAccount);
         depositAmount(toAccount,fund);

         userTransaction.commit();
      }catch (InvalidAccountException exception){
         userTransaction.rollback();
      }catch (InsufficientFundException exception){
         userTransaction.rollback();
      }catch (PaymentException exception){
         userTransaction.rollback();
      }
   }

   private void confirmAccountDetail(Account account)
      throws InvalidAccountException {
   }

   private void withdrawAmount() throws InsufficientFundException {
   }

   private void depositAmount() throws PaymentException{
   }
}
```

In this example, we made use of **UserTransaction** interface to mark beginning of transaction using **userTransaction.begin()** method call. We mark completion of transaction by using **userTransaction.commit()** method and if any exception occured during transaction then we rollback the complete transaction using **userTransaction.rollback()** method call.

EJB - SECURITY

Security is a major concern of any enterprise level application. It includes identification of user(s) or system accessing the application and allowing or denying the access to resources within the application. In EJB, security can be declared in declarative way called declarative security in which EJB container manages the security concerns or Custom code can be done in EJB to handle security concern by self.

Important Terms of Security

- **Authentication** - This is the process ensuring that user accessing the system or application is verified to be authentic.
- **Authorization** - This is the process ensuring that authentic user has right level of authority to access system resources.
- **User** - User represents the client or system accessing the application.
- **User Groups** - Users may be part of group having certain authorities for example administrator's group.
- **User Roles** - Roles defines the level of authority a user have or permissions to access a system resource.

Container Managed Security

EJB 3.0 has specified following attributes/annotations of security which EJB containers implement.

- **DeclareRoles** - Indicates that class will accept those declared roles. Annotations are applied at class level.

- **RolesAllowed** - Indicates that a method can be accessed by user of role specified. Can be applied at class level resulting which all methods of class can be accessed buy user of role specified.
- **PermitAll** - Indicates that business method is accessible to all. Can be applied at class as well as at method level.
- **DenyAll** - Indicates that business method is not accessible to any of user specified at class or at method level.

Example

```
package com.tutorialspoint.security.required;

import javax.ejb.*

@Stateless
@DeclareRoles({"student" "librarian"})
public class LibraryBean implements LibraryRemote {

   @RolesAllowed({"librarian"})
   public void delete(Book book){
   //delete book
   }

   @PermitAll
   public void viewBook(Book book){
      //view book
   }

   @DenyAll
   public void deleteAll(){
      //delete all books
   }
}
```

Security Configuration

Map roles and user groupd in configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Application Server 9.0 EJB
3.0//EN" "http://www.sun.com/software/appserver/dtds/sun-ejb-jar_3_0-0.dtd">
<ejb-jar>
   <security-role-mapping>
      <role-name>student</role-name>
      <group-name>student-group</group-name>
   </security-role-mapping>
   <security-role-mapping>
      <role-name>librarian</role-name>
      <group-name>librarian-group</group-name>
   </security-role-mapping>
   <enterprise-beans/>
</ejb-jar>
```

EJB - JNDI BINDINGS

JNDI stands for Java Naming and Directory Interface. It is a set of API and service interfaces. Java based applications use JNDI for naming and directory services. In context of EJB, there are two terms.

- **Binding** - This refers to assigning a name to an ejb object which can be used later.
- **Lookup** - This refers to looking up and getting an object of ejb.

In Jboss, session beans are bound in JNDI in following format by default.

- **local** - ejb-name/local
- **remote** - ejb-name/remote

In case, ejb are bundled with <application-name>.ear file then default format is as following.

- **local** - application-name/ejb-name/local
- **remote** - application-name/ejb-name/remote

Example of default binding

Refer to *EJB - Create Application* chapter's JBoss console output.

JBoss Application server log output

```
...
16:30:02,723 INFO  [SessionSpecContainer] Starting
jboss.j2ee:jar=EjbComponent.jar,name=LibrarySessionBean,service=EJB3
16:30:02,723 INFO  [EJBContainer] STARTED EJB:
com.tutorialspoint.stateless.LibrarySessionBean ejbName: LibrarySessionBean
16:30:02,731 INFO  [JndiSessionRegistrarBase] Binding the following Entries in Global
JNDI:

   LibrarySessionBean/remote - EJB3.x Default Remote Business Interface
   LibrarySessionBean/remote-com.tutorialspoint.stateless.LibrarySessionBeanRemote -
EJB3.x Remote Business Interface
...
```

Customized binding

Following annotations can be used to customized the default JNDI bindings.

