Debreceni Egyetem
Informatikai Kar

Java Enterprise Computing
Vállalati Java

Témavezető:
Dr. Fazekas Gábor
egyetemi docens

Készítette:
Kovács Dániel
programtervező informatikus

Debrecen
2009

Tartalomjegyzék

Tartalomjegyzék.....1

Bevezetés.....2

1. Áttekintés.....2

2. Web Technológiák.....2

    2.1 Szervlet.....3

    2.2 JavaServer Pages (JSP).....5

    2.3 JavaServer Faces (JSF).....6

3. Enterprise Java Beans (EJB).....8

4. Java Perzisztencia.....13

5. Java Database Connectivity (JDBC).....13

6. Eszközök.....14

    6.1 Struts.....14

    6.2 Hibernate.....15

7. Alkalmazásszerverek.....16

8. Példaprogram.....17

9. Összefoglalás.....29

Köszönetnyilvánítás.....30

Irodalomjegyzék.....31

Bevezetés

A fejlesztők manapság egyre jobban felismerik az igényt elosztott, hordozható alkalmazásokra melyek kihasználják a szerver oldali technológia sebességét, biztonságát és megbízhatóságát. Az információ technológia világában a vállalati alkalmazásokat egyre kevesebb pénzből, gyorsabban és kevesebb erőforrásból kell tervezni és készíteni. A Java EE segítségével ezek az igények könnyen és gyorsan megvalósíthatóak.

A szakdolgozat célja egy áttekintő képet adni a Java EE architektúráról, viszont a téma nagy terjedelme miatt nem fedi le az egész architektúrát és az egyes fejezetek sincsenek teljes körűen tárgyalva.

1. Áttekintés

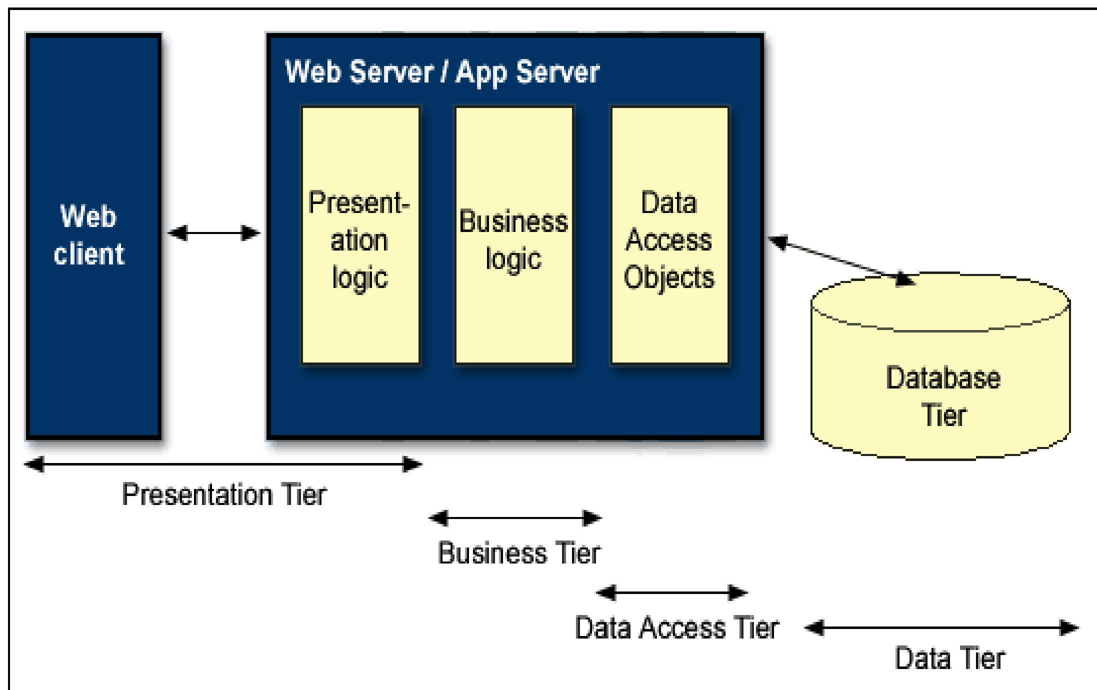
A Java technológia egy objektum-orientált, platform-független, többszálas programozási környezet. Három kiadása létezik: Java SE (Standard Edition), Java ME (Micro Edition) és Java EE (Enterprise Edition). A Java SE hordozható, általános használatú alkalmazások készítésére alkalmas. A Java ME egy robusztus, rugalmas környezetet biztosít mobil telefonon vagy más kicsi, erőforrás-korlátozott, beágyazott eszközökön futó alkalmazások számára. A Java EE egy nyílt, szabvány-alapú fejlesztői és telepítési platform n-rétgeű, szerver-centrikus, web és komponens alapú vállalati alkalmazások számára. Az Enterprise kiadás a Standard kiadásra épül. A Java nyelv szintaktikája és szemantikája ugyanaz bármely kiadás esetén és bárhol használva egy biztonságos, hordozható és robusztus környezetet biztosít.

2. Web Technológiák

A Java EE keretrendszerben a web komponensek dinamikus képességekkel egészítik ki a web szervert. Ezek a komponensek az alkalmazásszerveren belül futnak és kihasználják az általuk nyújtott számos futás idejű szolgáltatást, például: felhasználói session kezelés, biztonsági szolgáltatások mint autentikáció, titkosított kommunikáció, stb... A web komponensek más Java EE erőforrásokhoz és komponensekhez is hozzá tudnak férni úgymint Java Database Connectivity (JDBC), DataSource-ok, helyi és távoli EJB komponensek.

A web komponenseket három szabvány segítségével fejleszthetjük: Java szervletek, Java

Server Pages (JSP) és Java Server Faces (JSF)



Egy web alkalmazás architektúrája

2.1 Szervletek

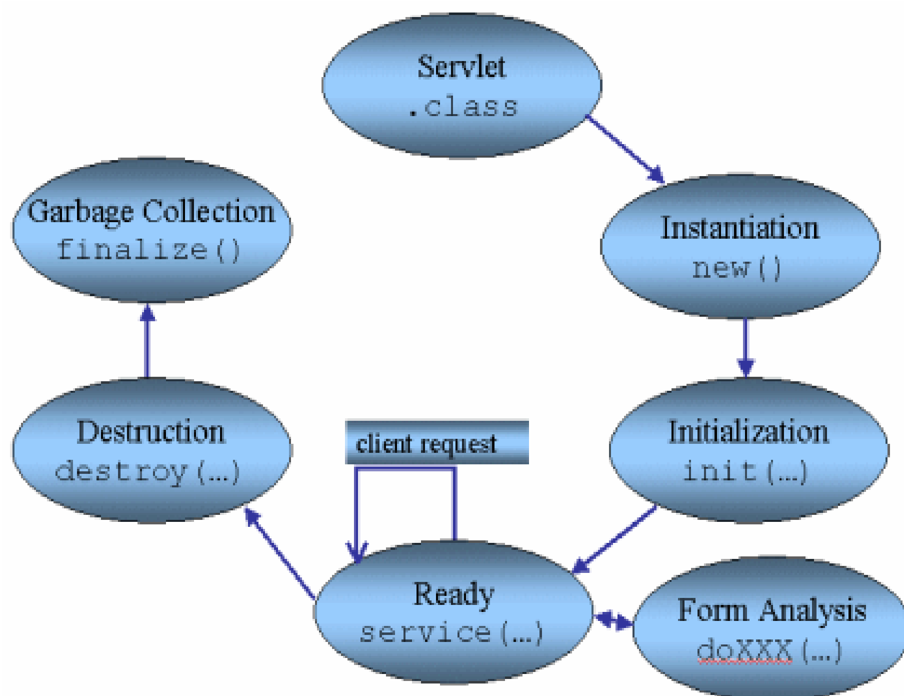
A szervlet egy Java objektum mely kiterjeszti a HTTP szerver funkcionalitását a dinamikus tartalomgenerálás lehetőségével. A szervlet kaphat HTTP kéréseket és nyújthat HTTP válaszokat. A leggyakoribb eset mikor a web böngésző egy kérést küld a web szervernek és Java szervlet ami megkapja a kérést végrehajt valamilyen műveletet (például egy JDBC hívást végez hogy információt kérjen le vagy egy EJB komponenset hív hogy valamilyen üzleti logikát hajtson végre) majd egy HTML választ küld vissza a böngészőnek.

Régebben a fejlesztők hozzá voltak szokva hogy CGI-t használjanak a dinamikus tartalomgeneráláshoz. Ez a megoldás nem volt hatékony mert a kliens minden HTTP kérése egy új folyamatot indított el, még akkor is ha a kérés ugyan attól a klienstől érkezett és ezért nehéz volt skálázni sok klienst egy időben. A Servlet API azért lett kifejlesztve hogy a Java platform előnyeit kihasználva megoldja a CGI és a levédett API-k problémáját. A szervletek

sokkal hatékonyabban kezelik a HTTP kéréseket mint a CGI mert nem kell minden klienstől érkező kérés után új folyamatot indítani hanem az csak egy új szálat indít. A szervlet a memóriában marad a kérések közt ellenben a CGI programot minden kérésnél be kell tölteni és elindítani. A szervlet életciklusát a konténer kezeli amelybe a szervlet telepítve lett. Amikor egy kérést egy szervlethez rendeli, a konténer a következőket hajtja végre:

1. Ha a szervletnek nem létezik példánya
   a. Betölti a szervlet osztályt
   b. A szervlet osztályt példányosítja
   c. Inicializálja a szervlet példányt az *init* metódus hívásával.
2. Meghívja a szolgáltatás metódust és átadja a kérés és válasz objektumokat

Ha a konténernek el kell távolítania egy szervletet akkor meghívja a szervlet *destroy* metódusát.



Egy szervlet életciklusa

Mivel a szervleteket az igen hordozható Java nyelven írják és egy szabvány keretrendszert követnek így azok eszközöket biztosítanak hogy szerver kiterjesztéseket fejleszthessünk szerver és operációs rendszertől függetlenül. A szervlet a legáltalánosabb formájában egy osztály példánya mely implementálja a *javax.servlet.Servlet* interfészt.

2.2 JavaServer Pages (JSP)

A JSP technológia lehetővé teszi hogy könnyen készítsünk webes tartalmakat melynek statikus és dinamikus komponensei is vannak. A JSP elérhetővé teszi a szervletek összes dinamikus képességét de egy termésesebb közelítést alkalmaz a statikus tartalmak készítéséhez.. A dinamikus elemek vagy referenciákkal Java bean-eket hívnak meg egyedi tag-ek (custom tag) segítségével vagy beépített Java kód részleteket alkalmaznak. A JSP egyértelműen elválasztja a megjelenítést az üzleti logika kódjától. Architektúrális szinten nézve a JSP tekinthető a szervletek egy magas szintű absztrakciójának. Egy tipikus gyártási környezetben mind a szervlet mind a JSP az úgynevezett MVC (Model-View-Controller) mintában használatosak. A szervlet kezeli a vezérlő (controller) részt a JSP pedig a megjelenítés (view) részt. A JSP fájlok szervlet kóddá alakulnak amit aztán a konténer automatikusan lefordít. Egy egyszerű JSP oldal:
(Nem aláhúzott rész: statikus tartalom, Aláhúzott rész: dinamikus tartalom)

```
<html>
    <body>
        Hello Világ!
        <br>
        A mai dátum <%= new java.util.Date() %>
    </body>
</html>
```

A JSP 2.0 bevezeti az Expression Language-t (EL) melyet azért terveztek hogy az oldal készítői könnyebben tudjanak logikai teszteket alkalmazni. A kifejezéseket amiket EL-ben írunk be lehet ágyazni statikus szövegbe vagy át lehet őket adni JSP tag-ek paramétereiben. Mindkét esetben a JSP motor kiértékeli a kifejezést és beágyazza az eredményt az oldalba. Az EL kifejezések $ jellel kezdődnek és {} zárójelek közt vannak. Itt egy egyszerű példa mely kiértékeli a 11 > 10 kifejezést, majd összeadja 5-öt és 6-ot:

