SZAKDOLGOZAT

Csete Gábor

DEBRECEN

2008.

Debreceni Egyetem

Informatika Kar

Témavezető:

Dr. Kuki Attila

adjunktus

Készítette:

Csete Gábor

Programtervező informatikus (Bsc)

Debrecen

2008.

Webes alkalmazásfejlesztés szabadon választott témában

Tartalomjegyzék

1 Bevezetés

A szakdolgozat írása során megpróbálom egy teljes alkalmazásfejlesztés életciklusát bemutatni, egy konkrét szoftver megtervezésén, és implementációján keresztül. Az alkalmazás egy elektronikus könyvtár lesz, amelyet E-Library-nek neveztem el. Az alkalmazáson keresztül szeretnék mélyebb betekintés nyerni a webes alkalmazás fejlesztésbe. Elsődleges célom, hogy egy jól működő, minden igényt kielégítő, hatékony szoftvert készítsek, amely segítségével megkönnyíthetem egy könyvtár működését. A program tervezésekor, arra számítok, hogy a program egyes felhasználói, egész nap használhatják majd a szoftvert, ezért fontosnak találom hogy a program kinézete, és kezelése egyszerű legyen. Egyszerű, könnyen átlátható felhasználói felületet szeretnék tervezni.

1.1 A Témaválasztás okai

A Debreceni Egyetemen eltöltött évek alatt, sok különböző technológiát ismertem meg. Különböző programnyelvekkel találkoztam, és sajátítottam el alap szinten. Ilyenek például a C, CLIPS, Java, PL/SQL, és korábbi tanulmányaimból a Pascal. Ezek közül számomra kiemelkedett a Java, amelyet tanulmányaim alatt legtöbbet használtam és legjobban megszerettem. Szabadidőmben is sokat foglalkoztam a Java nyelv megismerésével, így kerültek közel hozzám a Java alapú webes technológiák. Mivel korábbi tanulmányaimban nem tanultam erről a technológiáról, így a legtöbb technológia ismeretlen számomra. Azt várom, a szakdolgozat megírásától, hogy rálátásom lesz a Java alapú webes alkalmazásfejlesztésre, amely a későbbiekben sokat segíthet más hasonló technológiák elsajátításakor. Napjainkban a internet már-már minden háztartásban, illetve minden iparágazatban jelen van. Az iskola elvégzése után szeretnék ezen a szakterületen elhelyezkedni. A Rendszerfejlesztés című szemináriumon, megtanultam a rendszer megtervezéséhez, és implementációjához szükséges irányelveket, követelményfeltárási technikákat. A szakdolgozat egésze alatt hivatkozni fogok , az ott megtanultakra.

1.2 A dolgozatban felhasznált szoftverek, rendszerfejlesztési technológiák felsorolása

Úgy gondolom, hogy minden jól működő szoftver alapja a megfelelő követelmény feltárás, illetve követelményspecifikáció. A követelmények megfelelő feltárása nagyon fontos, mert pénzt, és időt spórolhatunk meg. A követelmények feltárását a megrendelővel együtt kell elvégezni. A megrendelő tudja pontosan megmondani, hogy mit akar, miért fizet. Ezeket a követelményeket dokumentálni kell ,ezzel megelőzhetjük a félreértéseket, és számtalan reklamációt. Nagy hangsúlyt fektettem a követelmények feltárására, amelyhez több technikát használtam. Elkészítettem az alkalmazáshoz a forgatókönyvet, fogalomszótárat, és az áttekintés dokumentumot. Ezek elkészítéséhez egyszerű

szövegszerkesztőt használtam.

A követelmények feltárásában nagy segítséget nyújtott az UML(Unified Modeling Language). Az UML, az objektumorientált programozás szabványos specifikációs nyelve. Az UML grafikus jelöléseket használ rendszerek absztrakt modelljének leírására. Az UML diagrammokat Netbeans 6.0.1 -ben készítettem el. A NetBeans „drag and drop" módszere segítségével könnyen elkészítettem a szekvencia diagrammot, az állapot átmeneti diagrammot, valamint a Use Case diagrammot.

A konkrét szoftverfejlesztés folyamán a Java DB adatbázist használtam, amely a NetBeans 6.0.1 feltelepítésével azonnal rendelkezésre állt. Az Apache Derby teljesen ingyenes, és a feladathoz tökéletesen megfelelő.

A webes alkalmazásomhoz szükségem volt, egy alkalmazás szerverre is. Itt szintén a NetBeans által felajánlott GlassFish nyílt forrású vállalati alkalmazásszervert használtam.

A megírt alkalmazást teszteltem Mozilla Firefox, illetve Internet Explorer segítségével is.

A rendszer fejlesztése során inkrementális rendszerfejlesztési modellt használok, amelyek közül a prototipus alapú rendszerfejlesztési modellt választottam, bár mivel kis rendszerről van szó, úgy gondolom nem fogom tudni kihasználni minden előnyét.

1.3 A szakdolgozat felépítése

Mivel a szakdolgozat egy rendszerfejlesztés teljes életciklusát kívánja bemutatni, így a szakdolgozat első részében, a követelmények feltárásával, és azok dokumentálásával foglalkozom részletesen. Ezután a dolgozat legterjedelmesebb részében foglalkozom a kód írása során felhasznált technológiákkal, mint például JSP oldalak tervezés, vagy a EJB nyújtotta lehetőségekkel.

2 Követelmények feltárása és dokumentálása

2.1 Követelményekről általában

A követelmények feltárása és dokumentálása több szempontból is nagyon fontos. A nem megfelelő követelmény feltárás és dokumentálás több problémát is okozhat:

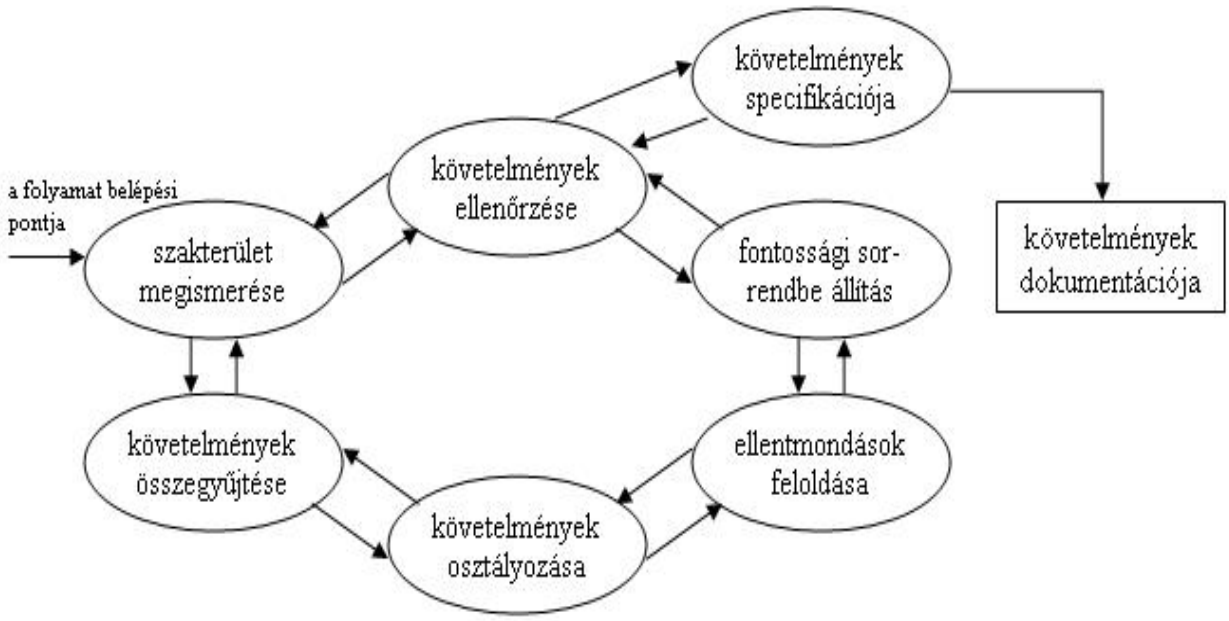
a) a teljesség hiánya, miatt lehetnek ellentmondások, illetve a reklamálásra adhat okot a megrendelő oldaláról
b) ha a követelmények nem konzisztensek, akkor nem lehet jól működő rendszert írni, és megint csak a megrendelő reklamálásával számolhatunk.
c) Pontatlanul megfogalmazott követelményeket kerülni kell

A követelményeket folyamatosan dokumentálni kell, és a megrendelővel aláíratni. A felhasználóval a követelmények meghatározásakor be kell vonni a felhasználókat. A felhasználók,

tudják elmondani elvárásaikat, igényeiket.

A követelményeket három nagy csoportba sorolhatjuk:

a) funkcionális követelmények: Ezek a követelmények amelyek a rendszer által nyújtott szolgáltatásokat jelenti.
   pl.: könyvkölcsönzés, könyvek listázása stb.
b) rendszertulajdonságok: az alrendszerektől függetlenek, a rendszer egészére jellemző dolgok.
   pl.: felhasználói felület
c) lehatárolás: Egyes esetekben könnyebb azt meghatározni egy rendszerrel szemben, hogy mit nem kell tudnia. Ilyenek lehet például a könyvtár programom esetén:
   pl.: Nem lehet könyvet vásárolni, így nem kell tudnunk a könyv értékét..



2.2 Követelményfeltárás életciklusa, forgatókönyv

Szakterület megismerése: A programozónak meg kell ismerkedni a szakterülettel, a hozzá kapcsolódó szakterületi fogalmakkal. Az én esetemben, mivel magamnak fejlesztem a rendszert, ezért csak a „józan ész”-re, és az interneten összeszedhető információkra támaszkodhatok.

Követelmények összegyűjtése:

Többféle technika létezik a követelmények feltárására. Ilyenek a következők:

- prototípus készítés: Ez olyan esetekben hatásos, amikor a megrendelő nem tudja pontosan

elmondani, hogy mit is szeretne. Ilyenkor csinálunk egy prototípust, és azon pontosítjuk a követelményeket. Hibája, hogy a prototípus készítése idő, és pénz igényes.

- Nézőpont-orientált megközelítés: Azok a személyek akiknek a rendszert tervezzük, más és más szemszögből láthatják a rendszert. Az én esetemben a könyvtárba beiratkozott személy, és a könyvtáros másképp látják a rendszert, és más-más elvárásaik is vannak a rendszerrel szemben. Minden kulcsfigura nézőpontját meg kell vizsgálnunk, hogy a teljes követelményrendszert fel tudjuk tárni. Az informatikusnak személyesen kell beszélnie a kulcsfigurákkal. Hibája lehet ennek a technikának, hogy bizonyos kulcsfigurák nem mondanak el olyan dolgokat, ami neki természetesek, illetve nem tudják pontosan kifejezni elvárásaikat.
- Etnográfia: Ez a technika megfigyelésen alapuló technika. Adott egy probléma kör, és a programozónak be kell helyeznie magát az adott környezetbe. Semmilyen kommunikációra nincs szükség, azonban nagyon hosszadalmas, és ezzel együtt költséges eljárás, és a megfigyelő embernek jól képzettnek, tapasztaltnak kell lennie. A rendszer tervezésénél megtehettem volna, hogy valamelyik könyvtár működését megfigyeltem volna.
- Forgatókönyv technika: A forgatókönyv technika már a 80-as években is létezett. Nagyon gyakran használt eljárás. Nagy erőssége abban rejlik, hogy a megrendelő számára is nagyon könnyen érthető. Nem tartalmaz informatikai szakfogalmakat, teljesen köznyelven íródott dokumentum, amely a rendszer követelményeinek pontosítására szolgál. A forgatókönyvben leírjuk a rendszer működési állapotait, hiba üzeneteit. A forgatókönyvben leírjuk a rendszerrel végezhető műveleteket, és az azokhoz tartozó fontosabb információkat. Néhány fontosabb, ebben a dokumentumban szereplő információ:
  - Induló feltétel: a rendszerrel végzett interakció megkezdése előtt a rendszert állapotát leíró feltétel.
  - Működés: A rendszer általános, hibamentes működésének leírása. Leírja, hogy mit kell a rendszernek tennie, egy adott interakció esetén.
  - Hibák: Az adott esemény végrehajtása során, felléphető hibák. Itt minden egyes hibaüzenetet meg kell jelenítenünk, amely előfordulhat a működés folyamán.
  - Végfeltétel: A szabályos lefutást követően, a rendszerünk ebben az állapotban lesz. Ilyen például, hogy egy sikeres regisztráció után, a felhasználó eltárolódik az adatbázisban.

  Néhány fontosabb részlet az általam az E-Library-hez elkészített forgatókönyvből:

  Regisztráció:

  Induló feltétel: Aki a rendszert használni szeretnél, annak regisztrálni kell magát a rendszerbe. Regisztráció nélkül csak nagyon kevés opció érhető el.

  Működés: A felhasználó a regisztráció gomb el segítségével, elindítja a regisztrációs

folyamatot. A rendszer egy űrlapot fog megjeleníteni ahol, a személy saját adatait, hozzáférési azonosítóját, jelszavát adhatja meg. Ha ezeket, az adatokat jól kitölti, akkor az adott személy regisztrálva lesz a rendszerben.

Hibák: A felhasználó helytelenül tölti ki az űrlapot. Ez esetben az űrlapot újra a felhasználó elé kell tenni javításra. Ha az újra elküldött űrlap is helytelen, akkor újra a felhasználó elé kerül, egészen addig, amíg meg nem szakítja a regisztrációs folyamatot, vagy pedig helyesen ki nem tölti az őrlapot. Azonos felhasználói név esetén újat kell választani. Ekkor ezt a rendszer hibaüzenettel jelzi.

Rendszerállapot befejeződéskor: A felhasználó nincs be jelentkezve, de már bejelentkezhet a rendszerbe. A regisztrálni kívánt személy bekerült az adatbázisba.

Bejelentkezés:

Induló feltétel: A felhasználónak már regisztrált felhasználónak kell lennie.

Működés: A felhasználó az azonosítója és a hozzátartozó jelszó segítségével, bejelentkezik a rendszerbe.

Hibák: Lehetséges hogy a felhasználó elfelejtette jelszavát, azonosítóját vagy elgépelt

valamit. A bejelentkezés ablakot újra be kell tölteni, addig amíg nem történik meg a sikeres bejelentkezés.

Rendszerállapot befejeződéskor: A felhasználó be van jelentkezve, és jogosultságainak megfelelő műveleteket végezhet a rendszerben.

Könyv Törlése:

Induló feltétel: Az informatikus könyvtáros bejelentkezés után törölhet könyveket az adatbázisból.

Működés: Az informatikus könyvtáros a könyv azonosításához szükséges adatok megadásával, és a törlés gomb megnyomásával eltávolíthat valamilyen könyvet az adatbázisból..

Hibák: A könyv adatai helytelenek. Semmi nem történik.

Rendszerállapot befejeződéskor: Az adatbázisból törlődött 1 könyv.

Könyv létrehozása:

Induló feltétel: Az informatikus könyvtáros bejelentkezés után vihet be új könyveket.

Működés: Az informatikus könyvtáros az „új könyv” nevű gomb segítségével felugró oldal kitöltése után hozhat létre egy új könyvet az adatbázisban.

Hibák: Nem töltött ki minden mezőt, vagy a könyv már létezik. A rendszer ezt hibaüzenettel jelzi.

Rendszerállapot befejeződéskor: Sikeres művelet esetén új könyv kerül az adatbázisba, egyébként nincs változás.

Könyv foglalása:

Induló feltétel: Az ügyfél számára bejelentkezés után végrehajtható művelet.

Működés: Az ügyfél kiválasztja a kikölcsönözi szánt könyvet, és a „kikölcsönöz” gomb segítségével kiveszi azt. A program üzenetben értesít a kölcsönzés eredményéről.

Hibák: Ha nincs az adott könyvből a könyvtárban, akkor hibaüzenetet kapunk.

Rendszerállapot befejeződéskor: Ha az ügyfél sikeresen kikölcsönzött egy könyvet akkor az megjelenik az adatbázisban is.

Ellentmondások feloldása:
A rendszer fejlesztése során nem engedhetjük meg, hogy a követelményeink ellentmondásosak legyenek.

