Oktatásszervező rendszer adatbázis-kezeléssel JSP-ben

Témavezető:
Kósa Márk
egyetemi tanársegéd

Készítette:
Hajdu Attila
programtervező informatikus

Debrecen
2010

Tartalomjegyzék

Bevezetés

A szakdolgozat célja, hogy egy projektben készítsünk el egy webes oktatásszervező illetve információs rendszert. Másodlagos cél, hogy a címben megadott alkalmazás fejlesztőjeként megismerkedjenek a projektben alkalmazott technikákkal, és a technika állása szerinti modern alkalmazásfejlesztéssel. Célom egy egyszerűen használható, gyors, minél több felhasználót kiszolgálni képes rendszer kifejlesztése volt. Funkciók terén törekedtem a lehető legtöbb általam szükségesnek vélt megvalósítására. Mintaként szolgált a Debreceni Egyetemen jelen pillanatban alkalmazott oktatásszervező rendszere, a Neptun, illetve a témavezetőm ötletei is.

Még egyetemi tanulmányaim megkezdése előtt hivatásszerűen webes technológiákkal foglalkoztam. Nem volt kétséges számomra, hogy szakdolgozatomat hasonló témakörben készítsem el. Egyetemi tanulmányaim alatt, miután megismertem a Java alapjait, elkezdett foglalkoztatni ezen ismeretek összeházasítása a megismert Java alapú technológiákkal. Itt szembesültem leginkább egy ilyen alkalmazással szembeni elvárásokkal: komoly rendszerek nagyfokú modularitásssal, robosztussággal, mégis a lehető hatékonyabb belső architektúrával kell rendelkeznie. Ez az ellentmondás végig kísérte az egész fejlesztés folyamatát. Takarékoskodni kellett az erőforrásokkal, hiszen egy ilyen rendszernek ki kell bírnia akár 5-10 ezer felhasználó napi tevékenységét is, melyek komoly adatbázis-terheléssel is járnak, a rendszert kiszolgáló szervereknek pedig véges a kapacitása.

A fejlesztés alatt megismertem néhány keretrendszert, technológiát, és egyéb eszközöket. Ezek közé tartozik például a JSP, XML, Hibernate Core, Hibernate Annotation, Hibernate Search, jQuery, JSON. Néhányat már előtte megismerkedtem, részben egyetemi tanulmányaim, részben külső projektekben. Annak ellenére, hogy tudásom róluk eléggé felületes volt, arra a döntésre jutottam, hogy bízván az eddigi programozói tapasztalatomban, ezt a témát választom. Döntésemet nem bántam meg, hiszen a későbbiek során is ezen a terülten kívánok dolgozni, így a megismert technológiák nagy hasznomra lettek, hiszen rengeteg tapasztalatot szereztem a projektek megszervezésétől egészen az egyes technológiák előnyeiig, hátrányaiig, képességéig.

A projekt jelenleg a végrehajtás utolsó fázisánál tart, mikoris megtörtént a meglevő funkciók finomhangolása. Mivel a technológiák megismerése rengeteg időt elvett, ezért sajnos több tervezett funkció implementálása nem történt meg.

Fejlesztői, és tesztelői környezet

A fejlesztésre Netbeans 6.8-as IDE-t használtam, mely beépített GlassFish szerverével megkönnyíti a fejlesztői környezet gyors kialakítását.
Adatbázis-kezelőnek (továbbiakban DBMS) az Oracle ingyenes verzióját, az Oracle Database 10g Express Edition-t választottam, mely nem járt jelentős korlátozásokkal, és tanulmányi célokra szabadon használható.
Alkalmazásomat Microsoft Internet Explorer 8.0, Mozilla Firefox 3.6 és Google Chrome böngészőkkel teszteltem.

A fejlesztés során használt eszközök, technológiák

GlassFish és a Java Enterprise Edition

Java alkalmazásomat egy GlassFish 3 szerver szolgálja ki. A GlassFish szerver együtt települ a Netbeans-szel, rögtön használatba vehető, mégis webes felületen nagymértékben konfigurálható, optimalizálható, mely jó hatással van a szerver válaszidejére, így meggyorsítva a tesztelés folyamatát. Létezik belőle kereskedelmi verzió is, mely a Sun Java System Application Server néven forgalmaz. Sokan idegenkednek ezen „egy kattintásos" szerverek alkalmazásától, de a megismerése után megváltozik a véleményük. Az adminisztrációs felülete egyszerű és letisztult. Képes mindenre, amit egy Java alkalmazásszervertől (J2EE Container) az elvárhatunk: JSP-k, servletek futtatása, valamint különféle EJB kezelése. A legtöbb IDE képes bele közvetlenül deployolni. Képesek vagyunk akár adminisztrációs felületen is feltölteni az alkalmazást, mely a web archive (továbbiakban WAR) konténerformátum alkalmazása miatt igen egyszerű. Más hardverre való migráláskor ez különösen hasznos, hiszen elegendő az alkalmazásunkat feltölteni, egy mentésből feltölteni az adatbázist, esetlegesen helyreállítani a kapcsolatot. Ezután már csak a finomhangolások vannak hátra, de a legtöbb esetben már ezek nélkül is működőképes az alkalmazás. Munkánkat megkönnyítendő, sok beépített osztályt tartalmaz, melyek egyszerűen felhasználhatóak. Ezek között van többet magam is felhasználtam.
A Java Enterprise Edition a Java nyelv azon része, melyet kifejezetten nagyvállalati felhasználásra terveztek. Így több programkönyvtárat (API-t) tartalmaz, mint a Standard Edition és az alkalmazásszerveren futó moduláris szoftverkomponensek segítségével

támogatja hibatűrő, sok felhasználós, többrétegű, elosztott alkalmazások készítését. Az átláthatóság miatt a különböző feladatokat csomagokba sorolták:[1]

- A javax.persistence.* csomagok tartalmazzák azon osztályokat, és interfészeket, melyek az adatok perzisztens módon való tárolásáért felelősek a relációs adatbázisban.
- A javax.ejb.* csomagok tartalmazzák a konkurenciakezelésért, a tranzakciókezelésért, az adatok konzisztenciájának figyeléséért, távoli elérésért felelős osztályokat.
- A javax.faces.* csomagok az egyes webes alkalmazásokban előszeretettel használt JavaServer Faces technológia komponenseit, és keretrendszerét tartalmazzák.
- A javax.enterprise.* csomagok segítséget nyújtanak a CDI használatához.
- A javax.jms.* csomagok felelősek a szoftverkomponensek közötti üzenetek küldéséért, fogadásáért, létrehozásáért.
- A javax.xml.stream.* osztályai felelősek az XML feldolgozásért.

Mivel vállalati technológia, fontos szerepelt tölt be a biztonság, a monitorozhatóság, magas rendelkezésre állás. Támogatnia kell az aszinkron üzenetküldést, hiszen ha éppen olyan erőforrásnak küldünk üzenetet, amely nem áll rendelkezésre, az üzenet eltárolódik, és ha az erőforrás ismét elérhető, megkapja az üzenetet. A vállalati cluster-be való könnyű beillesztés miatt jól skálázhatónak, és többszálúnak kell lennie, így egy alkalmazást egyszerre több felhasználó is elérhet. Több felhasználó esetén kiemelt szerepet kap a biztonság kérdése. Ez két problémakört vet fel: az authorizáció, vagyis arról, hogy az egyes felhasználók milyen szolgáltatásokhoz férhetnek hozzá, illetve beszélhetünk autentikációról, ami a felhasználókat egyértelműen azonosítja. Alkalmazásunk minden tevékenységét naplóznunk kell. Ezzel nem csak a biztonságot növeljük, hanem segítségével a szolgáltatás minőségének javítását segítő statisztikákat is kapunk.

Oracle Database 10g Express Edition

Korábbi tanulmányaim során megismerkedtem ezen DBMS több hasznos funkciójával, melynek nagy hasznát vettem a fejlesztés során, a szokásos eszközök mellett (elsődleges kulcsok, külső kulcsok támogatása, jogosultságkezelés, stb.) támogat több optimalizálási technikát is. Ezeknek az alkalmazása lehetővé teszi a lekérdezések optimalizációját. Rendszerem az adatbázis szerverrel JDBC driveren keresztül kommunikál.

[1] Forrás: http://en.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition

A JDBC (Java Database Connectivity) API definiálja a kliensek relációs adatbázishoz való hozzáférésének hogyanját. Elsőnek a J2SE 1.1-es verziójában jelent meg. Ezen API használatához szükség van egy az adott adatbázis-kezelőnek megfelelő típusú kliens oldali adapterre, az ún. driverre, amely átalakítja a szabványos API hívásokat a különböző DBMS-ek számára feldolgozható formátumúvá.

Hibernate

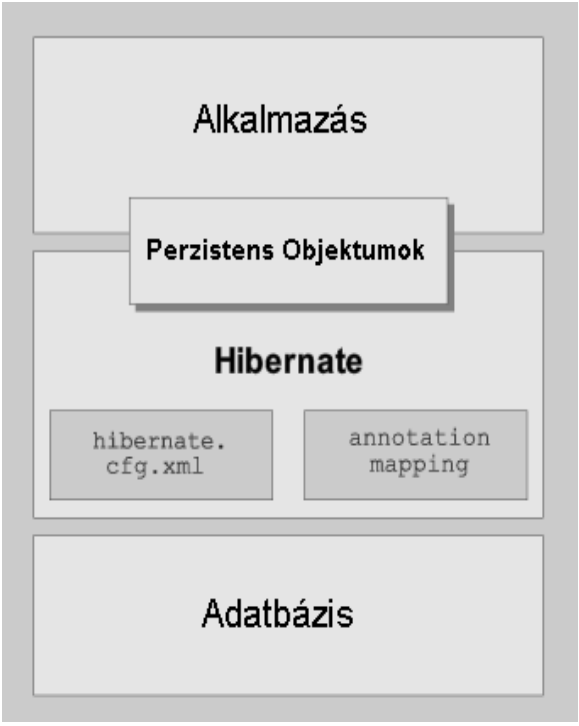
Több perzisztenciakezelő rendszer létezik (pl. Java Persistence API, Hibernate, TopLink), tanulmányaim során behatóbban a Hibernate-et ismertem meg. A Hibernate feladata a relációs adatbázis, és az objektum orientált világ közötti szemléletmód összehangolása, úgy, hogy köztes réteget biztosít a DBMS és az alkalmazás között. A Hibernate egy rendkívül kényelmes JPA alapú ORM rendszer.

Támogatja az OO perzisztens osztályok kezelését, az asszociációt, öröklődést, polimorfizmust, kompozíciót, és a kollekciót. Segítségével képesek vagyunk alkalmazásunkból egy vagy több adatbázissal kommunikálni, lekérdezéseket indítani, adatokat módosítani. Természetesen ehhez segédinformációkat kell adnunk a Hibernate-nek. Létre kell hoznunk a segédállományokat, amelyek leírják, hogy az objektumokat hogyan kell leképezni az adatbázisra. Meg kell mondanunk, hogy melyik táblába akarjuk menteni az objektumot, és azt is, hogy melyik mezőnek melyik relációs attribútum felel meg. Ezeket az állományokat mapping fájloknak nevezik. Két formájuk létezik: annotation kifejezések az entitás osztályokban, illetve XML állományok az entitás osztályok mellett. Alkalmazásomban annotation mapping leírókat használtam, így azt elemezném részletesen. Az XML mapping-ről érintőlegesen annyit érdemes tudni, hogy nem csak a Hibernate-nek segítenek az adatok leképzésében, hanem automatizált eszközökkel (pl. ant) a mapping állományokból legenerálhatjuk az adatbázis struktúráját definiáló DDL utasításokat, és a Hibrnate entitásokat is.

A Hibernate több alprojektet foglal magába. Ezek mind az egyik alprojekt, a Hibernate Core szolgáltatásain alapulnak. Alkalmazásomban a Core, az Annotations, és a Search szolgáltatásait használtam, így ezeket elemzem részletesen.

Hibernate Core

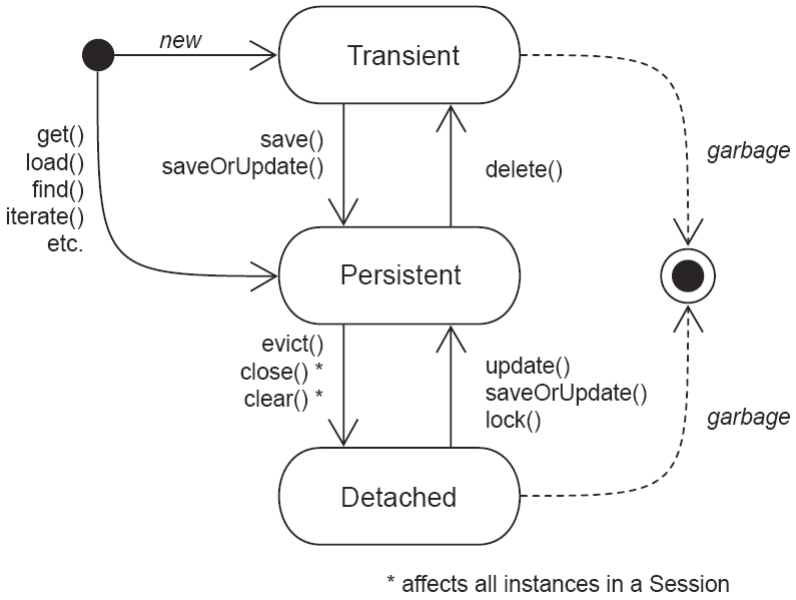
A Hibernate Core felépítését az 1. ábra mutatja.[2]



1. ábra

Az annotation mappingek által leírt formában kerülnek példányosításra a perzisztens objektumok az adatbázisból, melyeket a Hibernate átad az alkalmazás számára. A Hibernate lehetővé teszi ezen objektumok attribútumainak módosítását, melynek hatására az adatbázis egy UPDATE utasítással módosításra kerül, szinkronizálódva az objektumhoz. Az alkalmazás által létrehozott, még sohasem mentett tranzisztens objektumok egy INSERT utasítás képében válnak perzisztensé. Ezt a folyamatot, és a műveleteket reprezentáló metódushívásokat részletesen a 2. ábra mutatja.

[2] Forrás: http://www.inf.u-szeged.hu/~bilickiv/ProgRend/hibernate.ppt



2. ábra

A Core-ban két definiáltak két, a relációs adatbázisok kezelésére felkészített lekérdezőnyelvet, a HQL-t, és a Criteria-t. A HQL szintaxisa tekintetében az SQL-hez nagyon hasonló nyelv, míg a Criteria metódushívásokat használ. Mindkét nyelv teljes mértékben objektumorientált. Alkalmazásomban vegyesen alkalmaztam mindkét lekérdezésnyelvet.

Lényeges feladata a Hibernate Core-nak, hogy mivel a többi kiegészítő szolgáltatás, és az alkalmazások az ő szolgáltatásait használja, ezért tökéletesen megvalósítják a rétegarchitektúrát, hiszen egy réteg csak a közvetlenül felette, vagy alatta levővel kommunikál. Ez a tulajdonsága lehetővé tette, hogy a többi szolgáltatás számára biztosítsa az adatbázis elérését úgy, hogy az alkalmazás működőképes maradjon a DBMS cseréje esetén is, lényeges változtatásokat ne kelljen benne végrehajtani.

Ennek megvalósítására már a tervezéskor ügyeltek, széleskörűen konfigurálhatóvá tették a Hibernate-et. Ezeket a beállításokat annotációk használata esetén az org.hibernate.cfg. AnnotationConfiguration példány reprezentálja. Ezt több módon inicializálhatjuk, de vannak kitüntetett beállítások, amelyeket minden mód használata esetén meg kell adnunk. Ilyenek például, hogy melyik a már említett DBMS driver-t kívánjuk használni, az alapértelmezett sémát, felhasználónevet, jelszót. Alapértelmezésképp classpath-en levő hibernate.properties kulcs érték párokat tartalmazó állomány szerint történik meg ennek az inicializálása. Ha létezik hibernate.cfg.xml fájl, akkor erre a célra ezt használja. Előnye ennek, hogy a régebbi hibernate.properties állománynál jóval bővebb lehetőségeket kínáló XML formátumú konfigurációs állomány. Több, teljesítmény szempontjából lényeges paramétert is

konfigurálhatunk <property> elemek segítségével. Ezek közül az egyik legfontosabb a SessionFactory-ban található másodszintű cache-re vonatkozó beállítás, mely alapértelmezésképp aktív. Másik fontos paramétere show_sql, melynek részeként a Hibernate a beépített slf4j logger segítségével a Core által generált, és végrehajtottandó SQL utasítást kiírja a választott kimenetre, alapértelmezésképp a konzolra, nagyban megkönnyítvén a hibakeresést. Ezeknek az utasításoknak egy része nem kerül át a DBMS-nek, hiszen az entitások egy részét a Hibernate rögtön vissza tudja adni a másodlagos cache-ből.

Egy példa konfiguráció látható a 3. ábrán.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</property>
    <property name="hibernate.connection.driver_class">oracle.jdbc.OracleDriver</property>
    <property name="hibernate.connection.url">jdbc:oracle:thin:@localhost:1521:XE</property>
    <property name="hibernate.connection.username">user</property>
    <property name="hibernate.connection.password">user</property>
    <mapping class="database.User"/>
  </session-factory>
</hibernate-configuration>
```

3. ábra

Természetesen az esetleges paraméterek száma ennél jóval több lehet. További paraméterek megtalálhatóak a Hibernate weboldalán.[3] Az AnnotationConfiguration példányon meghívva a configure() metódust, a példány inicializálásra kerül a választott fájlformátumból. Azonban a Hibernate lehetővé teszi, hogy konfigurációs állomány nélkül, ezt mindössze a programkódból inicializáljunk, annak setter metódusai segítségével, illetve másik lehetőségként setProperties() metódusának egy Properties példányt átadva.

A Hibernate alapvetően egy HibernateUtil nevű osztályt használ a DBMSsel kapcsolódó session létrehozására, beállításainak beolvasára. Ezzel azonban van egy probléma: alapvető megközelítésbeli ellentétben áll a webalkalmazásokkal, mivel a HibernateUtil sok, egymással párhuzamosan létező session kezelésére alkalmas, míg egy webalkalmazás általában csak egy sessiont használ. Ezért a Hibernate fejlesztői javaslatára a HibernateFactory osztály használata terjedt el webfejlesztők körében, mely teljesíti ezeket a feltételeket.

[3] http://docs.jboss.org/hibernate/core/3.3/reference/en/html/session-configuration.html

Hibernate Annotation

A JDK 5 újításaként bevezetésre kerültek az annotációk. Segítségükkel a forráskódba (osztály, metódus, tagváltozó elé) metaadatokat illeszthetünk. A Hibernate Annotation esetében ezek a metaadatok adják meg az adatbázis kezeléséhez szükséges információkat a Hibernate-nek. Az alkalmazásomban legtöbb helyen EJB 3.0-ás annotációk végzik az objektum-reláció leképzést. A 4. ábrán látható az User entitás osztály forráskódjának egy részlete.

```
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.persistence.*;

@Entity
@Table(name = "USERS")
public class User implements Serializable
{
  @Id
  @Column(name = "USER_ID", nullable = false, length = 10)
  private String userId;
  @Column(name = "NAME_AT_BIRTH", nullable = false)
  private String nameAtBirth;
  @Column(name = "LAST_NAME", nullable = false)
  private String lastName;
  @Column(name = "FIRST_NAME", nullable = false)
  private String firstName;
  @Column(name = "MOTHERS_NAME", nullable = false)
  @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY, mappedBy = "user")
  private List<UserCheck> checks = new ArrayList<UserCheck>();
}
```

4. ábra

Látható, hogy szintaktikája egyszerű, jól kifejezi az adott reláció típusát. Az @Entity annotációval jelezzük a Hibernate-nek, hogy ez egy entitás lesz az adatbázisból. Hogy melyik táblát reprezentálja, azt a @Table annotáció adja meg, ami opcionálisan megadhatja a választott sémát is. Minden entity számára szükséges egy egyedi azonosító, ami a tábla elsődleges kulcsa. Az @Id annotáció egy egyszerű elsődleges kulcsot jelöl ki. Összetett elsődleges kulcs esetén az @EmbeddedId annotációt kell alkalmaznunk, és egy @Embeddable annotációval rendelkező osztály jelölünk ki az elsődleges kulcs elemeinek reprezentálására. @Column annotáció felelteti meg az egyes attribútumokat az adatbázis oszlopainak. Itt több lényeges információt adhatunk meg a Hibernate-nek az adott elemek hosszától egészen a nullitásán át. A mellékletben szereplő adatbázissémán látható, hogy a Checks tábla, és az Users tábla M:N típusú kapcsolatban van egymással, az Users_Checks

kapcsolótáblán keresztül. A @OneToMany annotációval jelezzük a kapcsolat típusát. Cascade paraméterrel megadhatjuk, hogy kaszkádosítás mely műveletek esetén történjen meg. Megadhatjuk a fetch módját, így definiáljuk, hogy betöltésnél az adott irányban be kell-e tölteni a hivatkozott entitásokat is. Ez az esetek többségében lusta (Lazy). Megadhatunk mohó (Eager) módot is, azonban ezt körültekintően kell használnunk, hiszen lehetséges, hogy sor felesleges adatot töltünk be vele a memóriába. Fontos az is, hogy ez a beállítás csak tanács a Hibernate számára, ami dönthet úgy, hogy mohó helyett lusta betöltési módot választ. További fontos paraméternek számít a mappedBy, mely kapcsolt osztályban levő, a kapcsoló osztályra mutató attribútum neve. Itt mutatkozik meg a Hibernate Annotation ereje: még az entitások közötti bonyolult kapcsolatok is egyszerűen leírhatók általa.

Hibernate Search

A Hibernate Search szintén egy a Hibernate csapat által készített eszköz, amelynek célja, hogy összekösse az objektum-relációs leképezés (ORM) és a szöveg alapú keresés világát. Ennek megvalósítására a Hibernate Search az Apache Lucene eszközére épít, ami egy nagy teljesítményű, nyílt forráskódú keresőmotor.

Sajnos az objektumok szintén nem lehet hatékony lekérdezéseket definiálni és végrehajtani. Részben ennek a kezelésére is megoldást kínál a Hibernate Search, mely az entitásokat indexek formájában menti el. Ezekben az indexekben hatékonyan kereshetünk, és ezek az indexek denormalizálják az adatokat, minden egyes keresni kívánt objektum gráfja (a keresés tervezett mélységének megfelelően) egyetlen kereshető, szöveg alapú Lucene dokumentumba kerül. Ezen indexen futtathatjuk a szöveges keresést, és a találatokból visszakapott ID-k alapján tudhatjuk, melyek a megfelelt, és immár példányosítható entitások. Ezek az indexek az első keresés előtt kell létrehozni, majd ezután ez folyamatosan, magától frissül, aktualizálódik.

A Hibernate Search alkalmazása mellett szól az a tény is, hogy bár az Oracle is biztosít full text search keresési módot, azonban akkor az OMR biztosította előnyök jelentős része elveszik.

Az 5. ábrán látható kód bemutatja az első lépést (a nulladik lépés az entitások, alap alkalmazás, DAO stb.) az indexelés beállítása.

