Szakdolgozat

Akszenovics Sándor

Debrecen

2011

Debreceni Egyetem

Informatikai Kar

Információ technológia Tanszék

Java Enterprise Computing

Konzulens:

Dr. Fazekas Gábor

egyetemi docens

Készítette

Akszenovics Sándor

Programtervező informatikus

Debrecen

2011

Tartalom

Bevezetés

Az utóbbi években rohamosan felgyorsult az internetes technológiák fejlődése, ennek következtében az üzleti szférában nyilvánvalóvá vált, hogy ahhoz, hogy egy üzleti alkalmazás versenyképes maradjon, követnie kell a technológiai változásokat. Emiatt egyre nagyobb teret hódítanak az elosztott, hordozható alkalmazások, melyekre igaz a biztonságosság, skálázhatóság, méretezhetőség és a nagy rendelkezésre állás, valamint az elosztott struktúrájuk miatt egyes komponensek viszonylag könnyen módosíthatók, bővíthetők.

A Java nyelven megírt üzleti alkalmazásokhoz a Sun Microsystems nyújt specifikációgyűjteményt, ez a Java Enterprise Edition, amely jelenleg a 6. verziószámnál tart.

A szakdolgozatom célja a Java EE 6 részeit képező egyes API-k működési elveinek a részletezése, de a téma nagy terjedelme miatt a teljes specifikációnak a fontosabb részleteit tekintem át.

Java EE felépítése

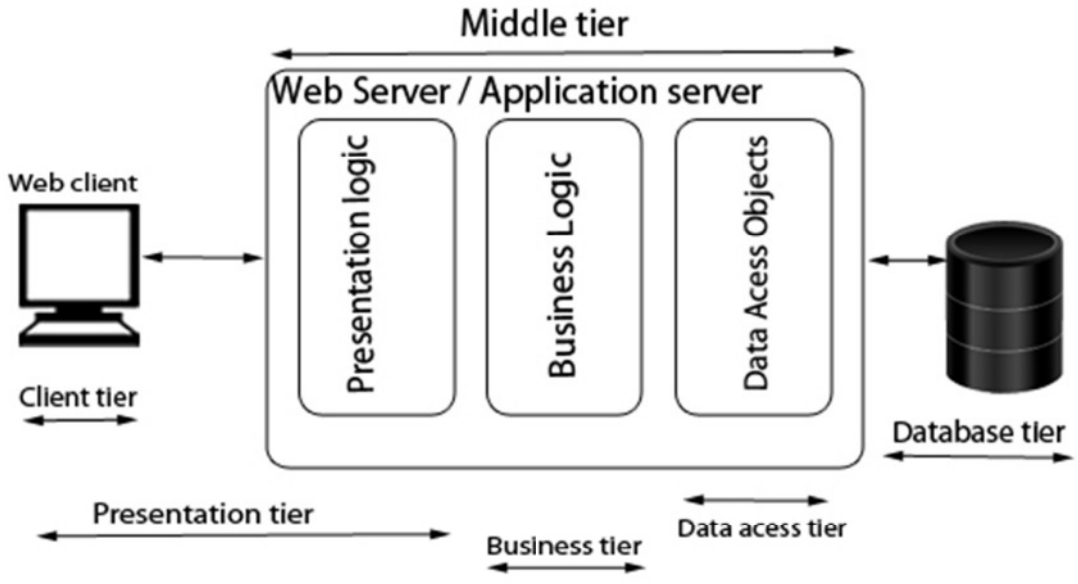
A JEE a többrétegű web alkalmazások elosztott, többrétegű architektúráját alkalmazza. A többrétegű architektúra alatt a következőt értjük:

- Database tier: csak adatokat tartalmaz
- Middle tier: itt található az üzleti és a megjelenítésre vonatkozó logika, valamint az adatbázis elérésére vonatkozó komponensek.
- Client tier: csak GUI (jellemzően egy weboldal)

A Java EE a Middle tier-re vonatkozóan tartalmaz specifikációkat, ajánlásokat.
Más, de hasonló szemlélet szerint a rétegezés a következő:

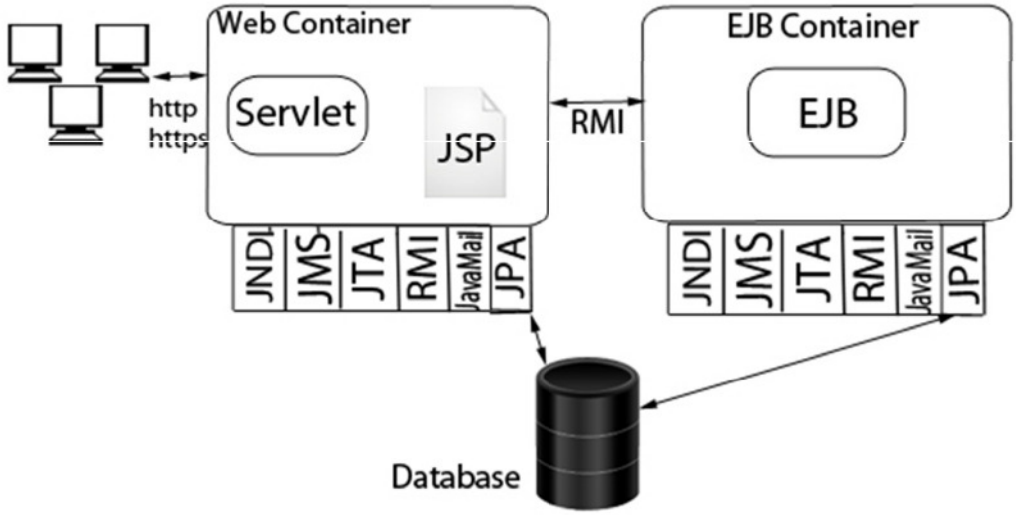
- Prezentációs réteg: GUI + adatok megjelenítésének logikája
- Üzleti réteg: az üzleti logikát tartalmazza
- Adathozzáférési réteg: a tárolt adatok kezeléséért felelős réteg

A két szemlélet sematikus ábráját az 1. ábra szemlélteti:



1. ábra *Egy elosztott web alkalmazás architektúrája*

A Java EE és komponenseinek felépítését a következőképp lehetne sematikusan ábrázolni:



2. ábra *A Java EE felépítése, az alkotó komponensek szerepköreinek sematikus ábrázolásával.*

Java EE alapfogalmak

Komponens

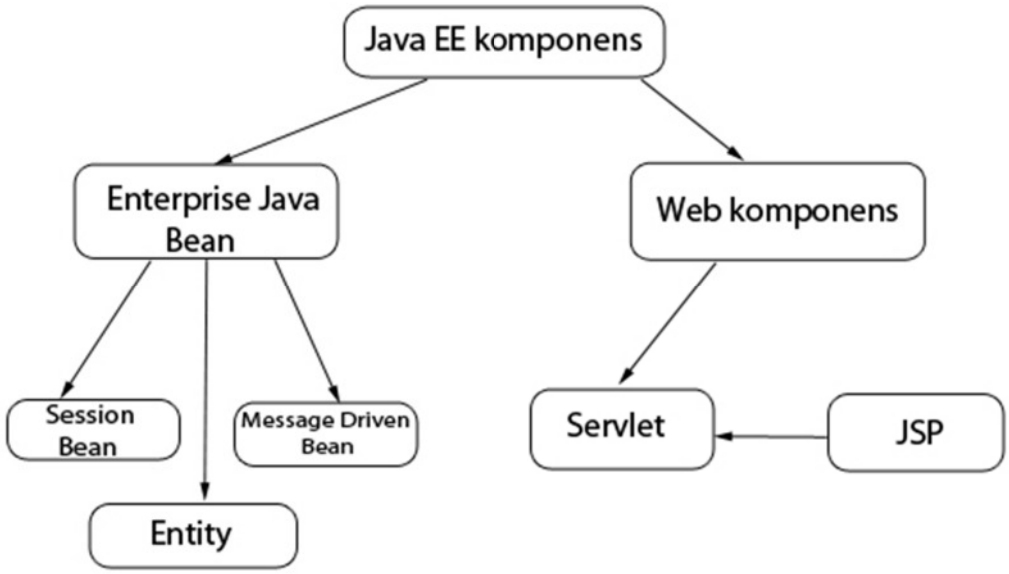
A Java EE alkalmazások komponensekből állnak. Ezek olyan programegységek, melyek Java nyelven íródnak, beépíthetőek a Java EE alkalmazásba, más komponensekkel kommunikálhatnak. Java EE specifikáció szerint kell felépülniük, egy konténerben futnak és a konténer által nyújtott szolgáltatásokat érhetik el.

Konténer

A konténer a komponens és az alacsonyszintű, platformfüggő funkcionalitás közötti elem. A komponenst le kell fordítani és a konténerbe helyezni – ezt nevezzük deployolásnak. A konténerek alapszolgáltatásai közé tartoznak:

- biztonság kezelése: felhasználók, jogosultságok azonosítása, kezelése
- tranzakciók kezelése
- JNDI
- Távoli eljáráshívás (RMI/IIOP)

A Web konténer a JSP és a Servlet komponenseket, az EJB konténer az EJB-ket kezeli.



3. ábra *Java EE komponensek*

Webszerver

A webszerver egy webes rendszer esetén az a kiszolgáló, amely a kliens böngészőjéből érkező kéréseket megválaszolja. Java EE világában a webszerver és a web konténer fogalma átfedi egymást. A web konténer legtöbb esetben egy egyszerűbb kiszolgálót takar, amely futtatni tud megfelelően megírt Java alkalmazásokat (servleteket). Egy példa a web konténerre: Apache Tomcat.

Alkalmazásszerver (App server)

A web konténer hatásköréből kieső, komplex üzleti feladatokat az alkalmazásszerver teljesíti. Egy app server egy EJB konténerből és implementációtól függően egy web konténerből áll össze. Egy alkalmazásszerver funkciói többek között az adatbázisban található adatstruktúra leképezése Java osztályokra, valamint ezek állapotának tárolása (JPA), az üzleti logika megvalósítása Enterprise JavaBean-ek formájában. Rengeteg rutinfeladatot vesz át a fejlesztőktől, mint például az adatbázis-kapcsolatok kezelését, vagy az erőforrások elosztását több kiszolgáló számítógép között. Az alkalmazásszerveren futó rendszernek mindegy, hogy minden egy gépre lett telepítve vagy több gépen elosztva fut a rendszer. Alkalmazásszerver implementációk: JBoss, Oracle WebLogic.

Egy Java EE alkalmazás fizikai felépítése

A könnyebb hordozhatóság és újrafelhasználás miatt a Java EE alkalmazásokat egy előre specifikált struktúrába szokás csomagolni.

