Debreceni Egyetem

Informatika Kar

KÖLCSÖNZŐ RENDSZER FEJLESZTÉSE JAVA EE PLATFORMON

Témavezető:

Kollár Lajos

Egyetemi tanársegéd

Készítette:

Buka Tamás

Programtervező Informatikus

Debrecen

2010

Tartalomjegyzék

1. Bevezetés

1990-ben a Sun Microsystems beágyazott operációs rendszereket fejlesztett elektronikai készülékek számára Green Project néven. Ebben a fejlesztésben vett részt James Gosling, Patrick Naughton és Mike Sheridan. 1991-ben a C++ fordító platform független, objektum orientált kiterjesztésével foglalkoztak mellyel különféle eszközök kommunikációját szerették volna megvalósítani. Gosling felismerte, hogy ez a C++ nyelvvel lehetetlen így megalkották a saját nyelvüket Oak-ot, ami angolul tölgyfát jelent. A név állítólag onnan ered, hogy mikor kinézett Gosling az ablakon egy tölgyfát látott meg. Az Oak név már létezett, mint bejegyzett név, így 1995-ben átnevezték Java-ra. A java hamarosan népszerű lett az internet és a World Wide Web gyors terjeszkedése miatt, és mint biztonságos nyelv vált ismerté. 1996-ban a Microsoft elkezdi fejleszteni a saját Java verzióját, hogy versenyképes maradjon. Ezt "polluted Java"-nak nevezte a szaknyelv, mivel a Microsoft próbálta ezt a windows operációs rendszerhez kötni, így már nem volt platformfüggetlen. A Sun pert indított, mivel ez is Java néven futott, így a Microsoft kénytelen volt felhagyni eme nyelv fejlesztésével, de ekkorra már meg alkották a .NET-et ami máig versenytársa a Java-nak.

A platformfüggetlenséget a Java-ban az biztosítja, hogy a forrás kódot a fordító egy bytekódra fordítja, amit pedig egy Java Virtual Machine futtat. A bytekód hordozható, platform független, kompakt. A JVM egy fizikai hardver működését szimulálja. Tervezéskor szempont volt, hogy minél több környezettel kompatibilis legyen, így széles körben lehessen használni.

A JVM fontosabb elemei:

- Osztálybetöltő (class loader), a főbb ellenőrzéseket végzi a byte-kódon, előkészíti futtatásra
- Szemétgyűjtő (garbage collector), működés közben a nem használt objektumokat eltávolítja a memóriából, ezzel helyet szabadít fel
- Végrehajtó motor (execution engine), a tulajdonképpeni végrehajtást végzi

A Java nyelv fejlesztésénél négy fontos szempontot tartottak szem előtt, ezek:

- Objektum-orientáltság;
- Függetlenség az operációs rendszertől, amelyen fut (platformfüggetlenség);

- Olyan kódokat és könyvtárakat tartalmazzon, amelyek elősegítik a hálózati programozást;
- Távoli gépeken is képes legyen biztonságosan futni.

A nyelv egyik legfontosabb és legjellemzőbb tulajdonsága az objektum-orientáltság, ami a programozási stílusra és a nyelv struktúrájára utal. Úgynevezett objektumokat hozunk létre és ezeknek az állapotát manipuláljuk. A fejlesztés célja, hogy nagyméretű projekteknél is átlátható kódot kapjunk, amiket az objektumokkal meg lehet valósítani. Így kevesebb lett az elrontott projekt. A Java három szerepet tölt be: programozási nyelv, middleware, platform. A Java legfontosabb eleme a JVM, ami mindenhol jelen van, így mind középszinten, mind platformon jól működik. Ugyanakkor platformfüggetlen is a nyelv, mivel a bytecode-ot interpretálja a JVM. Ez azt jelenti, hogy egy megírt program minimális módosítás után más platformon is működni fog. Innen ered az „írd meg egyszer, futtasd bárhol” elve, ami jelentős költségcsökkentést jelent és csökkenti a fejlesztési időt, mert csak egyszer kell megírni a kódot. Javaban megírt programok hasonlóan fognak futni különböző hardvereken. Ezt a JVM bytecode-ról gépikódra fordításával éri el. Vannak olyan fordítók is, amelyek a forráskódot közvetlen gépikódra fordítják le, ilyen a GCJ. Így a program elveszti hordozhatóságát, bár nagyobb sebességgel fut. A hordozhatóság megvalósítása nagyon komplex feladat volt, ami a Javafejlesztői berkein belül rengeteg vitát váltott ki. Az "írd meg egyszer, futtasd bárhol" szlogenből "írd meg egyszer, hiba keress mindenhol" lett. Ennek ellenére a Java nyelv sikeres lett a servlet, a JSP és Enterprise JavaBeans technológiákkal, amiket a Java Enterprise Edition tartalmaz. Ezt az elosztott, sok felhasználós, vállalati szoftverrendszerek fejlesztésére alkották meg. Az Enterprise Edition tartalmazza a Standard Edition API-jait is, így mindkét technológia felhasználható. A Java EE támogatja a vállalati méretű rendszerek fejlesztését, mégpedig úgy, hogy a felmerülő követelmények (pl: perzisztenci, biztonság, többszálúság) megoldására ad egy olyan keretrendszert, amelyben szolgáltatásokkal oldhatók meg ezek a követelmények. Ezek a követelmények általában a következők:

- Perzisztencia: Az adatokat úgy kell tárolni, hogy a program leállítása után is megmaradjanak. Erre egyik megoldás az adatbázisok használata. Olyan támogatást várunk mely elfedi előlünk az adatbázist és összehangolja az objektumorientáltságot az adatbázissal. Objektumorientált adatbázis esetén ez meglehetősen egyszerűvé válik.

- Tranzakciókezelés: Az adatoknak mindig konzisztens állapotban kell lenniük, akkor is, ha többszörös hozzáférésről beszélhetünk egy adott időben. Teljesülni kell az adatbázisokban elvárt összetartozó módosításnál, hogy vagy minden módosítás sikerüljön, vagy egyik se.
- Többszálúság: Egy rendszert több kliens is elérheti egy azon időben, ezért a gyorsaság biztosítása érdekében célszerű több szálat kezelni.
- Távoli elérés: Amennyiben webes felületről szeretnénk elérni a szervert, amin az adatok vannak, így ehhez szükségünk van HTTP protokollra. Ezen keresztül kommunikál a szerver a klienssel, ami jelenesetben egy böngésző. Ha más rendszer kapcsolódik a rendszerünkhöz, vagy úgy nevezett vastag kliens, akkor is szükségünk van protokollra, ami a kapcsolatot biztosítja.
- Névszolgáltatás: Elengedhetetlen követelmény, hogy ne IP-cím alapján érhessük el a számunkra hasznos objektumokat, hanem egy névvel, hiszen az IP-cím változhat.
- Skálázhatóság: A rendszer terhelése nagyságrendekkel megnőhet. Logikus követelmény, hogy ekkor is működjön elfogadható sebességgel és ne omoljon össze. Beszélhetünk függőleges és vízszintes skálázhatóságról. Míg a függőleges erősebb hardvert jelent, addig a vízszintes kisebb teljesítményű több gépet.
- Magas rendelkezésre állási idő: A rendszer nem állhat le hibák miatt, nem válhat elérhetetlenné.
- Aszinkron üzenetkezelés: Előfordulhat, hogy egy alkalmazás nem elérhető, ilyenkor nem akarunk hibaüzenetet kapni, azért, mert nem tudta továbbítani az üzenetet egy másik alkalmazás. Célszerű ilyenkor megtartani addig az üzenetet, amíg elérhetővé válik az alkalmazás, majd akkor elküldeni.
- Biztonság: Ez egyrészt a jogosultság kezelés, másrészt az adatbiztonság, rosszindulatú módosítás megelőzéséről szól. Célszerű kész megoldást választanunk.
- Monitorozás, beavatkozás: A futási folyamatok nyomon követésére és esetleges megváltoztatására felmerülhet az igény. Célszerű, ha ez már futásidejű szolgáltatásként elérhető.

Ezeket a szolgáltatásokat middleware-szolgáltatásoknak is nevezik, hiszen a Java EE-ben is egy köztes rendszer biztosítja ezeket. A Java EE is támogatja ezeket a követelményeket saját API-n keresztül, melynek egy rész már a SE-ben is szerepel.

2. Alkalmazás szerver és n rétegű architektúra

A Java EE futási környezete az alkalmazás szerver, mely egy n rétegű szoftverarchitektúrára illeszkedik. Jellemzője, hogy mindegyik egy speciális feladatot végez el az alatta lévő réteg segítségével. Az adatréteg feladata az adatok perzisztens tárolása és módosítása műveletekkel. Gyakran alkalmaznak erre relációs adatbázisokat, de újabban objektumorientált adatbázisok is egyre elterjedtebbek. Bővebben érthetjük az adatréteget úgy, hogy minden olyan alkalmazás beletartozik, amelyből adatot nyerhetünk ki. Ilyenkor szokás Enterprise Information System-nek, azaz EIS-nek nevezni.

Erre a rétegre épül az üzleti logika réteg, amely az üzleti szabályoknak megfelelően hívja meg az adatrétegben definiált szolgáltatásokat az alkalmazási terület igényeihez igazodva. Ilyen a kölcsönző rendszerben a filmek kikölcsönzésekor egy kölcsönzési adatok rögzítése.

A felhasználókkal kapcsolatot tartó réteg a kliensréteg, amely azt biztosítja, hogy a felhasználó olyan funkciókat tudjon meghívni, amelyek az üzleti logikában definiáltak. Majd pedig az eredményt kiírja a felhasználói felületre. Két fajta kliensről beszélhetünk. A hagyományos desktop alkalmazások vastag, míg a webes az úgynevezett vékony kliens. Újabban megjelennek a gazdag kliensek, mely a két klienstípus elmosódásából jön létre. Ezek a böngészőn nyújtják ugyanazt a szolgáltatást, mint egy vastag kliens. Lehetséges az alkalmazás szerverhez több fajta klienssel is csatlakozni, ilyenkor, ha az üzleti logika az alkalmazás szerveren van, akkor a különböző kliensekbe nem kell ezt újra belekódolni.

Webrétegre akkor van szükségünk, ha vékonyklienseket is ki akarunk szolgálni, hogy a böngészőktől érkező HTTP-kérésekkel meg tudjuk hívni az alkalmazásszerverbe kódolt üzleti logikát. Java EE esetén a webréteg is az alkalmazás szerveren van. Gyakran beiktatnak egy webszervert is, hogy a statikus kéréseket kiszolgálja, bár az alkalmazásszerver közvetlen is képes a böngészőkkel kapcsolatot tartani és a kiszolgálni őket. Ha van webréteg, akkor csak azokat a kéréseket továbbítják az alkalmazásszerverhez, amelyek üzleti logikát is igényelnek.

Erre az architektúrára szokás 3 rétegűként is hivatkozni, ekkor adat-üzleti logika-kliensréteg modelt értjük alatta. Helyesebb n rétegűről beszélni, hiszen bizonyos esetekben közé illeszthető a webszerver, azaz a webréteg. Érdemes megjegyezni, hogy a meglévő rétegek készen lévő szoftverekre épülnek, amelyek akár külön gépeken is futhatnak, de közülük több akár egyazon gépen, ezért jobb az n rétegű elnevezés.

Az alkalmazásszerver egyik fontos alappillére az Enterprise JavaBeans, azaz az EJB. Ezek azok a komponensek, amelyekben az üzleti logikát szokták megvalósítani. A fejlesztett EJB-k távolról is elérhetők a hálózat részletes ismerete nélkül a Remote Method Invocation over Internet Inter-ORB Protocol-on keresztül. Az RMI a szabványos módszer a kommunikációra az elosztott objektumok között. A távoli eljáráshívással, képesek az appletek és vastag kliensek kommunikálni az EJB-kel. Az RMI-IIOP és az RMI között annyi a lényeges különbség, hogy az előbbi az IIOP protokollt használja az adatok továbbítására és fogadására a hálózaton. Mivel az IIOP a CORBA protokollja, így CORBA kliensekből közvetlenül is meghívhatók az EJB-k.

A böngészők HTTP-n keresztüli kiszolgálását végző komponensek, olyan webkomponensek amelyek szervletek, JavaServer Pages vagy JavaServerFaces oldalak lehetnek. Ezek meghívhatják az EJB-ket vagy hagyományos metódushívással, vagy RMI-IIOP-n keresztül. Előbbi esetben követelmény, hogy közös alkalmazásszerveren fussanak.

Az utóbbi években kialakult az XML webszolgáltatások technológia. Ezek platformfüggetlen távoli elérést biztosítanak más webszolgáltatásokhoz vagy EJB-khez.

Az alkalmazásszerver komponenseinek az alkalmazásszerverhez befutó kérések kiszolgálása során hozzá kell férniük a háttérrendszerekhez. Ezek közös jellemzője, hogy mindegyik egy úgynevezett konténerben fut, ami a middleware szolgáltatásokat biztosítja futási időben. A komponenseket telepítési információ nélkül lehet fejleszteni a lazán kapcsoltságuk miatt. Ezen komponensek mindig újrafelhasználhatók, hordozhatók.

3. EJB

Az Enterprise JavaBeanek olyan komponensek, amelyek a Java EE szolgáltatás nyújtásában központi szerepet játszanak a rendszer építőelemiként. Három fajta EJB-ről beszélhetünk jelenleg. Ezek a session beanek, etity beanek és a message-driven beanek. A session az üzleti folyamatokat reprezentálja pl: filmkölcsönzés. Az entity az üzleti modelt pl: egy filmet, ez az adatbázishoz kapcsolódik. A message-driven bean pedig az aszinkron üzenetkezelésre használható.

Az EJB 3 a 2.1-es EJB egyszerűsítése volt, amiben sok dolog bonyolulttá tette a fejlesztést. Ezek a következők:

- Bonyolult a fájlok karbantartása, még ha támogatást is biztosít hozzá a rendszer.
- A javax.ejb.Session/Entity/MessagaDriven interfészt kötelező implementálni akkor is, ha nincs szükség bizonyos metódusaira.
- A JDNI-hez kötelezően többsornyi kódot kell írnia, ami fejlesztő eszköz hiányában nehézkes.
- Az Entity beanek csak session facade-dal együtt célszerű használni, ami redundanciát jelent a DTO-k karbantartása.
- EJB-k tesztelése szintén problémás, hiszen szükség van hozzá egy konténerre.

Az EJB 3 ezek megoldására született úgy, hogy az EJB előnyei továbbra is megmaradjanak. Ezek a megoldások az annotációk.

3.1. Annotációk

Olyan metaadat mely azt módosítja, hogy a különböző eszközök és osztály könyvtárak hogyan kezeljék az adott komponenst. Tehát nem közvetlen a komponenst módosítja, csak annak felhasználását. Az annotációk a metadatok beszúrásához és definiállásához határoz meg egységes szintaxist. Általános szintaxisa:

@AnnotáciNeve(paraméter1=érték1, paraméter2=érték2…)

ha csak egy paraméter van, akkor megengedett csak az érték átadása paraméter név nélkül:

@AnnotációNeve(érték)

ha nem adunk át paramétert megengedett a zárójel elhagyása így mind a @AnnotációNeve mind a @AnnotációNeve() használható.

A paraméter tömb is lehet ekkor a szintaxis: @AnnotációNeve(paraméter1={elem1, elem2,…}, paraméter2=érték2…) ami a szokásos tömb szintaxist követi.

Az annotáció típusát definiálni kell egy interfészben, csak így helyezhetjük el az adott annotációt a kódban. Itt kell megadni a nevét, paraméterei típusát. Használatához mindig importálni kell a tartalmazó interfészt. Alapértelmezett értéket egy paraméternek a default kulcsszóval adhatunk meg. Ha pedig csak egy paraméter van, annak mindig value a neve. A

paraméter típusai primitív típusok vagy ezek tömbje. Metaanotációk: olyan annotációk, amik az annotációk felhasználását módosítja.

A Java EE egy sor annotációt definiál, amit a fejlesztők használhatnak. Az annotációk átveszik a telepítésleírók szerepét, még pedig úgy, hogy az alkalmazásszerver értelmezi az annotációkat a komponensek telepítésekor és az annotációk által definiált futás idejű szolgáltatásokat biztosít a komponenseknek. A telepítésleírók azonban továbbra is használhatók, így vegyes használat esetén felüldefiniálják az annotációkat. Ez akkor fontos, ha kódunkat újrafelhasználhatóvá akarjuk tenni egy harmadik félnek úgy, hogy nem adjuk ki a forráskódot. Ebben az esetben képes felüldefiniálni anélkül, hogy látná a kódot. A megadandó metaadatok minimalizálása érdekében minden metaadatra ad alapértelmezett értéket a Java EE, így csak abban az esetben kell konfigurálni a komponenseket, ha el akarunk térni ezektől az alapértelmezett értékektől.

3.2. Session bean

Képzeljünk el egy olyan session beant, ahol minimalizálni akarjuk a leírt kódot és szükséges fájlok számát! Az EJB által nyújtott implicit middleware koncepcióból adódóan az interfész és az implementáció szétválasztása szükségesnek tűnik, hogy a kliensek az interfészt megvalósító, konténer által generált csonkot hívhassák. Így tehát két fájlra lesz szükségünk, melyek nevei a korábbiakhoz hasonlóan Enterprise bean (vagy implementációs) osztály, illetve a business interfész. A kódot minimalizálandó, nem akarjuk viszont, hogy ezeknek bármilyen EJB-specifikus interfészimplementálást vagy –kiterjesztést kelljen végezniük, vagyis azt szeretnénk, hogy POJO-k (Plain Old Java Object) legyenek. Továbbá nem akarunk telepítés leírót sem.

Az egyszerűsítés természetesen nem történhet a működőképesség rovására, így megoldást kell találni ezekre a problémákra:

- Össze kell rendelni az implementációs osztályt a bisuness interfésszel.
- Definiálni kell a session bean állapotkezelését.
- Az EJB életciklusa során értesülni akarunk bizonyos eseményekről.
- Inicializálni akarjuk a session beant, és referenciát szerezni rá.

A bean osztály és business interfészének összerendelésére kézenfekvő megoldás az implements kulcsszó. Ez valójában elegendő is lesz, csak azt kell tisztázni, hogy távoli vagy lokális lesz-e az interfész. Az EJB-specifikáció szerint a lokális az alapértelmezett, amit a @Remote annotációval változtathatjuk meg. Ezt akár a bean osztályon, akár a business interfészen alkalmazhatjuk. Elképzelhető, hogy a bean osztály több interfészt is implementál. Ekkor a @Remote, illetve a @Local annotáció paraméterében kell megadni, hogy ezek közül mely a távoli és mely a lokális interfész. Az implementált interfészek számának megállapításakor nem kell beszámítani a java.io.Serializable, java.io.Externalizable, és a javax.ejb csomagban lévő interfészeket.

Az EJB 3-ban szintén létezik állapottal rendelkező és állapotmentes session bean. Az állapotmentesség azt jelenti, hogy a kliens nem számíthat arra, hogy végig ugyanazzal a beannel fog kommunikálni, emiatt nem tárolhat benne állapotot.

Az állapotmentes session bean osztályát a @Stateless-szel, állapottal rendelkezőjét @Stateful-lal kell megjelölni. A korábban kötelezően implemetálandó callback metódusok helyett új koncepció jelenik meg. Csak akkor kell speciális annotációkkal megjelölt callback metódusokat írnunk, ha azt akarjuk, hogy a session bean értesüljön életciklusának bizonyos eseményeiről. Állapotmentes session bean esetén csak két életcilus-esemény következhet be: létrehozás után a konténer a @PostConstruct-tal megjelölt metódust hívja meg a beanen, ha van ilyen, megsemmisítés előtt pedig a @PreDestroy-jal jelöltet. Ezek a korábbi ejbCreate és ejbRemove metódusokat helyettesítik. Állapottal rendelkező esetben ez kiegészül az ejbPassivate és ejbActivate helyett megjelenő @PrePassivate és @PostActivate callback-ekkel. Ezenfelül, állapottal rendelkező esetben a kliensnek valamilyen módon lehetőséget kell adni arra, hogy elengedje a hozzá rendelt session beant. Ezt a @Remove-val megjelölt metódus meghívásával kell megtennie, aminek hatására a konténer a megsemmisítés előtt meghívja a PreDestroy-jal jelölt metódust. Az EJB 3 lehetőséget ad arra, hogy ezeket a callback metódusokat külön osztályba rakjuk és ne az implementációs osztályba.

A session bean kliense az alábbi módon szerez referenciát a beanre. Az első egyszerűsítés, hogy a session bean-nek nem muszály JDNI nevet adni a telepítés leíróban expliciten, mert alapértelmezetten kapja a teljesen kvalifikált nevét. Ha mégis más nevet akarunk neki adni, akkor azt a @Stateful vagy @Stateless name paraméterében tehetjük meg. Az EJB kliensei, amennyiben szintén konténerben futnak, azaz web-komponensek vagy más EJB-k, a

függőséginjektálással nagyon kényelmesen használhatják az EJB-t. Ez azt jelenti, hogy egy business intefész típusú változót @EJB-vel megjelölve a konténer feladata lesz példányt biztosítani azon a változónéven, ezzel kiküszöbölve a JDNI-keresést. A függőséginjektálás nemcsak EJB-k, hanem más erőforrás keresésére is alkalmas, az általános @Resource annotációval. A konténeren kívüli klienseknek továbbra is explicit módon kell JDNI keresést végrehajtaniuk, a keresés eredménye azonban már nem home interfész, hanem egyből referencia a bean példányra a business intefészén keresztül. További egyszerűsítés, hogy a megszerzett példányon hívott metódusok által dobott rendszerszintű kivételt nem kötelező elkapni, mert a fellépő rendszerszintű kivételeket a konténer a javax.ejb.EJBException-be csomagolva dobja tovább. Ez a RuntimeException gyermeke, vagyis nem kötelező lekezelni.

3.2.1. Interceptorok

Az interceptorosztályok olyanmetódusokat tartalmaznak, amelyeket az EJB-konténer egy session vagy message-driven bean metódusainak meghívása előtt hív meg. Az interceptormetódusok módosíthatják a bemenő paramétereket, akár meg is akadályozhatják a ténylegesen meghívott metódus meghívását. Bevezetésükkel az EJB 3 kezdetleges támogatást nyújt az aspektusorientált programozáshoz (AOP). Egy interceptor tetszőleges Java-osztály lehet, paraméter nélküli konstruktorral. Egyetlen interceptormetódust tartalmazhat, amit @AroundInvoke-kal kell megjelölni, és Object visszatérési értékkel kell deklarálni. Ez a metódus dobhat kivételt, használhat függőséginjektálást és a hívott EJB biztonsági és tranzakciós kontextusában fut. Bemenő paramétere egy InvocationContext típusú változó, ezen keresztül éri el a megszakított metódust, az annak átadott paramétereket, a bean példányt és ebbe rakhat névvel azonosított adatokat, amelyeket más interceptorok majd felhasználhatnak. Az eddig elhangzottak az ún. üzleti metódus interceptorokra vonatkoznak. Mint korábban említésre került az életciklus-eseménynek callback metódusait is ki lehet rakni interceptor osztályokba. Ilyenkor @AroundInvoke helyett megfelelően @PostConstruct, @PreDestroy, @PrePassivate, @PostActivate annotációkkal kell megjelölni őket és void visszatérési értékkel kell, hogy rendelkezzenek. Az interceptorok EJB-khez rendelése történhet osztályszinten, vagy az egyes metódusok szintjén is. Mindkét esetben több interceptorosztályt is megadhatunk a @Interceptors annotációval és azok sorrendjében fognak meghívódni az interceptormetódusok. Ha csak egy interceptort akarunk megadni, akkor a @Interceptor annotációt is használhatjuk, kiküszöbölve a tömbszintaxist. Így a

@Interceptors({Interceptor1.class}) helyett használhatjuk a @Interceptor(Interceptor1.class)-t. Az interceptorok EJB-hez rendelése történhet a telepítésleíróban is, amely felüldefiniálja az annotációs definíciókat. Sőt, a telepítésleíróban az EJB jar-fájlban lévő összes EJB-re érvényes alapértelmezett interceptorokat is definiálhatunk.

3.3. Entitások

Az entity beanek mindig is az EJB technológia legvitatottabb részét képezték. Kezdetben a BMP esetén fellépő N+1 probléma okozott jelentős teljesítményromlást. Ezt ugyan megoldotta a megjelenő CMP, de így is maradtak fejlesztési nehézségek:

- Az entity beanek alkalmazásakor szükséges DTO tervezési minta növelte a fejlesztendő osztályok számát.
- A CMP nem volt egyszerűen hordozható az EJB-konténerek között, mert az objektumrelációs leképezés részleteit konténer specifikus fájlokban kellett definiálni.
- Az entity beanek öröklésének támogatását nem írta elő a szabvány, így ez is konténerfüggő volt.

Az automata objektumrelációs leképezés ugyanakkor alapvetően hasznos képesség, ami sok JDBC kódolástól szabadítja meg a fejlesztőt, így több alternatív megoldás is született erre a feladatra. Ezek közös jellemzője, hogy EJB-konténertől független megoldások, vagyis Java SE-alkalmazásokban is elérhetők, és hogy a perzisztens objektumok egyszerű Java-osztályok példányai, így egyszerűbb fejlesztési modellt biztosítanak. Az EJB 3-specifikáció ezen keretrendszerek tapasztalatait felhasználva egy új perzisztenciamegoldást tartalmaz, a Java Persistence API-t (JPA), amely szintén POJO-alapú, és akár Java SE-ben is használható. Fontos megjegyezni, hogy bár a JPA is az entitás megnevezést használja a perzisztens objektumokra, a JPA-entitások nem tekinthetők az entity bean technológia új verziójának, mivel az EJB 3-specifikáció tartalmazza az entity beaneket is, a 2.1-es verzióhoz képest változatlan formában. Ugyanakkor várható, hogy a JPA – egyszerűségének és több lehetőségének köszönhetően – a jövőben lényegében felváltja az entity bean technológiát.

3.3.1. Objektumrelációs leképezés annotációkkal

Egy hagyományos Java-osztály egészen egyszerűen tehetünk EJB 3-entitássá. Tegyük fel, hogy a filmek adatait akarjuk tárolni. Ehhez egy argumentumot nem váró konstruktorral rendelkező Film osztályt kell írnunk, megfelelő attribútumokkal és a hozzájuk tartozó getter/setter párosokkal, majd két JPA-annotációt kell alkalmaznunk: @Entity-vel jelöljük meg az osztályt, @Id-vel az elsődleges kulcsot. Ezek mindegyike a javax.persistence csomagban található, mint a JPA összes többi típusa is.