```java
@Entity
@Table(name = "USERS")
@Indexed
@Analyzer(impl=StandardAnalyzer.class)
public class User implements Serializable
{
  @Id
  @Column(name = "USER_ID", nullable = false, length = 10)
  @DocumentId
  private String userId;
  @Column(name = "NAME_AT_BIRTH", nullable = false)
  private String nameAtBirth;
  @Field(index=Index.TOKENIZED, store=Store.NO)
  @Column(name = "LAST_NAME", nullable = false)
  private String lastName;
  @Field(index=Index.TOKENIZED, store=Store.NO)
  @Column(name = "FIRST_NAME", nullable = false)
  private String firstName;
}
```

5. ábra

Indexelendő entitásunk az User lesz, amelyet az alkalmazás számára kereshetővé kell tenni. Az indexelendő entitást megjelöljük az @Indexed annotációval. Megadhatunk (osztály és attribútum szinten is) Lucene Analyzer osztályokat, amelyek segítségével az indexelendő szöveget a Lucene feldolgozza, tokenizálja. A tokenizálás során a szöveg nyelvi (pl. toldalékolási) szabályok szerint kerül elemi bontásra a tárolás előtt. Ezen Analizer osztályok megadása sajnos nem tekinthető szükségesnek, mert jelenleg nincs jó magyar nyelvű, ingyenes, nyílt forráskódú Analizer. Legtöbször az entitás id-ja lesz egy dokumentum szintű egyedi azonosító, ezt a @DocumentId annotációval jelezzük. A többi indexelendő mezőre megadjuk a @Field annotációt. Ennek index attribútuma (alapértelmezett értéke az index = Index.TOKENIZED) adja meg, hogy az indexelés során az adott mező értékét tokenizálva, vagy feldolgozatlanul vegye át az indexbe. Az alapértelmezett analyzer például kisbetűsít mindent, kiveszi az angol szövegből a töltelékszavakat, illetve tövesíti a toldalékolt alakokat. A store attribútummal azt adhatjuk meg, hogy az indexbe tárolva legyen-e az eredeti alak, vagy ne. Mezők értékének tárolása komoly teljesítményromlással járhat, ugyanakkor tárolása nélkül a Hibernate Search tökéletesen elboldogul. @Field annotációkból több is megadható, így megtehetjük, hogy sorba rendezéshez tokenizálás nélkül is elmentünk egy mezőt az indexbe, a tokenizált formája mellé. Ezeket a formákat @Fields annotációval egyesítjük az indexbe. Az egyes mezőknek adhatunk súlyozást a @Boost annotációval. Ezzel azt szabályozhatjuk, hogy az egyes mezők milyen súllyal szerepeljenek majd a keresések során.

Egy példakeresés látható a 6. ábrán.

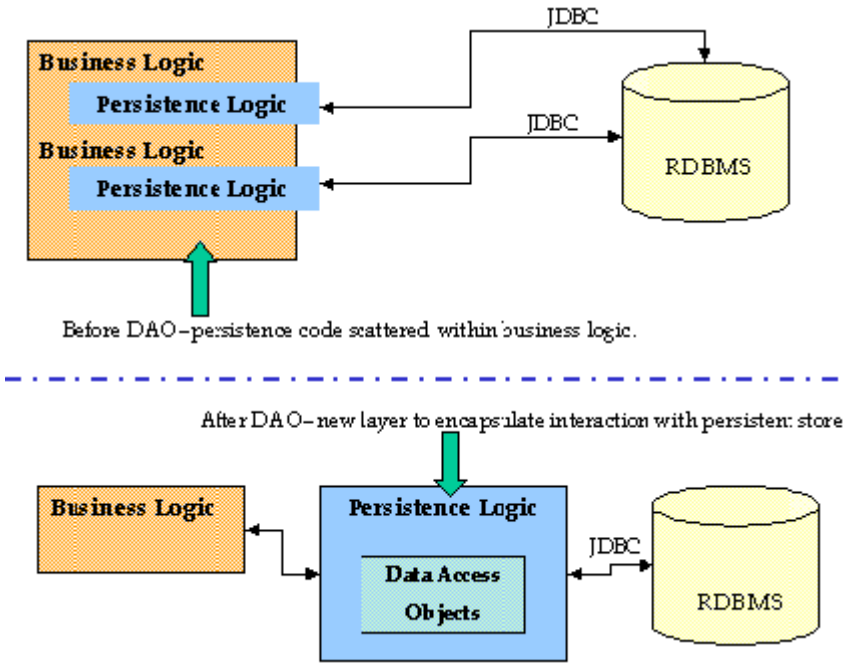
```
FullTextSession fullTextSession = Search.getFullTextSession(getSession());
fullTextSession.createIndexer().startAndWait();

String[] fields = new String[]
{
  "lastName", "firstName"
};
MultiFieldQueryParser parser =
        new MultiFieldQueryParser(
          Version.LUCENE_29, fields, new StandardAnalyzer(Version.LUCENE_29)
        );
org.apache.lucene.search.Query query = parser.parse(keyword);
org.hibernate.Query hibQuery = fullTextSession.createFullTextQuery(query, User.class);
results = hibQuery.list();
```

6. ábra

Data Access Object

Egy Hibernate-ben írt alkalmazás egyik legnagyobb problémáját jelenti a tranzakció-kezelés. Probléma esetén meg kell oldani az adott tranzakció visszagörgetését, mindezt a lehető legkisebb bonyolultsággal. Mikor az első Hibernate-es alkalmazásaimat írtam, sajnos én is bele estem abban a hibába, hogy pl. egy servlet törzsébe tranzakciókezelő, sessionkezelő utasításokat, és magát a lekérdezést írjak. Ezzel a legnagyobb probléma a kód újrafelhasználhatóság teljes hiánya. Egy lekérdezést csak egy adott servleten belül lehetséges elvégezni, más servletekben erre nincs lehetőség. Másik probléma, hogy adatbázis kezeléssel kapcsolatos kódok teljesen átszövik az üzleti logika kódjait. Ilyen helyzetben a legkisebb refactoring is kiterjedt problémákat okozhat. Szerencsére erre a problémára létezik megoldás, mégpedig a valamilyen Data Access Object (továbbiakban DAO) réteg közbeiktatása, mely adatbázis kezeléssel kapcsolatos kódokat elválasztja az alkalmazás többi részétől. Ennek köszönhetően megnő a kódunk olvashatósága, és jobban követhető az üzleti logika.

A 7. ábra bemutatja a DAO elvi felépítését, melyen jól látszik az üzleti logika követhetőségének javulása.



7. ábra

Ahhoz, hogy DAO réteget alakítsunk ki, szükséges, hogy az adatbázis-kezelő kódokat logikailag összetartozó csoportokba válogassuk. Erre a legcélszerűbb, ha az egy típusparaméterű generikus kollekciókat visszaadó kódokat az adott típust visszaadó kódokkal együtt egy DAO osztályba helyezzük.

Alkalmazásomban az újrafelhasználhatóság jegyében generikus réteget használtam, és abból származtattam le az egyes entitásokat kezelő DAO osztályokat. Ennek köszönhetően nem kellett olyan alap metódusokat megírni, mint az új példány perzisztenssé tétele, frissítése, törlése az adatbázisban, hiszen ezeket a szülőosztályban már generikusan megtettem. Az egyes osztályokban az adott osztály példányát, illetve példányok kollekcióját visszaadó metódusokat hoztam létre. A leszármazott osztályokban az elsődleges kulcsot reprezentáló attribútum, és az osztály típusát konkretizáltam.

Az entitások kezelését a következő módon végeztem: az adott entitást kezelő DAO osztály példányosítása után használhattam a bennük, és a szülőosztályban definiált metódusokat.

A megírt metódusokban kerültem üzleti rétegből lekérdezhető attribútumok visszaadását. Így például nem adtam vissza egy entitás egy-két attribútumát, hiszen ezek az attribútumok az üzleti rétegből lekérdezhetőek, mindazonáltal olyan azon HQL lekérdezés, mely az egyes példányok egyes attribútumait adja vissza, az esetek többségében hosszabb ideig hajtódnak végre, mintha az egész példány átadásra kerülne. Ennek oka a Hibernate struktúrájának alapkövét jelentő cache menedzselésében keresendő: egyes attribútumokat tartalmazó sellista

használata esetén az esetek többségében a lekérdezés SQL lekérdezőnyelvé alakított formája végrehajtásra kerül az adatbázison, míg teljes példány esetén általában ki tudja szolgálni a lekérdezést a cache-ből, a DBMS-hez fordulás nélkül.

JSON formátum és a Flexjson szerializáló

Mivel alkalmazásomat mindenképpen AJAX-os dinamikus tartalombetöltéssel akartam ellátni, így meg kellett találnom az optimális adatcsere formátumot. Korábbi tapasztalataimból kiindulva a JSON formátum használata mellett döntöttem. Első alkalommal, mikor döntenem kellett az adatcsere formátumáról, számba vettem a lehetséges előnyöket, hátrányokat az egyes formátumok között.

Az XML előnyére írható, hogy ez illeszkedik leginkább az AJAX eredeti koncepciójához. Nem meglepő tehát, hogy a legtöbb webes fejlesztés ebbe az irányba haladt. Az elterjedt használat motiválta a böngészők DOM értelmezőinek fejlesztését is. Az XML könnyen használható SOAP alapú web szerverek esetén is, így a szerver oldali kód e része újrafelhasználható. Jól olvasható, hiszen ez az egyik alapeleme. Hátránya azonban, hogy igen terjedelmes. Minden taget kötelező lezárni, és sok taget egymásba illesztve az olvashatósága is csökken.

XHTML formátum választása esetén a legtöbb előny abból származik, hogy feldolgozása a kliens oldalon meglehetősen egyszerű. Azonban jelentős hátrányai is vannak: komoly probléma, ha a válasz formot tartalmaz, illetve a kliens oldali formázást is a válaszban kell elhelyeznünk. Így projektünk egy idő után ellenőrizhetetlenné válik, komoly lemaradást okozhat egy-egy rossz helyen elhelyezkedő elem, amit a böngészők eltérő CSS értelmezése vált ki. Mindennek következménye a hálózati forgalom ugrásszerű terhelése, a szerver válaszidejének, és terhelésének növekedése.

A harmadik szóba jöhető formátum, a JSON. Ennek előnyei az egyszerű és erőforrás kímélőbb-adatfeldolgozás. Mivel alapötlete, hogy a szerver a válaszban egy olyan karaktersorozatot küldjön, ami könnyen JavaScript objektummá alakítható, így a JavaScript képes dinamikusan más JavaScript erőforrásokat importálni. Eddig ezt csak DOMmal tehettük meg, más módszer egyes böngészőkben nem várt problémákat okozott. A DOM azonban lassú, és erőforrás-igényes. A legtöbb szolgáltató felismerte előnyeit, így már nemcsak a SOAP standard webszolgáltatási formátum, hanem már a JSON is. A modern böngészők tervezői is felismerték a formátumban rejlő lehetőségeket, így natív támogatást nyújtanak a

feldolgozására. Ennek köszönhetően mára sebességében felülmúlta a másik legelterjedtebb formátumot, az XMLt, annak versenyképes alternatíváját nyújtva. Hátrányaként említendő, hogy olvasása megszokást igényel. A zárójelpárok összerendelése nem mindig egyértelmű. Ezt magam is tapasztaltam, hiszen a kapott, sokszor bonyolult válaszokat NotePad++ nevű ingyenes, nyílt forráskódú editor segített értelmezni. Ez előre vetíti, hogy JSON mellett döntöttem, hiszen az XML terjedelmes, de jól olvasható, az XHTML meg ezek egyikét sem biztosítsa, sőt, rengeteg problémát okozhat. Sajnos több helyen használják ezt a formátumot (még üzleti szoftverekben is), de szerencsére kiveszőben van.

A JSON (JavaScript Object Notation) egy emberek számára is olvasható–írható, programozottan könnyen feldolgozható és előállítható szabványos pehelysúlyú adatcsere-formátum. Programozási nyelvtől független, hasonlóan az XML-hez, de a C nyelvcsalád valamely tagjában jártas programozó számára nem megterhelő a megértése. A Python és LISP ugyan nem a C nyelvcsalád tagjai, de e nyelven programozók jelentős előnyben vannak az értelmezésénél. A JSON név-érték párok halmazát, illetve értékek rendezett halmazát tartalmazhatja.

A 8. ábrán látható egy példa kimenet a tesztadatokkal feltöltött alkalmazásomból[4].


```
{"id":{"courseId":"INAK221L-
K0","semesterId":"2009/10/01"},"lecturer":{"firstName":"Phrakonham",
"id":"seng","lastName":"Seng","type":"lecturer"},"room":{"roomId":"M
114","roomSpace":30,"type":null},"studentsLimit":999,"time":null}
```


8. ábra

A formátum alapelemeit objektumok alkotják. Az objektum név-érték párok rendezetlen halmaza. Egy objektum „{” jellel kezdődik és „}” jellel zárul. Minden nevet „:” követ. Egyes párokat egymástól „„” választ el. A tömb értékek rendezett halmaza. A tömb „[” jellel kezdődik és „]” jellel zárul. Az értékeket „„” választja el. Érték lehet idézőjelek közé írt karakterlánc, szám, logikai igaz, logikai hamis, null, objektum vagy tömb. A struktúrák egymásba ágyazhatók. A karakterlánc nulla vagy több, idézőjelek közé zárt Unicode karakter, szükség szerint visszaper-jellel kivédve. A karakter egy hosszúságú karakterláncnak felel meg. A karakterlánc nagyban hasonlít a C vagy Java karakterláncaihoz. A szám a C és Java számaihoz hasonló. A különbség az, hogy oktális és hexadecimális formátum itt nem használható.[5]

[4] Forrás: http://localhost:8084/STR/CurrentCourseAjax?courseId=INAK221L-K0&semesterId=2009/10/01
[5] Forrás: http://www.json.org/json-hu.html

A Flexjson egy JSON kimeneti formátumú, jól paraméterezhető, pehelysúlyú objektum szerializáló. Segítségével egyszerűen, és gyorsan alakíthatjuk kollekciótípusainkat, illetve objektumainkat JSON formátumúvá, melyet a rengeteg helyen használhatunk fel. Ezeket az alkalmazásomból kapott adatokat jQuery függvénykönyvtár segítségével dolgoztam fel.

A Flexjsont tervezője, Charlie Hubbard kimondottan a Hibernate-tel kapott entitások, és kollekciók szerializálására írta, hogy az így kapott JSON adatfolyamot jQuery-vel saját oldalába illeszthesse. Megteremtette annak a lehetőségét, hogy a Lazy betöltődésű attribútumok feldolgozása tiltható legyen, így megspórolván jó pár adatbázis műveletet. Kollekciótípusok bejárása alapból tiltott, de egyenként engedélyezhető.

Mikor elkezdtem használni a Flexjsont, belefutottam abba a problémába, hogy egy Date típusú objektum szerializálásakor a kapott érték a timestamp lett. Próbáltam a Hibernate entitás osztályaiban a Date típusú attribútumnak különböző TemporalType értékeket megadni, de nem értem célt. Jobban átolvasva a dokumentációt, rájöttem, hogy ezzel csak az adott attributum „pontosságát", nem pedig a szerializálás formátumát befolyásolom (ti. a Hibernate például TemporalType.DATE megadása esetén nem veszi figyelembe a megkezdett nap közben eltelt órákat, olyan timestamp-et ad vissza, amelyik az adott naptári nap 0 óra 0 percbeli timestampjét adja vissza).

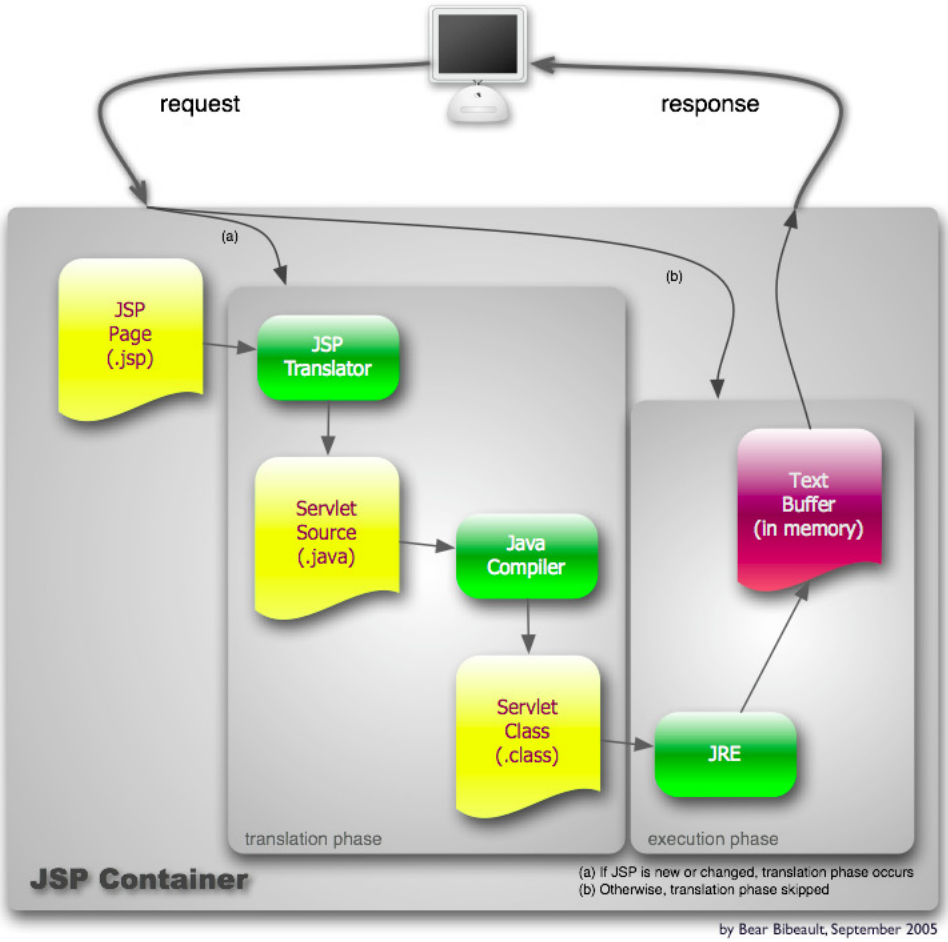
Ekkor első gondolatom a JavaScriptbeli formátumozás volt. Ezt hamar elvetettem, mert sajnos a JavaScript csak a formátumkódból képes generálni a kívánt formátumú időt, és dátumot, nem pedig a Javában már megszokott, a DateFormat statikus, final adattagjai által meghatározott paraméterek szerinti formátumozás, ahol nem kell törődni az egyes nyelvek szerinti helyes formátumkóddal, hiszen a SimpleDateFormat ezen paraméterek segítségével képes lokalizációfüggően formátumozni. Formátumkódot definiálni minden nyelv számára kényelmetlen, és hibalehetőségekkel teli tevékenység. Így elvetettem a kliens oldali formátumozás gondolatát, és a szerver oldali megoldásra koncentráltam. Erre nem találtam módot a Flexjsonben, így forráskódját letöltve írtam egy osztályt, amely konstruktorában SimpleDateFormat példányt fogadva képes a paraméterben kijelölt mezőt átalakítani a szintén paraméterként átadott lokalizáció szerint a már fentebb említett SimpleDateFormat segítségével.

A másik probléma kicsit később jelentkezett. Egyes felhasználók típusát az adatbázisba angolul mentettem el. Ez kilistázva szokatlanul nézett volna ki a lokalizált környezetben, így írtam még egy osztályt, amely paraméterül konstruktorában megkap egy HashMap példányt,

képes lecserélni a szintén paraméterül kapott mező értékét, azt indexként használva, a HashMapben tárolt értékre. Így ezt a HashMap-et a típusok lokalizált változataival feltöltve elérjük a kívánt eredményt.
Ezek az osztályok szerkezetüket tekintve igen egyszerűek. Implementálták a Flexjson beépített Transformer interfészét, így könnyű volt beilleszteni őket a meglévő struktúrába, kevés értékes időt vonva el más feladatoktól.
Mivel hasznosnak találtam e funkciókat, így jeleztem a problémát a fejlesztőnek, akinek tetszettek az ötleteim. Reményeim szerint valamely későbbi verzióban beépíti ezeket az osztályokat a Flexjson stabil kiadásába.

JSP

Alkalmazásom a megjelenítésre JavaServer Pages (továbbiakban JSP) technológiát használ. Hasonlóan más szerveroldali nyelvekhez, a JSP is képes HTML, XML, és egyéb dokumentumokat dinamikusan generálni. Segítségével a szervernek küldött kéréseket tudjuk megválaszolni, elküldve az kliens által óhajtott adatfolyamot. A JSP fájlok egy absztrakciós szint képviselnek a Javában megírt servletek fölött. Ezzel a dinamikus tartalomgenerálás problémáját oldja meg. A generált tartalom jellemzően HTML, de lehet például XML is. Általában egy servlet meghívására a servlet beállítja a kérés egy paraméterét, és a kérés továbbítódik egy JSP fájlnak, aki ezt a kérést átveszi, saját absztrakciós szintjén feldolgozza, módosítja, megjeleníti. Ebben segítséget a később tárgyalt JSTL technológia ad. Azonban lehetséges a kérést más servletnek átadni. Ezen kettősség furcsasága rögtön megoldódik, amint megnézzük, hogy a webszerver hogyan értelmezi a JSP fájlokat, illetve egy kérésre hogyan születik meg a válasz a 9. ábrán látható.



9. ábra

Jól látható, hogy a JSP fájl egyszerűen egy servletté konvertálódik. A JRE szintjén ekvivalens a JSP fájl egy servlettel. Fontos is ez az ekvivalencia, hiszen így lehetővé válik Java kódot beillesztése magába a JSP kódba, ami bár nem tanácsos, de megtehető (lásd JSTL fejezet).

A servleteket a servlet container kezeli, mely a webszerver egy komponense. Feladatai közé tartozik a servletek életciklusainak a kezelése és az URL-ek hozzárendelése a servletekhez, és az erre a címre érkező kéréseket servlet metódushívássá konvertálása. Ezen hozzárendelést a web.xml fájl alapján végzi, amiben egyéb, kiegészítő információkat is megadhatunk. Ilyenek például a session élettartalmára, inicializálási paraméterekre, filterekre (a filter olyan servlet rendszerű osztály, melyen keresztülfut az beállított típusú kérés, és az arra adott válasz, azokat a törzsében definiált logika szerint manipulálja) adott beállítások.

JSTL

A JSTL (JavaServer Pages Standard Tag Library) segítségével JSP oldalba HTML 4.01 szabványú tageket illethetünk, mely tagek használatával egyszerű vezérlési szekvenciákat, adatbázis, ill. xml állományok kezelését, lokalizációt és egyszerű String műveletek tudunk elvégezni JSP lapjainkon anélkül, hogy scriptlet elemeket kellene alkalmaznunk. Előnye ennek a technikának az egyszerű, átlátható oldalszerkezet. Lényeges elem, hogy ezeket a tageket a grafikusok által használt grafikus HTML editorok ignorálják, de jelenlétüket jelzik. Így a nélkül tudunk logikát vinni az oldalainkba, hogy az egyszerűen használható, eddig megszokott tervezőeszközeiket nélkülözniük kellene. Ezek az eszközök a már beépített logikát általában nem módosítják, így a webalkalmazás grafikus felülete módosítható anélkül, hogy ezek nem várt károsodást szenvednének.

A JSTL tageket JSP elemkönyvtárakba soroljuk. A beépített elemkönyvtárak:

- a core, mely vezérlési szekvenciák kezelésére használatos
- az xml, mely XML kezelő elemeket tartalmaz
- az sql, melynek segítségével SQL parancsokat adhatunk ki
- az fmt, mely lokalizációt, és formázást végez
- és az fn, mely szövegkezelő függvényeket tartalmaz.

Ezeknek a könyvtáraknak számos implementációja létezik, a legelterjedtebbek a Sun illetve az Apache féle implementációk. A GlassFish természetesen a Sun-féle implementációt beépítve tartalmazza. Természetesen készíthetünk saját könyvtárat is, ezzel plusz logikát hozzáadva a JSP oldalunknak. Szakdolgozatomban szükségesnek véltem megtervezni, és implementálni egy JSP 2.0-ás verziójú taget, melyek a grafikus felületen valósítja meg az authorizációt. Erről bővebben leírás a Biztonság c. fejezetben található. Mindamellett szükséges volt a core, és az fmt függvénykönyvtár használata, melyek segítségével az alkalmazásom viszonylag hamar kész felhasználói felülettel rendelkezett.

Ahhoz, hogy egy könyvtárat oldalunkon kezelni lehessen, el kell látnunk oldalunkat egy taglib direktívával, hogy melyik JSP elemkönyvtárat kívánjuk használni. Ezen direktíva paramétereként meg kell adni egy URI-t, ahol az elemkönyvtár leírófájlja található, valamint egy prefixumot, amivel később hivatkozni lehet az elemkönyvtár elemeire. Erre példa a 10. ábrán látható.

```
<%@ taglib prefix="myTags" tagdir="/WEB-INF/tags/myCustomTags" %>
```

10. ábra

Ezzel a myTags prefixumhoz hozzárendeltük a leíró URI-n található tag fájlt.
Említésre méltó még az include direktíva, mely segítségével egy teljes fájl tartalmát fordítás előtt, statikusan beillesztjük a JSP oldalba. A beillesztett fájlok kiterjesztése általában jspf, mely a JSP Fragment rövidítése. Erre a direktívára példa a 11. ábrán látható.

```
<%@include file="../jspf/head.jspf" %>
```

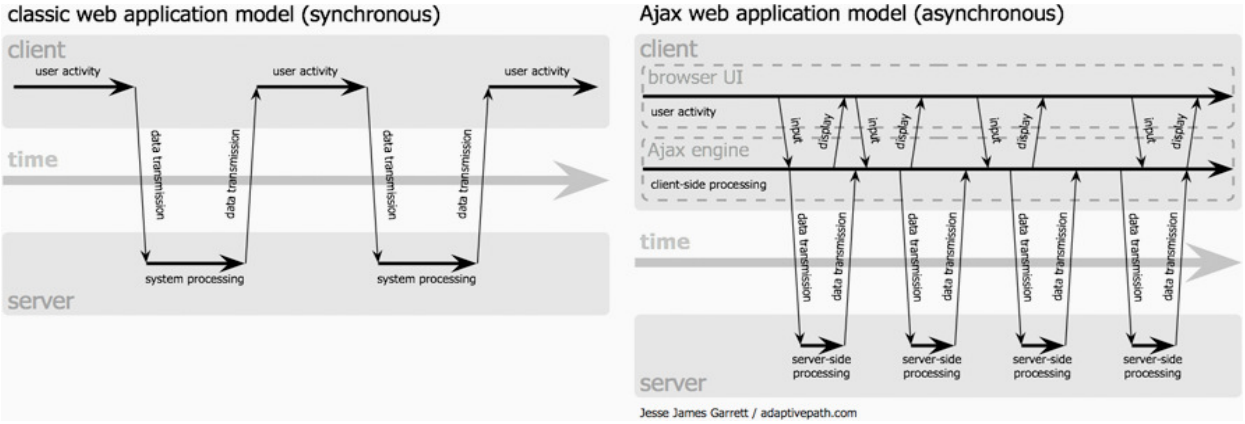
11. ábra

jQuery

A jQuery egy népszerű JavaScript függvénytár, mely kapcsolatot teremt a HTML kód, és a kliensoldali JavaScript között. A jQuery ingyenes, nyílt forrású szoftver. A grafikus felület fejlesztése során nagy hangsúlyt kapott az, hogy minél többféle böngészőt támogasson. Ebben nagy segítségemre volt a jQuery széleskörű böngészőtámogatása, mely segítségével sok függvény, és eseménykezelő már előre definiált volt, nem kellett a különféle böngészőkhöz igazítanom a JavaScript kódokat, csak a jQuery saját komponenseit használni. Ezen komponensek segítségével egyszerűen megoldható egy-egy HTML objektumra vonatkozó eseménykezelés, anélkül, hogy magába a tagbe kellene „on-direktívákat" (például onChange, onClick, stb.) beszúrnunk, és ezzel JavaScript függvényeket meghívnunk. Egyszerűen manipulálható az egyes elemekre vonatkozó CSS paraméterek. Segítségével effekteket, animációkat adhatunk az oldalunkhoz, melyek közül a legismertebb ilyen effekt az ún. lightbox, amely a böngészőbe megnyíló dialógusablak mögötti területet elsötétíti, elfedi, és kattinthatóságot letiltja. Széleskörű támogatást nyújt az AJAXos oldalak megírásához, ami használata nélkül a különböző böngészők eltérő szintaktikája miatt nehézkes lenne. Széleskörű, jól paraméterezhető plugin támogatással bír, melyek a legkülönfélébb feladatokat, és felületeket nyújtják a felhasználó felé. Ezen pluginok közül a legtöbbet használtakat beépítették a jQuery UI függvénytárába, így ezek rögtön használhatóak. Ezek közül alkalmazásomban a Dialog, és a DatePicker plugint használtam, a menü megvalósítására pedig egy nyílt forráskódú, nem kereskedelmi szoftverben szabadon felhasználható megoldást találtam, melynek grafikus felülete is ilyen licensz alá esett, így könnyítve meg a felület megtervezését.[6]
Miután megismerkedtem ezen pluginok használatával, elkezdődött a lényegi munka, a kliens oldali logika implementálása a jQuery eszközei segítségével.

[6] Forrás: http://apycom.com/

A jQuery használata igen egyszerű: a Sizzle kódnevű JavaScript-CSS szelektorral kiválasztunk a HTML oldalunkban egy DOM objektumot a CSS-ben megszokott szintaktikával, és manipuláljuk annak attribútumait a beépített metódusok segítségével. A jQuery tartalmaz olyan függvényeket is, melyeket nem objektumon hívunk meg. Ezen függvények közé tartozik például a $.getJSON(url, [data], [callback(data, textStatus)]), mely alkalmazásom leggyakrabban használt függvénye, mellyel a szervertől JSON formátumú adatokat kérünk le. Ezeket az adatokat az AJAX szemléletéhez méltó folyamatmodellel kérjük le, melyet a 12. ábra mutat be.



12. ábra

Jól látható a különbség a két szemléletmód között. Az AJAXszal ellentétben, mely képes frissíteni az oldal tartalmát a felhasználó interakcióinak megfelelően az egész oldal újratöltése nélkül, a klasszikus modellben minden interakcióra az oldal újratöltésre kerül. Az AJAX előnyére írható, hogy könnyű benne dinamikus weboldalakat írni, melyek kezdetbe csak minimális információtartalommal bírnak.

Tesztelés

Az alkalmazásom tesztelése közben egy-egy modul elkészülte után modulteszttel ellenőriztem, hogy a modul megfelel el a dokumentációnak. A tesztelés megkezdése előtt száraz tesztnek vetettem alá a modul kódját, szemmel látható hibák után kutatva. Száraz teszt után hiányosságtesztelést végeztem, mely során a szintaktikailag helyes kód szemantikai hibáira is fény derült. Ekkor történt meg az adott kód validálása, mely során a dokumentációnak való megfelelést vizsgáltam meg. Ha a validálás sikeres, integrációs tesztelés következett, annak ellenőrzése, hogy a rendszerbe illetve hogyan viselkedik az adott modul. Ha a tesztelendő modul egy servlet, akkor ennek során különös figyelmet fordítottam a felhasználói felülettel való kapcsolattartásra, elengedhetetlen volt tesztesetek tervezése.

Tesztesetek tervezésére egyszerű stratégiát választottam: a való életből vett adatokhoz hasonló adatokkal töltöttem fel az adatbázist. Ezután kimerítő tesztelésnek vetettem alá a rendszert, mely során ügyeltem arra, hogy minden kódrészlet legalább egyszer végrehajtódjon. Ez irányított tesztelés volt két próbafelhasználóval. Miután az összes funkciót kipróbálták, hagytam, hogy szabadon teszteljék a felületet, megismerjék, kattintgassanak, mozgassák az ablakokat, stb. Legutoljára a stressz-tesztelés következett be, mikor tesztadatokkal feltöltöttem, majd kérések sorozatát indítottam a szerver felé, miközben mértem a válaszidőt, és az erőforrás-foglaltságot.

Az egész tesztelési folyamatból kapott visszajelzések, hibákat felhasználtam a fejlesztés folyamán. Egy-egy hiba kijavítása után minimum modulteszttel ellenőriztem a modul működését, de általában integrációs tesztnek vetettem alá.

Az STR oktatásszervező rendszer

Felépítés

Alkalmazásomat a könnyebb áttekinthetőség végett a következő csomagokba szerveztem:

- default, csomagon kívüli: log4j, Hibernate beállítások, lokalizációs fájlok.
- action.*: a különböző, a Hibernate egyes entitásain végzett interakciókat megvalósító servletek. Közéjük tartozik például a felhasználó hozzáadása, kurzus törlése, terem törlése. Webes felületről ezek a servletek hívódnak meg, ha a felhasználó egy dialógusablakban egy gombra kattint. Ezen servletekben a legfontosabb a megfelelő jogosultságok hozzárendelése. Ezen jogosultságok ellenőrzése után (például felhasználó előfeltételeket teljesítette), a servlet lefut, és elvégzi a feladatát. Ezeket a servleteket általában $.get jQuery függvénnyel kérjük le, így azonnal megkapjuk a végrehajtani kívánt művelet sikerességéről szóló üzenetet.
- ajax.*: különböző JSON generátor servleteket magába foglaló csomag. A különböző entitások állapotait listázza ki a Flexjson eszköztárának segítségével. Egyes servletek Hibernate Search-öt használnak az entitás(ok) megkeresésére, míg mások megelégednek az Id szerinti lekérdezéssel, míg van, ami attribútum tulajdonságai alapján végez keresést. Például a felhasználó adatainak kilistázása tartozik ebbe a csomagba.
- config.*: olyan, környezetfüggő beállítások gyűjtőhelye, melyek a rendszer alapvető működését szabályozzák, azonban nem a jogosultságkezelést konfigurálják. Egyes lokilizációs állományok gyűjtőhelye.
- database.*: az adatbázis kezelésért felelős entitás osztályokat, és az azokat kezelő DAO osztályokat tartalmazza.
- filter.*: a filterek tárolására szolgáló csomag. Jelenleg egy filter található alkalmazásomban, ami egy UTF-8 characterSetter, mely gondoskodik a helyes karakterkódolásról mind input, URL beli paraméterek helyes kódolásáról, míg a rá küldött válasz formátumáról is.
- pages.*: oszályok tárolására szolgáló csomag, aholis olyan servletek találhatóak, melyekben a kérést RequestDispatcher-rel átküldtük egy másik, WEB-INFben helyezett JSP oldalnak. Előtte szükség szerint paraméterezzük fel a kérést magát a JSP rétegbeli absztrakciónak megfelelően.

- security.*: a biztonságért felelős osztályok gyűjteménye, de itt található a security.properties fájl is, mely az egyes jogkörök hozzáférési jogosultságait tartalmazza.
- util.*: külső feldolgozók gyűjteménye. Ezek általában forrásfájlként létező, több helyen használt osztályok, melyeket kis komplexitásuk miatt felesleges JAR fájlban tárolni.

Biztonság

A mai világban egy rendszer gyenge pontjai nem sokáig maradna elfedve. Ha egy alkalmazás kikerül az Internetre, azonnal akadnak vállalkozó kedvű script-kiddie-k, crackerek, esetleg kíváncsi felhasználók, etikus hackerek, akik rögtön elkezdik keresni alkalmazásunk hibáit, hiányosságait. Természetesen egy nagyvállalatnál erre különösen kell ügyelni, hiszen nem ritkán igen nagy pénzek mozognak ezeken a rendszereken keresztül, illetve nagy kárt lehet okozni egy egyszerű szolgáltatás megtagadás (DoS) támadás kivitelezése esetén is. Ezen rendszerek feltörése, megbénítása nemcsak a közvetlenül okozott anyagi kár miatt veszélyes, hanem annak közvetett hatásai miatt is. Egy ilyen támadás miatt megrendülhet az ügyfelek bizalma a szolgáltató iránt, ami a bevételek súlyos csökkenéséhez vezet. Minden támadási módot, és azok hatását nem e szakdolgozat célja részletesen kitárgyalni, csak a webes programozási felületről kezelhető támadásokkal fogok foglalkozni.

Sokat gondolkodtam, hogyan tudnám optimálisan megoldani a biztonság kérdését. Így arra jutottam, hogy nem használom a beépített biztonsági osztályokat, mert túl komplexek voltak a feladatra. Úgy döntöttem, saját magam oldom meg a jogosultságkezelést. Így jutottam el a Properties-ben tárolt kulcs érték párok által vezérelt jogosultságkezeléshez.

Megírtam a Security osztályt, melybe a biztonsággal kapcsolatos metódusokat helyeztem el. Ezek közül a három legfontosabb a getAccessCode(String className), a getRole(String role). Mikor a felhasználó be akar lépni a rendszerbe, meghívja az UserDAO login() metódusát, mely egy lekérdezést futtat le a megadott felhasználónév jelszóval. Ha ezen lekérés eredménnyel zárul, a felhasználó beléphet, nyitunk neki egy sessiont, és azt felparaméterezzük különféle változókkal. Ezek közül a legfontosabb a felhasználó típusát megadó paraméter, hiszen ehhez kapcsoltuk a jogköröket. Három jogkört definiáltam a felhasználók számára: (adminisztrátor: admin, oktató:lecturer, és a hallgató: student). Ezen jogköröknek egyedi azonosítót adtam, melyet a getRole() metódussal kérhetünk le, paraméterként átadva a jogkör nevét. Ezt a számot a Security osztályban definiált roles nevű

HashMap-ben tárolt értékek alapján kapjuk meg. Lekérdezzük az adott nevű servlethez tartozó jogköröket kiosztó számot a getAccessCode() metódusnak a servlet nevét paraméterként átadva. Ezen a két számon bitenkénti AND műveletet végrehajtva, ha a kapott szám nagyobb, mint nulla, akkor az adott felhasználó láthatja az elemet (ha e műveleteket a showItem tagben hajtjuk végre), illetve a hivatkozott servlet végrehajtásra kerül.

Térjünk vissza egy kicsit a login() metódusra! Ismerve az MD5 sebezhetőségét, inkább nem bíztam rá a felhasználók jelszavainak tárolását. A Java MessageDigest osztálya által támogatott sokféle kódolás közül választásom a SHA-1-re esett. Ez a mód megfelelő kompromisszumot ad a sebesség, és a biztonság terén, így legyen ez az első védelmi vonal. A kódolás után kapott bájttömbön célszerű végrehajtani egy Base64 encodingot, melynek segítségével bináris, illetve karakterként értelmezve speciális karaktereket tartalmazó adatokból ASCII karaktersorozat állítható elő, mégpedig 3 bájtból 4 bájtnyi ASCII karaktert. Ha a végeredmény hossza nem osztható néggyel, a végéhez a négyes csoportokba való oszthatósághoz szükséges számú „=” jel konkatenálódik. Igyekeztem mindenből a leggyorsabb, legkisebb erőforrás igényű verziót beépíteni. Nagyon megörültem, mikor látom, hogy a Javába is van beépített osztály erre a célra. Sajnos, mint kiderült, ezen osztály használata nem javasolt, hiszen az 1.4-es verziótól kedve depricated lett, támogatása nem biztosított, használata kerülendő, mellette még meglehetősen lassú is. Rövid keresés után rátaláltam a Mikael Grev által fejlesztett encoderre, aminél a szerző állítása szerint nem létezik olyan architektúra, amely vele fel tudná megnyerni a versenyt. Természetesen ezt az állítást sikerült – ugyan csak x86-os platformon – igazolnom.

Második védelmi vonal legyen „sózás” (salting) nevű technika, mikor a jelszavainkat hashelés előtt egy fix Stringgel megfűszerezzük. Így, ha a támadó képes is megszerezni az adatbázist, a használt String, és egyéb műveletek (egymásba ágyazott hashelések, SecureRandom által generált felhasználónként változó String, stb.) ismeretének hiányában, a támadó még mindig nem tud mit kezdeni a megszerzett, és esetlegesen visszafejtett jelszavakkal. Előszeretettel használom e technikákat, nem csak dolgozatomban, hanem valós, üzleti alkalmazások fejlesztése közben is.

Harmadik védelmi vonalat az egyes JSP fájlok elrejtése a külvilág elöl való elrejtése adja. Ha egy fájlt a WAR fájl gyökerébe (Netbeans projekt esetén a web mappába), vagy almappáiban helyezünk el, az a fájl a külvilág számára egyszerűen, böngészőből lekérdezhető. Ez plusz támadási felületet biztosít az támadóknak, hiszen nem nehéz kitalálni, hogy a fejlesztő

könnyebbségből a felhasználók kilistázását az Users servletre bízta. Egykori tanárom mondta mindig, hogy az ember alapvetően lusta, így nem nehéz rájönni, hogy az autentikáció után az Users servlet könnyebbségből a kérést az Users.jsp fájlnak továbbítja. Ha ezt a fájlt közvetlenül le lehet kérni, minden benne tárolt JavaScript hívás a lekérdező számára nyilvánvaló lesz, nevétől kezdve egészen a paraméterek nevéig, típusáig. Ezzel utat nyitottunk egy külső felhasználónak üzleti alkalmazásunk belső struktúrája felé.

Ennek megakadályozására két út áll rendelkezésünkre. Egyik lehetőségként valamilyen módon magába a JSP fájlba authorizáló kódot kell elhelyezni. Ez történhet scriptlet elemek beszúrásával, illetve a JSTL c. fejezetben tárgyalt tag fájlok használatával. Másik lehetőség, ha a kényes fájlokat a WEB-INF mappába helyezzük, hiszen ekkor ezek közvetlenül nem elérhetőek. Érthető okokból én az utóbbi lehetőséget választottam.

Negyedik védelmi vonal az alkalmazásunkban azon elemek elrejtése a felhasználó elöl, melyek által meghívott szolgáltatásokhoz nincs jogosultsága. Korábban említettem a két, általam írt, a jogosultságok kezelésére alkalmazott metódust, a getAccessCode(String className), a getRole(String role). Felmerült bennem az ötlet, hogy ezeket hogyan alkalmazhatnám a felhasználói felületen levő elemek megjelenítésére, valamilyen módon a JSP fájlokba beágyazva. Ekkor merült fel bennem, egy új elemkönyvtárak megírásának gondolata. Ezen elemkönyvtárak szerkezete egyszerű, meglevő környezetbe illesztésük könnyen megoldható, hiszen csak a korábban említett taglib direktívát kell előtte meghívni a JSP oldalban. Az általam írt showItem nevű elemkönyvtár a 13. ábrán látható, míg használatának módja a 14.-en.

```
<%@ tag body-content="scriptless" import="security.Security" %>
<%@ attribute name="permission" required="true"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<%
  if (session != null
      && (Security.getAccessCode(permission) &
      Security.getRole((String) session.getAttribute("_role"))) != 0)
  {
%>
<jsp:doBody />
<%
  }
%>
```

13. ábra

```
<myTags:showItem permission="Main">
  <li><a href="Main"><span><fmt:message key="main_title"/></span></a></li>
</myTags:showItem>
```

14. ábra

A 13. ábrán is jól látható a JSP-servlet kettőség, hiszen a session attribútumot jóval korábban, az login servletben lett létrehozva, mégis használható egy tagben is. A tag body-content paraméterének scriptless értéket adva beállítottuk, hogy az adott tagbe beágyazható legyen több másik tag, ami a 14. ábrán is látható. A permission paramétert kötelező megadni a tag meghívásának helyén. Ezen paraméter határozza meg, hogy mely értéket kérjük le a jogosultságokat tároló Properties fájlból. A _role sessionben tárolt változó értéke pedig az adatbázisban tárolt szöveges hozzáférési szint, melyhez lekérjük a hozzá tartozó hozzáférési szint numerikus kódját. Mint a 14. ábrán látható, a permission változó által a Main servlethez való jogosultságokat kérjük le, amire beágyazva is hivatkozunk. Célszerű lenne a tag törzsét fixen beágyazni a showItem.tagbe. Azonban jobban bele gondolva, ezt a taget felhasználhatjuk olyan JSP, illetve HTML elemek megjelenítésének kezelésére is, amelyek nem ezt a belső struktúrát mutatják, ezért lett ilyen általánosan megírva a kérdéses tag.

Mint említettem, a hozzáférési szinteket az adatbázisban szövegesen tárolja az alkalmazásom, majd ehhez a szövegesen tárolt szintekhez rendeli hozzá a különböző kódokat. Ennek a megoldásnak nagy előnye, hogy a későbbiek során, ha új hozzáférési szintet vezetünk be, illetve egyes szintek kódját megváltoztatjuk, akkor nem kell egy, az egész Users táblát érintő frissítést végrehajtani. A hozzáférés vezérlés ezzel a módszerrel könnyen konfigurálhatóvá válik, még globális, hozzáférési szinteket érintő módosítások is végrehajthatóak.

Sajnos azzal, hogy authentikáltuk a felhasználónkat, elrejtettük a fájljainkat, és a gombokat felületről elrejtettük, még nincs teljes biztonságban alkalmazásunk a rosszindulatú támadásokkal szemben. Két nagy sebezhetőségi problémakör létezik, az Cross Site Scripting, és az SQL Injection.

A Cross Site Scripting (továbbiakban XSS) lényege, hogy beviteli mezők, vagy URL címek által saját kódot juttassuk be a célpont oldalba, ami az oldalt megtekintő áldozatnál betöltődik. Ez a saját kód ellophatja a felhasználó nevét, jelszavát, JSESSIONID-ját, illetve rosszindulatú szoftvert telepíthet a számítógépére. A támadás kivitelezéséhez elengedhetetlen a JavaScript, és a HTML nyelv rejtelmeinek ismerete, illetve szükséges a szerver oldali scriptek működésével is tisztában lenni. Kivédésére sok kész módszer létezik. Közöttük az egyik leghatékonyabb a Creative Commons Attribution-ShareAlike 2.5 licenc alá tartozó, Joseph

O'Connell fémjelezte HTMLInputFilter.[7] Ennek használata meglehetősen egyszerű, használatának betanulása pedig rövid. Segítségével könnyen ellenőrizhetjük az egyes beviteli mezők tartalmát, illetve szükség esetén az URLben kapott paramétereket is. Így megakadályozom mind az XSS fertőzött kódok perzisztenssé válását, mind az esetleges e-mailben kapott URLekre való kattintással az áldozat gépére káros kód letöltését.

Az SQL Injection olyan módszer együttes, melyek során egy adatbázist használó alkalmazásnak valamilyen módon olyan adatokat juttatunk be, hogy a tervezett SQL parancsok helyett/mellett az általunk megadottakat végzi el. Sokféle támadási módszer létezik ebben a témakörben (kezdve a legegyszerűbbektől egészen a gyakorlatilag kivitelezhetetlen SQL Smugglingig), amelyek részletes tárgyalása jelentősen túlmutat e szakdolgozat keretein, így csak az ellene való védekezést mutatnám be. Általában a biztonsági rés közvetlen okozója az, hogy az alkalmazás nem szűrt speciális karakterek szöveges mezők esetén, illetve helytelenül kezeli a típusokat. A Hibernate beépített lehetőséget kínál e támadási forma kivédésére, mivel képes validálni a kapott adatot, így ezeket csak megfelelően kell használni. Lehetőségünk van mind HQL-ben nevesített paramétert használva, mind Criteria APIt használva élni ezzel, amire példa 15. és 16. ábrán látható.

```
Criteria crit = getSession().createCriteria(getPersistentClass())
        .createCriteria("courses")
        .add(Restrictions.eq("semester.semesterId", semesterId))
        .setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY);
```

15. ábra

```
Query query = getSession()
        .createQuery("from " +
                     "User p " +
                   "where " +
                     "p.userId = :pname and p.password = :ppass")
        .setString("pname", name)
        .setString("ppass", password);
```

16. ábra

A Criteria API automatikusan ellenőrzi az entitás osztályokban megadott adatok alapján a típust, a kapott paraméter méretét, hosszát, a belső felépítése pedig lehetetlenné teszi az adatbázis felé a biztonságra veszélyes adatok átpumpálását.

[7] Weboldal: http://josephoconnell.com/java/xss-html-filter/

Ezzel szemben a HQL a hívott metódus által várt típushoz egyezteti a kapott változó típusát, és Criteria APIhoz hasonlóan ugyanúgy megszűri az adatokat.

Fogalomszótár

Felhasználó
Olyan személy, akinek azonosítója és jelszava van a rendszerben, személyes adatai tárolva vannak benne.

Oktató
Olyan felhasználó, aki vizsgákat hirdethet meg az általa tartott kurzusokból, vagy meghirdetett tárgyai, illetve vizsgái meghirdetését vonhatja vissza. Az általa kiírt vizsgákra jelentkező hallgatók vizsgaeredményeit módosíthatja.

Hallgató
Olyan felhasználó, aki kurzusokat, vizsgákat vehet fel. Az eddigi eredményei tárolva vannak a rendszerben.

Adminisztrátor
Új felhasználókat definiálhat, azokat módosíthatja vagy törölheti a rendszerből. A felhasználók tárgyait, és vizsgáit módosíthatja. Az adatbázist nagyban érintő feladatokat végezhet.

Vizsga, szak, előfeltétel, kurzus, tárgy, szemeszter
A Debreceni Egyetem fogalomrendszerének megfelelően.

Folyamatok

Belépés
Egy, az adminisztrátor által létrehozott felhasználó megadja a felhasználónevét, jelszavát a webes felületen, megtörténik az autentikációja, létrejön a session.

Aktuális tárgyak megjelenítése
A hallgatói jogkörrel rendelkező felhasználó megtekintheti saját eddig felvett tárgyait szemeszterenként lebontva.

Kurzusfelvétel
Hallgatói jogkörrel rendelkező felhasználó felvehet, és leadhat tárgyakat, mégpedig a tárgyhoz tartozó kurzusok felvételével, leadásával.

Adminisztrátori jogkörrel rendelkező felhasználó felvehet egy tárgyat hallgatói szintű felhasználónak.

Számonkérésre jelentkezés
Hallgatói jogkörrel rendelkező felhasználó felvehet, leadhat számonkérést, megtekintheti eddigi számonkéréseit szemeszterenként, és tárgyanként lebontva.
Adminisztrátori jogkörrel rendelkező felhasználó felvehet egy számonkérést hallgatói szintű felhasználónak.

Számonkérés létrehozása, módosítása, törlése
Adminisztrátori, illetve oktatói jogkörrel rendelkező felhasználó létrehozhat, módosíthat, törölhet egy adott kurzusból számonkérést, melyhez termet rendel.

Kurzus létrehozása, módosítása, törlése
Adminisztrátori jogkörrel rendelkező felhasználó adott tárgyhoz létrehozhat, módosíthat, törölhet kurzust, melyhez kurzuskódot, időpontot, termet, és oktatót rendel.

Tárgy létrehozása, módosítása, törlése
Adminisztrátori jogkörrel rendelkező felhasználó létrehozhat, módosíthat, törölhet tárgyat.

Felhasználók létrehozása, módosítása, törlése
Adminisztrátori jogkörrel rendelkező felhasználó adott felhasználót létrehozhat, módosíthat, törölhet.

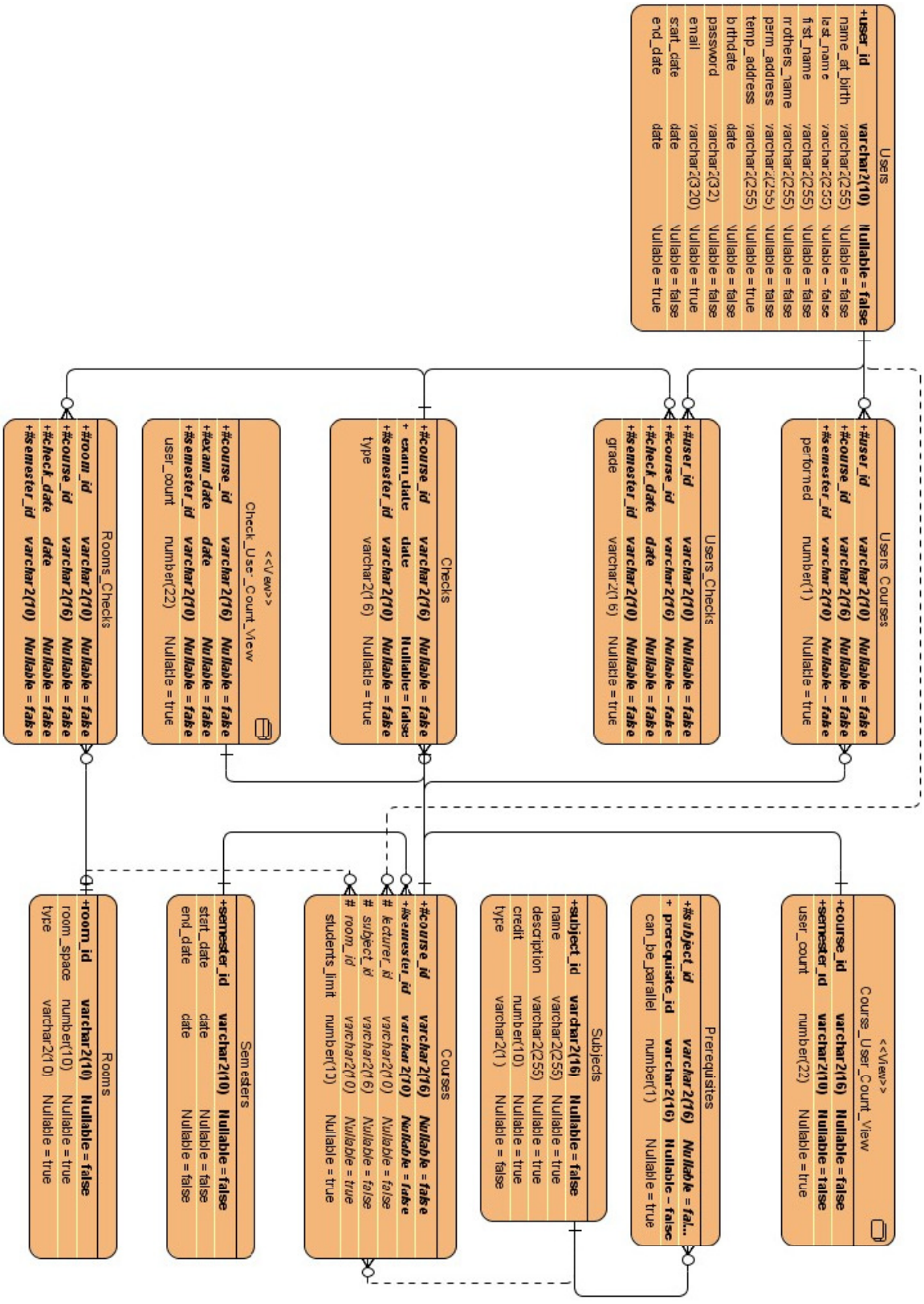
Termek létrehozása, módosítása, törlése
Adminisztrátori jogkörrel rendelkező felhasználó létrehozhat, módosíthat, törölhet egy adott termet.

Szemeszterek létrehozása, módosítása, törlése
Adminisztrátori jogkörrel rendelkező felhasználó létrehozhat, módosíthat, törölhet egy adott szemesztert.

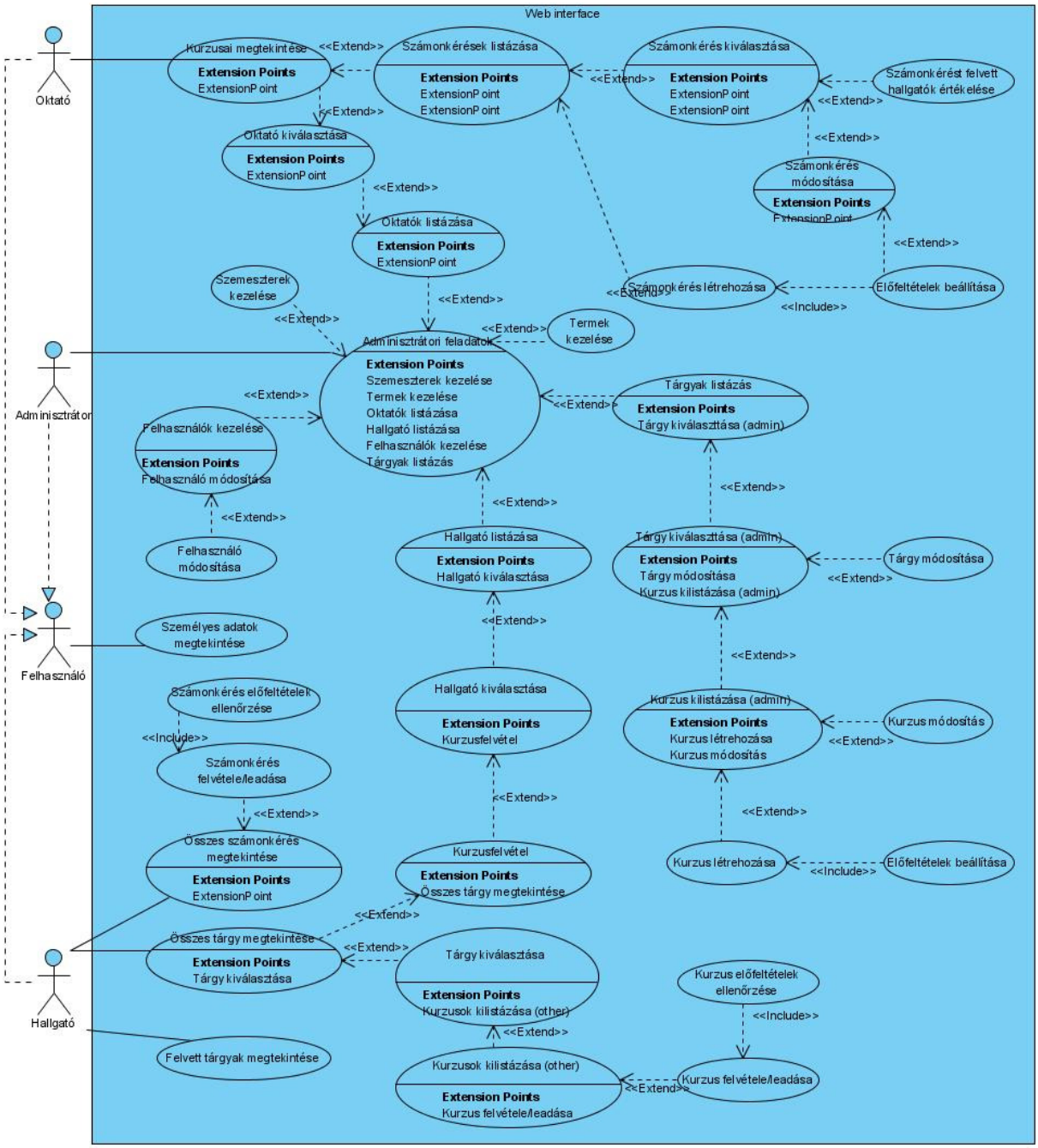
Saját felhasználó adatainak megtekintése
Felhasználó saját adatait, statisztikáit megtekintheti.

Függelékek

Adatbázisséma

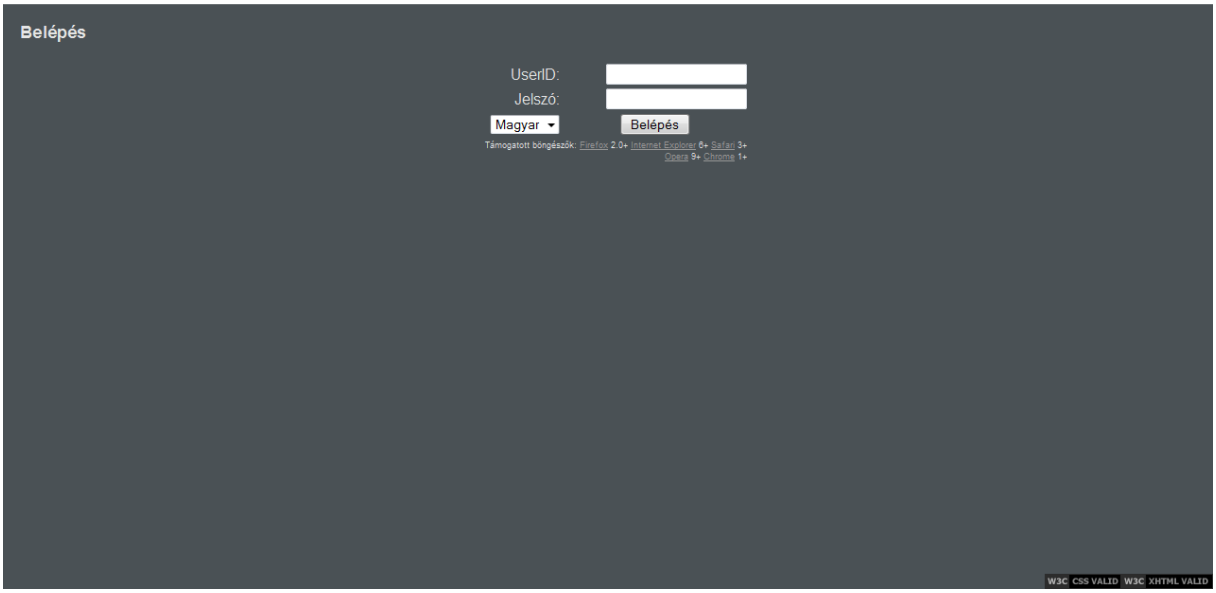


Use Case diagram

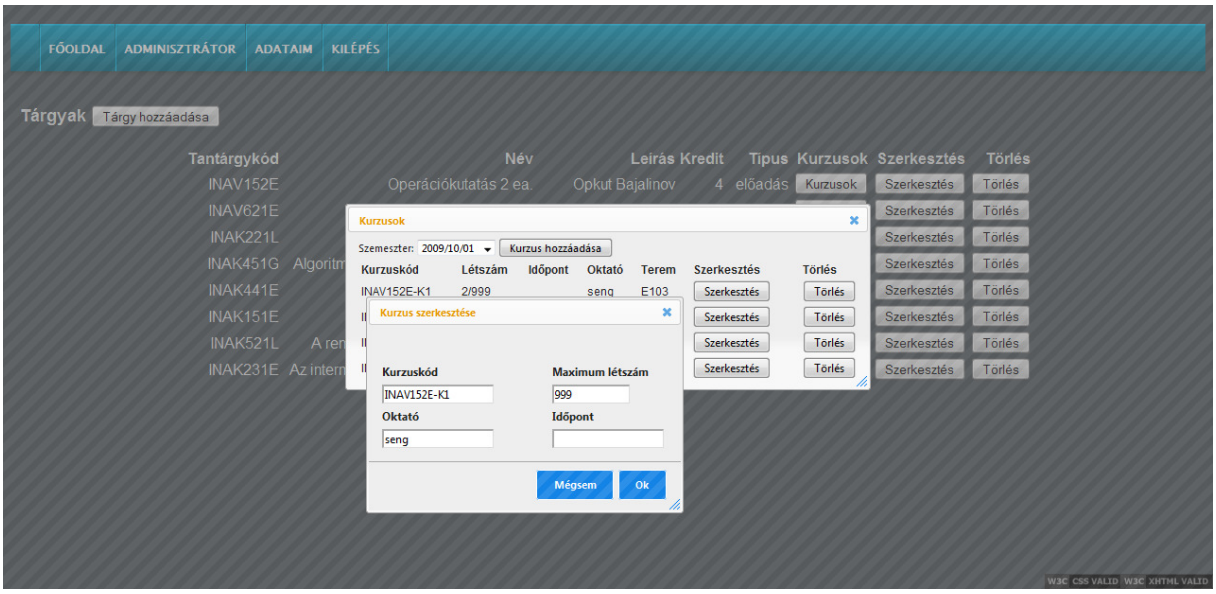


Képernyőképek

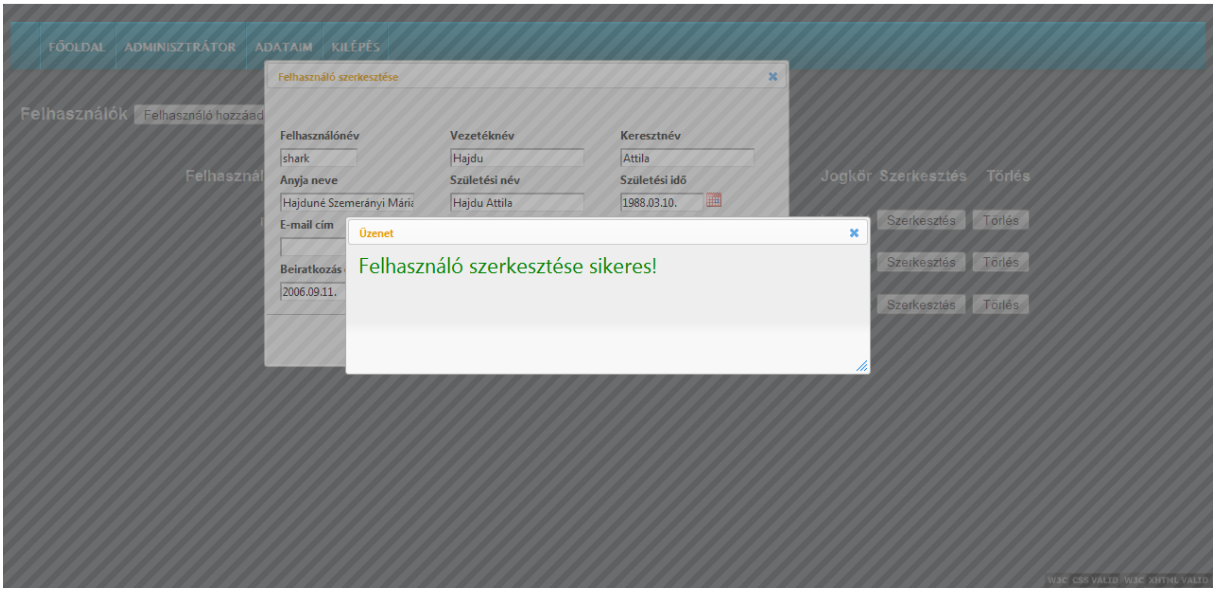
Bejelentkező képernyő:



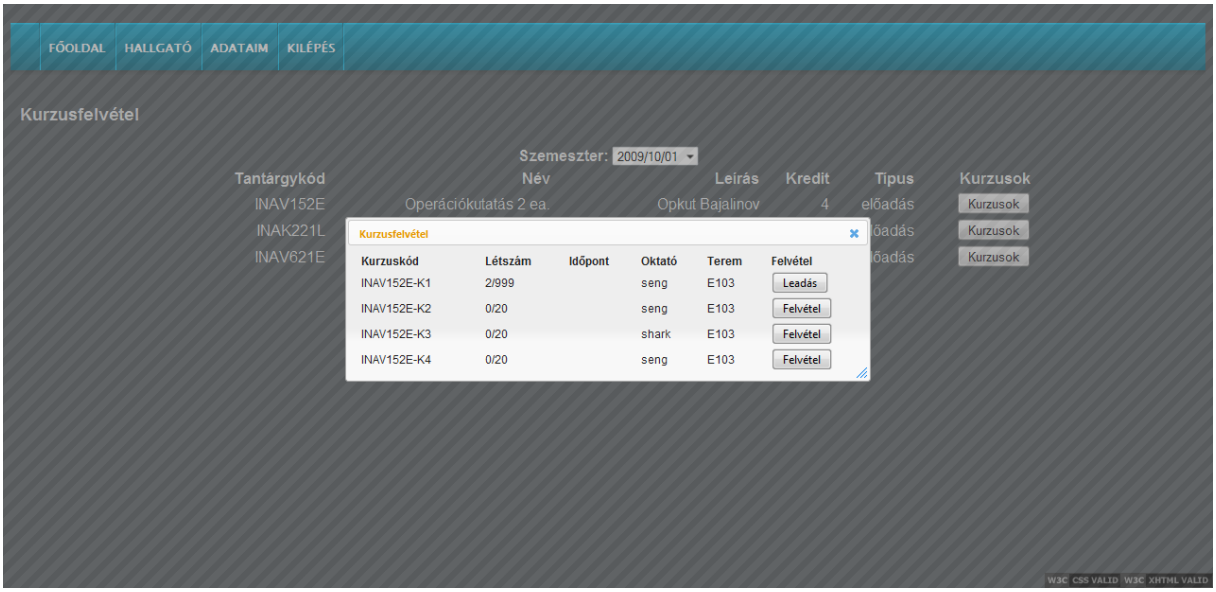
Tárgyak szerkesztése adminisztrátori felület:



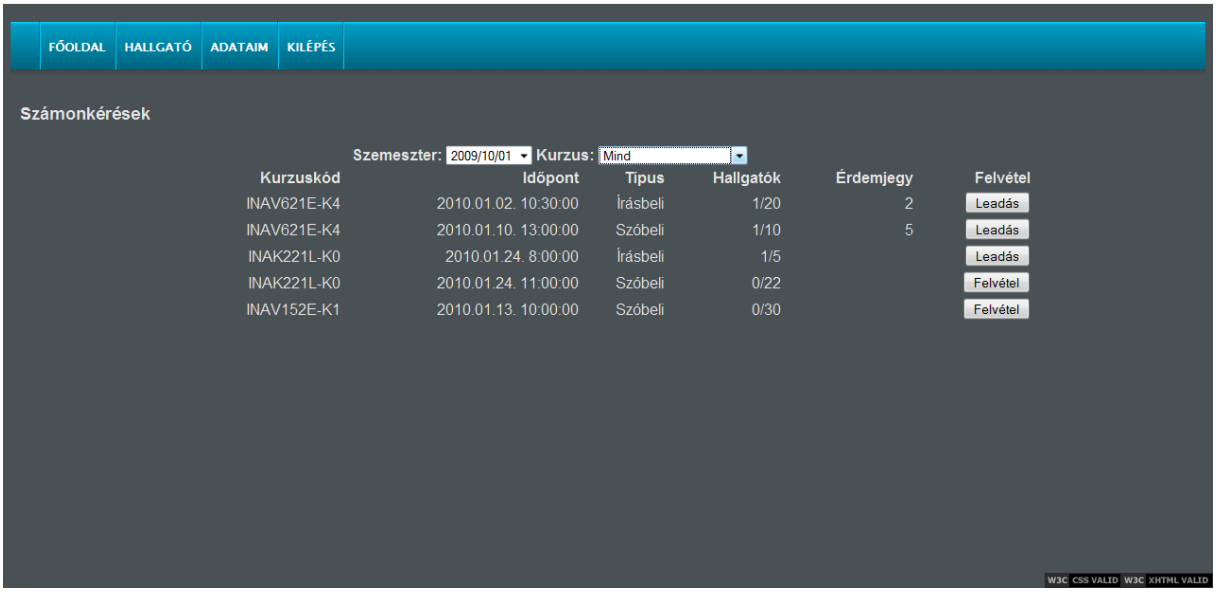
Felhasználó szerkesztése adminisztrátori felület:



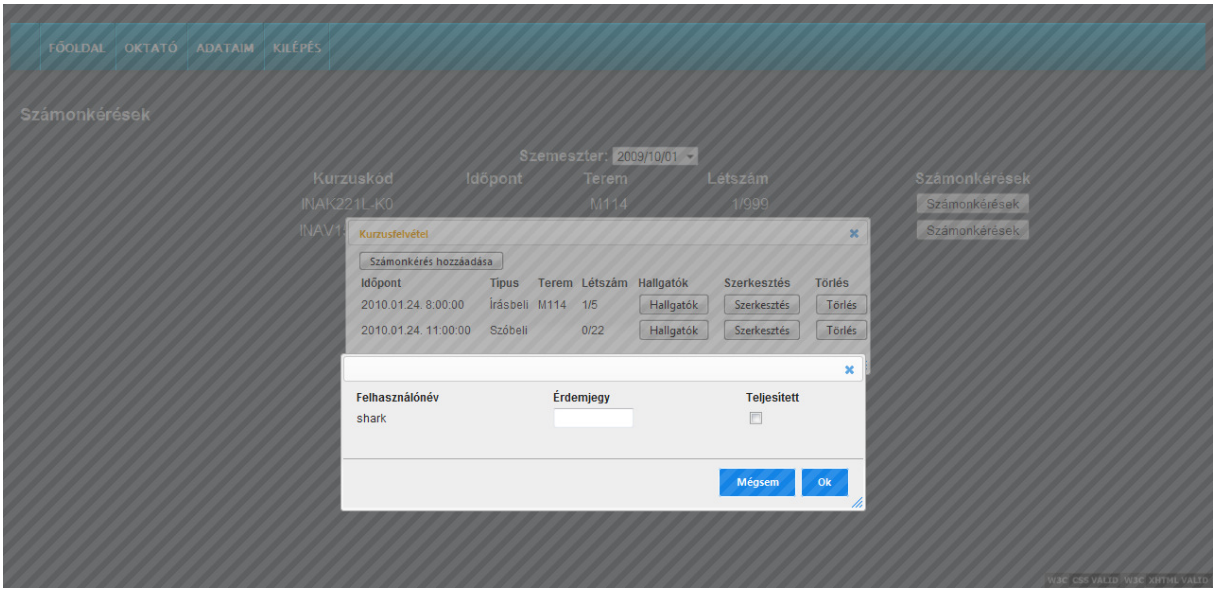
Tárgyfelvétel hallgatói felület:



Számonkérés felvétel hallgatói felület:



Számonkérés értékelése oktatói felület:



Összefoglalás

Rengeteg új technológiával ismerkedtem meg a fejlesztés folyamán. Ezen technológiák feltétlenül hasznosak lesznek a jövőbeli fejlesztéseim során. Egészen biztos, hogy nagy hasznot fogok húzni a diplomamunkám fejlesztése közben tapasztaltakból. A körültekintő tervezésnek hála, könnyen hozzáadható hozzáférési szint a demonstrátoroké, melyek hallgatók, mégis rendelkeznek oktatott tárgyakkal.

Sajnos azonban van pár funkció, melynek megvalósítására nem került sor. Korábbi terveim között szerepelt az OTP bank pénzügyi moduljának beépítése, amely azonban idő, és számos kapcsolódó funkció hiányában kimaradt. Előzetes terveim között szerepelt a jQuery cookie moduljának kihasználása is. Így a felhasználó gépére mentettem volna el olyan paramétereket, melyek a felhasználói felülettel kapcsolatos elemek megjelenítését befolyásolták volna (ilyen például a dialóg ablakok automatikus átméretezése, ha alapértelmezett méretüket a felhasználó már egyszer megváltoztatta).

Irodalomjegyzék

Sue Spielman: Practical Guide for JSP Programmers 1. kiadás, 2003

Shawn Bayern: JSTL in Action, 2002

Emmanuel Bernard, John Griffin: Hibernate Search in Action, 2009

Christian Bauer, Gavin King: Java Persistence with Hibernate - Revised Edition Of Hibernate In Action

Hibernate documentation: http://www.hibernate.org/

JQuery documentation: http://jquery.com/

Köszönetnyilvánítás

Szeretnék köszönetet mondani mindazok munkájáért, akik tudása, tapasztalata, tanítása nélkül ez a dolgozat nem jött volna létre.

Közülük legnagyobb köszönet illeti:

Kósa Márk tanár úrnak a sok segítségért, tanácsaiért, amit témavezetőként, és az általa oktatott tárgyak képében adott át nekem

Dr. Juhász István tanár úrnak a sok tudásért, és egy tanulmányi rendszer követelményrendszerének definiálásáért

Pánovics János tanár úrnak, az Adatbázisrendszerek megvalósítása 1-2 c. tárgyakon tanultakért

Krakomperger Róbertnek a Java alapú webes technológiák terén adott tudásért

Tajti Ákosnak, a Java nyelv megszerettetéséért, és szakmai tanácsokért