EAR

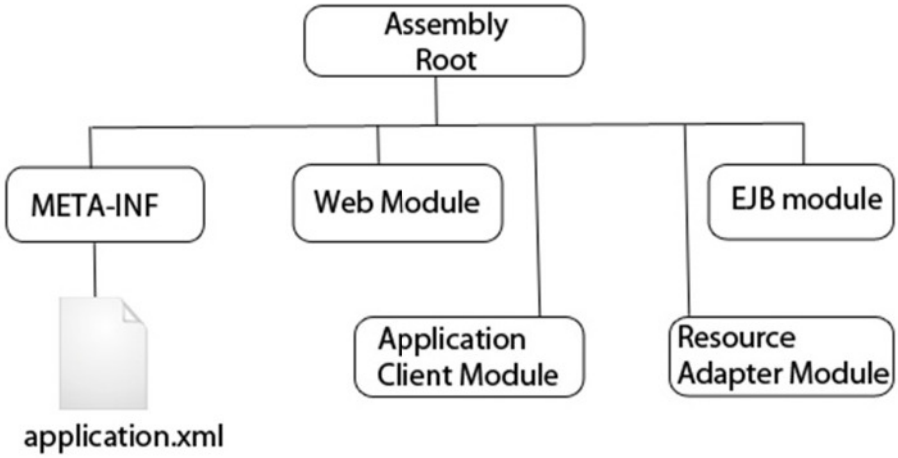
Az Enterprise ARchive rövidítése. Ez egy standard JAR (Java ARchive file) .ear kiterjesztéssel. Tartalmazza a Java EE modulokat és a deployment descriptort.

Deployment descriptor

Egy XML dokumentum, .xml kiterjesztéssel. Tartozhat alkalmazáshoz, modulhoz vagy komponenshez, az adott elem telepítéséhez szükséges adatokat tartalmaz. Megváltoztatható anélkül, hogy a kódot változtatni kellene.

Két típusa van:

- Java EE descriptor: a szabványos beállításokat tartalmazza
- runtime descriptor: alkalmazásszerver specifikus beállítások (pl: sun-application.xml)



4. ábra *Az EAR file szerkezete*

Modulok

Négy különböző típusú modult tartalmazhat az EAR file:

- EJB modulok
- Web modulok
- Alkalmazás kliens modulok
- Erőforrás adapter modulok

EJB modulok

Tartalmazza a tipikusan üzleti logikát tartalmazó Enterprose Java Bean osztályokat valamint az EJB descriptort, amit általában ejb-jar.xml –nek szokták elnevezni. A csomagolt file kiterjesztése .jar.

Web modulok

Servlet osztályokat, JSP fájlokat, esetleg egyéb, kisegítő osztályokat tartalmaz. Lehet benne még statikus tartalom is, mint például kép vagy HTML oldal. A csomagolt file kiterjesztése .war (Web ARchive). A WAR file tartalmaz egy speciális WEB-INF könyvtárat is, ebben van a webalkalmazás deployment descriptora, a web.xml. A classes alkönyvtárban lehetnek a servlet osztályok.

Alkalmazás kliens modulok

A szükséges osztályokat és a deployment descriptort tartalmazza. A descriptor neve általában application-client.xml. A csomagolt file kiterjesztése .jar.

Erőforrás adapter modulok

Egy adott EIS (Enterprise Information System) szolgáltatásainak eléréséhez implementált kapcsolati osztályokat, eszközöket valamint a deployment descriptort tartalmazza. A csomagolt file kiterjesztése .rar.

Servlet

Eredetileg a servlet technológia bármilyen típusú kérést tud kezelni, a gyakorlatban a HTTP kérések kezelése terjedt el. Így a servlet egy olyan Java objektum, amely HTTP kérést dolgoz fel és HTTP választ generál. Ezzel ugyanúgy a dinamikus tartalomgenerálás problémáját oldja meg, mint a PHP vagy a CGI. A generált tartalom javarészt HTML (web-alkalmazások révén), de lehet például XML is.

A servlet számos előnnyel rendelkezik a hasonló technológiákkal (mint például a CGI) szemben:

- a kliensektől érkezett kérések után nem egy új folyamat indul el, hanem csak egy új szál – erőforrások jobb kihasználása
- a servlet képes adatokat tárolni a kérések között, mivel a memóriában marad, ezzel elősegítve az olyan tevékenységeket, mint a session tracking
- Java nyelv biztonsági mechanizmusai érvényesek a servletekre is.

A servlet container az alkalmazásszerver azon komponense, amely a servleteket kezeli. A konténer dolga a servletek életciklusainak a kezelése és az URL-ek hozzárendelése a servletekhez. A servletek telepítési módja a konténerben sokféle lehet, az utóbbi implementációjától függően. Minden konténer esetében meg kell adni a telepítéskor azt az URL mintát, ahol a servlet elérhető. Az egyéb paraméterek konténerfüggőek.

Egy servlet életciklusa

A következő fázisokból áll:

1. Ha a servletnek nem létezik példánya, akkor a konténer betölti a Servlet osztályt, példányosítja azt az argumentum nélküli konstruktor meghívásával.

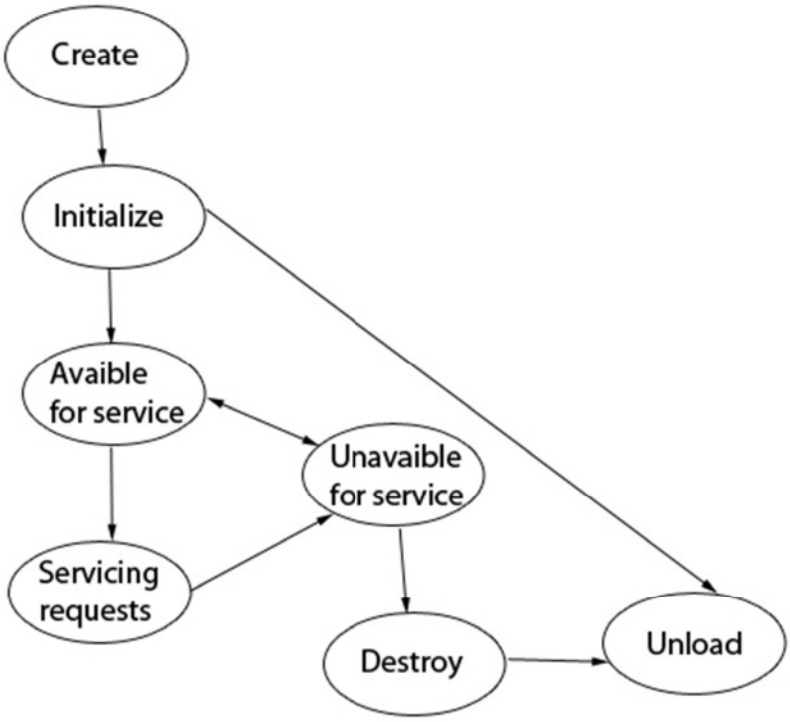
2. A konténer meghívja a servlet init() metódusát. Ez a metódus inicializálja a servletet és mindenképp le kell futnia mielőtt a servlet HTTP kéréseket tudna fogadni. Az init() metódus csak egyszer fut le a servlet életciklusa során.
   Az init() metódusban a következő feladatok hajtódhatnak végre:

   - globális változók inicializálása

- Servlet config kezelése. A ServletConfig példány minden servlethez külön létezik. Ez az objektum az egyes servletek inicializációjához szükséges paramétereket tartalmazza: adatbázis URL, servlet induló paraméterei, stb.
- Servlet context kezelése. A ServletContext csak egy van minden alkalmazásban. Ezt az objektumot minden servlet használhatja alkalmazásszintű információk és konténeradatok lekérdezésére
- Adatbázis kapcsolatok megnyitása

3. A konténer minden kérésre meghívja a servlet service() metódusát. Ha a Servlet egy HttpServlet, akkor a service() metódus a HTTP kérés típusának megfelelő metódust hívja meg (doPost(), doGet(), stb.).

4. Ha a konténernek el kell távolítania egy servletet akkor meghívja a servlet destroy() metódusát. Az init() metódushoz hasonlóan a destroy() is csak egyszer hajtódik végre a szerver életében.



5. ábra *Egy servlet életciklusa*

Web.xml

Ha a web alkalmazás csak JSP-ket tartalmaz, a file elhagyható. Servletek esetén viszont ebben a fájlban adható meg, hogy az adott URL kérést mely servlethez irányítsa át a web konténer.

A web.xml tartalmazhatja továbbá kontextus változók definícióját is, amelyek a servletekből hivatkozhatóak. Tartalmazhat továbbá környezeti függőségeket is, amelyeket az alkalmazásszervernek rendelkezésre kell bocsátania a futtatáshoz. Ilyen függőség lehet egy mail session e-mailok küldéséhez szükséges beállítás.

Példa egy web.xml-re:

```
<web-app id="MyWebApp" version="2.4"
        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 ">
<web-app>
  <display-name>
       Az alkalmazás neve
  </display-name>
  <description>
         Az alkalmazás leírása
  </description>
  <servlet>
    <servlet-name>MyServlet</servlet-name>
      <servlet-class>mypackage.MyServlet</servlet-class>
    </servlet>
    <servlet-mapping>
       <servlet-name>MyServlet</servlet-name>
       <url-pattern>/MyServlet.do</url-pattern>
    </servlet-mapping>
 </web-app>
```

JavaServer Pages

A JSP kialakításának legfőbb célja az volt, hogy megkönnyítsék a servlet fejlesztők munkáját – servletek esetén az output szöveg módosítása a forráskód újrafordításával valamint újratelepítésével jár.

Amíg a servletekben a Java kódban szerepelnek a szöveget a kimenetre író utasítások, addig a JSP oldalak esetén a rögzített szöveg közé rejtve szerepelhetnek az oldal tartalmát dinamikusan módosító utasítások. Így a JSP-k inline kódot tartalmazó HTML/XML oldalaknak tekinthetők.

Két fajta szöveget tartalmaznak: statikus adatot (például HTML kód) valamint JSP elemeket, amik a dinamikus tartalom generálásáért felelnek.

A JSP-k életciklusát servletekhez hasonlóan is a web konténer szabályozza.

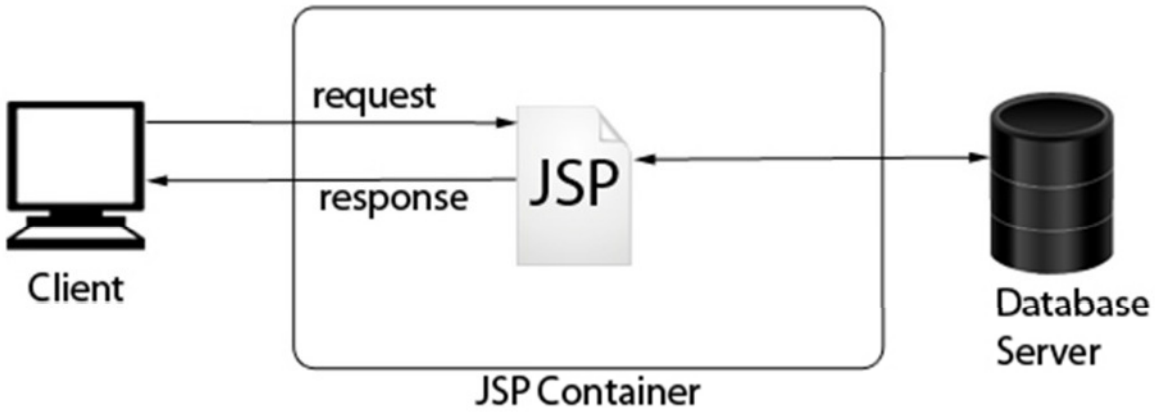
Egy JSP életciklusa

A következő fázisokból áll:

1. Amikor érkezik egy JSP fájlra mutató kérés, a konténer veszi át a vezérlést.
2. Amennyiben a JSP még nincs lefordítva, a JSP compiler (a konténer része, Tomcat esetén a Jasper) Java kódot hoz létre, amiből a szabványos Java fordító segítségével .class file jön létre, ami nem más, mint egy servlet.
3. A konténer a JSP-hez tartozó servletet betölti és a web konténer felé továbbítja a kérést. Az összes többi kérés közvetlenül a generált servlet felé irányítódik és a korábban taglalt servlet életciklus hajtódik végre.