CMP entity beanek esetében konténerspecifikus leírófájlokban kellett deklarálni, hogy az egyes attribútumok milyen nevű tábla milyen nevű oszlopára képződjenek le. A JPA-ban azonban ilyet nem találhatunk, mert alapértelmezett értéket definiál: az osztályéval azonos nevű táblának az attribútummal megegyező nevű oszlopában tárolódnak az adatok. Természetesen ez megfelelő annotációkkal felüldefiniálható: az osztályszinten alkalmazható @Table name, catalog, schema paraméterével, illetve a perzisztens attribútumokra alkalmazható @Column annotáció name paraméterével. Ez utóbbi annotáció számos egyéb paramétert is elfogad, melyekkel az oszlop egyéb tulajdonságait is megadhatjuk. Arra is van lehetőség, hogy több relációs táblában tároljunk egy entitástípust, ehhez a @SecondaryTable annotációt kell alkalmaznunk.

Az entitások rendelkezhetnek tetszőleges típusú nem perzisztens attribútumokkal is, ezeket transient módosítóval, vagy @Transient annotációval kell megjelölni. A perzisztens attribútumok típusára létezik némi megkötés, hogy biztosítható legyen az SQL-típusoknak való megfeleltetésük: primitív típusok, azoknak megfelelő objektumtípusok, String, java.math.BigInteger, java.math.BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, byte[], Byte[], char[], Character[], enum, más entitások, azok gyűteményei, illetve úgynevezett beágyazott osztályok képezhetik egy JPA-entitás perzisztens attribútumait. A beágyazott osztály egy olyan speciális típus, mely önmagában nem él perzisztens entitásként, csak egy perzisztens entitáspéldányhoz kapcsolódva.

Mint tudjuk a java.util.Date és java.util.Calendaar osztályok a napon kívül az időt is tartalmazzák, milliszekundum pontossággal. Az ennek megfelelő SQL-típus a TIMESTAMP, ezért ezek a Java-típusok alapértelmezetten erre fognak leképződni. Lehetséges azonban, hogy alkalmazásunkban, csak a dátum, vagy csak az időt akarjuk felhasználni, azaz az adatbázisban elég egy DATE, illetve egy TIME SQL-típus. Ilyenkor a @Temporal

annotációval kell megjelölnünk az entitás adott perzisztens attribútumát, a TemporalType.Date vagy TemporalType.Time enum értékkel.

A perzisztens attribútumok közül kitüntetett szerepe van a kötelező elsődleges kulcsnak. Ennek értékét tilos megváltoztatni az alkalmazásban, típusára pedig erősebb megkötések vonatkoznak: primitív típusok (a lebegőpontos kivételével), nekik megfelelő csomagoló osztályok, String, java.util.Date, java.sql.Date lehet. A 2.1-es entitiy bean egyik hátránya az volt, hogy nem támogatta az elsődleges kulcsok automatikus generálását. Az EJB 3 entitások erre már képesek, mindössze a @GeneratedValue-val kell megjelölni az elsődleges kulcs mezőt, aminek típusa ekkor valamilyen egész kell, hogy legyen. A generálás miként je a strategy paraméterben definiálható, amely a GenerationType enum 4 értékét veheti fel:

- SEQUENCE választásával egy adatbázis által kezelt számláló segítségével fognak generálódni az értékek.
- IDENTITY megadásával az adatbázisbeli tábla egy autoinkrementálódó oszlopára fog leképződni az elsődleges kulcs attribútum.
- TABLE választásával és a @TableGenerator annotáció egyidejű használatával egy adatbázisbeli tábla adott sorának adott oszlopában található értéke lesz a következő generált érték.
- AUTO esetén az előző három valamelyike fog érvényesülni, figyelembe véve az adatbázis-kezelő típusát.

Összetett elsődleges kulcs használata szintén megoldható, a @IdClass annotációval.

Ha az attribútumokat jelöljük meg az objektumrelációs leképezés részleteit definiáló annotációkkal. Akkor olyan működés érhető el, hogy amikor az objektum adatai betöltődnek az adatbázisból, közvetlenül a tagváltozóba töltődnek be, nem pedig a setterek hívódnak meg. Fordított irányban, vagyis az adatok visszaírásakor szintén nem gettereken keresztül elért adat fog az adatbázisba bekerülni, hanem közvetlenül a tagváltozók értéke. Lehetőség van arra is, hogy az annotációkat közvetlenül a tagváltozók helyett a getterekre vagy setterekre alkalmazzuk. Ekkor ezeken keresztül fog történni az adatok betöltése/visszamentése. Ennek megválasztásának akkor van jelentősége, ha a getter/setter nem változatlan formában adja vissza az attribútumok értékét. Az annotációk elhelyezésekor ügyelni kell a konzisztenciára, vagyis egy adott entitásnál mindvégig attribútumokat vagy mindvégig a hozzáférést biztosító

metódusokat kell megjelölnünk, ellenkező esetben nem definiált a működés. A metódusok megjelölése esetén fontos figyelembe venni, hogy nem garantálható, hogy a perzisztencia provider milyen sorrendben tölti be az adatokat a setterek segítségével.

3.3.2. Az EntityManager és a perzisztenciakontextus

Az entitások életciklusának kezelését, az adatbázis- és memóriabeli adatok szinkronizálását, tulajdonképpen minden perzisztenciával kapcsolatos futási idejű szolgáltatást az ún. perzisztencia provider biztosítja. A JPA, sok más Java EE API-hoz hasonlóan, alapvetően interfészeket definiál, amelyek mögött tetszőleges implementáció biztosíthatja az előírt működést. Elterjedt perzisztencia providerek például a TopLink, Hibernate, a Kodo, az Open JPA, melyek Java SE alkalmazásokhoz külön osztálykönyvtárak formájában kapcsolhatók, Java EE-alkalmazásszervereken pedig a konténer által biztosított futási környezet tartalmaz valamilyen JPA-implementációt.

Az alkalmazások vagy komponensek fejlesztésekor az entitásokat az EntityManager interfészén keresztül kezeljük. Minden EntityManager-példány entitások egy olyan halmazával dolgozik, amelyben minden elsődleges kulccsal rendelkező perzisztens példányhoz egyetlen egyedi memóriabeli példány tartozik. Az ilyen entitáshalmazokat perzisztenciakontextusnak hívjuk. Java EE-környezetben egy JTA tranzakció gyakran több komponensen keresztül ível át, és ésszerű igény, hogy egy tranzakcióhoz ugyanaz a perzisztenciakontextus tartozzon. Erre megoldás lehetne, hogy a fejlesztő a tranzakció elején megszerzett EntityManager-referenciát metódus argumentumként továbbadja minden egyes, a tranzakacióban érintett meghívott komponensnek. Ezt a feladatot azonban a specifikáció a konténerre bízza, vagyis amikor egy EJB-komponens referenciát kér egy EntityManager-re, a konténer feladata, hogy az EntityManager-példányhoz hozzárendelje az aktuális JTA-tranzakcióhoz tartozó perzisztenciakontextust. Ezért az ily módon megszerzett EntityManager-t konténer által kezelt entitáskezelőnek hívjuk.

Ritkábban ugyan, de előfordulhat Enterprise-környezetben, hogy olyan perzisztenciakontextusra van szükségünk, amely ne legyen elérhető más EntityManager-példányok számára, még akkor sem, ha azok ugyanabban a tranzakcióban vesznek részt. Ilyenkor az EntityManagerFactory interfész segítségével kell ún. alkalmazás által kezelt

EntityManager-példányt szerezni. Java SE-környezetben csak ez a lehetőség áll rendelkezésre.

Egy perzisztenciaegység konfigurációs információk, entitások, az O-R leképezés módját definiáló annotációk vagy xml-metadatok, egy EntityManagerFactory és az általa létrehozott EntityManager-ek halmaza. Egy ilyen perzisztenciaegység fizikailag lefordított class-fájlokból és egy META-INF könyvtárban lévő persistence.xml nevű konfigurációs fájlból áll, amelyek például egy EJB jar-fájlban vagy egy webalkalmazást magába foglaló WAR-fájl WEB-INF/classes könyvtárában vagy WEB-INF/lib könyvtárában lévő jar-fájlban lehetnek összetömörítve. Egy persistence.xml fájlban több perzisztencia egységet is definiálhatunk, mindegyiket külön névvel, amely névre hivatkozva az adott perzisztenciaegységhez alkalmazásunkból EntittyManager(Factory)-t szerezhetünk. Fontos tudni, hogy míg az EntityManagerFactory osztály szálbiztos, addig az EntityManager nem, vagyis többszálú használat esetén (ami szerencsére EJB környezetben nem fordulhat elő) a fejlesztőnek kell gondoskodnia a szinkronizációról.

Entitások életciklusa

Egy entitás mindig az alábbi négy állapot valamelyikében található: új, menedzselt, törölt, lecsatolt. Amikor a new operátorral példányosítunk egy entitás, az új állapotba kerül. Ekkor az objektum csak a memóriában jön létre, még nem létezik a neki megfelelő perzisztens adat, így az ilyen állapotú objektumon végrehajtott módosítások nem íródnak be az adatbázisba. Az EntityManager.persist() hívással menedzselt állapotba vihetjük az entitást, ami lényegében azt jelenti, hogy bekerül az EntityManager-példányhoz tartozó perzisztenciakontextusba. Ez szemléletesen úgy képzelhető el, hogy az EntityManager karban tart egy listát, amelyben elhelyezi minden menedzselt, azaz perzisztenciakontextusban lévő entitás referenciáját. A perzisztenciakontextusban lévő entitások legfontosabb jellemzője, hogy a tranzakciók végén a memóriabeli tartalom automatikusan szinkronizálódik az adatbázissal. Ha nem akarjuk megvárni a tranzakció végét, akkor az EntityManager.flush() metódussal kényszeríthetjük ki a teljes perzisztenciakontextus visszaírását az adatbázisba. Fordított irányban – az adatbázisból a memóriabeli objektumba – az EntityManager.refresh() metódussal lehet kérni egy menedzselt állapotban lévő entitás frissítését. A perzisztenciakontextus megszűnésekor az addig menedzselt entitások lecsatolt állapotba kerülnek. Lecsatolt állapotban, bár a nekik megfelelő egy vagy több perzisztens sor létezik, az entitásokon végrehajtott módosítások nem

kerülnek be az adatbázisba. Ilyenkor valójában hagyományos Java-objektumként viselkednek, és ebben az állapotban használhatók különböző szoftverrétegek között Data Transfer Objectként. Előfordulhat például, hogy egy lecsatolt entitás a webrétegbe kerül, ahol módosításokat hajtanak végre rajta. Ha ezt vissza akarjuk írni az adatbázisba, ismét menedzselt állapotba kell vinni, amit az EntityManeger.merge() metódus segítségével tehetünk meg, melynek bemenő paramétere a lecsatolt entitás, visszatérési értéke pedig a menedzselt entitás. Vagyis a lecsatolt példány nem válik menedzseltté. A negyedik állapot a törölt, ahova csak menedzselt állapotból kerülhet az objektum, az EntityManager.remove() meghívásának hatására. Ilyenkor, bár még a perzisztenciakontextushoz tartozik, már ki van jelölve törlésre, vagyis a tranzakció jóváhagyásakor törlődni fog az adatbázisból. Ha mégsem akarja törölni az EntityManager.persist()-tel visszavihető menedzselt állapotba.

Egy fontos kérdés nyitva maradt: mikor szűnik meg, és mikor jön létre a perzisztenciakontextus? A válasz nem is olyan egyszerű, figyelembe véve, hogy a perzisztenciakontextusnak kétféle ún. hatóköre lehet: tranzakció hatóköre, vagy kibővített hatóköre. Konténer által kezelt EntityManager esetén a függőséginjektáláshoz használt @PersistenceContext annotáció type paraméterében a PersistenceContextType.TRANSACTION, illetve PersistenceContextType.EXTENDED értékkel adható meg a hatókör, melyek közül az előbbi az alapértelmezett. Tranzakció hatókör esetén, ha egy tipikusan egy session bean által kezdeményezett JTA-tranzakció már aktív, de még nincs hozzá rendelve perzisztenciakontextus, és egy beinjektált EntityManager bármely műveletét meghívjuk, akkor létre fog jönni egy perzisztenciakontextus, amely hozzákapcsolódik az adott tranzakcióhoz. A tranzakcióban részt vevő összes további komponensbe beinjektált, vagy ott megkeresett EntityManager-hez ugyanezt a perzisztenciakontextust fogja hozzárendelni a konténer. A tranzkació végén, akár jóváhagyás, akár visszagörgetés a kimenetele, a kontextus megszűnik, az entitások pedig lecsatolódnak. Előfordulhat az is, hogy egy EntityManager meghívásának pillanatában még nem indult JTA tranzakció. Ilyenkor csak az adott hívás idejére jön létre egy perzisztenciakontextus, ami a hívás után azonnal meg is szűnik. Kibővített hatókörű perzisztenciakontextus csak állapottal rendelkező session beanbe injektált vagy ott megkeresett EntityManager-hez kapcsolódhat. Ennek élettartama az injektálás vagy megkeresés pillanatában kezdődik, és több tranzakciót is túlélhet, ugyanis a session bean megsemmisítésekor (@Remove metódus) szűnik meg.

Az entitások a következő annotációkkal megjelölt callback metódusok segítségével kérhetnek értesítést az EntityManager-től a rájuk vonatkozó életciklus-eseményről: @PrePersist, @PostPersist, @PreRemove, @PostRemove, @PreUpdate, @PostUpdate, @PostLoad. Ezek a metódusok akár egy külön osztályban is elhelyezhetők, amelyet a @EntityListener annotációval köthetünk az entitásokhoz.

Entitások keresése

A JPA a 2.1-es entity beaneknél jóval több lehetőséget kínál az entitások lekérdezésére, amit az EntityManager interfészen keresztül lehet megtenni. A legegyszerűbb lekérdezés típus, amikor elsődleges kulcs alapján találunk meg egy entitást. Erre az alábbi metódus szolgál:

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

A szignatúra kihasználja a Java 5 által bevezetett generikusok fogalmát. Ez lehetővé teszi, hogy a C++ template-ekhez hasonlóan típussal paraméterezzünk osztályokat vagy metódusokat. A find metódus így az elsődleges kulcs mellett egy T típussal paraméterezett Class generikus típust vár. A visszatérési érték pedig éppen T típusú lesz, vagyis egy olyan metódussal állunk szemben, amely bármilyen típusú entitással visszatérhet, anélkül, hogy a visszatérési értéket castolni kéne.

A bonyolultabb lekérdezések elvégzésére az EntityManager különféle metódusokat kínál, melyek mindegyike a Query interfészt megvalósító objektummal tér vissza. A lekérdezések megfogalmazhatók EJB-QL és natív SQL nyelven is. Ez utóbbira akkor lehet szükség, ha olyan adatbázis specifikus lehetőségeket akarunk a lekérdezésben kihasználni, amit az EJB-QL nem támogat. A JPA jelentősen kibővíti az EJB-QL lehetőségeit, amelyet így már Java Persistence query language-nek neveznek. A bővítések röviden összefoglalva:

- Többes törlés és módosítás, amelyek segítségével úgy törölhetünk vagy módosíthatunk egyszerre több entitást, hogy nem kell őket először egy lekérdezésben megtalálni, majd egyesével végigiterálva rajtuk elvégezni a törlést vagy módosítást.
- JOIN, GROUP BY, HAVING, allekérdezések támogatása.
- A bemenő paraméterek helyét ?1, ?2, stb. helyett :paraméterNév1, :paraméterNév2 stb. alakban definiálhatjuk a lekérdezésben.

- Projekció támogatása, vagyis lehetséges az entitásnak csak bizonyos attribútumait lekérdezni Object[] formában.
- Megoldható, hogy a visszakapott oszlopokat közvetlenül valamilyen általunk definiált objektumként kapjuk vissza, új objektumok létrehozásával a SELECT-ben.
- Lekérdezések dinamikusan, futási időben is létrehozhatók.

3.3.3. Entitások öröklése

A 2.1-es entity beanek komoly hiányossága volt, hogy nem követelték meg az öröklés támogatását. Így az objektumorientáltságnak ez az igen erőteljes eszköze csak esetleges alkalmazásszerver-specifikus kiterjesztések formájában állt rendelkezésre. Ezzel szemben az EJB 3 entitásaival tetszőleges öröklési hierarchiákat valósíthatunk meg hordozható módon. Az entitások öröklése POJO jellegükből következően egyszerűen az extends kulcsszóval történik, a leszármazott osztály példányai pedig a hagyományos öröklésfogalomnak megfelelően rendelkezni fognak az ősosztályban definiált attribútumokkal és metódusokkal. A hagyományos örökléshez képest mindössze az öröklési hierarchia relációsadatbázis-szintre történő leképezése jelent újdonságot. Ezt a feladatot természetesen a perzisztencia provider látja el, különböző stratégiák alapján, melyek közül megfelelő annotációk alkalmazásával választhat a fejlesztő.

Az első választható stratégia neve: Egy Tábla Egy Osztályhierarchiához. Mint a neve is mutatja, ilyenkor minden gyermekosztálypéldány egyetlentáblában tárolódik. Ennek a táblának természetesen tartalmaznia kell minden egyes gyermekosztályban definiált perzisztens attribútumnak megfelelő oszlopot. A típusinformáció ekkor nem olvasható ki ennek az egyetlen táblának a nevéből, ezért egy ún. diszkriminátor oszlopban tárolt érték írja le a típust. Ezen leképezés stratégiai hátrányai: a megoldás sok osztályt tartalmazó hierarchia esetén sok oszlopot eredményez, melyeknek nullázhatónak kell lenniük, hogy azokat a gyermekosztályokat is tárolhassuk a táblában, amelyekre az adott oszlop nem értelmezhető. A megoldás előnyeként említhető, hogy hatékony működést eredményez, mert az egyetlen tábla miatt nem szükséges a join művelet, ugyanakkor a típusinformációt megőrzi, miközben a polimorfizmus támogatása is könnyen megvalósítható. A JPA előírja e stratégia támogatását.

A második lehetséges stratégia az öröklés leképezésére a Külön Tábla Gyermekosztályonként. Mint a neve is mutatja, ennél a megoldásnál minden gyermekosztályhoz és hierarchia csúcsán

álló ősosztályhoz külön tábla tartozik. Minden tábla csak az adott gyermek által definiált új oszlopokat tartalmazza, ezenkívül az elsődleges kulcs egyben idegen kulcsként mutat az ősosztályhoz tartozó táblára. Megoldás hátránya: mély öröklési hierarchia esetén a sok join rossz teljesítményhez vezethet. Előnye viszont, hogy nincsenek fölösleges oszlopok a táblákban, és így akár nem nullázható oszlopok is definiálhatók. A polimorfizmust a korábbi megoldáshoz hasonlóan szintén támogatja. Fontos megjegyezni, hogy egy öröklési fán belül, csak a legfelső szintű ősosztályban definiálhatjuk a leképezési stratégiát, de ez a teljes hierarchiára érvényes lesz.

A harmadik stratégia neve Egy Tábla Egy Konkrét Gyermekosztályhoz. Ebben az esetben minden gyermekosztály külön táblában tárolódik, de a táblák az összes ősben definiált attribútumoknak megfelelő oszlopokat is tartalmazzák. Ennek a megoldásnak a jellegzetessége, hogy az egy táblás megoldáshoz hasonlóan hatékony, mert kiküszöböli a joinokat, ugyanakkor alkalmazása nem zárja ki a nem nullázható oszlopok definiálását. Nehézkes megoldani vele a polimorfizmus támogatását, ezért a JPA nem is követeli meg ezt. Nem minden perzisztencia provider lesz képes kezelni az ilyen entitásokat.

Eddig azt az esetet tárgyaltuk, mikor entitás származott entitásból. A specifikáció szerint azonban megengedett az is, hogy nementitásosztályok kerüljenek bele az öröklési fába. Nementitásosztály minden további nélkül származhat entitásosztályból, az örökölt attribútumok azonban természetesen nem lesznek perzisztensek. A másik irány, vagyis hogy nementitásosztályból származzon egy entitás, kicsit összetettebb. Ilyenkor az entitás örökli a nementitás ős nemprivát viselkedését és állapotát, az utóbbi azonban nem fog hozzátartozni az entitás perzisztens állapotához, vagyis nem fog adatbázisban tárolódni. Szintén fontos, hogy nementitásosztályt nem adhatunk át sem EntityManager sem Query interfész egyik metódusának sem. Lehetőség van arra, hogy egy nementitásosztályt a @MappedSuperclass annotációval megjelöljünk, és abból származtassunk entitásosztályokat. Ezzel azt érjük el, hogy a nementitásban definiált attribútumok mégis részei lesznek a leszármazott entitások perzisztens állapotának, viszont magához a @MappedSuperclass ősosztályhoz nem fog létrejönni külön tábla, és az továbbra sem lehet az EntityManager vagy a Query interfész metódusainak bemenő paraméterei. Végül az entitások öröklésének külön alesetét jelenti, amikor absztraktnak deklarálunk egy entitást, és abból származtatunk más entitásokat. Ilyenkor az absztrakt entitás nem példányosítható, viszont szerepelhet lekérdezésekben,

melynek visszatérési értékében természetesen az absztrakt entitásból leszármazott konkrét osztályok fognak szerepelni.

3.3.4. Kapcsolatok entitások között

A perzisztens entitások közötti kapcsolatok O-R leképezése, csakúgy, mint 2.1-es entity beanek esetén, JPA használatával is megoldható. Itt természetesen akár annotációk alkalmazásával is megtehetjük mindezt. A kardinalitásnak megfelelően a @OneToOne, @OneToMany, @ManyToOne, @ManyToMany annotációk közül választhatunk, melyeket a kapcsolat másik végét reprezentáló tagváltozóra vagy metódus alapú elérés esetén a hozzá tartozó getter metódusra kell alkalmazni. A kapcsolat lehet kétirányú vagy egyirányú, ez utóbbi esetben csak az egyik entitásból lehet a másikat oldalt elérni, fordítva nem. Egyirányú kapcsolat esetén a tulajdonos oldal egyértelmű, kétirányú esetben a tulajdonos oldal és az inverz oldal meghatározására az alábbi szabályok érvényesek:

- Egy-egy kapcsolatban az az oldal a tulajdonos, amelyik az idegen kulcsot tartalmazza.
- Egy-több kapcsolatban a több oldal a tulajdonos.
- Több-több kapcsolatban bármelyik oldal lehet tulajdonos. (Itt egy harmadik kapcsoló tábla tartalmazza a kulcsokat)

A tulajdonos és az inverz oldal meghatározása a kétirányú kapcsolatok összerendelésénél kap szerepet. Erre azért van szükség, mert két entitás között több kapcsolat is elképzelhető. A kétirányú kapcsolat inverz oldala köteles egy mappedBy attribútumban hivatkozni a tulajdonos oldali attribútumra, különben a perzisztencia provider egy egyirányú kapcsolat tulajdonosának fogja tekinteni.

A kapcsolatokat relációs táblákra idegen kulcsok segítségével lehet leképezni. Egy-egy kapcsolat esetén bármelyik oldalra elhelyezhető az idegen kulcs, egy-több kapcsolatnál a több oldalt tároló tábla tartalmazza az idegen kulcsot. Több-több kapcsolatnál pedig egy harmadik, ún. kapcsoló tábla szükséges, amely a kapcsolat mindkét oldalára mutató idegen kulcsokat tartalmazza. Az idegenkulcs-oszlopok nevét @JoinColumn, illetve a kapcsoló táblát a @JoinTable annotációban lehet definiálni, de használatuk akár mellőzhető is, ilyenkor a perzisztencia provider az entitás-, illetve attribútumnevekből származtatott alapértelmezett értékeket használja. Egy fontos különbség a 2.1-es entity beanekhez képest a kapcsolat

konzisztens állapotban tartásának felelőssége. Entity beanek esetén ezt a konténer biztosította, vagyis elég volt például egy egy-több kapcsolat több oldalán átállítani a kapcsolódó objektumot, a korábbi kapcsolódó objektumhoz tartozó Collectionből a konténer kivette, illetve az új kapcsolódó entitás gyűjteménybe berakta a módosított entitást. JPA-entitásoknál ezzel szemben az alkalmazás, vagyis a fejlesztő feladata a konzisztens kapcsolatkezelés, tehát a kapcsolatmódosításban érintett összes entitást módosítani kell, ellenkező esetben nem definiált a perzisztencia provider működése.

A CMP entity beaneknél is definiálható volt a kaszkádolt törlés a telepítésleíróban, mellyel egy-egy vagy egy-több kapcsolat esetén megoldható volt, hogy az egy oldalon álló entitás törlése esetén a kapcsolódó egy vagy több entitás is automatikusan törlődjön. JPA-entitásokra a kaszkádosítás nem csak törlésre adható meg, hanem az entitás életciklusainak három másik eseményre is: adatbázisból történő frissítés, perzisztenciakontextusba történő első bekerülés vagy oda lecsatolt állapotból való visszakerülés esetére is.

4. JavaServer Faces

Ez az egyik leginkább elterjedt webes keretrendszer. A JSP technológiára épült. Fejlesztő környezetekkel gyorsan és egyszerűen készíthetünk tetszetős felületet, még az üzleti logika nélkül. Ezek Drag-and-Drop jellegű szerkesztők.

4.1. Webes keretrendszerek

Az ideális webes keretrendszer célja, hogy elegánsan kapcsolja össze az üzleti logikát a grafikus felülettel. Egy webes fejlesztés során a következőket kell átgondolni.

4.1.1. Konverzió

Egy HTML felületen minden adat String típusú. Az üzleti logika viszont várhat lebegőpontos számot, vagy dátumot. Ekkor szükségünk van a konverterekre. Ezeket olyan sűrűn kell használni, hogy elvárható az újrafelhasználhatóságuk, így elég őket egyszer megírni. Sok keretrendszerbe beépített konverterek is vannak.

4.1.2. Validáció

A konverzió után szükséges ellenőrizni, hogy szemantikusan is helyes-e a szintaktikusan már ellenőrzött érték. Például egy kölcsönző alkalmazás során nem lehet negatív mennyiségű filmet kiadni. Ezeket az adatokat 5 részre oszthatjuk:

- Nem igényel ellenőrzést
- Igényel ellenőrzést és az adat az üzleti folyamatból következik
- Igényel ellenőrzést és hamis adatok kiszűrésére szolgál
- Belső összefüggés miatt igényel ellenőrzést.
- Egy másik mező értékével összehasonlítva igényel ellenőrzést.

Összefoglalva, tehát ideális esetben támogatja a mező és több mező alapú validációt a keretrendszer. Itt is követelmény az újrafelhasználhatóság.

4.1.3. Oldalak közötti újrafelhasználhatóság

Az oldalakon általában vannak olyan elemek, amiket többször felhasználhatunk. Célszerű, hogy ezeket ne keljen minden oldalon újra megírni, hanem csak hivatkozhassunk rájuk. Ilyenek például a menü, fejléc, lábléc.

4.1.4. Komponensek támogatása

Gyakran van arra is szükség, hogy az oldalt újrafelhasználó elemekből építsük fel, ezek a komponensek. Előnyös például, ha egy hivatkozással tudunk egy jelszó mezőt és jelszóellenőrző mezőt elhelyezni az oldalon.

4.1.5. Okos komponensek

A modern komponens alapú fejlesztés esetén már nem csak a kinézet alapján, hanem viselkedés alapján is lehetséges az újrafelhasználás. Így akár olyan komponensek is létrehozhatók, amik azt is tudják, hogyan kell egy adatot az adatbázisba menteni vagy éppen lekérdezni.

4.1.6. Navigáció

A felhasználó oldalak során halad végig a felület használat során. Fontos, hogy a keretrendszer adjon valamilyen támogatást arra, hogy a felhasználót egyik oldalról a másikra irányítsuk. Ez történhet programozottan és deklaratívan is.

4.1.7. Nemzetköziesítés támogatása

A többnyelvű honalapok legegyszerűbb megvalósítás, hogy az összes szöveget kigyűjtjük egy file-ba és a kódban csak hivatkozunk egy logikai azonosítóval. Gyakran más teendő is akad:

- A felületen esetleg más sorrendben kell megjeleníteni a mezőket.
- Bizonyos mezőket nem kell megjeleníteni.
- Eltérő mértékegységek használata miatt máshogy kell bekérni és kiírni az adatokat.
- Az országra jellemző konvertereket validátorokat külön kell implementálni.

4.1.8. Állapot elmentése

A HTTP-protokol állapotmentes. Azonban gyakran van szükség, hogy egy állapottal rendelkező beszélgetést indítsunk. Ilyen mikor bejelentkezünk egy oldalra és utána az oldal a bejelentkezés ismeretében engedélyezi a hozzáférésünket. Egy másik eshetőség lehet a virtuális bevásárló kocsi, amikor a termékek közt ki akarunk valamit választani, akkor elvárható, hogy addig ne ürüljön ki a már megtöltött kocsi. Így ahhoz kerül hozzáadásra az új termék.

4.1.9. Nézet és viselkedés szétválasztása

Már egy kisebb fejlesztés esetén is szükséges a szerepköröket szétválasztani, ezek:

- Üzletilogika-fejlesztő
- Komponensfejlesztő
- Weboldalfejlesztő
- Grafikus

Fontos, hogy ezek elkülönüljenek egymástól és lehetőleg ne keveredjenek.

4.1.10. Biztonság

Egy webes keretrendszer általában szűri az input és output adatokat is. Két gyakori probléma:

- Egy input mező értékét közvetlen felhasználjuk egy SQL parancsban, amit módosíthat egy támadó.
- Másik pedig, hogy egy hozzászólásban javascript kódot küldenek. Ezt elkezdi minden böngésző futtatni, ami az oldalt megtekinti.

4.1.11. Elterjedt webes technológiák használata

CSS komponensek esetén felmerül az igény, hogy lehessen minden oldalon egységes kinézetet megadni, így lehessen megadni „style” és „class” attribútumoknak értéket. A keretrendszer tegyen lehetővé javascript beszúrását. Ez a JSF-nél nincs megoldva, így minden komponens előfordulásnál beszúródik nem csak egyszer. Aszinkron javascript alapú HTTP-kérések elterjedésével felmerült az igény, hogy a keretrendszer támogassa ezeket. Ez az AJAX.

4.1.12. Eszköztámogatottság

Fontos, hogy a Java fejlesztőrendszerek támogassák a webes keretrendszert. Ez egyrészt a Drap-and-Drop felület elkészítésének támogatását, másrészt a konfigurációs XML fájlok sémájának ismeretét. JSF estén az oldalak közötti navigációt is grafikusan állíthatjuk be.

4.1.13. Bővíthetőség

A Sun úgy tervezte a JSF-et, hogy lehessen hozzá komponenskönyvtárakat írni, ezek valósítják meg a bővíthetőséget. Ezek újrafelhasználhatók.

4.2. JSF technológia

A JavaServer Faces specifikál egy komponens orientált keretrendszert, mely része a Java EE platformjának. A JSF egy specifikáció melynek számos implementációja létezik. A specifikációnak két elterjedt fajtája van használatban, azonban a régebbit célszerű lecserélni az új 1.2-es szabványra, mely számos újítást tartalmaz.

4.2.1. A Model-View-Controller architekturális minta

A leggyakrabban alkalmazott minta JSF nélkül architekturális tervezésre az MVC. A megvalósításban Modell alatt egy vagy több osztályt értünk, melyek a valóság egy darabját írják le. A megjelenítést rendezett formában a Nézet végzi. Ez képes megtervezett GUI-sémában megjeleníteni. A megjelenítés alatt a felhasználó és kiszolgáló párbeszédét a vezérlő irányítja.

HTTP alapú technológia esetén ez a következőképpen néz ki:

1. A böngészőből érkezik a kérés, hogy melyik oldalt milyen paraméterekkel szeretnénk megtekinteni más kiegészítő információkkal.
2. A kérést feldolgozza a Vezérlő, amennyiben több van, úgy az URL és paraméterek dekódolása után a megfelelő Vezérlőhöz továbbítódik.
3. A Vezérlő feldolgozza a paramétereket és előállítja a modellt.
4. A Nézet a modell alapján előállítja a felületet, és továbbítja a böngészőnek.
5. A felhasználó értelmezi az adatokat, majd új kérést küld.

A JSF esetén a szervletkonténer web.xml állományát úgy kell beállítani, hogy minden JSF-hez érkező kérés esetén a konténer a FacesServlet nevű szervletnek adja át a vezérlést. A FacesServlet ezután egy állapotgépet valósít meg, majd folytatja a feldolgozást. A Modell és Vezérlő szerepét JSF esetén menedzselt JavaBeanek játsszák. Nem tévesztendő össze az Enterprise JavaBean osztályokkal. Ezek egyszerű beanek. Egy JavaBean akkor lesz menedzselt, ha ezt külön jelezzük a keretrendszernek a konfigurációs állományok segítségével. Ezen beanek életciklusát a JSF keretrendszer fogja vezérelni. A Bean osztályok két feladata van: a nézetet adatokkal kell ellátnia. Ha a felhasználó adatokat küld a weboldalon keresztül, akkor a feldolgozás után szintén modellbe kell visszaíródniuk. Másik feladata: hogy olyan publikus metódusokat nyújtsanak a keretrendszernek, mellyel a keretrendszer viselkedését lehet befolyásolni.

4.2.2. A Nézet

Amikor létrehozunk egy JSF-tageket használó .jsp fájlt, akkor ezt a konténer egy szervletté fordítja. Amikor lefut a szervlet service mótudusa, akkor a JSF-tagek a memóriában felépítenek egy fa struktúrát. Ez a View, azaz a Nézet. A gyökér elemből bejárható az egész

fa. Ezt ViewRoot-nak nevezzük. Minden egyes JSF taghez tartozik egy objektum, mely attól függ, hogy milyen JSF elemhez tartozik.

Néhány megjegyzés:

- A StateHolder interfészt implementálja a UIComponentBase, amiből származtatott az összes többi komponens.
- A ViewRoot gyökerében mindenképpen UIViewRoot típusú objektum található. Ez az osztály rendelkezik az infrasturkturális szerepkörrel. Minden ViewId-hez tartozik egy ViewRoot példány. Ez a nézeteket azonosítja egy felhasználóhoz.
- Az UIForm osztály a h:form taghez tartozik. Amikor kérés érkezik a JSF-hez akkor ezen objektumok submitted boolean típusú tagváltozójának értéke beállításra kerül. Ez azért fontos, mert ha egy h:form false értékkel rendelkezi, akkor nem kell feldolgozni a gyermekeit.
- Az UICommand osztály akkor használható, ha olyan GUI elemeket megvalósító komponenseink vannak, amelyekkel a felhasználó parancsokat adhat, funkciókat aktiválhat. Amikor a dekódoló logika észreveszi a kérést, akkor meghívja az összes ActionListener interfészt megvalósító osztályt.
- UIMessage és UIMasseges osztályok olyan üzeneteket képesek megjeleníteni, amelyek kérésfeldolgozás közben keletkeztek. Az üzenet megjelenítése vagy a célmező mellett (UIMessage), vagy az oldal alján, tetején szokott történni (UIMessages).
- Az UIOutput osztály az értékek megjelenítésére szolgál. Az értékeknek String típusúnak kell lennie. Amennyiben nem String, akkor megpróbálja azt konverterrel átalakítani.
- Az UIInput osztály az UIOutput leszármazottja. Ez már nem csak megjelenítésre alkalmas, hanem adatok elmentésére a modellbe. Ehhez validátorokat és konvertereket használ.

A JSF-et úgy specifikálták, hogy ne csak HTML környezetben működjön, ezért az osztály hierarchia és a konkrét osztály között lenni kell egy megfeleltetésnek, ez a Renderkit. A leginkább használt a HTML Renderkit.

4.2.3. Kérésfeldolgozás

A JSF minden kérésfeldolgzásánál egy állapotgép vezérli a folyamatot. Hat fő állapot van. Az állapotok között történik az eseményfeldolgozás. Ekkor a komponensek tovább küldik a hozzájuk érkező eseményeket a regisztrált, FacesListener intefészt implementáló osztályoknak. A feldolgozási folyamatot megszakítására is itt van lehetőség. A folyamat több ok miatt is megszakadhat, pl: ha az oldal először kér egy felhasználót, mert ekkor még nincsenek paraméterei. Lehet még, hogy a konverter, vagy validátor hibát észlel, ekkor jelez a JSF-nek, hogy szakítsa meg a folyamatot.

4.2.3.1. A nézet visszaállítása

A keretrendszer eldönti, hogy olyan kérés történt-e, amely már meglévő állapot visszakérésére hivatkozik. Ez a postback. Ez legtöbbször azt jelenti, hogy a kérés arra a JSF oldalra irányul, mint az előző kérés. A klasszikus példa regisztrációnál az ország kiválasztása után a városok kilistázása. A keretrendszer eldönti, hogy postback kérés-e:

- Ha igen, akkor a JSF visszaállítja a komponensek állapotát, amik vagy a szerveroldalon vagy a kliens oldalon tárolódnak. Előbbi esetben session utóbbiban rejtett űrlapváltozókat használ a keretrendszer.
- Ha nem, akkor létrehozza a ViewRoot-ot, ami a komponensfa gyökere, majd az állapotgépet „Nézet generálása” állapotba helyezi.

4.2.3.2. A kérés paramétereinek rögzítése

Itt minden input komponens ellenőrzi, hogy van-e hozzárendelt paraméter, ha igen azt a UIInput ősosztály submittedValue változójában tárolja. A modellbe történő átadás előtt konvertálás és validálás történik ezzel az értékkel. Ha a komponens rendelése le van tiltva, akkor nem keres paramétert. Ha pedig be van kapcsolva az immediate attribútum, akkor megtörténik a konvertálás és validálás is a submittedValue beállításával egy időben.

4.2.3.3. Validációk futtatása

A komponensfa mélységi bejárásával történik a komponensek validásása a ViewRoot-ból indulva. Ebben a fázisban validálni kell azokat a komponenseket, melyekre igaz, hogy

- a felhasználó által küldött űrlapban szerepelnek,
- a kérésben található hozzájuk paraméter,

- a rendered attribútum true,
- az immediate attribútum false.

Ha mind teljesül, akkor elkezdődhet a validálás. Először a kérésben érkezett paramétert kell konvertálni a megfelelő Java-típusra, egy JSF-konverterrel. Amennyiben a komponenshez nincs explicit konverter rendelve, akkor a keretrendszer az alkalmazásszintű konverterek között próbál megfelelőt találni. Ha ez megtörtént kezdődhet a validáció. Ekkor az alábbi történik:

- Ha a komponens required attribútuma igaz, viszont a kérésben a komponenshez tartozó paraméter üres karaktersorozat, akkor a keretrendszer hibát jelez, hiszen a felhasználó nem töltött ki egy kötelező mezőt. Ekkor a JSF hiba üzenetet helyez el a komponens üzenetei között, melyet akár a felhasználónak is megjeleníthetünk a h:message vagy h:messages taggel.
- Ha van hozzá tartozó paraméter, akkor megkezdődhet a validálás. Minden komponenshez több validátor is tarozhat így, mindet le kell sorban futtani. Akkor tekinthető validáltnak a komponens, ha mindegyik lefutott hiba nélkül. Amennyiben egy validátor sikertelenül fejezi be a működését, akkor a többi validátor nem kerül futtatásra a továbbiakban. Ekkor hiba üzenetet helyez el a keretrendszer.

Ha a validálás sikeresen véget ért, akkor a keretrendszer beírja a konvertált, validált értéket a komponens value tagváltozójába. Amennyiben ez az érték nem egyezik meg a value eredeti értékével, akkor azt esemény létrehozásával jelzi. Ha valamilyen feladatot csak érték változáskor akarunk elvégezni, akkor ezt az eseményt célszerű figyelni.

Amikor validációs hiba történik, akkor az állapotgép „Nézet generálása” állapotba ugrik, hiszen ilyenkor a hibás értéket nem szabad rávezetni a modellbe.

4.2.3.4. A Modell aktualizálása

Itt történik a konvertált, validált értékek felvezetése a modellbe. Ehhez a komponenshez tartozó JSF-tag value attribútumát használja fel a keretrendszer. Lenni-e kell megfelelő setter és getter metódusoknak a megfelelő menedzselt bean-nek.

4.2.3.5. Az alkalmazás meghívása

Ebben a fázisban a fázishoz tartozó események továbbítódnak a komponenseknek, melyet azok továbbítanak az összes regisztrált osztálynak. A gyakorlatban ez azt jelenti, hogy ebben a fázisban kerül meghívásra az összes UICommand-ból leszármazó komponens actionListener és action attribútumában található metódus. Amennyiben a komponens JSF-tagjében az immediate attribútum igaz, úgy az alkalmazás meghívása a második „A kérés paramétereinek rögzítése” fázisban történik és nem ebben.

4.2.3.6. A Nézet generálása

Ennek a fázisnak a feladata, hogy a felépített komponensfát bejárva, a beállított Renderkit segítségével generálja le a szükséges kimenetet. Ebbe a fázisba többféleképpen is kerülhet a feldolgozás a JSF futása alatt:

- Ha az első fázisban észlelte a rendszer, hogy a kérés nem postback típusú, akkor jelenleg csupán a gyökérelem áll a komponensfában. Pontosan ekkor van szükség arra a mechanizmusra, amikor a JSF-tageket tartalmazó .jsp fáljt mint szervletet lefuttatja a JSF keretrendszer, így felépítve a teljes komponensfát.
- Ha postback-et érzékelt a rendszer, akkor nem kell újra felépíteni a fát, hiszen az már tárolva van.

5. Elsődleges felhasználói kézikönyv

5.1. Áttekintés

A Lending egy DVD kölcsönző cég számára készített nyilvántartó program, melynek feladata a DVD-k kölcsönzésének naprakészen tartása, DVD filmek lefoglalása, filmek keresése és kislistázása, ügyfelek nyilvántartása.

A rendszert háromféle személy használhatja, az ügyfél, a munkatárs és a rendszeroperátor. Az ügyfél kereshet a filmek között és kilistázhatja őket. A munkatárs ügyfeleket vihet fel a rendszerbe vagy törölhet, a kölcsönzéseket bejegyezheti és lezárhatja, filmeket kereshet és listázhat, hiba esetén filmeket törölhet, új filmeket vihet fel. Az operátor vehet fel új munkatársakat, vagy régieket törölheti.

5.2. Általános követelmények

1. A kölcsönzés árát a rendszer a kölcsönözni kívánt filmek darabszáma és kivitt napjainak száma alapján határozza meg. A munkatárs viheti fel a kölcsönzés kezdetét és zárhatja azt le a visszahozott film esetén.

2. Amennyiben a film megsérül vagy elvész az ügyfél kártérítést köteles fizetni.

3. Új munkatárs esetén az operátor viheti fel a rendszerbe őket. Régi elbocsátása esetén pedig törölheti a rendszerből.

4. Új ügyfeleket a munkatárs vihet fel, régieket törölhet (pl.: kártérítés elmulasztása esetén).

5. Filmek előre történő lefoglalása munkatárs által az ügyfél részére.

6. Filmek felvitele, módosítás vagy törlése munkatárs által.

5.3. Rendszerkövetelmények

Hardver: az implementáció során nem kell figyelembe venni
Programozási nyelv: Java Enterprise Edition
Operációs rendszer: a webkiszolgálóra Linux és Glassfish v3 szükséges, az adatbázisszerverre szintén Linux, valamint Oracle-szerver, a kliensgépeknek csupán egy böngészővel kell rendelkezniük.
Felhasználók száma: a programot maximum a rendszeroperátor, 2 munkatárs és 10 ügyfél fogja azonos időben használni.

5.4. Fogalomtár

ügyfél:

Az a személy, aki elviszi a filmet, megnézi, majd visszahozza. Ezért ellenértéket fizet a cégnek.

munkatárs:

Az a személy, aki a cégnél dolgozik, kapcsolatot tart fenn az ügyfelekkel. Felviszi a kölcsönzés adatait, lezárja a kölcsönzést. Új filmeket, ügyfeleket vihet fel, vagy törölhet.

rendszeroperátor:

Az a személy aki, új munkatársakat vihet fel, vagy törölhet.

kölcsönzés:

A film elvitele, majd vissza hozatala a cégnek.

film:

Az a tárgy, amit az ügyfél ki akar kölcsönözni. DVD lemezen tárolt hétköznapi értelemben vett film.

sérült film:

A film valamilyen okból kifolyólag nézhetetlenné válik, így elvesztette értékét. Ezt az ügyfél köteles megtéríteni.

elveszett film:

A filmet az ügyfél nem tudja visszaszolgáltatni a cégnek, így kártérítést kell fizetnie.

5.5. Szakterületi fogalmak

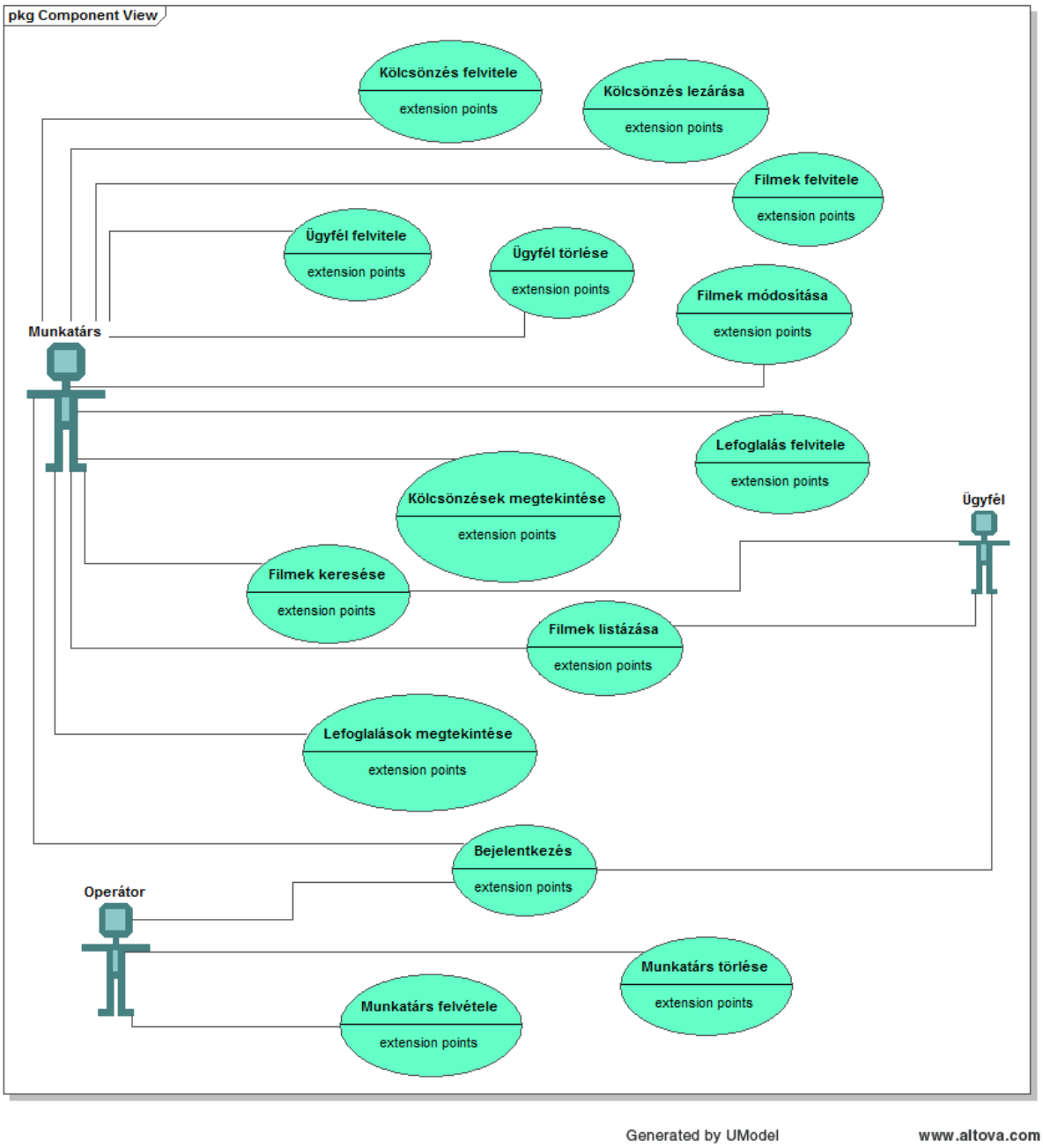
használatieset-diagram:

Célja a követelményrögzítés. Megmondja, hogy mit kell tudnia a rendszernek, és hogy milyen funkciói legyenek. Szemléletes, könnyen áttekinthető. A diagramon szereplő aktorok különböző célokkal kapcsolódnak a rendszerhez, eredményeket várnak a rendszertől.

aktivitásdiagram:

Az időben lezajló változásokat jeleníti meg.

5.6. Használatieset-diagram



5.7. Forgatókönyvek

(A használatieset-diagramok rövid, szöveges leírása)

Bejelentkezés:

A rendszer használatához a munkatársaknak és a rendszeroperátornak be kell jelentkezniük egy felhasználónév és egy jelszó használatával. Amennyiben a megadott felhasználónév

érvényes és a jelszó helyes, a munkatárs vagy a rendszeroperátor megkezdheti a program használatát. Az ügyfelek látogatóként jelentkezhetnek be, felhasználó név és jelszó nélkül.

Kölcsönzés felvitele:

A munkatárs egy űrlap segítségével rögzíti, hogy az ügyfél mely filmeket vitte el. A rendszer ekkor automatikusan csökkenti a bent lévő darabszámot. Ha nincs bent egy példány se a filmből, akkor nem kölcsönözhető ki.

Kölcsönzés lezárása:

A munkatárs felviszi a bejegyzett kölcsönzésbe, hogy visszahozták a filmet. Ezzel lezártnak tekinthető. A rendszer kiszámolja a fizetendő összeget az ár és kint lévő nap függvényében.

Kölcsönzések megtekintése:

A munkatárs kilistázhatja mely ügyfél mely filmeket vitte ki, vagy mely filmet melyik ügyfelek vitték ki.

Filmek felvitele:

A munkatárs új filmeket vihet fel.

Filmek módosítása:

A munkatárs módosíthatja a filmek adatait. Elveszett vagy tönkrement film esetén a darabszámot a munkatársnak kell csökkenteni.

Filmek keresése:

Az ügyfél és a munkatárs kereshet a film címe, rendezője és főszereplője szerint.

Filmek listázása:

Az ügyfél és a munkatárs kilistázhatja az összes filmet.

Ügyfél felvitele:

A munkatárs új ügyfelet vihet fel.

Ügyfél törlése:

A munkatárs ügyfelet törölhet.

Lefoglalás felvitele:

A munkatárs egy filmet jegyezhet elő egy ügyfélnek. Amennyiben a film minden példánya már előjegyzés alatt áll vagy ki vannak adva, akkor nem lehetséges az előjegyzés, csak a várható visszahozási idő után.

Lefoglalás megtekintése:

A munkatárs ki listázhatja az összes előjegyzést.

Munkatárs felvétele:

Az operátor új munkatársat jegyezhet be a rendszerbe. Ekkor felhasználó nevet és jelszavat is ad neki.

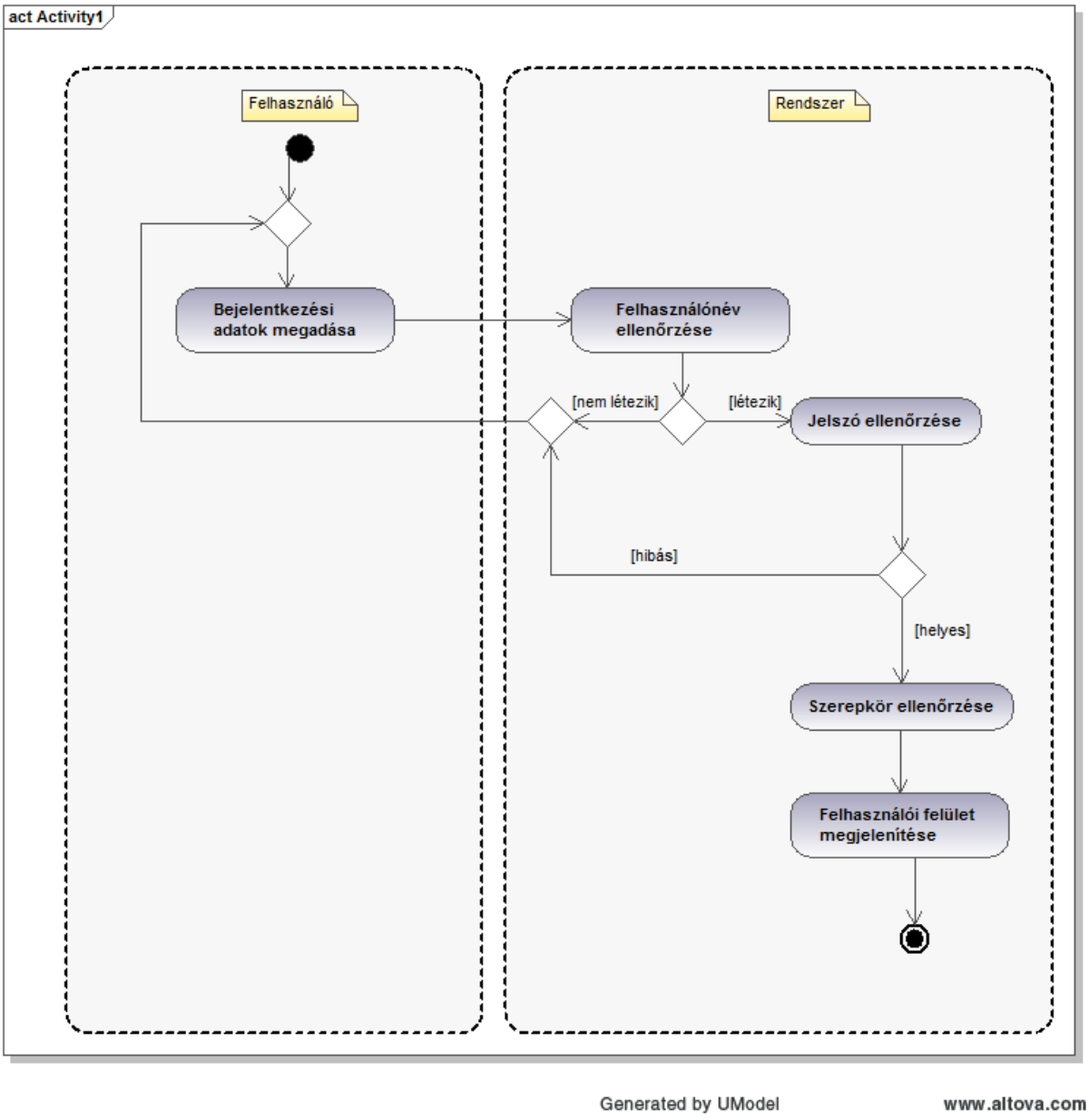
Munkatárs törlése:

Az operátor munkatársat törölhet a rendszerből munkaviszony megszűnése esetén.

