DIPLOMAMUNKA

Dévai Vince

Debrecen
2009

Debreceni Egyetem
Informatika Kar

Java Enterprise Programozás

Témavezető:
Dr. Fazekas Gábor
Beosztása: egyetemi docens

Készítette:
Dévai Vince
Programtervező Informatikus MSC

Debrecen
2009

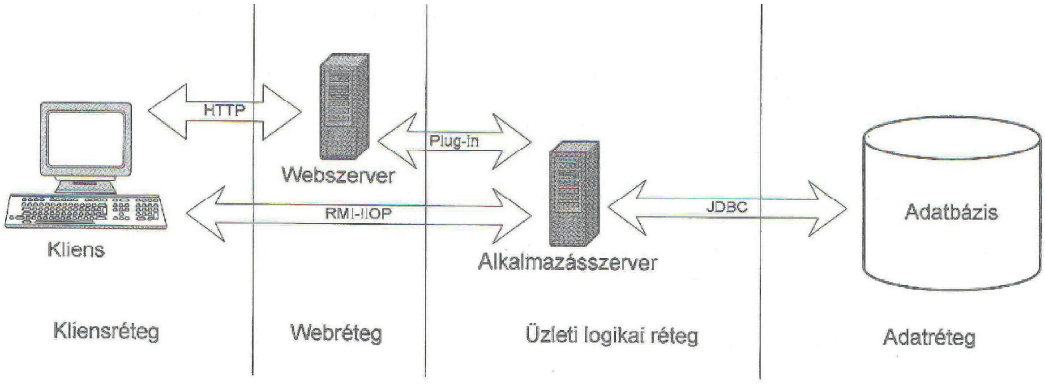
Tartalomjegyzék

Bevezető

A java nyelv megjelenése (1991) és ismertté válása (1995) óta maga a nyelv és a hozzá tartozó technológiák dinamikus fejlődésen mentek át, egyre több feladat ellátására lettek alkalmasak és így sokak körében elterjedt. Ezek következtében három kiadása jelent meg a nyelvnek, amelyekkel eltérő szoftvereket lehet fejleszteni. A Standard Edition (Java SE) a hagyományos desktop alkalmazások készítését célozza meg, a Micro Edition(Java ME) a mobil eszközökre való fejlesztést teszi lehetővé. A legnagyobb kiadás pedig a Java Enterprise Edition (Java EE) amely a több felhasználós vállalati méretű szoftverrendszerek létrehozásakor alkalmazandó. A Java EE –re a következő rövid meghatározás megfelelő lehet: architektúra vállalati méretű alkalmazások fejlesztésére Java és Internetes technológiák alkalmazásával. A Java EE olyan módon támogatja az ilyen rendszerek fejlesztését, hogy a gyakori felmerülő problémákra globális megoldást nyújt. Ezt pedig úgy teszi, hogy szolgáltat egy futási környezetet, keretrendszert, amely a fejlesztendő alkalmazásoknak nyújt különféle szolgáltatásokat az igények kielégítésére. Ezek az igények például lehetnek a perzisztencia vagy a többszálúság. A diplomamunka a JAVA Enterprise néhány technológiáját szeretném bemutatni többek között egy példa program segítségével. Ezen technológiák a következők lesznek: az Enterprise JavaBeans, ezen belül az EJB 2.1 és utána az EJB 3 ismertetése. Ehhez kapcsolódóan röviden a név- és directory szolgáltatásról, a tranzakció kezelésről lesz szó. Továbbá a Java EE alapszintű webes technológiáiról, a Servlet és a JavaServer pages-ről, melyek ismerete elkerülhetetlen a vállalati szoftverfejlesztésben.

1.Java EE alkalmazásszerver és az n rétegű architektúra

A Java EE esetén a futási környezet az alkalmazásszerver, ami a middleware szolgáltatásokat nyújtja. Ez egy réteges architektúrába illeszkedik bele, ami az 1.1. ábrán látható.



1.1 ábra- az n rétegű architektúra (Imre G. 2007)

Ennek az architektúrának jellemzője, hogy minden egyes réteg egy-egy jól definiált feladatot lát el, felhasználva közvetlenül az alatta levő réteg szolgáltatásait. Az adatréteg felel az adatok perzisztens tárolásáért és az azokon végrehajtható műveletekért (létrehozás, módosítás, törlés, lekérdezés). Leggyakoribb megvalósítás a relációs adatbázis. Ide tartozhat még minden olyan rendszer is, amelyből az alkalmazásunk adatokat nyerhet ki, vagyis ez akár egy korábbi alkalmazás is lehet.

Az adatrétegre épül az üzleti logikai réteg, amely az üzleti szabályok figyelembevételével hívja meg az adatréteg szolgáltatásait. Egy banki alkalmazásnál például az üzleti logika feladata az átutalás lebonyolítása, mely során ellenőrizzük a jogosultságot az átutaláshoz, majd ezután az adatréteget felhasználva csökkenti és növeli a megfelelő oldalon az egyenlegeket. Ezt a réteget telepítjük az alkalmazásszerverre a middleware szolgáltatások felhasználására. Az üzleti logikát Enterprise Java Bean komponensekbe szokás elhelyezni,

amelyek távolról is elérhetőek, ha a web kliens más alkalmazásszerverben van. Ekkor RMI (Remote Method Invocation) – távoli eljárás hívással érhetőek el az EJB-k.

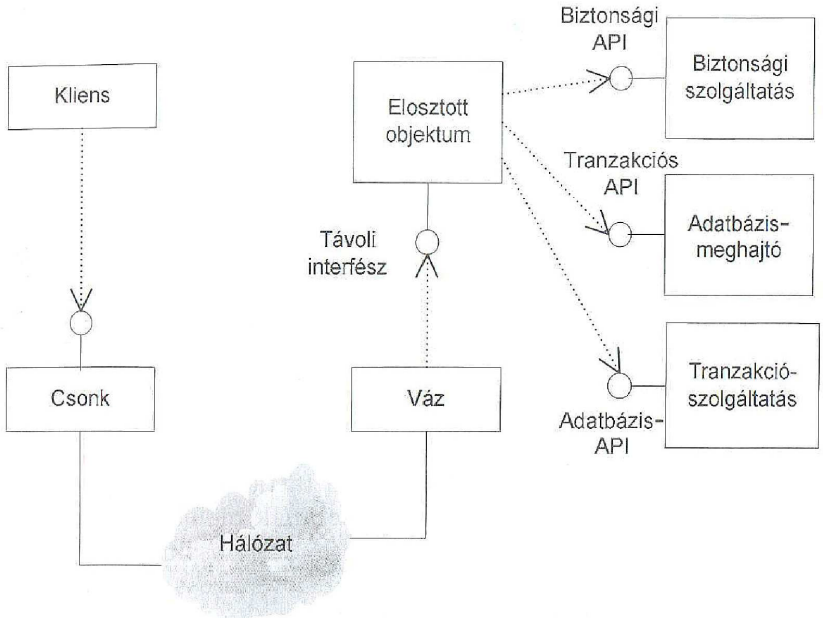
A kliensréteg biztosítja a felhasználói felületet, amelyen beavatkozva hívhatunk meg bizonyos üzleti logikai funkciókat és e funkciók lefutása után a frissített elemeket kapjuk a felhasználói felület egyes részein. A kliens lehet vastag vagy vékony. A vastag kliens hagyományos desktop alkalmazások lehetnek a vékony kliensek pedig a böngészők, ami webes alkalmazásokhoz biztosítja a letöltött oldalak megjelenítését. A két kliens típus közötti határ elmosódása egyre inkább megfigyelhető, mivel különféle technológiák segítségével a vastagkliens hez hasonló felhasználói élményt képesek nyújtani a böngészők is jelenleg.

Ha a vékony klienset is ki akarjuk szolgálni, akkor ebben az esetben szükségünk van egy webrétegre is, amely a böngészőkből érkező http – kéréseket értelmezi, továbbítja az üzleti logika felé és a megfelelő választ generálva visszaküldi a böngészőnknek. Java EE esetén a webréteg szintén az alkalmazásszerverre települ. Az alkalmazásszerver képes közvetlenül fogadni a kéréseket a böngészőtől, de webszerver is közbeiktatható mely a statikus erőforrásokat szolgálja ki.

Minden réteg már meglévő, készen kapható szoftverekre épül: az adatréteg egy adatbázis-kezelő rendszerre, az üzleti logika és a webréteg egy alkalmazásszerverre, a kliensréteg pedig vékony kliens estén a böngészőre. Ezek fizikailag külön gépeken, de akár mindegyikük egyazon gépen is futhat.

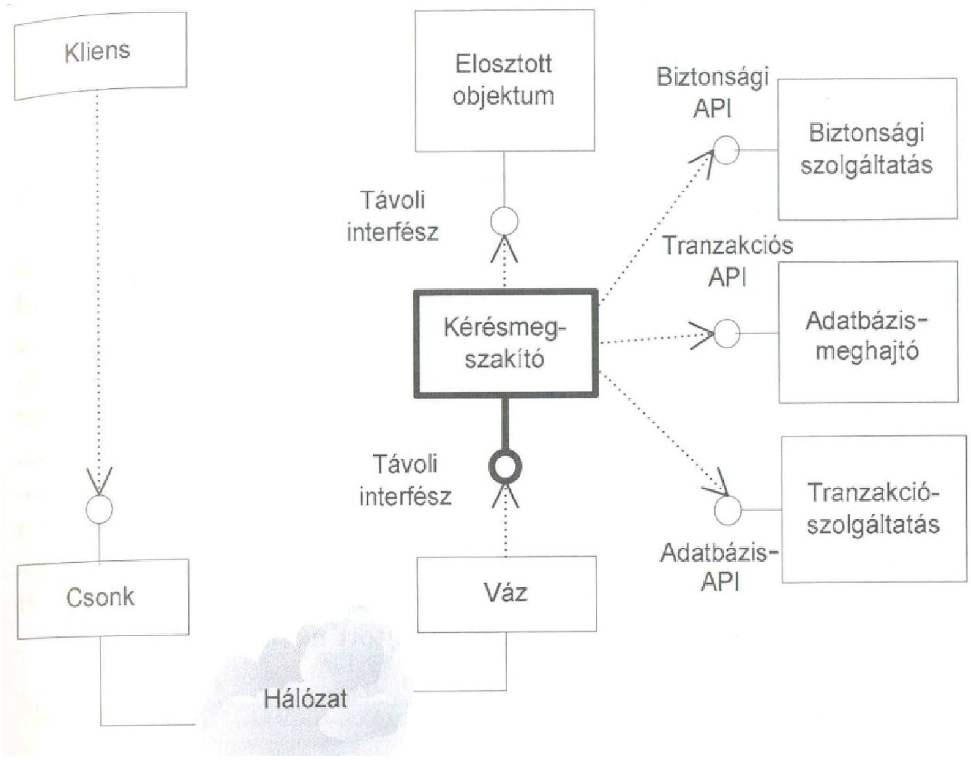
2. EJB 2.1 bemutatása

Az Enterprise JavaBean-ek olyan komponensek, melyek a Java EE alkalmazásokban központi szerepet töltenek be és szabványos felületet és szolgáltatásokat nyújtanak . A jelenlegi EJB specifikáció három fajta Enterprise JavaBean-t definiál. A session bean-ek az üzleti folyamatokat reprezentálják, bizonyos üzleti logikai funkciókat látnak el. Az entity beanek az üzleti modellt és ennek entitását reprezentálják, továbbá perzisztenciáért felelős adatbázissal tartják a kapcsolatot, vagyis a bean egy példánya a tábla egy sorának memóriabeli cacheként értelmezhető. A harmadik a message-driven bean, ami az aszinkron üzenetküldésre ad megoldást. A többrétegű architektúra esetén a middleware szolgáltatásokat a középső réteg látja el. Jelen esetben J2EE rendszerekben ez az EJB feladata. A middleware szolgáltatások elérésének hagyományos megoldása az ún. explicit middleware ami a 2.1-es ábrán látható.



