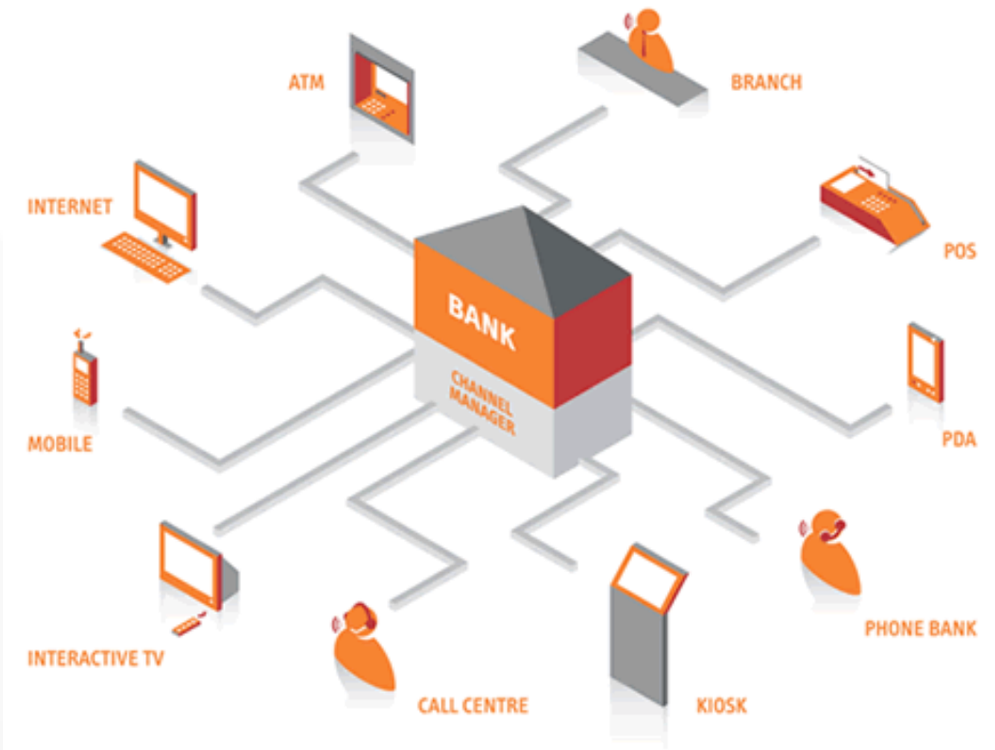
JavaEE és EJB bevezető

Simon Károly
simon.karoly@codespring.ro

Osztott rendszerek



Java EE

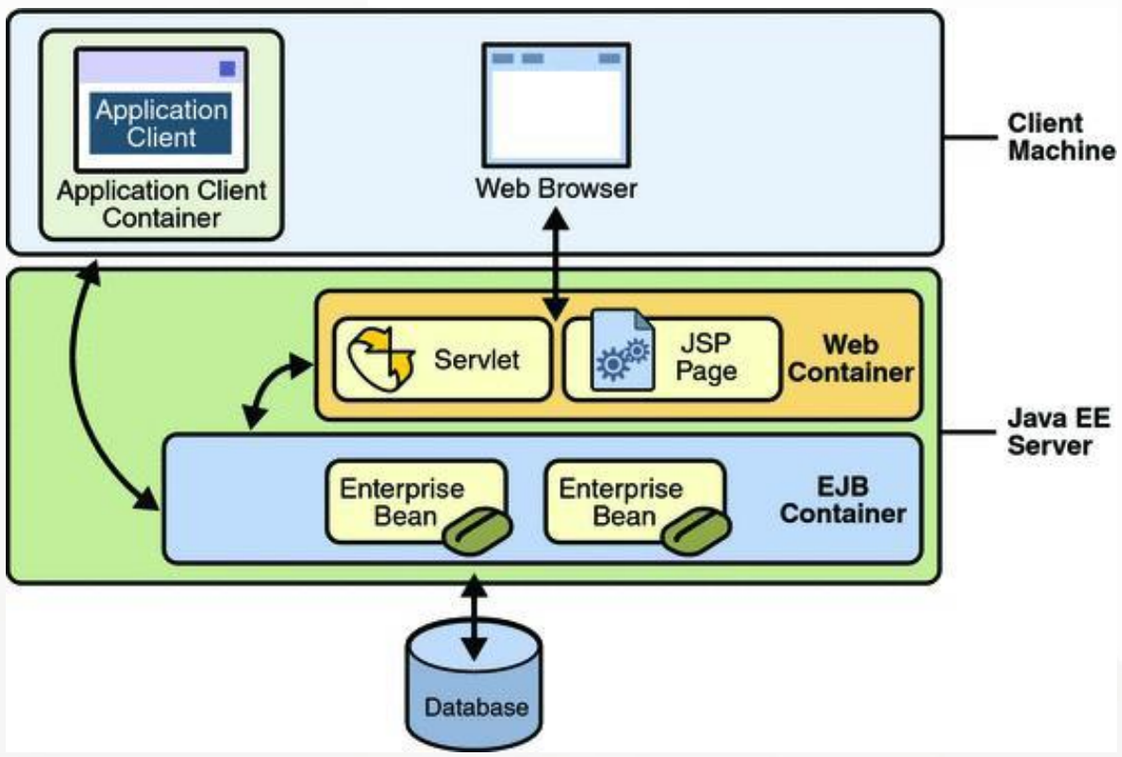
- Java Platformok: Java Card, Java ME, Java SE, **Java EE**, Java FX
- **Java Enterprise Editon**: komponens alapú fejlesztések, osztott vállalati rendszerek
- Fontosabb csomagok, API-k:
  - **Enterprise JavaBean (EJB) – javax.ejb**
  - Java Servlet API – javax.servlet
  - Java Server Pages (JSP)– javax.servlet.jsp
  - JSP – Standard Tag Library (JSTL) – javax.servlet.jsp.jstl
  - JavaServer Faces (JSF) – javax.faces
  - Java Message Service API (JMS) – javax.jms
  - Java Persistence API (JPA) – javax.persistence
  - Java Transaction API (JTA) – javax.transaction
  - Stb. stb.

Alkalmazásszerverek

- Alkalmazásszerverek:
  - Programok hatékony futtatása, a fejlesztési folyamat egyszerűsítése, támogatása
  - Szolgáltatások: tranzakció kezelés, biztonsági megoldások, központosított konfiguráció stb.
  - Továbbá: feladatok szétosztása, párhuzamos végrehajtás (clustering), meghibásodott modulok automatikus helyettesítése (fail-over), terhelés elosztása (load-balancing)
    → a fejlesztő az alkalmazás-logikára (business logic) koncentrálhat
- Java alkalmazásszerverek
  - **Glassfish** (Sun, open source), Oracle Glassfish Server
  - JBoss (JBoss/Red Hat)
  - Oracle Weblogic, IBM Websphere, Apache Geronimo, SAP Netweaver, JOnAS (open source), stb., stb.
- Java web-szerverek: alkalmazásszerverek (pl. Glassfish → Grizzly), **Apache Tomcat**, Jetty, stb.