JSP architektúrák

Egy egyszerű, kérés-válasz paradigmán alapuló alkalmazás akár kizárólag JSP oldalakból is állhat. Ezt nevezük JSP Model 0 –s architektúrának:

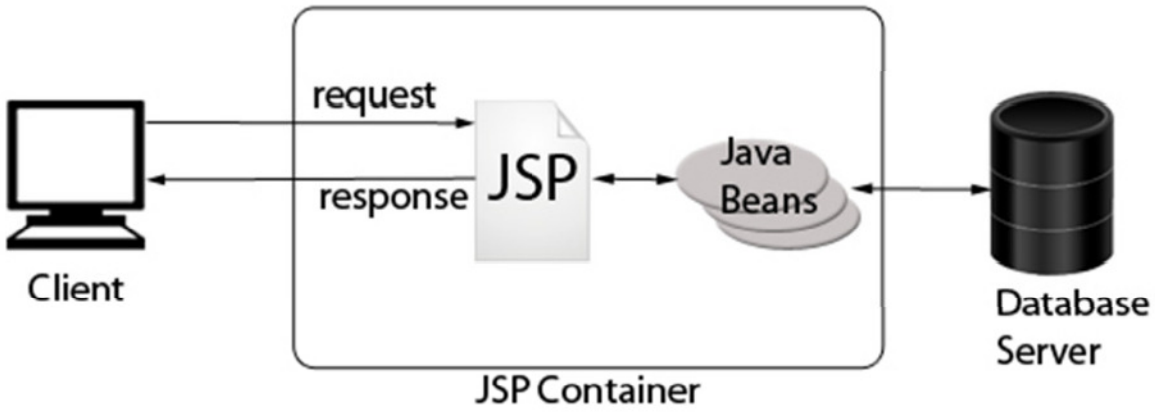


6. ábra *Model-0 -s architektúra*

A JavaBeans egy Java technológia, amely tulajdonképpen egy konvencióhoz igazított osztályokat jelent. A konvenció szerint egy Java osztály főleg attól egy JavaBean, ha

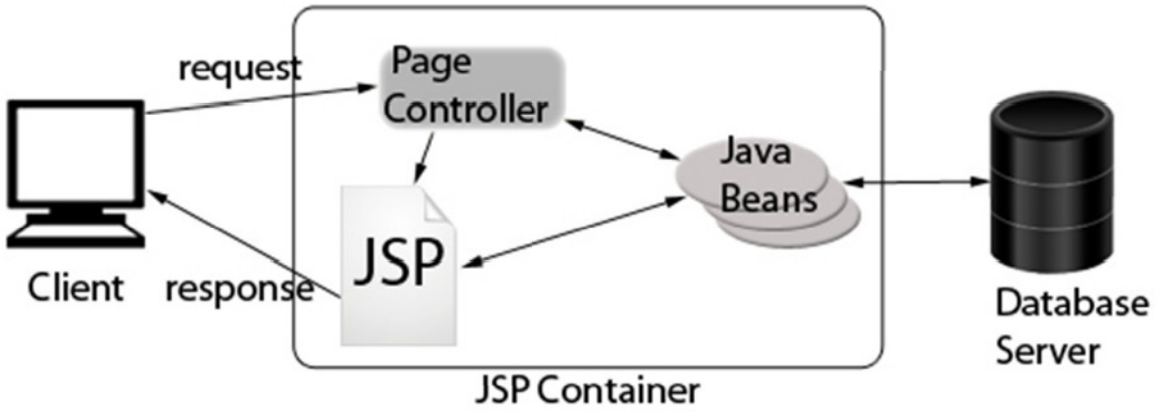
- van publikus, paraméterek nélküli konstruktora
- az összes attribútuma állítható és lekérdezhető get/set metódusokkal
- szerializálható.

Amennyiben bevezetjük a JavaBean-eket is a Model-0-s architektúrába, de az üzleti logikát még mindig nem választjuk el a megjelenéstől, ez esetben Model 1 –ről beszélünk:



7. ábra *Model-1 –es architektúra*

Amennyiben az adatokat, az üzleti logikát és a megjelenést is szeparáljuk, Model 2-s architektúráról van szó ami az MVC (Model-View-Controller) tervezési mintán alapszik:



9. ábra *Model-2 –es architektúra*

A Model egy JavaBean, amely adatokat közvetít a Controller és a View között.

A Controller az alkalmazás vezérléséért felelős alkalmazásréteg, amely szerepét a servletek biztosítják. Itt található a kérések eloszlása valamint az alkalmazáslogika.

A View feladata a Model adatainak megjelenítése, ez a JSP oldalak feladata.

Ez még nem egy valódi MVC implementáció, mivel az üzleti logikának a Model rétegben kell lennie, a Controller csak a vezérléssel kell, hogy foglalkozzon. A JavaBeans koncepció üzleti logikát tipikusan nem tartalmaz. Ellenben a Servlet-JSP-EJB hármas már alkothat valódi MVC implementációt.

Direktívák

A direktívák a JSP konténernek szóló utasítások, közös jellemzőjük, hogy

```
<%@ direktívanév attribútum_1="érték_1" attribútum_2="érték_2" … %>
```

alakúak.

A page direktíva

Az egész oldalra vonatkozó jellemzőket állíthatjuk be. A megadható attribútumok és lehetséges értékeik a következők:

- import: ezzel az utasítással a JSP oldalból generált servlet import listáját egészíthetjük ki.

```
<%@ page import="java.util.*" %>
```

- extends: azt adja meg, hogy a JSP oldalból generált osztály mely osztálynak legyen a leszármazottja.

```
<%@ page extends="MyClass" %>
```

- session : logikai értékkel határozzuk meg azt, hogy az oldalon akarjuk-e használni a session implicit objektumot. Az alapbeállítás `true`.

```
<%@ page session=false %>
```

- buffer: megadható a kimeneti puffer minimális mérete. Az alapértelmezett érték 8kb. Amennyiben none van megadva, ezesetben a kimenet közvetlenül a HTTP válasz PrintWriter objektumára íródik.

```
<%@ page buffer="32kb" %>
<%@ page buffer="none" %>
```

- autoFlush: logikai érték, amellyel meghatározhatjuk, hogy mi történjen akkor, ha a kimeneti puffer megtelik. Ha az értéke true, a puffer automatikusan ürítődik, ellenkező esetben kivétel váltódik ki puffer megtelése esetén. Alapértelmezett értéke `true`.

```
<%@ page autoFlush=false %>
```

- errorPage: a JSP kivételkezelési mechanizmus része. Annak a JSP oldalnak az URL-jét tartalmazza, amelyikhez az adott oldalon fellépő java.lang.Throwable osztályú kivételt továbbítani szeretnénk. A kivételkezelő oldal a kivételt a ServletRequest objektum javax.servlet.jsp.JspException típusú paramétereként kapja meg.

```
<%@page  errorPage="error.jsp"  %>
```

- isErrorPage: kivételkezelő oldal esetén `true` értékre állítva az átadott kivétel elérhetővé válik az implicit exception változón keresztül.
- contentType: az oldal MIME típusát megadhatjuk ezzel az attribútummal.

```
<%@page contentType="text/html;charset=ISO-8859-1"  %>
```

Az include direktíva

Egy adott file fordítás előtti beillesztésére szolgál. A beillesztés statikus, hasonlatos a C nyelv #include direktívájához vagy a PHP include() függvényéhez.

Egyetlen attribútuma van a file, amely a beillesztendő fájl relatív URL-ét tartalmazza.

```
<%@ include file="valami_file.jspf" %>
```

A taglib direktíva

Saját elemeket is felismertethetjük a JSP fordítóval, erre használatos a taglib direktíva. Paraméterként meg kell adni egy URI-t, ahol a saját elemeink könyvtárát leíró fájlja található, valamint egy prefixet, amellyel hivatkozni tudunk a saját elemeinkre.

```
<%@ taglib prefix="myPrefix" uri="taglib/myLeiro.tld" %>
```

Scriptelemek

A JSP oldalakba háromféleképpen lehet Java kódot beilleszteni: deklarációk, szkriptrészletek és kifejezések formájában.

Deklarációk

A deklarációs részben definiált kód teljes egészében bemásolódik a generált java servlet osztály forráskódjába. Adattagot vagy metódust is tartalmazhat.

```
<%! int myInstanceVariable = 100; %>

<%!

 public void increaseMyInstanceVariable() {

    myInstanceVariable++;

 } %>
```

Szkriprészletek

A szkriptrészletben megadott kód a generált servlet osztály _jspService() metódusába másolódik be.

```
<% increaseMyInstanceVarible();

 out.println(„Instance variable value is ” + myIstanceVariable); %>
```

Kifejezések

A kifejezés részben megadott kód futási időben értékelődik ki és az értéke a HTML kódba kerül bele.

```
<%= myInsanceVariable %>
```

Akciók

A JSP oldalakban az igazi, újrafelhasználható komponenseken alapuló paradigma az akciókban jelenik meg igazán.

Az akció tulajdonképpen nem más, mint egy XML tag a JSP oldalon belül, amely mögött egy újrafelhasználható Java osztály(ok) vannak jelen. A standard JSP akciók mellett az 1.1-es specifikáció óta lehetőség van saját akciók megírására is, így lehetőség nyílik arra, hogy teljesen eltüntethető legyen az inline Java kód a JSP oldalakból, ezzel erősítve az újrafelhasználást az alkalmazásban.

Standard JSP akciók

- <jsp:include>: az akció segítségével az oldal egyik pontján átadjuk a vezérlést egy másik oldalnak, ha az utóbbi válaszolt, akkor a vezérlés visszakerül a hívóhoz, a hívott oldal futási eredménye beillesztődik a hívott oldalba. Az a különbség az include direktíva és a <jsp:include> akció között, hogy míg a direktíva statikusan illeszti be egy másik oldal tartalmát fordítás előtt, addig a <jsp:include> akció dinamikusan, a HTTP kérés kezelésekor fut le és így dinamikusan generált tartalmat is be tud illeszteni.
  Példa:

```
<jsp:include page="/relative/url/path.jsp" flush="true />
```

  A page paraméter a behívandó oldal relatív URL-je, a flush pedig egy boolean érték, amellyel meghatározhatjuk, hogy a puffert kell-e üríteni a beillesztés előtt.