Követelmények ellenőrzése:
Ezen a ponton újra le kell ellenőrizni az egész követelményrendszert. Az utolsó esély, hogy nagyon fáradságos munkától kíméljük meg magunkat, mert ha az implementáció közben vesszük észre a hibát, lehetséges hogy az egész rendszerfejlesztést elölről kell kezdenünk.

2.3 Egyéb a rendszerhez elkészült dokumentum

2.4 Áttekintés dokumentum

Az áttekintés dokumentum a felhasználóknak szóló, egyszerű nyelven megfogalmazott dokumentum. Megrendelő és a programozó adategyeztetését, követelmények pontosítását szolgáló dokumentum.
Az áttekintés dokumentum 3 részből áll:

- Általános leírás: A rendszer általános, rövid leírása, amely tartalmazza a rendszer működését, hogy mire való a szoftver, milyen céllal készült el. Egyszerűen megfogalmazva, szakspecifikus formalizmus nélkül.
- Általános követelmények: Ebben a dokumentumban szerepelnek, az olyan szabványok amelyeket az elkészült alkalmazásnak be kell tartania, és egyéb olyan általános követelmények, amelyeket programozónak be kell tartania.
  Tartalmazza az azoknak a dokumentumoknak a listáját is, amelyet a szoftverrel

együtt át kell adni a megrendelőnek.
Például: többnyelvűség, kinézet

- Rendszerkövetelmények: Leírja azt a környezetet, amely szükséges a szoftver működéséhez. Ilyenek lehetnek például, minimális gépigény, szükséges más szoftverek.

I. Általános leírás

Az E-Library egy elektronikus könyvtár, amely segítségével a felhasználók, könnyen, otthonról foglalhatnak le könyveket. Az elektronikusan lefoglalt könyvet személyesen vehetik ki a könyvtárból, amelyet számukra a lefoglalás pillanatában félre tettek. Az E-Library teljeskörű dokumentálást tesz lehetővé könyvtárban kikölcsönzött, illetve bent lévő könyvekről.

Az informatikus könyvtáros feladata az elektronikus könyvtár naprakészen tartása. Az informatikus könyvtárosnak joga van új könyvet felvenni az adatbázisba, törölni az adatbázisból, és az ügyfelek adatlapjait kezelni.

A rendszert háromféle személy használhatja, az ügyfelek, akik az E-Library szolgáltatását igénybe veszi, azaz könyvet kölcsönöznek, az informatikus könyvtáros, aki az elektronikus könyvtár naprakészségéért felelős, illetve az adminisztrátor, aki a rendszer karbantartásáért felelős.

II. Általános követelmények

1. A szoftverfejlesztés minden lépése az ISO IEC 90003 2004 Software Standard-nak megfelelően kell történnie, melynek következetes betartásából származó dokumentumok a HTF Adventure számára hozzáférhetők kell, hogy legyenek. A dokumentálás "OpenDocument Text" (ISO/IEC 26300:2006) formátumban kell történnie.

3. Alkalmazottaink egész napos munkájukat gépnél töltik, ezért lényeges a felületek szép kinézete, és a szemet nem zavaró színek, zavaró funkcióelrendezések kerülése.

4. A rendszer minden művelet eredményéről tájékoztatja a szoftver használóját. Hiba esetén értesítést ad a probléma okáról a felhasználó minőségétől függően.

5. A rendszer minden elvégzett és végrehajtott műveletről naplózást kell, hogy végezzen.

6. A szoftvernek a cég által használt szakterületi fogalmakat kell használnia melytől nem térhet el. Ennek részletes listázása a „Fogalomszótár” című dokumentumban található.

7. A rendszer minden kommunikációja biztonságos csatornán kell, hogy történjen.

III. Rendszerkövetelmények

Operációs rendszer: Platform függetlennek kell lennie.

Programozási nyelv: Java technológia

Célhardver: A szoftver igényeihez fognak igazodni. Szoftver tervezésekor nem kell figyelembe venni.

Várható felhasználók száma: 1000 fő

2.5 Fogalomszótár

A fogalomszótár a szakterület kifejezéseit tartalmazó dokumentum. A E-Library esetén ennek a dokumentumnak nem látszódik a valódi ereje, mert ez egy kisebb rendszer.
Néhány példa a E-Library-hoz elkészül fogalomszótárból:

adminisztrátor: Informatikai végzettséggel rendelkező személy. Feladatai között találhatóak a következők: rendszer karbantartás, új informatikus könyvtáros felvétele

informatikus könyvtáros: Az E-Library-val dolgozó személy. Feladata a könyvtár tartalmának naprakészen tartása. Új könyvek felvétele az adatbázisba.

Ügyfél: Az E-Library felhasználója. Interneten keresztül képes könyvet lefoglalni magának , és azt később kivenni.

könyv: Egy olyan entitás amelynek a következő attribútumai vannak:

Cím, Szerző(k), Kategória, Kiadás éve, Kiadó, Kiadások száma, jegyzet, aktuális darabszám, darabszám

2.5.1 Szakterületi kapcsolatok és folyamatok dokumentum:

Egyszerű nyelven íródott dokumentum, amely a programozónak nyújt segítséget, az egyes folyamatok működésének megértéséhez.

Könyv foglalása– A felhasználó feladata, amely során a meglévő könyvek közül választ egy számár megfelelőt, és jelzi kivételi szándékát. Ha van a könyvtárban az adott könyvből, akkor egyet lefoglal magának.

Könyv hozzáadás**a** – Az informatikus könyvtáros feladata, amely során egy új könyvet vesz fel a rendszerbe, úgy hogy megadja a könyv minden attribútumát.

Könyv adatainak beállítása – az a folyamat, mely során az informatikus könyvtáros egy adott könyv adatait átállíthatja. Hibajavításra alkalmas.

Könyv törlése – az a folyamat, mely során az informatikus könyvtáros, eltávolít egy meglévő könyvet az adatbázisból.

Könyvek listázása – az a folyamat, amely során a felhasználók megtekintheti a könyvtárban lévő könyvek listáját.

Könyvek keresés – az a folyamat, amelynek során a felhasználó, vagy az informatikus könyvtáros keres egy könyvet a könyvtárban. Többféle keresési szempont lehetséges. Pl.: Cím alapján, Szerző alapján, Kiadó alapján stb.

Kérés törlése: az a folyamat amely során az informatikus könyvtáros egy könyv kivételét visszautasíthatja valamilyen okból.

3 Tervezés

3.1 Elkészített UML diagrammok, és azok leírása

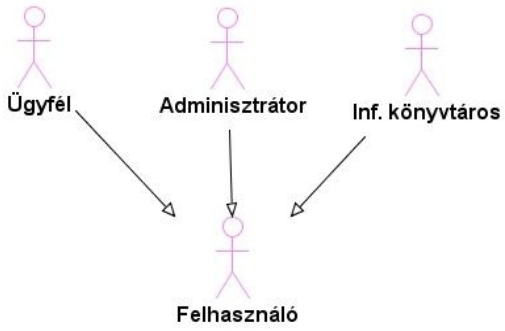
Az elkészített diagrammokat NetBeans 5.5.1 illetve 6.0.1-es IDE-vel készítettem, és ennek a jelöléseit ismertetem, és használom. A NetBeans-be beépített UML Plugin, könnyen kezelhető Drag and drop eszközt biztosít a tervezés teljes életciklusa alatt. Az NetBeans 5.5.1-es verziójában még nincs beépítve, de könnyen letölthető a Tools/Update Center menüpont alatt. A régebbi verziót használtam korábbi tanulmányaim alkalmával, és számtalan hiba is jellemezte. A legújabb NetBeans verzióban viszont a legtöbb problémát kijavították, és kellemes környezetet teremt a rendszer megtervezéséhez.

3.2 Használati eset diagramm

A Use Case diagramm egy funkcionális diagram, ennek megfelelően a rendszer funkcionalitását lehet vele modellezni. Jól átlátható, könnyen értelmezhető ábrákat lehet vele készíteni. A követelmények leírására, pontosítására szolgáló grafikus, modellező eszköz. A szoftverfejlesztés első fázisában használjuk. A használati eset diagramm leírja a rendszer, és az őt felhasználó külső szereplők (aktorok) közötti akciók, és reakciók(válaszok) sorozatát. Az aktorok nem tartoznak bele a rendszerbe, csak a kapcsolatokat jelölik.

A diagramm elemi:

- Aktorok: az aktorok a következők lehetnek:
  - Emberek különböző csoportjai. Az általam írt könyvtár programban 3 fajta felhasználó csoport van: Adminisztrátorok, Ügyfelek, Informatikus könyvtárosok
  - más a rendszerrel együttműködő programok. Az E-Library programban is található erre példa, mivel a programhoz tartozik egy adatbázis.
  - Hardvereszközök
- Rendszer: A rendszer jelölésére egy téglalapot használunk, amelybe beleírjuk a rendszer nevét. A téglalapon belül fognak elhelyezkedni a használati esetek.
- Használati esetek: A rendszer funkcionalitásainak jelölésére szolgál.
  Jelölése: A funkció neve egy oválisba írva.
- Relációk: Az előző alapelemek kapcsolatát leíró eszközök. Amelyek lehetnek a következők:
  asszociáció: Jelölés folytonos vonal, az aktorok, és a rendszer között csak ilyen szerepelhet. Az vonal két végén jelölhető a kapcsolat számossága. Természetesen az asszociáció nem irányított, mivel az asszociációban résztvevő felek együttműködéséről nem mondunk semmit.



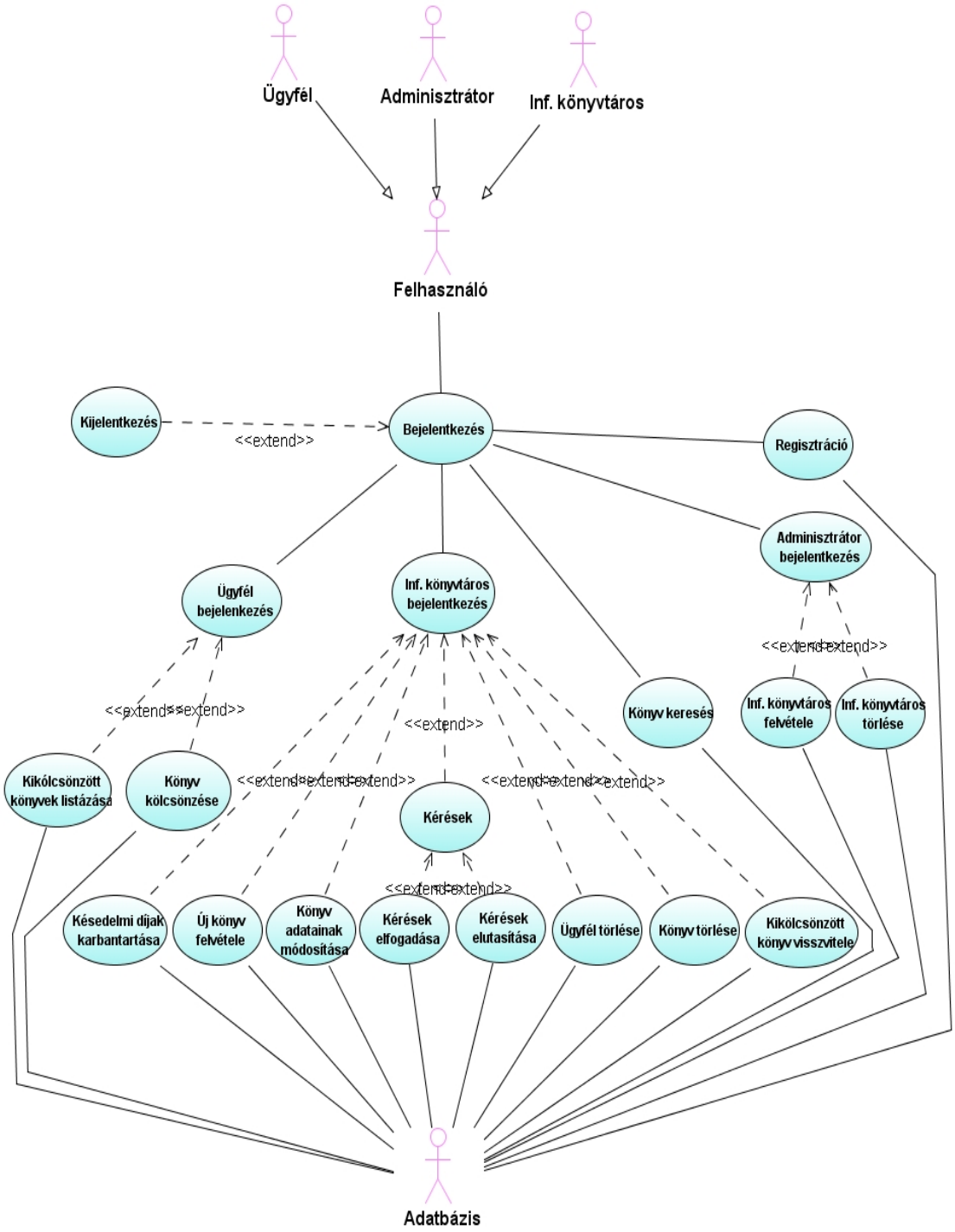
•Általánosítás: Az általánosítás értelmezhető aktor és aktor között, vagy használati estek között. Az általános aktornak helyettesíthetőnek kell lennie a specializált aktoréval. A helyettesíthetőség az aktorokhoz kapcsolódó használati esetek vonatkozásában értendő. Jelölésére egyvégű irányított nyilat használunk, amely az általánosított aktor irányába mutat. A képen is jól látható, hogy a E-Library-nek 3 különböző felhasználója van.

- Include: A használati esetek között értelmezett reláció, amelyet egyvégű, szaggatott nyíllal jelölünk, és egy <<include>> címkével látunk el. Nyíl talpánál álló használati eset tartalmazza, a nyíl hegyével jelzett használati esetet. A tartalmazott használati eset kiemelésére szolgál, és a tartalmazott használati eset funkcionalitása mindig hamarabb alkalmazásra kerül, mint a tartalmazó eset.

  Az include direktívára egy példa az E-Library esetén, amikor bejelentkezünk, akkor a bejelentkezés végrehajtása előtt végrehajtódik egy jelszó ellenőrzés.
- Extend: Más szóval kiterjesztés. Az alap használati esetet kibővíti egy másik használati egység. A kiterjesztett használati eset, egy bizonyos feltétel teljesülése során megszakítja az alaphasználati eset működését, és lefut.

  Jelölése: A kiterjesztett használati estből, az alap használati esethez húzott szaggatott nyíllal jelöljük, amelyre az <<extend>> sztereotípiát írjuk.

Az E-Library egy tipikusan 3 szintű alkalmazás amely a Use Case diagrammra nézve is látszódik.



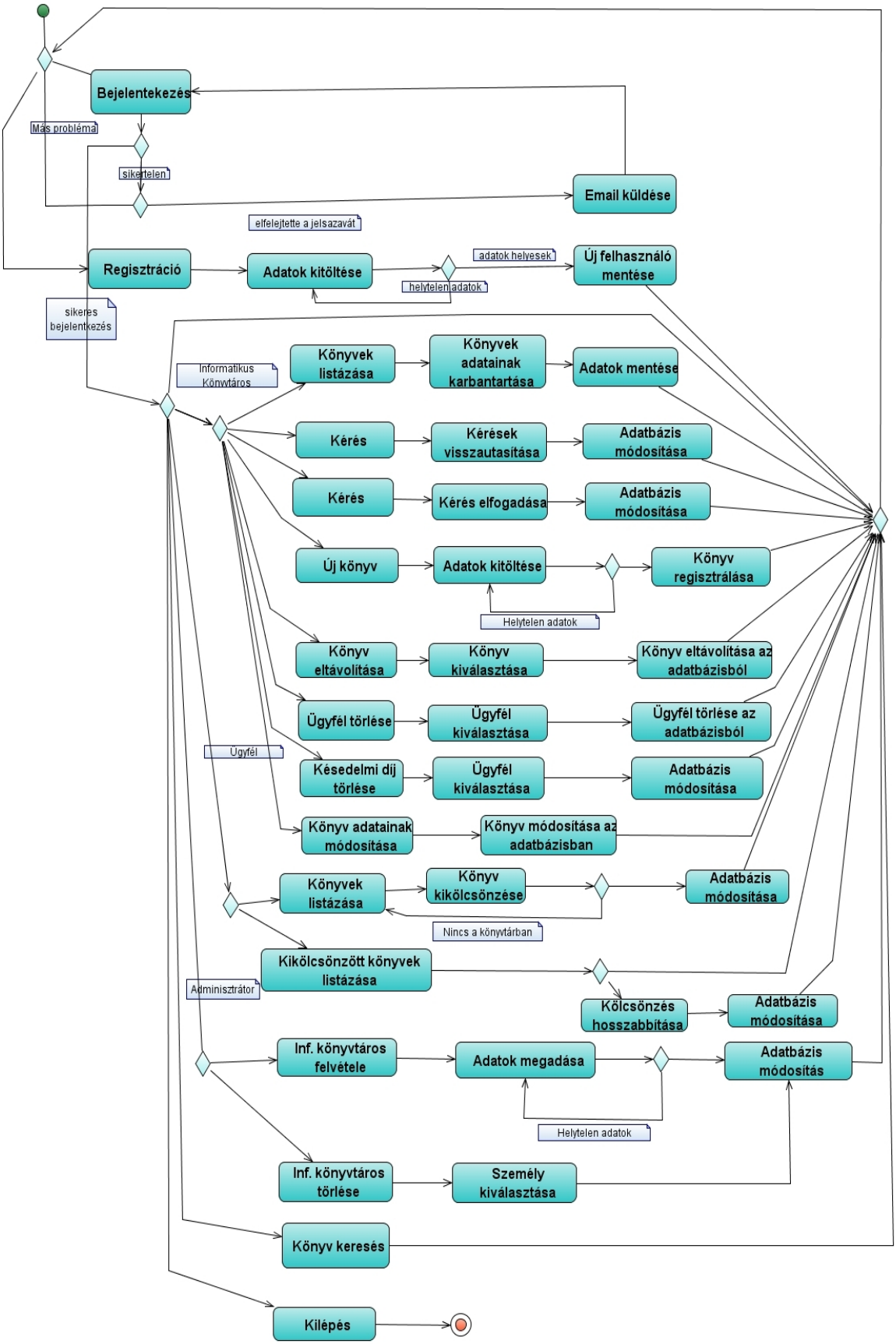
3.3 Aktivitási diagramm

Az aktivitási diagram a rendszer dinamikus viselkedését írja le úgy, hogy megadja a vezérlés menetét eseményről eseményre. A diagramm az alkalmazás lehetséges időbeli lefutásait modellezi. Minden egyes folyamatot, amely a rendszer futása közben lezajlik kiválóan modellezni tudunk ezzel az eszközzel. Ezek a folyamatok különböző jellegűek lehetnek, mint például függvények, eljárások algoritmusai, vagy egyes használati esetek lefutásai. Az aktivitási diagramm

szemcsézettsége tervezés függő. Az E-Library rendszer tervezésénél, a főfolyamatok modellezését tartottam fő célnak, és nem fejtettem ki minden egyes folyamatot részletesen. Például az „Változások mentése az adatbázisban” nevű folyamatot is tovább lehet bontani, és leírni az algoritmusát. A folyamatok szemcsézettségétől függően egy-egy aktivitás lehet akár egy függvény hívás, menüpont vagy akár utasítás is.

Jelölésrendszere:

Minden egyes aktivitási diagram egy belépési ponttal kezdődik, amelyet egy zöld körlappal jelölünk. Az egyes aktivitásokat egy lekerekített téglalappal jelöljük, amelybe beleírjuk az adott aktivitást. Az aktivitásokat egyvégű nyíllal kötjük össze, ami az egyik aktivitásból a másik aktivitásba való átmenetet jelöli. Az átmenet biztosít egyfajta szekvencialitást, tehát ha az egyik átmenet már befejezése után következhet csak a következő aktivitás, azaz időrendi sorrendet fejez ki a nyíl. A folyamatok modellezéséhez szükség van elágaztatásra is. Az elágaztatások kezelését őrszemek segítségével végezhetjük el. Az őrszem kinézete egy rombusz, amelyekből minden aktivitáshoz egy-egy vonalat húzunk, és azon az aktivitáson folytatódik a futás amelyre a feltétel igaz. A különböző ágaknak páronként kölcsönösen kizárónak kel lenniük. A modellünket elláthatjuk megjegyzésekkel is, amely egyszerű téglalapokba írt szöveges információkkal lehetséges, mint ahogy az ábrán látható is. Minden folyamatot úgy kell modelleznünk, hogy a végső lépés a kilépés legyen, és ne legyenek „zsákutcák”. A kilépést a diagrammon egy piros körlappal jelezhetjük.

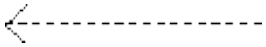
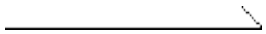


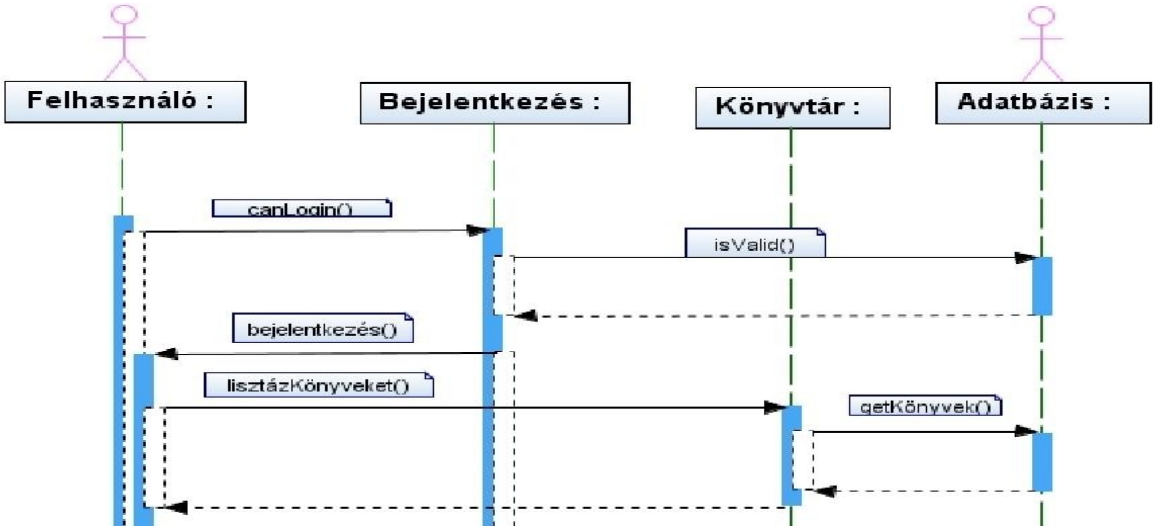
Természetesen van még több olyan eszköz, amelyet nem mutattam be, de azok az én rendszeremhez nem voltak szükségesek. Lehet például szinkronizációt, illetve végrehajtási jogokat is jelölni az aktivitási diagrammon.

3.4 Szekvenciális diagram

Az objektumok közötti üzenetváltások időbeli menetét szemlélteti, tehát az objektumok közötti interakciókat követi nyomon, időrendi sorrendben. A szekvenciális diagramm esetén, a résztvevő objektumokat a az oldal tetején jelöljük (Objektum: Objektum típusa) formában, és egy a neve alatt függőleges irányban húzott, úgynevezett életvonal segítségével. Az életvonalat szaggatott vonallal jelöljük, és az objektum addig létezik, amíg az életvonal tart. Az életvonal fentről lefele haladva meghatározza a végrehajtási sorrendet, azaz időbeli sorrendet definiál. Az objektumok üzenetváltások segítségével kommunikálnak. Az üzeneteknek több fajtája van:

- Szinkron: 

  Szinkron üzenet esetén a küldő objektum elküld egy üzenetet egy másik objektumnak, amit, amíg a másik objektum meg nem válaszol, és nem kerül vissza a vezérlés , addig tétlenül várakozik . Jelölése egyfejű, telt fejű nyíllal a küldő objektumtól a fogadó objektum életvonala között.
- Visszatérés:  A vezérlés visszatérését, illetve a válaszüzenetet jelöljük a képen látható módon.
- Aszinkron:  Asszinkron nyilat a küldő objektum életvonala, és a fogadó objektum életvonala között húzott félfejű nyíl segítségével jelölhetjük. Az asszinkron üzenet lényeg az, hogy az üzenet küldője elküldi az üzenetet, és nem foglalkozik azzal, hogy a fogadó megválaszolja-e.
- Öndelegáció: Amikor egy objektum egy saját funkcionalitását akarja meghívni, azt öndelegációnak nevezzük. Jelölése: A küldő objektum életvonaláról elindítunk egy üzenet, és ugyanazon az életvonalon is fejeződik be.



A nyilakon feltüntethetjük az üzenet nevét, és időtartamát is. Az objektum létrejöttét, az életvonalon megjelenő kis téglalappal jelölhetjük. Az elhalálozás jelölésére pedig az életvonal alatt elhelyezett

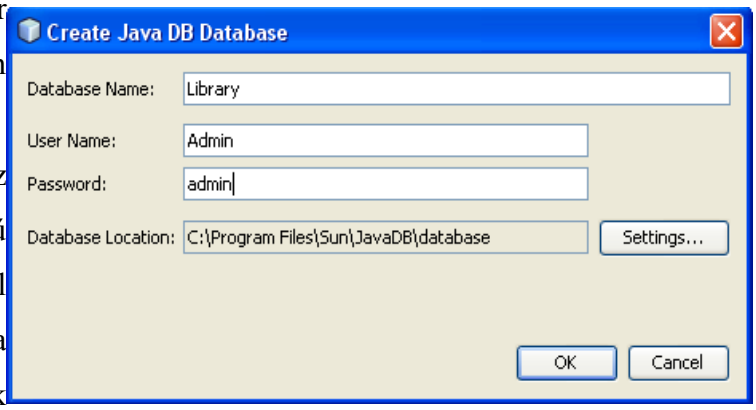
X-el jelölhetjük.

Az fenti ábrán a szekvencia diagramm egy kisebb részlete látható. Négy darab életvonal látható a lépen. Az első életvonal a felhasználó életvonala. Tulajdonképpen ő az, aki interakciót kezdeményez a rendszerrel. A második életvonal a Bejelentkezés életvonala, harmadik életvonal a könyvtárhoz, míg a negyedik életvonal az adatbázishoz tartozó életvonal. Jól látszik, hogy a bejelentkezés a rendszerbe, úgy megy végre, hogy először elindítjuk a bejelentkezés folyamatát. Megadjuk a bejelentkezéshez szükséges adatokat, majd az adatbázisból lekérdezzük, hogy van-e ilyen felhasználó, és a jelszó megfelelő-e? Ha minden rendeben van, és az adatok is megfelelőek akkor a felhasználó belép a rendszerbe.

3.5 Adatbázis megtervezése

Mint minden nagy mennyiségű adatokat kezelő program alapja, az E-Library program alatt is egy adatbázis működik. Mivel folyamatosan nyilván kell tartani a könyvtárhoz kapcsolódó adatokat. (könyvek, felhasználók, kölcsönzött könyvek, vagy a logolást) Adatbázist tárolhattam volna, a program által kezelendő adatokat egyszerű állományban is, mint például XML. Azonban ez a megoldás nem tűnik elég jónak, mivel a programnak sok adattal kell dolgoznia, és ezek között különböző viszonyt kell kialakítani, ráadásul az adataim táblázatban vagy fa struktúrában ábrázolhatók, ezért relációs adatbázist használok.

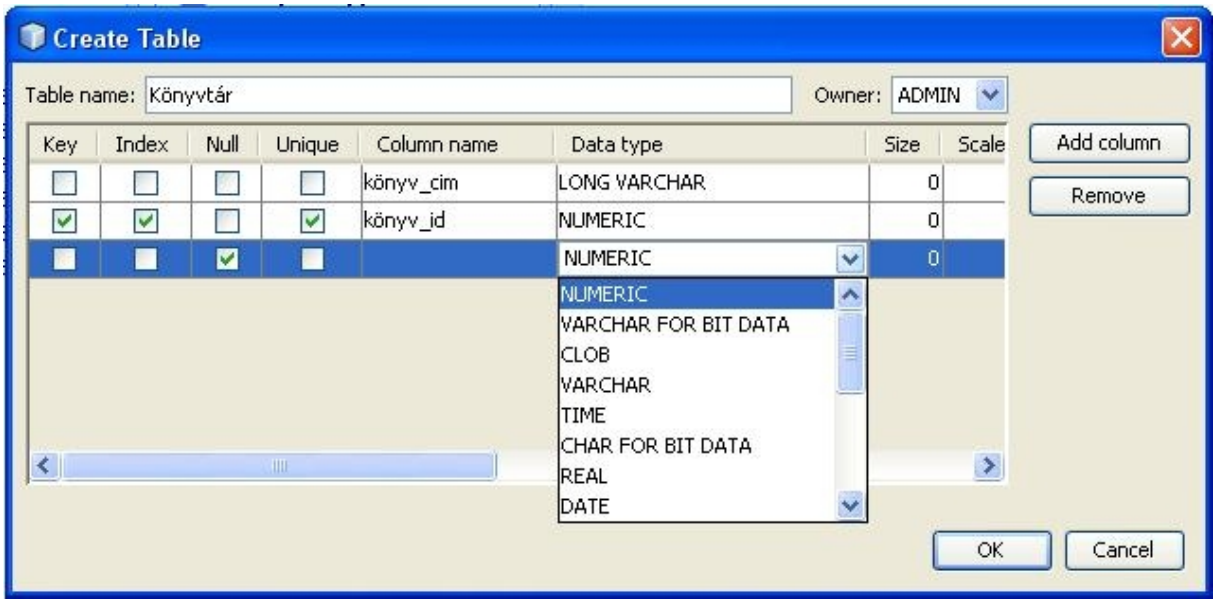
Célszerű olyan relációs adatbázist használni amely teljesíti a SQL92 szabványt, de Java esetén legjobb, ha JDBC4 verziójú JDBC driver alapú adatbázist használunk.. A legegyszerűbb megoldás a JavaDB. A NetBeans 6.0.1-ba bele van integrálva a JavaDb, amely tulajdonképpen egy nyílt forrású Apache Derby relációs adatbázis-kezelő. Az adatbázis létrehozásához a NetBeans könnyen kezelhető grafikus felületet biztosít. A Tools/Java DB Database menüpont alatt ki kell választani a Create new DataBase opciót. Ekkor megjelenik az itt látható, panel, amelyen beállíthatjuk az adatbázis adatait.



Ezek után az adatbázis elindításához nyomjuk le a CTRL+ 5 billentyű kombinációt. Ekkor a NetBeans 6.0.1 bal oldalán megjelennek az adatbázisok(ha van több is). Megkeressük az általunk létrehozott adatbázis, amelyen jobb egérgombot megnyomva elindíthatjuk az adatbázisszervert. A következő üzenet jelenik meg az output ablakban:

Apache Derby Hálózati kiszolgáló - 10.2.2.0 - (485682) elindult és kész a kapcsolatok fogadására a(z) 2008-03-31 17:23:22.578 GMT 1527 portján

Ezek után nincs más dolgunk, mint létrehozni a rendszerhez szükséges táblákat, amelyeknek két különböző módja van. Természetesen lehetőség van adatbázist szkript segítségével létrehozni, és emellett a Netbeans grafikus felületet is biztosít új táblák létrehozásához. .



A táblák létrehozásához saját szkriptet használtam. Az adatbázis alapjának 4 különböző táblát fogok használni. Az első táblába, a felhasználóhoz tartozó adatokat fogom tárolni. A felhasználó több entitást is tartalmaz, így a tábla nem lesz 3. normálformában, így az implementáció egyszerűbb lesz számomra.

A szkriptekben használt típusok:

- VARCHAR(x): Karakteres típus, amelynél az x egész szám, azt adja meg, hogy maximálisan hány karakter hosszú lehet.
- NUMERIC(x): Numerikus típus, amely hasonlóan a varchar-hoz, meg lehet határozni a számjegyek maximális számát.
- BOOLEAN: Egyszerű logika típus.

3.5.1 Felhasználó tábla

```
CREATE TABLE felhasznalo
 (
   email_cim  VARCHAR(30),
   v_nev    VARCHAR(20),
   k_nev    VARCHAR(20),
   jelszo   VARCHAR(15),
   szul_ev  NUMERIC(4),
   szul_ho  NUMERIC(2),
   szul_nap NUMERIC(2),
   telszam  VARCHAR(15),
```

```
   szerep   VARCHAR(15),
   ir_szam  NUMERIC(4),
   cim      VARCHAR(50),
   CONSTRAINT felhasznalo_pk PRIMARY KEY (email_cim)
 )
```

A tábla elsődleges kulcsának az e-mail címet gondoltam, mivel így könnyen elkerülhetem az összetett kulcs használatát, valamint azt feltételezve hogy az email címek egyediek, nem lesz probléma a felhasználó nevek kitalálásával. A tábla mezőit igyekeztem „beszédes" nevekkel ellátni. Azt hiszem ez sikerült is a rövidítések ellenére is. Talán az egyetlen mező, amely magyarázatra szolgál, az a *szerep* mező. Ennek a mezőnek a segítségével fogom eldönteni bejelentkezéskor, hogy melyik JSP oldalt kell megjelenítenem, azaz ki jelentkezik be. (felhasználó, informatikus könyvtáros, vagy adminisztrátor) Ennek megfelelően, három különböző értéket vehet fel. Ezek az *adminisztrátor, informatikus könyvtáros*, illetve *felhasználó*.

3.5.2 Könyv tábla:

A következő tábla a *könyv* tábla. Ebben a táblában fogom tárolni, a könyvtár birtokában lévő könyvek adatait. A *könyv* táblát a következő szkript segítségével lehet létrehozni:

```
CREATE TABLE konyv
 ( konyv_cim  VARCHAR(60),
   konyv_iro  VARCHAR(30),
   kiado  VARCHAR(30),
   kiadas_eve NUMERIC(4),
   kiadas_szam NUMERIC(3),
   darabszam NUMERIC(5),
   akt_szam NUMERIC(5),
    CONSTRAINT konyv_pk PRIMARY KEY (konyv_cim, konyv_iro, kiadas_szam )
 )
```

Ennek a táblának a létrehozásakor is figyeltem az egyszerű érthetőségre. 3 attribútumot mégis kiemelnék. A *kiadas_szam* mező, a könyvről árulja el hogy hányadik utánnyomásról van szó, mivel egyes könyvek esetében nagy különbségek lehetségesek, a kiadások számától függően(pl.: Java 2). A *darabszam* mező, azért szükséges hogy tudjuk a könyvtár, hány példánnyal rendelkezik az adott könyvből. Az *akt_szam* attribútumot segítségével a könyvtárban aktuálisan ki nem kölcsönzött példányok számát tárolom. Ebben a táblában nem összetett kulcsot alkalmazok, amely a következő attribútumokból áll:

- Könyv címe
- Könyv írója
- Kiadások száma

3.5.3 Kölcsön tábla

A harmadik táblában, a könyvtári kölcsönzéshez kapcsolódó elengedhetetlen információkat tárolom. Ezek alapján lehet majd, dokumentálni a kölcsönzéseket, és határidőket.

```
CREATE TABLE kolcson
 ( email_cim  VARCHAR(30),
   konyv_cim    VARCHAR(60),
   k_iro    VARCHAR(30),
   kiadas_szam NUMERIC(3),
   hatarido DATE,
   CONSTRAINT kolcson_pk PRIMARY KEY (email_cim),
   CONSTRAINT kolcson_fk1 FOREIGN KEY (email_cim) REFERENCES felhasznalo(email_cim),
   CONSTRAINT kolcson_fk2 FOREIGN KEY (konyv_cim,k_iro,kiadas_szam) REFERENCES
        konyv(konyv_cim,konyv_iro, kiadas_szam),
 )
```

Ennek a táblának tartalmaznia kell a kikölcsönzött könyve elsődleges kulcsát, valamint annak a személynek az adatait, aki az kikölcsönözte a könyvet. Ezeket külső kulcsként kell szerepeltetni a táblában. Ennek a táblának az elsődleges kulcsát, az összes attribútum alkotja együttesen. Azért van erre szükség, mert így el lehet kerülni, hogy egy felhasználó egyfajta könyvből több példányt is kivehessen.

A 4. táblám szerkezete szinte teljesen azonos az előző tábláéval. Annyi különbség lesz a két tábla között hogy egy plusz attribútum segítségével el kell tárolnom, hogy könyv kikölcsönzésről, vagy könyv könyvtárba történő visszajuttatásáról van-e szó Ebben a táblában csak az adatbázishoz tartozó dokumentálást végzem el.

3.6 GUI tervezésénél figyelembe vett szempontok

A felhasználói felületről oldalakat lehetne írni. Egy jó felhasználói felületnek figyelembe kell venni az ember képességeit, határait, valamint az emberek természetét. A általam írt programot a felnőtt, de legalábbis, olyan felhasználóknak szánom akik, már az általam írt szoftverben a funkcionalitást keresik. A szoftvert nem szórakoztatás céljából készítem. Célirányosan a mindennapi munka megkönnyebbítésére szolgál a program. A szoftver használatával otthonról, a számítógép előtt ülve kölcsönözhetünk ki virtuálisan könyveket, és később személyesen vehetjük át a könyvtárban.

Az oldal szerkesztése közben, igyekeztem figyelembe venni, hogy az ember egyszerre 6-8 dolgot tud tárolni a rövid távú memóriájában, éppen ezért igyekeztem az oldalakat nem túlzsúfolni.

A színek összeállításánál figyeltem rá hogy ne válasszak rikító színeket, valamint hogy a színek a lehetőségeknek megfelelően, passzoljanak egymáshoz. A zöld szín nyugtató hatású az ember számára. A kék szín egy hideg szín, azonban a zölddel kombinálva tetszetős.

Különös figyelmet fordítottam arra, hogy a színekhez nem rendeltem funkcionalitást. Ennek megfelelően az alkalmazás felülete lényegesen könnyebben átlátható, és bizonyos hátrányos helyzetű emberek is könnyebben eligazodnak a rendszer használata folyamán. Ennek ellenére, nem volt célom, hogy betartsam a WAI (Web Accessibility Initiative) ajánlásait, amelyek betartása esetén az oldalt akadálymentesnek lehetne nevezni.

A lapok tervezésekor, megpróbáltam azonos stílusban felépíteni az oldalak szerkezetét. Ennek megfelelően mindig ugyanolyan kinézetű gombokat, és ismétlődő menük esetén, ugyanoda elhelyezni azokat. Ezáltal is barátságosabbá, és ismerősebbé tenni az oldalt.

A felhasználói felület szerves részét képezi a *help* menü, habár a rendszer használata nagyon egyszerű, A felhasználó megtalálhatja benne a lehetséges hibaüzeneteket, és egyéb segítségeket.

4 Implementáció

4.1 JSP (Java Server *Pages*)

Az E-Library egy 3 rétegű alkalmazás, amelyben mint minden más 3 rétegű alkalmazásban, megtalálható a 1. szinten az adatbázis, második szinten az üzleti logika, míg harmadik szinten a felhasználói felület. Ennek a 3 rétegű modellnek köszönhetően, teljes mértékben szétválasztható egymástól ez a három szint fejlesztése. Az oldalak grafikus megjelenítését készíthetik más programozok, esetleg designerek, akik semmit nem tudnak az üzleti logika implementációjáról. A Java Server Pages egy klienstől érkező kérés alapján valamilyen szöveges, általában HTML vagy XML formátumú dokumentum dinamikus, szerveroldali előállítására szolgáló technológia. A JSP nagyon hasonló a Servletekhez, de ezt majd még később be is fogom mutatni.

A JSP oldalak futtatásához, szükségünk van egy webszerverre. A saját programomhoz a NetBeans 6.0.1 verzióba beépített GlassFish V2 alkalmazásszervert használom. A választásom, azért esett erre, mivel az általam használt fejlesztői környezet alapértelmezett módon tartalmazza, és ingyenes. A GlassFish-közösség és a Sun Microsystems közös fejlesztése során került a piacra ez a nyílt forráskódú termék.

A JSP szoros kapcsolatban áll a servletekkel. Amíg a serveletekben a Java kódba beágyazva szerepelnek a weboldal kinézetét meghatározó utasítások, addig a JSP oldalak teljesen fordítottan működnek. Ebben az esetben a weblap HTML vagy XML kódjába ágyazott Java kódról beszélhetünk.

A JSP oldalban a következő nyelvi elemek szerepelhetnek: :

- HTML kód. Az oldal statikus megjelenítéséért felelős kódrészlet.
- Direktívák: A direktívák a JSP konténernek szóló utasítások.
- Szkript-elemek: A JSP oldal tartalmazhat szkripteket. Pl.: Java Szkript
- Megjegyzés: Természetesen tetszőleges helyen, és mennyiségben elhelyezhető megjegyzés a JSP oldalakon belül. Ezek olyan kódrészletek, amelyet a JSP-fordító nem értelmez, és kihagy az oldal megjelenítésekor. Olyankor lehet hasznos ha többen is fejlesztenek egy oldalt.

  Pl.: *<%-- Ez egy megjegyzés --%>*
- Akcióelemek: Ilyenek lehetnek a következők: useBean, setProperty, getProperty, include, és még egyéb akcióelemek, köztük a programozó saját maga által írt akcióelemek.

A direktívák a JSP konténernek szóló utasítások. A direktívák nem módosítják az előállított oldal

szövegét. 3 -féle direktívát használhatunk a JSP oldalunkban. Ezek az include, page, illetve taglib direktívák.

Az inlcude direktíva müködése triviális,egy adott fájl tartalmát fordítás nélkül byte-ról byte-ra másolja be a tartalmazó oldalba. Ezzel, lehet például könnyedén kiemelni egyes olyan részleteket, amelyek több oldalon is megjelennek, így elegendő egy helyen módosítani, ha változtatni akarunk a kódon. Egyetlen paramétert vár, amely egy fájl, amelynek a típusa tetszőleges lehet. Egy példa a használatára az index.jsp oldal kódjából :

```
<%@ include file="include/_top.html" %>
```

ahol az include könyvtár _top.html nevű fájlt, beillesztettem a nyitó oldal kódjába.. A _top.html egy egyszerű html kódot tartalmazó fájl, amely a program külalakját határozza meg. Olyan kódrészleteket gyűjtöttem ki, amelyet minden oldalon használnom kell. A _top.html böngészőben megjelenítve a képen látható.

A webes alkalmazás minden oldalának alsó része is azonos. Szintén egy különálló fájlba kiemeltem a szükséges kódrészletet. Ebben a fájlban már használtam Java kódot, ezért ezt már nem egyszerű HTML-lapként mentettem, hanem JSP oldal formátumban. Ennek az oldalnak a beillesztése is ugyanolyan módon történt:

```
<%@ include file="include/_down.jsp" %>
```

A page direktívákkal a teljes oldalra vonatkozó jellemzőket adhatunk meg. A page direktívák az oldal forrásán belül bárhol elhelyezkedhetnek, de az import direktívát leszámítva, csak egyszer kaphatnak értéket. Például ezek segítségével lehet változtatni az oldal karakterkódolását, vagy szükséges osztályok importálását elvégezni. A most következő sor segítségével az egész lapra beállítottam az UTF-8-as karakterkódolást, illetve beimportáltam a java.util csomagban található eszközöket.

```
<%@ page import="java.util.*"  contentType="text/html" pageEncoding="UTF-8"%>
```

A JSP oldal betöltődésének mechanizmusa a következő:

(1) A kliens oldaláról, érkezik egy kérés a szerver irányába.

(2) A web- vagy alkalmazásszerver a .jsp kiterjesztés alapján felismeri, hogy egy JSP fájlra vonatkozik a kérés, így továbbítja azt a JSP konténer, ami lehet a webkiszolgáló része, vagy külön Plugin.

(3) Mivel ez volt az adott dokumentumra vonatkozó első kérés, a JSP fordító a .jsp forrásból sorról-sorra haladva előállítja a neki megfelelő java szervlet forrását.

(4) A java kódot a javac fordítóval lefordítja egy .class fájlba

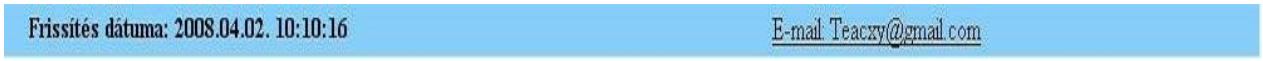
(5) Inicializálja a szervletet, majd a szervlet a kérést megkapva előállítja az oldal végleges szövegét

(6) Ezután a válasz visszajut a klienshez, és megjelenik a böngészőben.

Az itt leírt mechanizmus az Útikalauz Java programozóknak című könyvből származik. Ahogy a fentiekből is kitűnik a fordító minden egyes .jsp fájlból előállít egy szervletet. Éppen emiatt a fordítás miatt amikor betöltünk egy jsp oldalt az első hívás alkalmával lassabban töltődik be. Többszöri hívás esetén viszont, mivel már megvannak a .class fájlok lényegesen gyorsabb lesz, és azonnal válaszol a hívásra.

Terjedelmi problémák miatt nincs rá mód, hogy minden egyes általam írt jsp oldalt bemutassak, ezért csak 1-2 lényegesebb bemutatására szorítkozom.

Először is az általam már emlegetett _down.jsp oldal kódját, abból készített szervlet kódját, valamint a kinézetét mutatom be. Minden oldalam alján szintén megjelenik a itt látható kép.



Az oldal kódja nagyon egyszerű:

```
<%@ page import="java.util.*" %>
<HR ALIGN=center SIZE=8 WIDTH=1100>
<TABLE cellSpacing=0 cellPadding=4 width="100%" border=0>
 <TBODY>
 <TR bgColor=lightskyblue>
 <TD style="text-valign: middle"><TD>
   <DIV align=left style="font-size:11pt">
    <b>Frissítés dátuma: <%= (new java.util.Date() ).toLocaleString() %></b>
 </DIV>
</TD>
 <TD>
   <DIV align=left><a class=menu href=mailto:Teacxy@gmail.com>E-mail:
Teacxy@gmail.com</a >
 </DIV></TD>
 </tr>
 </tbody>
```

```
</table>
```

Látszik a rövidke kód alapján is, hogy nem csak html utasításokat tartalmaz. A <% %> között megadott utasítások, Java utasítások. Az első sorban a java.util csomag minden eszközét beimportálom, amelyre az aktuális dátum kiíratásakor van szükségem. A kód lényegében csak annyit csinál, hogy egy vékony sávban kiírja időt, amikor a lap betöltődött, valamint az e-mail címem.

A dátum kiíratásához a java.util.Date osztályból példányosítottam, és annak a toLocalString() metódusát meghívva írattam ki az aktuális dátumot. A fenti kódból, a fordító által készített szervlet kódja a következő:

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import java.util.*;

public final class _005fdown_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

  private static final JspFactory _jspxFactory = JspFactory.getDefaultFactory();

  private static java.util.Vector _jspx_dependants;

  private org.apache.jasper.runtime.ResourceInjector _jspx_resourceInjector;

  public Object getDependants() {
    return _jspx_dependants;
  }

  public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException {

    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    JspWriter _jspx_out = null;
    PageContext _jspx_page_context = null;
```

```
    try {
      response.setContentType("text/html");
      pageContext = _jspxFactory.getPageContext(this, request, response,
                                  null, true, 8192, true);
      _jspx_page_context = pageContext;
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
      _jspx_out = out;
      _jspx_resourceInjector = (org.apache.jasper.runtime.ResourceInjector)
application.getAttribute("com.sun.appserv.jsp.resource.injector");

      out.write(" \r\n");
      out.write("\r\n");
      out.write("<HR ALIGN=center SIZE=8 WIDTH=1100>\r\n");
      out.write("<TABLE cellSpacing=0 cellPadding=4 width=\"100%\" border=0>\r\n");
      out.write("<TBODY>\r\n");
      out.write("<TR bgColor=lightskyblue>\r\n");
      out.write("<TD style=\"text-valign: middle\"><TD>\r\n");
      out.write("<DIV align=left style=\"font-size:11pt\"><b>Frissítés dátuma: ");
      outsprint( (new java.util.Date() ).toLocaleString() );
      out.write("</b>\r\n");
      out.write("</DIV></TD>\r\n");
      out.write("<TD>\r\n");
      out.write("<DIV align=left><a class=menu href=mailto:Teacxy@gmail.com>E-mail:
Teacxy@gmail.com</a>\r\n");
      out.write("</DIV></TD>\r\n");
      out.write("</tr></tbody></table>\r\n");
      out.write("\r\n");
    } catch (Throwable t) {
      if (!(t instanceof SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
          out.clearBuffer();
        if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
      }
    } finally {
      _jspxFactory.releasePageContext(_jspx_page_context);
    }
  }
}
```

Az eredeti HTML kód szinte minden sora, bekerült valamilyen formába a *_jspService() metódus* törzsbe. A valamilyen forma alatt a következőket értem:

- Amennyiben Java kódról van szó akkor mindenféle változtatás nélkül, nyilván a <% %>-t elhagyva bekerül a metódus törzsébe.
- Az egyszerű HTML kódok pedig, minden esetben az out.write() metódus paramétereként. Minden egyes utasítás soronként.

Ezek alól kivételt a page direktívák jelentenek, amelyek a teljes oldallal kapcsolatban hordoznak információt. Jelen eseteben az (*<%@ page import="java.util.*" %>)* importálás egyszerűen bemásolódik a szervlet egyéb importálásai közé.

A továbbiakban mivel látszódik, hogy egy viszonylag rövidprogram kód is, ilyen hosszú szervletet von maga után, ezért több szervletet nem mutatok be, de minden egyéb bemutatott JSP oldalhoz tartozó szervletet, mellékelek a függelék részben.

A következő oldal amit bemutatok, egy a felhasználók regisztrációjánál szükséges adatlapot. A könyvek regisztrálására használatos oldal megjelenése, logikája, illetve a felhasznált eszközök teljesen hasonlóak lesznek, ezért azt be sem fogom mutatni.

```
<p>Új felhasználó regisztrálása a rendszerbe.</p><br/>
    <form action="ujFelhServlet" method="POST">
```

Az itt látható sorok segítségével, előbb egy egyszerű kiíratást végzek el, majd form-t hozok létre amely segítségével a forráskódban lentebb található gomb lenyomása következtében a vezérlés átadódik az *ujFelhServlet* szervletnek , amely az üzleti logikát implementálja a grafikus megjelenés alá. A szervlet feladata lesz a felhasználó adatainak ellenőrzése, és egyéb kritériumok ellenőrzése(nincs-e már ilyen regisztrált felhasználó), majd ha minden adat helyes, akkor új felhasználót regisztrál a rendszerben.

```
<table>
  <tr>
    <td>Vezetéknév: </td>
    <td><input name=vezetekNev type="text" width="15" value=
        "<%=vezetekNev==null?"":vezetekNev%>"></td>
  </tr>
```

Az itt látható forráskód egy egyszerű, 1 soros szövegbeviteli mezőt definiál. Ebben a sorban megjelenik néhány Jávában írt kód is. A <% %>-n belül látható kódot arra használom, hogy hibás adatbevitel esetén, amikor a felhasználó nem ad meg semmit a vezetéknév nevű mezőben, akkor a szervlet elindulásakor ne jelentkezzen NullPointerException kivétel. Ehelyett egy üres sztringet adok át. A következő elemek teljesen azonos elvűek az előbb leírtakkal.

```
<tr>
  <td>Keresztnév: </td>
  <td><input name="keresztNev" type="text" width="15" value="<
```

```
%=keresztNev==null?"":keresztNev%>"></td>
 </tr>
 <tr>
    <td>Felhasználó név: </td>
    <td><input name="email" type="text" width="15" value="<
%=email==null?"":email%>"></td>
 </tr>
```

Az itt látható jelszó beviteli mező annyiban különbözik az előzőektől, hogy mivel jelszóról van szó ezért az input mezőnek *password* típus van beállítva, amely következtében a jelszó beütéskor * jelenik meg az egyes betűk helyett, amely fontos a jelszó védelme miatt.

```
<tr>
    <td>Jelszó: </td>
    <td><input type="password" name="jelszo" type="jelszo" width="15"></td>
</tr>
```

Ennek az oldalnak ez az egyik legérdekesebb része. Egy *for* ciklus segítségével egy lenyíló ablakba 1900-tól 2007-ig bepakolom az évszámokat, így a felhasználónak majd csak egyszerűen ki kell választania a saját születési dátumát. A születés napját, illetve hónapját is hasonló módon oldottam meg.

```
<tr>
    <td>Szül. év:</td>
    <td><select name="szulEv" >
          <% for(int i=1900;i<2008;i++){ %>
          <option value="<%=i %>"><%=i %></option>
          <%}%>
    </select></td>
</tr>
<tr>
    <td>Szül. nap</td>
    <td><select name="szulHo">
          <% for(int i=1;i<=12;i++){ %>
           <option value="<%=i%>"><%=i%></option>
          <%}%>
    </select></td>
</tr>
<tr>
    <td>Szül. nap</td>
    <td><select name="szulNap">
          <% for(int i=1;i<32;i++){ %>
          <option value="<%=i%>"><%=i%></option>
          <%}%>
    </select></td>
```

```
</tr>
```

A következő beviteli ablakok, telefonszám, irányítószám, lakcím teljesen analóg módon készültek el mint a vezetéknév, vagy a keresztnév.

```
<tr>
   <td>Telefonszám: </td>
   <td><input name="telSzam" type="text" width="15" value="<
%=telSzam==null?"":telSzam%>"></td>
</tr>
<tr>
   <td>Irányítószám: </td>
   <td><input name="iranyitoSzam" type="text" width="4" value="<
%=iranyitoSzam==null?"":iranyitoSzam%>"></td>
</tr>
<tr>
   <td>Lakcím (város, utca, házszám): </td>
   <td><input name="cim" type="text" width="15" value="<%=cim==null?"":cim
%>"></td>
</tr>
 <tr>
```

A *form* vége előtt látható egy gomb, amely az *OK* gomb. Ez a gomb a *form*-on belül helyezkedik el. Erre a gomb vonatkozik a *<form action="ujFelhServlet" method="POST">* sor. Amikor a felhasználó kitöltötte a lentebb látható adatlapot, és lenyomja ezt a bizonyos gombot akkor a vezérlés átadódik az *ujFelhServlet* névre elnevezett szervletnek.

```
<td><td>
      <input type="submit" value="OK" size="20">
      </td>
   </tr>
 </table>
</form>
<%@ include file="include/_down.jsp" %>
```

Végül a képen az elkészült, majd megjelenített oldal látható. Az oldalhoz a JSP-fordító által generált szervlet kódját nem illesztem be, mivel 4-5 oldal hosszú, és lényeges eltérés nincs a korábban bemutatott szervlethez képest. A szakdolgozat függelék részében, viszont beillesztem a JSP-fordít által készített szervletet.

A taglib direktívával JSP elemkönyvtárakat lehet használni. Paraméterként meg kell adni egy URL-t, ahol az elemkönyvtár leírófájlja található, valamint egy prefixumot, amivel később hivatkozni lehet az elemkönyvtár elemeire. Az előbbi oldalban is használok taglib direktívát, amely segítségével a bevitt adatok validálásánál előforduló hibákat, jelenítem meg ugyanazon az oldalon.

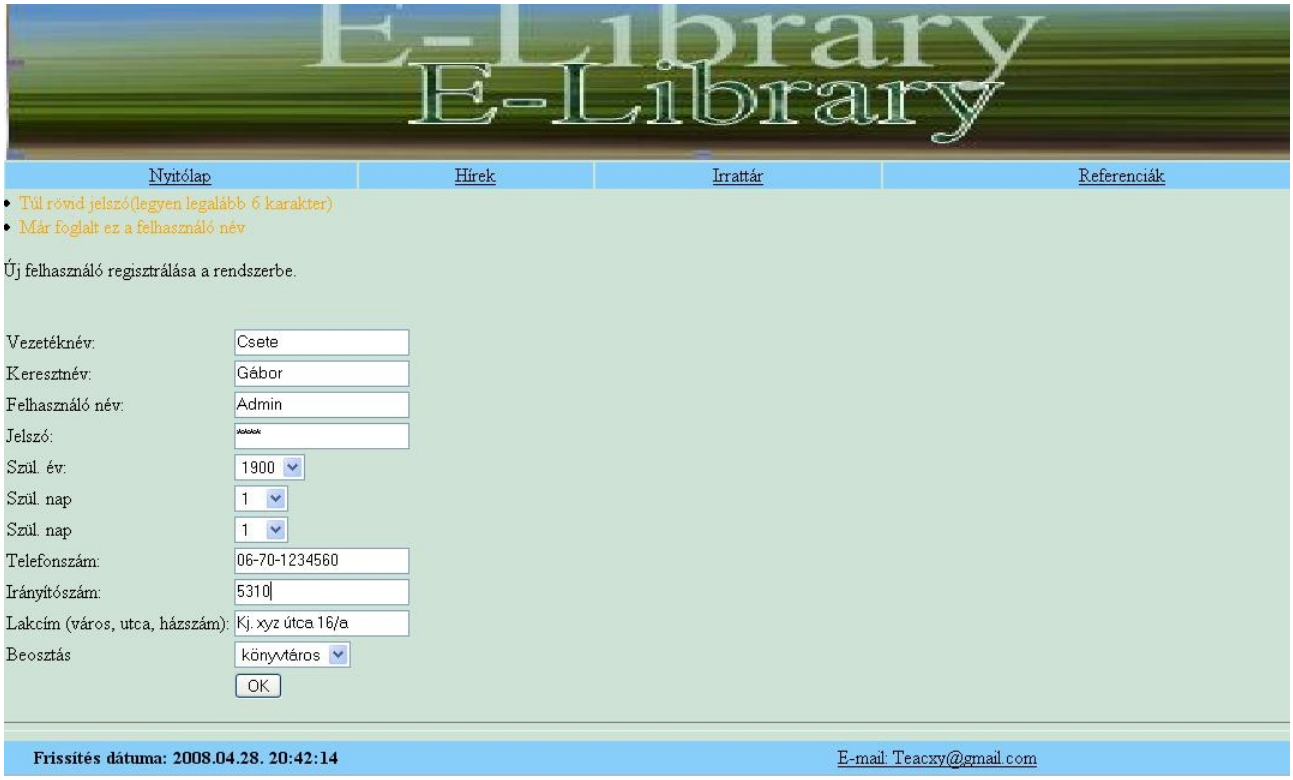
A taglib direktíva a következő: <%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

Ezek után nincs más dolgom, mint írni egy foreach ciklust, amely segítségével, a servlettől kapott hibalistát bejárom, és kiíratom. A kód a következő:

```
<c:forEach items="${hibaLista}" var="hiba">
    <li><font style="color: orange">${hiba}</font></li>
</c:forEach>
```

Az első betöltődéskor, semmit nem ír ki. Akkor fog kiírni, ha az OK gomb megnyomása után a regisztráció sikertelen. A servletben az adatok ellenőrzése során, ha valami nem jó (pl.: túl rövid jelszó, nem megfelelő irányítószám, foglalt név, vagy egyéb dolgok), akkor azt egy egyszerű ArrayList() típusú listába berakom, amelyet továbbítok a jsp oldalnak. A servletben a következő sorok segítségével lehet elküldeni a lista objektumot az uj_felh.jsp oldalnak:

```
request.setAttribute("hibaLista",lista);
        request.getRequestDispatcher("uj_felh.jsp").forward(request,response);
```



A könyvtárba történő, könyv regisztrálásoknál használt oldal teljesen analóg módon készült el az előbbi oldal segítségével.

4.2 Enterprise JavaBeans

4.2.1 JavaBeans

A JavaBeans, komponens alkalmazások összeállításakor használatos. Java nyelven való komponensalapú programozást segítő API. Elválik egymástól a programozó, és az alkalmazás

összeállító szerepköre. A programozó, az aki implementálja az adott komponenst, míg az assembler csak ezek felhasználásával állítja elő a kész szoftvert. Tulajdonképpen a JavaBeans komponensalapú újrafelhasználhatóságot hivatott szolgálni. Egy-egy bean-t akárhányszor újrafelhasználhatunk. A JavaBeans csak lokális komponensekben gondolkodik, és ezáltal nem használható elosztott rendszerek esetén, az üzleti logika implementálására.

4.2.2 Enterprise JavaBeans

Az Enterprise JavaBeans a Sun hivatalos megfogalmazása szerint, komponens alapú elosztott üzleti alkalmazások fejlesztését, és telepítését támogató architektúra specifikáció. Az Enterprise JavaBeans, a JavaBeans technológia kiterjesztése elosztott rendszerek számára. Több mindenben is különbözik a JavaBeans-ektől, mivel másra más környezetben is használják őket.. Az Enterprise JavaBeans-nek szüksége van egy úgynevezett Bean Container-re. A Bean Container-hez semmi köze nincs a programozó által írt Bean-ekhez, hanem egységes, szabványos környezetet biztosít a Bean-ek futásához.

4.2.3 Beanek-hez tartozó interfészek

- Távoli interfész (remote interace): A kliens, a bean osztályt nem közvetlenül hivatkozza, hanem a már említett konténer által generált objektumot képes elérni. A távoli interfészben azoknak a metódusoknak a specifikációját kell megadni, amelyeket a kliens által elérhetővé akarunk tenni. Így lehet szabályozni, hogy a kliens oldal mely üzleti metódusokat érhesse el.
- Lokális interfész (local interface): Az üzleti logika valamely metódusának meghívása nagyon költséges lenne, amennyiben távoli interfészen keresztül használnánk (hálózati költségek). Ha az EJB és kliense, illetve a többi EJB közös alkalmazásszerveren fut, a költségeket csökkenteni tudjuk. Erre találták ki a lokális interfészt. A lokális interfésznek a bean minden üzleti metódusának specifikációját tartalmaznia kell.
- Home interfész: A konténer által generált lokáli otthon objektum generálásához szükséges interfész, amely a beanek felügyeletéhez tartozó metódusokat tartalmazza. (pl.: létrehozás)

Az EJB 3.0-tól kezdve a programozónak nem kell foglalkoznia, ezen interfészek implementálásával, és frissítésével, mert legenerálja ezeket az annotációk segítségével a fordító, ezáltal elég egy osztály írnunk nem kell 4-5 osztályra figyelni.

4.2.4 Telepítési leírás (deployment descriptor)

A telepítési leírásban adjuk meg a bean-ek belső struktúráját leíró információkat, valamint az alkalmazás összeállítása során beállított implicit alkalmazásszerver szolgáltatások paramétereit.
Mindig tartalmazza a következőket:

- A Bean nevét, illetve az implementáló osztály nevét (<ejb-class> a bean osztály neve

</ejb_class>)

- A bean-hez tartozó (létező) interfészek nevét (pl.: <remote> távoli interfész neve </remote>)
- A beanek típusát (entity, session)
- Session bean esetén állapottal rendelkezik-e vagy sem
- Session bean: tranzakciókezelésére vonatkozó információkat
- Entity bean perzisztenciakezelésére vonatkozó információkat
- Entity bean elsődleges kulcs-osztályát
- Szükséges környezeti bejegyzések
- Biztonsági szerepkör

A telepítési leírás XML formátumú állomány, amelynek a kezelése létfontosságú az EJB használata esetén. A korábbi időkben a programozónak kellett ezt a fájlt is kezelnie. Szerencsére az EJB 3.0 megjelenésétől kezdve, a telepítési leírások helyett annotációkat használunk, amely segítségével könnyebben programozható, beállítható.

4.2.5 EJB Konténer

Az beanek példányai minden esetben az EJB konténerben jönnek létre, és dinamikusan, futásidőben az EJB konténer segítségével menedzselődnek. A konténer biztosíthat speciális szolgáltatásokat, de amennyiben az adott bean nem használ ezek közül egyet sem, akkor a konténertől függetlenül telepíthető. Amennyiben viszont felhasznál valamilyen speciális szolgáltatást, akkor csak olyan konténerbe telepíthető, amely nyújtja az adott szolgáltatást.

A konténer feladatai közé tartozik a bean lokális, illetve távoli elérésének publikálása is. Ezeket az információkat a fentebb említett lokális és távoli interfészek tartalmazzák. Ezeknek az interfészeknek meg kell felelnie EJB specifikációjának.

A konténer végzi továbbá a komponensek példányosítását, megszüntetését, tranzakció kezelést, és a perzisztenciát is.

4.2.6 Beanek típusai

4.2.6.1 Entity Bean

Az entitás beanek segítségével perszisztens objektumokat kezelhetünk a memóriában. Az E-Library esetén is használnom kell több entitást. Ilyenek lesznek a felhasznál entitás, vagy például a könyv entitás is. Az entitás beanek élettartama nem kötődik a kliens élettartamához, ezáltal túlélhet szerverösszeomlást is. A memóriában tárolt entitásokat leképezhetjük relációs adatbázisba. Ennek módja lehet, hogy explicit módon leprogramozzuk, avagy az erre célra megfelelő automatikus eszközt használunk. Én a TopLink nevű eszközt használom erre a célra. Az entitás beanek implementációja több különböző fájlból állhat. Minden entitás bean esetén létezik az entitás beant

tartalmazó enterprise bean osztály, valamint a különböző hozzátartozó interfészek.(pl.: otthon, távoli, lokális interfészek) Gyakran tartozik egy plusz osztály az entitás beanek reprezentációjához. Ez az osztály az elsődleges kulcsot reprezentáló osztály. Az elsődleges kulcs az adott entitás azonosítására szolgál. Bármilyen osztály megfelel erre célra, amennyiben teljesíti a következőket:

- implementálja a Serializable interfészt
- implementálja a hashCode()
- illetve a equals() metódusokat.

Az entitás beaneknek 2 fajtája van:

- CMP(Container Managed Persistence):

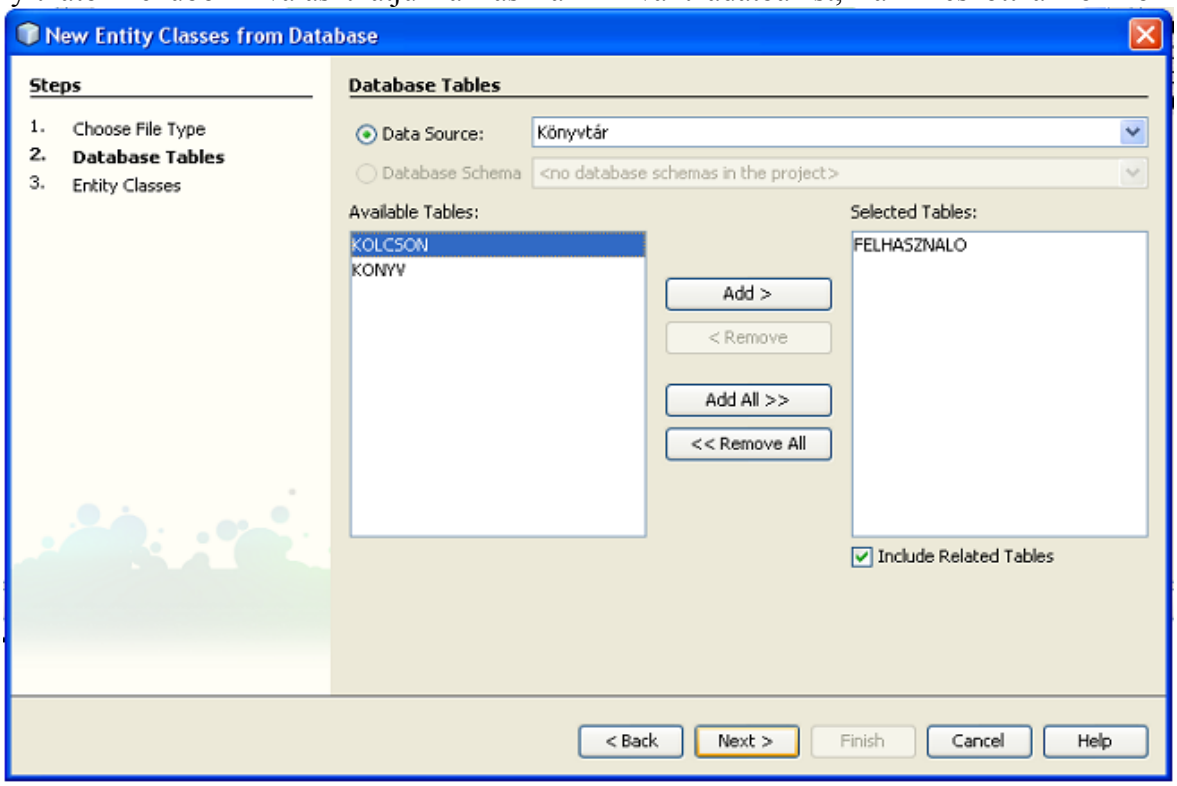
  A konténer által kezelt perzisztenciát esetén a konténer valósítja meg a bean leképezését a tárba (általában adatbázisba). Nem szükséges a perzisztenciával foglalkozni, a konténer legenerálja a működéshez szükséges metódusok implementációját. A CMP-nek több előnye is van:

  - jóval áttekinthetőbb a kód
  - programozói munkát spórolhatunk meg, amely gyorsabb fejlesztést eredményezhet
  - Perzisztens tártól független kódot készíthetünk, így egy másik adatbázisra való csatlakozás, a kód módosítása nélkül is megtehető.
- BMP(Bean Managed Persistence):

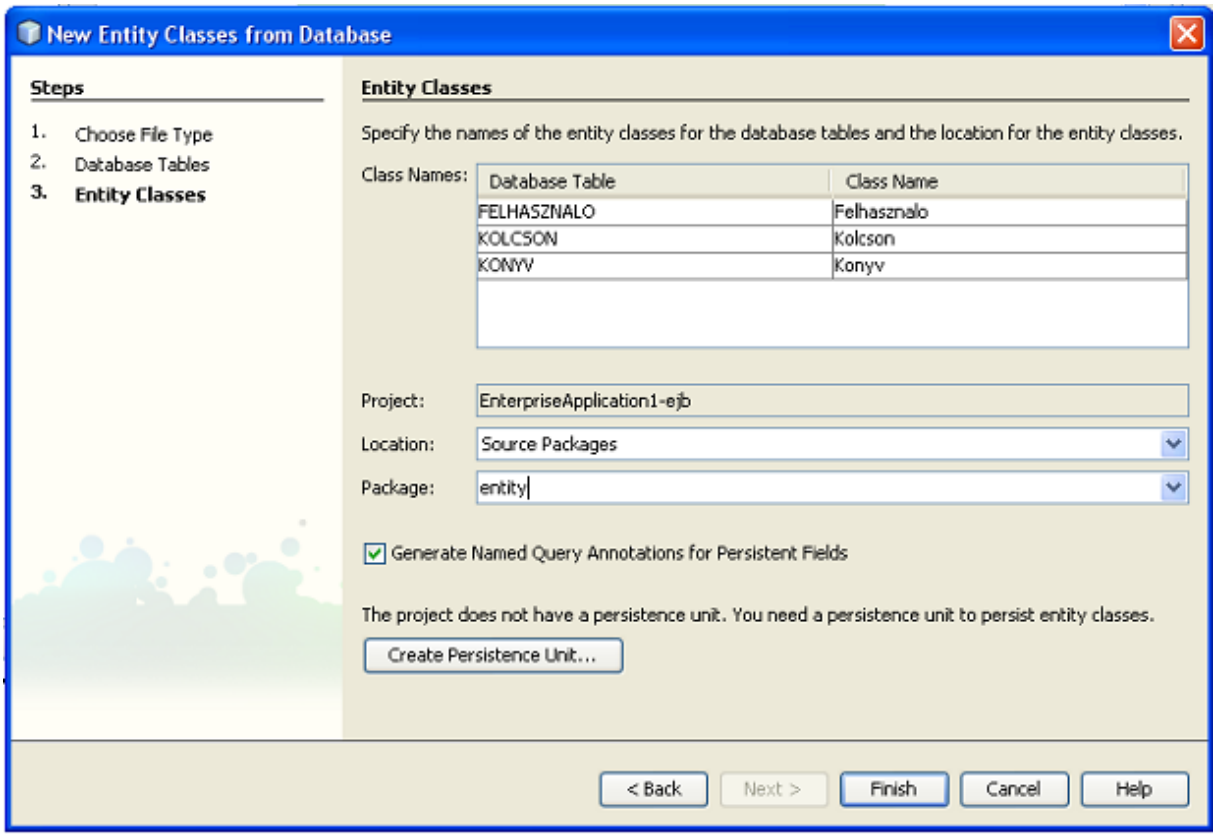
  A BMP esetén a bean gyártója implementálhatja a bean perzisztenciáját. A BMP esetén a programozónak kell implementálnia a perzisztencia kezelést. Ennek következtében konkrét JDBC hívásokra van szükség. A programozónak kötelezően implementálnia kell néhány metódust, mint például ejbCreate(), ejbRemove(), ejbFind(), ejbLoad(), és még egy pár metódust. A BMP-t akkor célszerű használni, amikor olyan szolgáltatásokra van szükségünk amelyeket a CMP nem képes teljesíteni. Ilyen lehet, amikor egy entitást több táblában szeretnénk tárolni, vagy több adatbázist használunk valamilyen okból. A BMP segítségével mindent tetszőlegesen testre szabhatunk, azonban programozói többlet munkát eredményez a CMP-hez képest. Hátránya még, hogy adatbázisok változása esetén, az egész implementációt elölről kell kezdeni, mivel csak adott esetekre képes működni.

Természetesen a saját programomhoz is el kellett készíteni néhány egyedtípust. Ilyenek például a felhasználó egyedtípus, vagy a könyv egyedtípus, de adatbázisban tárolom a kölcsönzés, és a logolást is, ennek megfelelően elkészítettem ezt a 2 további entitást is.

A korábban említett módon létrehoztam az adatbázisban a megfelelő táblákat. Az entitásoknak az adatbázisban tárolt entitásokkal illeszkedniük kell. Az általam használt NetBeans nagyon kényelmes eszközt biztosít az entitás beanek létrehozásához. El lehet végezni automatikusan az entitás beanek legeneráltatását az adatbázis tábláknak megfelelően, és ezzel rengeteg programozói munkát spórolhatunk. A saját adatbázis tábláimból a következő módszerrel lehet egyszerűen legyártatni az entitás beaneket. Az E-Library-ejb projekt nevén jobb gombot lenyomva megjelenő menüben a New /Entity Classes from Database menüpont segítségével. A felugró ablakban a Data Source lenyitható menüből kiválaszthatjuk a használni kívánt adatbázist, ha nincs ott akkor hozzá kell



adnunk a lenyitható menüpont utolsó opciója segítségével. Kiválasztjuk azon egyedeket (bal oldalon), amelyből generáltatni szeretnénk, és a jobb oldalra visszük. Ezután a Next gomb lenyomásával a következő fázisba érünk.



Ezen a panelen kiválaszthatjuk, hogy mely csomagba helyezkedjenek el. Itt lehet perzisztens egységet létrehozni, az entitás beanekhez, de én azt már korábban megtettem. A finish gombra kattintva elkészülnek az entitás osztályok. A könyv táblából létrejött bean osztályt egyes részleteit mutatom be:

```
package entity;
```

A Netbeans automatikusan elvégezte a szükséges osztályok beimportálását.

```
import java.io.Serializable;
import java.util.Collection;
import javax.persistence.*;
/**
 * @author Csete Gábor
 */
```

Az EJB 3.0-nak megfelelő annotációk is legenerálásra kerültek.

```
@Entity
```

A fentebb látható annotáció jelzi a fordító számára, hogy egy entity bean-ről van szó.

```
@Table(name = "KONYV")
@NamedQueries(
{@NamedQuery(name = "Konyv.findByKCim", query = "SELECT k FROM Konyv k WHERE
k.konyvPK.kCim = :kCim"),
 @NamedQuery(name = "Konyv.findByKIro", query = "SELECT k FROM Konyv k WHERE
k.konyvPK.kIro = :kIro").....
```

)

A különböző lekérdezési lehetőségeket is automatikusan legenerálta a NetBeans. Még több ilyen annotáció is volt a forráskódban, de azok teljesen hasonlóan néznek ki, ezért rövidítettem le.

```
public class Konyv implements Serializable {
    private static final long serialVersionUID = 1L;
    @EmbeddedId
    protected KonyvPK konyvPK;
    @Column(name = "KIADO")
    private String kiado;
    @Column(name = "KIADAS_EVE")
    private Short kiadasEve;
    @Column(name = "DARABSZAM")
    private Short darabszam;
    @Column(name = "AKT_SAZM")
    private Short aktSazm;
    @OneToMany(mappedBy = "konyv")
    private Collection<Kolcson> kolcsonCollection;

    public Konyv() {    }

    public Konyv(KonyvPK konyvPK) {
        this.konyvPK = konyvPK;    }

    public Konyv(String kCim, String kIro, short kiadasSzam) {
        this.konyvPK = new KonyvPK(kCim, kIro, kiadasSzam);    }

    public KonyvPK getKonyvPK() {
        return konyvPK;    }

    public void setKonyvPK(KonyvPK konyvPK) {
        this.konyvPK = konyvPK;    }

    public String getKiado() {
        return kiado;    }

    public void setKiado(String kiado) {
        this.kiado = kiado;    }
```

```
public Short getKiadasEve() {
    return kiadasEve;    }

public void setKiadasEve(Short kiadasEve) {
    this.kiadasEve = kiadasEve;    }

public Short getDarabszam() {
    return darabszam;    }

public void setDarabszam(Short darabszam) {
    this.darabszam = darabszam;    }

public Short getAktSazm() {
    return aktSazm;    }

public void setAktSazm(Short aktSazm) {
    this.aktSazm = aktSazm;    }

public Collection<Kolcson> getKolcsonCollection() {
    return kolcsonCollection;    }

public void setKolcsonCollection(Collection<Kolcson> kolcsonCollection) {
    this.kolcsonCollection = kolcsonCollection;    }

@Override
public int hashCode() {
    int hash = 0;
    hash += (konyvPK != null ? konyvPK.hashCode() : 0);
    return hash;    }

@Override
public boolean equals(Object object) {
    // TODO: Warning - this method won't work in the case the id fields are not set
    if (!(object instanceof Konyv)) {
        return false;
    }
    Konyv other = (Konyv) object;
    if ((this.konyvPK == null && other.konyvPK != null) || (this.konyvPK != null && !
```

```
            this.konyvPK.equals(other.konyvPK))) {
            return false;
        }
        return true;    }

    @Override
    public String toString() {
        return "entity.Konyv[konyvPK=" + konyvPK + "]";    }
}
```

Jól látható a forráskód alapján, hogy a Neatbeans minden beállító, és lekérdező metódust legenerált. Ezenkívül legenerálta a hashCode(), equals(Object object), és a toString() metódusokat. Az egyetlen érdekesség amely miatt a könyv entitást mutatom be, az hogy a könyvnek az elsődleges kulcsa összetett. Ezt a NetBeans is észrevette, és ennek megfelelően létrehozott egy, az elsődleges kulcsnak megfelelő osztályt. A KonyvPK is egy hasonló szerkezetű osztály az előbbhez. Tartalmazza az elsődleges kulcsot felépítő attribútumokhoz tartozó beállító és lekérdező metódusait. Ami érdekes lehet benne az a következő annotációk.

```
    @Column(name = "K_CIM", nullable = false)
    private String kCim;
    @Column(name = "K_IRO", nullable = false)
    private String kIro;
    @Column(name = "KIADAS_SZAM", nullable = false)
    private short kiadasSzam;
```

Itt jelzi a fordítónak, hogy melyik változó a tábla melyik oszlopának felel meg, valamint azt hogy ezek az értékek nem lehet null értékűek, mivel az elsődleges kulcs részét képezik.

Amint látható az entitás beanek létrehozása (adatbázisból) nagyon egyszerű. A programozónak nincs más dolga mint felhasználja az entitásokat. Természetesen létre lehet hozni entitás beaneket kézzel is, és néha lehet is rá szükség.

4.2.6.2 Session Bean

A session beanek az üzleti logika implementálására használható, újrafelhasználható komponens. A session beanek segítségével tudjuk megvalósítani a üzleti logikát, persze szinte minden esetben az entitás beanek felhasználásával. A session beanek élettartama lényegesen kisebb mint a entitás beaneké. Általában egy session bean élettartama egyenlő az őt hívó metódus élettartamával. Például amikor kikölcsönzünk egy könyvet a könyvtárból, akkor meghívódik a kölcsönöz nevű session bean, és amint lefutott, meg is szűnik. A konténer feladata, hogyha egy kliens meghívja a session bean valamely metódusát, akkor az adott session bean példányhoz ne férhessen hozzá másik kliens.

Ilyenkor a konténer a kérést egy másik példányhoz irányítja ha van. Ha nincs szabad példány, akkor hozhat létre újat is, vagy valamilyen más úton kezeli le. Kétfajta session bean létezik attól függően, hogy a 2 hívás között tárolják-e a kapcsolat állapotát.

4.2.6.2.1 Állapotmentes session bean

Az olyan üzleti folyamatok implementálására való amelyek esetén nem szükséges, hogy a hívások között megőrződjön a hívás állapota. Ilyen lehet például, a könyvtár program esetén, a könyvek, vagy új ember regisztrációja a rendszerben. Két könyv regisztrációja között nem kell semmilyen állapotot megjegyezni. Ezekből következően, az állapotmentes session bean metódusainak meghívásakor minden paramétert át kell adni. Én a Netbeans 6.0.1 által is támogatott EJB3.0-t használok amelyek használata sokkal egyszerűbb mint a korábbi verziók. Kevesebb teher hárul a programozóra. Az annotációk segítségével lényegesen gyorsabban lehet fejleszteni. Elsőnek a felhasználó entitáshoz generáltatott session beant mutatom be. A kódot szintén a NetBeans segítségével készítettem. Az Netbeans segítségével létre lehet hozni új session beaneket a már létező entitás beanekhez a File\New\Session Bean for EntityClasses menüpont alatt. A felugró ablak segítségével, grafikusan eszköz segítségével hozhatunk létre session beant. Beállíthatjuk hogy melyik entitáshoz szeretnénk létrehozni session beant, és hogy milyen interfészek jöjjenek létre vele együtt. Az említett kód a következő:

```
package session;
A forráskód elején a szokásos importálások láthatók.
import entity.Felhasznalo;
import java.util.List;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
/**
 * @author Csete Gábor
 */
```

A @Stateless annotáció segítségével jelezzük a fordítónak, hogy állapotmentes session beanről van szó. A FelhasználoFacade osztály implementálja a FelhasznaloFacadeLocal lokális interfészt. A lokális interfész tartalmazza az osztályban szereplő publikálandó metódusokspecifikációját.

```
@Stateless
public class FelhasznaloFacade implements FelhasznaloFacadeLocal {
    @PersistenceContext
    private EntityManager em;
```

Az entitások állapota elmenthető egy adattárba, úgy hogy bármikor visszaállítható. Az entitás

tekinthető perzisztens objektumnak is, de nem teljesen egyezik meg a két fogalom. A perzisztens objektum születése pillanatától perzisztens, míg az entitás csak perzisztenssé tehető. Az entitás perzisztenssé tételéhez az alkalmazás beavatkozására van szükség, vagyis az entitás perzisztenciája felett teljes mértékben az alkalmazás rendelkezik

A perszisztens objektum használatához szükségünk van egy Perszistence Unit létrehozására. A perzisztens unit beállításait a persistence.xml fájlban nézhetjük meg.A Java Persistence Api használatához továbbá szükségünk van egy J2EE 5 kompatibilis alkalmazásszerverre. A NetBeans által automatikusan telepített GlassFish V2 alkalmazásszerver teljesíti ezt a kitételt is. A Glassfish projekt fejlesztőinek meg kellett valósítania a szabvány Java Persistence Api-ra vonatkozó részét is, így készült el a „TopLink(TM)" kereskedelmi ORM (Objektum relációs leképezés) termék.

A Javax.persistence csomag tartalmazza a JPA által használt annotációkat

A fentebb látható Entity Manager szolgál arra, hogy segítségével elvégezzük a perszisztens objektumok memóriában történő kezelését.

Például a következő create(Felhasznalo felhasznalo) metódus segítségével hozhatunk létre új felhasználót az adatbázisban. Mint látható nem csinál semmi mást ez a pár sor, mint a paraméterként megkapott Felhasznalo típusú objektumot az em.persist(felhasznalo) metódus segítségével létrehozza. Teljesen elfedve az adatbázis kezelést a programozó elől. Az Entity Manager megoldja helyettünk, így nem kell JDBC hívásokat bajlódnunk.

```
public void create(Felhasznalo felhasznalo) {
    em.persist(felhasznalo);
}
```

A következő metódusok, ahogy a nevükben is benne van, az felhasználó entitás adatbázisban való módosítására, eltávolítására, keresésére vonatkozó metódusok. A keresés az elsődleges kulcsérték alapján megy.

```
public void edit(Felhasznalo felhasznalo) {
    em.merge(felhasznalo);
}

public void remove(Felhasznalo felhasznalo) {
    em.remove(em.merge(felhasznalo));
}

public Felhasznalo find(Object id) {
    return em.find(entity.Felhasznalo.class, id);
}
```

Az utolsó metódus segítségével az összes adatbázisban tárolt felhasználót kapjuk, meg egy

Felhasznalo típussal paraméterezett listában.

```
 public List<Felhasznalo> findAll() {
        return em.createQuery("select object(o) from Felhasznalo as o").getResultList();
    }
}
```

Az előző fájlhoz hozzátartozik a FelhasznalofacadeLocal nevű interfész, amelynek a kódját is szintén a NetBeans segítségével generáltattam le..

```
package entity;
import java.util.List;
import javax.ejb.Local;
@Local
public interface FelhasznaloFacadeLocal {
    void create(Felhasznalo felhasznalo);
    void edit(Felhasznalo felhasznalo);
    void remove(Felhasznalo felhasznalo);
    Felhasznalo find(Object id);
    List<Felhasznalo> findAll();
}
```

A következőkben egy már érdekesebbnek mondható session bean mutatok be, amelyre a könyvtárba való bejelentkezéskor használok.

```
package session;
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import entity.Felhasznalo;
import java.io.*;

/**
 * @author Csete Gábor
 */
@Stateless
public class FelhasznaloBeleptetesSessionFacadeBean implements
          FelhasznaloBeleptetesSessionBeanLocal {
    @PersistenceContext
    private EntityManager em;
    Felhasznalo felh ;
    boolean allapot;
```

Ezzel az osztállyal fogom elvégezni bejelentkezéskor a felhasználó azonosítását. A bejelentkezes(...) metódus paramétereket vár. A két paraméter a bejelentkezési név, illetve a hozzátartozó jelszó. A

megadott felhasználó nevet lekérdezzük az EntityManager segítségével. Ezek után megvizsgáljuk hogy a hozzátartozó jelszó megegyezik a paraméterként kapott jelszóval, és ez alapján beállítjuk az bejelentkezés állapotát a megfelelőre.

```
public boolean bejelentkezes(String felh, String jelszo) {
      Object o=(Felhasznalo)em.find(Felhasznalo.class,felh);
      if(o==null){
         allapot=false;
      }
      else if(o!=null && ((Felhasznalo)o).getJelszo().equals(jelszo)){
         allapot=true;
         this.felh=(Felhasznalo) o;
         return true;
      }
      else allapot=false;
      return false;
   }
   public String UserInfo() {
      return this.felh.toString;
   }
   public Felhasznalo getFelh() {
      return felh;
   }
```

Ez a metódus visszaadja a bejelentkezett felhasználó szerepkörét.

```
   public String getSzerep() {
      return felh.getSzerep();
   }
```

A kijelentkezes() metódus elvégzi a felhasználó logikai kijelentkezését.

```
public void kijelentkezes() {
      setFelh(null);
      setAllapot(false);
   }
     public EntityManager getEm() {
      return em;
   }
   public void setEm(EntityManager em) {
      this.em = em;
   }
   public void setFelh(Felhasznalo felh) {
```

```
        this.felh = felh;
    }
    public boolean getAllapor() {
        return allapot;
    }
    public void setAllapot(boolean allapot) {
        this.allapot = allapot;
    }
}
```

Ehhez az osztályhoz is szorosan hozzátartozik a lokális interfésze, amely teljesen analóg módon épül fent mint a korábban bemutatott esetben. Most ennek a session beannek a használatát mutatom be. Ehhez szükség van az őt hívó szervlet kódjára.

```
public class bejelentekezes extends HttpServlet {
```

Ez az egyik legfontosabb változás az EJB 3.0 és a korábbi verziók között. Az EJB 3.0-tól kezdve az enterprise bean-ek hívása sokkal könnyebben kezelhető, mint korábban volt. A @EJB annotáció segítségével, függőségi injektálást végezzük el.

```
    @EJB
    private FelhasznaloBeleptetesSessionBeanLocal user;

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
```

A bejelentkezéskor használatos jsp oldaltól lekérjük a megadott jelszó, illetve felhasználó nevet, és megpróbáljuk beléptetni a rendszer felhasználóját.

```
        if( user.bejelentkezes(request.getParameter("felNev"),
                request.getParameter("jelszo")))
        String szerep="";
```

A beléptetés után, amennyiben sikerül a belépés, megvizsgáljuk, hogy az adott személynek melyik jsp oldalt kell megjeleníteni, és a számára megfelelő oldalt jelenítjük meg, és a session.setAttribute("FelhBeleptetesLocal",user) segítségével elérhetővé tesszük a következő oldlak számára is user objektumot.

```
      if( user.bejelentkezes(request.getParameter("felNev"), request.getParameter("jelszo"))){

        String szerep="";
        if(user.isAllapot()){
            szerep=user.getSzerep();
        }
        HttpSession session=request.getSession(false);
```

```
if(szerep.equals("felhasznalo")){
    session.setAttribute("FelhBeleptetesLocal",user);
    request.getRequestDispatcher("ugyfel.jsp").forward(request,response);
}
else if(szerep.equals("inf_konyvtaros")) {
    session.setAttribute("FelhBeleptetesLocal",user);
    request.getRequestDispatcher("Informatikus_konyvtaros.jsp").forward(request,response);
}
else if(szerep.equals("admin")){
    session.setAttribute("FelhBeleptetesLocal",user);
    request.getRequestDispatcher("admin.jsp").forward(request,response);
}

}
else {
```

Ha probléma lép fel a bejelentkezés közben , akkor azt a korábban leírt taglib direktíva segítségével jelzem a felhasználó számára.

```
List lista=new ArrayList();
lista.add("Probléma a belépés folyamán, kérem válasszon a lehetőségek közül.");
 request.setAttribute("lista",lista);
    request.getRequestDispatcher("hibas_login.jsp").forward(request,response);
}
```

Ilyenkor a hibás_login.jsp oldal jelenítjük meg, ahol ismételten meg lehet próbálni a bejelentkezést.

4.2.6.2.2 Állapottal rendelkező session bean

A kapcsolat bean-ek másik típusa az állapottal rendelkező kapcsolat bean. A különbség az állapot nélküli típushoz képest , hogy a bean megőrzi a klienshez rendeltségét és egyedváltozóinak állapotát két hívás között. Amikor a kliens megszólít egy állapottal rendelkező session beant, akkor párbeszédet kezd vele, melynek az állapotát eltároljuk a beanben. Ennek az állapotnak mindig elérhetőnek kell lennie az egész kommunikáció alatt. Az állapottal rendelkező beanek esetén, minden egyes bean a memóriában egy-egy klienshez tartozik. Akkor van probléma, amikor egy több kliens próbál kommunikációt folytatni mint ahány példány lehet a memóriában. Ilyenkor kerül elő az aktivitás - passziválás technikája. Ha egy új példányra van szükség, akkor olyan megoldást alkalmazhatunk, hogy az éppen nem használt beant perzisztens módon eltároljuk a lemezen, vagy az adatbázisban, és majd csak akkor töltjük vissza amikor valóban szükségünk lesz rá.

5 Összefoglalás

A szakdolgozat elkezdése előtt, azt a célt tűztem ki magam elé, hogy megpróbáljak betekintést nyerni, egy általam még nem ismert webes technológiába, az által, hogy egy működő programot írok. A szakdolgozat, úgy gondolom, hogy többé-kevésbé elérte célját. A rendszerfejlesztés alatt, elsajátítottam a JSP oldalak, és az EJB használatát. Így végül elkészült az E-Library nevezetű program, amely az elektronikus könyvtáraktól elvárható alapfunkcionalitásokat nyújtja. A programot, ettől függetlenül „éles" helyzetben nem használnám, mivel továbbfejlesztésre szorul. A szoftverfejlesztés folyamata alatt, számos hibával, és nehézséggel találkoztam. Ezeket több kategóriába lehet sorolni. Voltak olyan nehézségek, amelyek a programozási környezettel voltak kapcsolatosak, hardveres hiba is nehezítette a fejlesztést, valamint természetesen számtalan programozásbeli probléma is felmerült a fejlesztés során. A szakdolgozat megírását követően több dolgot, másképp csinálnék. Nagy hangsúlyt fektettem a rendszer megtervezésére, mégis az adatbázisban tárolt entitások szerkezetén sokat változtatnék, valamint a designra is nagyobb figyelmet fordítanék.

A fejlesztés folyamán elkészült 19 darab jsp oldal, 4 az adatbázisban tárolt egyedekhez tartozó entitás bean, valamint 5 session bean, és természetesen 10-15 darab szervlet is.

A szakdolgozat végére érve fogalmazódott meg bennem a gondolat, hogy a programnak megpróbálom elkészíteni, egy újratervezett, bővített változatát, de mindezt már egy másik technológia segítségével.

Úgy gondolom, hogy a program megírása által szerzett tapasztalatok nagy előnyömre válhatnak, majd a közeljövőben. Szeretnék programozással foglalkozni a Programtervező informatikus szak elvégzése után.

6 Irodalomjegyzék

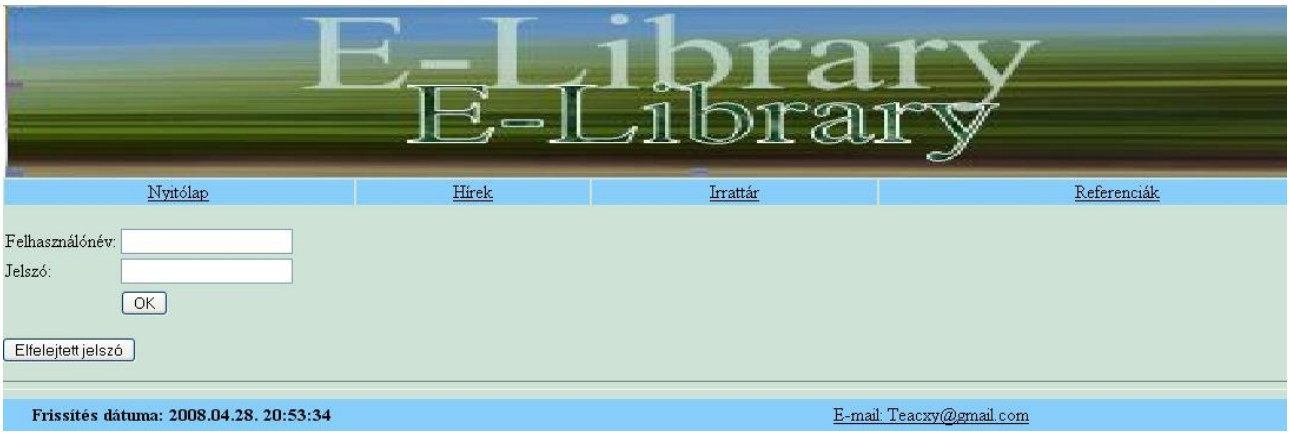
- Nyékyné Gaizler Judit Java2 Útikalauz programozóknak (1.3) , ELTE TTK Hallgatói Alapítvány
- Nyékyné Gaizler Judit J2EE Útikalauz Java programozóknak, ELTE TTK Hallgatói Alapítvány
- A Rendszerfejlesztés technológiája című előadás jegyzet (2007/2008)
- http://netbeans.org
- http://java.sun.com/
- The J2EE Tutorial (j2ee-1_3-doc-tutorial-draft3.pdf)
- Egyéb tutorialok

7 Függelék

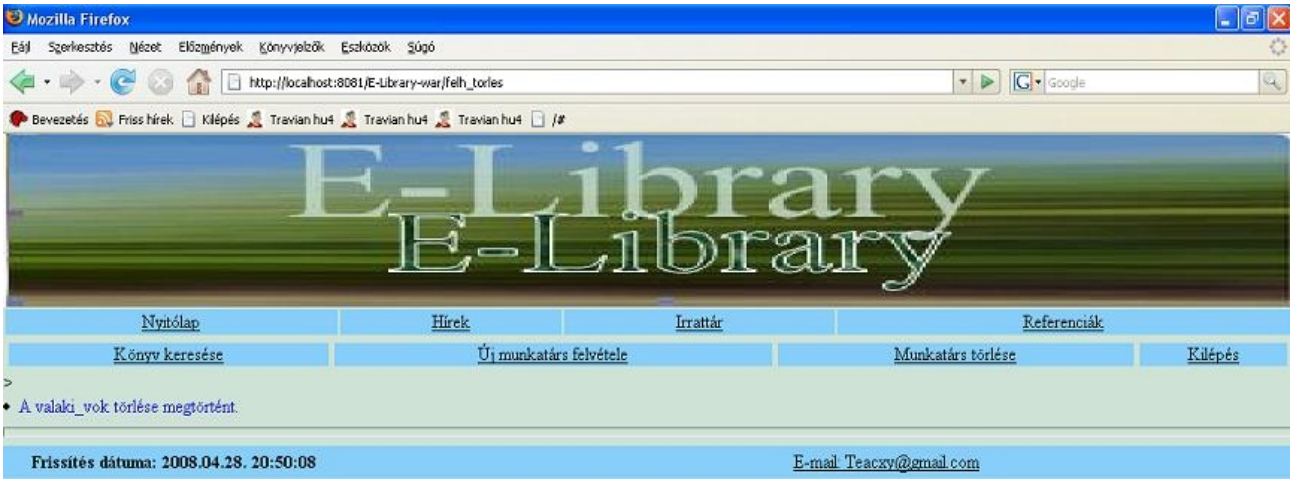
7.1 Képek

7.1.1 A kezdőoldal

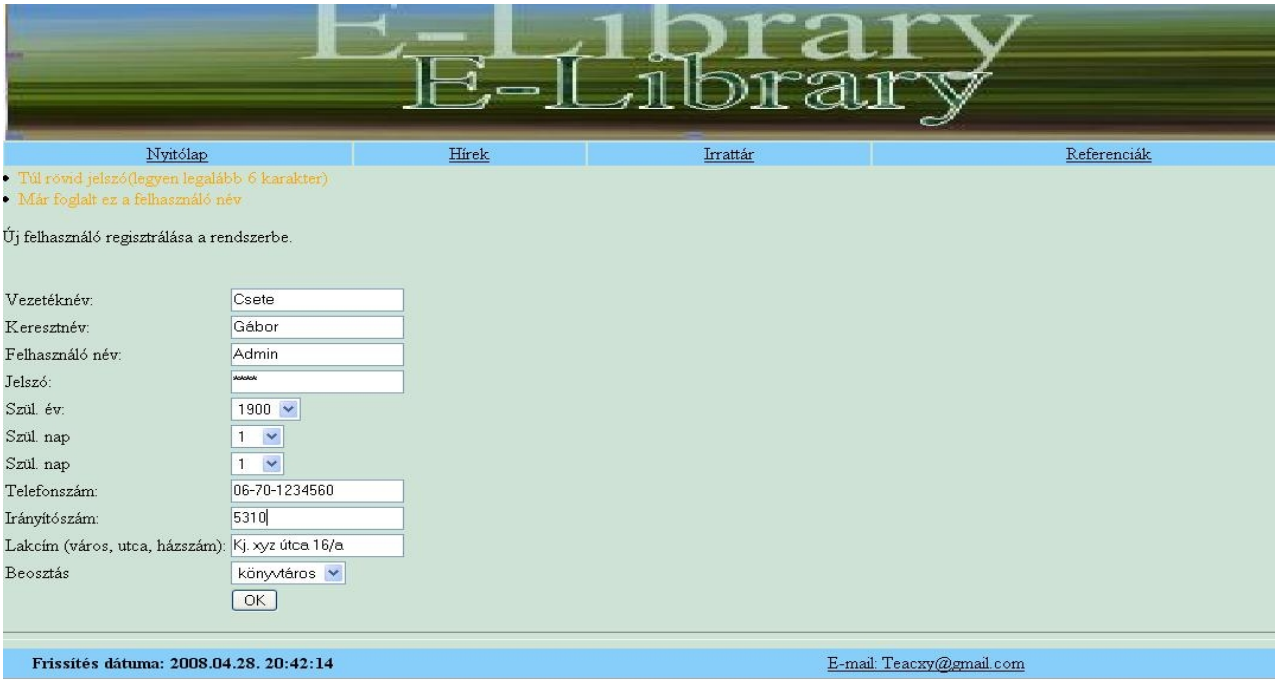
7.1.2 A bejelentkezés felülete



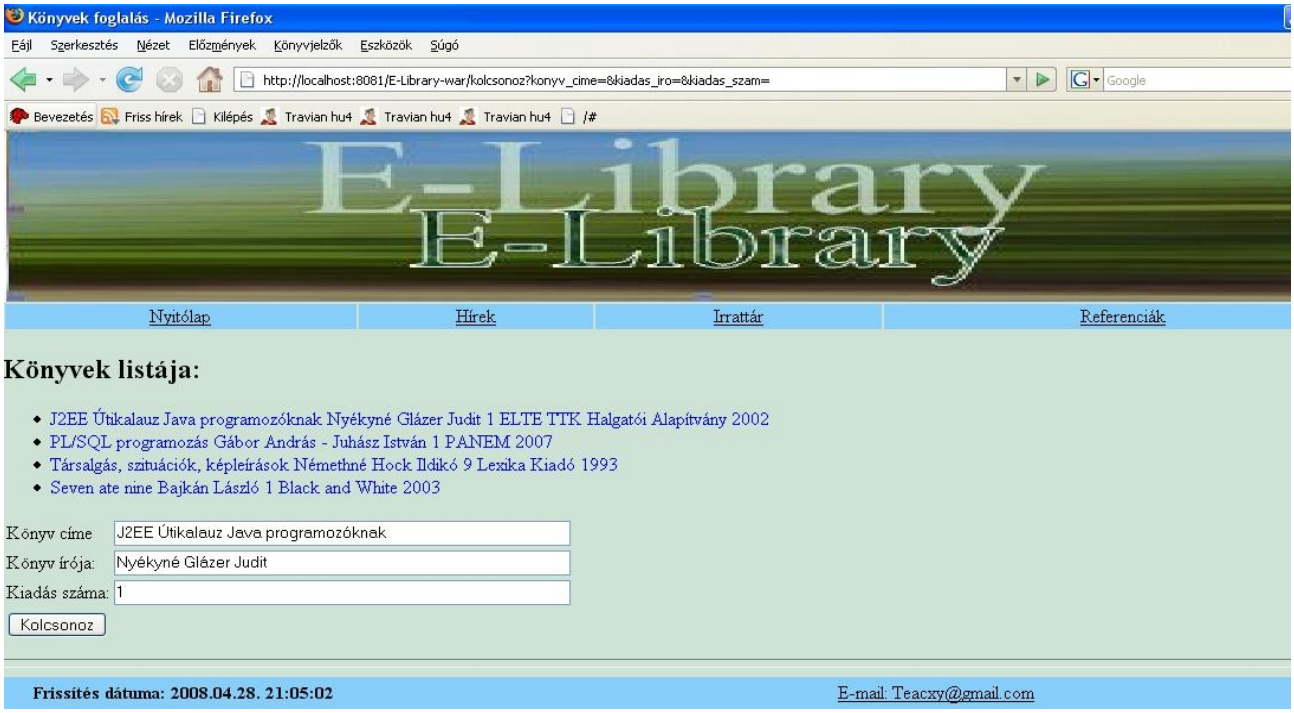
7.1.3 Felhasználó törlése

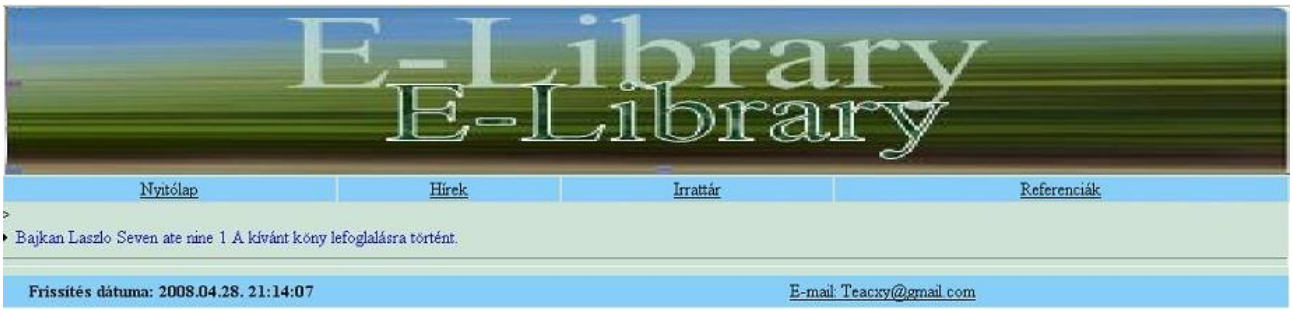


7.1.4 Új felhasználó létrehozása

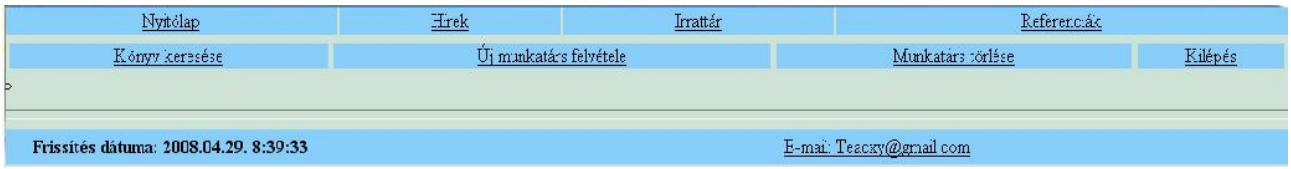


7.1.5 Könyvek listázása, illetve egyben kölcsönzése is

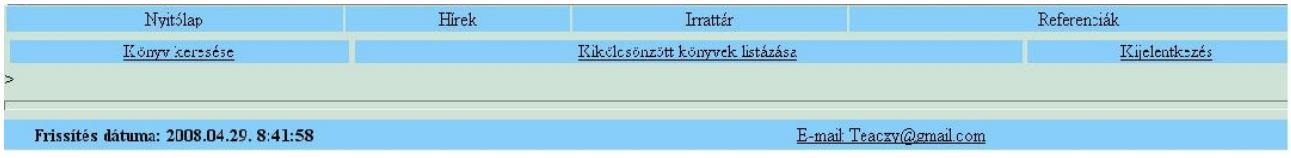




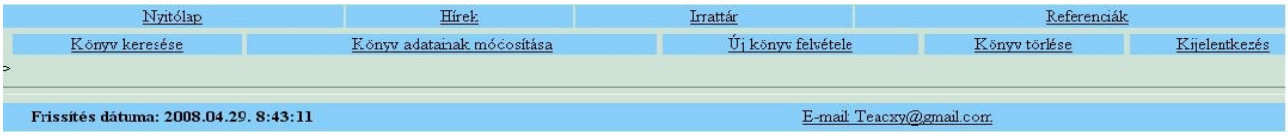
7.1.6 Adminisztrátor lehetőségei



7.1.7 Ügyfél lehetőségei



7.1.8 Könyvtáros lehetőségei



7.2 Felhasználó entitás bean

```
package entity;
import java.io.Serializable;
import java.util.Collection;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.OneToMany;
import javax.persistence.Table;
/**
 * @author Csete Gábor
 */
@Entity
@Table(name = "FELHASZNALO")
@NamedQueries({@NamedQuery(name = "Felhasznalo.findByEmailCim", query = "SELECT f
FROM Felhasznalo f WHERE f.emailCim = :emailCim"), @NamedQuery(name =
"Felhasznalo.findByVNev", query = "SELECT f FROM Felhasznalo f WHERE f.vNev = :vNev"),
@NamedQuery(name = "Felhasznalo.findByKNev", query = "SELECT f FROM Felhasznalo f
WHERE f.kNev = :kNev"), @NamedQuery(name = "Felhasznalo.findByJelszo", query = "SELECT f
FROM Felhasznalo f WHERE f.jelszo = :jelszo"), @NamedQuery(name =
"Felhasznalo.findBySzulEv", query = "SELECT f FROM Felhasznalo f WHERE f.szulEv
= :szulEv"), @NamedQuery(name = "Felhasznalo.findBySzulHo", query = "SELECT f FROM
Felhasznalo f WHERE f.szulHo = :szulHo"), @NamedQuery(name = "Felhasznalo.findBySzulNap",
query = "SELECT f FROM Felhasznalo f WHERE f.szulNap = :szulNap"), @NamedQuery(name =
"Felhasznalo.findByTelszam", query = "SELECT f FROM Felhasznalo f WHERE f.telszam
= :telszam"), @NamedQuery(name = "Felhasznalo.findBySzerep", query = "SELECT f FROM
Felhasznalo f WHERE f.szerep = :szerep"), @NamedQuery(name = "Felhasznalo.findByIrSzam",
query = "SELECT f FROM Felhasznalo f WHERE f.irSzam = :irSzam"), @NamedQuery(name =
"Felhasznalo.findByCim", query = "SELECT f FROM Felhasznalo f WHERE f.cim = :cim")})
public class Felhasznalo implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Column(name = "EMAIL_CIM", nullable = false)
    private String emailCim;
    @Column(name = "V_NEV")
    private String vNev;
    @Column(name = "K_NEV")
    private String kNev;
    @Column(name = "JELSZO")
```

```
private String jelszo;
@Column(name = "SZUL_EV")
private Short szulEv;
@Column(name = "SZUL_HO")
private Short szulHo;
@Column(name = "SZUL_NAP")
private Short szulNap;
@Column(name = "TELSZAM")
private String telszam;
@Column(name = "SZEREP")
private String szerep;
@Column(name = "IR_SZAM")
private Short irSzam;
@Column(name = "CIM")
private String cim;
@OneToMany(mappedBy = "emailCim")
private Collection<Kolcson> kolcsonCollection;
public Felhasznalo() {
}
public Felhasznalo(String emailCim) {
    this.emailCim = emailCim;
}
public String getEmailCim() {
    return emailCim;
}
public void setEmailCim(String emailCim) {
    this.emailCim = emailCim;
}
public String getVNev() {
    return vNev;
}
public void setVNev(String vNev) {
    this.vNev = vNev;
}
public String getKNev() {
    return kNev;
}
public void setKNev(String kNev) {
    this.kNev = kNev;
```

```
}
public String getJelszo() {
    return jelszo;
}
public void setJelszo(String jelszo) {
    this.jelszo = jelszo;
}
public Short getSzulEv() {
    return szulEv;
}
public void setSzulEv(Short szulEv) {
    this.szulEv = szulEv;
}
public Short getSzulHo() {
    return szulHo;
}
public void setSzulHo(Short szulHo) {
    this.szulHo = szulHo;
}
public Short getSzulNap() {
    return szulNap;
}
public void setSzulNap(Short szulNap) {
    this.szulNap = szulNap;
}
public String getTelszam() {
    return telszam;
}
public void setTelszam(String telszam) {
    this.telszam = telszam;
}
public String getSzerep() {
    return szerep;
}
public void setSzerep(String szerep) {
    this.szerep = szerep;
}
public Short getIrSzam() {
    return irSzam;
```

```
    }
    public void setIrSzam(Short irSzam) {
        this.irSzam = irSzam;
    }
    public String getCim() {
        return cim;
    }
    public void setCim(String cim) {
        this.cim = cim;
    }
    public Collection<Kolcson> getKolcsonCollection() {
        return kolcsonCollection;
    }
    public void setKolcsonCollection(Collection<Kolcson> kolcsonCollection) {
        this.kolcsonCollection = kolcsonCollection;
    }
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (emailCim != null ? emailCim.hashCode() : 0);
        return hash;
    }
    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not set
        if (!(object instanceof Felhasznalo)) {
            return false;
        }
        Felhasznalo other = (Felhasznalo) object;
        if ((this.emailCim == null && other.emailCim != null) || (this.emailCim != null &&
!this.emailCim.equals(other.emailCim))) {
            return false;
        }
        return true;
    }
    @Override
    public String toString() {
        return "entity.Felhasznalo[emailCim=" + emailCim + "]";
    }}
```