Debreceni Egyetem

Informatika Kar

WEBALKALMAZÁS FEJLESZÉS GOOGLE WEB TOOLKIT FELHASZNÁLÁSÁVAL

Témavezető:

Dr. Kuki Attila

Egyetemi adjunktus

Készítette:

Kiss Antal

PTI Bsc.

Debrecen
2010

Tartalom jegyzék

1 Bevezetés

1.1 Előszó

A web alkalmazás fejlesztés sokak számára még mindig egyenértékű a statikus HTML oldalak összeállításával, holott jelenleg az egyik legdinamikusabban fejlődő informatikai ágazat. Gondoljunk csak a SilverLight-ra, vagy a hamarosan megjelenő HTML5 szabványra, melyek képességeinek kihasználásával akár háromdimenziós grafika is életre kelthető egy egyszerű böngészőn keresztül.

A jelen sokkal inkább a WEB2.0 –ás felfogás, közösségi oldalak elterjedése és tartalom megosztás köré építkezik. Ezek alapja jelenleg a javascript és az AJAX. Viszont azt mindannyiunknak be kell látnunk, hogy a javascript fejlesztése nehézkes (gyengén típusos mivolta miatt), nehezen megoldható a csoportokban való fejlesztés, illetve a tesztelés és a ténylegesen böngészőfüggetlen kód fejlesztése. Ezen problémákra ad megoldást a Google által 2006. május 16-án útjára bocsájtott Google Web Toolkit (továbbiakban GWT). A különlegességét az adja, hogy a fejlesztés Java nyelven folyik, kihasználva annak sajátosságait és kiterjedt támogatottságát. Az így előálló, java nyelvű forrásállományokból böngészőfüggetlen Javascript-et fordít. Sokan helyesen AJAX framework-nek titulálják, mivel egyszerű és jól kezelhető eszközöket nyújt aszinkron szerver hívások kezelésére, ezen képességek birtokában elég hamar elterjedt. Folyamatosan jelennek meg kiválóan alkalmazható kiterjesztései (pl.: GXT), illetve egyéb nagy keretrendszerek is elkészítik a támogatását (pl.: struts, struts2, maven).

Ennek a technológiának nemrégiben jelent meg a legújabb 2.0-ás verziója, amely sok újdonságot hozott és sok vitatott problémát, hiányosságot orvosolni látszik. Ezt az alkalmat kihasználva és a technológiák gyors fejlődését némiképp követve a GWT alapjait és előnyeit egy Bugtracker alkalmazáson keresztül szeretném bemutatni.

2 Saját GWT project

2.1 Alap öltet

Korábbi munkáim során volt lehetőségem megismerkedni néhány bugtracking rendszerrel (Gforge, Google által szolgáltatott tracker), melyek nagyban megkönnyítik az egy alkalmazás életciklusa alatt feltárt hibák dokumentálását, és a hibák hatékony kezelését. Így bár az említett trackerek valójában teljesen kielégítették az elvárásaimat, mégis úgy gondolom, hasznos lehet egy ilyen rendszer elkészítése a sokrétű továbbfejlesztési lehetőségei miatt is.

2.2 Kezdeti követelmények

Az alap követelmény, hogy több külön álló projectet tudjunk kezelni. Egy kitüntetett adminisztrátor szintű felhasználó képes legyen projectet létrehozni, és a rendszerben regisztrált fejlesztőket az adott projecthez rendelni. A project tulajdonosának képesnek kell lennie arra, hogy a projecthez bejelentett hibákat rendszerben kezelje, és hibajegyeket adjon ki a fejlesztők számára. Mintegy felülbírálva a bejelentett hibát, így biztosítandó, hogy a fejlesztők ne kaphassanak hibásan kitöltött, esetenként téves hibajegyeket. A hiba bejelentéséhez nem szükséges regisztráció, ezért a rendszerrel ilyen módon kapcsolatba kerülő személyt nevezzük inkább ügyfélnek. Az ügyfélnek lehetőséget kell biztosítani a projectek böngészésére, illetve új bug bejelentésére. A rendszerben kezelni kell a hibajegyek, és a hibajelentések állapotait is, melyek egymásra hatnak. A bejelentett hibáknak és a hozzájuk tartozó ticketeknek a következő állapotokkal kell bírniuk:

- Nyitott: Egy újonnan bejelentett hiba ebben az állapotban kezdi az életciklusát.
- Feldolgozás alatt: Ha a project egyik kitüntetett felhasználója létrehoz egy bugticketet az adott hibához.
- Visszautasított: Amennyiben a bíráló úgy találja, hogy a hiba nem áll fenn.
- Lezárt: Abban az esetben, ha a hozzátartozó bugticket lezárt állapotú.

3 Technológiák és felhasznált eszközök

3.1 Java nyelv

A Java nyelv napjaink egyik legelterjedtebb objektum orientált programozási nyelve, sikerét a könnyű elsajátíthatóságának, kiterjedt támogatottságának és platformfüggetlenségének köszönheti, mely abból ered, hogy a forrásállományokat köztes byte kódra fordítja (IL), ami egy egységes környezetben fut így biztosítva, hogy az adott kód minden környezetben ugyanúgy viselkedjen.

A SUN 1990-ben kezdte el fejleszteni egyik belső projectjének keretei között James Gosling vezetésével kezdetben Oak néven. A 90-es évek közepén az internet rohamos terjedése újra felvetette a platformfüggetlenség igényét, így a projectet újraélesztették. Jelenleg a Java 1.6-os verziója a legújabb. Hamarosan érkezik az 1.7-es kiadás ismételten sok újdonságot és egyszerűsítést hozva magával.

3.2 Google Web Toolkit, GXT widget könyvtárral

3.2.1 Google Web Toolkit alapgondolatai

3.2.1.1 Böngészőfüggetlenség

A GWT alapötlete az, hogy a Java nyelvű fejlesztés kiforrottságát, a ráépülő technikák kidolgozottságát kihasználva lehessen könnyen és gyorsan aszinkron működésre képes böngészőfüggetlen JavaScript kódot készíteni. Erre született meg az a megoldás, hogy a Java forrásállományból egy külső fordító segítségével böngésző típusonként JavaScript kód készüljön. Jelenleg támogatott böngészők az IE, Firefox, Mozilla, Safari és Opera, tehát lényegében az összes elterjedtebb böngésző. A kisebb böngészők pedig nagyrészt e böngészők javascript motorját használják fel, így azokon is optimálisan és helyesen fog működni az alkalmazás.

3.2.2 Aszinkron működés

Az aszinkron működés alapját az AJAX (Asynchronous Javascript And XML) adja, melynek abban rejlik az előnye, hogy míg az adott oldal a kiszolgálóval kommunikál, maga az alkalmazás továbbműködhet, illetve weboldalak esetén megspórolható segítségével a teljes oldal újratöltése mondjuk egy egyszerű szöveg lecserélése miatt. Mint a betűszó feloldásából is látszik a kommunikáció alapja az XML illetve manapság egyre inkább a kevesebb járulékos adatmozgással és egyszerűbb feloldhatósággal működő JSON.

3.2.3 GWT-ben használatos fogalmak

3.2.3.1 RPC

A GWT RPC–t (Remote Procedure Call) használ a szerver oldal elérésére, viszont ez nem a szó szoros értelmében vett RPC, mivel ez valójában egy AJAX hívás HTTP protokoll felett, illetve az itt megjelenő service fogalom nem egyenlő a webservice-szel mivel ennek a megvalósítása tulajdonképpen egy servlet.

Az elkészített osztályok egy részéből javascript kód fordul, melyek általában a „client" csomagban találhatóak. Ezek szükségesek ahhoz, hogy el tudjuk érni a szerver oldalt, amely a csomag szerkezetben a „server" csomagban találhatóak. Egy RPC service létrehozásához összesen 3 osztályt kell elkészítenünk.

<u>YourService:</u>

Az service szinkron változata, lényegében egy szimpla interface, amit a szerveroldalon kell implementálnunk.