- <jsp:forward>: az akció segítségével a http kérés egy másik URL-re kerül át, vezérlés visszatérés nélkül.
  Egyetlen paramétere a page, amely a hívott erőforrás relatív URL-jét tartalmazza.
  Példa:

```
<jsp:forward page="/relative/url/path.jsp" />
```

- <jsp:param>: az akció csak <jsp:include>, <jsp:forward> és <jsp:params> akció blokkokban alkalmazható, a paraméterek átadására almalmazható, kulcs-érték párokat tartalmazhat. Két paramétere van: a name és a value.
  Példa:

```
<jsp:include page="/relative/url/path.jsp" flush="true" >
<jsp:param name="paramName" value="paramValue" />
</jsp:include>
```

- <jsp:useBean>: az akció segítségével létrehozhatunk vagy újra felhasználhatunk egy már létező JavaBeant. Az újrahasznosítás egy paraméterül megkapott id alapján történik egy szintén paraméterként megkapott névtéren belül.
  Példa:

```
<jsp:useBean id="myBean" scope="request" class="java.util.List" />
```

  Az id az újonnan létrehozott/újrafelhasznált bean azonosítója, a scope az a névtér, ahol a keresett bean található, a class a java bean példány osztályának a teljes neve.

- <jsp:getProperty>: paraméterként magkapott java bean azonosító és a paraméternév segítségével kinyerhetjük az adott azonosítójú java bean paraméternevű attribútumát. Az akció a lekérés eredményét a JSP oldal szabványos kimenetére írja ki.
  Példa:

```
<jsp:useBean id="myBean" scope="request" class="java.util.List" />
<jsp:getProperty name="myBean" property="attributeName/>
```

- <jsp:setProperty>: a <jsp:getProperty> akcióval ellentétben beállítjuk a paraméter java bean attribútumát.

  Példa:

```
<jsp:useBean id="myBean" scope="request" class="java.util.List" />

<jsp:setProperty name="myBean" property="attributeName"
value="attributeValue"/>
```

Elemkönyvtárak

A JSP oldalakban saját elemkönyvtárakat (custom tag library) is alkalmazhatunk. A saját elemkönyvtár a következőkből áll:

- egy vagy több Java osztályból, ahol a tényleges logika implementálva van.
- vagy egy taglib direktívából vagy a web.xml webalkalmazás leíró fájlban egy taglib leíróból.

  Példa:

```
<web-app>
  <taglib>
  <taglib-uri>http://localhost:8080/myApp/taglibs
  </taglib-uri>
  <taglib-location>/WEB-INF/foo.tld
  </taglib-location>
</web-app>
```

- elemkönyvtár leíró XML file-ból, ennek a kiterjesztése .tld, az angol tag library descriptor rövidítésből. Példa egy TLD fájlra:

```
<taglib>
  <tlib-version>2.0</tlib-version>
  <short-name>fooTag</short-name>
  <description>
     Here comes the description of a tag library
  </description>
  <tag>
  <name>foo</name>
  <tag-class>my.package.MyTag</tag-class>
    <body-content>tagdependent</body-content>
  <attribute>
    <name>attribute_1</name>
  </attribute>
  <attribute>
    <name> attribute _2</name>
  </attribute>
  <attribute>
    <name> attribute 3</name>
  </attribute>
  </tag>
</taglib>
```

Java Persistence API

A Java EE platform egyik fontos része az üzleti adatok, üzleti objektumok állapotainak megőrzése, perzisztenciája. Erre remek megoldást nyújt a Java Persistence API (röviden JPA), egy olyan keretrendszer, amely relációs leképezési technikájával az üzleti alkalmazásokban használt objektumokat relációs adatbázisban tárolhatjuk. Ez a technika az object-relational mapping (ORM), ami az objektum-relációs leképezést jelenti. Ezzel lehetőség van arra, hogy viszonylag egyszerű konfigurálással a Java objektumokat relációs adatbázis tábláinak feleltessük meg.

A JPA az EJB 3.0 specifikáció része, a korábban alkalmazott Entity Beanek egyszerűsítésére, leváltására lett létrehozva. A JPA egyik legjobb és közismert implementációja a Hibernate.

A JPA három részből áll:

- Entity Manager API
- Java Persistence Query Language
- ORM konfigurációs meta-adatok

Entity Manager API

Egy olyan API, amellyel az entitásokon elvégezhetőek a CRUD (Create, Read, Update, Delete) műveletek. Mivel az entitások a leképezési adatokat önmagukban tartalmazzák, ezek a meta-adatok segítségével az Entity Manager tudja végrehajtani a perzisztencia műveleteket. Emellett a JPA számára további szolgáltatásokat nyújt, mint például a skálázhatóság, életciklus kezelése vagy a tranzakció kezelés.

JPQL

Az SQL-hez hasonló szintaktikájú lekérdező nyelv, amellyel az adatbázisban tárolt entitásokat el lehet érni.

ORM konfigurációs meta-adatok

Segítségükkel történik az entitásoknak táblákhoz való rendelése.

JPA kulcsfogalmak

- Entity
- Persistence Unit
- Persistence Context

Entity

Az Entitások egyszerű java osztályok – POJO-k, ami a Plain Old Java Object rövidítése. Egyszerű Java osztályok, new() operátorral jönnek létre, nincs kötelező interfész, amit implementálniuk kell, de támogatják az öröklést és a polimorfizmust.

Általában egy entitás egy táblát reprezentál egy relációs adatbázisban és minden entitás példány megfelel a tábla egy-egy rekordjának. Az entitások kapcsolatban lehetnek más entitásokkal és ezeket a kapcsolatokat az objektum/relációs meta-adatokban fogalmazzuk meg. Ezek a meta-adatok vagy közvetlenül az entitás osztály forrásfájljába annotációk alakjában vagy egy külön XML leíró fájlban vannak jelen.

Az entitások szerializálhatóak – azaz byte sorozattá alakíthatóak, a JPA lekérdező nyelvével kérhetőek le, futásidőben az Entity Manager API-n keresztül érhetőek el.

Persistence Unit

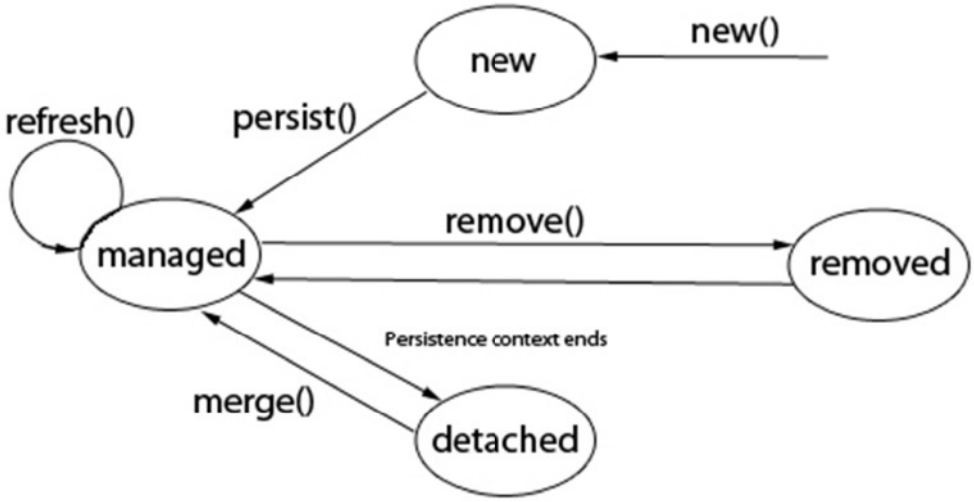
A csomagolás egysége, entitások és kapcsolódó osztályok halmazából áll. Tartalmazza az objektum relációs leképezési információkat, így a relációs adatbázis nézetét a Java-ból, annotációkat és XML leírókat, amelyekkel definiálja a lekérdezések és az entitás relációknak a hatáskörét. A persistence provider (így pl. a Hibernate) számára a konfigurációs információ a persistence.xml –ben található meg.

Persistence Context

Ez egy futásidejű alkalmazás környezet, amely egy persistence unit-hoz tartozó futási időben kezelt entitások halmazát tartalmazza. Élettartama lehet tranzakció alapú – egy vagy több tranzakció élettartamáig tartó, vagy alkalmazás által vezérelt.

Entitások állapotai

- new: new-val létrehozva kerül ide, csak a memóriában létezik, a módosítások nem kerülnek adatbázisba.
- managed: létezik az adatbázisban, és hozzátartozik egy perzisztencia kontextushoz. Ez azzal jár, hogy a módosítások tranzakció commit végén, vagy explicit flush() hívással bekerülnek az adatbázisba.
- detached: adatbázisban megvan, de nem tartozik perzisztens kontextushoz.
- removed: még perzisztencia kontextushoz tartozik, de már ki van jelölve, hogy törlésre kerül az adatbázisból.



9. ábra *Entitások életciklusa*

JNDI

Osztott feldolgozású rendszerek esetén gyakran jelent problémát az egymástól elválasztott rendszerrészek, erőforrások lokalizációja. Mivel a JEE architektúra alapvetően elosztott rendszerben gondolkodik, rendelkezésre bocsájt egy megoldást erre a problémára. Ez a megoldás a JNDI, azaz Java Naming and Directory Interface, magyarul Java névleképezés és katalógusinterfész – egy olyan API, amely lehetőséget nyújt adatok illetve objektumok név szerinti regisztrációját, amely alapján ezen erőforrások szükség esetén azonosíthatóak és karbantarthatóak.

Mivel a JNDI egy interfész, így független a tényleges implementációtól, mögötte lévő megvalósítás tetszőleges lehet, így katalógustechnológia esetén a LDAP, névszolgáltatás esetén egy file rendszer vagy a DNS.

A JNDI legfontosabb szerepe az elosztott üzleti alkalmazásokban használt enterprise beanek és adatbázis kapcsolatok azonosítása és paraméterezése.

A JNDI két fő részre bontható fel: alkalmazói - (API) és szolgáltatói interfészre (SPI). Az API-t a kliensek használják, míg az SPI-t azok az alkalmazások (pl.RMI), melyek szolgáltatásaikat ezen keresztül akarják elérhetővé tenni.

Egy alkalmazás JNDI struktúrája az esetek java részében fastruktúrát vesz fel. A fa csomópontjai a context nevet, a gyökérelem az initial context nevet viseli. Ha ebbe a szerkezetbe egy elemet illesztünk be, akkor kötésről (binding) beszélünk. A kötések egy összetartozó halmazát névtartománynak (namespace) nevezzük.

Alapfogalmak

- Név: pontos szintaktikai szabályok alapján épül fel, ezek a névkonvenciók, amelyek névszolgáltatás specifikusak. Így egy név lehet /folder/subfolder/data.property file rendszer esetén vagy subdomain.maindomain.org DNS esetén.

- Atomi név: a neveknek a névszolgáltatás szabályai által meghatározott, tovább már nem bontható része. Például a folder vagy a maindomain

- Kötés: objektum hozzárendelése atomi névhez. Például egy Property objektum hozzárendelése egy data.property filehoz.

- Kontextus: összetartozó egyedi kötések listája. Feladata a kikeresési művelet, a lookup, megvalósítása. Egy kontextuson belül létrehozható egy olyan kötés, amely egy másik kontextust rendel atomi névhez, így a kontextusok hierarchikus struktúrába rendezhetők. Így egy nem atomi név kikeresése úgy történik, hogy a név alkotóelemeit sorba véve a JNDI kikeresi a kérdéses objektumot/adatot.

- Névszolgáltatás: összekapcsolt azonos típusú kontextusok összessége, amelyekre ugyanazok a névkonvencionális szabályok érvényesek.

- Névtartomány: egy névszolgáltatáson belül érvényes nevek összessége.

A JNDI –n belül a névszolgáltatás ki van egészítve azzal, hogy attribútumok kapcsolhatók a regisztrált objektumokhoz. Ezáltal az objektumok attribútumok szerint is kereshetők, nemcsak név szerint.

Egy JEE alkalmazás telepítésekor minden EJB komponens, és minden regisztrált erőforrás (pl. adatbázis, üzenetsor) kap egy JNDI nevet, amit az alkalmazás szerverben lévő JNDI provider tart nyilván. A komponensek a külső erőforrásokat és más komponenseket JNDI név alapján keresik meg.

JNDI esetén egy név alatt egy objektumot lehet regisztrálni. Az ütközéseket a JNDI-fa felépítésekor kell kezelni. Az egyik megoldás a névütközések kiküszöbölésére az indirekt JNDI nevek használata. Az indirekt elérés lényege, hogy a komponens forráskódjában minden JNDI keresés csak logikai névre vonatkozzon. Ehhez készíteni kell egy leíró fájlt, ami tartalmazza ezeket a logikai neveket. Az alkalmazás összeállítása és telepítésekor kell feloldani, hogy ezek a logikai nevek milyen tényleges JNDI nevekre képződjenek le.

Rövid példa a JNDI használatára:

```
Context initialContext = new InitialContext();
Object obj = initialContext.lookup(„MyClassNameInJNDI”);
```

Java Message Service

A JMS egy, a vállalati üzenetkezelő rendszerek, szoftverkomponensek számára kifejlesztett Java API.

Az elosztott rendszerekben az üzenetküldés egy úgynevezett lazán csatolt, aszinkron jellegű kommunikáció. Ez azt jelenti, hogy a szoftverkomponensek nem közvetlenül kommunikálnak egymással, hanem egy köztes üzenetkezelő komponens segítségével. Az ilyen kommunikáció egyik előnye az, hogy az üzenetek küldőinek nem is kell pontosan ismerniük a fogadókat, mert minden kommunikáció az üzenetsor (message queue) segítségével történik.

JMS elemei

- JMS provider (szolgáltató) : A JMS szolgáltató nem más, mint egy Message Oriented Middleware (MOM) implementáció. Ez lehet tiszta Java implementáció, de lehet egyéb nyelven megírt MOM implementációhoz készített Java-s adapter is.
  A provider kezeli a munkameneteket és a sorokat. Több nyílt forráskódú (OpenJMS, JBoss Messaging, stb.) és kereskedelmi (Weblogic, WebSphere MQ, stb.) implementáció van jelen.

- JMS kliens: egy üzeneteket küldő és/vagy fogadó alkalmazás vagy folyamat.

- JMS producer: üzeneteket készítő JMS kliens

- JMS consumer: üzeneteket fogadó JMS kliens

- JMS message: a kézbesítendő adatokat tartalmazó objektum

- JMS queue: sor adatszerkezet implementációja, ebben találhatóak meg a kézbesítendő üzenetek. A kiküldés az érkezési sorrendben történik, kézbesítés után az üzenet törlődik a sorból.

- JMS topic: ez a mechanizmus szolgálja ki az üzenetek elosztását több címzett esetén

A JMS API két különböző üzenetküldési modellt támogat:

- point-to-point modell

- publish-subscribe modell

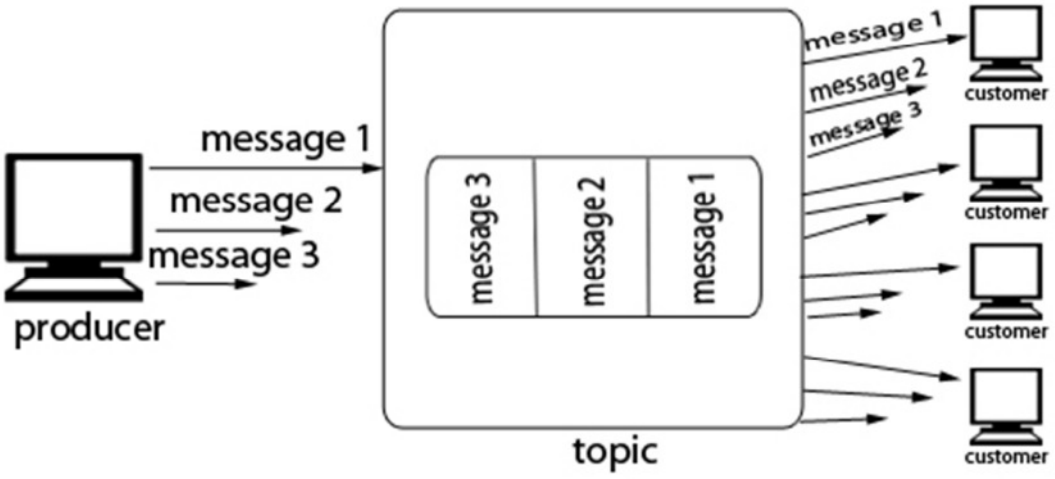
Pont-to-point modell

Két kliens között zajlik a kommunikáció: a producer üzenetet helyez el a sorba, a consumer pedig kiolvassa azokat. A producer ismeri a costumert, közvetlenül a fogadó félhez tartozó sorba pakolja az üzenetét. Az üzenetküldés aszinkron módon történik, azaz a fogadónak nem kell futnia, amikor az üzenet a queue-ba kerül, és ez fordítva is igaz a fogadáskor. A customer mindig visszaigazolja, ha sikeresen megkapta az üzenetet.

10. ábra *Point-to-point modell*

Publish-subscribe modell

Egy küldő és több fogadó fél van jelen. A producer létrehoz egy topicot, amelyre a customerek tetszőleges száma feliratkozhat. Üzenetküldéskor csak azok a customerek kapják meg az üzenetet, akik fel vannak iratkozva. A kommunikációs felek nem ismerik egymást. Ahhoz, hogy egy customer megkapjon egy üzenetet, két mód van: vagy folyamatosan aktívnak kell lennie, vagy tartós feliratkozást kell használnia, ami annyit jelent, hogy egy pszeudo point-to-point kapcsolat jön létre a customer és a producer között, így a fogadó fél inaktív lehet üzenetküldéskor, aktiv állapotában újracsatlakozáskor megkapja a „kimaradt” üzeneteket.



11. ábra *Publish-subscribe modell*

Enterprise JavaBeans

Az Enterprise JavaBeanek az elosztott, moduláris vállalati szoftverekben alkalmazott, üzleti logikát implementáló, szabványos interfésszel rendelkező szerveroldali komponensek specifikációja, a JEE platform része. Ezt a technológiát akkor alkalmazzák, amikor a következő követelmények java részének teljesülnie kell:

- perzisztencia kezelése: alkalmazás leállítása után az üzleti objektumok adatai nem vesznek el. A perzisztens tár szerepét javarészt az adatbázisok veszik át.
- tranzakció kezelés: perzisztens adatok konzisztenciáját biztosítva érjük el azokat.
- újrafelhasználhatóság: elvárás egy üzleti objektummal szemben, hogy különböző komponensekkel, más alkalmazásban is felhasználható legyen.
- távoli elérés: egy üzleti objektumot és annak szolgáltatásait távolról is el lehessen érni.
- transzparens hibatűrés: ha egy szerver elérhetetlenné válik, a másik szerver átveszi a kliensek kiszolgálását, anélkül, hogy az utóbbiak ezt észrevennék.
- biztonság: az üzleti objektumok által nyújtott szolgáltatásokhoz védelmi mechanizmus rendelhető hozzá, meghatározható, hogy milyen felhasználói vagy szerepköri privilégiumokkal érhető el.

Az EJB konténer

Hasonlóan a servletekhez, az EJB is a container tervezési mintára épül. A container pattern lényege az, hogy a közös funkciókat kiemeljük és a konténerben koncentráljuk. Ezek után a konténer és az üzleti alkalmazáslogika között interfészt definiálunk. A webalkalmazásoknál ilyen interfész van jelen a servlet és servlet konténer között. Servlet konténer esetén a HTTP-kezelés és a célcímnek megfelelő erőforrás elérése volt a feladat.

Jelen esetben az alkalmazáslogikát a kötött interfésszel rendelkező Jáva osztályokban, Enterprise JavaBeans komponensekben implementáljuk és ezeket az EJB-ket regisztráljuk az EJB konténernél.

Az EJB konténer tulajdonképpen egy szoftver, amely az EJB specifikációkat implementálja, egy olyan környezetet biztosít, amelyben az EJB komponensek léteznek.

Az EJB konténer funkciói a következők:

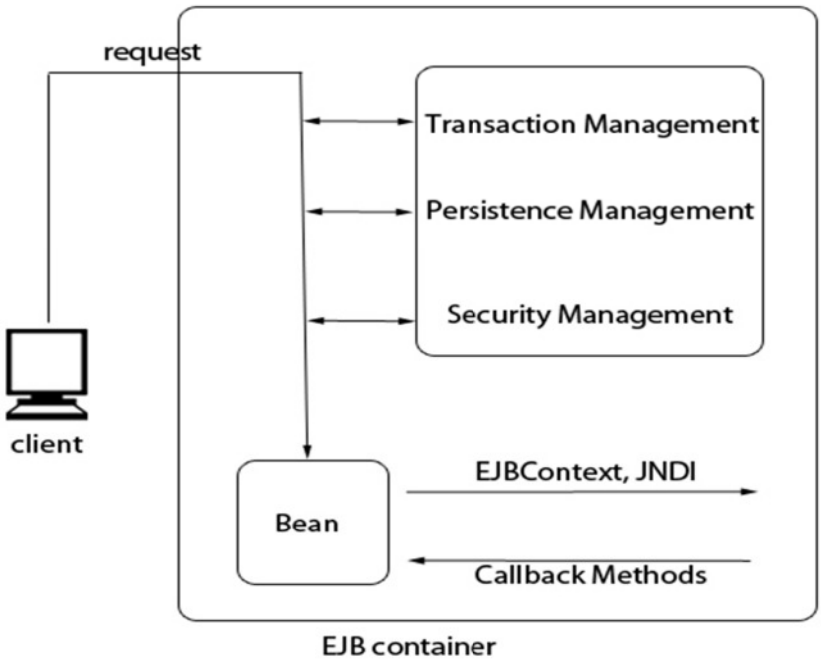
- Az EJB-k életciklusának vezérlése, az EJB-k létrehozása és elpusztítása
- Az EJB-k egymással való kommunikációjának lehetővé tétele
- Az EJB-k elérhetőségének biztosítása a kliensek felől
- Az EJB-k hozzáférésének ellenőrzése, autentikációs és autorizációs szabályok betartatása
- Az EJB-k katalógusának nyilvántartása, hogy a kliensek és más EJB-k név szerint hivatkozhassanak az EJB-re (a JNDI segítségével, mivel minden EJB konténer része egy JNDI katalógus)
- Az adatbázishoz való hozzáférés ellenőrzése, és biztosítása

Ezen felül egy EJB konténer skálázhatósági és hibatűrési szolgáltatásokat is nyújthat cluster-es környezetben. Az EJB technológia egyik legfontosabb képessége pont ezeknek a szolgáltatásoknak az igénybevételében van, az EJB specifikációban megírt alkalmazás a megfelelő alkalmazásszerveren telepítve skálázható, azaz kapacitása további számítógépek hozzáadásával elvileg korlát nélkül növelhető, valamint hibatűrő lesz, ami azt jelenti, hogy ha egy gép kiesik a cluster-ből, az csak kapacitáscsökkenést okozhat, de a rendszer leállását nem.

Az Enterprise JavaBeaneknek két típusa van:

- Message-driven bean (üzenetvezérelt bean)
- Session bean

A 2006-os EJB 3.0-s verzió előtt a specifikáció definiálta még egy 3. típusú EJB-t, az entity beant, csakhogy az a 3.0-s specifikáció óta sikeresen helyettesítve van a JPA entitásokkal.



12. ábra *A kliens és az EJB kapcsolata*

Session bean

A szerepük az üzleti folyamatok reprezentálásában van, főleg olyan műveletekre alkalmazzák, mint az adatbázis elérés vagy bonyolult számítás. Tulajdonképpen úgy áll elő egy session bean, hogy az egyes üzleti használati eseteknek metódusokat feleltetjük meg, az összetartozó metódusokat pedig egy-egy session beanhez rendeljük.

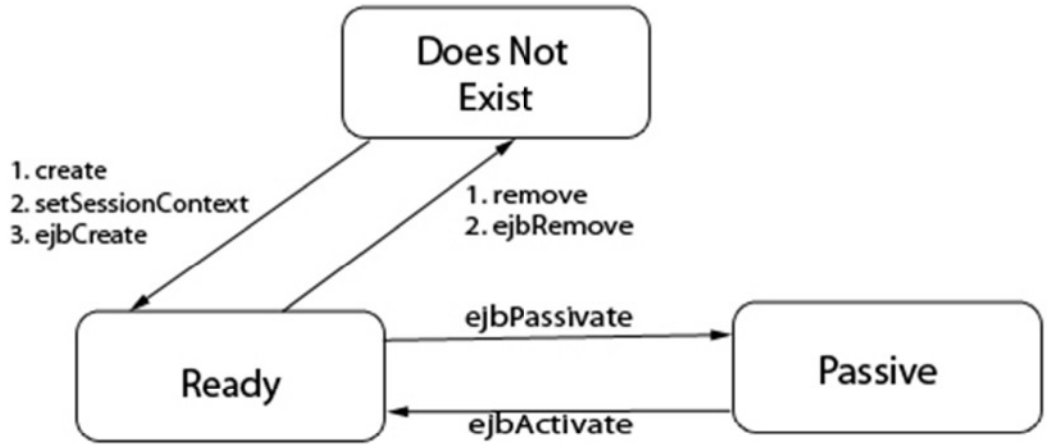
A session bean elnevezése onnan ered, hogy az élettartamuk az őt hívó klienssel való kapcsolatának az időtartamáig terjed, így amennyiben a kliens valamilyen okból nem folytatja a kapcsolatot a beannel, ez esetben a konténer meg is szüntetheti az adott session bean példányát. Általában egy session beanhez egy kliens fér hozzá.

A session beannek további három altípusa különböztethető meg:

- statefull session bean
- stateless session bean
- singletone session bean

Stateful session bean

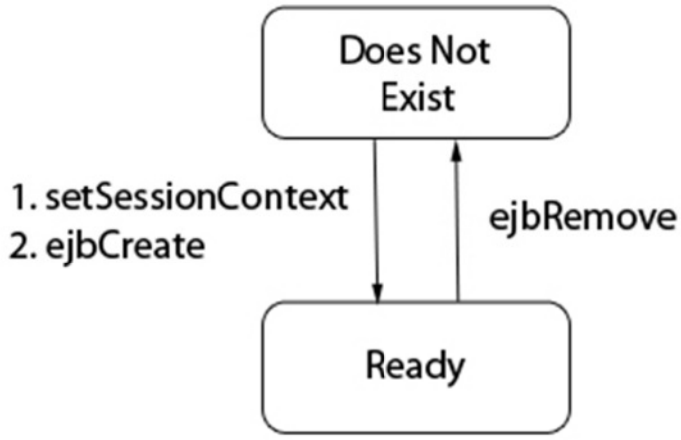
A kliens egyes metódushívásai között az adatok megmaradnak, ezek az adatok a memóriában tárolódnak el. Összekötő elemként szerepelhet a kliens és más alkalmazás komponens között. Akkor érdemes alkalmazni, amikor az üzleti logika a kliens állapotának függvénye. Az állapotmegőrzés miatt a stateful beanek kezelése nehezebb a server számára, mint a stateless session beaneké.



13. ábra *Stateful Session Bean életciklusa*

Stateless session bean

Semmilyen klienssel kapcsolatos információt nem őriz meg a hívások között, csak egy metódus végrehajtásának idejére tud adatokat megőrizni. A stateful session beannél hatékonyabb lehet, web service-t implementálhatunk vele.



15. ábra *Stateless Session Bean életciklusa*

Singleton session bean

Egy alkalmazáson belül csak egy példány létezhet- erre utal a neve is. Az életciklusa az alkalmazás élettartamával azonos, a kliensek konkurens módon használhatják. Főleg az alkalmazás indulásával kapcsolatos inicializálási feladatokat láthat el, de webservice-t is implementálhatunk a segítségével.

Session bean elérése

Egy kliens mindig egy interfész, a business interface, segítségével tud hozzáférni egy session beanhez. Az interfész határozza meg azt, hogy milyen metódusokat érhet el a kliens, a bean többi része a hívó fél részére el van rejtve.

A hozzáférési interfész típusa a következő lehet:

- remote interface
- local interface
- web service

Remote interface

A távoli interfész definiálja a session bean azon üzleti metódusait melyek az EJB konténeren kívüli kliensekből érhetőek el. A távoli kliens számára a keresett bean fizikai holléte lényegtelen. A távoli interfész egy egyszerű Java interfész melyet egy @javax.ejb.Remote annotációval kell ellátni.

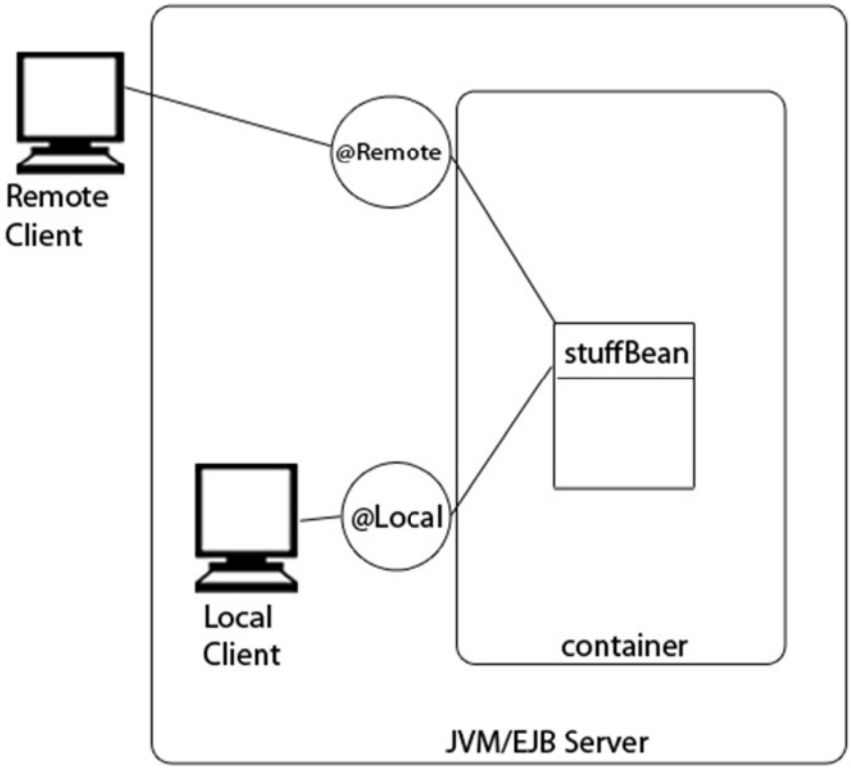
```
@Remote
public interface InterfaceName { ... }
```

Előnyei a mobilitásban, a klienstől való izolációban, a fail-over kezelésben és a skálázhatóságban rejlenek.

Local interface

A lokális interfész definiálja a session bean azon üzleti metódusait melyeket más beanek vagy web komponensek is használhatnak ugyanabban az EJB konténerben. A lokális hívónak ugyanabban a JVM környezetben kell futnia, mint a hívott beannek ahhoz, hogy hozzá tudjon férni. A lokális interfész egy egyszerű Java interfész melyet egy @javax.ejb.Local annotációval kell ellátni. Ha egy interfészhez nem adunk annotációt, akkor az alapértelmezetten lokális lesz.

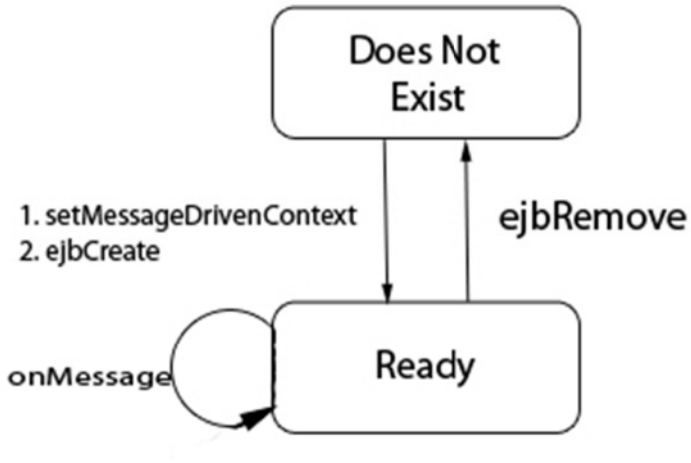
```
@Local
public interface InterfaceName { ... }
```



15. ábra *A Local és a Remote interfészek szemléltetése*

Message-Driven bean

Olyan Enterprise Bean, amely a JMS üzenetek figyelésére képes. Egy kliens üzenet hatására lép működésbe, aszinkron módon működik. Viszonylag rövid életű, nem tartalmaz perzisztens adatokat, de képes adatbázis elérésre, tranzakcióban futhat. Ha egy üzenet érkezik, a konténer meghívja az onMessage metódusát. Az MDB példányok ekvivalensek, ugyanazt a példányt többféle kliens is tudja használni.



16. ábra *Message-Driven Bean életciklusa*

Ejb-jar.xml

Az ejb-jar xml az EJB deployment descriptora. Példa:

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


<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  version="3.0"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd">

  <enterprise-beans>

    <session>

      <ejb-name>BeanName</ejb-name>

    </session>

  </enterprise-beans>

</ejb-jar>
```

Az alkalmazás bemutatása

A benyújtott alkalmazás egy JEE technológiákkal készült roppant egyszerű üzleti logikával rendelkező – csupán felhasználók és azok számláinak kezelésére alkalmas – de emellett óriási továbbfejleszthetőségi potenciállal rendelkező webalkalmazás. Az elsődleges célom az volt, hogy bemutassam, hogy a JEE technológiákkal milyen egyszerű egy komoly vállalati program megírása.

Az alkalmazás által használt eszközök

Apache Tomcat 6

Az Apache Tomcat egy tisztán Java nyelven készült webszerver, amely implementálja a Sun által specifikált Java Servlet és a JavaServer Pages technológiákat. Az Apache Tomcat 6 szerves része a NetBeans 6.9.1 fejlesztőkörnyezetnek, így jelentősen megkönnyíti a Java webfejlesztők munkáját.

MySQL 5.5.8

Az adatbázis réteget az alkalmazás számára a MySQL adatbázis szerver szolgáltatja. Habár nem része a NetBeans fejlesztőkörnyezetnek, egyszerűen telepíthető és konfigurálható. Az InnoDB táblaszerkezet segítségével a tranzakció kezelés is egyszerű.

Hibernate3

A Hibernate egy magas teljesítményű objektum orientált illetve relációs adatmodellek közti leképezésnek a legelterjedtebb eszköze. Hibernate annotációkat illetve egy sajátos lekérdező nyelvet, a Hibernate Query Language –t (HQL) alkalmaz.

Az alkalmazás funkciói

Felhasználók kezelése

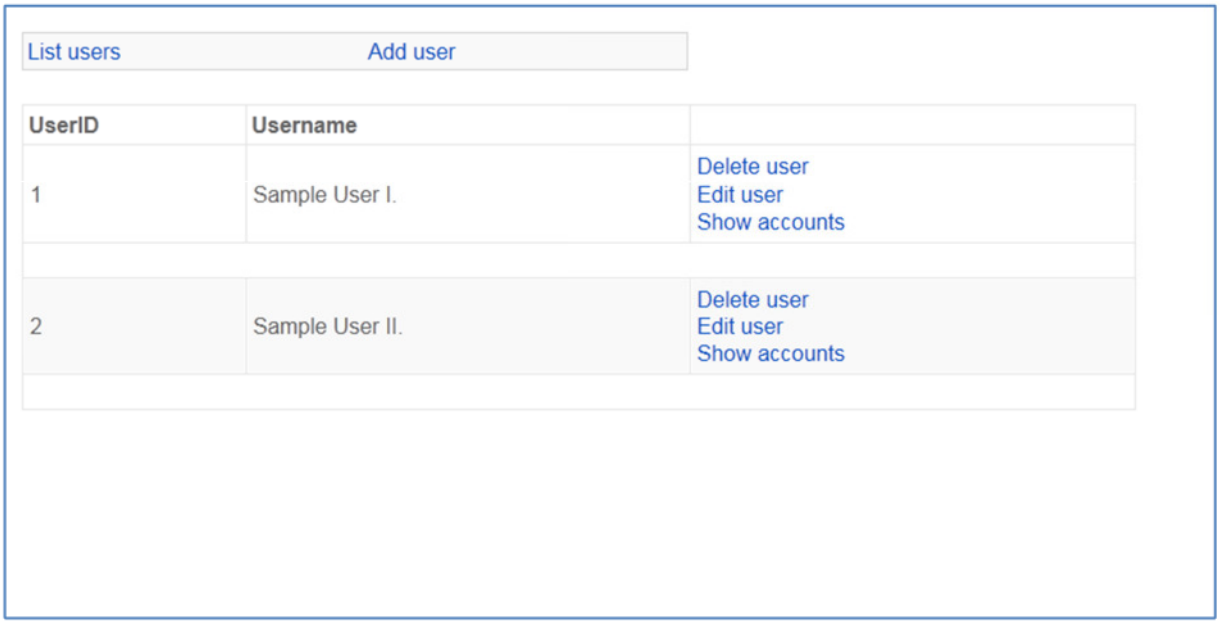
A usereknek saját entitás objektuma van, a User osztály. Ennek az osztálynak a relációs táblareprezentációja a user_table adatbázis tábla. Ezen alkalmazásban a User entitás csumán egy elsődleges kulcsból (id) és egy szöveges értékből áll (name).

Egy-egy userhez tetszőleges számú számla tartozhat.

User entitás:

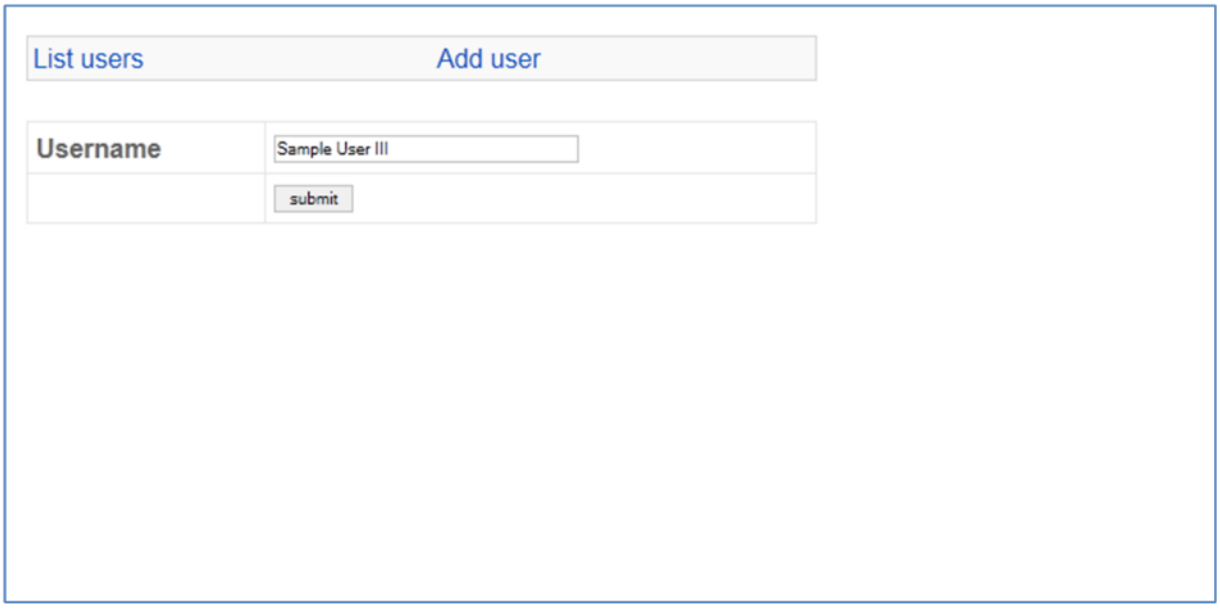
```
@Entity
@Table(name = "user_table")
public class User implements Serializable {


    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private Integer id;


    @OneToMany(mappedBy = "owner")
    private Collection<Account> accounts;
    @Column(name = "name")
    private String name;
…
```

Az alkalmazás lehetővé teszi a CRUD műveletek elvégzését a User entitásokon. A felhasználók listaszerűen jelennek meg, minden egyes listaelem mellett megtalálhatóak a bejegyzés kezelésére alkalmas vezérlők.



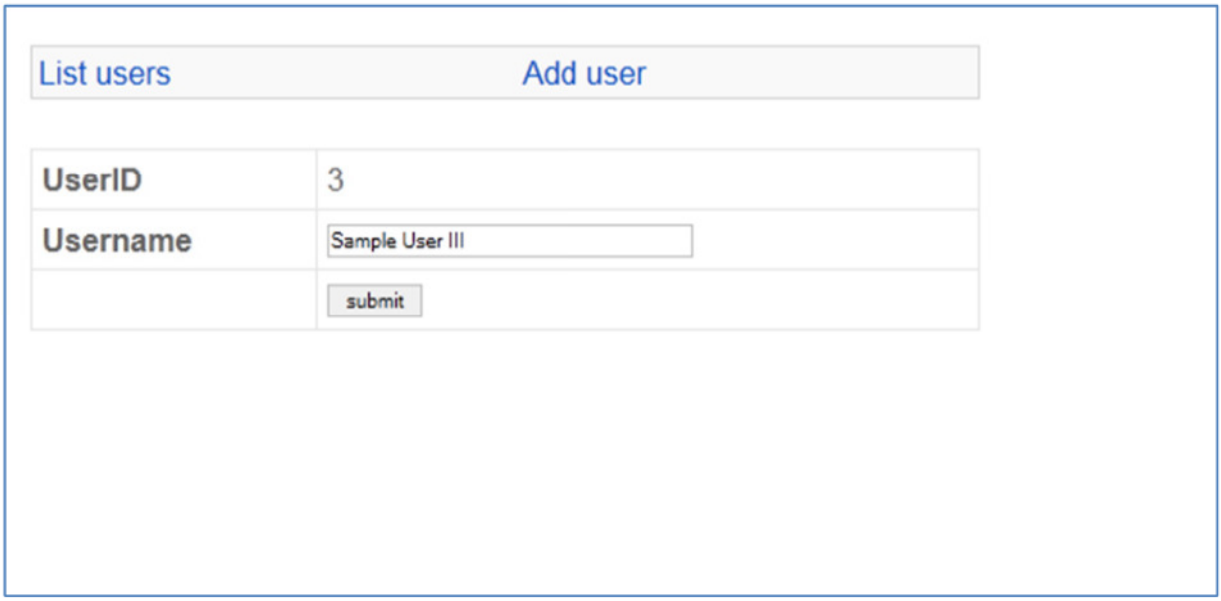
17. ábra *A felhasználók listázása*

Az „Add user” linkre kattintva egyszerűen felvehető egy újabb User entitás:



18. ábra *Új felhasználó felvétele*

Ugyanilyen egyszerű egy meglévő felhasználó adatainak módosítása is:



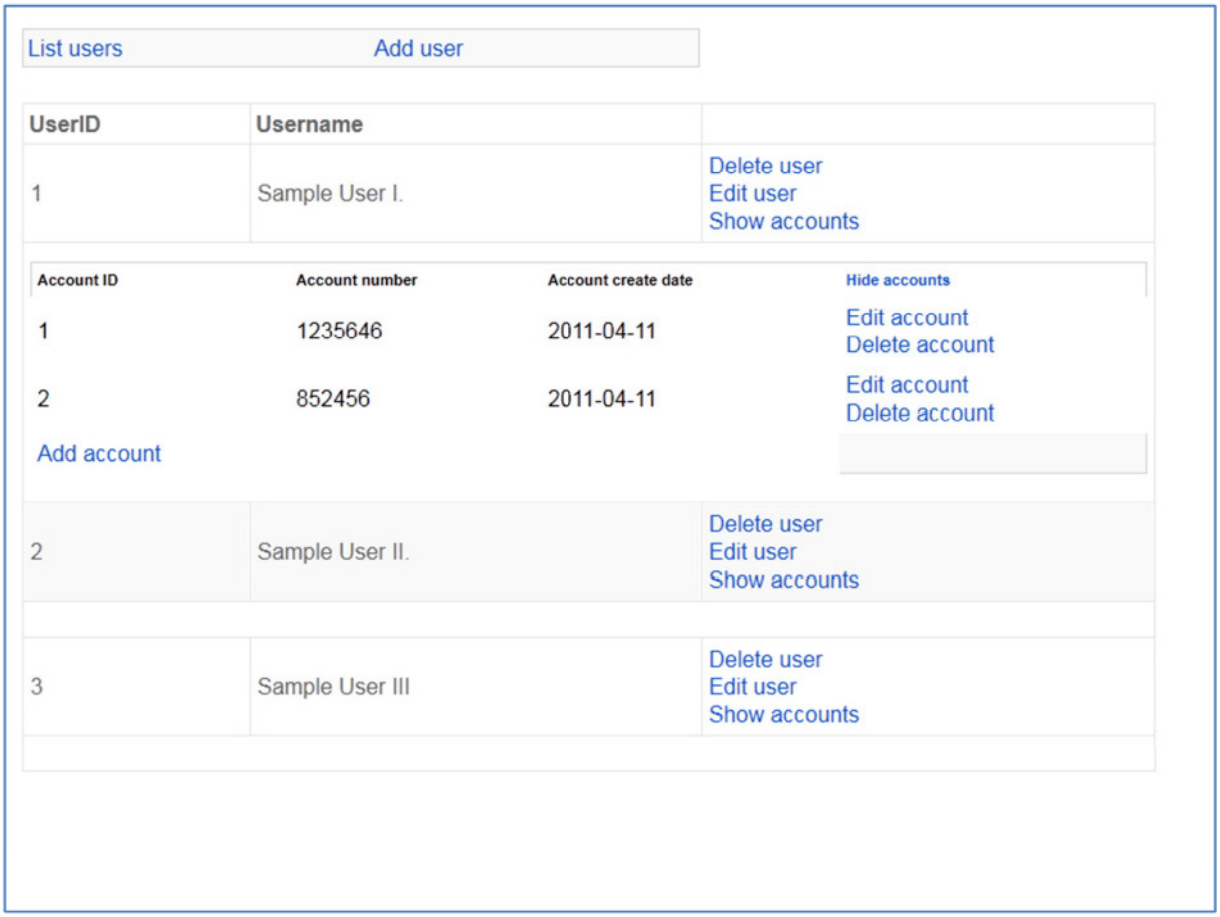
20. ábra *Egy felhasználó módosítása*

Számlák kezelése

Egy Account entitás egy elsődleges kulcsból (id), egy számla számból (number) és egy létrehozási dátumból (cdate) áll. Egy-egy Account entitásnak egy-egy rekord felel meg az account_table adatbázistáblában.

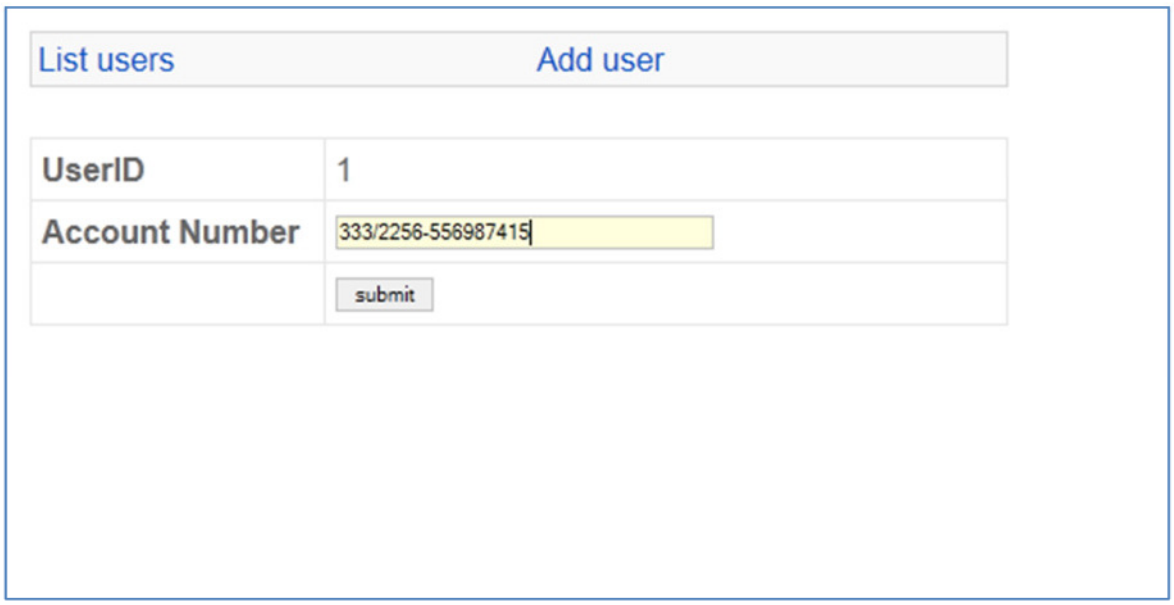
```
@Entity
@Table(name = "account_table")
public class Account implements Serializable {


    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private Integer id;



    @Column(name = "number")
    private String number;


    @Column(name = "cdate")
    private Date cdate;


    @ManyToOne
    @JoinColumn(referencedColumnName = "id", name = "user_id")
    private User owner;
…
```

Egy számla csak egy felhasználóhoz tartózhat. A számlákról szóló információ megjelenik a „Show accounts” linkre kattintva:

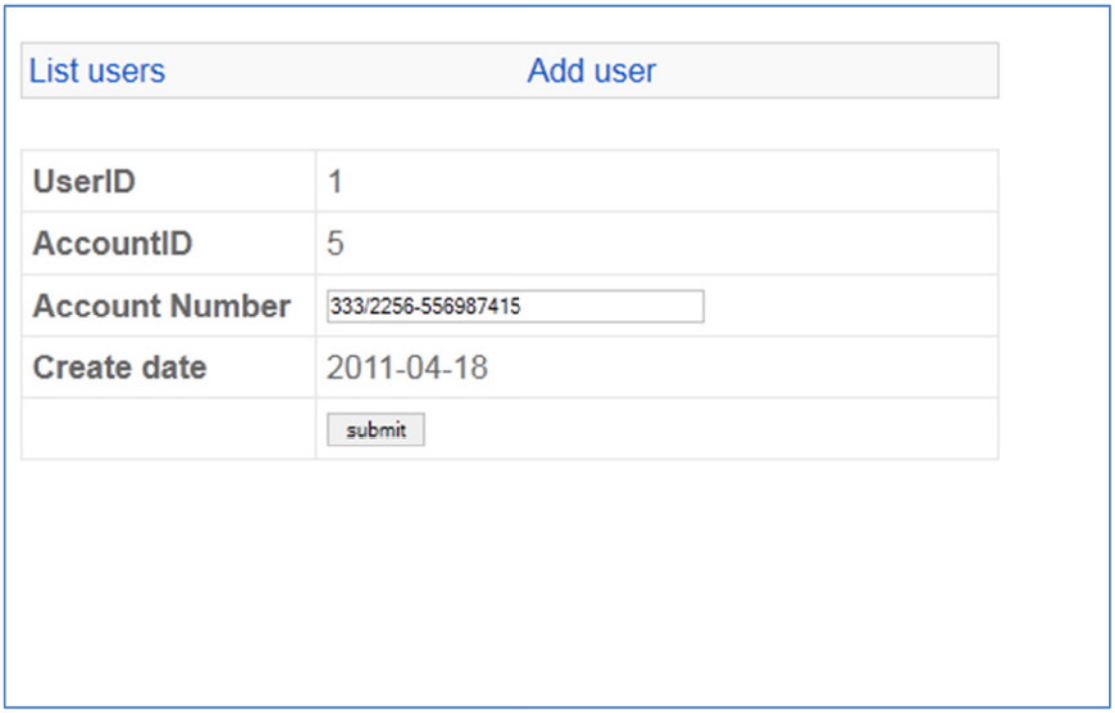


20. ábra *Egy felhasználóhoz tartozó számlainformáció megjeleníttetése*

Egy új számla egyszerűen adható hozzá egy felhasználóhoz:



21. ábra *Egy új számla felvétele*



22. ábra *Egy számla módosítása*

Vezérlésátadás

Az alkalmazásnak két kontrollere van – egy-egy servlet, az egyik az AccountServlet, a másik a UserServlet. Tulajdonképpen ezek végzik el az összes üzleti logikát, létrehozzák, módosítják vagy éppen törlik az adott entitást, a POST paraméterektől függően. A POST paraméterek a JSP oldalakon lévő formok submitálásakor kerülnek át egyes kontrollerekhez.

A servletekben van behúzva a modell réteg – egy-egy entitáskezelő osztály – AccountHandler illetve UserHandler. Egy-egy üzleti logikai lépés után a servlet visszaadja a vezérlést valamely JSP oldalnak.

Így a JSP-Servlet-Entitáskezelő hármassal sikerült implementálnom az MVC tervezési mintát.

Összefoglalás

Jelenleg számos vállalati alkalmazások összeállítására alkalmas technológia van jelen, mint például a Zend vagy a .NET. Ezek közül a Java Enterprise Edition mindenképp az egyik legkézenfekvőbb választás, hiszen számtalan előnnyel rendelkezik a többi architektúrával szemben.

Erőssége főleg a Java nyelvben és a nagyszámú rajongóban rejlik. Szinte minden, vállalati alkalmazási problémával szemben nyújt megoldást vagy a Java EE specifikáció maga, vagy pedig a támogató közössége – fórumok, oktatóanyagok segítségével.

Irodalomjegyzék

EJB & JSP: Java On The Edge, Unlimited Edition, by Lou Marco

Java Enterprise in a Nutshell, 3rd Edition by William Crawford, Jim Farley

JavaServer Pages, Second Edition, by Larne Pekowsky

J2EE Útikalauz Java programozóknak, Nyékyné G. Judit et al.

http://gnu.private.webstar.co.uk/docs/other/websphere/doc/whatis/icsrvlet.html

http://www.oracle.com/technetwork/java/javaee/overview/index.html

http://pallergabor.uw.hu/hu/java-app/EJB.html

Wikipedia, http://en.wikipedia.org

http://en.wikipedia.org/wiki/Java_Message_Service

http://en.wikipedia.org/wiki/Java_Persistence_API

http://en.wikipedia.org/wiki/Session_Beans

RoseIndia, http://www.roseindia.net

Java EJB 3.0 tutorials, http://www.roseindia.net/ejb/

Java EE 5 tutorial, http://www.roseindia.net/java/jee5/

http://web-dot.ru/javaee

http://sbp-program.ru/SBP-JavaEE6.htm

Köszönetnyilvánítás

- Köszönetet szeretnék mondani konzulensemnek, Dr. Fazekas Gábornak, aki elvállalta szakdolgozatom felügyeletét, valamint időt szánt a felmerülő kérdések megvitatására.
- Köszönet illeti Monori István és Zsíros Szabolcs programozókat, akik a szakmai tudásukkal és a szakdolgozat témájában való jártasságukkal segítettek a dolgozat megírásában.
- Köszönöm szüleimnek a támogatást, amit a tanulmányaim alatt kaptam tőlük.