

Diplomamunka

Szaniszló Viktor

Debrecen

2010

Debreceni Egyetem

Informatikai Kar

Webes alkalmazásfejlesztés

Témavezető:

Kósa Márk

egyetemi tanársegéd

Készítette:

Szaniszló Viktor

programtervező matematikus

Debrecen

2010

Tartalomjegyzék

1. Bevezetés.....	1
2. Java EE szolgáltatások.....	2
3. A három- vagy n rétegű architektúra	5
4. Az Ada és az EJB	7
5. Enterprise Java Beans.....	8
5.1. Session Bean	9
5.2. Entity Bean	11
5.3. Java Persistence API	12
5.3.1. JPA Entitások	12
5.3.2. Lekérdezések	14
5.3.3. Entitások közötti kapcsolatok.....	15
5.3.4. Persistence Unit	18
5.3.5. Perzisztenciakontextus	18
5.3.6. Entitások életciklusa	19
5.3.7. Öröklés.....	21
5.3.8. Tranzakciókezelés	22
6. Webes keretrendszerek.....	25
6.1. Webes keretrendszerekkel szembeni igények	25
7. JavaServer Faces.....	28
7.1. Működése.....	28
7.2. Nézet.....	28
7.3. Menedzselt beanek	29
7.4. Konverterek, validátorok	30
7.5. A JSF navigációs modellje	31
7.6. Többnyelvű JSF oldalak	32
8. Biztonság	33
9. JavaEE szerepalapú biztonsági modell.....	34
10. Az alkalmazás bemutatása.....	35
10.1. Az alkalmazás által használt eszközök	35
10.2. Design kialakítása.....	37
10.3. Képek kezelése.....	37
10.4. Az alkalmazás funkciói.....	38

10.4.1. Alkalmazottak karbantartása	38
10.4.2. Vásárlók karbantartása.....	42
10.4.3. Készlet karbantartása	43
10.4.4. Rendelések	46
10.4.5. Kosár	47
10.4.6. Vásárló felület	49
Összefoglalás	51
Köszönetnyilvánítás	52
Irodalomjegyzék	53

1. Bevezetés

Napjainkban legtöbben személyi számítógépet az internet miatt vesznek. Ez a mintegy 15 éve tartó folyamat, ipari igényeket vet fel, a webes alkalmazások minél hatékonyabb, minél gyorsabb, megbízhatóbb előállítására. Középpontjába állította a cégek fejlesztéseit, melyek jelentős hányada a webre irányul. Minden alkalmazáshoz igyekeznek webes elérést biztosítani, még ha ez gyakran felesleges és értelmetlen is. Ennek ellenére ez a tendencia egyre erősödni látszik, valószínűleg az ezen a téren elért szolgáltatások és eredmények miatt, melyek jelentősen megkönnyítik a fejlesztést. Továbbá, mert a fejlesztők is ezen a területen szereznek tapasztalatot, tehát biztonságérzetet jelent számukra az általuk megismert eszközök használata.

Kétségtelen tény, hogy a webes technológiák alapvető ismerete mára követelménnyé vált és ezáltal a Java, esetleg a .Net technológiák ismerete is. A felmerülő problémákra adott válaszok ugyanis e technológiák köré csoportosultak, mivel a Java kezdetektől kötődik a webhez, a szerveroldalról letölthető appletok révén, a .Net technológia pedig, piackihívó szerepe miatt újabb megoldásokra ösztönzi a szakembereket.

Az átlag felhasználók szempontjából, talán a legelterjedtebb webes alkalmazások az internetes áruházak. Ezért választottam magam is egy ilyen alkalmazás fejlesztését, hogy ezáltal ne csak a technológiákra, hanem egy gyakorlati életből vett alkalmazás fejlesztésére is szerezzek némi rálátást.

A webáruházak valószínűleg nem fogják átvenni az áruházak, boltok szerepét, amíg az érzékszerveinkkel kapcsolatos ingereket (szaglás, tapintás, ízlelés) nem képes közvetíteni. A személyes kontaktus hiánya, az eladó általi döntés megerősítés és egyéb összetevők miatt, leginkább valamilyen kiegészítő szolgáltatást nyújtanak. Általában a termékről valamilyen előismerettel rendelkező vásárlók rendelnek interneten keresztül vagy mozgásukban korlátozott személyek, akik számára ez jelentős könnyítést jelent a termékek elérhetősége szempontjából.

2. Java EE szolgáltatások

A Java nyelvnek jelenleg 3 kiadása van. A Standard Edition, amely a hagyományos desktop alkalmazások és appletek fejlesztését célozza meg. A mobil eszközökre történő fejlesztést megcélzó Micro Edition, mely a Standard Edition osztályainak és interfészeinek részhalmazát tartalmazza. Továbbá az Enterprise Edition, amely a vállalati méretű, elosztott rendszerek fejlesztését célozza meg.

A Java EE egy keretrendszert biztosít a rendszerek számára és olyan szolgáltatásokat nyújt, melyeket egy hasonló fejlesztés során magának a fejlesztőnek is implementálnia kellene. Ebben a futási környezetben általános, szabványosított megoldásokat kaphatunk, melyek biztosítják az egyes keretrendszerek közötti hordozhatóságot.

Egy ilyen keretrendszernek több alapvető szolgáltatást kell ellátnia, ezek a middleware-szolgáltatások, melyek közül bemutatok néhányat:

- **Perzisztencia:** Az adatok tárolásánál biztosítani kell, hogy a program karbantartása, leállítása vagy esetleges összeomlása esetén is megmaradjanak. Általában ezt adatbázis rendszerek segítségével biztosítják. Ehhez olyan támogatást kell nyújtania a keretrendszernek, mely elfedi a fejlesztő elől a gyártó specifikus részleteket, összeegyezteti az objektum orientált és a relációs szemléletmódot.
- **Többszálúság:** A rendszereknek támogatnia kell, hogy egy időpillanatban több felhasználó kérését dolgozza fel. Ezen kéréseket párhuzamos feldolgozását, a szálkezelés részleteinek elrejtésével, nyújtson szolgáltatást.
- **Tranzakciókezelés:** Az adatok konkurens elérésénél, biztosítani kell az adatok konzisztenciáját. A kliensek szempontjából azt, hogy úgy lássa a kliens, mintha csak ő kezelné az adatokat.
- **Monitorozás és beavatkozás:** Felmerülhet olyan igény, hogy egy rendszer működése közben változtatásokat kell alkalmazni vagy nyomon tudjuk követni a rendszer működését. Ehhez nyújt szolgáltatást, mely futási időben is igénybe vehető.

- **Távoli elérés:** A futási környezetnek a rendszer elosztott jellege miatt biztosítani kell számunkra olyan kommunikációs protokollokat, melyek támogatják mind a vastag, mind a vékony klienssel történő elérést. Web esetén http protokoll, vastag kliens esetén a távoli metódushívással történő eléréshez kell biztosítani ilyen kommunikációs lehetőséget.
- **Névszolgáltatás:** Az objektumok elérését egyedi nevekre történő hivatkozásokkal lehet megvalósítani, melyekhez egy olyan rendszert kell biztosítani, melybe neveket tudunk regisztrálni az egyes erőforrásokhoz. A nevek alapján más objektumok el tudják érni ezeket az erőforrásokat, a hálózati cím ismerete nélkül. Ezen problémák összességét a névszolgáltatás oldja meg.
- **Skálázhatóság:** Azt jelenti, hogy a terhelés növekedése ellenére sem tapasztalható komoly teljesítmény csökkenés, a programkód módosítása nélkül.
A skálázást két féle képen tehetjük meg:
 - *Függőleges skálázás:* Erősebb, ésszerű áron elérhető hardver eszközre történő cserét jelent.
 - *Vízszintes skálázás:* Egy klaszter kialakítását jelenti. Erre eleve fel kell készíteni a rendszert, a névszolgáltatással történő elérést kell biztosítani és távolról hívható komponenseket kell fejleszteni.
- **Magas rendelkezésre állás:** Egyes rendszereket rendelkezésre állásánál alapvető kritérium, hogy állandó elérést biztosítsanak. Például: webáruház. Esetleges hibák vagy technikai problémák miatt azonban ezek a rendszerek elérhetetlenné válnak, melyek akár jelentős bevételkieséssel is járhatnak. Erre legmegfelelőbb megoldás a vízszintes skálázás, mely biztosítja a szolgáltatás elérését, ha egy gép esetleg nem elérhető.
- **Biztonság:** A rendszerek mára megkövetelik a felhasználó azonosítást, jogosultságkezelést, adatvédelmet. Az alkalmazás szerverek olyan szolgáltatásokat nyújtanak, melyek megoldják ezeket a problémákat, így ezeket a területeket érintőlegesen kell ismernie a programozónak.

- **Aszinkron üzenetkezelés:** Amennyiben rendszerünk más rendszerekkel áll kapcsolatban, rendszerhiba lép fel, ha a kapcsolatban álló rendszerek valamelyike nem áll rendelkezésre, mikor együtt kéne működnie a rendszerünkkel. Ezt hibaként észleli a felhasználó. Az aszinkron üzenetküldés jól kezeli ezt a hibát, mert ha a partner rendszer nem áll rendelkezésre, akkor megőrzi az üzenetet, melyet akkor küld el, ha ismét kapcsolatba tud lépni a rendszerrel.

3. A három- vagy n rétegű architektúra

A nagyvállalati rendszerek felépítése egy réteges szoftver architektúra, melyben minden réteg jól definiált feladatot lát el. Az egyes rétegek mögött készen kapható szoftverek állnak. Ebbe a rétegelt környezetbe illeszkedik az alkalmazáserver, mely a Java EE futási környezete. A rétegelt architektúrák esetén minden réteg igénybe veszi a közvetlenül alatta levő réteg szolgáltatásait.

Három réteget lehet elkülöníteni:

- **Adatréteg:** az adatok perzisztens tárolásáért felelős réteg, mely felelős az elemi műveletek elvégzéséért. Általában ez valamilyen relációs vagy objektum relációs adatbázis szerver, de elképzelhető bármilyen adattárolási rendszer.
- **Üzleti logika:** az alkalmazási terület igényeinek megfelelő funkcionalitást biztosít, mely az adatréteg szolgáltatásait hívja. Ez a réteg a Java EE réteg, melyet az alkalmazáserverre telepítenek.
- **Kliensréteg:** a felhasználói interakciók fogadására alkalmas felület, mely a beavatkozások hatására hívja a megfelelő üzleti funkciót.

A klienseket két típusba sorolhatjuk:

- *Vastag kliens:* a hagyományos desktop alkalmazások sorolhatóak ide.
- *Vékony kliens:* a böngésző, amely biztosítja a megjelenítést. A vékony kliensek egyre inkább olyan szolgáltatásokat nyújtanak, melyek a vastag kliens felhasználói élményéhez hasonlóak.

A két klientsípus közti határ egyre inkább elmosódik és a vékony kliens szolgáltatásai miatt a webes felületű elérést preferálják a cégek.

A vékony kliensek kiszolgálásához egy további réteg is szükséges:

- **Webréteg:** mely a megfelelő üzleti logikát hívja meg és megfelelő választ generál.

A webréteg a kéréseit közvetlenül az alkalmazás szervernek továbbítja.

Ezen kívül általában egy újabb réteggel bővítik az architektúrát, mely a webréteg és az alkalmazáserver közé illesztenek:

- **Webszerver:** a statikus erőforrásokat szolgálja ki.

Ebből a felsorolásból látható, hogy az architektúra legalább három jól elkülönülő rétegre bomlik szét, mely rétegeket további szükséges rétegekkel bővíthetünk, tehát akár n rétegből is állhat.

Az alkalmazáserverre mi fejleszthetünk és telepíthetünk szoftver komponenseket, melyek az egyéb hozzá kapcsolódó rendszerekkel állnak kapcsolatban. Ezek a komponensek az üzleti logika és a webkomponens megvalósítását jelenti.

Az üzleti logikát Enterprise JavaBeans komponensekben valósítják meg, ezen komponensek távoli eljárashívással is elérhetőek, azaz RMI segítségével. A legelterjedtebb rendszer a CORBA, mely IIOP protokollal támogatja az objektumok elérését. Tetszőleges CORBA kliensből is hívhatóak az EJB-k, ezzel függetlenítve a vastag kliensek programozásának nyelvét. Ez azért fontos szempont, mert kihasználhatóak más nyelvek előnyei a kliens implementálásánál.

A webkomponensek megvalósításához általában a JSF azaz a Java ServerFaces technológiát használják, mely széles körben elterjedt a felületek a Model-View-Controller tervezési minta megvalósításának köszönhetően. Az üzleti logikát meghívhatja a kérés kiszolgálása során, mely ha egyazon szerveren fut az EJB réteggel, akkor hagyományos hívás is lehet, mely kiküszöböli a távoli eljárashívás overheadjét.

Az EJB réteg elérése akár XML webszolgáltatásokon keresztül is elérhető és a kliensek is hívhatnak más webszolgáltatásokat. Ezt a szolgáltatási modellt, szolgáltatás orientált architektúrának nevezzük.

Az alkalmazáserveren a komponensek konténerekben futnak, mely web- és EJB konténerre oszlik. A kliens oldalon applet- és vastagkliens-konténer található. Ezek a konténerek biztosítják a komponensek számára a szolgáltatásokat.

4. Az Ada és az EJB

Mielőtt ismertetném az EJB technológiát, összehasonlítom Java és az Ada nyelvet a típuskorlátozás szempontjából, mellyel rávilágítok az EJB módszertani hiánypótló szerepére.

Az Ada nyelv lehetőséget biztosít a típus korlátozására a **limited** kulcsszóval, mely azt jelenti, hogy egy típus esetén megtiltjuk az értékadás, az = és /= operátorok generálását a fordítónak. Ezzel a korlátozással, olyan típusokat hozhatunk létre, melyek példányai valamilyen funkcionalitást látnak el, melyet az EJB technológia Üzleti logikának hív, tehát értelmetlen rájuk a fent említett három művelet. Ilyen típus például az **Ada.Direct_IO** sablon csomag **File_Type** típusa, melynek egy példányával adott típusú direkt állomány kezelését lehet végrehajtani.

A Java nyelvben az egyenlőség vizsgálatot az **equals()** metódus szolgálja, mely programozó általi implementálással és nem fordító általi implicit módon történő generáltatással hozható létre az egyes osztályok esetén. Az értékmásolás műveletét szintén a programozónak kell implementálnia, melyet a **Cloneable** interfész implementálásához köt. A **clone()** metódus implementálásával válik másolhatóvá az objektum.

A fent elmondottakból kiderül, hogy a Java helytelenül választja szét az értékadás és az egyenlőségvizsgálat műveletét, melyet egyetlen interfész implementálásához köthetne. Valószínűleg többek közt azért vezeti be az EJB technológia a Session és az Entity Bean fogalmát, hogy ezzel orvosolja, ezt a módszertani hibát.

```
generic
  type Elem_Type is private;
package Sorok is

  type Sor is limited private;

  procedure Put(S : in out Sor; E : in Elem_Type);
  procedure Get(S : in out Sor; E : out Elem_Type);
  ...
end Sorok;
```

5. Enterprise Java Beans

Az Enterprise Java Beanek, vagy röviden EJB-k, olyan szerveroldali komponensek, amelyek a rendszer megírásánál építőelemekként használhatóak fel. Jelenleg háromféle EJB-t definiál.

- **Session bean**, valamilyen üzleti folyamatot, üzleti logikát reprezentál.
- **Entity bean**, az üzleti modell entitásait reprezentálja, mely egyértelműen megfeleltethető egy relációs adatbázis valamelyik sorának.
- **Message-driven bean**, az aszinkron üzenetfeldolgozást valósítja meg, amely vagy üzenetsorban érkezett, vagy topicban publikált üzenetet jelent.

Az EJB 2.1 technológia egyszerűsítéseként jelent meg az EJB 3, amely az EJB technológia terén jelentős előrelépést jelent. Kihasználva a Java 5 új eszközeit, elegáns megoldást nyújt a 2.1-es specifikáció egyszerűsítésére.

Az EJB 2.1 nehézkessé tette a fejlesztést, mivel egyszerre több fájlt kellett karbantartani, Entity beanek esetén felesleges metódusokat kellett megvalósítani, akár üres metódusként is és jelentős kódot jelentett egy EJB-metódus meghívása is.

A Java 5 egyik új eszköze az annotáció, mellyel olyan metaadatokat helyezhetünk el a forráskódba, amellyel befolyásolhatjuk, hogy az adott komponenst, hogyan használják a különböző eszközök és osztálykönyvtárak.

Az annotációkkal történő metaadat forráskódokban történő elhelyezése, az XDoclet technológia kiváltásaként is értelmezhető. Ez a technológia speciális szintaxisú megjegyzések segítségével állítja elő a távoli/lokális interfészt, a home/lokális home interfészt és még telepítésleíró is elő tud állítani.

Véleményem szerint ez a megoldás egyértelműen helytelen, hiszen a megjegyzés szövegszinten más szerepet tölt be, ami nemcsak hogy módszertanilag helytelen, de a kód olvashatóságát is jelentősen rontja. A fejlesztő esetlegesen figyelmen kívül hagyhatja, a megjegyzésbe ültetett metaadatot, ezáltal a hibajavítás nehézkessé válhat, amely a szoftver minőséget is rontja.

Az EJB 3 jelentőségét tehát nagyban növeli, hogy nyelvi eszközök segítségével ad választ az EJB 2.1 problémáira, az annotációk széleskörű alkalmazásával.

5.1. Session Bean

Az EJB 3 a Session Beaneknél azt az igényt célozta meg kielégíteni, hogy ne kelljen EJB-specifikus kiterjesztéseket végezni, amivel jelentősen növekedhetne az esetleg felesleges, akár implementáció nélküli interfészek száma. Ez jelentősen rontaná a kód átláthatóságát, karbantarthatóságát. Tehát az alapvető igény, hogy az implementációs osztályok POJO-k legyenek (Plain Old Java Object).

Az EJB koncepciójában a kliensek egy interfészt megvalósító, konténer által generált csonkot hívnak. Ezért az implementációs osztály és a business interfészt össze kell rendelni, melyet az `implements` kulcsszó biztosít. Azt, hogy lokális vagy távoli interfészről van-e szó, annotáció segítségével adhatjuk meg:

- **@Remote**: jelöli a távoli interfészt
- **@Local**: a lokális interfész megjelölésre szolgál.

Mivel több interfészt is implementálhat egy osztály, ha szükséges, akkor az annotációk paramétereiben felsoroljuk ezeket.

Egy session bean rendelkezhet állapottal vagy lehet állapotmentes is:

- **Állapottal rendelkező** beanben több híváson keresztül állapotot tárol a kliens, vagyis mindig ugyanazzal a beannel kell kommunikálnia. Ezt a konténerrel is tudatnunk kell, hogy egy adott bean rendelkezik állapottal, amit a **@Stateful** annotációval jelölünk.
- **Állapotmentes** beanek nem tárolnak állapotot, mert egyes beanek esetén felesleges az állapot tárolása. Ezeknél nem számíthat arra a kliens, hogy ugyanazzal a beannel fog kommunikálni. A konténer számára a **@Stateless** annotációval jelöljük.

Nem feltétlenül csak valamilyen tudatos megkülönböztetés miatt van szükség erre a tagolásra, hanem a memória végeessége miatt is, mivel az állapot nélküli beanek újrafelhasználhatóak, oly módon, hogy ha már egy kliensnek nincs szüksége rá, akkor egy másik kliens csatlakozhat hozzá. Amennyiben egy állapottal rendelkező beant egy másik klienshez kell kapcsolni, akkor az állapotot valamilyen háttértárra menti a konténer, és amikor ismételtén visszakapja az eredeti kliens, akkor az állapotot vissza kell töltenie.

Továbbá a **@Stateless** vagy a **@Stateful name** paraméterében adjuk meg az általunk preferált JNDI nevet, ha az alapértelmezett nem lenne megfelelő, például valamilyen névütközés miatt.

Az EJB 2.1-ben kötelezően implementálandó metódusok helyett, csak azokat kell implementálni, amelyek esetén valamilyen életciklus esemény bekövetkeztéről kér tájékoztatást. Állapotmentes session bean esetén csak két ilyen esemény lehetséges, a létrehozás és a megsemmisítés:

- **@PostConstruct:** a létrehozás után hívódik meg közvetlenül, mely a korábbi ejbCreate megfelelője.
- **@PostDestroy:** a megsemmisítés előtt hívódik meg az általa megjelölt metódus, mely az ejbRemove metódust helyettesíti.

Az állapottal rendelkező esetben kiegészül további annotációkkal:

- **@PrePassivate:** passziválás előtt hívódik meg, mely az ejbPassivate megfelelője, és a passziválás előtti erőforrás felszabadítására használják, melyeket nincs értelme letárolni. Pl.: adatbázis-kapcsolat.
- **@PostActivate:** az ejbActivate megfelelője. Azon erőforrások igénylését, melyeket aktiválás után kell végrehajtania a beannek, az ezáltal jelölt metódusban kell végrehajtani. Tekinthető a **@PrePassivate**-tel jelölt metódus inverzének is.
- **@Remove:** a kliens ennek segítségével engedi el a hozzárendelt session beant, majd a konténer ennek hatására a **@PreDestroy** metódust is meghívja.

Az **@EJB** annotációval, úgynevezett függőség injektálás hajtható végre, amennyiben az EJB kliens szintén konténerben fut. Az **@EJB**-vel megjelölt változóhoz, a konténer kötelessége példányt biztosítani, ezáltal helyettesítve a JNDI keresést. Ez kényelmes eszközt biztosít, mely több változó esetén, felesleges programsoroktól kíméli meg a programozót, továbbá az átláthatóságot is javítja. A **@Resource**-szal a függőséginjektálás előnyeit terjesztették ki, további erőforrások megszerzésére. A konténeren kívüli kliensekhez továbbra is JNDI keresés szükséges.

5.2. Entity Bean

Az Entity Bean feladata az attribútumainak perzisztens tárolásáról való gondoskodás és az attribútumok elérésének biztosítása. Az entitás osztályok és a relációs adatbázis táblái közt egyértelmű leképezés, az objektum relációs leképezés, mely az Entity Beant az adatbázis egy táblájának, az attribútumait pedig a tábla oszlopainak felelteti meg. A Entity Bean egy példánya tehát az adatbázis egy sorának a memóriabeli reprezentációjának tekinthető.

Az EJB 2.1-es szabványban az Entity Beannek két altípusa van:

- **Bean Managed Perzisztence (BMP)**, az életciklus szakaszához megfelelő metódusokban implementálják a szükséges JDBC kapcsolatot és lekérdezéseket. Komoly hátránya, hogy egy tábla összes elemének lekérdezéséhez $N+1$ lekérdezést kell végrehajtani.
- **Container Managed Bean (CMP)** esetén viszont a konténer eszközeivel generálódik minden JDBC kód. Komoly hátránya a nehéz hordozhatóság.

Ezért EJB konténertől független megoldások születtek, a perzisztencia providerek megjelenésével (pl: Hibernate, TopLink). Ezek nem csak konténerfüggetlenek, hanem Java SE környezetben is használhatóak.

Jelentős előnyük a perzisztencia providereknek, hogy nem csak az Entity Beant felelteti meg, az adatbázis egy táblájának, hanem az adatbázis tábláinak generálása is végrehajtható velük. Amellett, hogy az adatbázis tábláinak DDL utasításait nem kell megírunk, további előnye, hogy az esetleges fejlesztés közbeni változtatások során csak egy forráskódot kell módosítani.

5.3. Java Persistence API

Az EJB3 válaszként megalkotta a Java Persistence API-t (JPA), mely a piacon megjelent perzisztencia providerek tapasztalatait egyesíti. Szintén használható Java SE környezetben és POJO alapú.

5.3.1. JPA Entitások

A `java.persistence` csomag annotációnak alkalmazásával adjuk meg az Entity osztály objektum-relációs leképezését.

Sok perzisztencia provider kezdetben, XML fájlok segítségével képezték le az osztályokat, de a későbbiekben ezek a rendszerek is kiegészültek annotációkkal. Az annotációk alapértelmezett értékek segítségével képezik le az osztály nevét és az attribútumokat, velük megegyező tábla- és oszlopnevekre. Ezeket az alapértelmezett értékeket, természetesen mi is felüldefiniálhatjuk.

A JPA egyik talán legnagyobb előnye, hogy az entitás bean forráskódjában együtt kezeljük az objektum-relációs leképezést és az osztályhoz tartozó lekérdezéseket is. A karbantarthatóság szempontjából ez sokkal előnyösebb, mint az XML fájlokkal történő megfeleltetés.

Osztályokra vonatkozó annotációk

- **@Entity**: entitás osztályt hozunk létre.
- **@IdClass**: összetett elsődleges kulcsot jelöl, amit az @Entity annotáció alatt kell elhelyezni, benne megadva a Class típust.
- **@Table**: az alapértelmezett táblanév definiálható felül, a name, catalog, schema paraméterekkel.
- **@Column**: az alapértelmezett oszlopnevet definiálhatjuk felül, paramétereivel pedig az oszlop tulajdonságait is befolyásolhatjuk (nullable, scale, precision, length)
- **@SecondaryTable**: egy újabb relációs táblában is tárolhatjuk az entitást.
- **@Embeddable**: olyan osztályt jelölünk meg, mely csak valamilyen entitáshoz kapcsolódva létezhet. Pl.: Cím.

Attribútumokra/beállító műveletekre vonatkozó annotációk

Ezeket az annotációkat az attribútumokra vagy getter/setter műveletekre is alkalmazhatjuk. Az alkalmazásbeli különbség ott jelentkezik, hogy ha az attribútumokra alkalmazzuk, akkor azok közvetlenül kapnak értéket az adatbázisból. Ha azonban a getter/setter műveletek valamelyikére, akkor azok segítségével állítódnak be az attribútumok értékei.

- **@Id**: mely az elsődleges kulcs megjelölésére szolgál.
- **@GeneratedValue**: támogatja az elsődleges kulcsgenerálást. Az **@Id** annotációval jelölt attribútum jelölhető meg vele, amennyiben valamilyen egész értéket reprezentáló objektum vagy változó. 4 féle kulcsgenerálási stratégiát támogat, amelyet a GenerationType enum típusú paraméterként kell megadni:
 - **SEQUENCE** adatbázis által kezelt számláló segítségével generál egyedi kulcsot.
 - **IDENTITY** egy adatbázis tábla autoinkrementáló oszlopára fogja leképezni.
 - **TABLE** a **@TableGenerator** használatával, valamely adatbázis tábla adott sorának adott oszlopa fog leképeződni.
 - **AUTO** esetén az adatbázis kezelőtől függő módon az előzőek valamelyike fog érvényesülni.
- **@Transient**: a tranziens attribútumok jelölhetőek meg, melyek nem kerülnek letárolásra az adatbázisban.
- **@Temporal**: a java.util.Date és a java.util.Calendar típusok leképezését változtatja meg. Ezek a típusok, automatikusan képeződnek le a TIMESTAMP SQL típusra, mely milliszekundum pontossággal tárolja az időt. Ha azonban elegendő számunkra az SQL DATE, illetve TIME típusaira történő leképezés, akkor **@Temporal** annotációval kell megjelölni és paraméterként a TemporalType.Date, illetve a TemporalType.Time értéket kell megadni paramétereként.
- **@Embedded**: olyan típusú osztály attribútumot jelöl, amely csak valamilyen entitáshoz kapcsolódva létezhet.
- **@AttributeOverride**: a beágyazott osztály egy attribútumához rendelhetünk vele **@Column** annotációt. Több ilyen annotációt egy **@AttributeOverrides** annotációval

foghatunk össze, ekkor az annotáció paramétere az **@AttributeOverride** annotációk egy tömbje lesz.

5.3.2. Lekérdezések

Az EntityManager interfészen keresztül kínál lehetőséget a lekérdezésekhez. A kulcs alapján történő kereséshez, egy generikus metódust definiáltak, mely a Java 5 eszközeit kihasználja, melyet osztállyal és elsődleges kulccsal paraméterezve keresi meg az entitást.

`<T> T find( Class<T> entityClass, Object primaryKey)`

Az összetettebb lekérdezések elvégzésére különféle műveletek állnak rendelkezésre, melyeket az EntityManager kínál. Minden ilyen metódus egy Query interfészt megvalósító osztállyal tér vissza. EJB-QL és SQL nyelven is írhatunk lekérdezéseket.

A JPA kibővíti az EJB-QL lekérdezéseket, amely a Java Persistence Query Language nevet viseli. Ezek a bővítések:

- **JOIN, GROUP BY, HAVING** támogatása
- Bemenő paramétereket nem csak a paraméter helyével lehet megadni, hanem neveket is adhatunk nekik. Például: :termekKod.
- Egy entitásnak lehetséges csak bizonyos attribútumait lekérdezni.
- Többszörös törlés és módosítás, ezáltal a lekérdezések és törlések összefoghatók egy műveletté, nem valamilyen iteráció sorozataként jelennek meg.
- Dinamikus lekérdezések létrehozása.
- Egy oszlopot valamilyen általunk definiált objektumként kérdezzünk le.

Az egyes lekérdezéseket az osztályhoz kapcsolhatjuk annotációk segítségével, melyeket névvel látunk el. Ezt a **@NamedQuery**, segítségével tehetjük meg:

- **name** paraméterében az lekérdezéshez rendelt nevet.
- **query** paraméterben a lekérdezést adhatjuk meg.

Több lekérdezést a **@NamedQueries** annotációban adhatunk meg, mely paramétere a **@NamedQuery** annotációk egy tömbje. Ezzel nagyban javul az osztályhoz tartozó lekérdezések karbantarthatósága az osztály esetleges változtatása esetén.

További előnye, hogy a láthatósággal szabályozza az osztályhoz tartozó lekérdezések használatát, mivel csak ott érhető el a névvel ellátott lekérdezés, ahol a típus neve is látható.

5.3.3. Entitások közötti kapcsolatok

A **kardinalitás** egy egyed típus előfordulásainak maximumát jelzi egy kapcsolatban, ennek megfelelően választhatunk az annotációk közül, melyek **@OneToOne**, **@OneToMany**, **@ManyToOne** és a **@ManyToMany** annotációk. Többes kapcsolat esetén **Collection**, **List**, **Set** típusú tagváltozók vagy velük visszatérő metódusok lehetnek.

A kapcsolatokat tagolhatjuk irány szerint:

- **Egyirányú kapcsolat:** ebben az esetben csak az egyik oldalról lehet elérni az egyik entitást, a másik oldalon nem lehet rá hivatkozni. Ekkor a tulajdonos oldal egyértelműen meghatározható.
- **Kétirányú kapcsolat:** elérhetik egymást mindkét oldalról az entitások. A tulajdonos oldal és az inverz oldal meghatározása a kapcsolatok típusától függ. Aminek három esete lehetséges:
 - **Egy-egy kapcsolatban:** a tulajdonos az idegen kulcsot tartalmazza.
 - **Egy-több kapcsolatban:** az az oldal a tulajdonos, amelyik a **@ManyToOne** annotációt tartalmazza.
 - **Több-több kapcsolatban:** bármely oldal lehet a tulajdonos, mivel egy kapcsoló tábla tartalmazza az idegen kulcsokat.

Kapcsolatok idegen kulcsokra történő leképezése:

- **Egy-egy kapcsolat:** bármely oldalra elhelyezhető az idegen kulcs.
- **Egy-több kapcsolat:** a több oldalt tároló tábla fogja tartalmazni.
- **Több-több kapcsolat:** egy harmadik, úgynevezett kapcsoló tábla tartalmazza az idegen kulcsokat, melyek mindkét oldalra mutatnak.

Idegen kulcsokra vonatkozó annotációk:

- **@JoinColumn**: amelyben az idegen kulcsot tartalmazó oszlop nevét adhatjuk meg paraméterként.
- **@JoinTable**: a kapcsoló táblát definiálhatjuk vele.

Használatuk nem kötelező, ekkor a perzisztencia provider fog az entitás vagy az oszlopnevek alapján előállítani egy nevet.

A kapcsolatok konzisztens állapotban tartása a JPA entitások esetén a fejlesztőre hárul, tehát az összes entitást módosítani kell egy kapcsolatmódosításban, különben nem definiálható a perzisztencia provider működése.

A kaszkádolt törlés lehetősége a JPA esetén is fenn áll, mint a CMP esetén, de ez kiegészült és nem csak törlésre adható meg, hanem az életciklus többi eseményeire is:

- törlés
- adatbázisból történő frissítés
- perzisztencia kontextusba történő bekerülés
- perzisztencia kontextusba történő visszakerülés

Ezek kaszkádolása, a kapcsolatdefiniáló annotációkban szereplő cascade paraméter tömbben kell megadni. Nincs alapértelmezett érték. A cascade paraméter lehetséges értékei:

- **CascadeType.REMOVE**: entitás törlése esetén törli a kapcsolódó entitást.
- **CascadeType.REFRESH**: entitás frissítése esetén frissíti a kapcsolódó entitást.
- **CascadeType.PERSIST**: entitás letárolása esetén letárolja a kapcsolódó entitást.
- **CascadeType.MERGE**: entitás újratárolása esetén újra letárolja a kapcsolódó entitást.
- **CascadeType.ALL**: az előző négy művelet mindegyikét engedélyezi.

A betöltés kérdését is tisztáznunk kell, ugyanis nem mindegy, hogy egy entitás betöltésekor a kapcsolódó entitások is betöltődjenek-e. Két eset lehetséges:

- **Agresszív vagy mohó betöltés**, mely minden esetben betölti a memóriába a kapcsolódó entitásokat. Ez gyorsabb megoldás, de esetleges felesleges memórafoglalással jár.
- **Lusta betöltés**, mely csak akkor tölti be a kapcsolat másik oldalán álló entitásokat, ha azokra tényleges hivatkozás történik.

Ezeket a betöltési módokat, a kapcsoló entitások annotációjának **fetch** paraméterében kell megadni:

- **FetchType.EAGER**: mohó vagy agresszív betöltési mód, ez az alapértelmezett érték.
- **FetchType.LAZY**: a lusta betöltési mód, melyet nem kötelező a perzisztencia providernek végrehajtania, ez csak egy javaslat.

A lecsatolt állapotba kerülő objektumok esetén a már betöltött kapcsoló objektumok lesznek elérhetőek. Ezek kétféle képen okozhatnak problémát:

- A kapcsoló objektumra történő hivatkozás kivételt eredményez, ami legkésőbb a tesztelés során derül ki. Még lecsatolás előtt explicit hivatkozhatunk a kapcsoló objektumokra, így betöltődnek a szükséges objektumok.
- A lecsatolt példányt újra menedzselt állapotba hozzuk az EntityManager interfészének **merge()** metódusával. Ilyenkor a kapcsolat hiányát szándékos törlésként értelmezi a perzisztencia provider, ekkor az adatbázisban is megszünteti az egyes entitások közötti kapcsolatot. Ennek elkerüléséhez gyakran kulcs alapján keresik meg az entitásokat és a lecsatolt entitásból átmásolják az attribútumokat vagy reflexióval is megoldhatjuk, amivel elkerüljük a getterek/setterek használatát.

A fetch paraméterek az egyszerű perzisztens attribútumokra is alkalmazható, így például csak akkor kerülnek betöltésre, ha tényleg szükséges. Például a LOB annotációval ellátott attribútumok.

5.3.4. Persistence Unit

- Konfigurációs információk
- Entitások
- O-R leképezés módját definiáló annotációk, XML-metaadatok
- EntityManagerFactory és az általa létrehozott EntityManagerek halmaza

Egy ilyen egység fizikailag lefordított osztályokból és egy persistence.xml állományból áll. A persistence.xml-ben több perzisztencia egység is definiálható, amelyre az alkalmazásból hivatkozva az adott perzisztencia egységhez EntityManagert szerezhethetünk. Ennek segítségével akár több perzisztencia provider is használható, kihasználva az általuk nyújtott egyedi szolgáltatások esetleges előnyeit, valamilyen adott helyzetre.

5.3.5. Perzisztenciakontextus

Perzisztenciakontextus: olyan entitáshalmaz, melyben minden elsődleges kulccsal rendelkező perzisztens példányhoz egyetlen egyedi memóriabeli példány tartozik.

Mivel gyakran több komponensen ível át egy tranzakció, ezért értelemszerű igény, hogy egy tranzakcióhoz ugyanaz a perzisztenciakontextus tartozzon. Ezt függőséginjektálás segítségével érhetjük el, a **@PersistenceContext** annotációval, vagy **JNDI** kereséssel. Ekkor egy referenciát szerez az EntityManagerre, amely a konténer által kezelt entitáskezelő.

A **@PersistenceContext** paraméterei:

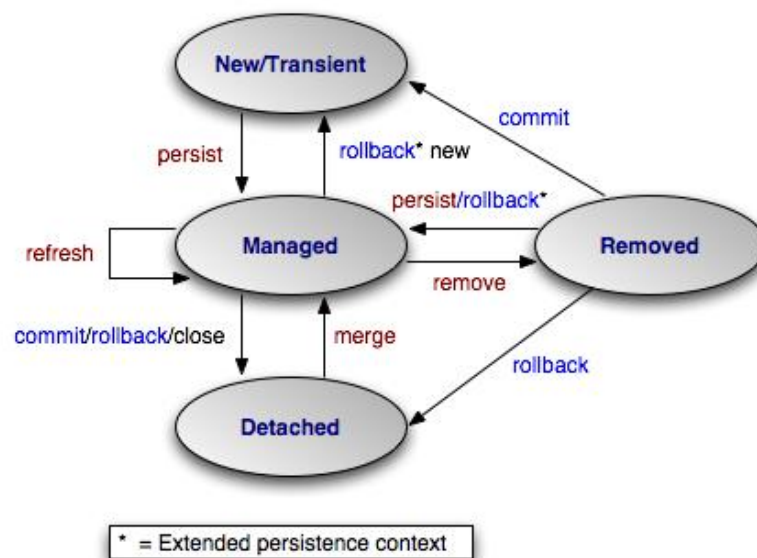
- **name:** névvel látja el a perzisztenciakontextust.
- **unitName:** több persistence unit esetén kell megadni, hogy a persistence.xml-ben megadottak közül melyikre vonatkozzon.
- **type:** a tranzakcióra **PersistenceContextType.TRANSACTION**, vagy több tranzakcióra terjeszti ki az élettartamot **PersistenceContextType.EXTENDED**. A **TRANSACTION** az alapértelmezett.

- **properties:** olyan beállításokat lehet megadni, melyeket az adott perzisztencia provider perzisztenciakontextusára vonatkoznak. Ha nem ismeri fel valamelyiket, akkor ignorálja.

A perzisztencia providerek biztosítanak minden perzisztenciával kapcsolatos szolgáltatást, melyeket a JPA által definiált interfészekon keresztül érhetünk el. Az EntityManager interface szolgál az entitások kezelésére, melyhez a konténer rendeli hozzá az aktuális JTA-tranzakcióhoz tartozó perzisztenciakontextust.

EntityManagerFactory interfész segítségével szerezhetünk egy alkalmazás által kezelt EntityManager példányt, mellyel olyan perzisztenciakontextust állíthatunk elő, mely más EntityManager példányok számára elérhetetlen.

5.3.6. Entitások élelciklusa



Egy entitás négy állapotban lehet:

- Új
- Menedzselt
- Törölt
- Lecsatolt

Új állapotba a példányosítással kerül, de ekkor még nem jön létre a neki megfelelő perzisztens adat.

Menedzselt állapotba a perzisztenciakontextusba való bevitellel kerül, az EntityManager interface **persist()** metódusának segítségével. Minden EntityManager egy listában tárolja a menedzselt entitásokat, melyeket a tranzakció végén szinkronizál az adatbázissal. Kényszeríthetjük az adatbázissal történő szinkronizációt:

- **flush()** metódussal kiírhatjuk az adatbázisba az entitásokat, még a tranzakció lezárulta előtt;
- **refresh()** metódussal a menedzselt entitás frissítjük.

Lecsatolt állapotba a perzisztenciakontextus megszűnésével kerülhet, de az entitáson végrehajtott változtatások nem kerülnek be az adatbázisba. Ismételten menedzselt állapotba hozható az EntityManager interfész **merge()** metódusával, mely előállít egy lecsatolt entitásból egy új menedzselt entitást. Tehát egy lecsatolt entitást nem tehetünk ismételten menedzseltté.

Törölt állapotba a **remove()** metódus meghívásával vihetjük. Ilyenkor kijelöljük törlésre és a tranzakció lezárultával törlődni fog.

Az EntityManagertől **callback** metódusok segítségével kérhetnek értesítést az entitások az adott életciklus eseményről. Ezek a metódusok külön osztályban is elhelyezhetők, melyet az **@EntityListener** annotációval jelölünk:

- **@PrePersist**: az adatbázisba történő írás után hívódik meg.
- **@PostPersist**: az adatbázisba történő írás előtt hívódik meg.
- **@PreRemove**: az adatbázisból való törlés előtt hívódik meg.
- **@PostRemove**: az adatbázisból való törlés előtt hívódik meg.
- **@PreUpdate**: az adatbázisba történő frissítés előtt hívódik meg.
- **@PostUpdate**: az adatbázisba történő frissítés után hívódik meg.
- **@PostLoad**: az adatbázisból történő betöltés után hívódik meg.

5.3.7. Öröklés

A 2.1-es EJB-k esetén az öröklődés nem volt a specifikáció része, ami csak alkalmazáserver-specifikus környezetben volt elérhető.

Az EJB3 részévé tette az öröklés támogatását, mely a POJO jelleg miatt az `extends` kulcsszó segítségével végezhető el. A relációs adatbázisba történő leképezést a perzisztencia provider látja el, amelyet annotációk segítségével szabályozhatunk.

Az **@Inheritance** annotációval és az **InheritanceType** típusú paraméterként adott értékkel adjuk meg a leképezési stratégiát, melyet az osztály definíciója előtt kell elhelyezni.

Három leképezési stratégiát támogatnak a perzisztencia providerek:

- Egy Tábla Egy Osztályhierarchia
- Külön Tábla Minden Gyermeosztályhoz
- Egy Tábla Egy Gyermeosztályhoz

Egy Tábla Egy Osztályhierarchia esetén minden gyermeosztály példány egy táblában tárolódik, amely tartalmaz minden perzisztens attribútumnak megfelelő oszlopot. A típusinformációt egy diszkriminátor oszlop segítségével teszi megkülönböztethetővé.

A leképezés szabályozáshoz szükséges annotációk:

- **@Id**, melyet csak az osztály tartalmaz, mivel ezt minden osztály örökli tőle.
- **@DiscriminatorValue**, ez határozza meg a típus azonosításához szükséges értéket, melyet minden osztály definíció előtt el kell helyezni. A típushoz rendelt táblában egy **DTYPE** nevű oszlop lesz a diszkriminátor, mely a paraméterként adott Stringet fogja tartalmazni, ezzel megkülönböztetve a típust.
- **@DiscriminatorColumn**, a diszkriminátor oszlop nevének és a típusának felüldefiniálásához használható.

Az **@Inheritance** annotáció paraméterében az **InheritanceType.SINGLE_TABLE** értéket adjuk meg ehhez a stratégiához.

Külön Tábla Minden Gyermeosztályhoz esetén az osztályhierarchia minden eleméhez külön tábla tartozik, amely csak az elsődleges kulcsot és a származtatott osztály perzisztens attribútumához tartozó oszlopot tárolja. Az elsődleges kulcs további szerepe, hogy idegen

kulcsként mutat az ősosztályhoz tartozó táblára. Az egyes lekérdezések során az eredményt join műveletek segítségével határozza meg, amely komoly hatékonyságromlást okozhat mély öröklési hierarchia esetén.

Ehhez a stratégiához az **@Inheritance** annotáció paraméterében a következő értéket kell megadni: **InheritanceType.JOINED**.

Egy Tábla Egy Gyermeosztályhoz esetén az osztályhierarchia minden eleméhez külön tábla tartozik, minden egyes táblában tárol minden ősosztálytól örökölt perzisztens attribútumot, és az adott származtatott osztály attribútumait is tárolja.

Az **@Inheritance** annotáció paraméterében az **InheritanceType.TABLE_PER_CLASS** értéket kell megadni, ehhez a leképezési stratégiához.

A származtatási hierarchiába nem csak entitás osztályok kerülhetnek be, ebben az esetben ezen osztályok attribútumai nem kerülnek letárolásra. A nem entitás ősosztály esetén az öröklési fában absztrakt osztályként is szerepelhet, ekkor öröklí a belőle származtatott osztály, az ősosztály metódusait és tagváltozóit, de ez nem fog a perzisztens állapotához tartozni.

A **@MappedSuperclass** annotációval hozzávehetjük a nem perzisztens ősosztály attribútumait is, de külön tábla egyetlen leképezési stratégia esetén sem jön létre hozzá.

5.3.8. Tranzakciókezelés

A tranzakciókezelés nem csak programozottan az EntityTransaction interfészen keresztül mehet végbe, hanem deklaratívan is.

Tranzakciós attribútumot adhatunk meg a **@TransactionAttribute** annotációval, mely paraméterekén a következő értékeket adhatjuk meg, melyek közül a **REQUIRED** az alapértelmezett:

- **REQUIRED**: ha még nem indított tranzakciót a konténer, akkor a tranzakciós metódus meghívásakor, indít egyet. Amennyiben már létezik egy tranzakció a hívás pillanatában, akkor abba fog becsatlakozni.
- **REQUIRES_NEW**: ha még nem indított tranzakciót, akkor a hívás pillanatában a konténer indít egyet. Ha a hívás pillanatában él egy tranzakció, akkor felfüggeszti azt

és indít egy újat. Az új tranzakció kimenetelétől függetlenül folytatja az előzőt és egyik kimenetele sem függ a másiktól.

- **SUPPORTS:** Csak akkor csatlakozik tranzakcióhoz, ha már a hívás pillanatában is van egy élő tranzakció. Ha nincs ilyen tranzakció, akkor nem indít újat, tehát tranzakció nélkül fut le a metódus.
- **MANDATORY:** ha még nem indult tranzakció, akkor ezt kivétellel jelzi a konténer. Létező tranzakció esetén viszont csatlakozik, a már megkezdetthez.
- **NOT_SUPPORTED:** a konténer nem indít új tranzakciót és e nélkül fut le a metódus. Ha már létezik élő tranzakció a hívás pillanatában, akkor az felfüggesztődik és tranzakció nélkül fut le a metódus, majd a felfüggesztett tranzakció folytatódik.
- **NEVER:** ha a hívás pillanatában még nincs élő tranzakció, akkor a konténer sem indít, így tranzakció nélkül fut le a metódus. Ha van élő tranzakció, akkor a konténer kivételt dob.

JPA esetén probléma merülhet fel tranzakció visszagörgetésekor, ugyanis az EJB 2.1 a visszagörgetést követően az entitásokat is visszaállítja a tranzakció előtti állapotba. Ezzel az entitások újrafelhasználhatóvá válnak. A JPA entitások esetén visszagörgetéskor a perzisztenciakontextusba tartozó entitások lecsatolt állapotba kerülnek, és azt az állapotukat tartják meg, ami a visszagörgetés előtt volt, tehát nem történik szinkronizáció az adatbázis jelenlegi állapotával. Ez olyan hiba, mely inkonzisztenciához vezethet és az entitások újrafelhasználását jelentősen rontja.

A konkurencia kezelés hangolására, két lehetőséget biztosít:

- Optimista konkurenciakezelés
- Explicít zárok

Az optimista konkurenciakezelés nem alkalmaz zárat. Azon a feltételezésen alapul, hogy a kliensek nem ugyanazt az adatot módosítják. Ha mégis egyszerre próbálják ugyanazt az adatot módosítani a konkurens kliensek, akkor az időben később módosító kliens kap értesítést, a változásról. Ekkor a kliens újraolvassa az adatot és eldönti, hogy van e értelme az általa végrehajtandó műveletnek. Ennél a konkurenciakezelésnél, az entitás egyik adattagját vagy egész számnak vagy **java.sql.Timestamp** típusúnak választjuk és a **@Version** annotációval látjuk el. Ezt az adattagot a konténer fogja kezelni, mint a **@GeneratedValue** annotációval

ellátott adattagot. **OptimisticLockException**-t dob a perzisztencia provider, ha nem egyezik a verziószám a memóriabeli példány legutolsó kiolvasásával.

Explicit zárkezelés esetén, ha az alkalmazás egyes pontjain a tranzakciónk erősebb izolációját szeretnénk elérni, akkor a perzisztencia provider kiküszöböli a piszkos és a megismételhetetlen olvasást is, az adatbázis izolációs szintjétől függetlenül. Ezt az EntityManager interfész **lock(Object entity, LockMode mode)** metódusán keresztül érjük el.

A mode paraméter két értéket vehet fel:

- **LockMode.READ**, esetén a verziószámot a perzisztencia provider nem növeli, de a specifikáció nem tiltja, ha a megfelelő izolációt garantálni tudja.
- **LockMode.WRITE**, esetén mindenképpen növeli a verziószámot.

6. Webes keretrendszerek

A weboldalak dinamikus generálásra képes technológiák elterjedésével a webes alkalmazások a hagyományos desktop alkalmazások egy részét képessé vált kiváltani. Az első szerveroldali webes technológia a Servlet volt, majd megjelent a JavaServer Pages.

Ezt a két technológiát felváltotta a JavaServer Faces, mely igen népszerű, hiszen épít a JavaServer Faces technológiára és a fejlesztőkörnyezetek támogatják a gyors és hatékony fejlesztést elősegítő eszközökkel.

6.1. Webes keretrendszerekkel szembeni igények

A webes keretrendszernek biztonságosan és hatékonyan kell összekapcsolnia a felülettervező által készített felületet az üzleti logikával.

- **Konverzió:** a felület mögötti üzleti logika nem csak String típusú értékeket vár, tehát az adatokat konvertálni kell a megfelelő típusú értékekre. A konverzió során hibák léphetnek fel, melyeket egyrészt ellenőrizni, másrészt jelezni kell. Egy keretrendszernek ezeket az igényeket támogatnia kell, mert fokozottan szükség van rájuk.
- **Validáció:** a szintaktikai ellenőrzést követően szükséges a szemantikai ellenőrzés végrehajtása is. Az adatokat öt részre oszthatjuk:
 - Nem kell ellenőrizni, mivel ellenőrizhetetlen. Pl.: név.
 - Üzleti folyamatból következő ellenőrzést igényel. Pl.: Nem lehet negatív mennyiségű ruhát rendelni.
 - Hamis adatok megadásának elkerülése érdekében, igényel ellenőrzést.
 - Belső összefüggés alapján igényel ellenőrzést.
 - Másik mező értékével összehasonlítva.

Tehát a mező alapú és a több mező együttesének validációját kell támogatnia egy keretrendszernek.

- **Oldalak közötti újrafelhasználhatóság:** az oldalak közti sablon újrafelhasználás támogatása, hogy az állandó elemeket ne kelljen minden oldalon implementálni, hanem elég legyen rájuk hivatkozni. Erre az esetleges hibák elkerülése miatt van szükség.
- **Komponensek támogatása:** olyan építőkövekből rakjuk össze az oldalt, melyek maguk is újrafelhasználhatóak. Így nem csak az oldalt, hanem az építő elemeit is újra felhasználhatjuk. Ezzel egyetlen hivatkozássá foghatjuk össze az egyes komponenseket, melyek így a fejlesztést megkönnyítik és meggyorsítják.
- **Okos komponensek:** nem csak a nézet, hanem a viselkedés terén is jelentkezik az újrafelhasználhatóság. A komponens az adatok konvertálását, validálását, és akár az adatbázisba való mentését, lekérdezését és megjelenítését is tudják kezelni.
- **Navigáció:** a felhasználó az egyik oldalról a másik oldalra történő navigálásához támogatást kell nyújtania a keretrendszernek. Ez kétféle módon történhet:
 - *Deklaratív módon:* a mindig azonos link szerepel az oldalon.
 - *Programozottan:* csak bejelentkezés után megjeleníthető oldal.
- **Nemzetközisítés:** kulcs-érték párként kigyűjtjük a felületen található szöveget és a kódban csak a kulcsra hivatkozunk. De ezen kívül egyéb esetekkel is foglalkozni kell:
 - Eltérő sorrendben kell megjeleníteni a mezőket.
 - Egyes mezők megjelenítése értelmetlen.
 - Az input adatok bekérése és az output adatok megjelenítése különböző lehet.
 - Az országra jellemző konverterek, validátorok eltérhetnek.
- **Állapotok elmentése:** mivel a http protokoll állapotmentes, de egyértelmű igénye a felhasználónak, hogy egy adott oldalra visszatérve ugyanazt az állapotot találja. Ehhez menteni kell az oldalak állapotát, a komponenseknek az állapotmentést is meg kell oldaniuk.

- **A nézet és viselkedés szétválasztása:** Az egyes szerepeket szükséges szétválasztani, hogy a fejlesztést párhuzamosan tudjuk végezni, vagy legalábbis megpróbáljuk párhuzamosítani. A fejlesztők ezért jól elkülöníthető fejlesztői szerepeket kaphatnak.
 - *Grafikus:* színséma, felület, grafikus elemek tervezése a feladata.
 - *Weboldalfejlesztő:* az üzleti logikát a komponensek és a webes keretrendszer segítségével a felülethez kapcsolja.
 - *Komponensfejlesztő:* webes komponensek fejlesztését végzi.
 - *Üzletlogika-fejlesztő:* az üzleti logika fejlesztését végző személy.
- **Biztonság:** Általában az input és output adatok szűrését jelenti. Ehhez kiemelkedően magas támogatást kell nyújtania egy keretrendszernek.
- **Elterjedt technológiákhoz való illeszkedés:** olyan technológiákkal tudjon együttműködni, melyek kiegészítik a felület készítés eszközszerét. Ilyen technológiák, a CSS, az Ajax vagy a JavaScript.
- **Eszköztámogatottság:** Mint minden programozási eszköznél, itt is fontos, hogy a fejlesztőeszközök olyan támogatást nyújtsanak, mely segíti a fejlesztések felgyorsítását. Drag-and-Drop eszközök a felületelemek elhelyezéséhez, további eszközök a konfigurációs XML fájlok karbantartásához.
- **Bővíthetőség:** Ezalatt az újrafelhasználható komponens könyvtárak megalkotását értjük, melyeket több projekten keresztül is felhasználhatunk.

Ezek a fontosabb szolgáltatások, melyeket egy mai webes keretrendszernek mindenképpen támogatnia kell.

7. JavaServer Faces

A Java EE 5 platform specifikált része, mely egy komponens orientált webes keretrendszer. Számos implementációja létezik vagy egyéb kiegészítő szolgáltatásokat nyújtó keretrendszerek. A jelenlegi legújabb verzió a 2.0, többek közt az AJAX támogatása miatt jelent jelentős előrelépést, de egyéb szolgáltatásaival is kiemeli a többi keretrendszer közül, mert a Webes keretrendszerekkel szembeni igények legtöbbjét kielégíti.

7.1. Működése

Minden JSF eszközt használó jsp oldalra irányuló kérést a FacesServlet kap meg, mely létrehoz egy állapotgépet. A View szerepét tehát a jsp oldalak, míg a Modell és a Vezérlő szerepét a menedzselt JavaBean látja el. Olyan interfészeket kell nyújtania a Beaneknek, melyek segítségével a keretrendszer meg tudja változtatni a modell állapotát, továbbá az oldalak közti navigációhoz is támogatást kell nyújtaniuk. A JavaBeant menedzselté kell tenni, és ezzel összekötni a megfelelő oldallal, melynek háttereként szolgál, ezt az összerendelést a facesconfig.xml konfigurációs fájlban tehetjük meg. Az életciklusát is itt szabályozhatjuk, és ha már nincs rá szükség, akkor használaton kívül helyezi a keretrendszer, mely ekkor szemétgyűjtésre kerül.

7.2. Nézet

Mivel a JSF erőteljesen a jsp technológiára épít, ezért minden jsp oldal szervletté fordítódik. A fordításkor felépít egy memóriabeli fát, mely a View. A gyökerében található a ViewRoot, innen a teljes nézet bejárható, és a fa minden csomópontjához tartozik egy objektum, mely az adott JSF elem típusát veszi fel.

7.3. Menedzselt beanek

A menedzselt beanek látják el a modell és a vezérlő szerepét is.

Felhasználási módjaik:

- Adatot szolgáltathat egy komponensnek.
- Egy menedzselt osztály adattagjaiba, a getter/setter metódusokkal menthet le értékeket, melyeket az alkalmazás használ fel.
- Valamilyen felhasználói beavatkozás esetén egy metódushívással értesítheti a keretrendszer a menedzselt osztály aktuális példányát.
- Komponensek tulajdonságait változtathatjuk meg.

Általában minden jsp oldal mögött egy menedzselt osztály áll.

Az osztályok paraméter nélküli konstruktort kell, hogy tartalmazzanak, mivel ezeket az osztályokat a keretrendszer fogja példányosítani. A menedzselt osztályok életciklus eseményeiről értesítést is kérhetünk, melyekhez annotációkat kell alkalmazni:

- **@PostConstruct**: létrehozás után, közvetlenül meghívja a keretrendszer.
- **@PreDestroy**: az objektum megszüntetése előtt hívódik meg.

A facesconfig.xml állományban kell elhelyezni a következő tageket, melyek egy egyszerű Java osztályt menedzselt osztállyá tesznek.

- **<managed-bean>** ezzel megadhatunk egy újabb menedzselt osztályt.
- **<managed-bean-name>** a menedzselt osztálynak alias nevet adhatunk, mellyel elérhetjük a JSF oldalon.
- **<managed-bean-class>** az osztály teljesen specifikált nevét kell megadni.
- **<managed-bean-scope>** ezzel szabályozhatjuk a példány életciklusát.

A managed-bean-scope lehetséges értékei:

- **request**: kérés feldolgozás alatt példányosítja az osztályt a keretrendszer.
- **session**: addig él, amíg a felhasználó sessionje él.
- **application**: minden oldalon elérhető, mely ugyanabban az alkalmazásban van.

A menedzselt példány metódusaihoz a **value** attribútummal lehet értéket rendelni. Az **UIOutput** leszármazottai azok a komponensek, amik megjelenítenek és meg is

változtathatnak egy értéket. Az **UICommand** leszármazottai esetén, az action attribútum vagy metódushívás, vagy pedig egy String kell, hogy szerepeljen.

A facesconfigban a beanhez rendelt azonosítóval `#{bean_neve.tulajdonság}` alakban érjük el az egyes getter/setter metódusait.

Kényelmes megoldást nyújt az **Map**, **Array** és a **List** típusú tulajdonságok elérésére, **Array** esetén tömb indexel, kollekciók esetén a set/get metódusokkal használhatjuk.

7.4. Konverterek, validátorok

A konverterek a várt input adatokat konvertálják a megfelelő Java típusra, míg a validátorok a konvertált érték szemantikai helyességét ellenőrzik.

Ezeket az eszközöket úgy használjuk, hogy konvertálandó input mező tagjei közt helyezzük el a validátort, illetve a konvertert. Természetesen minden alaptípushoz létezik konvertálási lehetőség, akárcsak a dátum konvertálására.

A validáció során fellépő hibák észlelése, a validációk futtatását végző fázisban történik. Az **UIInput setValid** metódusát hívja meg **false** értékkel, és elhelyez egy üzenetet a komponens üzenetlistáján. A modell aktualizálása előtt megvizsgálja, hogy történt-e valamilyen hiba, ha igen, akkor a nézet generálását végzi el. Ekkor nem történik navigáció, ugyanazt az űrlapot látja a felhasználó.

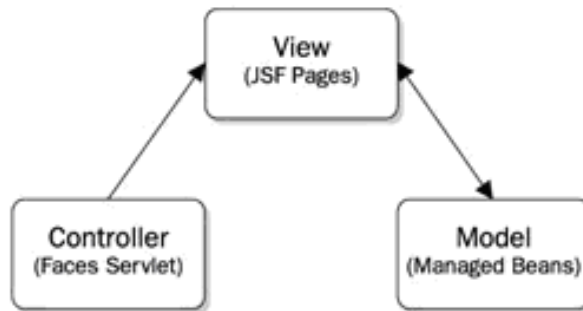
Mivel a validálás és a konvertálás hibás adatok esetén valamilyen hibaüzenettel jelzi a felhasználó számára a hibát, ezt az üzenetet meg kell jeleníteni, melyet a **h:messages** komponensével tehetünk meg. A **for** attribútumban kell megadni annak a komponensnek az azonosítóját, melytől az üzenetet várjuk.

Felléphet olyan helyzet, mely során a validátorok és konverterek lefutása értelmetlen. Például, ha nem töltjük ki az űrlapot és szeretnénk visszanavigálni az előző oldalra. Ebben az esetben alkalmazható az **immediate** attribútum, mely a kérés paraméterek rögzítése pillanatában hajtja végre a konverziót vagy a validációt.

A konverterek és validátorok használata nem csak valamilyen alaptípusú érték esetén lehetséges. Egyes esetekben nekünk kell konvertert vagy validátort írni, mely komponensek előállításához nyújt támogatást a JSF. Ezeket az eszközöket a faces-config.xml konfigurációs

állományban kell regisztrálni. Az egyes validátor komponenseket, pedig a megfelelő Java osztályból kell származtatással előállítani.

7.5. A JSF navigációs modellje



Irányított gráf vezérli az oldalak közti navigációt, melyben a csomópontok az egyes oldalak, a nyilak a navigációs szabályok. A navigációs út a felhasználói interakcióktól függ, ekkor **UICommand** komponenseket aktivál.

A **h:commandLink** és a **h:commandButton** komponensek action attribútumában kell megadni a navigációt. A navigáció két csoportba sorolható:

- **statikus:** a parancs az action paraméterben Stringként szerepel. Ahol nincs döntési helyzet, ott alkalmazható.
- **dinamikus:** a vezérlést végző menedzselt bean egyik módszerének visszatérési értéke adja meg a navigációs parancsot. Döntési helyzetben a megfelelő navigációs parancsot adja ki, amennyiben null a navigációs parancs, akkor egyszerűen nem történik navigáció, ekkor postbacket hajt végre.

A keretrendszer a facesconfig.xml fájlban megadott navigációs szabályok alapján hajtja végre a megfelelő vezérlési műveleteket.

- **<navigation-rule>** tagekel adjuk meg a navigációs szabályokat.
- **<form-view-id>** meghatározza, hogy mely oldalak esetén kell alkalmazni, használható a * is mely minden oldalra kiterjeszti a végrehajtási kötelezettséget.
- **<form-action>** -ben adjuk meg a JSF Expression Language kifejezést, amelyet az action attribútumban alkalmazhatunk.

- **<from-outcome>** feltétel, ami csak akkor működik, ha az elemben megadottal egyenlő.

Amennyiben postback történik, akkor a felépített fát, újra felhasználja, egyébként pedig felépíti az új oldalhoz tartozó komponens fát és azt állítja be ViewRootnak.

7.6. Többnyelvű JSF oldalak

Alapvetően négy feladatot kell ellátni a nemzetköziesítés terén.

- Felhasználó nyelvének automatikus eldöntése.
- Felhasználó által választható nyelv, ha nem lenne megfelelő az automatikusan kiválasztott.
- Egyes elemek különböző nyelveken történő megjelenítése.
- Keretrendszerből érkező üzenetek lokalizációja

Mindezen feladatokra kényelmes megoldást nyújt a JSF. A Böngésző nyelvéből eldönti HTTP kérés segítségével a nyelvet, így a fejléc figyelésével változtatja a tartalmat.

A gyakorlatnak megfelelően itt is kulcs-érték párokként kell megadni egy properties fájlban. Ekkor megadjuk azokat a neveket, melyekre a felületen hivatkozni fogunk és a keretrendszer ez alapján helyettesíti a megfelelő értékkel.

A properties fájl neve alapján dönti el, hogy melyik nyelvi fájlt kell használnia a rendszernek, ami egy névből és egy országcódból kell, hogy álljon. A **facesconfig.xml** konfigurációs fájlban ezért meg kell adni a fájl nevét és felsorolni azokat az országcódokat, mely nyelveken rendelkezésre állnak az üzenetek. Az alapértelmezett nyelvet a **<default-locale>** taggel adjuk meg, a további nyelveket a **<supported-locale>** taggekkel sorolhatjuk fel. A **<message-bundle>** segítségével adhatjuk meg az üzenetfájl teljes elérési útját, ezekből többet is fel lehet sorolni.

A felületen az **f:loadBundle** taggel tesszük elérhetővé a fordításokat, mely **var** címkéjében megadjuk azt a nevet, amellyel hivatkozhatjuk a properties fájlban található címkéket

A konverterek/validátorok üzeneteit az üzenet fájlban felüldefiniálhatjuk, ezzel megoldható ezek adott nyelven történő megjelenítése.

8. Biztonság

Elengedhetetlen a biztonság néhány alapfogalmának ismerete.

Autentikáció: az a folyamat, mely során a felhasználótól elkért adatok alapján, identitást rendelünk hozzá a rendszeren belül, mellyel felhasználói tevékenységet végezhet.

Authorizáció: egy már autentikált felhasználóról eldönti, hogy jogosult-e egy művelet elvégzésére.

Aktív támadó: a hálózati forgalmon átmenő vagy perzisztens adatokat megpróbálja módosítani.

Passzív támadó: az adatok olvasását akarja elérni, mindenféle módosítás nélkül.

Adatintegritás: ha az adatokat nem rendeltetésszerűen módosítják és ezt a rendszeren belül detektálni tudjuk.

Adatok bizalmassága: nem csak az adatok módosítását, de az adatok olvasását is csak az arra jogosultak hajthatják végre.

Letagadhatatlanság: utólagosan bizonyítható, hogy egy adott felhasználó milyen műveletet végzett a rendszeren.

Biztonsági kockázat: egy várhatóérték, mely a sikeres támadás valószínűségének és az általa okozott veszteség szorzata.

Mivel nem zárható ki egyáltalán a sikeres támadás valószínűsége, ezért annyira meg kell nehezíteni, hogy annak költsége nagyságrendekkel haladja meg a vele szerezhető hasznot.

9. JavaEE szerepalapú biztonsági modell

Az alkalmazásszerverek feladata támogatni a szerepalapú autorizációt, amit a **JAAS**-re épülve oldanak meg. Ezzel az EJB modulokhoz szerepeket definiálhatunk, melyeket annotációkkal vagy telepítésleírókkal adhatunk meg. Ekkor Stringként adjuk meg a szerepkört, melyet a komponensek fejlesztői szabadon adhatnak meg, és melyekre szabadon hivatkozhatnak. A fizikai felhasználói csoportok és a szerepkörök közötti leképezést telepítési időben kell elvégezni, így felhasználói nyilvántartás nélkül is kifejleszthetők az egyes alkalmazáskomponensek.

Az alkalmazásszerverek újabb eszköz bevezetésével nyújtanak támogatást a szerepalapú biztonsági modellhez, melyet realmnek vagy user registrynek hívnak. Olyan adattárolók, melyek nyilvántartják a felhasználók autentikációs adatait és csoport tagságait. Saját magunk is fejleszthetünk realmet, melyet valamilyen adott osztályból származtatással és a megfelelő metódusok felüldefiniálásával hozhatunk létre. Ezt az osztály az alkalmazás szerver osztálykönyvtárában kell elhelyezni, gyakorlatilag a szerverhez kell kiegészítő csomagot készíteni, majd a konfigurációs fájlokban a megfelelő információkat elhelyezni.

10. Az alkalmazás bemutatása

10.1. Az alkalmazás által használt eszközök

Glassfish

Glassfish V3 alkalmazásszervert használtam, mely a kezdő Java EE fejlesztők által előszeretettel használt szerver. Használhatósága tapasztalataim szerint némileg megkérdőjelezhető, mivel komoly hibákat tapasztaltam a fejlesztés során.

MySQL

Az adatbázis réteget egy MySQL szerver szolgáltatja, mely ideális egy ilyen alkalmazás számára. Az Oracle termékcsalád részét képezi, melyet a TopLink-el való kompatibilitás miatt is választottam.

TopLink

A program az objektum-relációs leképezéshez, a TopLink perzisztencia providert használja, melyet az Oracle gyárt és az Oracle Fusion Middleware eszközeinek részét képezi. Mivel a Java jelenleg az Oracle termékcsalád része, ezért megfelelő választásnak tűnt, a TopLink használata. Kiváltképpen az egyéb szolgáltatásai miatt, gyorsítótárat hoz létre az objektumok azonosításához és több lekérdező nyelvet is támogat.

Nem csak az adatbázis táblák és az Entitás Beanek közti kapcsolat megteremtését, hanem az Entitás Beanekből annotációk alapján történő adatbázis táblák generálását is végrehajtottam vele.

JavaServer Faces

A webréteg megvalósításánál a JSF technológiát alkalmaztam, mely minden szempontból megfelelt az alkalmazáshoz. A legelterjedtebb webes keretrendszer, mivel minden ilyen keretrendszerrel szembeni igényt kielégít és Drag-and-Drop fejlesztőeszközökkel is támogatott.

XHTML-t és CSS-t kiegészítő technológiaként, a felület dizájn megalkotásához. Ezek olyan elterjedt technológiák, melyekhez illeszkedik a JSF.

Apache MyFaces

A MyFaces egy JSF implementáció, mely megfelel a JSF specifikációjának, és egyéb olyan kiterjesztéseket tartalmaz, melyek szolgáltatásai megkönnyítik a fejlesztést. A MyFaces olyan szolgáltatásokat nyújt, mint például a fájl feltöltés, melyet én is kihasználtam az alkalmazásban.

A fejlesztők körében elfogadott implementáció, a JSF szolgáltatásainak szükséges kiegészítései miatt.

Apache Http Web Server

Az egyik leggyakrabban használt Web Server, főként a PHP programozók körében, mivel nagyon stabil és megbízható. Az alkalmazásban a képek kiszolgálására használtam a Web Servert.

10.2. Design kialakítása

Mivel a design kialakítását a web designerek végzik, ezért ezt csak részben éreztem feladatomnak.

Egy két színből álló felületet hoztam létre, mely a táblázatok esetén egy harmadik színt is használ a sorok elkülönítésére. A CSS technológiát használtam, mely illeszkedik a JSF keretrendszerhez és a tagek **styleClass** attribútumával rendelhető az adott elemhez a CSS osztály.

Minden felület háttereként NavajoWhite színt használtam, mely szín leginkább a pasztellszínek közé sorolható, ezzel a folyamatos munka során nem fárasztja a dolgozó szemét. A betűk színe szürke, mely jól láthatóan elkülönül a háttérszíntől és a felületen megjelenő szöveg elemeket jól olvashatóvá teszi. A táblázatok esetén egy PeachPuff színt alkalmaztam, mely élesen elkülöníti a páratlan sorokat a páros soroktól.

10.3. Képek kezelése

Az alkalmazásban a ruhákhoz tartozó képek kezelését további eszközökkel kellett megoldanom, mivel a JSF keretrendszer nem nyújt megfelelő támogatást erre a problémára.

A **h:graphicImage** tag segítségével lehet képeket megjeleníteni a felületen, melynek a kép url címét is megadhatjuk. Emiatt a képek kiszolgálására alkalmaztam egy negyedik réteget, egy web servert. Az adatbázisban így csak a képek nevei kerülnek letárolásra, mellyel a felületen kiegészíti az alkalmazás az url címet.

A web server beiktatása a rétegelt architektúrába javítja a rendszer skálázhatóságát és nem igényel olyan JSF kiterjesztéseket, mint amiket a képek tállózásához kellett használnom.

A képek neveinek adatbázisban való tárolásához egy MyFaces kiterjesztést alkalmaztam, mely file browser szolgáltatást is nyújt, ennek segítségével tállózhatjuk a szükséges képet, melynek a kiválasztás után a nevét kérdezzük le és azt tároljuk az adatbázis táblában.

Az url cím tárolását egy Java osztállyal oldottam meg, ennek segítségével az url cím változása esetén az alkalmazásban csak az osztály adattagját kell megváltoztatni, ezzel elkerülve az esetleges változtatásokkal járó hibalehetőségeket.

10.4. Az alkalmazás funkciói

10.4.1. Alkalmazottak karbantartása

A webáruház dolgozóinak nyilvántartását egy **Alkalmazottak** adatbázistáblában tároltam le. Ezt a táblát a perzisztencia provider generálja le a többi táblával együtt az alkalmazás telepítése során.

Az **Alkalmazottak** entitás osztály az alkalmazott nevét, címét, telefonszámát, személyi adatokat, mint például az adó azonosítót tárolja, ezek mindegyikét külön osztályok valósítják meg, melyeket beágyazott osztályként kapcsoltam az entitás osztályhoz az **@Embedded** annotációval. Ezzel használtam ki a beágyazott osztályok által nyújtott lehetőségeket, melyek megkönnyítik az esetleges módosításokat.

```
public class Alkalmazott implements Serializable {  
  
    @Id  
    @GeneratedValue  
    @Column(name = "ID")  
    private Long id;  
  
    @Embedded  
    @AttributeOverrides({  
        @AttributeOverride(name = "vezetekNev",  
                           column = @Column(name = "VEZETEK_NEV", length = 50)),  
        @AttributeOverride(name = "keresztNev",  
                           column = @Column(name = "KERESZT_NEV", length = 50))  
    })  
    private Nev nev;  
  
    ...  
}
```

A beágyazott osztályok egyes attribútumaihoz adatbázisbeli oszlopneveket és az egyes attribútumokra megszorításokat tettem, melyeket az **@AttributeOverride** annotációkkal hajtottam végre, ezeket az annotációkat az **@AttributeOverrides** annotáció paraméterében tömbként adtam meg.

Az alkalmazás egy külön táblában tárolja az alkalmazottakhoz tartozó azonosítót, jelszót és azt, hogy mely felhasználói szerepkör tartozik az adott alkalmazotthoz. Az entitás osztályban ezt egy **Azonosito** entitás osztály adattag és egy entitások közötti kapcsolatot definiáló **@OneToOne** annotáció írja le. A **@OneToOne** annotációnál kihasználtam a **fetch** paraméterének alapértelmezett értékét, mely minden **Alkalmazottak** táblát érintő lekérdezés során betölti a memóriába az adott alkalmazotthoz tartozó azonosítót is.

Az alkalmazottak három szerepkört tölthetnek be a rendszerben:

- **Adminisztrátor**, akinek minden rendszer tevékenységhez joga van.
- **Árufeltöltő**, aki a ruhák és a járulékos adatok karbantartását végzi.
- **Rendelésfelvevő**, aki a rendelések teljesítését végzi.

A három szerepkör a **Csoportok** osztályban statikus adattagként adtam meg egy **List** objektumban, melyek közül a felületen választhatjuk ki az alkalmazottnak szánt szerepkört. Az alkalmazottak szerepköre határozza meg a rendszeren belüli jogosultságát egy alkalmazottnak.

Az osztályhoz tartozó lekérdezéseket a **@NamedQueries** annotációban **@NamedQuery** annotációkból álló tömbben adtam meg, mellyel kihasználtam, hogy együtt tárolható az entitás és a hozzátartozó lekérdezések. Így a fejlesztés menetén történő változtatásokat közvetlenül a lekérdezéseken is végre hajthattam, emiatt nem kellett több felesleges forráskódot karbantartani.

A lekérdezések az összes alkalmazott lekérdezését, egy adott felhasználó módosítását és az alkalmazott törlését teszik lehetővé, azaz egy **SELECT**, egy **UPDATE** és egy **DELETE** utasítás, melyet **Java Persistence Query Language**el írtam le.

Az **EntityManager** interfész segítségével kezeltem az alkalmazottak perzisztenciával kapcsolatos műveleteit. Az SQL utasításokhoz megfelelő műveleteket definiáltam az **AlkalmazottManager** osztály számára, mely eljárások az **modosit(Alkalmazott alkalmazott)**, **torol(Alkalmazott alkalmazott)**, **List<Alkalmazott> osszesAlkalmazott()**. Továbbá az **ujAlkalmazott(Alkalmazott alkalmazott)** metódus az **EntityManager persist** metódusával helyezi a perzisztenciakontextusba az új alkalmazott adatait.

```
@NamedQueries({  
    @NamedQuery(name="Alkalmazott.all", query="SELECT a FROM Alkalmazott a"),  
    @NamedQuery(name="Alkalmazott.update", query="UPDATE Alkalmazott a SET"  
        +" a.nev.vezetekNev=:vezetekNev, a.nev.keresztNev =:keresztNev,"  
        +" a.cim.iranyitoSzam=:iranyitoSzam, a.cim.telepules =:telepules, a.cim.utca =:utca,"  
        +" a.cim.haz =:haz,"  
        +" a.telefon.otthoni=:otthoni, a.telefon.mobil =:mobil, a.telefon.fax =:fax,"  
        +" a.szemelyAdatok.taj=:taj, a.szemelyAdatok.szemelyiSzam =:szemelyiSzam , "  
        +"a.szemelyAdatok.adoAzonosito =:adoAzonosito"  
        +" WHERE a.id = :id"),  
    @NamedQuery(name="Alkalmazott.delete", query="DELETE FROM Alkalmazott a"  
        +" WHERE a.id = :id")  
})
```

Az alkalmazottak adatainak rendszerbe történő felvitelét és módosítását űrlapok segítségével oldottam meg. Ezeken az űrlapokon az egyes mezők elemeire validációt hajt végre a JSF feldolgozó rendszer és hiba üzenetekkel jelzi a hibás adatok bevitelét.

A rendszerbe felvitt alkalmazottakat egy táblázatban tekinthetjük meg. Ebből a táblázatból kiválaszthatjuk az egyes alkalmazottakat adataiknak módosítására vagy akár rendszerből való törlésre.

Alkalmazott módosítása

Vezeték név:	Kis
Kereszt név:	Pista
Irányító szám:	4032
Település:	Debrecen
Utca:	Titok utca
Ház:	8
Felhasználó név:	kispista
Jelszó:	kispista
Csoport:	Rendelésfelvevő
Otthoni:	21344567891
Mobil:	21344567892
Fax:	21344567893
Taj:	034123435
Személyi szám:	9783423323AD
Adó azonosító:	976434234234

Az alkalmazott adatainak módosítását vagy rendszerből való törlését egy külön oldalon tehetjük meg, melyre a kiválasztás után navigál át a rendszer. Itt szintén egy űrlap segítségével módosíthatjuk az adatokat, melyet a rendszer kitölt a felhasználó aktuális adataival. Vagy a **Módosít** gombbal jóváhagyjuk a módosításokat vagy a **Töröl** gombbal töröljük az alkalmazottat, de ha változtatás nélkül akarjuk hagyni, akkor a **Vissza** gombbal tehetjük meg.

10.4.2. Vásárlók karbantartása

Az alkalmazottak mellett a vásárlók adatainak karbantartását is ellátja a rendszer. A vásárlók a vásárlói felületről tudnak elnavigálni egy regisztrációs oldalra, melyen megadhatják adataikat, a rendszerbe történő belépéshez és vásárláshoz szükséges felhasználó nevüket és jelszavukat.

A regisztrált vásárlót egyetlen szerepkörrel látja el a rendszer:

- **Vásárló**, a bejelentkezett vásárlót feljogosítja rendelés feladására.

A vásárlók adatait a **Vasarlo** adatbázistábla tárolja, melyet a **Vasarlo** entitás osztályból generál a perzisztencia prvider. Az entitás osztály a vásárló nevét, címét, telefonszámát és az **Azonosito** entításra egy referenciát tárol, melyet a **@OneToOne** annotációval kapcsol össze mint az alkalmazottak entitás osztálya esetén.

@OneToOne

private Azonosito azonosito;

...

Itt is beágyazott osztályokkal adtam meg a név, cím, telefonszám osztályokat, mert ezzel újra felhasználtam az **Alkalmazott** osztályhoz kapcsolt beágyazott osztályokat.

Az osztályhoz kapcsolódó lekérdezések és a Manager osztály metódusai követik az alkalmazottak karbantartásánál elmondottakat.

A vásárlók adatainak szerkesztését csak az adminisztrátor teheti meg, és táblázatból választhatja ki szerkesztésre az egyes vásárlókat. Egy felhasználó kiválasztása után egy oldalra navigál a rendszer, melyen szerkeszthetjük és menthetjük a rendszer által kitöltött aktuális adatokat. A vásárló adatainak törlésével az általa feladott rendeléseket is törli a rendszer és visszahelyezi az adatbázisba az adott ruhához tartozó megrendelt mennyiségeket. Így a felelőtlen felhasználók által leadott megrendelések konzisztens módon kezelhetők, a felhasználó törlésével.

A vásárlók regisztrációját nem csak a vásárló, hanem a rendszer adminisztrátora is megteheti, melyet ugyanaz az oldal lát el, azzal a különbséggel, hogy az adminisztrátor a rendszerből érheti el az oldalt és nem navigálja ki a rendszerből az adatok felvitele után.



Vásárló regisztráció

Vezeték név:

Kereszt név:

Irányító szám:

Település:

Utca:

Ház:

Felhasználó név:

Jelszó:

Otthoni:

Mobil:

Fax:

Ezen az oldalon egymás után több felhasználó regisztrációját végezheti el, míg a vásárlót az adatok mentése után a bejelentkező oldalra navigálja a rendszer. Ezt a JSF dinamikus navigációs modelljével oldottam meg, mely feltételtől függő vezérlést tesz lehetővé.

10.4.3. Készlet karbantartása

A ruhák karbantartása nem csak a ruha adatainak a felvitelét, hanem a járulékos adatok nyilvántartásba vételét is jelenti, mint például a ruhák mérete és az adott méretből raktáron lévő mennyiségek számát, továbbá a termékekről készült kép nevét. A ruha, a kép neve és a ruha méreteinek és raktáron lévő mennyiségének tárolása mind egy-egy adatbázistáblában történik, melyekre egy készlet tábla hivatkozik külső kulcsok segítségével. A telepítés során a

perzisztencia provider generálja le az adatbázis táblákat a **Ruha**, **Kep**, **Meretdarab** és **Keszlet** entitás beanekből.

A **Ruha** osztály tárolja a ruhák jellemzőit melyek lehetnek külső- és termékjellemzők. Ezek a jellemzők mind egy-egy beágyazott osztályként jelennek meg. A külső jellemzők közé tartoznak a szín és az anyag. A termékjellemzők pedig a termék kód, termék leírás, a márka név és a termék kategória. A minőség és az ár egy-egy további perzisztens adattagja az osztálynak. SQL lekérdezései a **DELETE** és **UPDATE**, melyekhez megfelelő műveleteket definiáltam a **Manager** osztályban.

A **Kep** osztály tárolja a kép nevét, melyet a felületen használ fel a program. A képek kezeléséhez SQL lekérdezéseket **DELETE** és **UPDATE** írtam, melyekkel karbantarthatóak a képekkel kapcsolatos műveletek.

A **Meretdarab** osztály tárolja egy ruha méretét és a raktáron lévő darabszámot. A **Manager** osztály itt is a törlés és a módosítás műveletét látja el, mint az előző osztályok esetén, az új elem adatbázisba írásán kívül.

```
public class Keszlet implements Serializable {  
  
    @Id @GeneratedValue  
    private Long id;  
  
    @OneToOne  
    private Ruha aru;  
  
    @OneToOne  
    private Kep kep;  
  
    @OneToMany  
    private Collection<Meretdarab> meretdarabok;  
  
    ...  
}
```

A **Készlet** osztály adattagjai a **Ruha**, **Kep** osztályok és egy **Meretdarab** osztályokat tartalmazó **Collection**, melyekre értelemszerűen a **@OneToOne** és a **@OneToMany** annotációkkal láttam el. Mivel egy ruhához több méret is tartozhat, ezért választottam a **Collection**-el történő letárolást. Ekkor a perzisztencia provider egy kapcsoló táblát hoz létre, melyet a készlet és a méreteket tároló entitás beanek neveiből generál. Mivel nem használtam olyan eszközt melynek szüksége lett volna arra, hogy külön névvel lássam el a kapcsoló táblát, ezért ezt nem kellett külön annotációval ellátnom.

Az összes készlet lekérdezését és a készlet törlését valósítottam meg, melyekkel minden készletmódosítás elvégezhető.

A rendszerbe az **Árufeltöltő** és az **Adminisztrátor** viheti fel az egyes ruhák adatait. Ezt egy űrlap segítségével teheti meg, melyen meg kell adnia a ruhára vonatkozó összes adatot és ki kell választania a ruhához tartozó képet. Erről az oldalról tovább navigál a rendszer egy olyan oldalra, ahol megadhatjuk a ruhához tartozó méreteket és azok raktáron lévő mennyiségét. Az oldal háttérében egy session életciklusú Menedzselt Bean osztály áll, így minden oldal betöltése esetén megmaradnak a felvitt értékek, melyeket tovább bővíthetünk a mentéssel.

Ruhakészlet

	Márka: Niki Termek leírás: Niki piros nadrág, kellemes viselet. Termek kód: b1232 Termékkategória: Nadrág	Képet módosít Ruhát módosít Darabot módosít
	Márka: Gábor Termek leírás: Gábor zsebes rövidnadrág, mely nyáron kellemes viselet. Termek kód: sd34349 Termékkategória: Nadrág	Képet módosít Ruhát módosít Darabot módosít
	Márka: Gábor Termek leírás: Gábor kényelmes pulcsi, mely kellemes tavaszi viselet. Termek kód: st23f3 Termékkategória: Felső	Képet módosít Ruhát módosít Darabot módosít

A már felvitt ruhákat táblázatos formában tekinthetjük meg és választhatjuk ki karbantartásra a képeket, a méreteket és a ruha adatait. A ruha adatait a felvitelhez hasonló űrlapos formában szerkeszthetjük. A már felvitt méretekhez adhatunk meg darabszámot, egy másik oldalon. A

képek cseréjét egy oldalon tehetjük meg, melyen kiválaszthatjuk a képet és megtekinthetjük, majd az adatbázisba menthetjük a kép nevét.

A ruha törlésével a készlet is törlődik, tehát a kép neve és a hozzátartozó méretek. A készlet törlése a ruhára történt rendeléseket is törli, mivel értelmetlenné válnak a rendelések egy nem létező ruhára.

10.4.4. Rendelések

A rendeléseket a **Rendelésfelvevő** alkalmazott vagy az **Adminisztrátor** teljesítheti. Egy rendelés teljesítése abból áll, hogy az alkalmazott egy táblázatból kiválasztja az egyik rendelési tételt megtekintésre, majd a rendszer átnavigálja egy olyan oldalra, ahol megjelenik a rendeléssel kapcsolatos összes információ, azaz a rendelés időpontja, a ruhára vonatkozó adatok és a vásárló adatai. Miután az alkalmazott feladta a megadott címre a ruhát a **Teljesít** gomb segítségével törli a rendszerből a rendelést.

Rendelés teljesítése



Megrendelés időpontja: 2010.05.09.

Márka: Niki

Termek leírás: Niki, piros nadrág, kellemes viselet.

Termékkód: b1232

Termékkategória: Nadrág

Méret: 47

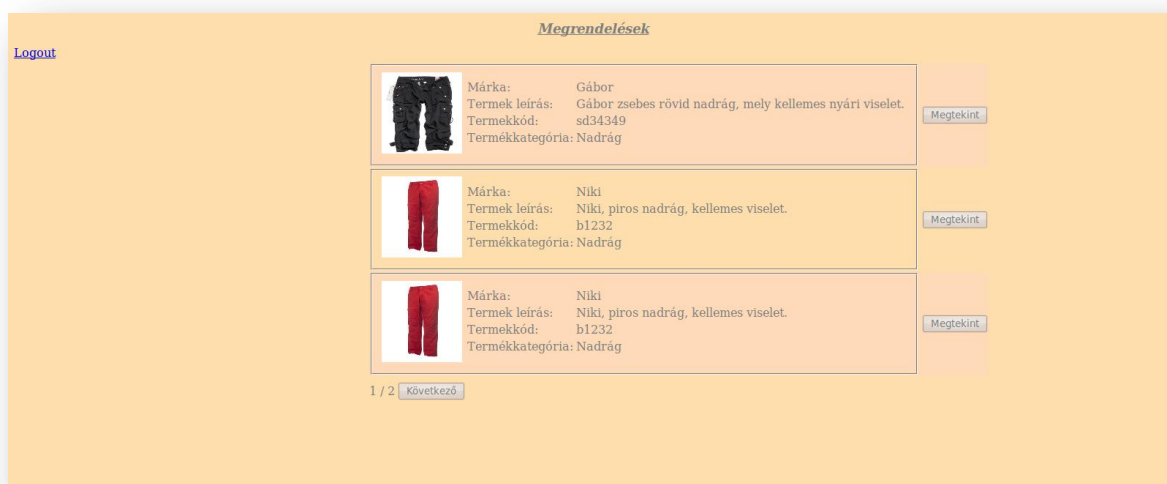
Vásárló neve: Simon Sándor

Vásárló címe: 4321 Debrecen Fenyves 12

Otthoni telefon: 02134567812

Mobil telefon: 02134567813

A táblázatos megjelenítésnél egy oldalon egyszerre három rendelés jelenik meg, lapozással tudjuk elérni a további megrendeléseket. Ehhez minden lapozásnál lekérdezi az összes megrendelést, mellyel azonnal értesül a **Rendelésfelvevő** az újabb megrendelésekről.



A rendelések akkor is törlődnek, ha vagy a megrendelő vásárlót vagy a ruha terméket töröljük a rendszerből.

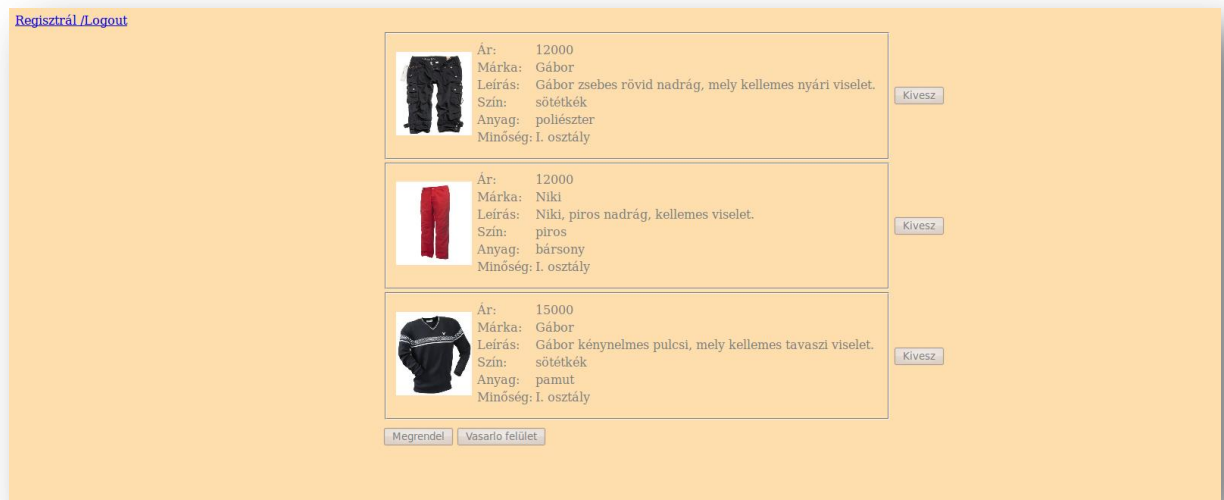
10.4.5. Kosár

A kosarat a hétköznapi életben a megvásárolni kívánt termékek összegyűjtésére és a vásárlásig történő tárolására használják. Egyszerre csak egy terméket lehet a kosárba tenni, mivel általában a vásárló is így cselekszik. Egy termék kosárba helyezése megfelel a készlet egyel csökkentésének is, mivel egy bevásárlás során, ha kosárba helyez egy terméket, és ha ebből a termékből nem található több a polcon, akkor a többi vásárló számára ez a termék elérhetetlenné válik. A készlet csökkentése és növelése nem tartozik a kosár funkcionalitásai közé, mivel ezt a vásárlás során nem a kosár hajtja végre, hanem a vásárló.

A kosár termékeket tárol, melyeket a vásárló a vásárlás során megtekinthet vagy akár ellenőrizheti a termékek árait, tehát ezt az alapvető funkcionalitást egy webáruház kosaránál is el kell látni.

Egyéb követelmények is fellépnek, melyek a hétköznapi értelemben vett kosárhasználatra ritkán jellemzőek. Ilyen probléma lehet, hogy a vásárló megszakítja a vásárlást és nem helyezi vissza a termékeket a készletbe. Ez az eset külön figyelmet igényel.

Tipikusan a Session beanek közé tartozik, ez a **Stateful Bean** látja el a kosár szerepét, mivel a kliensnek mindig ugyanazzal a Beannel kell kapcsolatban lennie. Az alkalmazásban ettől a tipikus megvalósítástól kénytelen voltam eltérni, mivel a szerver nem szolgáltatta megfelelő módon a specifikációban leírt működést és a kliens minden hivatkozáskor egy újabb Kosár példánnyal került kapcsolatba.



A megvalósítást egy egyszerű Java osztály segítségével hajtottam végre, mely példányosítását a vásárló felület vagy a kosár tartalmának megtekintését szolgáló oldal mögött álló Manager Bean hajtja végre, mely akkor példányosítja, ha még nem létezik. A kosarat a **FacesContext Map** kollekciója tárolja, mely az adott klienshez kapcsolódik, ezzel biztosítva, hogy csak az adott kliens érheti el.

```
public class Kosar {  
  
    @EJB private MeretdarabManagerLocal meretdarabManager;  
  
    private Vector<KosarElem> kosarelemek;  
  
    @Override  
    public void finalize() {  
        for(KosarElem e: this.kosarelemek)  
            this.meretdarabManager.novelEgyel(e.getMeretDarab());  
    }  
  
    ...  
}
```

A **Kosar** osztály **KosarElem** osztályokat tárol egy kollekcióban, ahol a kosárba helyezést vagy onnan történő kivételét a **kosarbaTesz(KosarElem kosarElem)** illetve a **kosarbolKivesz(KosarElem kosarElem)** metódusok látják el.

A modellezésnél elmondott egyéb követelmény, azaz a felhasználó által megszakított vásárlás esetére vonatkozó helyzetet a **Kosar** osztály **finalize()** metódusának felüldefiniálásával oldottam meg, mely a **FacesContext** és a kosár megszüntetése előtt hívódik meg, ezzel visszatéve a készletbe a kosárból a ruhákat.

A **KosarElem** osztály a termék képének, a ruhának és egy méretet és darabszámot tároló objektum referenciáját tárolja. Nem elegendő a ruha adatait tárolni, hiszen követelmény, hogy vizuális élménynek is kell társulnia az adott ruhához és adott méretű ruhát lehet vásárolni. A **Meretdarab** osztály a készletre vonatkozó darabszámot is tárolja, melyet csak a készlet egyel történő növeléséhez vagy csökkentéséhez használ fel a program, ezzel kiküszöbölve a felesleges lekérdezéseket.

10.4.6. Vásárló felület

A kezdő oldalon a készlet termékei jelennek meg táblázatos formában. Itt választhatjuk ki az egyes ruhákat megtekintésre. A megtekintés egy külön oldalra navigál, ahol kiválaszthatjuk a rendelkezésre álló méretek közül melyikből szeretnénk a kosárba tenni. a kosárba tevést a **Kosárba tesz** gomb megnyomásával teheti meg a felhasználó.



Ár:	12000
Márka:	Gábor
Leírás:	Gábor zsebes rövid nadrág, mely kellemes nyári viselet.
Minőség:	I. osztály
Termék kategória:	Nadrág
Szín:	sötétkék
Anyag:	poliészter
Termék kód:	sd34349

49 ▾

Kosárba tesz!

Vissza

A főoldalról a kosárhoz navigálhatjuk a rendszert, melyben megjelennek az egyes ruhák, melyeket kiválasztottunk megrendelésre. Az egyes ruhákat a **Kivesz** gombbal vissza tehetjük a készletbe, vagy feladhatjuk a megrendelést.

A megrendelés feladása bejelentkezéshez kötött, ezért, ha a felhasználó nincs bejelentkezve, akkor a rendszer a bejelentkező oldalra irányítja a felhasználót.

Összefoglalás

Az Ada és az EJB részben elmondottakból világosan kiderül, hogy bizonyos szempontból az EJB technológia inkább a Java egy módszertani hiányosság pótlásának is tekinthető. Véleményem szerint, ezzel a hiánypótlással, a memória használat miatt megtett egyéb eszközbővítések és a Java 5 által bevezetett nyelvi eszközöknek, az annotációknak köszönhetően, melyek kiküszöbölik a nyelvi eszközök helytelen használatát, az EJB 3-mal az EJB technológia nem csak elméleti, hanem gyakorlati szempontból is megalapozottá vált.

Sajnálatos módon a technológiát támogató eszközökről általánosan elmondható, hogy nem minden esetben nyújtják megfelelően a specifikációban leírt működést. Emiatt a gyakorlatban a fejlesztések jelentős hányada megbukik vagy jelentősen csúszik. Ezekkel a problémákkal magam is találkoztam a fejlesztés során és véleményem szerint nem csak a technológiák, hanem az őket támogató eszközök terén is jelentős tapasztalattal kell rendelkeznie egy fejlesztőnek.

Sikerült betekintést nyernem a JPA használatának alapjaiba, melyről elmondható, hogy jól használható kényelmes eszközt nyújt az adatbázissal kapcsolatos műveletekhez. Azonban az is helytálló meglátás, hogy csupán a JDBC eszközök használatát küszöböli ki. Mint minden magasabb szintű eszköznél ennél is megfelelőnek a szakemberek a hatékonyságról és komoly háttér munkával tudják csak felderíteni a problémákat. Mindezek ellenére a kód karbantartását jelentősen javítja, ezzel kiküszöböli a forráskódok összehangolását.

Az általam fejlesztett alkalmazás, egy webáruház minden funkcionalitását ellátja, a rendelés feladásával, a felhasználók és termékek karbantartásával, és a rendelések teljesítésével. Tehát minden szakterületi követelménynek eleget tesz.

Köszönetnyilvánítás

Köszönetet mondok Kósa Márknak, aki kiemelkedő szaktudásával és kulcsfontosságú észrevételeivel elősegítette diplomamunkám elkészülését.

Irodalomjegyzék

Nyékiné Gaizler Judit(szerk.): Az Ada95 programozási nyelv, ELTE Eötvös Kiadó 1998

Imre Gábor(szerk.): Szoftverfejlesztés Java EE platformon, Szak kiadó 2007

TopLink

<http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-annotations.html>

GlassFish

<https://glassfish.dev.java.net/docs/project.html>

Java EE

<http://java.sun.com/javaee/5/docs/tutorial/doc/?wp406143&PersistenceIntro.html#wp78460>

JavaServer Faces

<http://java.sun.com/javaee/jaserverfaces/reference/api/index.html>