Implementáció:

```
// servlet-url megadása
        @RemoteServicePath(„service")
        public interface YourService extends RemoteService{
                String getAnswer();    //service metódus, ami vissza ad majd egy Stringet       }
```

YourServiceAsync:

Szintén egy interface, viszont vannak bizonyos megkötései. Először is a neve kötelezően a szinkron interface neve és az Async szó. Másodszor a benne lévő metódusok nevei meg kell egyezzenek a szinkron interface-ében lévőkkel, a szignatúrájuk és a visszatérési értékük viszont annyiban módosul, hogy az aszinkron metódusok visszatérése kötelezően void, illetve az utolsó formális paraméter egy AsyncCallback példány.

```
public interface YourServiceAsync{
        void getAnswer(AsyncCallBack<String> callback);
}
```

YourServiceImpl:

Ez az osztály kötelezően a szerveroldali csomagok valamelyikében található, valamint kiterjeszti a RemoteServiceServlet osztályt és implementálja a service szinkron interface-ét.

```
public class YourServiceImpl extends RemoteServiceServlet implements YourService{
        public String getAnswer(){
        return „My answer is true!";
        }
}
```

web.xml

Természetesen mivel servlet-ről van szó, a web.xml-ben be kell állítani az elérhetőségét.

```
<servlet>
<servlet-name>YourServlet</servlet-name>
<servlet-class>[csomagnév].YourServiceImpl</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>YourServlet</servlet-name
<url-mapping>[modul név]/service</url-mapping> //ahol a „service" az annotációból jön.
</servlet-mapping>
```

3.2.3.2 Widget

A GWT felfogás szerint a widget az, amit a felhasználó lát, tehát a felületen bármilyen módon megjelenik és tartalmaz természetesen előre definiált widgeteket.

Néhány példa:

- Button
- TextBox
- Tree
- TextArea
- Checkbox
- Radio button
- Date Picker
- Hyperlink

Természetesen definiálhatunk saját widgeteket, illetve meglévő widgetek kompozíciójából létrehozható úgy nevett compozit widget. Viszont jelen alkalmazás elkészítése során nem a saját widgetek készítése a cél, ezért külső widget könyvtárat fogok használni: a GXT-t. Ettől függetlenül kompozit widgetek elkészítése elkerülhetetlen a jól áttekinthető kód eléréséhez.

3.2.3.3 AsyncCallback

Az RPC-nél már tárgyalt AsyncCallback interface-en keresztül ér vissza az RPC-service hívás, melynek két metódusa van:

- void onSuccess(Object result);
- void onFailure(Throwable caught);

Ezek megvalósítása általában névtelen osztályban történik. A service hívás success ágban tér vissza, amennyiben a hívás feldolgozása közben nem keletkezett kezeletlen kivétel, ellentétes esetben onFailure ágon folytatódhat a feldolgozás.

Az RPC hívás során a kiszolgáló és a kliens között kizárólag szerializálható, és a kliens oldalon kezelhető osztályok példányai mozoghatnak. Ez egy esetben kifejezetten kellemetlen, mégpedig a kiszolgálás során keletkezett nem várt hibák felületen való megjelenítése során. Ekkor ugyanis kliens oldalra kizárólag arról kapunk értesítést, hogy nem sikerült szerializálni a kivételt. Ennek megoldására elkészítettem egy statikus factory osztályt, amely a paraméterül kapott Throwable példány hívási láncát (stacktrace), az üzenetét és az eredeti osztály nevét String reprezentációban egy kliens oldalon található exception példányra fordítja át. Így az onFailure-ágban információt kapok a kiszolgálás során keletkezett esetleges hibákról.

3.2.3.4 Handler

Az esemény vezérlés alap eszköze. Minden felhasználói interakcióhoz tartozik egy handler.

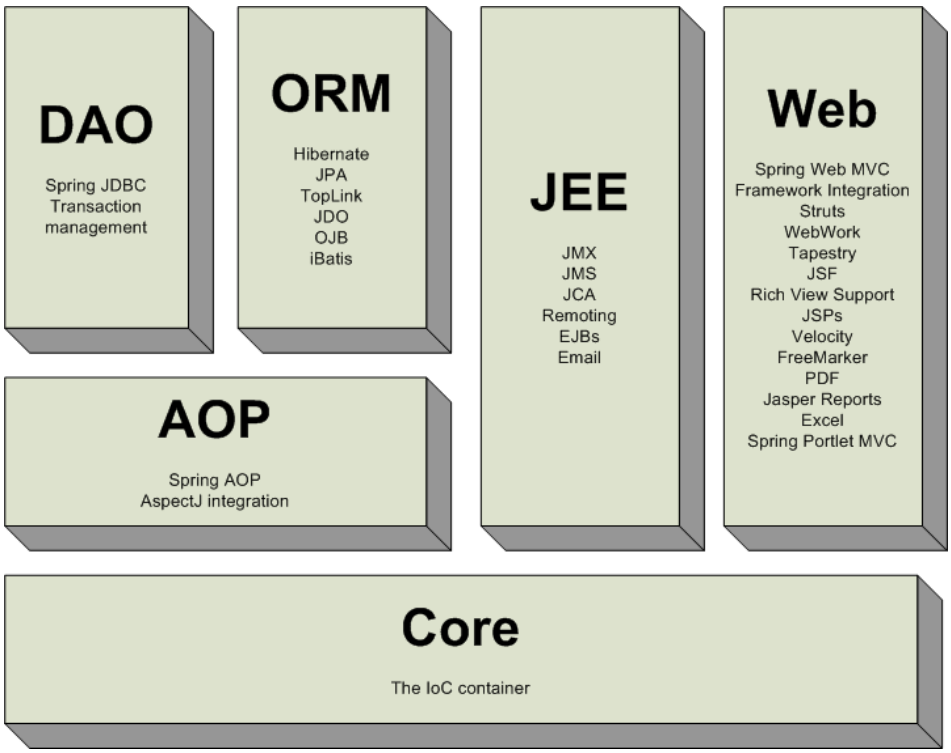
3.2.3.5 EntryPoint

A modul úgynevezett belépési pontja itt kerül összerendelésre a generált javascript kód és a HTML oldal tartalma futás időben. A RootPanel reprezentálja a HTML oldalt, ahol a script elhelyezkedik. Ennek a get(String id) metódusával kérhető el a HTML oldalon az adott azonosítóval megjelölt elem, és visszatérése szintén egy RootPanel, ahol az add(Widget w) metódus segítségével hozzá adhatunk elemeket az adott panelhez.

3.3 Szerver oldal:

3.3.1 Spring a kontextus felállításához

A Spring a közösség válasza a Sun által kiadott EJB keretrendszerre, mivel az EJB megkövetel egy teljes alkalmazásszervert a működéséhez, illetve már önmagában is nagy és komplex. A közösség elkészítette ugyan azt a funkcionalitást egy könnyebben kezelhető kisebb súlyú megoldással, amely működéséhez már nem szükséges alkalmazásszerver. Ez a megoldás később nagyon elterjedt és ma több, mint egy tucat modulból áll.



1. ábra Teljes Spring keretrendszer

Ezekből jelen alkalmazásban a következő négyet használom fel:

1. Spring IoC (Inversion of Control)

A Spring egyik Core szolgáltatása, ami az üzleti logika fejlesztését segíti és használatával szolgáltatásokat, továbbá komponenseket adhatunk az alkalmazásunkhoz. Három függőségkezelési módot ad:

- Setter injection:
  Ezzel a módszerrel a befogadó objektum setter metódusain keresztül állítja be a függőségeket. Feltételezi, hogy létezik a befogadó objektumnak paraméter nélküli konstuktora, illetve hogy van a konvencióknak megfelelően elnevezett setter metódusa.
- Construktor injection:
  A függőségek injektálása példányosításkor történik a konstruktor paraméterein keresztül. Ennek megvalósítására sorrendi és típusbeli kötés alkalmazására is lehetőséget ad.
- Interface injection:
  Annyit jelent, hogy egy interface helyére annak bármely megvalósítását injektálhatjuk a másik két módszer felhasználásával.