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
```

```
    <body>
        11 nagyobb 10-nél? ${11 > 10}
        5 és 6 összege:  ${5+6}
    </body>
</html>
```

A JSP 2.1 legfőbb újdonsága a Unified Expression Language (unified EL) mely a JSP 2.0 által nyújtott expression language és a JSF technológia által használt expression language unióját képviseli.
A JSP szintaktika további XML-szerű tag-eket vezet be melyeket JSP action-nek hívnak. Ezekkel beépített funkciókat lehet meghívni. Továbbá a technológia lehetővé teszi JSP tag könyvtárak (tag library) létrehozását ami az alap HTML és XML tag-ek kiterjesztéseként viselkednek. A tag könyvtárak platform független módon terjesztik ki a web szerverek képességeit. A JSTL (JavaServer Pages Standard Tag Library) a gyakran használt funkciók egy szabvány halmaza. A JSTL-nek vannak tag-jei általános strukturális feladatok elvégzésére mint iteráció és feltételek, tag-ek XML dokumentumok manipulálására, többnyelvűséget lehetővé tevő tag-ek és adatbázishoz SQL használatával való hozzáférést lehetővé tevő tag-ek. Hogy a JSTL tag-ek elérhetővé váljanak a web alkalmazásban egy névtérhez kell rendelni őket minden JSP-ben. Ezt egyszerűen meg lehet tenni ha a következős sort beillesztjük minden oldal elejére:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

Konvencionálisan a JSTL tag-eknek c prefixet adunk.

2.3 JavaServer Faces (JSF)

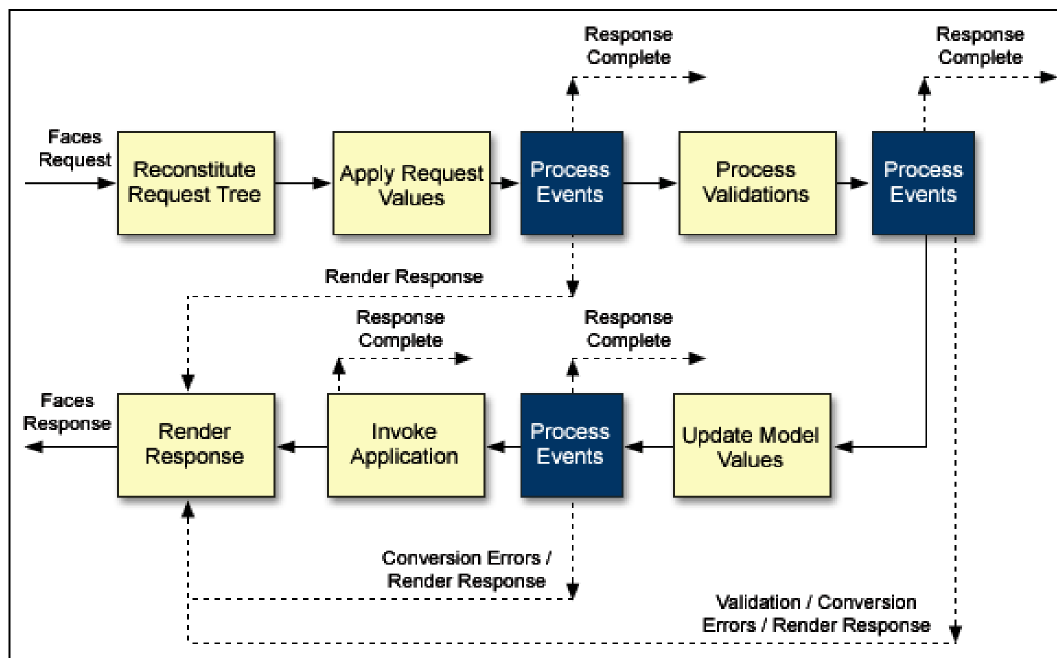
A JavaServer Faces API egy szabvány keretrendszert biztosít web alkalmazások megjelenítési rétegének készítésére. A JSF egy halom előre megírt felhasználói interfész (user interface – UI) komponenst ad a felhasználóknak és egy esemény modellt ami lehetővé teszi hogy kialakítsunk interakciót az UI és az alkalmazás alatt fekvő objektumok közt. Az UI komponensek állapota eltárolódik mikor a kliens egy új oldalt kér és a kérés visszaérkezésekor visszatöltődik.

A JSF tartalmaz:

- Egy API halmazt UI komponensek reprezentációjához és állapotuk kezeléséhez, esemény kezeléséhez és input validációhoz, az oldal navigálásának meghatározásához és a többnyelvűség támogatásához
- Egy JSP egyedi tag könyvtárt (custom tag library) JSF interfészek használatához egy JSP oldalon belül.

A jól definiált programozási modell és a tag könyvtárak jelentősen megkönnyítik a létrehozását és karbantartását a web alkalmazásoknak melyek szerver oldali UI-t használnak. Minimális erőfeszítéssel tehetjük következőket:

- Komponenseket adhatunk egy oldalhoz komponens tag-ek hozzáadásával
- A komponens generálta eseményeket összeköthetjük a szerver oldali kóddal
- A oldalon lévő komponenseket összeköthetjük a szerver oldali adatokkal
- Újrafelhasználható és kiterjeszthető komponensekkel UI-t konstruálhatunk
- Elmenthetjük és visszaállíthatjuk az UI állapotát



A JSF életciklusa

3. Enterprise Java Beans (EJB)

Az enterprise bean-ek szerver oldali komponensek melyek egy alkalmazás üzleti logikáját tartalmazzák. Egy EJB komponens JavaBean komponens is egyben mivel JavaBean konvenciókat használ hogy definiálja a tulajdonságaihoz hozzáférő metódusokat. Az EJB architektúra egy keretrendszert biztosít melyben a fejlesztő könnyen kihasználhatja az EJB konténer által biztosított lehetőségeket mint tranzakció feldolgozás, biztonság, perzisztencia, stb...

Egy EJB komponens (enterprise bean) a következő dolgokat teheti:

- Üzleti logikát hajt végre. Például kiszámolja az adót a bevásárlókorárban lévő termékekre, gondoskodik róla hogy csak az arra jogosultak hagyhassák jóvá a rendelést vagy email küldése a vásárlás megerősítéséről.
- Adatbázishoz való hozzáférés. Például egy könyv megrendelésének rögzítése, pénz átutalása két bankszámla közt vagy tárolt eljárások hívása. Az enterprise bean-ek különböző technikákkal férhetnek hozzá az adatbázisokhoz melyek közül az egyik a JDBC (Java Database Connectivity) API.
- Más rendszerekkel való egyesítés. Az enterprise bean-ek különféle módon egyesülhetnek más alkalmazásokkal melyek közül az egyik a Java EE Connector Architecture.

Az EJB két fő típusa:

- Session bean
- Üzenet-vezérelt bean (message-driven bean)

Session bean-ek:

Akkor hasznosak ha más bean-ekkel való interakciót szeretnénk leírni vagy implementálni bizonyos feladatokat. Munkamenetek, folyamatok vagy feladatok modellezésére használjuk vagy valamilyen tevékenység irányítására (foglalás, vásárlás, …) Nem jelennek meg az adatbázisban de használhatjuk őket arra hogy olvassunk, frissítsünk vagy beszúrjunk adatokat egy üzleti folyamatba. A fő különbség a session bean-ek és más bean-ek közt az élettartama. Egy session bean példány viszonylag rövid életű objektum. Nagyjából egyenlő az élettartama

egy session-el vagy a kliens kóddal ami meghívta a bean-t. A session bean példányok nincsenek megosztva a különböző kliensek közt. Minden enterprise bean párbeszédet folytat a kliensekkel valamilyen szinten. A párbeszéd egy interakció a kliens és a bean között. Kétfajta session bean létezik: állapottal rendelkező (stateful) és állapotmentes (stateless)

Állapottal rendelkező session bean (stateful session beans):

Ez a fajta bean fenntart egy kliens specifikus állapotot (session state) az egyik metódus hívástól a következőig. A bean példány változóinak az értéke megmarad ugyan azon kliens hívásai közt. Bean példányok vannak fenntartva minden klienshez amik a kliens kérésére jönnek létre és utána lesznek eltávolítva. A kliens lehet aktív vagy inaktív állapotban. Az állapot nem marad meg szerver meghibásodás esetén. Felső kategóriás kereskedelmi szerverek fenntarthatják a kliens állapotát szerver meghibásodás esetén is úgy hogy a szerver állapotát perzisztensen tárolják vagy ugyan azt az állapotot több szerveren is tárolják.

Állapotmentes session bean (stateless session beans):

Az állapotmentes session bean egy olyan bean mely a párbeszédet egyetlen metódus hívás idejére tartja fenn. Nem őrzi meg a kliens specifikus állapotot az egyik metódus hívástól a másikig. A bean példányt újra ki lehet osztani hogy egy másik klienst szolgáljon ki ha az aktuális metódushívás befejeződött. A terhelés elosztás és a hiba esetén átkapcsolás EJB szerverek közt könnyebben megvalósítható mivel nem szükséges állapotot megőrizni. Bármely bean példány bármely EJB szerveren bármely kliens hívást ki tud szolgálni. Magas a skálázhatóság, hogy kezelni tudjuk a kliensek megnövekedett számát csak több memóriát vagy több EJB szervert kell beiktatni.

Üzenet-vezérelt bean-ek (message-driven beans):

Az üzenet vezérelt bean-eket az EJB 2.0 vezette be hogy támogassa az aszinkron üzenetek feldolgozását. Úgy működik mint egy JMS (Java Message Service) üzenet figyelő ami hasonló egy esemény figyelőhöz kivéve hogy az JMS üzeneteket kap események helyett. A futás idejű életciklusát és a konténer kezelését tekintve az üzenet-vezérelt bean-ek természete az állapotmentes session bean-ekéhez van a legközelebb. A legfeltűnőbb különbség az üzenet-vezérelt bean-ek és a session bean-ek közt hogy a kliensek nem interfészeken keresztül férnek

hozzá az üzenet-vezérelt bean-ekhez. Mivel indirekt és aszinkron módon hívjuk őket így nem tartják fenn a párbeszéd állapotát és a konténer összegyűjtheti őket hogy a bejövő üzeneteket kezeljék. Az EJB 3.0 nem igazán terjeszti ki az előző változatok képességeit de a konfigurációt egyszerűbbé teszi az annotációk használatával.

Kliens hozzáférés interfészek segítségével

Egy kliens egy session bean-hez csak a bean üzleti interfészében található metódusok segítségével tud hozzáférni. (Az üzenet-vezérelt bean-ek nem interfészek segítségével valósítják meg a kliens hozzáférést) Az üzleti interfész definiálja hogy a kliens mit lát a bean-ből, a bean többi része pedig el van rejtve a kliens elől. A jól megtervezett interfészek leegyszerűsítik a fejlesztését és a karbantartását egy Java EE alkalmazásnak. Az interfészek alkalmazása azért is jó mert ha valamely metódus implementációját megváltoztatjuk de a specifikáció változatlan marad akkor a kliens kódját nem kell átírnunk. A sesion bean-eknek több üzleti interfészük is lehet.

Mielőtt elkezdjük fejleszteni a bean-jeinket el kell döntenünk hogy milyen fajta kliens hozzáférést biztosítunk hozzájuk. Ezek a következőek lehetnek: távoli (remote), lokális (local) vagy web szolgáltatás (web service).

Távoli interfész (Remote interface):

A távoli interfész definiálja a session bean azon üzleti metódusait melyek az EJB konténeren kívüli alkalmazásokból érhetőek el. A távoli kliens lehet egy web komponens, egy alkalmazás vagy más bean. A távoli kliens számára az enterprise bean holléte lényegtelen. A távoli interfész egy egyszerű Java interfész melyet egy *@javax.ejb.Remote* annotációval kell ellátni.

@Remote
public interface InterfaceName { ... }

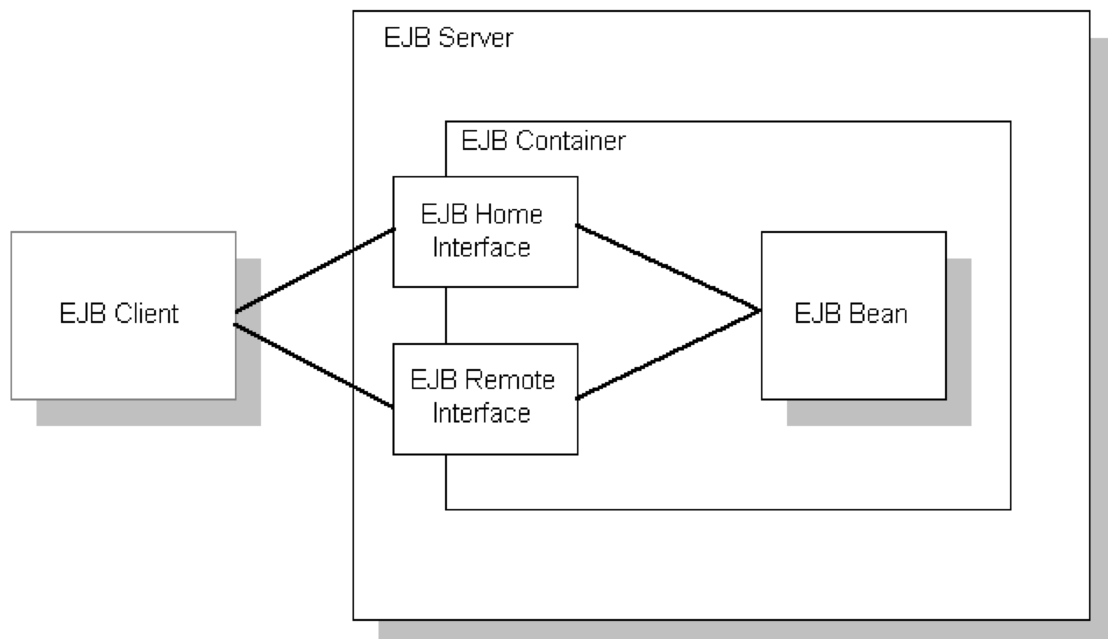
Lokális interfész (Local interface):

A lokális interfész definiálja a session bean azon üzleti metódusait melyeket más bean-ek vagy web komponensek használhatnak ugyan abban az EJB konténerben. A lokális kliensnek ugyan abban a JVM-ben kell futnia mint az enterprise bean-nek 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 { ... }

A bean osztálynak lennie kell legalább egy lokális vagy távoli interfészének. A bean általában implementálja ezeket az interfészeket, de ez nem kötelező. Az EJB konténer általában meghatározza hogy a session bean távoli vagy lokális-e abból kiindulva hogy milyen interfészt implementál. A session bean osztályunkat még el kell látni egy *@javax.ejb.Stateful* vagy *@javax.ejb.Stateless* annotációval attól függően hogy milyen típusú a bean.

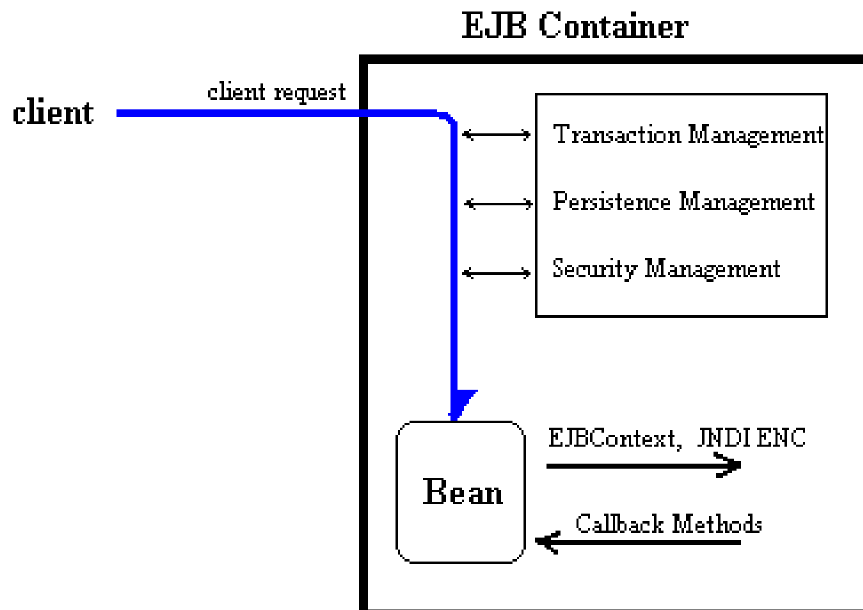


Üzenet-vezérelt bean-ek:

Az üzenet-vezérelt bean-ek egy üzenet interfészt (message interface) implementálnak melyek azokat a metódusokat definiálják melyeken keresztül egy üzenő rendszer (messaging system) - mint a JMS - üzeneteket tud küldeni a bean-nek. Az üzenet-vezérelt bean-ek egy vagy több üzenet kézbesítő metódust implementálnak (pl. onMessage()). A konténer ezeket a metódusokat hívja meg ha új üzenet érkezik. A bean osztályt egy *@javax.ejb.MessageDriven* annotációval kell ellátni.

EJB Konténer:

Az EJB konténer egy szoftver ami implementálja az EJB specifikációt. Azért hívják konténernek mert egy olyan környezetet biztosít melyben az EJB komponensek léteznek. A bean példányok itt jönnek létre és konténer biztosítja a távoli és helyi elérésüket.



EJB Containers manage enterprise beans at runtime

Entitás bean-ek (entity beans):

Az EJB 3.0 specifikáció előtt az entitás bean-ek egy objektum orientált modellt biztosítottak mellyel a fejlesztők könnyebben tudtak létrehozni, módosítani vagy törölni adatot egy adatbázisból. Az entitás bean egy adatbázisban vagy más perzisztens tárhelyen lévő adatot reprezentál.
Az EJB 3.0 specifikácóval egy teljesen új perzisztencia technológiát tudunk használni amit Java Persistence-nek neveznek.

4. Java Perzisztencia (Java Persistence)

A perzisztencia kulcsfontosságú a Java EE platformban. A Java Persistence API egy objektum/relációs leképezési technikát nyújt a Java fejlesztőknek hogy kezelni tudják a relációs adataikat Java alkalmazásokban. A Java Persistence külön lett választva az alap EJB specifikációtól. Három területből áll:

- The Java Persistence API
- A lekérdező nyelv
- Objektum/relációs metaadatok

Az entitás egy perzisztencia objektum. Az entitások nem az előző verziókból ismert entitás bean-ek továbbfejlesztései hanem inkább egy teljesen új programozási koncepció. Tipikusan egy entitás egy táblát reprezentál egy relációs adatbázisban és minden entitás példány megfelel egy sornak a táblában. Az entitások általában kapcsolatban vannak más entitásokkal és ezeket a kapcsolatokat az objektum/relációs metaadatok fejezik ki. Ezeket a metaadatokat közvetlenül beleírhatjuk az entitás osztály forrásfájljába annotációk segítségével vagy külön XML leíró fájlt használva.

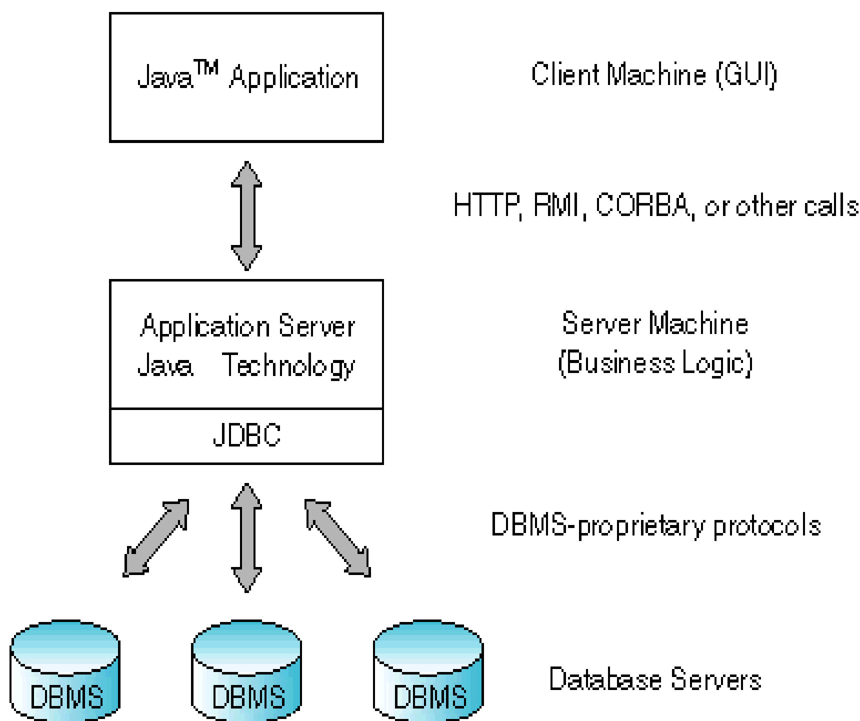
Nem szükséges EJB konténer vagy Java EE alkalmazásszerver hogy futtassuk azokat az alkalmazásokat melyek perzisztenciát alkalmaznak. A Java persistence API-t használhatjuk a Java SE kiadásban, weben és EJB alkalmazásokban egyaránt.

5. Java Database Connectivity (JDBC)

A JDBC egy szabvány Java API relációs adatbázisokhoz való hozzáféréséhez mely elrejti az adatbázis specifikus részleteket az alkalmazás elől. A JDBC tartalmaz alapvető funkciókat adatbázishoz való hozzáféréshez és általában az SQL egy bizonyos részhalmazát. A JDBC API interfészeket definiál melyek főbb adatbázis funkcionalitásokat zár magába mint lekérdezések futtatása, eredmények feldolgozása és konfigurációs információk meghatározása. Egy adatbázis forgalmazó vagy egy harmadik fél írhat egy JDBC driver-t mely olyan osztályok halmaza amely megvalósítja ezeket az interfészeket az adott adatbázis vonatkozásában. Egy alkalmazás több driver-t is használhat.

A JDBC API legnagyobb része a *java.sql* csomagban található (pl. DriverManager,

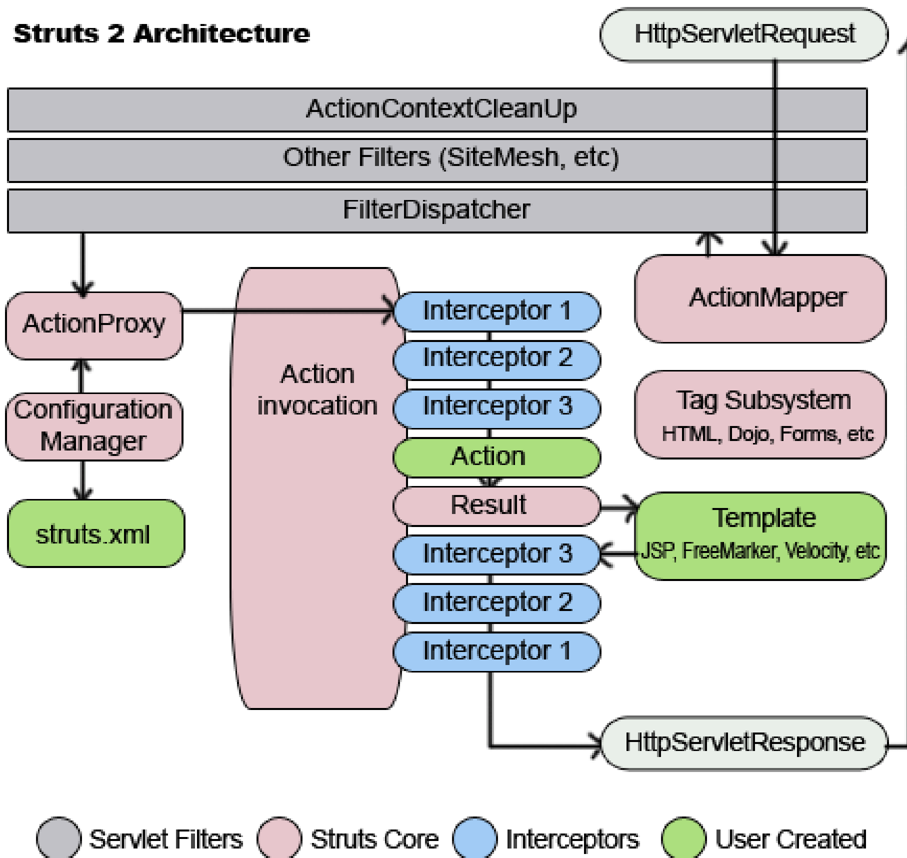
Connection, ResultSet, DatabaseMetaData, ResultSetMetaData, PreparedStatement). Más haladó funkcionalitás a *javax.sql* csomagban található (pl. DataSource).



6. Eszközök

6.1 Struts

A Struts egy nyílt forráskódú keretrendszer mellyel Java EE webalkalmazások készíthetők. A keretrendszer a web alkalmazások készítéséhez szükséges infrastruktúra nagy részét biztosítja. Rábízhatjuk magunkat a szilárd tervezési alapelvek mentén felépített, tesztelt kódra ahelyett hogy hogy saját megoldásunkat kellene megvalósítani. A Struts a Model-View-Controller (MVC) tervezési minta köré épült amely szétválasztja a webalkalmazás különböző részeit, így a kód karbantarthatóbbá válik. A Struts a web alkalmazásokat szervletek és JSP segítségével építi fel. Többféle mintát felhasznál: singleton, composition view, delegate.
Az Apache Struts 2 eredetileg WebWork 2 néven volt ismert. Néhány évig függetlenül dolgoztak majd a WebWork és Struts közösség összefogott hogy megalkossa a Struts2-őt. Ez a Struts új változata melyet egyszerűbben lehet használni.



6.2 Hibernate

A Hibernate objektum-relációs leképező (object-relational mapping - ORM) keretrendszer amely lehetővé teszi hogy a relációs adatainkat objektum-centrikus módon kezeljük. A Hibernate egy nyílt forráskodú szoftver mely a GNU Lesser General Public License alatt terjeszthető.

A Hibernate leképező stratégiájának köszönhetően az objektum-orientált sémánkat átkonvertálhatjuk adatbázis sémára és vissza. Tartalmaz még egy SQL-hez hasonló nyelvet: Hibernate Query Language (HQL). Ezzel SQL-szerű lekérdezéseket írhatunk a Hibernate adatobjektumai használatára. A Java osztályok adatbázis táblákhoz való rendelése egy XML fájl vagy Java annotációk használatával történik. Amikor XML fájlt használunk, a Hibernate képes generálni forráskód vázat a perzisztens osztályokhoz. Annotációk használatával ez szükségtelen.

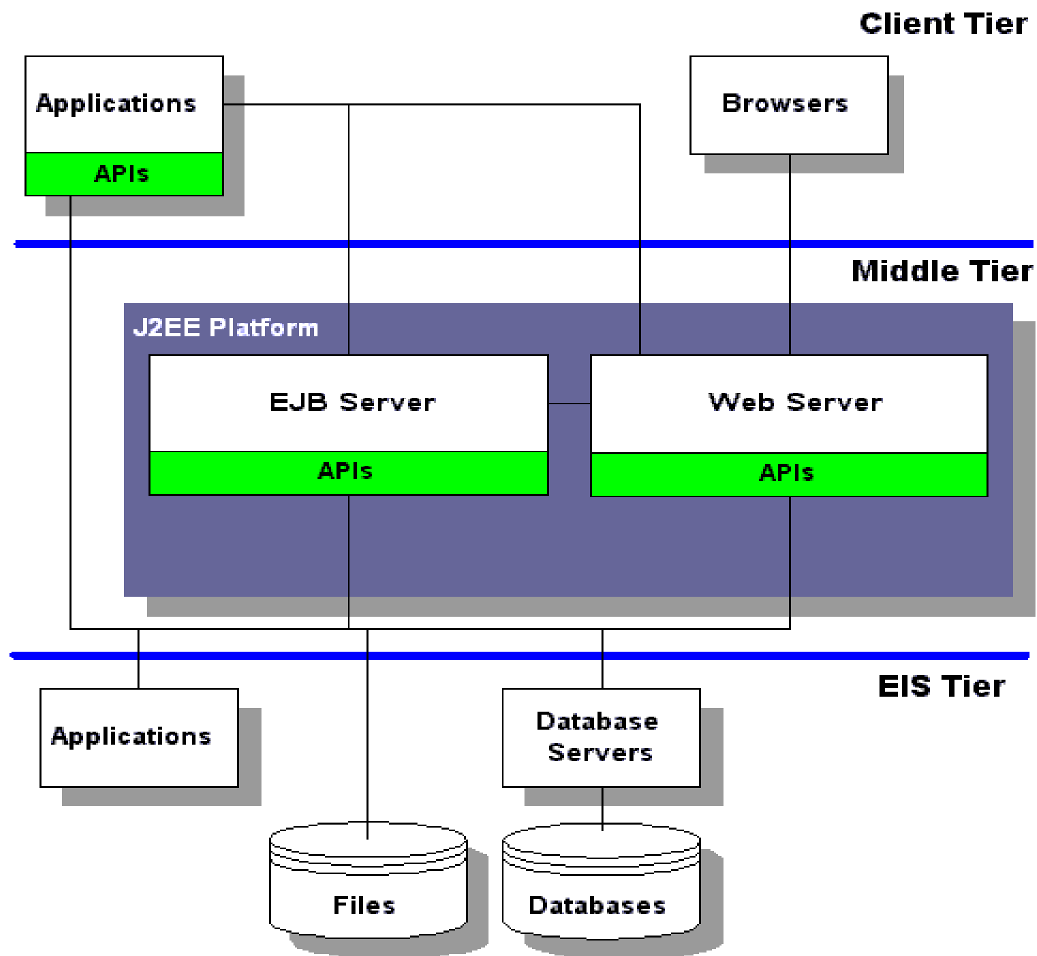
7. Alkalmazásszerverek

Az alkalmazásszerver egy komponens alapú termék mely a szerver központú architektúra középső rétegében helyezkedik el.

A Java alkalmazásszerverek a Java EE-ön alapulnak. A Java EE egy három rétegű, elosztott modellt alkalmaz. Ez a modell általában magában foglal egy kliens réteget, egy középső réteget és egy EIS (Enterprise Information System) réteget. A kliens réteg lehet egy vagy több alkalmazás vagy böngésző. A Java EE a középső rétegben helyezkedik el és egy Web szerverből és egy EJB szerverből áll. (Ezeket a szervereket konténereknek is nevezik). További alrétegek is lehetnek a középső rétegben. Az EIS rétegben találhatóak a létező alkalmazások, fájlok és adatbázisok. Az adatbázisokhoz hozzáférni a JDBC, SQLJ vagy JDO API segítségével tudunk.

Az ismertebb Java EE alkalmazásszerverek a következőek [7.3]:

- WebSphere Application Server (IBM)
- Sybase Enterprise Application Server (Sybase Inc)
- WebLogic Server (BEA)
- JBoss (Red Hat)
- JRun (Adobe Systems)
- Apache Geronimo (Apache Software Foundation)
- Oracle OC4J (Oracle Corporation)
- Sun Java System Application Server (GlassFish Application Server-en alapulva) (Sun Microsystems)
- Glassfish Application Server.
- SAP Netweaver AS (ABAP/Java) (SAP)



8. Példa Alkalmazás

A példa alkalmazásom alapjául a JavaEE tutorial-ban szereplő Duke's Bank nevű alkalmazás szolgál [4]. Annak a szolgáltatásait, funkcióit bővítettem ki. A fejlesztéshez NetBeans 6.5 IDE-t, Glassfish V2 alkalmazásszervert és Derby adatbázis-kezelő rendszert használtam. Az általam készített kiegészítések a következőek:

1. Kiegészítés:

A kliens alkalmazásban az AccountInfo panelen beszúrtam egy táblázatot mely az adatbázisban levő összes Account-nak kiírja az Id, Type, Description és Balance tulajdonságát. Ha kiválasztjuk a megfelelő sort akkor az Open gombbal meg is lehet nyitni az Account-ot.

File Edit

Customer Info | AccountInfo

Account Id: Open

Account Description:

Account Type: Savings

Balance:

Credit:

Beginning Balance:

Customers:

Account Created:

New Account | Create Account | Remove | Cancel

Account Actions

Customer Id: | Add Customer to Account | Remove Customer From Account

Account Id	Type	Desription	Balance
5005	Money Market	Hi Balance	1800.00
5006	Checking	Checking	2458.32
5007	Credit	Visa	-279.97
5008	Savings	Super Interest Account	59601.35

Open

Messages

Megvalósítás:

Az Account entitásban létrehoztam egy NamedQuery-t mely lekéri az Account táblából az összes rekordot:

```
@NamedQuery(name = "Account.GetAll", query = "SELECT a FROM
Account a")
```

Majd az AccountControllerBean nevű enterprise beanben írtam egy metódust mely meghívja az előző NamedQuery-t és az Account-ok listájával visszatér :

```
public List<AccountDetails> getAccounts() {
//Visszatér az összes Account-tal
     Debug.print("AccountControllerBean getAccounts");

     Collection accounts = null;
     try {
          accounts =
          em.createNamedQuery("Account.GetAll").getResultList();
     } catch (Exception ex) {
          throw new EJBException(ex);
     }
return copyAccountsToDetails(accounts);
}
```

A copyAccountsToDetails metódus az Account-ok listáját átalakítja egy AccountDetails segédosztály példányiból álló listára. Ez a feldolgozást segíti.
A kliens alkalmazás egy osztályból áll: BankAdmin. Ide be van injektálva az AccountControllerBean:

```
@EJB
private static AccountController accountController;
```

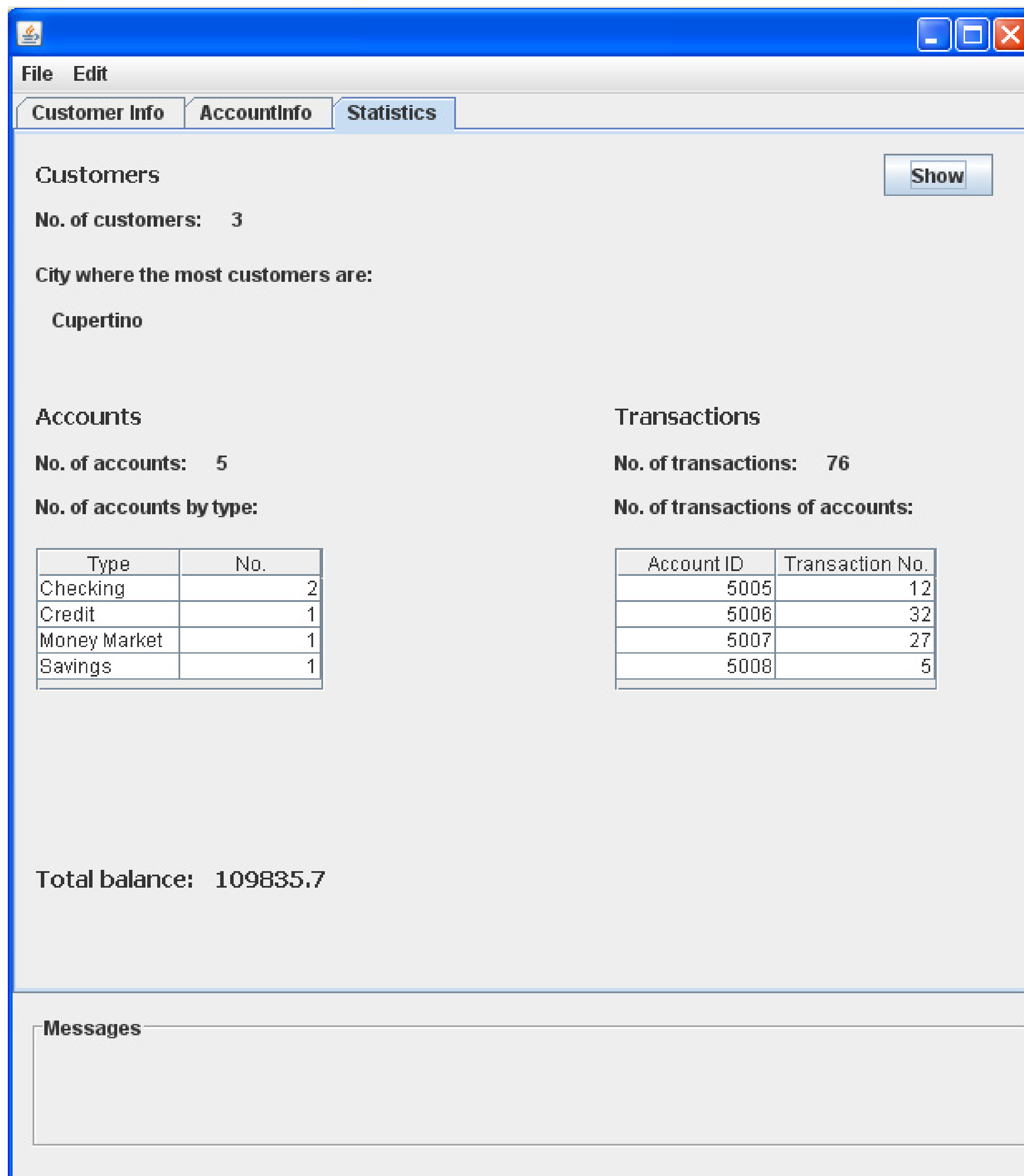
Beszúrtam egy JTable táblázatot melyhez saját tábla modellt írtam beépített osztályként:

```
class MyTableModel extends AbstractTableModel{
     List<AccountDetails> acc_det = null;
     private String[] columnNames = {"Account Id",
                                     "Type",
```

```
                                        "Desription",
                                        "Balance"};
    public MyTableModel() {
        this.getData();
    }
    public void getData(){
        acc_det = accountController.getAccounts();
    }
    @Override
    public String getColumnName(int col) { return
        columnNames[col];}
    public int getRowCount() { return acc_det.size(); }
    public int getColumnCount() {return columnNames.length;}
    public Object getValueAt(int row, int col) {
        switch (col){
        case 0: return acc_det.get(row).getAccountId();
        case 1: return acc_det.get(row).getType();
        case 2: return acc_det.get(row).getDescription();
        case 3: return acc_det.get(row).getBalance();
        default: return null;
        }
    }
}
```

2. Kiegészítés:

A kliens alkalmazásban létrehoztam egy új fület Statistics néven ahol különféle statisztikai adatokat lehet megtekinteni.



A 'Show' gomb megnyomására jelennek meg a statisztikák. A 'Customers' címke alatt a 'No. of customers' után láthatjuk hogy mennyi ügyfél van a összesen rendszerben. A 'City where the most customers are' felirat alatt láthatjuk azt a várost ahol a legtöbb ügyfél található.

Az 'Accounts' címke alatt a 'No. of accounts' után látható hogy összesen mennyi számla van nyitva a bankban. Alatta pedig egy táblázat található ami az egyes számlatípusok számát

mutatja.

A 'Transactions' címke alat a 'No. of transactions' után látható hogy eddig összesen mennyi tranzakció ment végbe. Alatta egy táblázat mutatja hogy az egyes számlákhoz kapcsolódóan mennyi tranzakció történt eddig. A 'Total balance' utám láthatjuk a bank egyenlegét.

Megvalósítás:

Létrehoztam egy enterprise bean-t amely a különböző statisztikákat számolja. Ez a StatisticsCounterBean. Ehhez a távoli interfész melyen keresztül elérhetőek a szolgáltatásai a következő:

```
@Remote
public interface StatisticsCounter {
    int customersNumber();
    String mostCustomerCity();
    int accountsNumber();
    int transactionsNumber();
    double totalBalance();
    Map<String, Integer> noOfAccountsByType();
    Map<Long, Integer> noOfTransactionsOfAccounts();
    public void init();
}
```

A 'customersNumber' nevű metódus a összes ügyfél számát adja vissza. A 'mostCustomerCity' azzal a várossal tér vissza melyben a legtöbb ügyfél található. Az 'accountsNumber' a bankban nyitott összes számla számát adja meg, a 'transactionsNumber' pedig az eddig végbement összes tranzakció számát. A 'totalBalance' a bank egyenlegét mutatja, vagyis az összes számlán található pozitív vagy negatív pénzmennyiség összege. A 'noOfAccountsByType' metódus egy olyan Map-el tér vissza melyben a kulcs az egyes számlatípusok neve, az érték pedig hogy a számlatípusokból mennyi van létrehozva. A 'noOfTransactionsOfAccounts' szintén egy Map-el tér vissza melyben a kulcs a számlák elsődleges kulcsa, az érték pedig azt mutatja hogy az adott elsődleges kulcsú számlához kapcsolódóan mennyi tranzakció történt. Az 'init' metódus lekéri az összes számlát, ügyfelet és tranzakciót az adatbázisból a bean

számára.

Az interfészt megvalósító bean pedig a következő:

```
@Stateless
public class StatisticsCounterBean implements
                                        StatisticsCounter {

private EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("dukesbank-ejb");
    private EntityManager em = emf.createEntityManager();

    private Collection customers = null;
    private Collection accounts = null;
    private Collection txs = null;

    public StatisticsCounterBean() {
        init();
    }

    @Override
    public int customersNumber() {
        return customers.size();
    }

    @Override
    public String mostCustomerCity() {
        Iterator i = customers.iterator();
        Map<String, Integer> cityNo = new HashMap<String,
                                                Integer>();
        int max=0;
        String maxcity="";
```

```
    while(i.hasNext()){
        Customer customer = (Customer)i.next();
        String city = customer.getCity();
        if(cityNo.containsKey(city)){
          int no = cityNo.get(city);
          cityNo.put(city, no+1);
        }
        else{
          cityNo.put(city, 1);
        }
    }

    for(String s:cityNo.keySet()){
        if( cityNo.get(s) > max){
          max=cityNo.get(s);
          maxcity=s;
        }
    }
    return maxcity;
}

@Override
public int accountsNumber() {
    return accounts.size();
}

@Override
public int transactionsNumber() {
    return txs.size();
}
```

```
@Override
public double totalBalance() {
     double balance= 0.0;
     Iterator i = accounts.iterator();
     while(i.hasNext()){
          Account account = (Account) i.next();
          balance+=account.getBalance().doubleValue();
     }
     return balance;
}

@Override
public Map<String, Integer> noOfAccountsByType() {
     Map<String, Integer> abt = new TreeMap<String,
                                                 Integer>();
     Iterator i = accounts.iterator();
     while(i.hasNext()){
          Account account = (Account)i.next();
          if(abt.containsKey(account.getType())){
            String s = account.getType();
            int no=abt.get(s);
            abt.put(s, no+1);
          }
          else{
            abt.put(account.getType(), 1);
          }
     }
     return abt;
}

@Override
```

```
public Map<Long, Integer> noOfTransactionsOfAccounts() {
    Map<Long, Integer> toa = new TreeMap<Long, Integer>();
    Iterator i = txs.iterator();
    while(i.hasNext()){
        Tx tx = (Tx)i.next();
        if(toa.containsKey(tx.getAccount().getId())){
          long l = tx.getAccount().getId();
          int no=toa.get(l);
          toa.put(l, no+1);
        }
        else{
          toa.put(tx.getAccount().getId(), 1);
        }
    }
    return toa;
}

private void getCustomers(){
    customers =
em.createNamedQuery("Customer.GetAll").getResultList();
}

private void getAccounts(){
    accounts =
em.createNamedQuery("Account.GetAll").getResultList();
}

private void getTxs(){
    txs =
em.createNamedQuery("Tx.GetAll").getResultList();
}
```

```
    @Override
    public void init(){
        this.getCustomers();
        this.getAccounts();
        this.getTxs();
    }
}
```

A 'getCustomers', 'getAccounts' és a 'getTxs' metódus egy NamedQuery-t hoz létre melyeket a Customer, Account és Tx entitásokba írtam:

```
@NamedQuery(name = "Account.GetAll", query = "SELECT a FROM
            Account a")
@NamedQuery(name = "Customer.GetAll", query="SELECT c FROM
            Customer c")
@NamedQuery(name = "Tx.GetAll", query="SELECT t FROM Tx t")
```

A 'BankAdmin' nevű osztályba beinjektáltam a bean-t:

```
@EJB
private static StatisticsCounter statisticsCounter;
```

Majd a 'Show' gomb megnyomásakor a következő metódus fut le:

```
private void
showStatButtonActionPerformed(java.awt.event.ActionEvent evt)
{
    statisticsCounter.init();
this.noOfCustLabel.setText(statisticsCounter.customersNumber()
+"");
this.mostCustCityLabel.setText(statisticsCounter.mostCustomerC
```

```
ity());
this.noOfAccLabel.setText(statisticsCounter.accountsNumber()
+"");
this.noOfTransLabel.setText(statisticsCounter.transactionsNumb
er()+"");
this.totalBalanceLabel.setText(statisticsCounter.totalBalance(
)+"");

    Map<String, Integer> abt =
           statisticsCounter.noOfAccountsByType();
    Map<Long, Integer> toa =
           statisticsCounter.noOfTransactionsOfAccounts();

    int row=0;
    for(String s:abt.keySet()){
           abtTable.setValueAt(s, row, 0);
           abtTable.setValueAt(abt.get(s), row, 1);
           row++;
    }

    row=0;
    for(Long l:toa.keySet()){
           toaTable.setValueAt(l, row, 0);
           toaTable.setValueAt(toa.get(l), row, 1);
           row++;
    }
}
```

9. Összefoglalás

A Java Enterprise Edition nem az egyetlen eszköz vállalati alkalmazások létrehozására, de mindenképp az egyik legkézenfekvőbb választás, hiszen számtalan előnye és lehetősége van ennek a feladatnak a megoldására. A Java EE közösség nagyon nagy az egész világon és számtalan angol és más nyelvű dokumentáció áll rendelkezésre, így az elsajátítása igen egyszerű. Ha már tapasztalt fejlesztők vagyunk, akkor is akadhatnak olyan problémák melyek hátráltatják a munkánkat, viszont a fórumokon vagy levelezőlistákon gyorsan rábukkanhatunk a hibára. Más részről a Java EE az ipar igényei szerint fejlődik így korszerű megoldást nyújthat a vállalatok számára.

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek, Dr. Fazekas Gábornak és mindazoknak akik segítették a munkámat a szakdolgozat megírása során.

Külön köszönet a szüleimnek akik támogatták a tanulmányaimat.

Irodalomjegyzék

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

[2] Enterprise JavaBeans, 3.0 By Bill Burke, Richard Monson-Haefel

[3] Mastering Enterprise JavaBeans 3.0 By Rima Patel Sriganesh, Gerald Brose, Micah Silverman

[4] The Java EE tutorial, http://java.sun.com/javaee/5/docs/tutorial/doc

[5] http://java.sun.com

[6] http://www.javapassion.com/j2ee

[7] Wikipedia, http://en.wikipedia.org

[7.1] http://en.wikipedia.org/wiki/Java_EE

[7.2] http://en.wikipedia.org/wiki/Enterprise_JavaBean

[7.3] http://en.wikipedia.org/wiki/Application_server