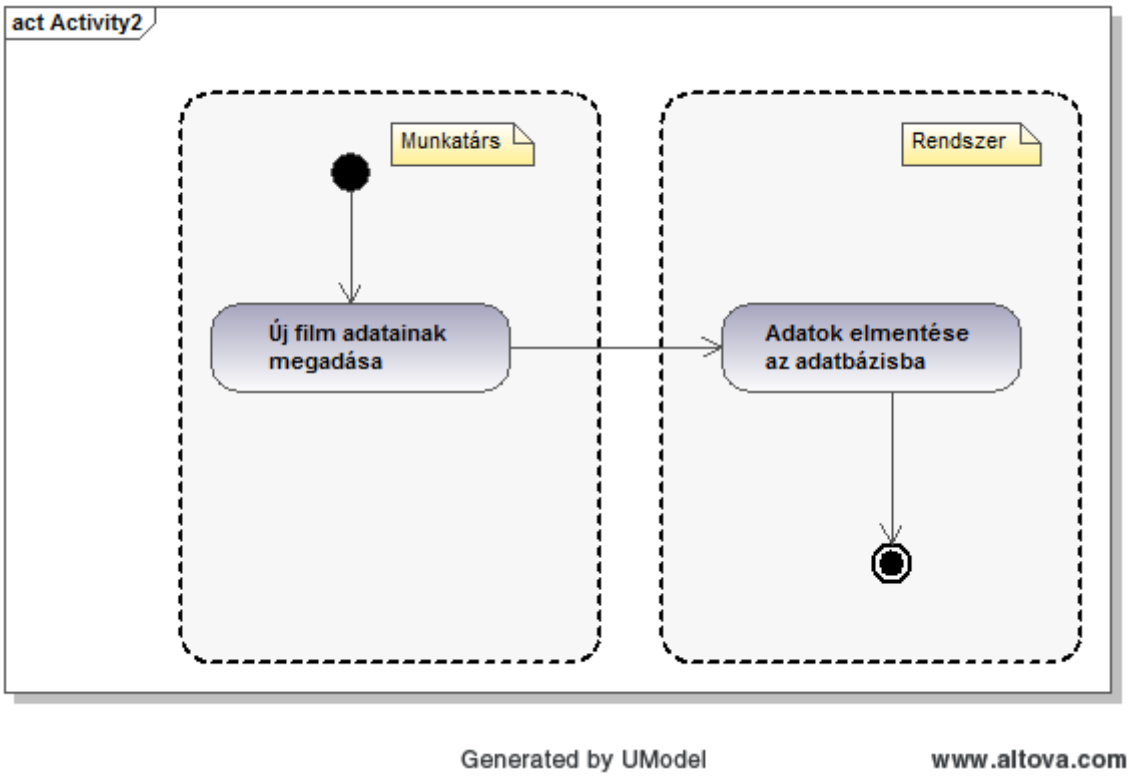
6. Szakterületi modellezés

6.1. Aktivitásdiagramok

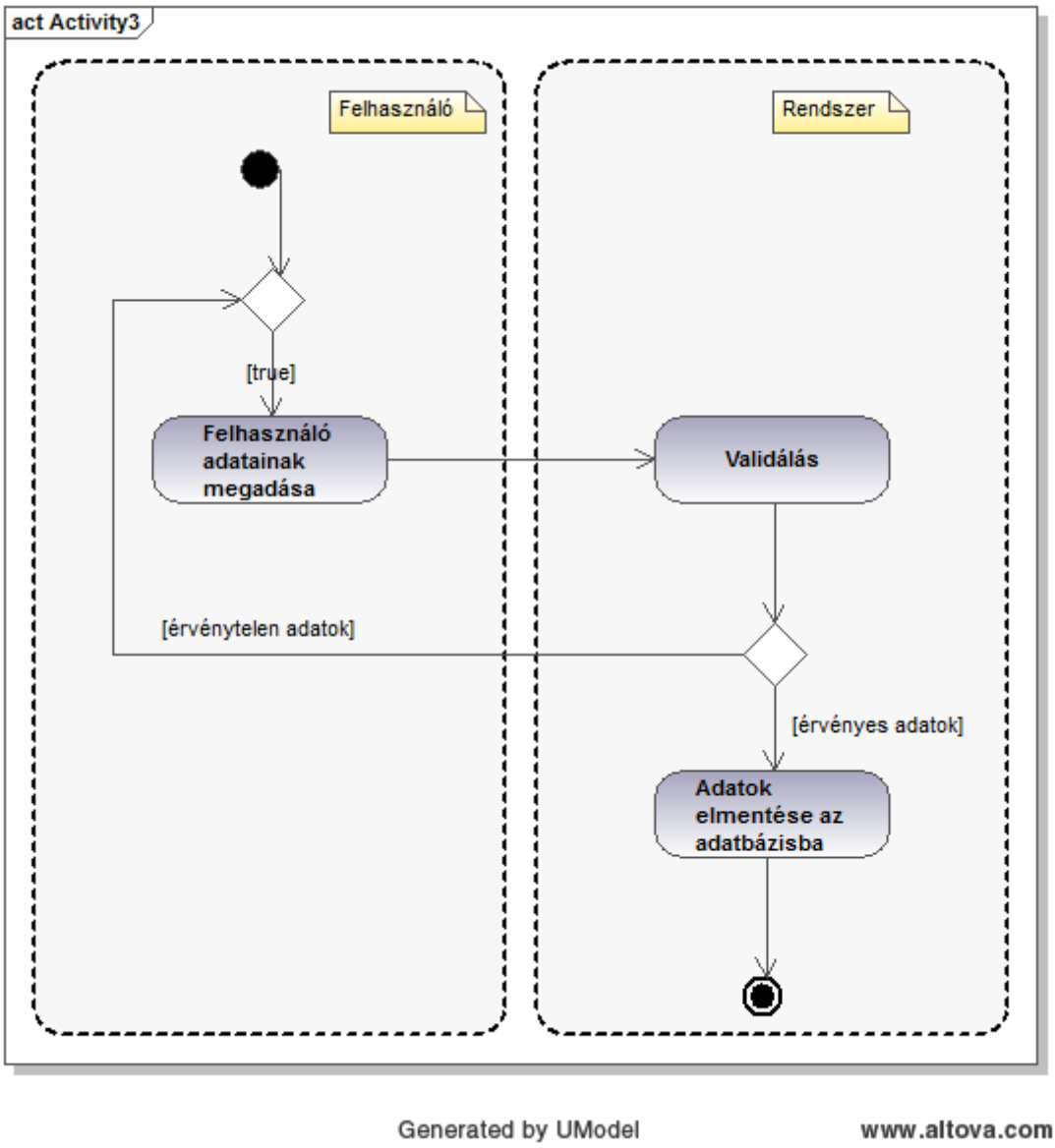
6.1.1. Bejelentkezés



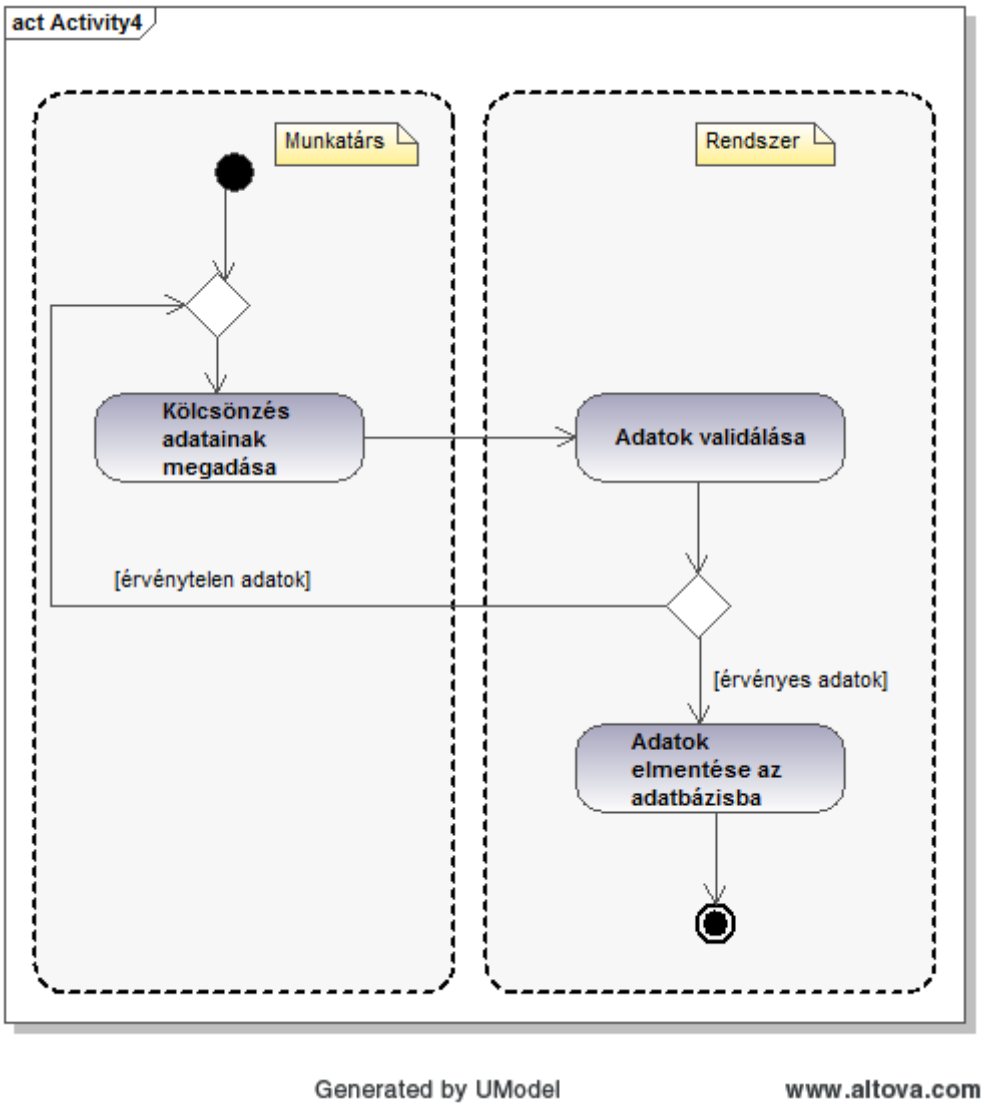
6.1.2. Új film felvitele



6.1.3. Új felhasználó felvitele

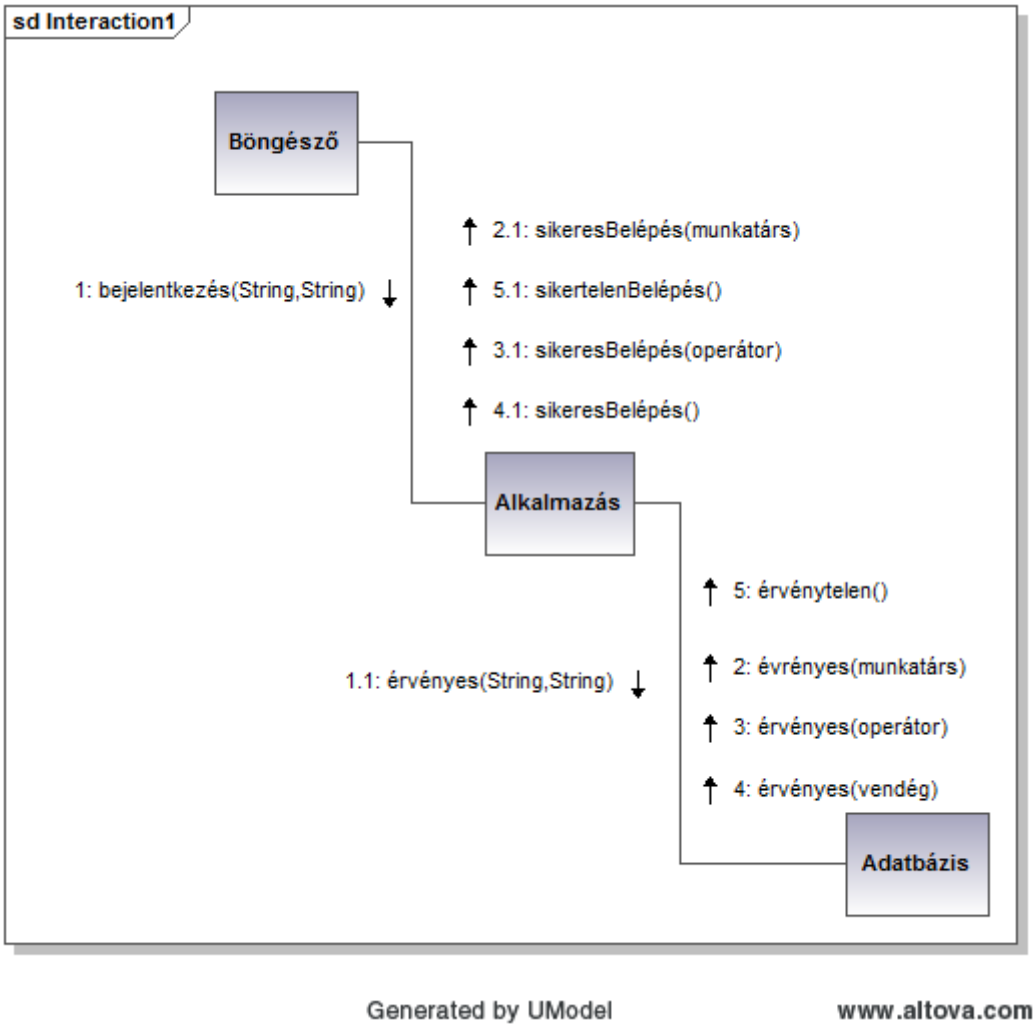


6.1.4. Új kölcsönzés felvitele



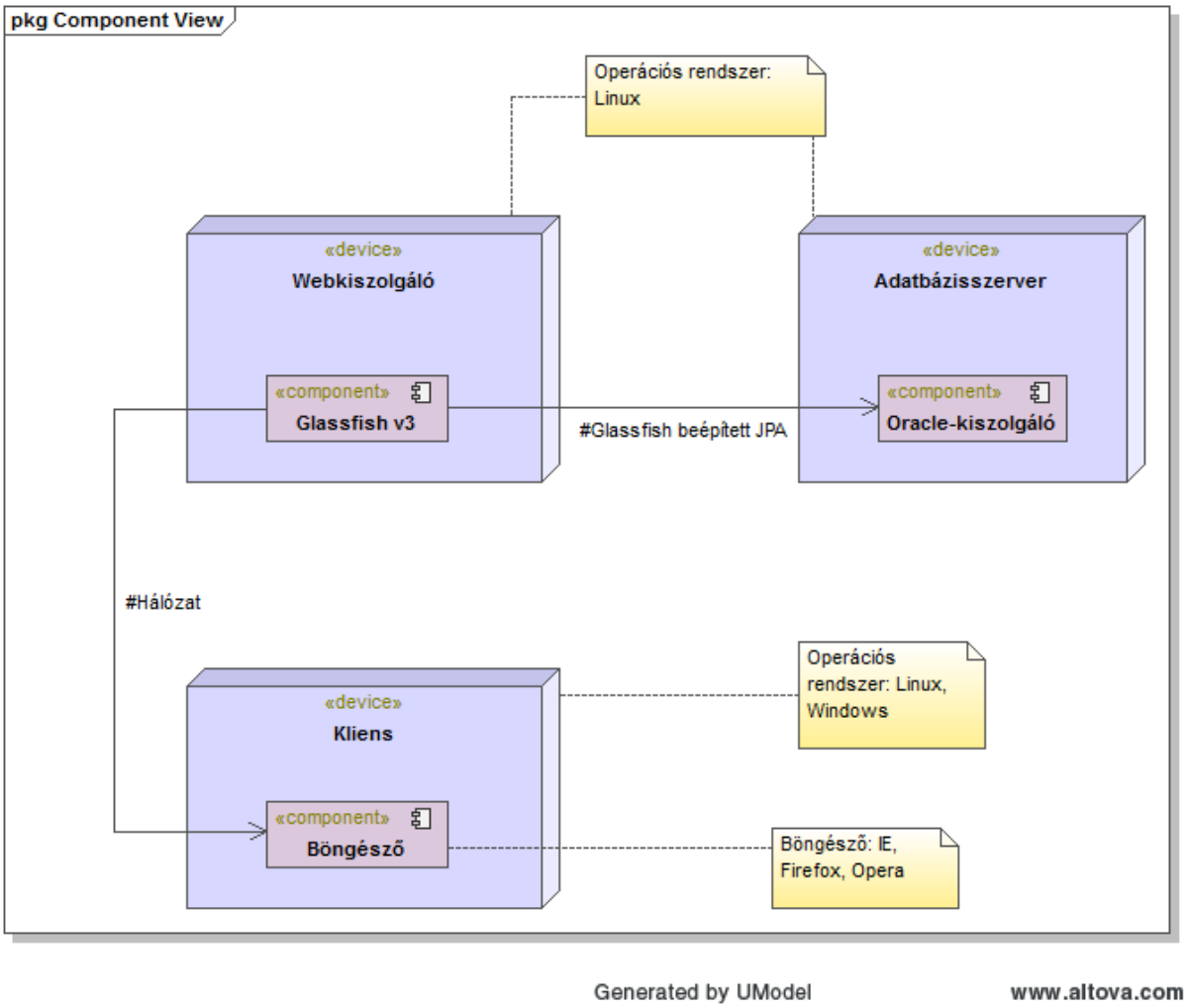
7. Kommunikációs diagram

7.1. Bejelentkezés

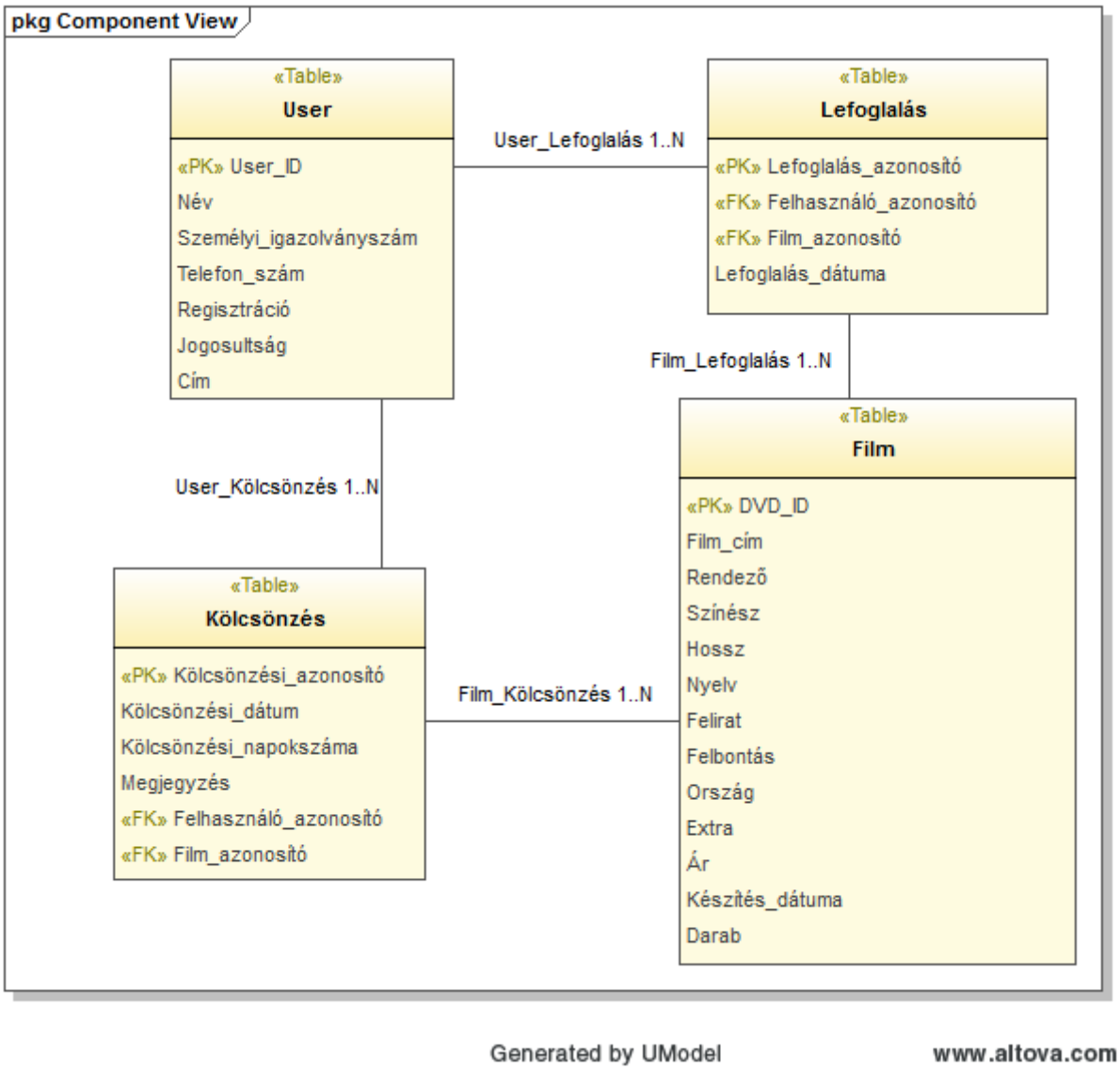


8. Kihelyezési diagram

A rendszerben résztvevő számítógépek két, elkülönülő részre oszthatóak. Az első kategóriába két szerver, az adatbázis- és a webkiszolgáló tartozik. A használt alkalmazások leolvashatóak az ábráról. A webkiszolgálóhoz a helyi hálózaton keresztül korlátlan mennyiségű kliensgéppel lehet csatlakozni egyetlen böngésző segítségével.


pkg Component View
Operációs rendszer: Linux
«device»
Webkiszolgáló
«component»
Glassfish v3
«device»
Adatbázisszerver
«component»
Oracle-kiszolgáló
#Glassfish beépített JPA
#Hálózat
«device»
Kliens
«component»
Böngésző
Operációs rendszer: Linux, Windows
Böngésző: IE, Firefox, Opera
Generated by UModel
www.altova.com

9. Adatbázisséma



User tábla: a rendszer felhasználóinak adatait tárolja

- User_ID: a tábla elsődleges kulcsa, egyedi azonosító
- Név: a felhasználó neve
- Személyi_igazolványszám: a felhasználó személyigazolvány száma
- Telefon_szám: a felhasználó telefonszáma
- Regisztráció: a beregisztrálás ideje
- Jogosultság: a felhasználó szerepköre
- Cím: a felhasználó postai címe

Film tábla: a kölcsönözhető filmek adatait tárolja

- DVD_ID: a tábla elsődleges kulcsa, egyedi azonosító
- Film_cím: a film címe
- Rendező: a film rendezőjének neve
- Színész: Híresebb filmben szereplő színész neve
- Hossz: a film lejátszási ideje
- Nyelv: a film hangjának nyelve
- Felirat: a filmhez tartozó feliratok nyelvei
- Felbontás: a film képernyőhöz tartozó aránya
- Ország: mely országban készült
- Extra: a lemezen lévő egyéb extra mellékletek
- Ár: a film egy napra kivehető ára
- Készítés_dátum: a film kiadásának dátuma
- Darab: készleten lévő darabszám a filmből

Lefoglalás tábla: a lefoglalt filmek listáját tartalmazza

- Lefoglalás_azonosító: a tábla elsődleges kulcsa, egyedi azonosító
- Felhasználó_azonosító: a lefoglalni kívánó személy azonosítója, külső kulcs a User táblára
- Film_azonosító: a lefoglalni kívánt film azonosítója, külső kulcs a film táblára
- Lefoglalás_dátuma: a lefoglalás ideje

Kölcsönzés tábla: a kölcsönzés adatait tartalmazza

- Kölcsönzési_azonosító: a tábla elsődleges kulcsa, egyedi azonosító
- Kölcsönzési dátum: a kölcsönzés kezdeti ideje
- Kölcsönzési_napok: a film elvitelének napjainak száma
- Megjegyzés: megjegyzés a kölcsönzéshez
- Felhasználó_azonosító: a kölcsönözni kívánó személy azonosítója, külső kulcs a User táblára
- Film_azonosító: a kivett film azonosítója, külső kulcs a Film táblára

10. Összefoglalás

A Java Enterprise Edition kiválóan alkalmas vállalati méretű rendszerek fejlesztésére.

Az adatbázissal összekapcsolva a JPA segítségével egyszerűen leképezhetők az objektumok a relációs-adatbázis szintjére. Az EJB 3 segítségével ezek könnyedén kezelhetők az eddigi 2.1-es verzióhoz képest. Az egyszerűsítés könnyebb érhetőséget adott a Java EE ezen vitatott részéhez.

A JSF keretrendszer lehetővé teszi a webes elérhetőséget az alkalmazások számára. A tervezőkkel egyszerűen és gyorsan lehet látványos felületet tervezni, ami mögé kerül az üzleti logika a Faces Servlet segítségével.

A kölcsönző alkalmazás eme technológia által került megtervezésre.

A szakdolgozat célja a Java EE technológia fontosabb részeinek bemutatása és a kölcsönző alkalmazás megtervezése és implementálása Glassfish v3 alkalmazásszerver és Oracle adatbázisszerver segítségével. Az alkalmazás nem került éles tesztelésre és az előjegyzés funkció nem került implementálásra benne.

Remélem sikerült bemutatnom a Java EE fontosabb alapjait.

Köszönetet szeretnék mondani témavezetőmnek, Kollár Lajosnak, aki lehetővé tette, hogy nála írjam a szakdolgozatomat és családomnak, akik támogattak ebben.

Irodalomjegyzék

- Imre Gábor: Szoftverfejlesztés Java EE platformon
- http://hu.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition
- http://hu.wikipedia.org/wiki/Java_(programozási_nyelv)