A bean-ek hatáskörökkel rendelkeznek, amelyek lehetnek prototype, singleton, illetve a spring 2.0 megjelenése óta request és session hatáskörűek, továbbá definiálhatunk saját hatásköröket is. Ha a kontextusleíró állományban nem nyilatkozunk másként, a bean alapértelmezetten singleton hatáskörrel jön létre, tehát egy példány létezik belőle az alkalmazás kontextusban. Prototype hatáskör esetén minden bean kérésnél új példány jön létre.

A függőségeket XML állományban kell leírni, illetve a java5 megjelenése után már annotációk segítségével is megadható, viszont ezek az alkalmazás dinamikusságának rovására is mehetnek.

2. Spring ORM (Object Relational Mapping)

Kiváló integrációt biztosít objektum relációs leképezést végző eszközök integrálására, többek között JPA (Java Persistence API), illetve Hibernate integrációval is rendelkezik. Remekül integrálható a Spring tranzakciókat kezelő moduljával.

3. Spring TX:

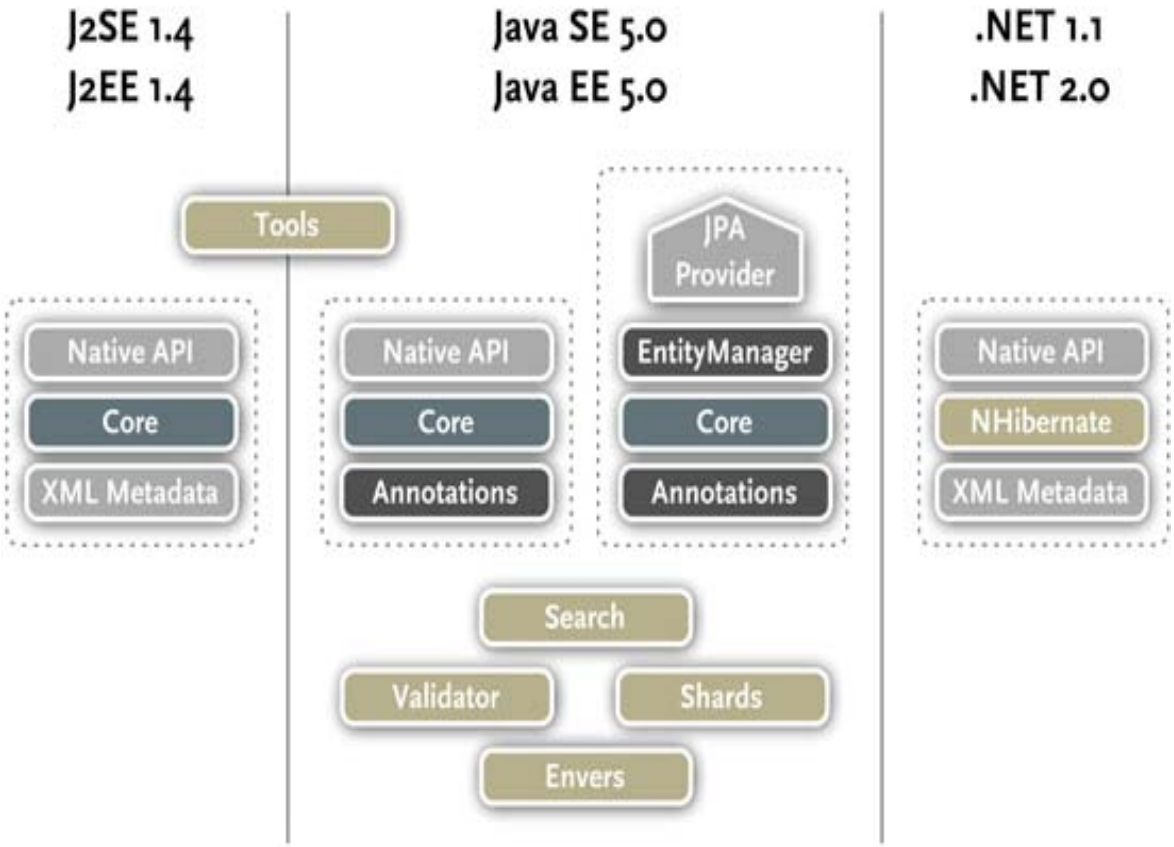
A javax.transaction csomag használatára kíván egyszerű alternatívát adni, aminek eredményként nem szükséges teljes értékű alkalmazásszerver a program működéséhez. Komplett tranzakció kezelő keretrendszer, alkalmas adatbázis tranzakciók és akár rendszerek között kiépített tranzakciók kezelésére is.

4. Spring web-MVC:

Az MVC minta köré több nagy keretrendszer is készült például struts, struts2 és a Spring-MVC is. Utóbbinak a webes használatra készített változata a web-MVC. A Control és a View kapcsolatához ad eszközöket, kezeli a jogosultságokat és az átirányításokat. Alapját egy DispatcherServlet képezi, ami a szerver oldalra beérkező hívásokat elosztja a Controllerek között. Használatával jól elkülöníthető egymástól a controller, a validálás, a view, a model és a különböző form objektumok. Jelen esetben számomra hasznos eszköze a ContextLoadListener, amely a web.xmlben bejegyezve az alkalmazásszerver indítása után felállítja az alkalmazás kontextusát.

3.3.2 Hibernate az adatbázis kezeléséhez

A hibernate egy teljes perzisztencia kezelő eszközrendszer amely több, az adatok kezelését megcélzó projectet foglal magában és a legtöbb nyelvhez létezik adoptációja. Rendkívül sokrétűen paraméterezhető, megvalósítja az objektum relációs leképezést entitás osztályok segítségével. Konfigurálását végezhetjük XML-en keresztül, illetve JPA annotációk segítségével is.



2. ábra Hibernate moduljai és adoptációi

A hibernate definiál egy saját lekérdező nyelvet (HQL), de képes natív SQL kódot is futtatni. A HQL első ránézésre hasonlít az SQL-re viszont nagy előnye vele szemben, hogy teljesen objektum orientált tehát ismeri az öröklődés, polimorfizmus és az asszociáció fogalmát. Ennek használata lehetséges HQL utasításokkal, illetve a Criteria API használatával. A Criteria API egy jobban olvasható kódolást segítő keretet ad a HQL utasítások használatára.

3.4 PostgreSQL adatbáziskezelő

A PostgreSQL az egyik legszélesebb körű támogatottsággal és legnagyobb funkcionalitással rendelkező, nyílt forráskódú adatbáziskezelő rendszer. Természetesen nem egy Oracle, viszont kis-, és középvállalatok adattárolási igényeit teljes mértékben képes kielégíteni. Ezen felül támogatja a tárolt eljárásokat, triggereket, objektum azonosításos tárolást, illetve a plsql nyelvet is. Képességeit prezentálandó néhány adat:

- Maximális adatbázis méret: nincs limit
- Maximális tábla méret: 32 TB
- Maximális sor méret: 1.6 TB
- Maximális rekord méret: 1 GB
- Maximális sorszám táblánként: nincs limit
- Maximális oszlopszám táblánként: 250 - 1600

3.5 Build folyamat és dependencia kezelés

3.5.1 Maven2

A maven2 egy kiváló fordítási folyamat, dependencia-, riport- és dokumentációkezelő eszköz egyben. A project leírását egy központosított helyen lehet megadni a project gyökerében elhelyezett pom.xml segítségével. Ezen XML a Javahoz hasonlóan támogatja az öröklődést, tehát megadható a projectnek egy szülő projectje, aminek ez által örökli a függőségeit és kiegészítéseit. Rendkívül nagy előnye, hogy a project kódbázisától elválasztva egy a felhasználó saját könyvtárában tárolja a szükséges jar fájlokat, amelyek verziókezelő rendszerben való tárolása lényegében értelmetlen. A függőségeket leíró fájlban megadva a fordítás során központi tárolókból tölti le, kezelve azok függőségeit is, így a fejlesztés során nem, vagy csak nagyon keveset kell törődnünk az egyes verzióütközések megoldásával és a jar fájlok keresgetésével. Megadható a függőségek hatásköre, ezzel például a csak fordítási időben használt eszközöket nem fogja egybe csomagolni az elkészült alkalmazással. Támogatja a fordítási profilok használatát, amik segítségével egy leírásban több különböző fordítást, illetve profilonként akár más-más függőségeket és pluginokat használhatunk.