Többrétegű web alkalmazások

- Kliens réteg: a kliens gépén futó komponensek
- Alkalmazás réteg:
  - Web réteg: a Java EE szerveren futó webes komponensek, a szerver web-konténerben futnak (Servlet, JSP)
  - Business réteg: a Java EE szerveren futó üzleti logikát megvalósító komponensek, a szerver EJB konténerében futnak (EJB)
- Adathozzáférési réteg: Enterprise Information System (EIS) szerver



Java EE alkalmazás részei

- Webes kliens, vagy asztali kliensalkalmazás (kliens oldal)
  - A web-kliens a szervertől érkező oldalakat mutatja meg (HTML, XML, stb. alapú dinamikus weboldalak, amelyeket a szerveroldali komponensek generálnak) (böngésző/browser)
- Java EE komponensek (szerver oldal): önálló funkcionalitásokat biztosító, egymással kommunikáló szoftverkomponensek. Java osztályok és erőforrás állományok, amelyek összerakásuk (assembly) után egy alkalmazás-szerverre lesznek telepítve (deployment). Meg kell feleljenek a vonatkozó Java EE specifikációnak.
  - Web komponensek (Servlet, JSP): a Java EE szerver web-konténerében (pl. Tomcat) futnak
  - EJB (Enterprise JavaBeans): szerveroldali, üzleti logikáért felelős komponensek (entity beans, session beans, message-driven beans), az alkalmazásszerver EJB konténerében futnak
- Összeállítás: különböző konténer beállítások, konfigurációs állományokban → a szerver ezek alapján biztosítja szolgáltatásait (tranzakció-kezelés, biztonság, stb.)

Enterprise Java Beans

EJB - Általános bevezető

EJB

- Sun definíció:
  - The Enterprise JavaBeans architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and deployed on any server platform that supports the Enterprise JavaBeans specification.
- Bill Burke és Richard Monson-Haefel egyszerűsített definíciója:
  - Enterprise JavaBeans is a standard server-side component model for distributed business applications.
- EJB komponensek:
  - **Entity beans** – a rendszer központi entitásainak reprezentációja, adatrögzítés, perzisztencia (JPA)
  - **Session beans** – az üzleti logika megvalósítása
  - **Message-driven beans** – aszinkron üzenetek, a komponensek közötti kommunikáció (JMS)
- A komponensmodell alapján meghatározható a szerver-oldali komponensek viselkedése (tranzakciók, biztonság, perzisztencia).

JPA és Entity Beans

- Objektumok állapotának tárolása adatbázis rendszerekben, a JDBC feletti (arra épülő) absztrakciós szint
- Specifikáció: Java Persistence API (JPA) (az EJB 3.0-tól különálló)
- JPA: standard eljárás definiálása a POJOk adatbázisba történő leképezésének
- Entitiy Beans – POJOk, amelyek a JPA meta-adatok (annotációs mechanizmus) segítségével le lesznek képezve egy adatbázisba. Az adatok mentése, betöltése, módosítása megtörténhet anélkül, hogy a fejlesztőnek ezzel kapcsolatos kódot kelljen írnia (pl. JDBC hozzáféréssel kapcsolatos kód nem szükséges)
- A JPA meghatároz egy lekérdező nyelvet is (a funkcionalitások azonosak az SQL nyelvek által biztosított funkcionalitásokkal, de Java objektumokkal dolgozhatunk, az objektumorientált szemléletmódnak megfelelően)
- Az új JPA specifikáció meghatároz egy teljes ORM leképezést, a komponensek hordozhatóak (a mechanizmus már nem függ gyártótól, vagy alkalmazásszerver típustól), sőt, hagyományos Java alkalmazásokon belül is használhatóak.

JMS és Message-driven Beans

- Aszinkron üzenetek: az EJB az RMI mechanizmuson kívül az aszinkron üzenetküldést is támogatja. Az alkalmazások üzeneteken keresztül kommunikálhatnak. Az üzenetek üzleti adatokat és hálózattal kapcsolatos (routing) információkat tartalmaznak.
- MOM (message-oriented middleware) rendszerek alkalmazása: megfelelő üzenetkezelés, hiba-tolerancia (fault-tolerance), terhelés elosztás (load-balancing), skálázhatósággal és tranzakció-kezeléssel kapcsolatos szolgáltatások.
- Címzettek: virtuális csatornák alkalmazása, több alkalmazás regisztrálhat, fogadhatja az üzeneteket (a feladó és címzettek között nincs kötés)
- JMS: bár a konkrét üzenetküldési mechanizmus MOM-specifikus, a fejlesztői API azonos, a JMS API bármilyen megfelelő MOM rendszer esetében alkalmazható (a JDBC-hez hasonlóan)
- Message-driven beans: standard JMS bean, aszinkron JMS üzeneteket küldhet és fogadhat.
- Az új specifikáció szerint már a JMS-en kívül más szolgáltatások, különböző protokollok is használhatók. A kiterjesztés lehetővé tette a JCA (Java Connector Architecture) bevezetését. Ha egy gyártó saját Message-driven Bean komponenst használ, az a JCA segítségével hordozható az EJB szerverek között (hardware analógia: USB)

Session Beans

- Üzleti logika, Bean-ek közötti interakció, műveletsorok (taskflow)
- Nem rögzítenek adatokat, de hozzáférnek azokhoz
- Állapottal rendelkező (stateful) és állapotnélküli (stateless) beanek
- **Stateless**: szolgáltatások (metódusok) kollekciója, állapotát nem őrzi meg két metódushívás között.
- **Stateful**: adott klienshez kapcsolódó műveletek elvégzése, az állapot megőrzése, folyamatos kommunikáció(bean-kliens) → conversational state → a metódusok adatokat írhatnak és olvashatnak ebbe/ebből, az adatok meg lesznek osztva a metódusok között. A bean-ek eltávolítása történhet a kliens explicit kérésére, vagy automatikusan egy megadott timeout periódus lejárta után.
- Az életciklus szempontjából a stateless bean-ek hosszabb életűek: nincsenek hozzárendelve egy klienshez, így egy adott feladat elvégzése után következhet egy új klienskérés kiszolgálása. Ezek a bean-ek is eltávolíthatóak (vagy timeout rendelhető hozzájuk), de ez a művelet csak a kliens referenciát érinti, a bean példánya nem lesz eltávolítva a konténerből.

Entity Beans

- POJOk (a szó szoros értelmében tulajdonképpen nem Enterprise bean-ek), üzleti logikával kapcsolatos, "főnevek" által meghatározható entitások reprezentációi (kliens, raktári tárgy, hely, felszerelés, stb.). Az adatok általában adatbázisban rögzítettek.
- Elsődleges kulcs (**primary key**): azonosítja a bean objektumot a memóriában, és ugyanakkor egy egyedi azonosítóként szolgál a megfelelő adatbázis bejegyzés számára.
- Bean osztály (**bean class**): perzisztens adatok reprezentációja objektumok segítségével. Ezen kívül, bizonyos esetekben tartalmazhat vonatkozó, üzleti logikával kapcsolatos kódot (validáció, stb.). POJO, nem kell semmilyen interfészt megvalósítania, és szerializálhatónak sem kell lennie. A **@javax.presistence.Entity** annotációval kell megjelölni, kell tartalmaznia egy mezőt, vagy getter metódust az elsődleges kulcs részére, amelyet a **@javax.persistence.Id** annotáció jelöl. Ezen kívül rendelkezésünkre állnak további annotációk, amelyekkel a teljes ORM leképezés megvalósítható.
- Nem komponensek, abban az értelemben, hogy az alkalmazások közvetlen módon férnek hozzájuk, nem interfészeken keresztül.

Entity Beans – Példa

```
package dev.com.titan.domain;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="CABIN")
public class Cabin implements java.io.Serializable {

    private static final long serialVersionUID = 1L;
    private int id;
    private String name;
    private int deckLevel;
    private int shipId;
    private int bedCount;

    @Id
    @Column(name="ID")
    public int getId() { return id; }
    public void setId(int id) { this.id = id;}
```

Entity Beans – Példa

```
    @Column(name="NAME")
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    @Column(name="DECK_LEVEL")
    public int getDeckLevel() { return deckLevel; }
    public void setDeckLevel(int deckLevel) { this.deckLevel = deckLevel; }

    @Column(name="SHIP_ID")
    public int getShipId() { return shipId; }
    public void setShipId(int shipId) { this.shipId = shipId; }

    @Column(name="BED_COUNT")
    public int getBedCount() { return bedCount; }
    public void setBedCount(int bedCount) { this.bedCount = bedCount; }

    public static long getSerialversionuid() { return serialVersionUID; }

    public String toString() { return id + " " + name; }

}
```

Telepítés leírók és csomagok

- XML Deployment Desciptors and JAR Files
- Entity Bean osztályok megírása → deployment (telepítés)
- Entity Bean-ek halmaza: perzisztencia egység (**persistence unit**) → megfeleltetés egy adatbázissal (a **Persistence Provider**-nek tudnia kell, hogyan dolgozza fel az adatokat)
- Perzisztencia egyég menedzsmentje: **EntityManager** szolgáltatás
- XML leíró (szükséges): **persistence.xml** (META-INF katalógus)

```
<persistence>
    <persistence-unit name="titan">
        <jta-data-source>MySQL</jta-data-source>
    </persistence-unit>
</persistence>
```

- Megfeleltetés: annotációk alkalmazása a bean osztályban
- Más lehetőség: XML leíró állományok → mapping
  - Az annotációk helyett/mellett alkalmazhatóak. Ha már léteznek az annotációk, az XML leírók ezeket felülírhatják, vagy további meta-adatokkal egészíthetik ki
- Miután megvannak a bean osztályok és XML leíró állományok, egy JAR állományt kell készíteni belőlük

Session Beans – interfészek és bean osztály

- A **Remote** interfész: üzleti logikával kapcsolatos metódusok, amelyeket az EJB konténeren kívül futó (kliens)alkalmazások meghívhatnak (a "külvilág" számára nyilvános metódusok). Egy egyszerű Java interfész, amelyet a **@javax.ejb.Remote** annotációval látunk el.
- A **Local** interfész: üzleti logikával kapcsolatos metódusok, amelyeket az EJB konténeren belüli más bean-ek meghívhatnak (az azonos JVM-en futó bean-ek számára nyilvános metódusok). Egy egyszerű Java interfész, amelyet a **@javax.ejb.Local** annotációval látunk el. Haszna: a konténeren belüli bean-ek osztott objektum protokollok nélkül kommunikálhatnak, ez jobb teljesítményhez vezet.
- Az **Endpoint** interfész: üzleti logikával kapcsolatos metódusok, amelyeket az EJB konténeren kívülről, SOAP protokollon keresztül érhetünk el. Java interfész, amelyet a **@javax.ejb.WebService** annotációval látunk el. A JAX-RPC Java API-n alapszik, főként a web-szolgáltatások alkalmazzák.
- **Bean** osztály: az üzleti logika megvalósítása, amelyhez tartozik legalább egy Remote/Local/Endpoint interfész (lehet több is, adott típusból is). Java osztály, amelyet a **@javax.ejb.Stateful**, vagy **@javax.ejb.Stateless** annotációval látunk el.

Session Beans – példa

```
import javax.ejb.Remote;

@Remote
public interface CalculatorRemote {
      public int add(int x, int y);
      public int substract(int x, int y);
}
```

```
import javax.ejb.Stateless;

@Stateless
public class CalculatorBean implements CalculatorRemote {

      public int add(int x, int y) {
            return x + y;
      }

      public int substract(int x, int y) {
            return x - y;
      }

}
```

Session Beans – taskflows

- Session Beans → üzleti logika → műveletsorok (taskflows) → tranzakciók
- Példa: kliens – foglalás (utazási ügynökség, tengeri utazás)

```
//get information (credit card number, etc.) from the text fields
String creditCard = textField1.getText();
int cabinId = Integer.parseInt(textField2.getText());
int cruiseId = Integer.parseInt(textField3.getText());

//create a Customer object
Customer customer = new Customer(name, address, phone);

//create a new TravelAgent session, passing in a reference
//to a Customer entity bean
TravelAgentRemote travelAgent = … ; //use JNDI to get a reference
travelAgent.setCustomer(customer);

//set cabin and cruise IDs
travelAgent.setCabinId(cabinId);
travelAgent.setCruiseId(cruiseId);

//using the card number and price, book passage
//this method returns a Reservation object
Reservation res = travelAgent.bookPassage(creditCard, price);
```

Session Beans – taskflows

```
…  // imports
@Stateful
public class TravelAgentBean implements TravelAgentRemote {
      @PersistenceContext private EntityManager entityManager;
      @EJB private ProcessPaymentRemote process;                    //injection

      private Customer customer;
      private Cruise cruise;
      private Cabin cabin;

public void setCustomer(Customer cust) {
         entityManager.create(cust);
         customer = cust;
}

public void setCabinId(int id) {
         cabin = entityManager.find(Cabin.class, id);
}

public void setCruiseId(int id) {
         cruise = entityManager.find(Cruise.class, id);
}
```

Session Beans – taskflows

```
public Reservation bookPassage(String card, double price)
                    throws IncompleteConversationalState {
          if (customer == null || cruise == null || cabin == null) {
                    throw new IncompleteConversationalState();
          }
          try {
                    Reservation reservation =
                              new Reservation(customer, cruise, cabin,
                                        price, new Date());

                    entityManager.persist(reservation);

                    process.byCredit(customer, card, price);

                    return reservation;
                } catch(Exception e) {
                    throw new EJBException(e);
                }
        }

    }
```

Message-driven Beans: interfész és osztály

- Rövidítés: **MDB**
- A **Message** interfész: azok a metódusok amelyeknek segítségével az üzenetküldő rendszer, például a JMS üzeneteket juttathat el a bean-hez.
- A **bean** osztály: az interfész(ek)ben definiált üzenetküldéssel kapcsolatos metódusok megvalósítása. Példa: **onMessage()**. A konténer akkor hívja meg ezeket a metódusokat, amikor egy új üzenet érkezik. Java osztály, amelyet a **@javax.ejb.MessageDriven** annotációval látunk el.
- Az EJB 3.0 konténereknek támogatniuk kell a JMS alapú, **javax.jms.MessageListener** interfészt implementáló MDB-ket. Ezen kívül más üzenetküldő rendszerek üzeneteit feldolgozó MDB-ket is támogatnak.
- Az MDB-k nem implementálnak remote, local vagy endpoint interfészeket, de együttműködhetnek session bean-ekkel, azok interfészein keresztül. Ezek a session bean-ek lehetnek velük egy konténerben, és ebben az esetben a kommunikáció a Local interfészen keresztül történik, vagy lehetnek más konténerben, és ebben az esetben a kommunikáció a Remote vagy Endpoint interfészeken keresztül történik.
- MDB példa: ReservationProcessorBean

MDB – példa

```
… //imports
@MessageDriven
public class ReservationProcessorBean implements javax.jms.MessageListener {
        @PersistenceContext private EntityManagare entityManager;
        @EJB private ProcessPaymentRemote process;
        public void onMessage(Message message) {
                try {
                        MapMessage reservationMsg = (MapMessage) message;
                        Customer customer =
                                (Customer) reservationMsg.getObject("Customer");
                        int cruisePk = reservationMsg.getInt("CruiseId");
                        int cabinPk = reservationMsg.getInt("CabinId");
                        double price = reservationMsg.getDouble("Price");
                        String card = reservationMsg.getString("Card");
                        entityManager.persist(customer);

                        Cruise cruise = entityManager.find(Cruise.class,cruisePk);
                        Cabin cabin = entityManager.find(Cabin.class, cabinPk);
                        Reservation reservation =
                                new Reservation(customer, cruise, cabin,
                                                price, new Date());
                        entityManager.create(reservation);
                        process.byCredit(customer, card, price);
                } catch (Exception ex) {
                        thorw new EJBException();
                }
        }
}
```

Enterprise beans - megjegyzések

- MapMessage: az üzeneteknek több formája lehet, a példában használt MapMessage név-érték párokat tárol/hordoz
- A MDB-n belüli műveletek ugyanazt a funkcionalitást látják el, hasonlóan működnek, mint a TravelAgent session bean esetében. A különbség, hogy az MDB-nek nem kell válaszolnia, nem térít vissza Reservation referenciát.
- Az MDB-k az állapot nélküli session bean-ekhez hasonló szerepet töltenek be, műveletsorok, tranzakciók végrehajtásáért, üzleti logikával kapcsolatos műveletek elvégzéséért felelősek. A különbség, hogy az MDB-khez a kérés egy aszinkron üzenet formájában jut el. A session bean-eknek válaszolniuk kell az interfészükön keresztül meghívott, üzleti logikával kapcsolatos metódusokra. Az MDB az onMessage metódus segítségével reagál a kérésre, a kliens nem vár választ.
- Coarse-grained ↔ fine-grained komponensek: a fine-grained komponensek publikus interfészeiken keresztül sok, belső működésükkel kapcsolatos információt közölnek. A coarse-grained komponensek elrejtik a részleteket. A távoli hívások, Remote interfészek esetében a coarse-grained megközelítés a javasolt – a kliens nem érdekelt a komponens működésével kapcsolatos részletek ismeretében.

Enterprise beans - megjegyzések

- A kliensek soha nem közvetlenül kommunikálnak a bean-ekkel, hanem mindig az interfészeken keresztül.
- Tulajdonképpen proxikkal és stub-okkal kommunikálnak (még a Local interfész alkalmazásának esetében is, ugyanis a konténer ezeket használja a bean és kliens kommunikációjának monitorizálására, bizonyos szolgáltatások (biztonság, tranzakciók) biztosítására.
- A proxik és stub-ok dinamikusan, futási időben lesznek generálva (lehetőség van arra is, hogy statikusan generáljuk őket).
- A bean-ek példányainak menedzsmentje a konténer feladatköre, ő felelős azért, hogy ezeket a szerver a megfelelő módon tárolja. A konténer rengeteg "háttérmunkát" végez, több eszközt biztosít (bean-ek és adatbázis táblák közötti megfeleltetés, kódgenerálás az interfészek alapján, stb., stb.)
- Elnevezési konvenciók: session bean: TravelAgent EJB (a teljes EJB – az interfészek és az osztály) → TravelAgentRemote, TravelAgentLocal, TravelAgentWS, TravelAgentBean

Annotációk, telepítés-leírók, JAR állományok

- Az EJB konténernek tudnia kell, hogy milyen módon biztosítsa a szolgáltatásait (tranzakciók, biztonság, stb.), és ezt az interfészek/bean-ek nem határozzák meg, a konténer futási időben tudja megszerezni a szükséges információkat, annotációk és leíró állományok alapján.
- Az egyszerűség kedvéért alapértelmezett beállításokat biztosít (pl. transaction – REQUIRED, security – UNCHECKED, stb.)
- Annotációk alkalmazása: egyszerűbb, a környezet kódkiegészítési lehetőséget biztosíthat, dokumentációs forrásként szolgálhat
- XML telepítés-leírók: lehetőséget adnak telepítés-függő beállításokra, újrafordítást nem igényelő, dinamikus beállításokra. A környezetek eszközöket biztosíthatnak a könnyebb szerkesztésre.
- Az osztályok, interfészek, telepítés leírók elkészítése után JAR csomagot hozunk létre, ez lesz telepítve a szerverre.

Hello EJB!

Egy egyszerű példa

Előkészítés

- Java SDK letöltése és telepítése
- Java EE platform és alkalmazásszerver letöltése és telepítése
  → Glassfish (open-source edition, v4)
- Fejlesztői környezet letöltése, konfigurálása
  → Eclipse IDE for Java EE Developers
- Az alkalmazásszervernek megfelelő Eclipse plug-in letöltése, telepítése
- Adatbázisszerver letöltése, telepítése, konfigurálása
- Adatbázis-specifikus driver letöltése (MySQL Connector), a jar állomány **bemásolása** az alkalmazásszerver megfelelő könyvtárába (glassfishv4/glassfish/lib)
- Adatbázis létrehozása
- JRE beállítása az Eclipse környezetben
  - Preferences → Java → Installed JREs → a **JDK** könyvtár hozzáadása (SDK szükséges!)
- Server beállításai az Eclipse környezetben:
  - Preferences → Server → Runtime Environments → Glassfish server
  - Beállítások: szerver könyvtár (glassfishv4/glassfish), JRE

Adatbázis

- Adatforrás beállítása az Eclipse környezetben
  - Data Source Explorer → New → beállítások
- Adatbázis táblák létrehozása (Eclipse → new SQL file → execute SQL files)

  Példa:

```
create table CABIN
(
  ID int primary key NOT NULL,
  SHIP_ID int,
  BED_COUNT int,
  NAME char(30),
  DECK_LEVEL int
)
```

Szerver konfigurálása

- Eclipse, Servers view → szerver (Glassfish) hozzáadás → indítás
  → view Admin Console
- Adatforrás beállítása:
  - Resources → JDBC → Connection Pools → New
    - Pool name: **MySQLPool**
      Resource type: **javax.sql.DataSource**
      Database vendor: **MySQL**
    - Datasource class name: **com.mysql.jdbc.jdbc2.optional.MysqlDataSource**
      Server name (pl. localhost)
      Database name (pl. test)
      URL (pl. jdbc:mysql://localhost:3306/test)
      Url (pl. jdbc:mysql://localhost:3306/test)
      User (pl. root)
      Password (pl. root password)
    - Ha szükséges, további tulajdonságok beállítása
  - Resources → JDBC → JDBC Resource
    - JNDI name (pl. MySQL)
    - Pool - MySQLPool

Projekt létrehozása

- Eclipse → New EJB project
  - Megjegyzés: szükségesek a Glassfish lib könyvtárában található jar állományok (az Eclipse ezeket automatikusan hozzárendeli): appserv-rt.jar, javaee.jar, jndi-properties.jar
- META-INF → persistence.xml létrehozása

```
<persistence>
      <persistence-unit name="titan">
                <jta-data-source>MySQL</jta-data-source>
      </persistence-unit>
</persistence>
```

- Az ejbModule mappában a csomagok létrehozása (pl. domain, travelagent)
- Bean-ek létrehozása:
  - Entity bean-ek (pl. Cabin)
  - Interfészek (pl. TravelAgentRemote)
  - EJB-k (pl. TravelAgentBean)
- Projekt fordítása, telepítése a szerverre: Export → EJB JAR file
  (megjegyzés: ha a szerver konfigurálásánál hozzárendeltük a projektet, és bekapcsoltuk az autodeploy opciót, az Eclipse automatikusan a megfelelő könyvtárba másolja a csomagot: pl. glassfish/domains/domain1/autodeploy/ejb_example.jar)
- Projekt futtatása: Run on server
- Megjegyzés: a log konzolon megtekinthetjük a bean-ekhez hozzárendelt azonosítókat/neveket (JNDI)

Entity bean

```
package dev.com.titan.domain;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="CABIN")
public class Cabin implements java.io.Serializable {

    private static final long serialVersionUID = 1L;

    private int id;
    private String name;
    private int deckLevel;
    private int shipId;
    private int bedCount;

    @Id
    @Column(name="ID")
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
```

Entity bean

```
@Column(name="NAME")
public String getName() { return name; }
public void setName(String name) { this.name = name; }

@Column(name="DECK_LEVEL")
public int getDeckLevel() { return deckLevel; }
public void setDeckLevel(int deckLevel) { this.deckLevel = deckLevel; }

@Column(name="SHIP_ID")
public int getShipId() { return shipId; }
public void setShipId(int shipId) { this.shipId = shipId; }

@Column(name="BED_COUNT")
public int getBedCount() { return bedCount; }
public void setBedCount(int bedCount) { this.bedCount = bedCount; }

public static long getSerialversionuid() { return serialVersionUID; }

public String toString() { return id + " " + name; }
}
```

Remote interface

```
package dev.com.titan.travelagent;

import javax.ejb.Remote;
import dev.com.titan.domain.Cabin;

@Remote
public interface TravelAgentRemote {

   public void createCabin(Cabin cabin);
   public Cabin findCabin(int id);

}
```

Session bean

```
package dev.com.titan.travelagent;

import dev.com.titan.domain.Cabin;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;


@Stateless(name="TravelAgent", mappedName="ejb/TravelAgentRemote")
public class TravelAgentBean implements TravelAgentRemote {

     @PersistenceContext(unitName="titan") private EntityManager manager;

    public Cabin findCabin(int id) {
              return manager.find(Cabin.class, id);
    }

    public void createCabin(Cabin cabin) {
              manager.persist(cabin);
    }

}
```

Kliens létrehozása

- Kliensalkalmazás projekt létrehozása
- A Glassfish modules könyvtárából a gf-client.jar csomag hozzárendelése

Példa:

```
package dev.com.titan.clients;
import javax.naming.InitialContext;
import dev.com.titan.domain.Cabin;
import dev.com.titan.travelagent.TravelAgentRemote;
public class StandaloneClient {
    public static void main(String args[]) {
    try {
            InitialContext ic = new InitialContext();
                TravelAgentRemote tar = (TravelAgentRemote)
                        ic.lookup("java:global/ejb_example/TravelAgent");
                Cabin c1 = new Cabin();
                c1.setId(1);
                c1.setName("valami");
                c1.setDeckLevel(1);
                c1.setBedCount(2);
                tar.createCabin(c1);
                Cabin c2 = tar.findCabin(1);
                System.out.println(c2);
            } catch(Exception e) {
                e.printStackTrace();
            }
        }
    }
```

Kliens, más példa

```
package dev.com.titan.clients;
import dev.com.titan.domain.*;
import dev.com.titan.travelagent.*;
import javax.naming.*;
import java.util.Properties;
import javax.rmi.PortableRemoteObject;
public class RemoteClient {
    public static void main(String[] args) {
        try {
            Context jndiContext = getInitialContext();
            Object ref = jndiContext.lookup(
                                "java:global/ejb_example/TravelAgent");
            TravelAgentRemote dao = (TravelAgentRemote)
                        PortableRemoteObject.narrow(ref, TravelAgentRemote.class);
            Cabin c1 = new Cabin();
            c1.setId(1); c1.setName("valami");
            c1.setDeckLevel(1); c1.setBedCount(2);
            dao.createCabin(c1);
            Cabin c2 = dao.findCabin(1);
            System.out.println(c2);
        } catch (javax.naming.NamingException ex) {
            ex.printStackTrace();
        }
    }
```

Kliens, más példa

```
public static Context getInitialContext() throws javax.naming.NamingException
    {
    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");
        return new InitialContext(props);
    }

}
```

EJB: Erőforrás menedzsment

Erőforrás menedzsment és alapvető szolgáltatások

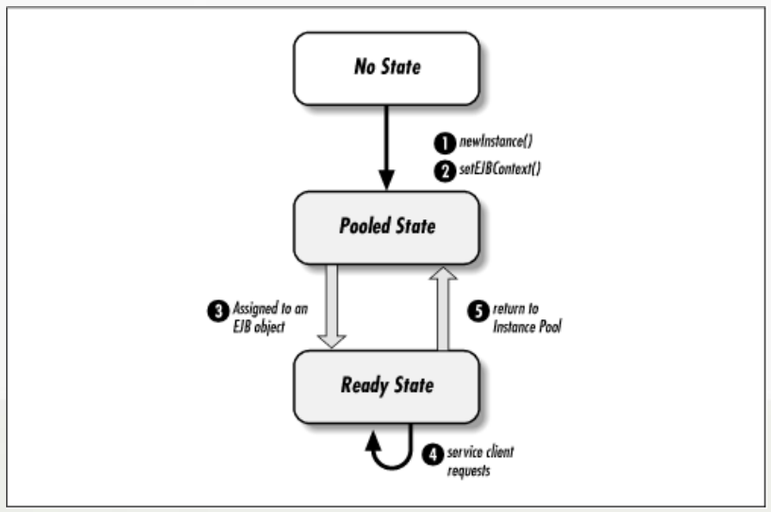
Simon Károly
simon.karoly@codespring.ro

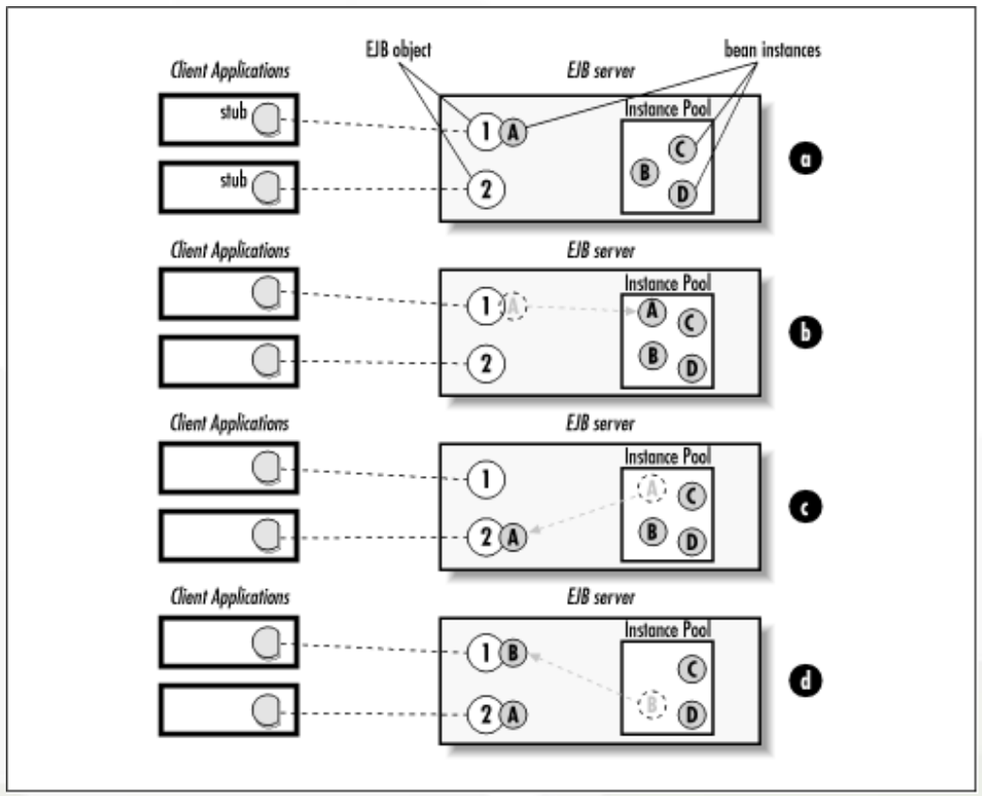
Erőforrás menedzsment

- Bonyolult vállalati rendszerek → nagyon sok kliens → egyszerre nagyon sok (több ezer/millió) használt objektum
- Szinkronizálással, tranzakció menedzsmenttel kapcsolatos problémák
- EJB megoldások a teljesítmény növelésére: instance pooling (stateless session bean-ek és MDB-k esetében) és activation (stateful session bean-ek esetében)
- Pooling mechanizmus: pl. adatbázis kapcsolatok
- EJB: az EJB konténerek a szerver-oldali komponensek esetében alkalmazzák az instance pooling mechanizmust, ezen kívül a JCA (Java EE Connector Architecture) szintén támogatja a pooling mechanizmust (erőforrások, adatbázis kapcsolatok, stb.)
- Az enterprise komponensekhez soha nem direkt módon férünk hozza, hanem interfészeiken keresztül (a kliens csak proxy stub-okkal van kapcsolatban) → lehetséges az instance pooling alkalmazása
- Pl. néhány állapot nélküli session bean több száz kliens kéréseit kiszolgálhatja: egy kérés kiszolgálásának ideje (metódushívás) tipikusan sokkal rövidebb, mint a két kérés között eltelt idő ("szünet").

Instance pooling

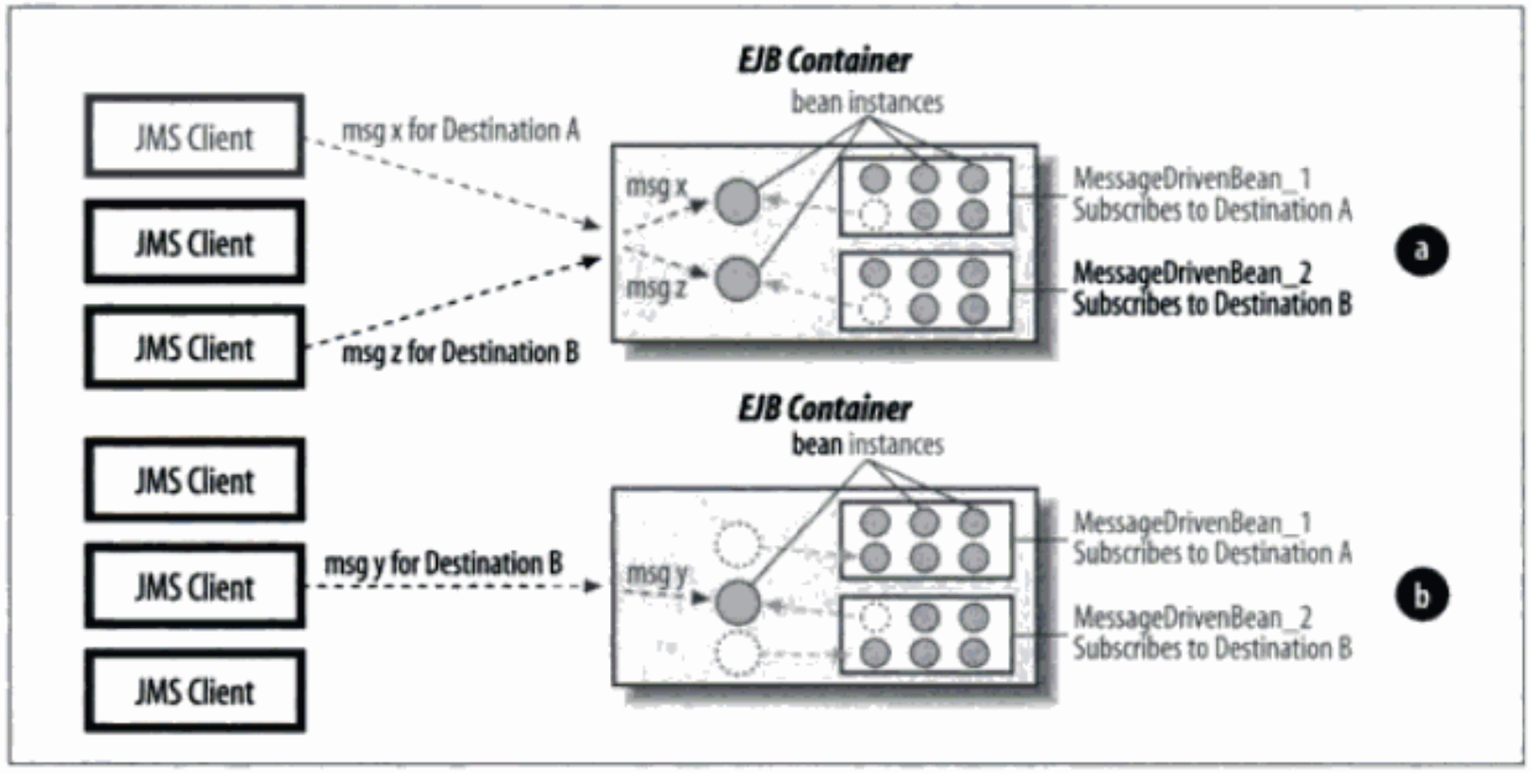
- Állapot nélküli session bean-ek életciklusa (a bean "állapotai"):
  - No state: még nem történt meg a példányosítás
  - Pooled state: a példányosítás megtörtént, de a bean még nem volt hozzárendelve kéréshez
  - Ready state: hozzá lett rendelve egy kéréshez, és készen áll az üzleti logikával kapcsolatos metódushívások fogadására
- Nem szükséges két metódushívás között megőrizni állapotinformációkat → lehetőség a kéréseket kiszolgáló példányok cseréjére (swapping)
- EJBContext: interfész a bean és konténer között, információkat szolgáltat a kliensről (+hozzáférés a stub proxy-hoz). Az erre mutató referencia a pool-ba helyezés után lesz átadva a bean-nek.





MDB - Instance pooling

- Minden MDB típusnak megfelel egy instance pool. Amikor kérés érkezik a konténer hozzárendel egy objektumot a pool-ból, majd az onMessage metódus visszatérése után visszahelyezi az objektumot a pool-ba.
- Ugyanahhoz a címzetthez több (száz) üzenet is érkezhet ugyanabban a pillanatban: a konténer minden üzenethez hozzárendelhet egy-egy MDB példányt a pool-ból.

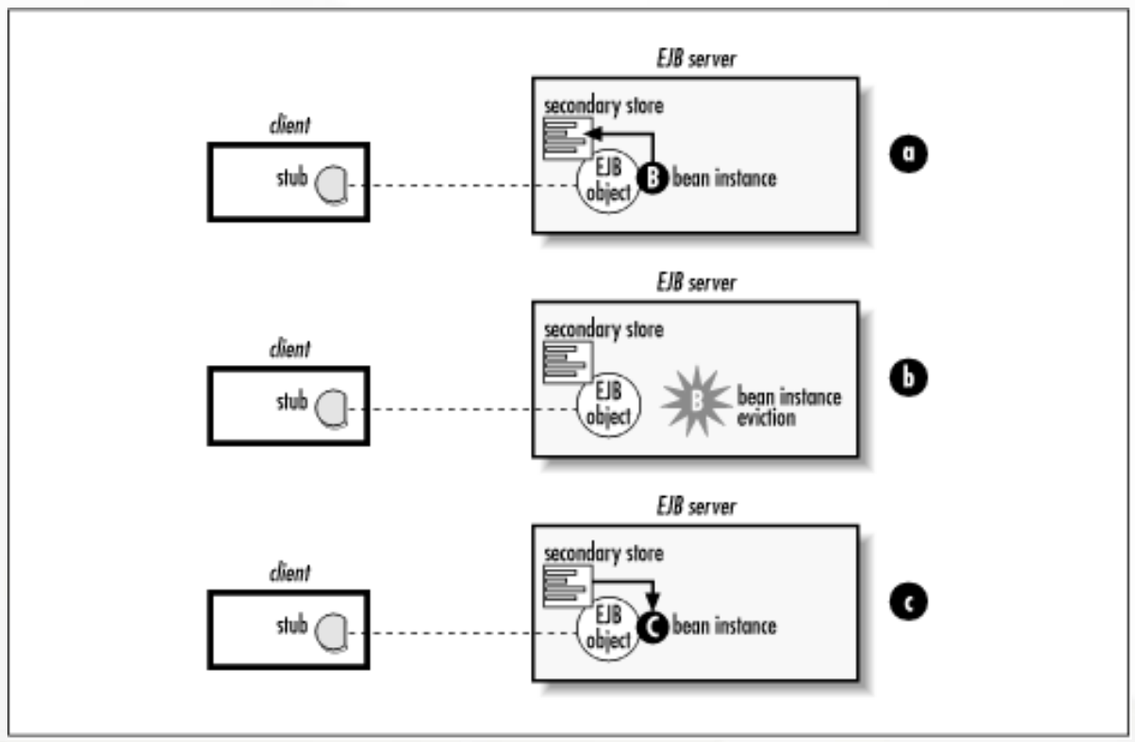


Activation Mechanism

- Stateful session bean → instance pool helyett "kilakoltatás" (eviction)
- A bean-eknek meg kell őrizniük az állapotukat (conversational state) → nem alkalmazhatjuk az instance pooling mechanizmust, de lehetőségünk van arra, hogy a conversational állapotot szerializáljuk, eltároljuk és ideiglenesen felszabadítsuk a memóriát (törölve belőle a bean példányát)
- Amikor új kérés érkezik újra példányosít a konténer, az elmentett adatok segítségével feltölti az állapotinformációkat, majd meghívja a megfelelő metódust
- Passivation: az állapot lementése és a memória felszabadítása (a kliens csak a proxy stub-bal van kapcsolatban, így ez a kapcsolat továbbra is fenntartható)
- Activation: példányosítás és az állapot visszatöltése
- A bean-nek nem kötelező szerializálhatónak lennie, így a konkrét állapot-mentési mechanizmus konténer specifikus. Következményként a transient adattagok viselkedése is implementáció függő (nem feltétlenül a megszokott). De: alkalmazhatóak a @javax.ejb.PostActivate és @javax.ejb.PrePassivate annotációk.

Activation Mechanism

- @javax.ejb.PostActivate annotációval ellátott metódus: az aktiválás után lesz meghívva (pl. transient adattagok inicializálására használható)
- @javax.ejb.PrePassivate annotációk: a passziválás előtt lesz meghívva (pl. erőforrások felszabadítására használható)
- A passziválás során rögzíteni kell: más bean-ekre mutató remote referenciákat, a SessionContext-et, a JNDI környezetet, a komponens interfészeket, az EntityManager szolgáltatást, a UserTransaction objektumot.



JCA

- Java EE Connector Architecture: interfész EIS (Enterprise Information System) rendszerek (adatbázis szerverek, message-oriented middleware rendszerek, ERP rendszerek stb.) és a Java EE konténer között.
- A Java EE standard, szolgáltató-független API-kat biztosít (JDBC, JMS, JNDI, Java IDL, JavaMail stb.), de a konkrét rendszerekkel történő kommunikációhoz ennél több kell (gyártó-specifikus megoldások) → JCA bevezetése
- Push üzenetküldő modell (JCA 1.5): az EIS rendszerek anélkül is továbbíthatnak üzeneteket, hogy azt a kliensek explicit módon kérnék → aszinkron üzenetküldés, MDB programozási modell.
- Konténer-konnektor interfész meghatározása
- Pl. X gyártó elkészít egy Java EE Connector-t egy Mail Delivery Agent (MDA) számára (a szoftver e-mail üzeneteket kézbesít). Meghatároz egy EmailListener üzenet-figyelő interfészt, és ezt bármelyik Email-MDB megvalósíthatja, így feliratkozhat az üzenetekre. A konténer az érkező üzeneteket továbbítja a megfelelő MDB-ekhez (push mechanizmus).

Alapvető szolgáltatások

- Szinkronizálás, párhuzamos hozzáférés:
  - a session bean-ek nem támogatják a párhuzamos hozzáférést (alaptulajdonságaik alapján egyértelmű). A párhuzamos kérések megfelelő kezelése a konténer feladata.
  - Az EJB specifikáció tiltja a synchronized kulcsszó használatát, a szinkronizálást a szerver biztosítja, nem a fejlesztő feladatköre.
  - A bean-eknek tilos saját végrehajtási szálakat létrehozniuk.
  - Az entity bean-ek esetében a persistence provider feladata a megosztott adatok védelme, a párhuzamos hozzáférés kezelése. Megoldás: tranzakció-alapú megközelítés, az objektumokból másolat készül, mindenik tranzakció számára. A párhuzamos hozzáférés lehetséges, a megfelelő lekezelést a konténer biztosítja, adott mechanizmus alapján. Pl. verziómezők alkalmazása (optimistic concurrency), lock-ok alkalmazása, SERIALIZED JDBC isolation level.
  - Párhuzamos hozzáférés (egyszerre több üzenet) MDB-k esetében: az instance pooling mechanizmus szolgáltatja a megoldást.

Alapvető szolgáltatások

- Tranzakciók:
  - Tranzakciók: több összefüggő művelet, amelyeket egy "atomi" műveletbe kell összevonni, mindeniket végre kell hajtani ahhoz, hogy a tranzakció sikeres legyen (pl. bookPassage metódus: ne vegyünk le pénzt a kártyáról, ha nem jön létre a foglalás)
  - A tranzakció-kezelést a konténer automatikusan végzi, a fejlesztőknek a bean-ek megírása során nem kell erre figyeljenek, nem kell speciális API-kat használniuk. Egyszerűen a telepítéskor (deployment) transactional-nak kell deklarálni a megfelelő attribútumokat. Ezen kívül lehetőség van explicit tranzakció-kezelésre is (példák később).
- Perzisztencia:
  - EntityManager szolgáltatás: entity bean-ek "kapcsolása" a konténerhez (metódusok a létrehozásra, lekérdezésre, módosításra, törlésre). Egy tranzakció befejezése után a példányok "lekapcsolhatóak" a konténerről, elküldhetőek távoli klienseknek, és esetleges módosítások után lehetőség van a "visszakapcsolásra" (EntityManager.merge()), a perzisztens adatok szinkronizálására. A mechanizmus feleslegessé teszi a DTO minta alkalmazását.
  - ORM támogatás (különböző kapcsolattípusok, öröklődés, multitable mappings, EJBQL stb.)

Alapvető szolgáltatások

- Osztott objektumok:
  - EJB client view: a kliensek az enterprise bean-eket csak interfészeiken keresztül látják (kivételt képeznek az entity bean-ek, amelyek ha szerializálhatóak, átküldhetőek a hálózaton).
  - Az objektumok elérésére bármilyen osztott objektum protokoll alkalmazható. Az EJB 3.0 specifikáció szerint a szervernek támogatnia kell a Java RMI-IIOP protokollt (Java RMI API használata, CORBA IIOP protokoll alkalmazásával), a JAX-RPC API-n keresztüli SOAP 1.2 protokoll támogatását.
  - A bean-ekhez más programozási nyelvben megírt kliensek is hozzáférhetnek (CORBA vagy SOAP alkalmazásával).
  - Aszinkron Az EJB konténer biztosítja az megfelelő üzenetkezelést. Ide több szolgáltatás is tartozik: rendszerhiba esetén meg kell próbálni az üzenetet később továbbítani, közben lehetséges, hogy perzisztenssé kell tenni (el kell menteni), egy adott számú sikertelen próbálkozás után egy speciális "dead message" repositoryba kell helyezni, stb.
  - Az üzenetküldés tranzakciószerű

Alapvető szolgáltatások

- EJB Timer szolgáltatás:
  - Időzített értesítések eljuttatása a beanek-hez (entity, session és MD bean-ekhez egyaránt hozzárendelhető). Pl. banki rendszer adott időközönként ellenőrzi, hogy a tartozásokat fizették-e, stb.
- Security:
  - Azonosítás (authentication): felhasználók azonosítása
  - Engedélyezés (authorization), hozzáférés ellenőrzése (access control): felhasználók jogosultságainak ellenőrzése
  - Biztonságos kommunikáció (secure communication): a kliens és szerver közötti kommunikációs csatorna biztosítása, kriptált kommunikáció. A konténereknek támogatniuk kell a Secure Socket Layer (SSL 3.0) és Transport Layer Security (TLS 1.0) protokollokat.
- Naming:
  - Osztott objektumok és erőforrások azonosítása, lokalizálása. Azonosítók objektumokhoz rendelése (object binding), keresési mechanizmus (lookup).
  - JNDI API alkalmazása. Különböző szolgáltatók alkalmazhatóak, de minden konténernek támogatnia kell a CORBA naming szolgáltatást (CosNaming).
- Együttműködés (interoperability):
  - CORBA (IIOP) és JAX-RPC (SOAP, WSDL)