2.1. ábra – Explicit middleware (Imre G. 2007)

Itt látható, hogy a kérést a kliensoldali csonk továbbítja a szerveroldali váz felé, amely révén meghívja az Elosztott objektum megfelelő metódusát. Az elosztott objektum különféle middleware API– kat hív meg a metódus implementációjában. Így például a komponens eladásakor kénytelenek vagyunk közzé tenni a forráskódot, ha a vevő másképp akarja a middleware szolgáltatásokat használni. Ennek kiküszöbölésére jó az implicit – middleware. Itt egy külön leíró fájlban deklaráljuk a használni kívánt szolgáltatást, így a programkódban ténylegesen csak az üzleti logika lesz. Ezáltal eladhatóvá válik, hiszen nem kell a forráskódot kiadni. Mindez az Elosztott objektumhoz érkező Kérésmegszakítóval oldható meg, ezt a megszakítót nem kell nekünk megírni, generálódik egy leíró fájl alapján, amely segítségével hívódnak meg a middleware API- k. Az implicit middleware a 2.2. –es ábrán látható.



2.2. ábra - Implicit middleware (Imre G. 2007)

Az EJB előnye, hogy implicit middleware szolgáltatásokat vehetünk igénybe. Ezt az EJB konténer biztosítja, amely az alkalmazásszerver kulcsfontosságú része és számos kódolást átvállal a fejlesztőtől. Annak érdekében, hogy a konténer elvégezhesse feladatát egy EJB komponenst több osztályból, interfészből és egyéb fájlból kell felépíteni. Ezek egy részét

nekünk fejlesztőknek a feladata megírni, másik részét a konténer generálja fejlesztési vagy telepítési, futási időben. (Imre G. 2007)

2.1. Session Bean

A session bean az alábbiakból épül fel : EJB objektum, távoli/lokális intefész, Home interfész, Home objektum, Enterprise Bean osztály, telepítésleíró és a gyártóspecifikus fájl. Az EJB komponens építőelemei bemutatására egy egyszerű faktoriálist előállító kódot is felhasználok.

2.1.1. Névszolgáltatásról röviden

Java-ban a különféle névszolgáltatások JNDI (Java Naming and Directory Interface) API segítségével érhető el. Java EE alkalmazás telepítésekor a telepítő minden EJB komponensnek ad egy JNDI nevet, amit az alkalmazásszerveren levő névszolgáltatás nyílván tart. Az EJB – kliensei JNDI név alapján keresik meg a használni kívánt EJB-t. A keresés során a névfeloldás történhet közvetlen vagy indirekt módon. 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.

2.1.2. EJB objektum

Az EJB objektum az egyik építőeleme az EJB komponensnek, ez a kérésmegszakító. Ez egy leíró fájl alapján middleware API- kat hív meg és delegálja a kérést a bean osztály felé. Az EJB objektumot a konténer generálja, azt viszont, hogy milyen metódusokat tartalmazzon, egy interfészben kell leírni. Ezt az interfészt hívják távoli interfésznek (remote intarface) vagy lokális interfésznek (local interface). A lokális interfészt abban az esetben lehet használni, ha azonos JVM-en fut az EJB és a kliens. Ebben a részben kell deklarálni azt a metódust amit a kliensek akarnak meghívni. Ez itt a faktor() metódus lesz. A kódja a következő:

```
package faktor;
public interface FaktorRemote extends javax.ejb.EJBObject {
```

```
        double faktor(double n) throws java.rmi.RemoteException;
    }
```

A FaktorRemote távoli interfész az EJBObject interfészt terjeszti ki, és java.rmi.RemoteException-t dobhat. A megírt interfészt a konténer által generált EJB objektum fogja implementálni.

2.1.3. Home objektum, home interfész

Egy másik építőelem a home objektum. Ez játszik szerepet az EJB objektumra való referenciaszerzésben. Ezt a Factory method(gyártó metódus) tervezési minta segítségével valósítja meg, vagyis a gyártó metódus szerepet itt a home objektum kapja. Ennek segítségével tud a kliens létrehozni olyan csonkokat, melyek metódusait hívva a kérés az EJB objektumon keresztül eljut a bean osztályhoz. A kérdés, hogy szerezzünk referenciát a home objektumra. A home objektum hivatkozásakor a Java EE erőteljesen felhasználja a névszolgáltatást. A kliens valamilyen logikai néven hivatkozik a home objektumra, amely feloldása a telepítéskor valósul meg. A home objektumot szintén a konténer hozza létre, és amikor egy kliens referenciát kér egy EJB objektumra a konténer dönti el, hogy újat hoz létre vagy már meglévő objektumra ad referenciát. Szintén a konténer dönti el, hogy adott bean-hez vagy esetleg többhöz megy a kérés, a többszálú kezelést az utóbbi esetben a konténer végzi el szintén. Az EJB telepítésekor JNDI néven regisztrálódik az EJB home objektuma, és ezt a kliens általában indirekt néven meg tudja keresni. Azt, hogyan kell a home objektumban inicializálni az EJB objektumot, egy interfészben kell megadnunk. Ez a home interfész és ezt az interfészt implementálja a home objektum, továbbá itt kell megadni a bean osztályinicializáló metódusait is. Még egy dolog: ha a hívó és az EJB, amit hív egy szerveren fut, akkor a lokális home interfészt használjuk. A home interfész kódja:

```
package faktor;

public interface FaktorRemoteHome extends javax.ejb.EJBHome {
        faktor.FaktorRemote create() throws
                javax.ejb.CreateException, java.rmi.RemoteException;
}
```

Itt a home interfészben deklarálva lett egy create() metódus, ez CretaException-t és RemoteException-t is dobhat. A create() metódus a távoli interfésszel tér vissza. (Imre G. 2007)

2.1.4. Enterprise bean osztály

Egy EJB komponens másik építőeleme az Enterprise Bean osztály más néven az implementációs osztály. Ez tartalmazza a kliensek felé felkínált metódusok implementációit, amely az üzleti logikát tartalmazza. A kliensek viszont sohasem érik el ezt a bean osztályt csak a kliensoldali csonkját, ami egy generált fájl. A faktorBean implementációs osztály kódja a következő:

```
package faktor;
import javax.ejb.*;

public class faktorBean implements javax.ejb.SessionBean {
    private javax.ejb.SessionContext context;

    //A session interfész által megkövetelt metódusok
    public void ejbCreate(){
    System.out.println(”ejbCreate()”);
    }
    public void ejbRemove(){
    System.out.println(”ejbRemove()”);
    }
    public void ejbActivate(){
    System.out.println(”ejbActivate()”);
    }
    public void ejbPassivate(){
    System.out.println(”ejbPassivate()”);
    }
    public void setSessionContext(SessionContext ctx){
    this.ctx = ctx;
    System.out.println(”setSessionContext()”);
    }
    // A tényleges üzleti metódusok
    public double faktor(double n){
            if (n=1) return n;
            return n*faktor(n-1);
    }
}
```

Ez az implementációs osztály tartalmazza a kliensek felé nyújtott szolgáltatásokat, ez itt épp a faktor metódus, továbbá mivel implementálja a javax.ejb.SessionBean interfészt így egyéb metódusokat is tartalmaz.

- Az ejbCreate(), mely azután hívódik meg a konténer által, miután új példány jött létre a bean osztályból. A kliens sohasem közvetlenül példányosítja az EJB-t. A kliens a home interfészen keresztül a create metódust meghívva szerez referenciát a példányra. A konténer viszont itt dönthet úgy, hogy új példányt hoz létre, ekkor a kliens oldali create() hívás után a konténer meghívja az ejbCreate() metódust, továbbá dönthet úgy is, hogy nincs szükség új példányra, ekkor újrahasznosít egy már létező példányt és így ekkor nincs ejbCreate() hívás.
- Az ejbRemove() metódussal a konténer szabadíthat fel egy bean példányt, ami nem kell, hogy egybeessen a kliens remove() metódusával.
- Az ejbActivate() és ejbPassivate() metódusoknak csak az állapottal rendelkező session bean esetén van jelentőségük. Az állapotmentes sessionbaen esetén a kliens nem számíthat arra, hogy mindvégig ugyanazzal a bean példánnyal lesz kapcsolata és nem lehet ezért állapotot tárolni benne, nem lehet például személyhez kötődő információt tárolni. Mikor arra van szükség, hogy állapotot tároljunk a session beanben több hívás eltelte után is, akkor használjuk az állapottal rendelkező session beant. Jó példa erre az online bevásárlókocsi. Itt szükség van arra, hogy a meglévő beaneket elvegyük a régebbi klinsektől és az újabbakhoz rendeljük. Ekkor a bean állapotát a konténer valamilyen háttértárra menti, ez a passziválás. Amikor ismét a régi klienst akarjuk kiszolgálni, akkor az állapotot vissza kell tölteni egy másik bean példányba, amit természetesen passziválni kell, ha szükséges. Ez a visszatöltési folyamat az aktiválás. Mindehhez szükség van arra, hogy a bean minden elmenteni kívánt példányváltozójának szerializálhatónak kell lennie, az elmenteni nem kívánt változókat a transient kulcsszóval kell megjelölni. A konténer feladata eldönteni mikor passzivál egy beant betartva, hogy kötelező a memóriában tartani a beant, ha épp érintett tranzakcióban vagy metódust hajt végre.
- Tartalmazza még a setSessionContext() metódust is, amelyet a konténer hív meg az ejbCretae() előtt, továbbá átadja neki paraméterként a sessionContext-et. Ez a metódus egy példányváltozóba tárolja a megkapott értéket. Ez a sessionContext

egyfajta gateway a konténer felé. Ennek segítségével tart kapcsolatot a bean a konténerrel, és így információkat is kaphatunk a konténertől, például, hogy melyik felhasználótól érkezett hívás .

2.1.5. A telepítés leíró és a gyártóspecifikus fájl

Telepítés leíróban először megadjuk, hogy a komponens tartalmaz egy session beant és adunk neki egy nevet, amin lehet később hivatkozni rá (ejb-name). A fejlesztés és telepítés során a display-name elemben levő érték ad információt a grafikus eszközök számára. Ezután következhet a beant alkotó interfészek és osztályok felsorolása. Megadjuk még, hogy a bean állapotmentes és a tranzakciókért a konténer felelős.

A gyártóspecifikus fájlban adjuk meg, hogy telepítési időben milyen JNDI nevet kapjon a Faktor EJB. (Imre G. 2007)

2.1.6. A Faktor kliens használata vastag kliensből

Első lépésként meg kell keresnünk a home objektumot. A névszolgáltatás gyökerét a javax.naming.Context objektum adja meg. Ennek a lookup() metódusával rákeressünk a JNDI névre, aminek eredménye a home objektum lesz, amit megfelelő típusra kell konvertálni. Ezt követően a create() meghívásával referenciát kapunk az EJB objektumra, vagyis a kliensoldali csonkra, ami implementálja a távoli interfészt. Így pedig már meghívhatjuk annak metódusait. A kód a következő:

```
package faktor;
import javax.naming.Context;
import.javax.naming.InitialContext;

public class FaktorClient {
        public static void main(String[] args) throws Exception{
                Contex ctx = new InitialContext; //névszilgáltatás inicializásása
                Object obj = ctx.lookup(”ejb/FaktorBean”); //a home objektum keresése
                CalculatorRemoteHome home = (FaktorRemoteHome)
                        Javax.rmi.PrtableRemoteObject.narrow(obj,FaktorRemoteHome.class); // a keresési
eredmény konvertálása
                FaktorRemote fakt = home.create(); //referencia szerzés az EJB obj. Csonkjára
                System.out.println(fakt.faktor(10)); //metódus meghívása
```

```
                fakt.remove(); //objektum eltávolítása, kliens nem használja már az EJB-t

            }
        }
```

2.2. Entity bean

Feladata az általa reprezentált objektum adatait perzisztensen eltárolni, illetve annak elérését biztosítani. Az ún. objektumrelációs leképezést (Object-Relational-Mapping) az entity beanek oldják meg. Úgy történik, hogy az adatbázis táblái osztályoknak, a táblák attribútumai az osztályok attribútumainak kerül megfeleltetésre. Vagyis egy adott osztály példányai az adatbázis egy sorának felel meg. Ahogyan az adatbázis rendelkezik elsődleges kulccsal, úgy az entity bean is rendelkezik ilyen kulccsal, ami egyértelműen azonosítja azt. A leképezésben az ejbCreate() metódusban SQL INSERT-et hajtunk végre, az ejbRemove() metódusban pedig SQL DELETE-t. Az adatszinkronizációért az ejbLoad() és az ejbStore() felel, ezeket minden entity bean példánynak implementálnia kell. Az utóbbi két metódus az SQL SELECT illetve az SQL UPDATE utasításokat tartalmazzák, és ezeket a konténer hívja meg. Vagyis egy attribútum megváltoztatásához nekünk ún. setter/getter párosokat kell hívnunk, ami hatására az ejbStore/ejbLoad() meghívódik. Lehetőség van bizonyos finder metódusok hívására, amikkel konkrét entity bean példányok kereshetők meg tetszőleges paraméterek alapján. Ekkor SQL SELECT utasítások találhatók ezek implementációiban. Ezen finder metódusok visszatérési értéke lehet egy találat esetén ezt a bean példányt reprezentáló csonk (vagyis távoli vagy lokális interfésze az entity beannek), több találat esetén egy java.util.Collection, amelyben a entity bean távoli vagy lokális interfészeit megvalósító objektumok találhatók.

Az entity beannek két típusa létezik. Ezek a Bean Managed Persistence és a Container Managed Persistence. A BMP – nél a perzisztenciáért felelős kódot a programozónak kell megírni JDBC API segítségével a megfelelő metódusokban(ejbCreate(), ejbLoad()…). CMP esetén pedig a konténer átvállalja ezeket, vagyis a JDBC kód generálódik. (Imre G. 2007)

2.2.1. BMP (Bean Managed Persistence)

A lokális Interfész:

A perzisztens attribútumok elérését biztosító getterek/setterek alkotják ezen interfészt.

A lokális home interfész:

Tartalmaz egy create() metódust, melyben az elérni kívánt attribútumok adhatók meg. Továbbá tartalmazhat finder metódusokat, melyekkel tetszőleges keresést oldhatunk meg konkrét bean példányokra.

Az implementációs osztály:

Az entity bean példánynak megfelelő relációbeli sor létrehozását az ejbCreate(), törlését az ejbRemove() metódusok végzik. Tartalmazza még az adatbázisnak megfelelő memóriabeli példányváltozókat és az ezekhez tartozó értékek szinkronizálását végző metódusokat, melyek a következők: ejbLoad(),ejbStorre().

Továbbá még tartalmazza a kereső metódusokat a findereket. Ezek a metódusok használják a JDBC API-t:

1. Connection objektum elkérése, vagyis az adatbázis kapcsolat szerzése JNDI nevek segítségével történik getconnection() metódusban.
2. Connection segítségével PreparedStatement létrehozása az SQL utasítást tartalmazó stringgel, paraméterek helyén kérdőjellel.
3. Paraméterek beállítása
4. PrepareStaement végrehajtása, lekérdezésnél feldolgozni a ResultSet-et while ciklusban.
5. PreparedStatement és Connection bezárása egy finally blokkban, hogy mindenképpen megtörténjen.

A telepítésleíró és a gyártóspecifikus fájl:

A telepítésleíróban (ejb-jar.xml) definiáljuk, hogy a bean egy entity bean és, hogy milyen interfészek és osztályok alkotják. Meg kell adni a perzisztencia típusát vagyis, hogy BMP-t használunk és az elsődleges kulcs típusát. Összetett kulcs esetén egy osztályt kell létrehozni, amiben megtalálhatóak a kulcsot alkotó attribútumok. Meg kell adni még, hogy milyen típusú külső erőforrást és milyen logikai néven akarja elérni a bean-ünk.

A gyártóspecifikus fájlban (sun-ejb-jar.xml) megadjuk, hogy telepítés után milyen JNDI néven lesz elérhető az EJB és megadjuk azt az adatbázis JNDI nevet, amit az alkalmazásszerveren regisztráltunk az adatbázishoz. (Imre G. 2007)

2.2.2 CMP(Container Management Persistence)

A kliens számára nem lényeges, hogy az entity bean CMP vagy BMP technológiával készült e, így a lokális interfész amiről BMP esetén beszéltünk itt változatlan marad. A különbség az implementációs fájlban illetve a telepítésleíróban, gyártóspecifikus fájlban van.

Jelentős különbség az implementációs fájlban a BMP –hez képest, hogy **abstract** az osztály, mégpedig azért mert az elérést biztosító getter/setter párosok is **abstract**- ak. Így a perzisztenciával kapcsolatos műveleteket a konténer által generált osztályok végzik ténylegesen. A konténer ezt az abstract implementációs osztályt definiálja felül, így az ezekben levő abstract metódusokat is felüldefiniálja és ezeket már meg lehet hívni. Ez a 2.1. es ábrán látható. Különbség még itt a BMP- hez képest, hogy az ejbStore(), ejbRemove() és az ejbLoad() metódusokat a generált implementációs osztály definiálja felül, ezért itt üresek. Látható még, hogy a home interfészben levő finderek implementációi is üresek, ezeket a telepítésleíróban kell megadni eredményezve azt, hogy az implementációs osztály JDBC kód mentes lesz, így csak az adatlogikára koncentrálhatunk.

A telepítésleíróban(ejb-jar.xml) az első különbség, hogy megadjuk a konténer által kezelt perzisztenciát. Ezt követően egy abstract perzisztencia sémában megadjuk, melyek a perzisztens attribútumok és, hogy melyik az elsődleges kulcs ezek közül. A bean-hez kapcsolódó finderek közül a findByPrimaryKey-t nem kell definiálni, elég megadni az elsődleges kulcsot. A többit viszont meg kell adni itt ún. EJB-QL nyelven. Szintaktikája hasonló az SQL-hez, bemenő paraméterekre sorszámmal lehet hivatkozni.

A gyártóspecifikus fájl(sun-ejb-jar.xml) egy valamiben különbözik a BMP-től. Ez pedig az, hogy <cmp-resource> elemen belül kell megadni, hogy az alkalmazásszerveren milyen néven van beregisztrálva a perzisztens tároló.
Azt, hogy a telepítésleíróban megadott perzisztens attribútumok mely tábla mely oszlopaira képződjenek le egy gyártóspecifikus fájlban kell megadni. Ez a sun-cmp-mappings.xml. (Imre G. 2007)

Kapcsolatok O-R leképezése:

Ennek a leképezésnek a lényege, hogy az idegen kulcson keresztül tárolt kapcsolatok objektumorientált szemléletre képezze le. Ilyenkor a két kapcsolatban részt vevő entity bean lokális interfészei tartalmaznak kapcsolatmenedzselő metódusokat.
A kapcsolatok leképezése BMP és CMP esetén nem ugyanúgy történik. BMP esetén az implementációs osztályban történik join művelettel a lekérdezésben. A kapcsolatok kezelését BMP esetén Bean Manager Relationships –nek nevezzük. A CMR viszont teljes mértékben a konténerre bízza a kezelést. Az implementációs osztályban abstract-nak deklaráljuk a kapcsolatmenedzselő metódusokat, a telepítésleíróban pedig megadjuk, hogy mely entitások között van kapcsolat és milyen a multiplicitása.

2.2.4.Tranzakciókezelésről röviden

Sok konkurens felhasználót kezelő rendszereknél kulcsfontosságú a tranzakciókezelés. A tranzakciókezelés megoldásával lehet biztosítani az ACID tulajdonságokat.

- Atomicitás(Atomicity): több összetartozó adatbázis-művelet közül vagy mindegyiknek le kell futnia sikeresen, vagy egyiknek sem.
- Konzisztencia(Consistency): Sikeres és sikertelen tranzakció futás után is konzisztens állapotban kell lennie az adatbázisnak.
- Izoláció(Isolation): A konkurens kliensek nem látják egy másik kliens által kezdett de még be nem fejezett tranzakciók hatásait.
- Tartósság(Durability): Sikeres tranzakciót követő változások hardver hiba esetén sem veszhet el.

Az izolációt kicsit részletesebben nézve:

Tranzakciót lehet kezdeményezni kliens rétegben, ekkor a kliens réteg kódja indítja, illetve hagyja jóvá/görgeti vissza a tranzakciót. A Java EE specifikáció ennek a támogatását nem követeli meg, így bizonyos alkalmazásszerver implementációk esetén nem alkalmazható.
Lehetőség van bean által kezelt tranzakcióra is, itt a telepítésleíróban kell ezt jelezni <transaction-type>bean</transaction-type>. Ennél a megoldásnál a bean feladata a tranzakció indítása, lezárása, visszagörgetése. Itt az átláthatóság miatt érdemes ugyanazon metóduson belül indítani és lezárni egy bizonyos tranzakciót.
A harmadik lehetőség a konténer által vezérelt tranzakció kezelés, az EJB objektum tartalmazza a kódot. Ekkor a Container szót kell a már látott elemek között megadnia a telepítésleíróban. Működését is a telepítésleíróban lehet metódusszinten szabályozni a tranzakciós attribútumokkal. Hatféle tranzakciós attribútum van:

- Required: Ha még nem indult, akkor a konténer indít egy tranzakciót, ha már indítva van egy, akkor a metódus abba csatlakozik.
- RequiresNew: Ha nincs még indítva, akkor a konténer indít egyet, ha már van egy T1 tranzakció, akkor az felfüggesztődik és a metódus egy új T2-ben fut le. Majd, ha lefutott, akkor a T1 is folytatódik.
- Supports: Ha nem indult még, akkor a konténer sem indít, így tranzakció nélkül fut le a metódus. Ha már van tranzakció, akkor ehhez csatlakozik a metódus.
- Mandatory: Ha még nem indult tranzakció, a konténer egy kivételt dob. Ha már van egy élő T a metódus meghívásakor, akkor ebbe csatlakozik.
- NotSupported: Ha nem volt T, akkor nem indít a konténer sem, és tranzakció nélkül fut le a metódus. Ha van már élő T akkor ezt felfüggeszti a konténer és tranzakció nélkül fut le a metódus. Majd utána folytatódik T.
- Never: Ha nincs élő tranzakció a meghívásakor, akkor tranzakció nélkül fut le, ha már van, akkor a metódus kivételt dob.

3. EJB 3 és a Servlet, JSP bemutatása

3.1. Az alkalmazás telepítése

Legelőször telepítenünk kell a NetBeans IDE-t, mégpedig azt, amelyik tartalmazza a JAVA EE-t és a GlassFish alkalmazásszervert. Az IDE –vel együtt telepített alkalmazásszerver tartalmazni fogja az integrált adatbázisszervert, ami a Derby és ez a szerverrel együtt fog elindulni.

Egy JNDI néven be kell regisztrálni az adatbázist az alkalmazásszerveren. Elindítjuk a NetBeans –t és a service fülre kattintunk. Itt a servers csomópontot lenyitjuk és a GlassFish V2-n jobb klikk, és itt a startot választjuk. Ekkor elindul az alkalmazásszerver és az adatbázis szerver, ezt Application server startup complete jelzi. A GlassFish V2-n még egy jobb klikk és kiválasztjuk a view admin console-t. Ekkor az alapértelmezett böngészőben elindul az alkalmazásszerver webes adminisztrációs felülete. Itt az admin loginnal és az adminadmin jelszóval léphetünk be, ha a telepítéskor ezt nem változtattuk meg. Az oldal bal oldalán keressük meg a resources csomópontot és lépjünk bele. Ezután a JDBC-t válasszuk, majd a JDBC Resources-t. Itt adjunk meg egy új JNDI elérést, ami a jdbc/transfersystem legyen (ezt fogjuk megadni az alkalmazásunk perzistence.xml-ben). A Pool Name értékeként adjuk meg a DerbyPool-t. Ezután lépjünk vissza egyet és a transfersystem sorában lévő DerbyPool-ra kattintsunk. Ezután Additional properties fül és itt adjuk meg adatbázis névnek a transfersystem-t, loginnak az admin-t és jelszónak az adminadmin-t.

A következő lépés az adatbázis létrehozása. A service fülön legyünk a NetBeans-ben és nyissuk le a Databases csomópontot, itt a Java DB-n jobb klikk és Create Database. Ezután megadjuk a transfersystem nevet az adatbázis névnek és az admin, adminadmin-t a loginnak és a jelszónak. Így létrehoztuk az adatbázisunkat.

Most következik az alkalmazás fordítása. A Toolbars-on vagy a File menüből kiválasztjuk az Open Project- et, és keressük meg az alkalmazást transfersystem néven. Az alkalmazásunk két projektből áll és a NetBeans abszolút útvonalakkal tárolja a projektek közötti és a projektek – külső osztálykönyvtárak közötti függőségeket. Ezért gyakran előfordulhat, hogy Reference problem felirattal ellátott ablak jelenik meg. Ezt megoldhatjuk azzal, hogy a pirossal jelzett projekten jobb klikk és kiválasztjuk a Resolve Reference Problems-t, aminek

következtében egy következő ablakot kapunk, ahol megkaptuk, hogy milyen más projektekre hivatkozik, és ezeket kiválasztva feloldhatjuk a hivatkozási problémát. Ha az összes hivatkozási probléma megoldva, akkor jobb klikk a TransferSystem projekten és Build, ezután a böngészőből már elérhetjük az alkalmazásunkat a http://localhost:8257/StoreRegistry-war/ index.jsp címen.

3.2. A TransferSystem alkalmazás

Az alkalmazás kis összegű tranzakciók lebonyolításra alkalmas, felhasználva az EJB 3, Servlet és JSP technológiákat. A projekt tartalmaz egy EJB és egy web alkalmazás modult. Az alkalmazás kliensei webes felületen érhetik el az applikációt. A webréteg szervletekre és JSP-re épül, az EJB réteg session és entity bean-ekre.

Alkalmazás projektjei, csomagjai, osztályai:
Az alkalmazás két projektből áll. Ezek a TransferSystem-ejb és a TransferSystem-war. A TransferSytem-ejb - ben található három csomag. A manager csomagban találhatóak az business interfészek és implementációs osztályok, az object csomagban az entitások, a helper-ben a felhasználó állapotával, szerepkörével kapcsolatos műveletek elvégzéséhez segítséget nyújtó osztályok. Ebben a projektben található még a persitence.xml. Az alkalmazás webes projektjében a web oldalakban vannak a JSP-oldalak, ezen belül két mappában különülnek el a tulajdonosi hatáskörbe illetve a felhasználói hatáskörbe tartozó JSP-k. A Source Packages – ben három csomag van. Egyikben a szervletosztályok vannak, a helper csomagban egy az átutalások listázását segítő osztály van. A harmadikban az űrlapok fogadott értékeinek felhasználását segítő java osztályok. Ehhez a projekthez tartozik még egy konfigurációs fájl, ez a web.xml.
A továbbiakban a fent említett csomagokban levő osztályok, fájlok kerülnek bemutatásra.

persistence.xml: Ebben az XML fájlban kell beállítani a perzisztencia provider-t, ami esetünkben TopLink, ez felel a perzisztenciáért az entitásokra nézve. Meg kell még adni itt az alkalmazásszerben beregisztrált JNDI nevet, amely alapján az alkalmazás eléri az adatbázist, és megadjuk még a TopLink automatikus táblagenerálás attribútumát create-re.

LoginStatus.java: Bejelentkezés során ebben az enumerációban rögzítjük a bejelentkező állapotát, hogy engedélyezett e a belépés a login és a hozzá tartozó jelszó alapján. Ez három értéket vehet fel. FORBIDDEN-tiltott, ALLOWED-engedélyezett és PENDING- folyamatban levő, vagyis a felhasználó még csak regisztrált.

TransferException.java: Ez egy kivételosztály, mely a java.lang.Exception-ből van származtatva. Átutalás közbeni kivételek során ennek keletkezik példánya, mely továbbítja a kivétel üzenetet az ősosztály felé.

UserRole.java: Ezzel az enumerációval ellenőrizzük az adatbázisban tárolt szerepek alapján a felhasználói szerepkört. Amely lehet REGULARCLIENT, REGISTERED, OWNER.

transfersystem.ejb.manger csomag :

Itt találhatók a business interfészek és az implementációs osztályok. Elég lenne a lokális interfészeket használni, mivel a webes kliensek ugyanebben az alkalmazásszerverben vannak, ahol az EJB komponensek futnak. A business interfészekben deklaráljuk a használni kívánt metódusokat és megadhatjuk a tranzakciós attribútumot, ami itt REQURED értéket kap a transfer metódus esetén. (Alap esetben is ez a REQUIRED érték az alapértelmezett). A REQUIRED attribútum esetén a metódus meghívásakor, ha még nem indult tranzakció, akkor a konténer indítani fog egyet, ha viszont már van élő tranzakció akkor ebbe csatlakozik bele. Az implementációs osztályban is elhelyezzük a tranzakciós attribútum beállítására vonatkozó sort, ami az: @TransactionAttribute(TransactionAttributeType.REQUIRED). Az implementációs osztályok elején annotációk találhatók. A @PersistenceContext()

EntityManager em;

két sorral referenciát kérünk az EntityManager-re és az annotáció hatására konténer feladata lesz a tranzakcióhoz tartozó perzisztencia kontextust a EntityManager-hez rendelni, és ezáltal kezelni tudjuk az entitásainkat.

A @EJB

private UserManagerLocal userManagerBean; két sor hatására a konténer biztosít referenciát a másik implementációs osztályra lokális interfészen keresztül. Továbbá még a metódusok implementációi találhatók a két implementációs osztályban.

Transfer.java: A Transfer entitást ebben az osztályban adjuk meg, melynek implementálnia kell a Serializable interfészt. Az entitásnak megfelelő perzisztens tábla neve az osztály név lesz mivel mást nem adtunk meg az @Entity annotációhoz. Megadjuk @NamedQueries és

@NamedQuery annotációkkal két statikus lekérdezést EJB-QL nyelven, mely igen hasonló az SQL-hez. @Id annotációval megadjuk az elsődleges kulcsot, és a strategy = GenerationType.AUTO- val biztosítjuk az automatikus generálását. @Column(name=”név”) annotációval és az utána levő változóval megadjuk, hogy milyen oszlopai legyenek, illetve attribútumai a táblának és az entitásnak. A @OneToMany annotációval létrehozzuk a külső kulcsot, mellyel a user táblára hivatkozunk. Eztán megadjuk még a szükséges getter/setter párokat.

User.java: Ebben az osztályban hozzuk létre a user entitást és annak minden oszlopát az elsődleges kulcsot is beleértve. A @ OneToMany annotációkkal reprezentáljuk a kapcsolat másik oldalát. A mappedBy attribútummal hivatkozunk a tulajdonosi oldal attribútumaira, mivel itt a több oldal a tulajdonos. Ezután itt is a getter/setter párosok következnek.

confirm.jsp, usermanagement.jsp, UserManagementServlet : A userManagement.jsp táblázatosan kiírja a felhasználókat és biztosítja a felhasználók megerősítését vagy törlését a szerepkörnek megfelelően űrlapon keresztül, melyben egy szkriptletben történik a megfelelő funkció kiválasztása. A két link a userManagementServlet-re irányul, ahol megtörténik a megerősítés a confirm.jsp közbeiktatásával, illetve a törlés. A confirm.jsp-ben az űrlap által átadott paramétereket tartalmazó param objektum id paraméterét használja a UserManagementServlet-nek, hogy az id által meghatározott user-nek állítsuk be a kezdő összeget.

list.jsp : A felhasználók személyes adatainak listázását valósítja meg. Itt is mint a legtöbb jsp oldalon megtalálható a <c:foreach> akcióelem, amivel az error lisitát íratjuk ki az aktuális oldalra. Minden sor után jelen van még egy módosító link, amellyel a módosítást elvégző JSP oldalakra(**addressmodify.jsp, birthDateModify.jsp, nameModify.jsp**) tehetünk hivatkozást. Ezek a JSP-k a **ModifyServlet**-tel odják meg a módosítást.

list_transfer.jsp, TablePage.java : Az átutalások kiíratásához nyújt segítséget a TablePage osztály, a caption és urlParameter példányváltozókkal, és a hozzájuk tartozó getter/setter párosokkal. A bal oldali menüben az átutalások listázása link a TransferServlet-nek továbbítja vezérlést a command paraméter értékadása mellett, így a szervlet megfelelő sora fog végrehajtódni. Innen pedig a JSP oldal kapja meg a kérés, válasz objektumot, ahonnan az oldalszakaszok kiválasztásával újara a szervlet jön, és így tovább.

main.jsp : A felhasználó bejelentkezése utána jobb oldali ablakban ez jelenik meg. Láthatjuk a bejelentkezett felhasználó nevét és egyenlegét, melyet a munkamenet (session) loggedinUser objektumának paramétereivel adjuk meg. Ezután a jobb oldali menüben kiválaszthatjuk a szükséges menüpontot.

modify.jsp : Itt a jelszó és a login módosítható, mely szintén a ModifyServlet-re továbbítja a vezérlést a módosítás gomb hatására, ahol megtörténik a módosítás kivitelezése.

new_transfer.jsp

footer.jsp: A menu.jsp-ben egy tábla van, aminek egy sora és ez két oszlopra tagolódik. Az első oszlopban a menü szkriptlet, a másik content azonosítóval ellátott oszlopban pedig mindig az aktuális tartalom. Ezt a tartalmat zárja le az oszlop, sor és tábla lezáró tag-gel.

head.jsp: Ennek a JSP oldalnak a beillesztésével használjuk a CSS fájlunkat, ami a web mappa gyökerében van.

index.jsp, LoginServlet, LoginForm : Include akcióelemmel kerül bele a menu.jsp, ezután <jsp:usebean> felhasználásával a sessionbe megkeresett loginform objektum propertjeivel töltődik fel, így megoldható, hogy regisztrálás utáni visszairányítás az index-re már a megfelelő login és jelszó értékekkel történik meg. Az űrlap kitöltését követően a submit gombbal a szervletre kerül a vezérlés, és megtörténik a felhasználó érvényesítése, ha rendben van, akkor sendRedirection-nal main.jsp-re kerül az irányítás. De előtte még a session objektum loggedinUser attribútumába kerül a bejelentkező user entitása.

menu.jsp : A bal oldali menü szkriptlet segítségével történik, ahol a felhasználói szerepkörök szerint jelenítődik meg a megfelelő menü.

RegisterServlet, register.jsp, Registerform: Ezek együttesen végzik el a regisztrálást az alkalmazásban. A JSP oldal segítségével űrlapot kitöltjük, és a szervlethez irányítódik a vezérlés az adatok perzisztens tárolóra töltéséért. Az adatok ellenőrzése a RegisterForm.java osztály és **FormException** kivételosztály segítségével valósul meg.

style.css: Ezzel a stílusleíró lappal formázhatjuk könnyedén a megjelenített tartalmat.

web.xml: Itt megadjuk a szervletek neveit és a hozzájuk tartozó szervlet osztályokat, megadjuk még az url elérhetőségüket az alkalmazásban.

LoginForm.java, RegisterForm.java, TransferForm.java: Ezekben az egyszerű java osztályokban tárolódnak ideiglenesen az entitások attribútumai a különböző műveletek alatt a

szervletekben. A JSP oldalaknál javaBean-ek ezekre az osztályokra hivatkoznak, vagyis ilyen típusúra konvertálódnak a class attribútumban megadott érték miatt.

LogoutServlet.java : Itt a session objektum elkérése után megtörténik az aktuális munkamenet törlése, és a bejelentkező képernyőre irányít.

TransferServlet.java : Az elején mindkét implementációs osztályra referenciát kérünk a konténertől az @EJB annotációval, amelyen keresztül elérhetjük az üzleti réteg metódusait. Ebben a szervletben van az új tranzakciót illetve annak a megerősítését végző üzleti logika, amely a JSP oldalakban levő űrlapokból nyeri az információkat, mint pl. a fogadó fél azonosítóját. Amellyel a userManagerBean implementációs osztály findUser metódusán keresztül megkeressük a fogadó user entitást, és ezután a transfeManagerBean transfer metódusát meghívja az átutalás elvégzésére. Itt található még a tranzakciók listázását megvalósító logika is.

3.2.1. EJB 3 előnyei

Az EJB 3 által egyszerűsödött a technológia. A fájlok száma zavaró lehetett az EJB 2.1-nél. A bean osztálynak kötelező implementálni a javax.ejb.Session/Entity/ MessageDrivenBean interfészt, így bizonyos callback metódusokat, mint az ejbCreate() akkor is kötelező megvalósítani, ha nem kerül bele semmi lényegi funkció. Egy EJB metódus meghívása kódok sorát jelentette, home interfész megkeresés JNDI-n keresztül, megtalált home interfész create() metódusának meghívása. Telepítésleíróban a logikai nevek leképezése tényleges JNDI nevekre plusz sorokat eredményezett. 2.1-es EJB esetén a session és entity együttdolgozására érdemes használnia session facade-t amelyhez DTO-kat kell definiálni (Data Transfer Object), ami redundanciát okoz mivel az entity bean-ek attribútumai a DTO-kban megismétlődnek.
Ezekre nyújt megoldást az EJB 3 technológia.

3.2.2. Annotációk

Az annotációk forráskódba illesztett metaadatok, közvetve befolyásolják, hogy az egyes komponenseket, programrészeket a különböző eszközök, osztálykönyvtárak miként kezeljék. Java 5 esetén az annotációk általános szintaxisa:
@AnnotációNeve(paraméter1=érték1,paraméter2=érték2)
Csak egy paraméternél: @AnnotációNeve(értéke)
Ha nem adunk paramétert: @AnnotációNeve, @AnnotációNeve()
Paraméter tömb is lehet:
@AnnotációNeve(paraméter1={érték1,érték2,…},paraméter2=érték2)

A Java EE több annotációt definiál a fejlesztők számára. Az alkalmazásszerver ezeket az annotációkat a komponens telepítésekor értelmezni fogja. Futási idejű szolgáltatásokat biztosít így a komponens számára, amellyel átvette a telepítésleíró szerepét, de továbbra is használhatunk telepítésleírót is, együttes használatkor felüldefiniálja az annotációt. A Java minden metaadatra alapértelmezett értéket definiál, így minimalizálták a metaadatok használatát. Csak akkor kell megadni egy metaadatot, ha ettől különböző.

3.2.3. Session Bean

Mivel az interfész és az implementáció szétválasztása szükséges az implicit middleware szempontjából így két fájlra lesz szükség ahhoz, hogy a kliensek az interfész által adott csonkot hívhassák. A bean osztály és az interfész összerendelése implements kulcsszóval történik, mint ahogyan látható a UserManagerBean és a UserManagerRemote (UserManagerLocal) vagy a TransferManagerBean és a TranferManagerRemote (TransferManagerLocal) esetén az alkalmazásban. Itt megjegyzem, elég lenne a lokális interfész, mivel a webrétegbeli EJB-kliensek ugyanabban az alkalmazásszerverben vannak. Az EJB specifikáció a lokális interfészt veszi alapértelmezettnek, ezt a @Remote annotációval változtathatjuk meg és alkalmazhatjuk a business interfészen és a bean osztályon is. Itt is mint az EJB 2.1. –ben létezik állapotmentes és állapottal rendelkező session bean. Az állapotmentes implementációs osztályt a @Stateless annotációval kell ellátni, az állapottal

rendelkezőt a @Stateful-al. Esetünkben látható, hogy @Stateless annotációval jelöltük meg mindkét bean-ünk implementációs osztályát.

A session bean- re referencia szerzésért, a kliensnek nem kell JNDI nevet adni a telepítésleíróban, mert alapértelmezetten a teljes specifikált nevet kapja meg. Ha más nevet akarunk adni, azt a @Stateless illetve a @Stateful annotáció name paraméterének értéket adva tehetjük meg. Referencia szerzésre az EJB kliensek(más EJB-k, webkomponensek) használhatják az @EJB annotációt az úgynevezett függőséginjektálás (dependency injection) segítségével.

Vagyis egy az @EJB –vel megjelölt business interfész típusú váltózóval a konténer biztosít bean példányt, ami megfigyelhető az alkalmazás szervleteiben. A konténeren kívüli klienseknek explicit módon kell a JNDI keresést végrehajtani, de itt már nem a home interfészre, hanem közvetlenül a bean példányra(vagyis kliensoldali csonkjára) szerezhetünk referenciát. A megszerzett példány metódusai hívásakor dobható java.rmi.RemoteException-t nem kötelező elkapni, mivel a konténer ezeket javax.ejb.EJBException-ba csomagolva dobja tovább, ez pedig a RuntimeException gyermeke, tehát nem kötelező kezelni.

3.2.4 Entitások

Az EJB 3 specifikáció egy új perzisztencia megoldást tartalmaz, ami a JPA (Java Persistence API). Ez is egy POJO (Plain Old Java Object) és Java SE-ben is alkalmazható. A JPA is az entity nevet használja, ugyanakkor ez nem tekinthető az entity bean technológia új verziójának, mivel az EJB 2.1-et a JPA teljese egészében tartalmazza.

O-R leképezés annotációkkal:

Először egy argumentumot nem váró konstruktorral rendelkező osztályt kell írnunk. Ennek az osztálynak tartalmaznia kell a megfelelő argumentumokat, és getter/setter párosokat. Ezután annotációkat kell alkalmaznunk. Az osztályt @Entity annotációval, az elsődleges kulcsot pedig @Id annotációval kell megjelölnünk. Ezekhez a javax.persistence csomagot kell importálni. Alapesetben a tábla neve az osztályéval, az oszlopnevek az attribútumokkal lesznek megegyezőek, ez viszont felüldefiniálható, ahogyan az látszik is az alkalmazás User és Transfer osztályában. Osztály szinten alkalmazható a @Table name paramétere vagy az

@Entity name paramétere, attribútumot jelölni pedig a @Column annotáció name paraméterével lehet. Entitások tartalmazhatnak tranziens adatokat, ezeket a @Transient kulcsszóval kell megjelölni. A perzisztens attribútumokra nézve megkötöttség van, hogy megfeleljenek az SQL adattípusokra. Ezek a primitív típusok, azoknak megfelelő objektum típusok, String, java.math.BigIntiger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.TimeStamp .
Az EJB 2.1-el szemben itt támogatva van az elsődleges kulcs automatikus generálása a @GeneratedValue annotációval, ilyenkor a típusa csak egész lehet továbbá strategy paraméterben lehet megadni a generálásra vonatkozó meghatározásokat, értéke a GenerationType enum 4 értékének egyike lehet. Ezek a következők:

- SEQUENCE: adatbázis által kezelt számláló segítségével generálódik a kulcs
- IDENTITY: adatbázis tábla autoinkrement oszlopára képződik az elsődleges kulcs
- TABLE: a @TableGenerator annotációval együtt használva egy tábla adott oszlopában adott sorában levő érték lesz a kulcs
- AUTO: adatbázis kezelő típusától függően a fenti három valamelyik lesz érvényben.

EntityManager és a perzisztenciakontextus:

Az entitások minden futási idejű perzisztenciával kapcsolatos kezelését a perzisztencia provider biztosítja. Ilyen provider a TopLink, Hibernate, melyek Java SE alkalmazáshoz külön osztálykönyvtár formájában kapcsolhatóak. Java EE alkalmazásszerveren a konténer biztosítja a JPA- implementációt. Az entitásokat EntityManager interfészen keresztül kezeljük komponens vagy alkalmazás fejlesztésekor. Minden EntityManager példány entitások egy bizonyos halmazával foglalkozik. Ebben a halmazban minden elsődleges kulccsal rendelkező perzisztens példányhoz egyetlen egyedi memóriabeli példány tartozik. Ezt a halmazt nevezzük perzisztenciakontextusnak. Egy JTA tranzakció több komponensen megy át, így szükséges, hogy ezekhez ugyanaz a perzisztenciakontextus tartozzon. Ha egy EJB komponens @PersistenceContext annotációt használva referenciát kér egy EntityManager-re, akkor a konténer feladata az aktuális JTA tranzakcióhoz tartozó perzisztenciakontextust hozzárendelni az EntityManager-hez. Ez látható is az alkalmazás UserManagerBean és a TransferManagerBean implementációs osztályában.

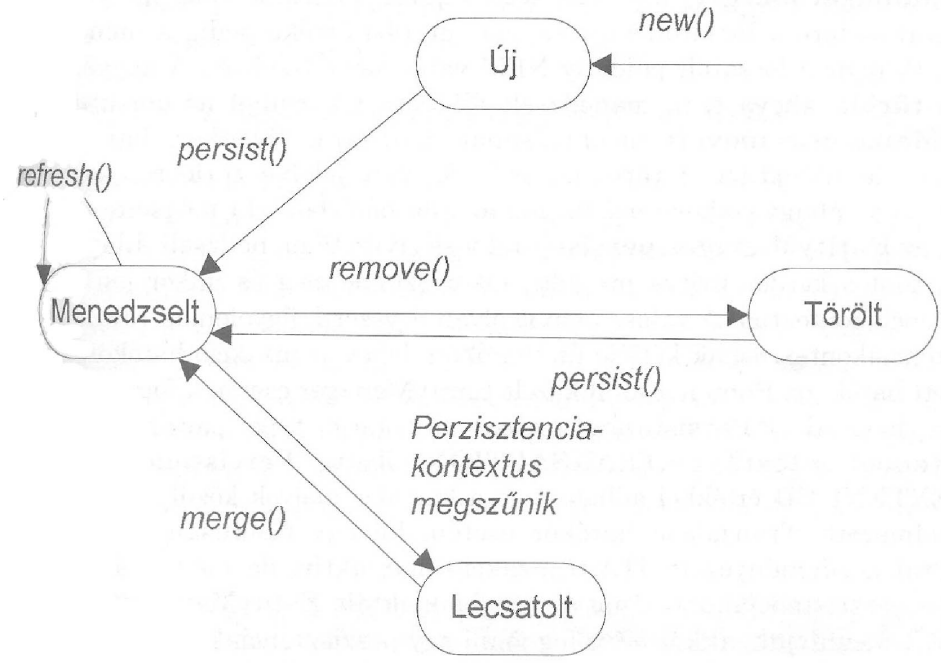
A perzisztenciakontextus hatóköre lehet tranzakció vagy kibővített. Ezeket a @PersistenceContext annotáció type paraméterében lehet megadni a függőséginjektálás közben EntiyManager-nél. Ez a paraméter felveheti a PersistenceContexTypeTRANSACTION és a PersistenceContexTypeEXTENDED érétket.
Tranzakció hatókör esetén létrejön egy perzisztenciakontextus, ha egy beinjektált EntityManager egy műveletét is meghívjuk, és a kontextus hozzákapcsolódik a adott tranzakcióhoz. Ekkor perzisztenciakontextus commit vagy rollback esetén egyaránt megszűnik és az entitások lecsatolt állapotba kerülnek. Meglehet az is, hogy nem jött még létre tranzakció, amikor meghívtuk az EntityManagert, ekkora hívás ideje alatt lesz csak meg a contextus, a hívás után azonnal megszűnik a perzisztenciakontextus.
Kibővített hatókör esetén csak állapottal rendelkező megkeresett vagy bean-be injektált EntityManager-hez kapcsolódhat. Több tranzakción is átmehet, mivel a session bean megszűnésekor szűnik csak meg.

Entitás életciklusa:

Az életciklust a 3.1-es ábrán találjuk. A new operátorral lehet **új** állapotba helyezni egy entitást. Itt még csak a memóriában van jelen, perzisztens adat nem létezik hozzá.
Az EntityManager.persist() –el **menedzselt** állapotba helyezhető, vagyis az entitás bekerül az EntityManager példányhoz tartozó perzisztenciakontextusba. Jellemzője, hogy tranzakció végén automatikusan szinkronizálódik a tartalma az adatbázissal.
Ha ezt nem szeretnénk megvárni, akkor az EntityManager.flush() metódussal ki lehet kényszeríteni a teljes perzisztenciakontextus kiírását az adatbázisba.
A menedzselt állapotban levő entitás (entitás halmaz) frissítését a refresh() metódussal tehetjük meg.

Perzisztenciakontextus megszűnésekor a menedzselt entitások **lecsatolt** állapotba kerülnek. Ilyen állapotban a végrehajtott műveletek az entitásokon nem kerülnek az adatbázisba. Ilyenkor DTO- ként működnek. Például, ha egy lecsatolt entitás a webrétegbe kerül, ott módosítjuk az entitást és vissza akarjuk hozni az adatbázisba, akkor menedzselt állapotba kell hozni, amit az EntityManagerMerge() metódussal tehetjük meg. Bemenő paramétere a lecsatolt entitás, visszatérési értéke a menedzselt entitás (a lecsatolt nem lesz menedzselt).

A törölt állapotba csak menedzselt állapotból kerülhet az entitás. Ezt az EntityManagerRemove() metódussal tehetjük meg. Ilyenkor gyakorlatilag a perzisztenciakontextusban van, de ki van jelölve törlésre, vagyis a tranzakció jóváhagyásakor megtörténik a törlés. Ha mégsem akarjuk törölni az EntityManager.persist()- el visszahozható menedzselt állapotba a tranzakció vége előtt. Ezeknek az életciklus metódusoknak egy része megfigyelhető a UserManagerBean és a TransferManegrBean implementációs osztályban.



3.1. ábra Entitás életciklus (Imre G. 2007)

Entitások közötti kapcsolatok:

A kapcsolatok O-R leképezéséhez a @OneToOne, @ OneToMany, @ManyToMany, @ManyToOne annotációk egyikét kell alkalmazni a multiplicitásnak megfelelően a kapcsolat másik végén levő változóra(többes esetén List, Set, Collection típusú). Az alkalmazásban a User entitásban ezek a receivedTransfer és a sentTransfer válzozók. Lehet egy vagy kétirányú a kapcsolat. Egyirányú esetén csak a tulajdonosi oldalról lehet elérni a kapcsolatot. A kétirányú esetén a tulajdonosi oldal egy-egy kapcsolatnál az, amelyik az idegen kulcsot tartalmazza. Egy-több kapcsolatnál a több oldal (ahol a @ManyToOne), több-több esetén bármelyik lehet a tulajdonosi oldal, mivel egy kapcsolótábla tartalmazza az idegen kulcsokat. A kétirányú kapcsolat inverz oldala köteles hivatkozni a tulajdonos oldali

attribútumra. Látható, hogy a transfer a tulajdonosi oldal, így a user entitás hivatkozik a tulajdonosi oldal attribútumára a @OneToMany sorok mappedBy attribútumával. Esetünkben a transfer entitásban recipient és a sender változója van megjelölve @ManyToOne annotációkkal, ezzel hivatkoznak a user entitásra. A kapcsolat kardinalitását meghatározó annotációk cascade paraméterével állítható be a kaszkádolás. Erre szükség is van, hiszen a fejlesztő feladata EJB 3 esetén a kapcsolat konzisztens kezelése. Alapesetben semmi sem kaszkádolt.

A lehetséges kaszkádolások: CascadeType.REMOVE/REFRESH/PERSIST/MERGE/ALL. Ezekkel az attribútumokkal érhető el, hogy a kapcsolódó táblában is végbemenjenek a fent említett műveletek hatásai.

Kapcsolatoknál fontos kérdés, hogy a kapcsolódó entitások is a memóriába kerüljenek az adott entitás memóriába kerülésekor. Ha bekerülnek, akkor mohó, ha nem akkor lustabetöltésről van szó. Alapesetben ez mohó (FetchType.EAGER), de a kapcsolatdefiniáló annotációk rendelkeznek egy fetch paraméterrel, amellyel lustává(FetchType.LAZY) tehető.

Lekérdezések entitásoknál:

A legegyszerűbb lekérdezés, ha elsődleges kulcs alapján keresünk entitást. Ez a következőképpen néz ki, az EntitManager interfészen keresztül tehetjük meg.

```
@PersistenceContext()
   EntityManager em;
public User findUser(Long id)
   {
      return (User) em.find(User.class,id);
   }
```

A bonyolultabb lekérdezések a Query interfészt megvalósító objektummal térnek vissza. A lekérdezést megadhatjuk EJB-QL vagy natív SQL nyelven.

@NamedQurey annotációval adhatunk meg statikusan előre definiált lekérdezést, ahogy ez a user és a transfer osztályban látható is. Ez ManagerBean-ekben az EntitManager Query createNamedQuery(String queryname) metódusával használható. Ezen kívül van lehetőségünk az EntityManager natív sql törlést, módosítást végző Query createNativeQuery(String sqlString) metódusát használni. Alkalmazhatjuk még az EJB-QL dinamikus lekérdezését Query createQuery(String ejbqlString) metódussal.

Tranzakciókezelés beállítása entitásoknál: Java EE környezetben a JTA tranzakciókezelés az általános. A konténerhez tartozó tranzakciómenedzser közvetlenül a JDBC drivert(erőforrás kezelőt) hívja az EntityManager kihagyásával. A tranzakciós attribútumokat a session bean-ek metódusaira a @TransactionAttribute annotációval meg lehet adni. A TransactionAttributeType enum–ban megadott MANDATORY, NEVER, NOT, SUPPOTED, REQUIRED, REUIRS_NEW, SUPPORTS étékeket veheti fel a tranzakciós attribútum.

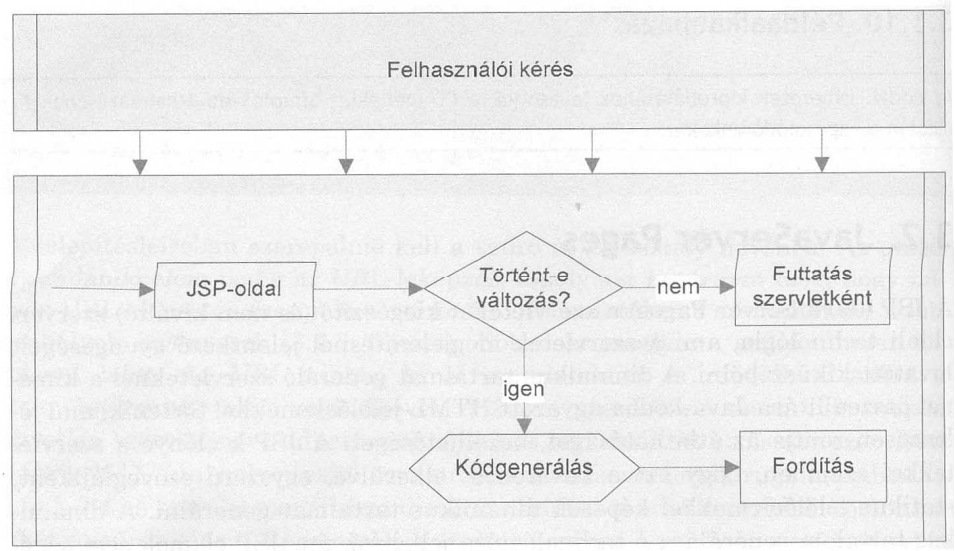
3.3. Servlet, JSP

Servlet:

A szervletek Java nyelvű osztályok, melyekkel dinamikus webes tartalmat fejleszthetünk. A szervletek mindig egy **webalkalmazás** részei, melyek esetünkben tartalmaznak JSP oldalakat és egyéb java osztályokat, konfigurációs fájlokat. A javax.servlet csomagban definiált szervletek segítségével tetszőleges protokoll szerinti kérésre generálhatunk választ. A javax.servlet.http csomagban levő osztályok és interfészek kifejezetten http kérések feldolgozására alkalmasak, melyeket webalkalmazásokhoz használnak. A **szervletkonténer, webkonténer** egy szoftverkomponens, mely a szervlet életciklusát kezeli. A webszerverrel együttműködve biztosítja a kérések illetve válaszok megfelelő helyre juttatását. A webkiszolgáló fogadja a kéréseket a böngészőtől, és ha ezek egy szervletre vagy JSP oldalra irányulnak, akkor a vezérlés a szervletkonténerhez kerül, lefut a kért erőforrás és a választ visszaküldi a kiszolgálóhoz, aki a klienshez továbbítja azt. Az oldal vezérlő logikáját szervletekben érdemes magvalósítani.

JSP:

Ez egy a szervleteket kiegészítő technológia, amelyekben a megjelenítés megvalósítását érdemes megadni. Szervleteknél a kimenetet eredményező java kódba ágyazott HTML jelölőelemek jelentősen rontják az átláthatóságot, kezelhetőséget. A dinamikus tartalom generálását JSP elemek segítségével érdemes tenni. A JSP oldalak szervletekké fordulnak le, és az ehhez tartozó doXXX() metódus hajtódik végre. JSP oldalhoz érkező kérés esetén valójában egy speciális webkonténer által generált szervlethez érkezik a kérés. Ez megvizsgálja, hogy a JSP oldal újabb-e a hozzá tartozó szervletnél, ha igen akkor szervletté transzformálódik, és ide jön a kérés(3.2- es ábra), ha nem akkor JSP-t kikerülve a szervlethez megy egyenesen a kérés, mivel nem kell újragenerálás.



3.2 ábra – JSP oldal szervletté alakítása (Imre G. 2007)

3.3.1. Servlet, JSP életciklusa

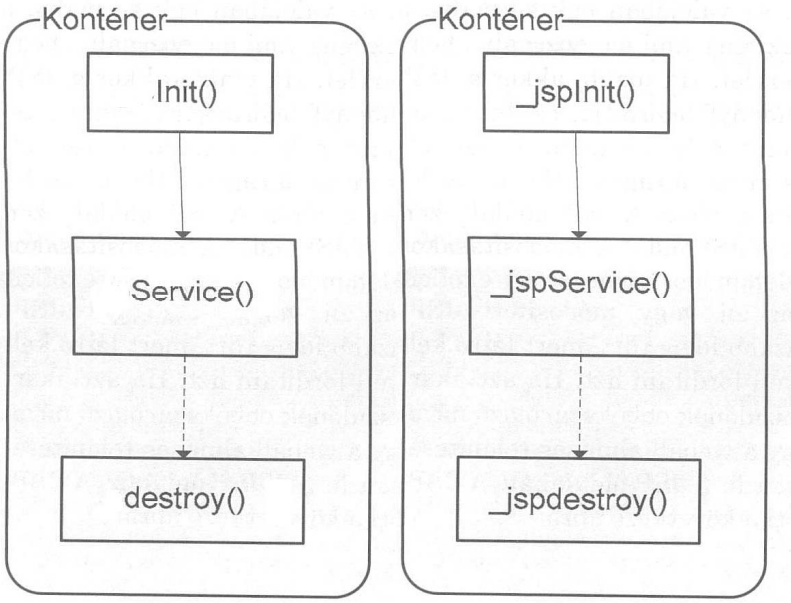
Servlet:

Az életciklus három fázisra bontható fel: inicializáció (init), kiszolgálás (service) és az eltávolítás (destroy), megvalósítása kötelező. Kérés érkezésekor a webkonténer betölti a szervletosztályt és példányosítja, ha még nincs. Ezután meghívódik az init metódus, amelyben végrehajtódik az inicializáció, és ha kell az erőforrások lefoglalása. Minden kérés érkezésekor meghívódik még a service metódus, ami két paramétert kap. Ezek a request és response. A HttpServlet –böl származtatott szertvletnél a service metódus, úgy van felüldefiniálva, hogy a http kérés típusától függően a megfelelő doGet(), doPost() … metódusoknak továbbítódjon a ServletRequest,ServletResponse – ból származó HttpServletRequest, HttpServletResponse típusú kérések és válaszok. Így mi nem definiáljuk fölül a service metódsut, hanem a doGet() és doPost() metódusokat írjuk meg céljainknak megfelelően. Több kérés esetén több szál indul, melyek mindegyike ugyanazon szervletpéldány service metódusát hívja meg, így nekünk kell gondoskodni a szálkezelésről. Syncronized kulcsszóval ellátni a kód kritikus szakaszait, kerülni a példányváltozókat, vagy az EJB rétegbe tenni a konkurens műveleteket. Ha egy ideig nem érkezik kérés, akkor a webkonténer dönthet úgy, hogy eltávolítja a memóriából a szervletet, de előtte meghívódik a destroy metódus, mely elengedi a fenntartott

erőforrásokat. Az alkalmazásban levő valamennyi szervletben megfigyelhetőek a service által továbbított kéréseket, válaszokat felhasználó metódusok. A szervlet életciklusa a 3.3 – as ábrán látható.

JSP:

A 3.4 –es ábrán látható. Itt csak a metódusok nevében van különbség a szervlethez képest.



3.4 – szervlet életciklusa 3.3 JSP életciklusa
(Imre G. 2007)

3.3.2. Http protokoll

Ez egy kérés válasz alapú állapotmentes protokoll, ami a kliens és a webszerver közötti kommunikációt teszi lehetővé. Egy http kérés egy parancsból áll, melynek két paramétere van. Első az a lekérendő erőforrást a második pedig a használt protokoll verzióját azonosítja. Emellett van egy fejrész és egy törzsrész, mely üres is lehet. A válaszban egy válaszkód után a fej és a törzsrész következik. A szervlet service metódusa különböző java metódusokhoz továbbítja a kérést. Ezek lehetnek: GET, HEAD, POST, DELETE, OPTIONS, TRACE, minden másra bad request a válasz.

A doGet és a doPost a leggyakrabban használt, mindkettő erőforrás lekérésre alkalmazható. A doGet – tel az URL –ben lehet korlátozott méretű adatot továbbítani. A doPost – tal pedig a törzsben lehet adatot továbbítani, mérethatár nélkül. PUT –tal erőforrásokat tölthetünk fel a

webszerverre, DELETE –tel törölhetjük azokat. OPTIONS: milyen műveletet végeztek az adott erőforráson. TRACE: kérés pontos mása juttatható a klienshez vissza. HEAD: GET-hez hasonló azzal a különbséggel, hogy nincs törzs rész.
Http válaszkódok:

- 401: a kliensnek nincs joga a kért erőforráshoz
- 404: a kért erőforrás nem található
- 405: a kért http parancs nem támogatott
- 500: belső szerverhiba. A szervletünk kezeletlen kivételt dob.

3.3.3 Válasz generálás, Válasz fejlécek, Válasz átirányítás, Kérés feldolgozás, Űrlapadatok feldolgozása, Kérés delegálása

A **válasz generálása** szervleteknél a HttpServletResponse típusú objektummal történik. Ennek a getWriter(), getOutputStream() metódusát használva írhatunk a kimenetre. Az előbbivel szöveges a másodikkal bináris adatot (képet, hanganyagot, dokumentumok) küldhetünk. A getWrite()- ot használva három metódust használhatunk: println(), print(), write().

A HttpServletResponse segítségével lehetőségünk van megváltoztatni a válasz fejlécet, hogy pl. a szerverrel kapcsolatos illetve a generált tartalommal kapcsolatos információt küldjünk. Ezen metódusok:

- void addHeader(String name, String value): az adott nevű és értékű fejlécet ad hozzá a fejrészhez.
- boolean containsHeader(String name): jelzi, ha az adott nevű fejléc már szerepel.
- void setHeader(String name,String value): adott nevű fejléc értékének beállítására szolgál, létező fejléc értékét felülírja.
- vois setIntHeader(String name,String value): mint a setHeader, csak a beállítandó érték típusa int.
- void setDateHeader(String name,long date): dátum típust tudunk beállítani(1970.jan.1. óta eltelt időt kell megadni milliszekundumban, java.util.Date getTime() metódusával megadhatjuk)

- void addIntHeader(String name, int value): addHeader –hez hasonló, csak int értéket ad hozzá.
- void addDateHeader(String name,long date): addHeader-hez hasonló, csak dátum értéket adunk hozzá, mint a set-es változatnál, csak itt nem íródik felül a régebbi.

A válasz átirányítása: gyakori használat miatt a szervlet API egy metódussal könnyíti meg. Ez a send.Redirect(String url), aminek bemenő paramétere a célerőforrás URL. Itt a webalkalmazás gyökeréből nézve abszolút útvonalat adjunk meg. Ennek alkalmazására is van példa az alkalmazás szervletei között. Ha az applikáció nem a gyökér URL-ben van akkor a sendRedirect(request.getContextPath() + ”/erőforrás”) használandó.

A kérés feldolgozása: A kérés elérése a feldolgozómetódus HttpServletRequest segítségével történik. A kérésosztály a ServletRequest interfészből öröklődik. A kliensoldali alkalmazásról, a várt válasz típusáról a kérés fejrészben található fejlécekből informálódik a szerver. A következő metódusokkal lehetséges a kérés olvasása:

- String getHeader(String Name): az adott nevű fejléchez tartozó értéknek megfelelő stringgel tér vissza
- Enumeration getHeaders(String name): sztringeket tartalmazó enumerációval tér vissza, ami az adott nevű fejléchez tartozó értékeket tartalmazza.
- Enumeration getHeaderNames(): sztringeket tartalmazó enumerációval tér vissza ami a fejrészben található fejléceket tartalmazza.
- int getIntHeader(String name): adott nevű fejléchez tartozó int értékkel tér vissza, nem létező fejléc esetén -1 –gyel.
- Long getDateHeadre(String name): adott nevű fejléchez tatozó dátum értékkel tér vissza, long típusként kapjuk az 1970.jan1. óta eltelt milliszekundumok számát.

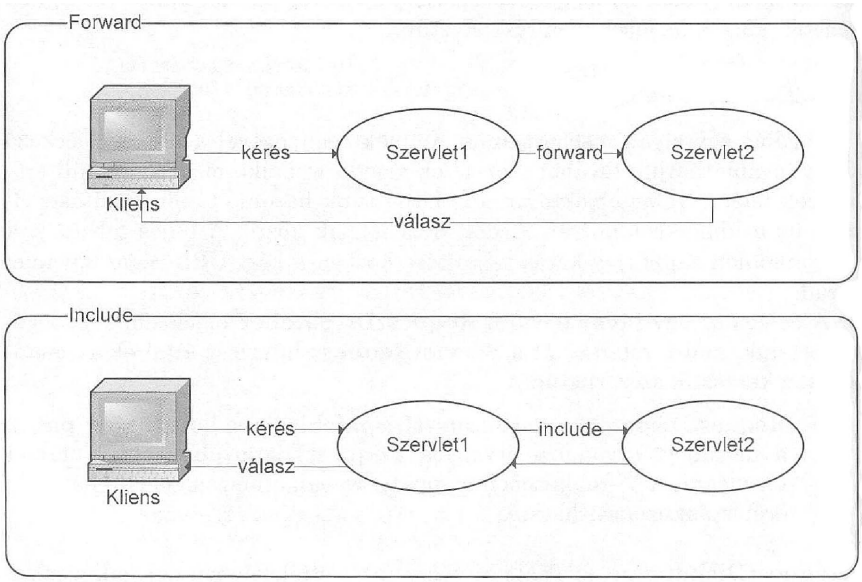
Űrlapadatok feldolgozása: Az űrlapok bemeneti vezérlői a szervletoldali feldolgozás szempontjából két részre bontható. Egyik a gombok, bemeneti mezők, listák, melyekkel az adatok kinyerése a szervlet API segítségével könnyedén elvégezhető. A másik a fájlok feltöltése, ahol nem áll rendelkezésre támogatás a szervlet API –ban(külső fájlfeltöltő modulok használata javasolt). Az adatok küldésének módját (MIME típust) a FORM enctype

attribútumával állíthatjuk, ennek értéke alapértelmezésben application/x-www-form-urlencoded. Ebben az esetben az űrlapadatok kinyerésére az alábbi metódusok használhatók, amelyek közül láthatunk is az alkalmazásban a form-kat tartalmazó JSP-hez tartozó szervletekben:

- String getParameter(String parameterName): adott nevű paraméter értékét kapjuk meg sztringben,
- String[] getParameterValues(Srting parameterName): ha többször előfordul az adott nevű paraméter, akkor mindegyikhez tartozó értéket megkapjuk,
- Enumeration getParameterNames(): a kérésben levő paraméterek nevét adja vissza.

A kérés delegálása: Egy olyan mechanizmus, mely segítségével a klienstől érkező kérést továbbíthatjuk más szervletek illetve webalkalmazáson belüli szervletek felé. Maga a delegálás a javax.servlet.RequestDispatcher objektummal történik, erre referenciát a ServletRequest interfésszel szerezhetünk: RequestDispatcher getRequestDispatcher(String path). Kétféleképpen delegálhatjuk a kérést, ami a 3.5 –ös ábrán szemléletesen láthatunk:

- forward(ServletRequest request, ServletResponse Response): a kérés továbbítása hajtódik végre, a válasz generálása a fogadó erőforrás feladata, hívási lánc esetén az utolsó feladata(a küldő szervlet csak előfeldolgozást végezhet). Ha a fogadó erőforrás statikus fájl akkor a tartalma belegenerálódik a válaszba. Ha dinamikus a fogadó erőforrás, ami lehet szervlet, JSP, akkor dinamikusan generálják ezek a választ.
- include(ServletRequest request, ServletResponse Response): Ezzel más erőforrás válaszát fűzhetjük a válaszba. A továbbítás hasonló a forward –hoz néhány megkötéssel. A befűzött erőforrások a válasz fejrészét nem változtathatják, a státuszkódot és nem zárhatják le a választ.



3.5 ábra- Kérés továbbítása (Imre G. 2007)

3.3.4. Szkriptelemek, Direktívák, Megjegyzések, Akcióelemek, Core elemek, UEL

A szkriptelemek: Ezekkel tehető dinamikussá egy oldal a legkönnyebben. Három féle szkript elemet különböztetünk meg: deklaráció, kifejezés, és a szkriptleteket.

1. Deklaráció: az osztály számára elérhető változó, metódus deklarációk. Szintaxisa: <%!deklaráció%>. Az így deklarált változó az oldalból generált szervlet osztályba kerül bele. <%! String s=”szöveg” %>
2. Kifejezések: a kiszolgáló metódusban szereplő print() vagy write() hívások. Szintaxisa: <%= kifejezés%>. Ez a JSP oldal végrehajtása során kiértékelődik, sztinggé konvertálódik és a válaszba kerül. Pl. <%= new java.util.Date()%> a mai dátum kiíratása, pontosvesszőt nem kell tenni a kifejezés végére.
3. Szkriptletek: a kérésnek megfelelő doXXX() metódusban elhelyezett kódrészletek. Szintaxisa: <%kód%>. A kód helyére beirt programkód a generált szervlet doXXX() metódusba kerül bele, nem lehet benne osztály, metódus definíció.

A szkriptleteknél, kifejezéseknél hozzáférhetőek az ún. implicit objektumok. Ezek automatikusan definiálódnak minden JSP oldalon, és mindig ugyanolyan néven lehet elérni őket. Ezek közül néhány:

- request, response: a kérést és a választ reprezentáló HttpServletRequest típusú objektum
- out: JspWriter típusú objektum, ezen keresztül írhatjuk a választ, mint a szervleteknél a PrintWriter-rel.
- pageContext: az adott oldalhoz tartozó PageContext típusú objektum, amely egységes hozzáférést biztosít különböző hatókörű objektumokhoz, mint látható a usermanagement.jsp szkripletében a user objektum megszerzéséhez.
- page: a this szinonimaként tekinthető
- session: a kéréshez tarozó HttpSession típusú objektum, amely automatikusan létrejön.

Direktívák: Ezek a konténernek küldött üzenetek, segítségükkel az oldal tulajdonságait állíthatjuk be, más oldalakat illeszthetünk be.

- include: szintaxisa <%@ incluse file=”relatív url” %>. Hatására az url-ben magadott fájl statikusan bemásolódik a JSP- be, és ez transzformálódik szervletté.
- taglib: A JSP oldal olyan felhasználó által definiált elemeket használ, melyet az uri attribútumban megadott elemkönyvtár definiál. Ezt a prefix attribútumban megadott előtaggal használjuk az oldalon. Szintaxisa: <@ taglib uri=” uri” prefix=”prefix” %>
- page: Az egész oldalra jellemző értékeket állíthatunk be. Szintaxisa: <@ page jellemző=”érték” %>, néhány jellemző:
  - import: azok az osztályok és csomagok melyeket a jsp oldalon használni akarunk.
  - session: értékét false –ra állítva nem jön létre az oldalon a session objektum.
  - errorPage: itt megadott oldalnak adódik át a vezérlés ha kezeletlen kivétel dobódik az oldalon.

Megjegyzések: Kétféle megjegyzést helyezhetünk el a JSP oldalon. Az egyik a <%-- --%> között elhelyezhető, ami nem kerül bele a válaszba, ez a JSP megjegyzés. A másik a <!-- --> között levő megjegyzések a HTML megjegyzések és ezek bekerülnek a válaszba, és ha JSP elemet tartalmaz, az végrehajtódik.

Akcióelemek: A leggyakrabban használt akcióelemek a következők.

jsp:include : Az url-ben magadott erőforrás, ha statikus akkor egyszerűen bemásolódik a válaszba. Ha a page paraméternél szervletet, JSP- oldalt adunk meg, akkor a vezérlés ehhez kerül, és a válaszba kerül a kimenete, majd visszaadja az irányítást az akcióelemet tartalmazó oldalnak. Egy példa az alkalmazás JSP oldaláról: <jsp:include page="../../head.jsp" />.

jsp:forward : Ebben az akcióelemben megadott erőforrásnak véglegesen továbbítódik a kérés, hatása megegyezik a RequestDispatcher forward() metódusával, amit a szervleteknél figyelhetünk meg. Szintaxisa: <jsp:forward page="{relatív url| <%=kifejezés%>}"/>

jsp:usebean, jsp:setProperty, jsp:getproperty : JavaBean komponenseket tölthetünk ezekkel az elemekkel, és később meghívhatjuk azok metódusait. A JavaBean komponensek olyan osztályok, melyek példányai bizonyos jellemzőkkel rendelkeznek és bizonyos követelményeket teljesítenek. Jellemző lehet csak olvasható, vagy írható is. A JavaBean-ek segítségével sokkal egyszerűbbé tehető a JSP oldal.

Az alkalmazásból egy példa:

```
<jsp:useBean id="registerform" scope="session" class="transfersystem.web.forms.RegisterForm">
        <jsp:setProperty name="registerform" property="birthLoc" value="" />
</jsp:useBean>
```

Az elem inicializálja a beant, vagyis csak akkor hajtódik végre a setProperty beállító rész, ha az objektumot nem megtalálta a useBean, hanem inicializálni akarja. Ha useBean már létrehozta vagy megtalálta az adott nevű jellemzőt, akkor ezt a getProperty-vel szúrhatjuk a válaszba.

A useBean hatókörét a scope attribútummal állíthatjuk be. Ezek lehetnek:

- page:a bean az adott JSP oldalon érhető, amíg a válasz nem megy klienshez, vagy másik weberőforráshoz.
- request: elérhető a bean minden JSP oldalról, melyhez a kérés eljut.

- session: adott munkamenet alatt elérhető, ha nincs letiltva page direktívával a session.
- application: mindem JSP oldalon elérhető, amely ugyanabban az alkalmazásban van, mint létrehozó oldal.

Unified Expression Language: Az első EL a JSTL 1.0 részeként jelent meg, hogy adatokhoz könnyen lehessen hozzájutni JSP oldalon. Az UEL-nél kétfajta kiértékelési módszer használható.
Azonnali kiértékelés: ${param.id}, itt pl. egy űrlap paramétereit tartalmazó param implicit objektum id paraméterének értéke az oldal generálásakor azonnal kiértékelődik. Egyszer hajtódik végre. A következő ugyanezt eredményezi: ${param[”id”]}
A halasztott kiértékelésnél a JSP konténer helyett a kifejezés kezelését az UEL-t használó technológia veszi át. JSF esetén a kiértékelés a JSF életciklusához igazodik. Szintaxisa: #{kifejezés}

Core elemek: Először néhány szó a JSTL – ről (JavaServer Pages Standard Library). A megjelenítést véghezvivő JSP oldalak kialakításkor is szükség lehet összetettebb akcióelemkre, amelyekkel könnyebben elérhetjük célunkat. A JSTL a JSP 2.1 specifikáció része. JSTL akcióelemekkel alakíthatjuk dokumentumunkat.

Használatukhoz taglib direktíva szükséges, a <c:out> elemet említem meg először, amelyet többször is használok a kimenetre íráshoz az alkalmazásban.
Ez a <%@taglib uri=”http://java.sun.com/jsp/jstl/core” prefix=”c””%> .
A <c:out> elem segítségével a JspWriter objektumnak küldhetünk kifejezéseket, egyszerű szövegeket. Egy másik core elem <c:catch>, ennek használatával az oldalon dobott kivételeket kaphatjuk el.
Az iterációt a <c:foreach> és <c:forTokens> akcióelemmel alkalmazhatjuk. A tömb tartalmazhat primitív típusokat vagy objektumokat is. Az iteráció a java.util.Collection/Terator/Enumeration–t implementáló objektumon vagy a ”,”-vel elválasztott sztringen mehet végig. <c:foreach> attribútumai:

- var: annak a változónak a neve, amibe az aktuális elem kerül
- items: elemek gyűjtemény, amin iterálunk

- begin: a kezdőindex értéke
- end: ennél a indexnél fejeződik be az iteráció
- step: lépték megadása
- varStatus: az iteráció státuszát tartalmazó váltózó

<c:forToken> : egy sztring tetszőleges karakterrel elválasztott elemein lehet iterálni ezzel az akcióelemmel. Attribútumai megegyeznek a foreach-nél látottakkal eggyel kiegészülve. Ez pedig a delims, ami az elválasztó karakterek listája.
<c:set>elemmel a JavaBean jellemző érékét a setProperty-hez hasonlóan. Attribútumai:

- value: kifejezést adhatunk meg, amin átadódik
- var: ebben a változóban tárolódik a kiértékelés eredménye
- scope: page, scope, request, application érvényességi kör lesz érvényes a változóra
- target: JavaBean komponens nevét adhatjuk meg, ennek a jellemzőjét állíthatjuk be.
- property: a target-ben megadott JavaBean jellemzőjének nevét adjuk meg itt.

A <c:set> akcióelemnek két féle falhasználása van. Beállíthatjuk JavaBean komponens jellemzőit, ahogy fentebb említettem is, szintaxisa: <c:set value=”kifejezés” target=”célobjektum” property=”célobjektum jellemzője”> . Másrészt beállíthatjuk egy változó értékét: <c:set var=”változó neve” value=”érték”>.
<c:if>: ennek törzse csak akkor hajtódik végre, ha a test attribútuma a kiértékelést követően true értékű lesz. Egy var változóban adhatjuk meg a kiértékelés eredményét tartalmazó váltózó nevét. A scope attribútummal megadhatjuk még a var-ban megadott változó érvényességi körét(page,request.session,application).
<c:choose>, <c:when>, <c:othervise> használatával kiválthatjuk a szkriptletek switch és if-else tagjait. A choose–on belül a when test attribútumában adható meg a feltétel. Ha több when van, akkor a kiértékelés sorban történik, és az hajtódik végre, amelyiknek a test attribútuma true lesz a kiértékelés után. Ha az egyik eset sem lesz true, akkor az otherwise attribútumnak megy át a vezérlés.

4. Összefoglalás

A diplomamunkában megpróbáltam bemutatni az Enterprise Java Bean-ek működését. Ezen belül a session bean-el és az entitásokkal foglalkoztam. Egyszerű példán keresztül röviden ismertettem az EJB 2.1 komponenseit, mind a session mind az entity bean-t. Ezt követően egy nagyobb alkalmazással bemutatásra kerültek az EJB 3 technológia által nyújtott lehetőségek. Az alkalmazás a NetBeans IDE környzetben készült, mint Enterprise Application, ami webes felülettel érhető el. A webes felület kialakítása szervletek és JSP oldalak segítségével történt, amely továbbfejlesztési lehetőségeket rejt. Célok között szerepel, az EJB 2.1-es technológiák valamennyi, a dolgozatban említetett ágával alkalmazást készíteni, mind webes, mind vastag klienssel. Az alkalmazás továbbfejlesztésében nagy szerepet kaphat a Java Server Faces technológia, amely igen elterjedt webes keretrendszer, amely a JSP technológiára épül.

Irodalomjegyzék

- **Dirk Louis, Peter Müller, Java 5 Belépés a programozás világába, Panem könyvkiadó, 2006**
- **Imre Gábor, Szoftverfejlesztés Java EE platformon, SZAK Kiadó, 2007**
- **Nyékiné Gaizler Judit, J2 EE útikalauz Java programozóknak, ELTE TTK Hallgatói Alapítvány, 2002**
- **Java EE 5 Tutorial - http://java.sun.com/javaee/5/docs/tutorial/doc/**