Mióta kapcsolatba kerültem a Maven2-vel nem találtam olyan problémát, amire ne lett volna plugin. Gondot jelentett például az, hogy az adatbázist telepítés után demó adatokkal tudjam feltölteni. Alig pár perc után rátaláltam a DBUnit maven pluginra, melynek segítségével az adatbázis teljes tartalmát egy paranccsal kiírhatom egy XML állományba, és nemes egyszerűséggel egy másik parancs segítségével visszatölthetem az adatokat.

Természetesen rendelkezik a riportok generálásához is eszközökkel, például a mvn site parancs kiadása után a saját formátumában generál egy kész, statikus weboldalt, amin megjeleníti a függőségeket, az egyes riport pluginek eredményeit, a project leírását és verziókezelő elérhetőségeket is. Ennek eredményét a saját project sikerességének tárgyalásánál fogom részletesen bemutatni.

3.6 Servlet container

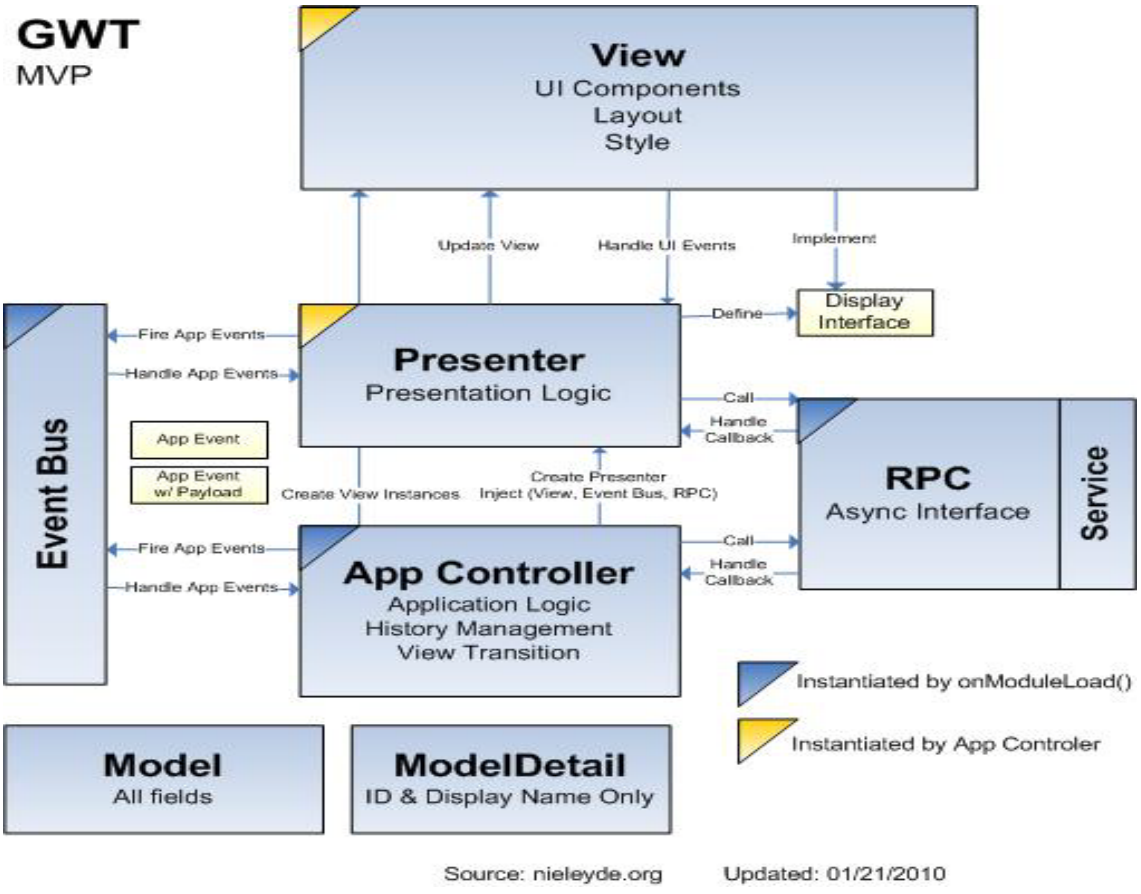
3.6.1.1 Apache Tomcat 6.0.26

Az Apache Tomcat egy nyílt forráskódú servlet-container. Rendelkezik a servletek futtatásához szükséges eszközökkel és a JSP állományok előfordításához szükséges Jasper fordítóval. A Springnek köszönhetően képességei elegendőek az alkalmazásom futásához.

4 Felhasznált tervezési minták:

4.1 MVP

Az MVP (Model-View-Presenter) minta az MVC egy tovább gondolt változata. Ezt egy kicsit másként értelmezve az MVC view fogalmának tovább bontásaként használom fel, mivel a GWT lehetőségeit, így szépen és strukturáltan ki lehet használni. A GWT alapesetben úgy nyilatkozik a felépítésről, hogy mindent az EntryPointban kell elhelyezni, és csak abban az esetben készítsünk widgetet, ha az az újrafelhasználást szolgálja. Ezzel ellentétben az MVP egy struktúrált felépítést eredményez, ahol a Model a szervertől érkező VO (Value Object avagy DTO) objektumok bizonyos kollekcióját jelenti. A Presenter a megjelenítési logikát tartalmazza, a View pedig egy kompozit Widget, ami implementálja a Presenter.View belső interface-ét, így hozzáférhetővé téve a rajta megjelenő adatokat.



3. ábra MVP pattern felépítése

Elemei:

- Model:

VO-k kollekciói, nem kerül külön tárolásra, mivel referenciaként átadódik a presenterek között.

- View:

Widgetek kompozíciója. Csak a megjelenítést szolgálja, nem tartalmazhat megjelenítési logikai kódot és semmiképpen sem üzleti logikai kódot.

- Presenter:

Megjelenítési logika, például itt valósul meg a kontextus érzékeny megjelenítés. Felhasználó jogosultságaitól függ, milyen funkciók jelenhetnek meg.

- Dispatcher:

Facade mintát követő üzenetközvetítő. Példányosításakor létrehozza a kapcsolatot a kiszolgálóval, minden szerverhívás rajta keresztül történik.

- EventBus:

A presenterek nem szerezhetnek közvetlenül referenciát egymásra, egymás állapotának befolyásolása eventbuson történik. Szabványos GWT eventeket küldenek egymásnak.

4.2 MVC (Model – View - Controller):

A régi jól bevált struktúra. Model mint perzisztencia réteg, View mint megjelenítő réteg, kontroller ami leírja az üzleti logikát, és ezek eredményétől függően változtatja a megjelenítő réteget. A megjelenítő réteg nálam továbbontásra került az MVP segítségével, így némiképp megtörni látszik az alkalmazásom MVC mivolta, mivel valamennyi üzleti logika felhozható megjelenítési logika szintjére, ennek elkerülésére kiemelt figyelmet fordítottam az alkalmazás fejlesztése során.

4.3 Factory method:

Létrehozó minta, tehát segítségével adott osztályok új példányait készíthetjük el. Lehetőség szerint igyekeztem minden többször példányosításra kerülő, jól összefogható, komplex példány létrehozására factoryt létrehozni.

4.4 Facade:

Több szolgáltató osztály feladatait foglalja össze egy közös osztályba, kiterjeszti a benne foglalt szolgáltató osztályok interfészeit és tartalmazza egy-egy példányukat melynek továbbküldi a feldolgozást. Előnye akkor jelentkezik, ha több alrendszert magában foglaló rendszerrel kapcsolatban nem fontos a kliens számára, hogy melyik alrendszer hajtja végre az adott kérést. Az alkalmazásomban ezt a különböző feladatokat ellátó üzleti logikai osztályok összefogására használom, mivel a kliens számára nem fontos ki és hogyan oldja meg a kérés feldolgozását, viszont fejlesztés közben szépen strukturálhatóvá válik az üzleti logikai réteg is.