- **local** - org.jboss.ejb3.LocalBinding
- **remote** - org.jboss.ejb3.RemoteBindings

Update LibrarySessionBean.java. Refer to *EJB - Create Application* chapter

LibrarySessionBean

```
package com.tutorialspoint.stateless;

import java.util.ArrayList;
import java.util.List;
import javax.ejb.Stateless;

@Stateless
@LocalBinding(jndiBinding="tutorialsPoint/librarySession")
public class LibrarySessionBean implements LibrarySessionBeanLocal {

    List<String> bookShelf;

    public LibrarySessionBean(){
       bookShelf = new ArrayList<String>();
    }

    public void addBook(String bookName) {
       bookShelf.add(bookName);
    }

    public List<String> getBooks() {
        return bookShelf;
    }
}
```

LibrarySessionBeanLocal

```
package com.tutorialspoint.stateless;

import java.util.List;
import javax.ejb.Local;

@Local
public interface LibrarySessionBeanLocal {

    void addBook(String bookName);

    List getBooks();

}
```

Build the project. Deploy the application on Jboss and verify the following output in Jboss console.

```
...
16:30:02,723 INFO  [SessionSpecContainer] Starting
jboss.j2ee:jar=EjbComponent.jar,name=LibrarySessionBean,service=EJB3
16:30:02,723 INFO  [EJBContainer] STARTED EJB:
com.tutorialspoint.stateless.LibrarySessionBean ejbName: LibrarySessionBean
16:30:02,731 INFO  [JndiSessionRegistrarBase] Binding the following Entries in Global
JNDI:

   tutorialsPoint/librarySession - EJB3.x Default Local Business Interface
   tutorialsPoint/librarySession-com.tutorialspoint.stateless.LibrarySessionBeanLocal -
EJB3.x Local Business Interface
...
```

Repeat the above steps for Remote and check the result.

EJB - ENTITY RELATIONSHIPS

EJB 3.0 provides option to define database entity relationships/mappings like one to one, one to many, many to one and many to many relationships. Following are the relevant annotations.

- **OneToOne** - Objects are having one to one relationship. For example, a passenger can travel using a single ticket at time.

  li>

  OneToMany - Objects are having one to many relationship. For example, a father can have multiple kids.

- **ManyToOne** - Objects are having many to one relationship. For examples, multiple kids having a single mother.

- **ManyToMany** - Objects are having many to many relationship. For examples, a book can have mutiple authors and a author can write multiple books.

We'll demonstrate use of ManyToMany mapping here. To represent ManyToMany relationship, three tables are required.

- **Book** - Book table having records of books
- **Author** - Author table having records of author
- **Book_Author** - Book_Author table having linkage of above mentioned Book and Author table.

Create tables

Create a table **book author**, **book_author** in default database **postgres**.

```
CREATE TABLE book (
```

```
   book_id     integer,
   name   varchar(50)
);
```

```
CREATE TABLE author (
   author_id   integer,
   name   varchar(50)
);
```

```
CREATE TABLE book_author (
   book_id     integer,
   author_id   integer
);
```

Create Entity Classes

```
@Entity
@Table(name="author")
public class Author implements Serializable{
   private int id;
   private String name;
   ...
}
```

```
@Entity
@Table(name="book")
public class Book implements Serializable{
   private int id;
   private String title;
   private Set<Author> authors;
   ...
}
```

Use ManyToMany annotation in Book Entity

```
@Entity
public class Book implements Serializable{
   ...
   @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE}
      , fetch = FetchType.EAGER)
   @JoinTable(table = @Table(name = "book_author"),
      joinColumns = {@JoinColumn(name = "book_id")},
      inverseJoinColumns = {@JoinColumn(name = "author_id")})
   public Set<Author> getAuthors()
   {
      return authors;
   }
   ...
}
```

Example Application

Let us create a test EJB application to test entity relationships objects in EJB 3.0.

Step	Description
1	Create a project with a name *EjbComponent* under a package *com.tutorialspoint.entity* as explained in the *EJB - Create Application* chapter. Please use the project created in *EJB - Persistence* chapter as such for this chapter to understand embedded objects in ejb concepts.
2	Create *Author.java* under package *com.tutorialspoint.entity* as explained in the *EJB - Create Application* chapter. Keep rest of the files unchanged.

```
package com.tutorialspoint.entity;


import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Table;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;

@Entity
@Table(name="book")
public class Book implements Serializable{

   private int id;
   private String name;
   private Set<Author> authors;

   public Book(){
   }

   @Id
   @GeneratedValue(strategy= GenerationType.IDENTITY)
   @Column(name="book_id")
   public int getId() {
      return id;
   }

   public void setId(int id) {
      this.id = id;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public void setAuthors(Set<Author> authors) {
      this.authors = authors;
   }

   @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE}
      , fetch = FetchType.EAGER)
   @JoinTable(table = @Table(name = "book_author"),
      joinColumns = {@JoinColumn(name = "book_id")},
      inverseJoinColumns = {@JoinColumn(name = "author_id")})
   public Set<Author> getAuthors()
   {
      return authors;
   }
}
```

LibraryPersistentBeanRemote.java

```
package com.tutorialspoint.stateless;

import com.tutorialspoint.entity.Book;
import java.util.List;
import javax.ejb.Remote;

@Remote
public interface LibraryPersistentBeanRemote {

   void addBook(Book bookName);
```

```
   List<Book> getBooks();

}
```

LibraryPersistentBean.java

```
package com.tutorialspoint.stateless;

import com.tutorialspoint.entity.Book;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

@Stateless
public class LibraryPersistentBean implements LibraryPersistentBeanRemote {

   public LibraryPersistentBean(){
   }

   @PersistenceContext(unitName="EjbComponentPU")
   private EntityManager entityManager;

   public void addBook(Book book) {
      entityManager.persist(book);
   }

   public List<Book> getBooks() {
      return entityManager.createQuery("From Book").getResultList();
   }
}
```

- As soon as you deploy the EjbComponent project on JBOSS, notice the jboss log.
- JBoss has automatically created a JNDI entry for our session bean - **LibraryPersistentBean/remote**.
- We'll using this lookup string to get remote business object of type - **com.tutorialspoint.interceptor.LibraryPersistentBeanRemote**

JBoss Application server log output

```
...
16:30:01,401 INFO  [JndiSessionRegistrarBase] Binding the following Entries in Global
JNDI:
   LibraryPersistentBean/remote - EJB3.x Default Remote Business Interface
   LibraryPersistentBean/remote-
com.tutorialspoint.interceptor.LibraryPersistentBeanRemote - EJB3.x Remote Business
Interface
16:30:02,723 INFO  [SessionSpecContainer] Starting
jboss.j2ee:jar=EjbComponent.jar,name=LibraryPersistentBean,service=EJB3
16:30:02,723 INFO  [EJBContainer] STARTED EJB:
com.tutorialspoint.interceptor.LibraryPersistentBeanRemote ejbName: LibraryPersistentBean
16:30:02,731 INFO  [JndiSessionRegistrarBase] Binding the following Entries in Global
JNDI:

   LibraryPersistentBean/remote - EJB3.x Default Remote Business Interface
   LibraryPersistentBean/remote-
com.tutorialspoint.interceptor.LibraryPersistentBeanRemote - EJB3.x Remote Business
Interface
...
```

EJBTester (EJB Client)

jndi.properties

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jboss.naming:org.jnp.interfaces
java.naming.provider.url=localhost
```

- These properties are used to initialize the InitialContext object of java naming service
- InitialContext object will be used to lookup stateless session bean

EJBTester.java

```
package com.tutorialspoint.test;

import com.tutorialspoint.stateful.LibraryBeanRemote;
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class EJBTester {

   BufferedReader brConsoleReader = null;
   Properties props;
   InitialContext ctx;
   {
      props = new Properties();
      try {
         props.load(new FileInputStream("jndi.properties"));
      } catch (IOException ex) {
         ex.printStackTrace();
      }
      try {
         ctx = new InitialContext(props);
      } catch (NamingException ex) {
         ex.printStackTrace();
      }
      brConsoleReader =
      new BufferedReader(new InputStreamReader(System.in));
   }

   public static void main(String[] args) {

      EJBTester ejbTester = new EJBTester();

      ejbTester.testEmbeddedObjects();
   }

   private void showGUI(){
      System.out.println("**********************");
      System.out.println("Welcome to Book Store");
      System.out.println("**********************");
      System.out.print("Options \n1. Add Book\n2. Exit \nEnter Choice: ");
   }

   private void testEmbeddedObjects(){

      try {
         int choice = 1;

         LibraryPersistentBeanRemote libraryBean =
         (LibraryPersistentBeanRemote)
         ctx.lookup("LibraryPersistentBean/remote");

         while (choice != 2) {
            String bookName;
            String authorName;
```

```
            showGUI();
            String strChoice = brConsoleReader.readLine();
            choice = Integer.parseInt(strChoice);
            if (choice == 1) {
               System.out.print("Enter book name: ");
               bookName = brConsoleReader.readLine();
               System.out.print("Enter author name: ");
               authorName = brConsoleReader.readLine();
               Book book = new Book();
               book.setName(bookName);
Author author = new Author();
author.setName(authorName);
Set<Author> authors = new HashSet<Author>();
authors.add(author);
               book.setAuthors(authors);

               libraryBean.addBook(book);
            } else if (choice == 2) {
               break;
            }
         }

         List<Book> booksList = libraryBean.getBooks();

            System.out.println("Book(s) entered so far: " + booksList.size());
            int i = 0;
            for (Book book:booksList) {
               System.out.println((i+1)+". " + book.getName());
               System.out.print("Author: ");
               Author[] authors = (Author[])books.getAuthors().toArray();
               for(int j=0;j<authors.length;j++){
                  System.out.println(authors[j]);
               }
               i++;
            }
         } catch (Exception e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
         }finally {
            try {
               if(brConsoleReader !=null){
                  brConsoleReader.close();
               }
            } catch (IOException ex) {
               System.out.println(ex.getMessage());
            }
         }
      }
}
```

EJBTester is doing the following tasks.

- Load properties from jndi.properties and initialize the InitialContext object.
- In testInterceptedEjb() method, jndi lookup is done with name - "LibraryPersistenceBean/remote" to obtain the remote business object (stateless ejb).
- Then user is shown a library store User Interface and he/she is asked to enter choice.
- If user enters 1, system asks for book name and saves the book using stateless session bean addBook() method. Session Bean is storing the book in database.
- If user enters 2, system retrieves books using stateless session bean getBooks() method and exits.

Run Client to access EJB