4.5 Command pattern:

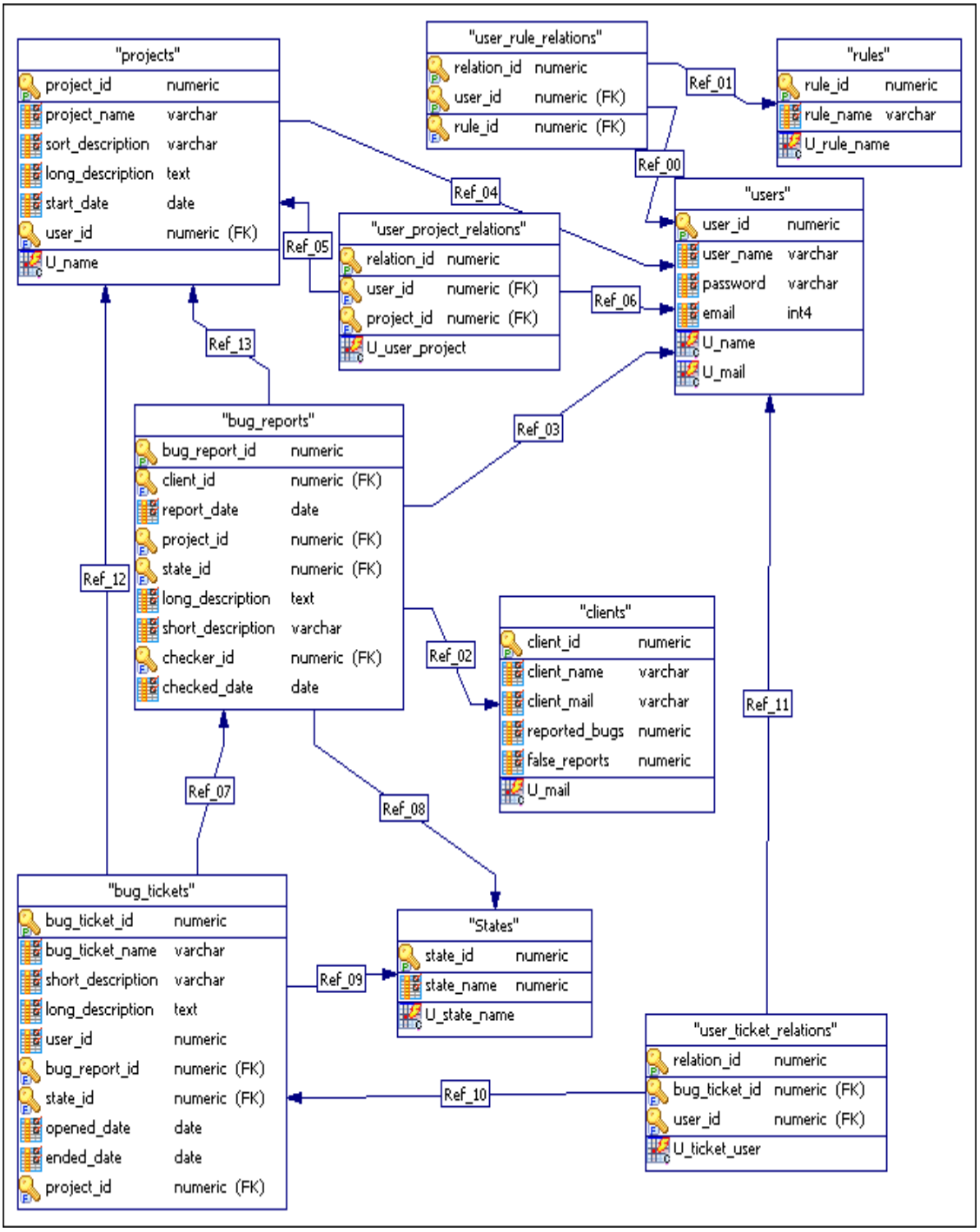
A Command pattern egy megvalósítását láthatjuk az AsyncCallback megvalósításakor, annyi különbséggel hogy ebben az esetben egy névtelen osztályt példányosítunk, és abban implementáljuk a metódust. Abban az esetben hasznos, hanem férhetünk hozzá közvetlenül a hívó példányhoz, például ebben az esetben a hívó osztály példánya nem szerializálható így a szerver oldalról nem tudjuk meghívni a kezelő metódusát.

4.6 DAO:

Az adatbázis adatainak eléréséhez a DAO (Data Access Object) mintát használom, ebben a kontextusban egy adott entitás adatainak elérését fogtam össze egy dao interface alá

5 Megvalósítás

A megvalósítást az adatbázis szerkezet megtervezésével kezdtem(4. ábra), mivel egy megfelelő adatbázis struktúra az alapja egy jól használható alkalmazásnak.



Egy bugtracker legfontosabb adatai maguk a bugok, azok tárolása és állapotuknak megfelelő módon való kezelése. Az adatbázis megtervezése során cél volt, hogy az adatokat az alkalmazás lecserélése esetén is teljes egészében fel lehessen használni.

- Felhasználók(ábrán: users): Az alkalmazásban kezelt olyan természetes személy, akiről adatokat tárolunk. Rendelkezik felhasználó névvel, kódolt állapotban tárolt jelszóval, e-mail címmel és jogosultságokkal.
- Jogosultságok (ábrán: rules): minden jogosultság rendelkezik egy elsődleges kulcsként használt azonosítóval és egy egyedi névvel.

A jogosultságok kezelése n-m kapcsolatot igényel, mivel egy felhasználó tartozhat egyszerre több szerepkörbe és természetesen egy szerepkörben lehet több felhasználó is.

- Projectek(ábrán: projects): Minden project rendelkezik egy azonosítóval, egy egyedi névvel, rövid leírással, hosszú leírással egy indítási dátummal, és egy tulajdonossal, ami külső kulccsal hivatkozik a felhasználóra.
- Felhasználó – Project kapcsolat (ábrán: user_project_relations): a projectekhez hozzárendelhetőek felhasználók. Akik a megvalósításért felelősek szintén n-m kapcsolat szükséges, mivel egy projectet több felhasználó valósít meg.
- Hiba bejelentések(ábrán: bugreports): Egy hibajelentés rendelkezik azonosítóval, rövid leírással, hosszú leírással, külső kulcs segítségével hivatkozik arra a projectre, amelyre bejelentették, illetve statisztikai szempontokat figyelembe véve a Clients táblában tárolt hibát bejelentő személy azonosítójára. Rendelkezik továbbá egy, az állapotát leíró külső kulccsal a States táblára.
- Hiba jegyek (ábrán: bugtickets): hibajelentésből készül, melyet egy felhasználó hoz létre. Projecthez és résztvevő felhasználókhoz kapcsolódik.

A következő lépésben az adatok rendszerben való kezelésére szolgáló Entitás osztályokat hoztam létre. Próbáltam kihasználni az OR mapping adta lehetőségeket, ami két okból bukott meg ebben az alkalmazásban.

Az egyik az adatokat hordozó VO osztályok feltöltése, amit a Spring-beans modul BeanUtils osztályának copyProperties metódusával oldottam meg, ami reflection segítségével a forrás osztály get metódusainak visszatérési értékével meghívja a cél osztály megfelelő setter metódusait viszont listák konvertálását nem tudja megoldani. A másik ok a hibernate3 újdonsága a fetching típusai körüli hiba, miszerint a kapcsolódó osztályok példányait két féle módon kezelhetjük. Mohón azaz azonnali tárolással ez a mód nemes egyszerűséggel nem működött, vagy késői, ami valamikor a tranzakció végén üres járatban írja ki az adatokat. Viszont ez azt feltételezi, hogy nem akarom azonnal visszaolvasni az adott tranzakcióban az adatokat csakhogy a felületnek interaktívnak kell lennie, tehát a módosításnak azonnal meg kell jelenniük felületen is. Az adatbázis elsődleges kulcsainak generálását a hibernate-re bíztam, ami az automatikus beállítással objektumszintű azonosítókat generál, azaz az adatbázis minden táblájának minden azonosítója egyetlen szekvenciából kerül ki. Ezt követően az entitásokat immár a hibernate-et segítségül hívva szerettem volna az adatbázisba leképezni, ehhez viszont szükséges volt az alkalmazás kontextusának felállítása, amelyet a Spring IoC (Inversion of Control) végzett el. A Spring kezeli a service bean-ek létrehozását és egymásba ágyazását a contextust leíró XML dokumentum alapján. Setter injection esetén reflection segítségével meghívja a bean-ek paraméter nélküli konstruktorát majd a setter a metódusok meghívásával állítja be a paramétereket, így az alkalmazás XML –en keresztül dinamikusan újrafordítás nélkül változtatható. Az én alkalmazásomban teljesen megfelelő volt a singleton ugyanis, nálam a bean-ek kizárólag üzleti logikát tartalmaznak, így nem tárolnak semmiféle állapotot. Viszont ebben az esetben elég a kontextust az alkalmazásszerver indítása után felállítani, mivel meglehetősen erőforrásigényes az XML felolvasása és a reflection-nel való példányosítás. Ennek megoldását a Spring Web-MVC nevű modulja hozta el, ahol lehetőségem van az alkalmazás web.xml-jében egy kontextusfigyelő beállítására a következő módon:

```
<listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:trackerContext.xml</param-value>
</context-param>
```

Ennek hatására az alkalmazásszerver indítása után a Spring felállítja az XML-ekből a kontextus-t, melyből már csak a bean-eket kell elkérni futás közben, ezzel lényeges erőforrás pazarlástól szabadítva meg az alkalmazást. Megpróbáltam felhasználni a Spring Web-MVC kontrollok kezelésére szolgáló megoldását is, de végül nem találtam hasznosnak, mivel a View egyetlen HTML oldalra építve kerül megjelenítésre, így nem kell átirányítást kezelni.

Ezek után az RPC szerver oldali megvalósításában a következő módon férhetek hozzá az üzleti logika szolgáltatásaihoz:

```
private ServiceFacade getService() {
    final WebApplicationContext wac = WebApplicationContextUtils
            .getRequiredWebApplicationContext(getServletContext());
    return (ServiceFacade) wac.getBean("service");
}
```

Mint láthatjuk, itt már szó sem esik az XML-ről, hanem a már felállított kontextusból kerülnek ki a bean-ek. Jelen esetben a ServiceFacade osztály egy példánya, ami magában foglalja az üzleti logikát megvalósító osztályok egy-egy példányát, amelyek metódusainak segítségével egy osztályként közös interface-t ad az üzleti logikai osztályok fölé.

Az adatbáziskapcsolatot és a hibernate munkamenet felállítását szintén a spring egyik moduljára bíztam, még pedig a Spring-ORM-re. Ezzel az adatbázis elérésének leírása szintén property fájlba került, és a fejlesztés során már nem kellett erre külön figyelmet fordítani. Definiáltam egy session factory bean-t, egy datasource bean-t, továbbá megadásra került a tranzakció és entitás kezelés módja, amit annotáció vezértelt módon valósítok meg. Az általam definiált session factory bean a Spring-ORM annotáció vezérelt session factoryját használja.

```
	<bean id="SessionFactory"
	class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
		<!-- DataSource-->
		<property name="dataSource" ref="DataSource" />
		<property name="hibernateProperties">
			<props>
				<prop key="hibernate.dialect">${db.dialect}</prop>
				<prop key="hibernate.show_sql">${hibernate.show_sql}</prop>
				<prop
key="hibernate.hbm2ddl.auto">${hibernate.hbm2ddl.auto}</prop>
<prop key="hibernate.format_sql">${hibernate.format_sql}</prop>
<prop
key="hibernate.current_session_context_class">${hibernate.current_session_context_class}</prop>
				<prop
key="hibernate.transaction.auto_close_session">${hibernate.transaction.auto_close_session}</prop>
				<prop key="hibernate.connection.autocommit">true</prop>
			</props>
		</property>
		<property name="configLocation"
value="classpath:trackerContextHibernateMapping.xml" />
	</bean>
```

Az annotációkkal történő tranzakció kezelést a Spring-TX modul valósítja meg. Ennek használata nem igényel sok beállítást, viszont fejlesztés során egyszerűbb a tranzakciókat megadni, mint xml-ben konfigurálni. Az annotációk használatához a következő sorok kerültek a leíró állományba:

```
<bean id="transactionManager"
	class="org.springframework.orm.hibernate3.HibernateTransactionManager">
	<property name="sessionFactory" ref="SessionFactory" />
</bean>
<tx:annotation-driven transaction-manager="transactionManager" />
```

A tranzakciókat üzeti logikai metódusonként állítom, mivel egy üzleti metódus minden adatbázis módusítása egy teljes üzleti folyamatot ír le, amelynek teljes tárolása kritikus, mivel adat inkonzisztenciához vezetne például, ha egy felhasználó hozzáadása során nem mentődnének le a jogosultságai. A tranzakciók indításáért, lezárásáért és hiba esetén a rollback funkciók egyaránt a Spring TX modulja a felelős. A metódusokat @Transactional annotációval ellátva adom tudtára, hogy az adott metódus tranzakció kezelt, a kezelés módját ezen annotációban paraméterként kell megadni.

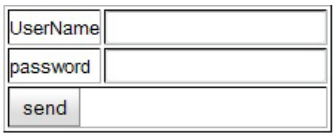
A Spring-TX a következő típusokat támogatja:

MANDATORY
Kötelezően tranzakcióban fut, viszont ha nincs megnyitott tranzakció, akkor hibát okoz.
NESTED
Párhuzamos tranzakcióban fut és ha nincs ilyen, akkor létrehoz egyet.
NEVER
Tranzakció nélkül fut, és ha van tranzakció, akkor kivételt okoz.
NOT_SUPPORTED
Tranzakció nélkül fut ez is, és ha tranzakcióban hívják, szünetelteti az adott tranzakciót.
REQUIRED
Ha tranzakcióban hívják csatlakozik ahhoz, viszont ellentétes esetben létrehoz egy újat.
REQUIRES_NEW
Minden esetben létrehoz egy új tranzakciót.
SUPPORTS
Ha van tranzakció, akkor felhasználja azt, ha nincs az sem okoz gondot.

Az alkalmazásom megfelelő működése érdekében nekem csak a Supports és a Required módot kellett felhasználnom. A gondolatmenetem szerint minden olyan üzleti logikai metódus, amely módosítást végez az adatbázis bármely elemén az Required, mivel így bármilyen hiba esetén létezik rollback pontom. Ami nem végez módosítást annak elegendő a supports, mert nem vezethet adatvesztéshez vagy inkonzisztenciához.

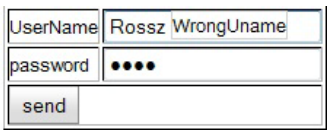
Miután a szerver oldal összeállításával végzetem képes volt az üzleti logikát rendelkezésre bocsájtani, illetve az adatbázis is elkészült elkezdtem a felületet és az RPC service-t implementálni. A felület igényeihez mérten mintegy inkrementálisan a felülettel haladva annak igényei szerint építettem ki az üzleti logika szolgáltatásait.

A login volt az első funkció, amit megvalósítottam. Az alkalmazás felépítése alapján a be nem jelentkezett felhasználó böngészhet a projectek között, illetve hibákat jelenthet be. Az alkalmazásban egy szabványos bejelentkező form található, egy felhasználó név és egy jelszó mezővel.



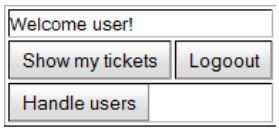
4. ábra Login panel

A hibás adatokra az alkalmazás figyelmeztet:



5. ábra Login hiba visszajelzés

Sikeres bejelentkezés esetén megjelennek a funkciók és a kijelentkezés gomb:



6. ábra Bejelentkezett felhasználó

A bejelentkeztetés egyetlen nehézségét az átirányítás hiánya okozta, viszont a bejelentkezést követően mindenképpen újra kellett tölteni az oldalt, mivel meg kellett jeleníteni a felhasználóra vonatkozó projecteket és az azokhoz kapcsolódó jogosultságokat is.

Nem tartottam jó megoldásnak minden jogosultsági szinthez külön megjelenítést csinálni, ráadásul hosszas keresgetés árán sem találtam rá használható megoldást. Így erre elkészítettem egy saját megoldást. A kliens oldalon nem illik, illetve komoly biztonsági rést jelenthet a felhasználó adatok tárolása, így ezt el kellett kerülnöm. Arra a megoldásra jutottam, hogy felhasználom a szerveroldalon a sessionben tárolt felhasználó azonosítóját, a kliensoldal jogosultság szerint kritikus elemeit pedig a Command pattern alapján egy névtelen osztály példányának success és failure metódusában kezeltem. Így a megjelenítés során egy aszinkron hívás lemegy a szerverhez, az után a sessionből elkéri a bejelentkezett felhasználó adatait, majd a szükséges jogosultság azonosítójával és a felhasználó azonosítójával az üzleti logika rétegben ellenőrzi, hogy a felhasználó rendelkezik ezekkel a jogokkal.

A szerver hívás és visszatérésének kezelése:

```
public void isUserInRequiredRule(final RULES rule, final AuthRequiredProcess process) {
    isUserInRule(rule, new AsyncCallback<Boolean>() {
        public void onFailure(final Throwable caught) {
            process.onAfterAuthFailed();
        }

        public void onSuccess(final Boolean result) {
            if (result) {
                process.onAfterAuthSuccess();
            } else {
                process.onAfterAuthFailed();
            }
        }
    });
}
```

Így az egy jogosultságkörbe tartozó elemek a következőképpen kerülhetnek ellenőrzésre:

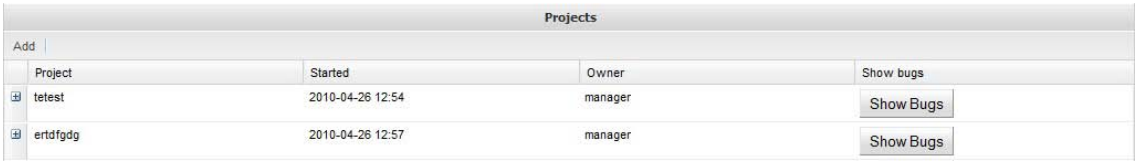
```
    getDispatcher().isUserInRequiredRule(RULES.ADMIN, new AuthRequiredProcess() {

                public void onAfterAuthSuccess() {
                 // TODO Auto-generated method stub
}
                public void onAfterAuthFailed() {
                   // TODO Auto-generated method stub
                }
            });
```

Ahol az AuthRequiredProcess névtelen osztálybeli megvalósításának egy példányát adom tovább, és az ellenőrzés sikerességének függvényében a success vagy a failed ágban tér vissza.

A táblázatok az extjs által készített GXT widget könyvtár elemei. Egyetlen kellemetlenség a használatuknál, hogy a VO osztályaimnak ki kell terjesztenie a BaseModelData osztályt, ami pedig egy Map-ben tárolja az adott osztály példány attribútumait. Ez annyiban okozott meglepetést, hogy így a GXT lib-et futásidőben is rendelkezésre kell bocsájtani. A táblázat használatához oszloponként meg kell adni, hogy hogyan is jelenítse meg az értéket, majd RPC híváson keresztül elért listát kell átadni a táblázatnak.

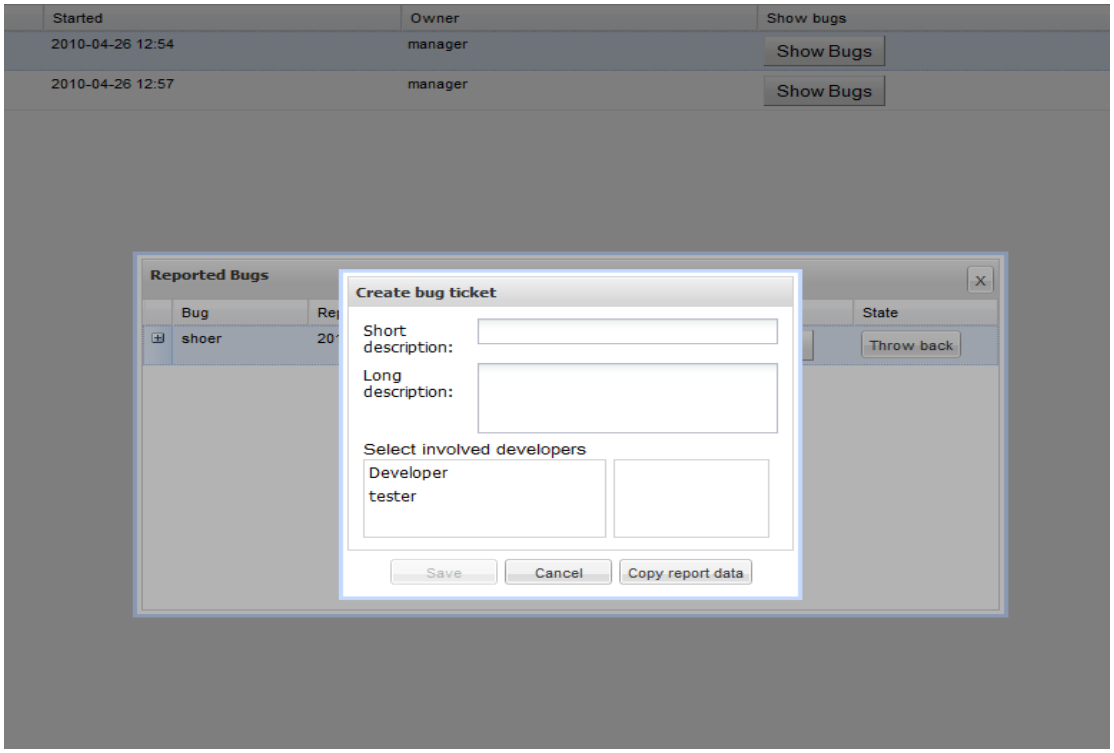
Az így elkészült táblázat:



7. ábra: Így néz ki egy GXT táblázat.

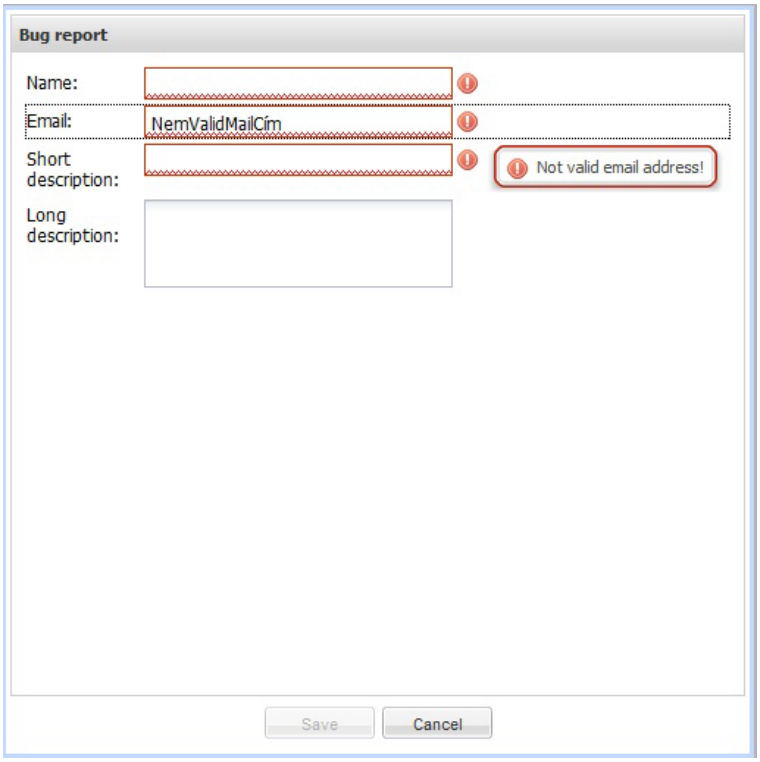
Természetesen a plusz funkciók, például a gomb megjelenítése a fejlécben, illetve az adatsorokban és a lenyithatóság további konfigurációt igényelt. Az alkalmazásban a munkamenet minden esetben egy-egy ilyen táblázatból indul, és ide tér vissza. Ezen elgondolás és az aszinkron működés miatt figyelmet kellett fordítani arra, hogy a kritikus sorrendiségű folyamatok a munkamenetnek megfelelő sorrendben hajtódjanak végre. Erre a megjelenítési logikáért felelős absztrakt osztályban elkészítettem egy

töltés, illetve feldolgozást jelző panelt, ami a töltés ideje alatt elszürkíti a felületet, miközben letiltja a felhasználói műveleteket. Ezt két metódus hívás segítségével a megjelenítési logika bármely pontján képes vagyok elindítani és leállítani. A munkamenetben is kritikus lehet, hogy a megfelelő sorrendben haladjon a felhasználó, így egy hasonló működésű panelre építettem fel a formokat.



8. ábra: Lépcsőzetes form elrendezés.

Ahogy az előző képen is látszik, ennek segítségével szinteket lehet beállítani a munkamenetben, ami jól láthatóan mutatja, mit milyen sorrendben kell haladni az adott elemek használata során. Az alkalmazásban megjelenő minden form validált, tehát amennyiben valamely adat nem felel meg az elvárásoknak, akkor azt nem engedi menteni a gomb letiltásával, illetve megjeleníti a hozzárendelt figyelmeztetést is. Az e-mail cím regexp validációval, míg a kötelezően kitöltendő adatok méret szerint kerülnek ellenőrzésre.



9. ábra: Validált form.

Az alkalmazásban szerettem volna mindenképpen felhasználni valamilyen formában a drag and drop támogatást. Erre egy egészen kézenfekvő okot adott a felhasználók projectekhez, illetve bugticketekhez való rendelésének problémája. A két értéklista között egy egyszerű mozdulattal át tudom helyezni a felhasználókat, mint ahogyan a kép is illusztrálja.

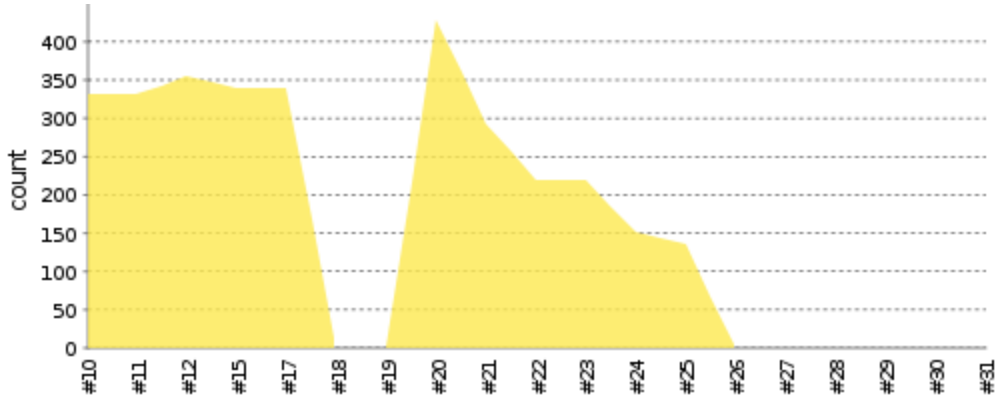
6 Konklúzió, elkészült alkalmazás minősége

6.1 Követelmények teljesítettsége

Az elkészült alkalmazás képes ellátni a kezdeti követelményben leírt funkciókat, viszont különálló rendszerként jelen funkcionalitással nem, vagy csak fenntartásokkal alkalmazható. Az ilyen és ehhez hasonló rendszereket általában nagyobb project kezelő rendszerekkel együtt működve használják. Az elkészült alkalmazás ezt a tényt szem előtt tartva került elkészítésre, ezért nem ad lehetőséget például project és bugticket módosítására, illetve törlésére, mivel egy kiadott ticket-tel a fejlesztőknek foglalkoznia kell, ami pedig a projectre szánt időből vonódik el. Így biztosítva van az, hogy minden dokumentálva maradjon, és minden esetleges hibás bugticket megmarad a rendszerben.

6.2 Kódminőség

Az elkészült kód minőségének ellenőrzésére az iparban is gyakran használt CheckStyle statikus kódelemző eredményeit vettem alapul. Saját beállításokat használtam, mivel alapesetben olyan metrikákat is használ, amelyek megfelelő teljesítése a megjelenítési oldalon gondokat okoz, például a Class-Fan-Out metrika, amely egy maximális értéket definiál arra vonatkozóan, hogy egy adott osztályban hány más osztály példányát használhatom fel. Ezt az értéket néhány megjelenítésért felelős osztályban nem lehetett nem túllépni. A statikus kódelemzés folyamatosságának biztosítása érdekében verziókezelő és folyamatos integrációt végző rendszert is használtam a fejlesztés során. Így ha nem is minden nagyobb módosítás után, de próbáltam rendszeresen futtatni az elemzést, melyről diagram is készült:



10. ábra Checkstyle trend a fejlesztés alatt

6.3 Továbbfejlesztési lehetőségek

Mint a bevezetésben is említettem, nagyon sokrétűen fejleszthető tovább az alkalmazás. Ennek megfelelően az architechtúra kialakításakor próbáltam ezt a vonalat követni, hogy minél egyszerűbben lehessen egyéb funkciókkal bővíteni a már meglévőeket, például a bug bejelentésekhez illetőleg a bugticketekhez nyitott és zárt fórumok létrehozása is egy remekül bevált gyakorlat az ilyen rendszereknél. Továbbá csatolmányok hozzáadásának lehetőségével bizonyos nyílt forrású projectek hibáinak javításában komoly segítséget jelenthez ez által a közösség. Reális igény alakulhat ki mondjuk egy report engine bevezetésére is, ami információkat ad az egy-egy hiba feltárása és kijavítása között eltelt időről, illetve az alkalmazás karbantarthatóságáról, így az esetleges ügyfelek tájékozódhatnak az adott project támogatottságáról, ami üzleti környezetben kiemelt fontosságú szempont. Megfontolandó dolog egy role engine bevezetése is, aminek segítségével pontosabban és dinamikusabban lehet a jogosultságokat kezelni. Hasznos funkció lehetne az is, ha a projecthez folyamatos integrációs eszközöket és verziókezelő rendszertámogatást nyújtana az alkalmazás, így a bugticketek javítását verziószámmal ellátott módon lehetne kezelni. Mindezeken kívül jó kiindulási alap lehet egy komolyabb projectkezelő rendszer fejlesztéséhez is, melynek egy moduljaként funkcionálhat éles környezetben. Ezek megvalósítása még másik három szakdolgozat témájául is szolgálhatna, így sajnos ezek nem tudtak bekerülni a jelenlegi alkalmazás repertoárjába.

7 Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek Dr. Kuki Attilának a szakdolgozatom elkészítése során rám fordított idejéért, és sok segítségéért. Köszönettel tartozom még munkatársaimnak, akik lehetővé tették, hogy időben elkészüljek, illetve a közös munka során elsajátított hasznos tudásért és tapasztalatokért. Ezúton szeretném megköszönni menyasszonyomnak türelmét, és a dolgozat lektorálására fordított idejét.

8 Felhasznált irodalom

8.1 Online:

Tervezési minták:

http://sourcemaking.com/design_patterns

Spring IoC leírás:

http://static.springsource.org/spring/docs/2.0.x/reference/beans.html

Spring ORM leírás:

http://static.springsource.org/spring/docs/2.0.8/reference/orm.html

MVP pattern:

http://code.google.com/intl/hu-HU/webtoolkit/articles/mvp-architecture.html

MVP kép:

http://www.nieleyde.org/SkywayBlog/post.htm?postid=37782056-c4e1-4dfb-9caa-40ab9552ca3b

Spring kép:

http://static.springsource.org/spring/docs/2.0.x/reference/introduction.html

Hibernate modulok kép és dokumentáció:

http://www.hibernate.org/

Maven2:

http://maven.apache.org/

8.2 Könyv:

ANGESTER ERZSÉBET (2003): Objektumorientált tervezés és programozás, 4Kör Bt.

JEFF DWYER (2008): Pro Web 2.0 Application Development with GWT, Apress Kiadó.

JOSHUA EICHORN (2007): Undersanding AJAX. Using JavaScript to Create Rich Internet Applications, Pearson Education.

9 Ábra jegyzék: