SZAKDOLGOZAT

Cselle Lajos

Debrecen
2011

Debreceni Egyetem

Informatika Kar

Üzleti vizualizáció

Témavezető:
Vágner Anikó
egyetemi tanár

Külső konzulens:
Bettenbuk Zoltán
IT-Services Hungary Kft.

Készítette:
Cselle Lajos
programtervező informatikus

Debrecen

2011

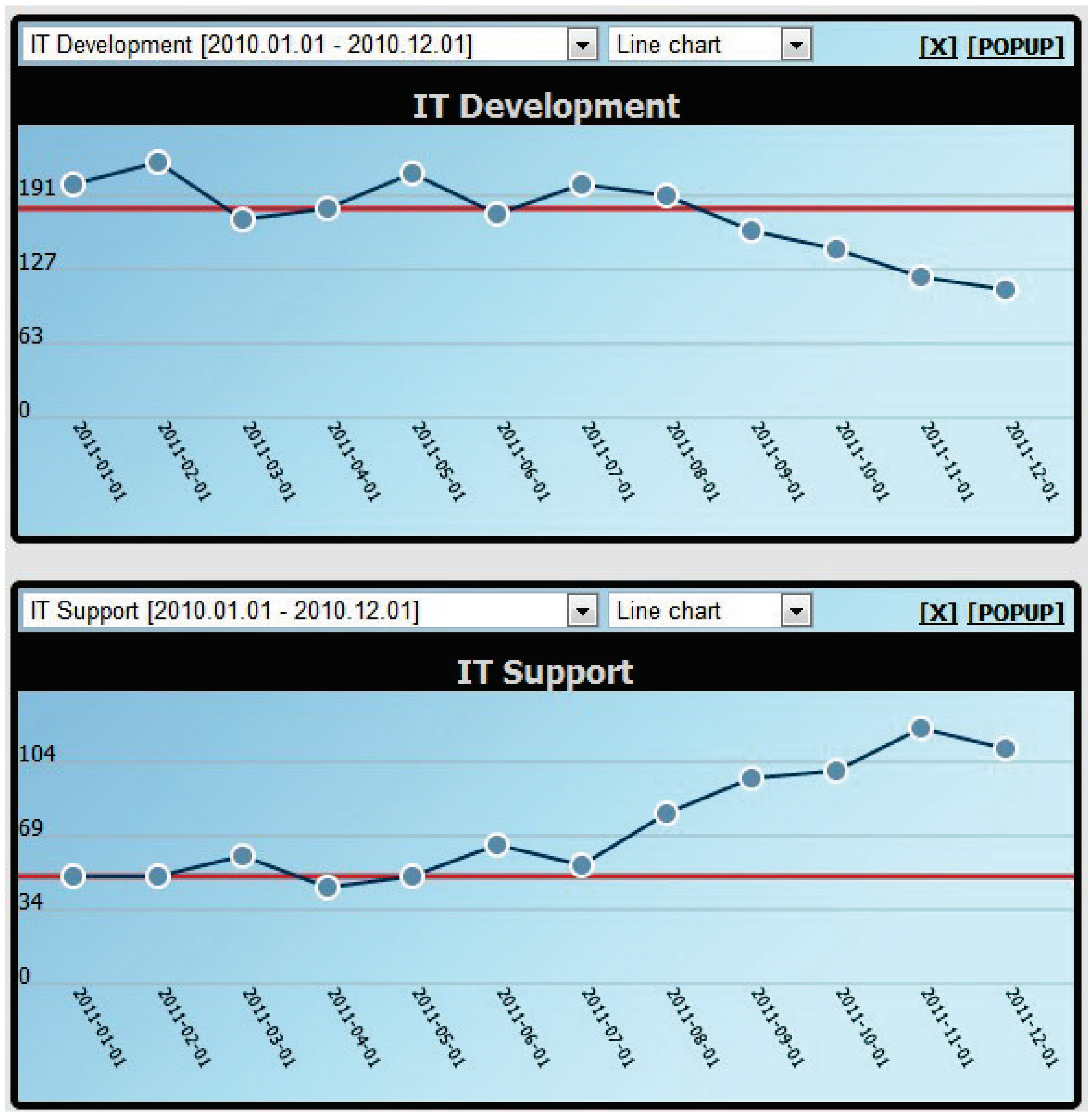
Tartalomjegyzék

1.Bevezetés....4
2.Üzleti-intelligencia (BI)....7
2.1.A vizualizáció....7
3.A Fő teljesítménymutatók (KPI-k)....9
4.Az alkalmazás....10
4.1.Bejelentkezés....11
4.2.Regisztráció....11
4.3.Dashboard....12
4.4.Chart típusok....12
4.5.Dashboard beállítások....14
4.6.Adatmódosítás....14
5.Tervezés....15
6.Megvalósítás....17
6.1.Az eszközök....17
6.2.A platform....17
6.3.A backend....19
6.3.1.Az adatbázis....19
6.3.2.A perzisztenciaréteg....20
6.3.3.Connection poolozás manuális ORM-el....21
6.3.4.Az ORM....23
6.3.5.Az EJB-k....23
6.3.6.Segédosztályok....25
6.4.A Frontend....27
6.4.1.A Google Web Toolkit (GWT)....27
6.4.2.A GWT szerver és kliens oldala....27
6.4.3.Szerver-kliens kommunikáció, avagy AJAX a GWT-vel....29
6.4.4.A Model-View-Presenter minta....32
6.4.5.Kommunikáció a felhasználóval....34
6.4.6.Böngésző előzmények és navigáció a lapok között....35
6.4.7.SVG (Scalable Vector Graphics)....35
6.4.8.A chartok felépítése és algoritmusa....36
7.Szoftverevolúció....40
7.1.A jó kód fejleszthető....40
7.2.A következő verzió....41
7.3.A jövő....41
8.Befejezés....43
9.Köszöntenyílvánítás....44
10.Irodalomjegyzék....45
11.Függelék....46
11.1.Képek jegyzéke....46

1. Bevezetés

Napjainkban az egyre terjeszkedő vállalatok elképesztő méreteket öltve hódítják meg a piacot. Ez az óriási kiterjedtség egy hatalmas veszélyt hordoz magában. Az áttekinthetetlenséget. Egy ilyen vállalkozásban mindenkinek meg van a maga pozíciója, munkaköre, amiért felel, amihez ért. A vezetőknek teljes körű képet kell kapniuk a részlegükről: mit, hogyan, mennyi idő alatt teljesítettek. Ezt az adatbázis-rekordok tömkelegéből nehézkesen tudnák kiszűrni. Ember legyen a talpán, aki egy halom számadatból képes megmondani, hogy a részlege éppen tönkremegy, vagy fennállása legkeményebb hónapjában is megállta a helyét. Az ilyen típusú problémák kezelésére különböző eszközök születtek, melyek Üzleti Intelligencia (Business Intelligence) gyűjtőnéven váltak ismerté. Ezen eszközök közül az egyik legfontosabb és leghasználhatóbb a vizualizáció vagy adatvizualizáció.
A dolgozatom célja bemutatni néhány vizualizációs eszközt és rávilágítani jelentőségükre. Bemutatom továbbá dolgozat írása alatt fejlesztett vállalati vizualizációs szoftveremet, a felhasznált technológiák és az általam követett irányvonalak ismertetése kíséretében.
Szerencsémre alkalmam nyílt a dolgozatot az IT-Services Hungary Kft-nél írni, így láthattam mennyire fontos egy több száz alkalmazottat foglalkoztató vállalatnál a temérdek adat érthető formában való prezentálása.
Kezdeném egy esettanulmánnyal, amely egy fiktív vállalaton keresztül mutatja be a vizualizáció jelentőségét. Képzeljünk el egy céget, amely 40 alkalmazottal 2 telephelyen működik. A cégben 2 osztály működik, melyek a munka jellege szerint elkülönülnek egymástól. A két telephely más országban helyezkedik el, így nehézkes a részlegek kézben tartása. A vállalat alapvetően fejlesztő és szolgáltató cég. Új alkalmazásokat fejleszt és az eladott termékek üzemeltetését és karbantartását is elvégzi. A cég több nagy szoftvert fejleszt egy időben, különböző felhasználói igényekre szabva, amely bonyolítja a támogatást. Minden felmerülő probléma egyedi, így rengeteg erőforrást emészt fel. Az erőforrások hiánya miatt az új szoftverek fejlesztési idejei lerövidülnek, de egyre több hibát tartalmaznak, ami még több terhet ró a támogatást végző részlegre. A vezetők azt látják, hogy az egyes funkciók folyamatosan kerülnek be az alkalmazásokba és az ügyfelek elégedettek. Egy darabig. A sűrűn kibocsátott, új funkciókkal bővített szoftverek egy idő után annyi javítgatásra szorulnak, hogy a megrendelők szemében lassan használhatatlanná válnak. A megrendelők nem adnak több pénzt a fejlesztésekre, a cég becsődöl.
Lássuk mi történt volna, ha a cég vezetősége rendelkezik egy vizualizációs alkalmazással:

Az alábbi ábra ezen fiktív cég egy szoftverének új funkciókkal való bővítését és a vele kapcsolatban felmerülő problémák számosságát ábrázolják egy éves intervallumban. Vízszintesen a hónapok, függőlegesen pedig az abban a hónapban a munkára fordított idő szerepel, órában megadva.



(1. ábra) Fiktív cég fejlesztési és támogatási idejei (órában)

Ebből egyértelműen kiderül, hogy az új funkciókat fejlesztő csapat túl sok hibával dolgozik, és lehetetlen helyzetbe hozza a céget. Ideje beszélni a fejlesztői gárdával, és valószínűleg időben egyenesbe állhat a cég.

2. Üzleti-intelligencia (BI)

Az üzleti-intelligencia tehát egy gyűjtőfogalom, amely magába foglalja a különböző döntéstámogató rendszereket (Decision Support System, DSS), a vezetői információs rendszereket (Management Information System, MIS) és a felső vezetői információs rendszereket (Executive Information System, EIS).

Íme néhány technológia az üzleti-intelligencia megoldások közül:

- Adattárházak
- OLAP (Többdimenziós (multidimensional) adatbázis-kezelők)
- Üzleti tervező (Planning), előrejelző (Forecasting) és konszolidáló alkalmazások
- Riportkészítő (Reporting) alkalmazások
- Irányítópultok (Dashboard), mutatószám-rendszerek (Scorecard)
- Teljesítmény-monitorozó (Performance Monitoring) eszközök
- Adat-, szöveg- és hangbányászat (Data Mining, Text Mining és Voice Mining)
- Adatvizualizáció (Data visualization)

Egy üzleti-intelligencia összeállítása több technológia együttes használatát igényli. Megkülönböztetünk Riportkészítő (Reporting), Elemző (Analisys), Monitorozó (Monitoring) és Előrejelző (Prediction) technológiákat. Ezek egymásra épülő és egyre komplexebb technológiák. Az általam oly fontosnak vélt eszköz, az adatvizualizáció egy elemző technológia melyet a dolgozatom első felében bemutatnék.

2.1. A vizualizáció

A vizualizáció egy olyan folyamat, mely során az adatokat bárki számára könnyen értelmezhető formába öntjük. Ez a forma szinte minden esetben valamilyen – az adott területet legjobban kifejező – diagram. Így egy pillantást vetve néhány diagramra, pontos képet kaphatunk a vállalat teljesítményéről. Egy adat önmagában kevés ahhoz, hogy messzemenő következtetéseket vonhassunk le. A vizualizáció segítségével olyan egymáshoz viszonyítható adatokat jeleníthetünk meg egyetlen képen, amin könnyedén észrevehetünk viselkedéseket, összefüggéseket, így óriási szerepe van az üzleti döntések meghozatalában.

Hol is használjuk a vizualizációt mint eszközt? Lássunk néhány példát az élet különböző területeiről: Bizonyára mindenki hallott már a manapság az iparban oly felkapott módszertanokról. A legjelentősebb és legmodernebb az agilis módszertanok használata. Az agilis módszertanok a folyamatosan változó követelmények kezelésére születtek, és a munkafolyamat egy-egy kisebb, kezelhetőbb részre osztásán alapulnak. Ilyen módszertanok a Scrum és a Kanban. Nem ismertetném részleteiben ezen módszertanokat, de egy közös tulajdonságot kiemelnék: mindkettő előírja a munkafolyamat láthatóvá tételét. Ez annyit jelent, hogy a kis részekre bontott feladatokat cetlikre írva, egy táblára helyezzük, amit az egész fejlesztői csapat lát és naprakész állapotban tart. Ez nagyban javítja a feladatok rendszerezését, a munkafolyamat áttekinthetőségét, tehát a projekt ezáltal tervezhetőbb lesz. A feladatokat egy táblázatkezelő soraiban is tárolhatnánk és nyomon követhetnénk de ahhoz, hogy egy ilyen táblázatból egy adott feladat tulajdonságait, állapotát és a többi feladathoz viszonyított szerepét megtudjuk, talán percekre van szükség. Ezzel ellentétben egy Scrum táblára vetett pillantással ami néhány másodperc alatt képes ugyanannyi információt eljuttatni az érdeklődőhöz.
A témában való kutatás közben bukkantam David McCandless, „Az adatvizualizáció szépségei" című előadására. Egy példát emelnék ki az említett előadásból, a „Billion dollar o-gram"-ra keresztelt diagramot. Ez az ábra a médiában elhangzott óriási pénzmennyiségek egymáshoz való relatív viszonyítását teszi lehetővé azzal, hogy az információkat vizualizálja. A képen négyzetek láthatóak, amik méretarányban vannak az általuk ábrázolt pénzösszegekkel. Egymáshoz viszonyíthatjuk az értékeket, és így érezzük igazán hogy egy összeg mennyire kicsi, vagy éppen nagy. Az azonos típusú kiadások egy színűek, így azt is láthatjuk, hogy például egy ország mennyit költ hadseregre, a közlekedésfejlesztéshez képest.
Miért ilyen hatékony eszköz a vizualizáció? Minden ember különböző, más és más módon élünk, kommunikálunk és tanulunk. De kijelenthető, hogy a legtöbbünk a vizuális típus. Sokat segít a tanulási folyamat során egy grafikon vagy egy rajz. Erre a képre asszociálva a visszaemlékezés sikeressége is valószínűbb. A minket érő környezeti ingerek 75-80%-a vizuális eredetű, tehát ismereteinket elsősorban képi úton szerezzük. Ez az oka, hogy a vizualizáció ennyire hatékony és egyszerű a tanulási és megértési folyamatokban. Mindemellett óriási jelentősége van az üzleti életben, hiszen ez az eszköz magas szintű átláthatóságot és érthetőséget biztosít vállalkozásunk üzleti folyamatainak követéséhez. De vajon mik azok a tényezők, amiknek szerepelnie kell az egyes diagramokban ahhoz, hogy valós információt láthassunk a részlegünkről? Ezt az utat jelölik ki számunkra a KPI-k (Key Performance Indicator) avagy a fő teljesítménymutatók.

3. A Fő teljesítménymutatók (KPI-k)

A teljesítménymutatókat a cégek a teljesítmény mérésére használják, és egyre elterjedtebb főleg a nagyvállalatok körében. A KPI-k megtervezése nagyon fontos dolog, olyan ember kell, hogy végezze, aki tisztába van azzal, hogy mit igényel a vállalkozás, mi lenne a leghasznosabb a fejlődéshez.

A mutatókat kategorizálhatjuk az alábbi módon:

- Mennyiségi (quantitative) mutatók: egyszerű számadatok reprezentálása
- Tapasztalati (practical) mutatók: a cég folyamatainak ábrázolása
- Irány (directional) mutatók: tipikusan az derül ki belőle, hogy javult vagy romlott-e valami
- Változási (actionable) mutatók: a változások hatásainak ábrázolása
- Pénzügyi (financial) mutatók: teljesítmény indexek megállapítása (teljesítménymérés esetén)

A fő teljesítménymutatóknak általában 5 jellemzője van:

- Név: a mutató neve, aminek ki kell fejeznie, hogy milyen adatokra vonatkozik a KPI
- Elvárt érték: az az érték amit szeretnénk, hogy elérjen (közelítsen) minden a KPI-ben foglalt érték.
- Értékek: a mintavételezett értékek, amik meghatározzák a KPI-t
- Intervallum: egy idő-intervallum ami alatt az értékeket rögzítettük
- Mértékegység: az értékek mértékegysége (óra, Ft…)

Figyelembe vehetünk egy hatodik tényezőt, amely segítségével pontosabb képet kapunk. Azt, hogy az adott érték hány megfigyelés alapján kaptuk meg.

4. Az alkalmazás

Dolgozatom második részében bemutatom az általam készített vizualizációs eszközt, melyet nemes egyszerűséggel „Visualization Dashboard" névre kereszteltem.

A program egy belső, vállalati szoftver amelytől az alábbiakat várhatjuk el:

- magas rendelkezésre állás
- felhasználók hierarchikus kezelése
- külső rendszerekkel való együttműködés (adatok importálása)
- adatok megbízható tárolása
- több típusú adatvizualizáció

Az alkalmazás tehát külső adatforrásokból előzetesen importált KPI adatokat képes különböző módon megjeleníteni a felhasználók számára.

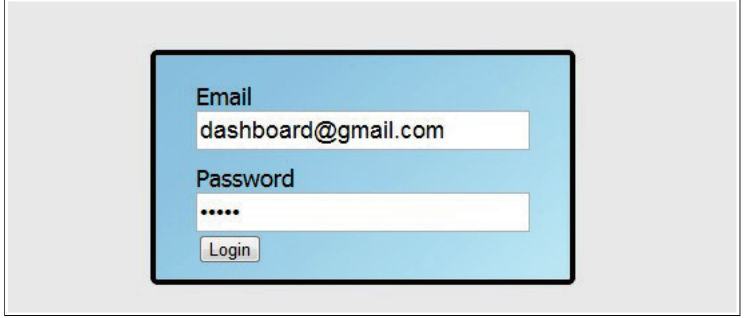
Az elsődleges szempont egy ilyen típusú vállalati szoftvernél a rendelkezésre állás és a könnyű telepíthetőség. Az internet mindenhol ott van bárhová nézünk: elérhető a számítógépünkön, notebookunkon és már a mobiltelefonunkon is. Ezért döntöttem a webalapú alkalmazás mellett, amely magas rendelkezésre állása, illetve egyszerű használhatósága miatt oly felkapott napjainkban. Az alkalmazást egy központi szerverre telepítve mindenki számára elérhetővé válik, és egy böngésző segítségével bárki használatba veheti. Ezután az üzemeltetésen túl a karbantartás és a szoftver evolúció is könnyebb, mint az asztali alkalmazások esetén, hiszen a rendszert a központi szerveren módosítva, mindenki az új verziót használhatja tovább.

A következő cél az alkalmazás egyszerű használhatósága. Ehhez a manapság népszerű és hatékony műszerfal (dashboard) típusú felépítést használtam, ahol a fő szempont a letisztultság és könnyű kezelhetőség.

A szoftver tehát webes alkalmazás, böngészőben használható bejelentkezés után. Lássuk az egyes funkciókat.

4.1. Bejelentkezés

A bejelentkezés egy e-mail címmel és egy jelszó beírásával történik meg, amennyiben a felhasználói fiók létezik, a felhasználó a saját kezdőoldalára vagy workplace-ére kerül. Hibás adatok megadása esetén a rendszer egy hibaüzenettel jelzi a sikertelen bejelentkezést. Bejelentkezni tehát csak regisztrált felhasználók tudnak a rendszerbe.



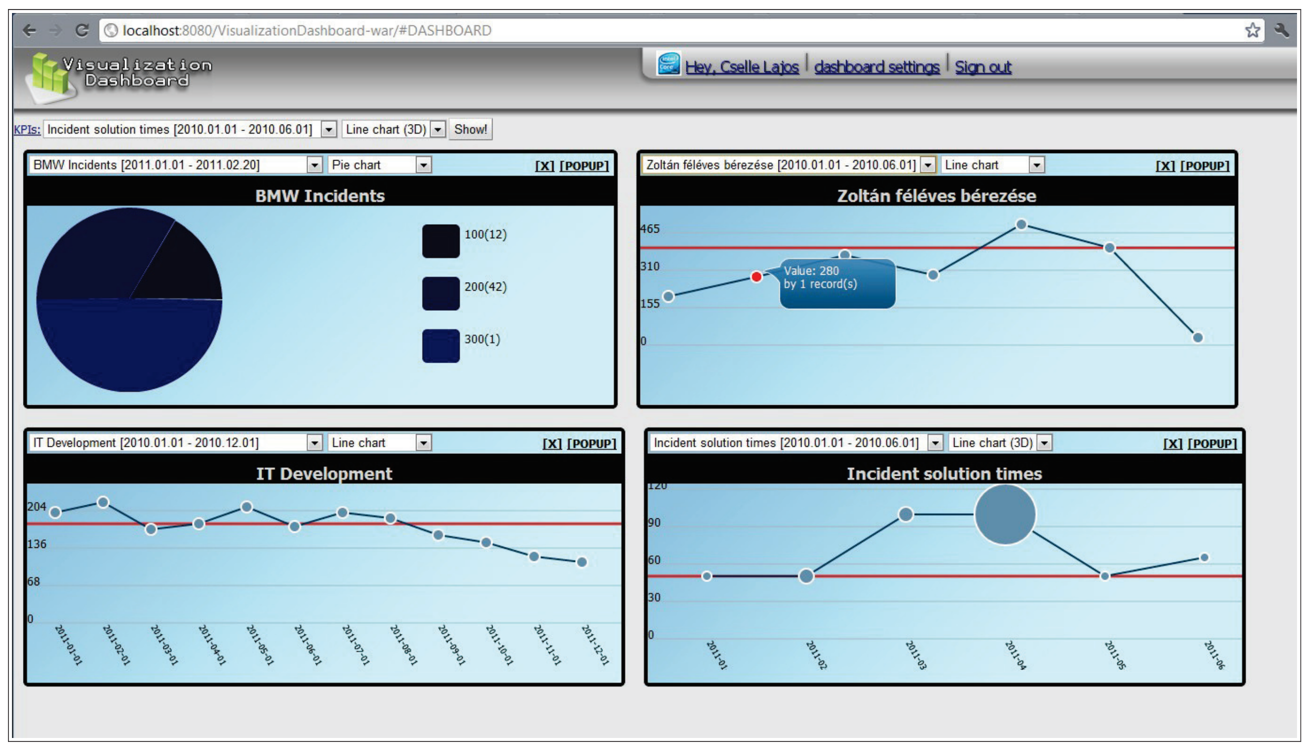
(2. ábra) A bejelentkezési képernyő

4.2. Regisztráció

Véleményem szerint a nyílt regisztráció egy ilyen jellegű vállalati rendszernél nem megfelelő és indokolatlan. Ezért a jól ismert meghívás alapú regisztrációt valósítottam meg. Egy már regisztrált felhasználó a meghívás során egy e-mail címet ad meg, amelyre egy meghívó küldődik el. Az e-mail tartalmaz egy linket, amelyre kattintva a meghívott a regisztrációs lapra navigál. Itt alapvető adatokat (név, jelszó) kitöltve bekerül a rendszerbe mint felhasználó. Meghívót az küldhet, aki a programban regisztrálva van. A meghívók száma nincs korlátozva, nincsenek különböző rangú felhasználók, de egy bizonyos hierarchia épül minden új felhasználónál, amelyet az alábbiakban részletezek.

4.3. Dashboard

A dashboard vagy műszerfal a kezdőképernyő a bejelentkezett felhasználóknak. Egy listából választhatunk KPI-ket, melyeket különböző grafikonokon (chart-okon) jeleníthetünk meg. Két típusú KPI jelenik meg a listában: a saját magunk által létrehozottak, és a minket meghívó felhasználó által létrehozottak. Így, a meghívás során elfogadjuk, hogy a meghívott egyén a saját KPI-ket is megtekintheti. Miután kijelöltük, hogy melyik KPI-t milyen típusú chart-on szeretnénk megjeleníteni, a „Show" gombra kattintva megjelenik a diagram a dashboard-on. Tetszőleges számú KPI jeleníthető meg a felületen, ezek kettesével egymás alatt fognak elhelyezkedni. Az alkalmazás figyelembe veszi a képernyő méretét, illetve felbontását és ennek megfelelő méretű diagramokat generál, melyek egy kattintással a képernyő teljes méretére nagyíthatóak vagy törölhetőek a felületről.



(3. ábra) Visualisation Dashboard

4.4. Chart típusok

Három előre definiált diagramon jeleníthetők meg a KPI-k.

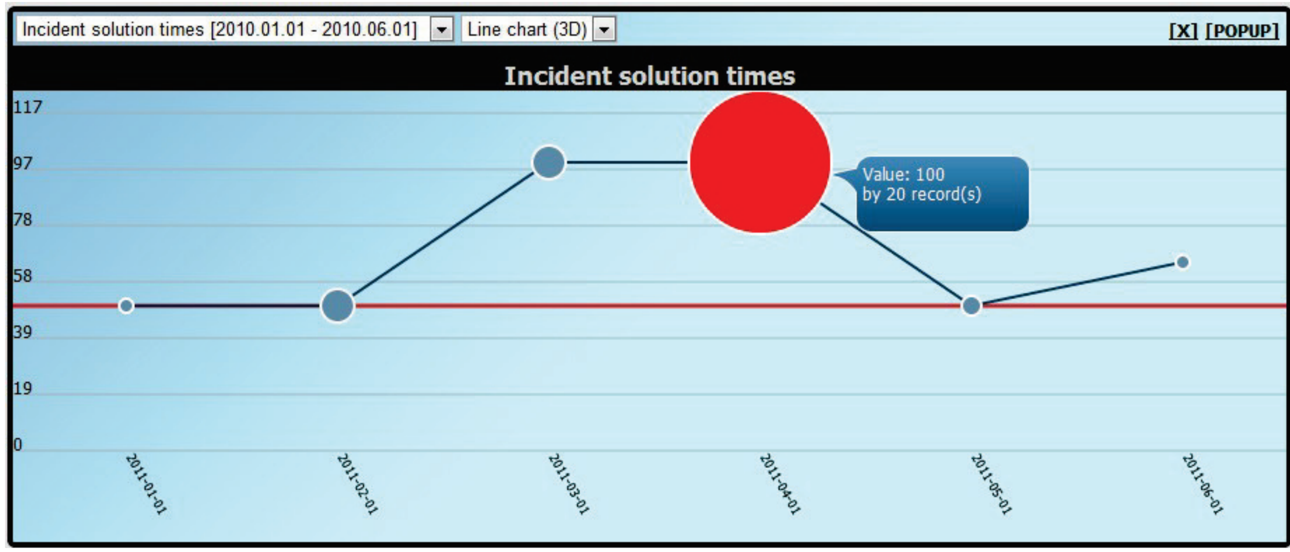
- Kördiagram (pie chart), amely a kört arányos szeletekre osztva jól érzékelteti a KPI értékek egymáshoz való viszonyát. A különböző színű szeletekhez jelmagyarázat tartozik amely a

diagram mellett, jobb oldalon jelenik meg.

- Vonaldiagram (line chart), amely az adatokat x és y koordinátán ábrázolja, x koordináta az idő eltelte (illetve a KPI értékek egymásutánisága) az y pedig az érték nagyságának függvényében változik. Az értékeket körök jelölik, amelyek egyenes vonalakkal vannak összekötve.
- Háromdimenziós vonaldiagram (3d line chart): ez a típusú diagram bevezet egy harmadik dimenziót az adatok ábrázolásakor, ez az adott érték nyomatékosságát jelzi a többihez képest. Hasonlóan a két dimenziós társához, körökkel jelzi az értékek helyét a grafikonon, de a körök mérete aszerint változik, hogy azt az értéket mennyi mintavételezés alapján szereztük.

Az első két típus triviális és jól ismert grafikon típus, de a 3 dimenziós vonaldiagramra bemutatnék egy életszerű példát.

Célszerű ilyen típusú diagramon ábrázolni a részlegünkre háruló probléma megoldási időket. Az x koordináta a hónapokat, az y pedig a problémákra fordított időt jelenti órában. Itt a körök mérete nagyon sok információt hordoz magában hiszen azt jelzi, hogy az adott hónapban hány problémát oldottunk meg. Mivel nem mindegy, hogy egyik hónapban azért 100 óra a probléma megoldások ideje, mert 50 vagy 5 probléma volt. Az első esetben a részlegünk gyorsan oldja meg a problémákat, valamelyik más részlegen keresendő a hiba, mivel valaki pontatlanul dolgozik, míg a második esetben a kevés hiba elhárítása vett igénybe sok időt, amit valószínűleg az ezen a részlegen dolgozó emberek lassúsága okoz.



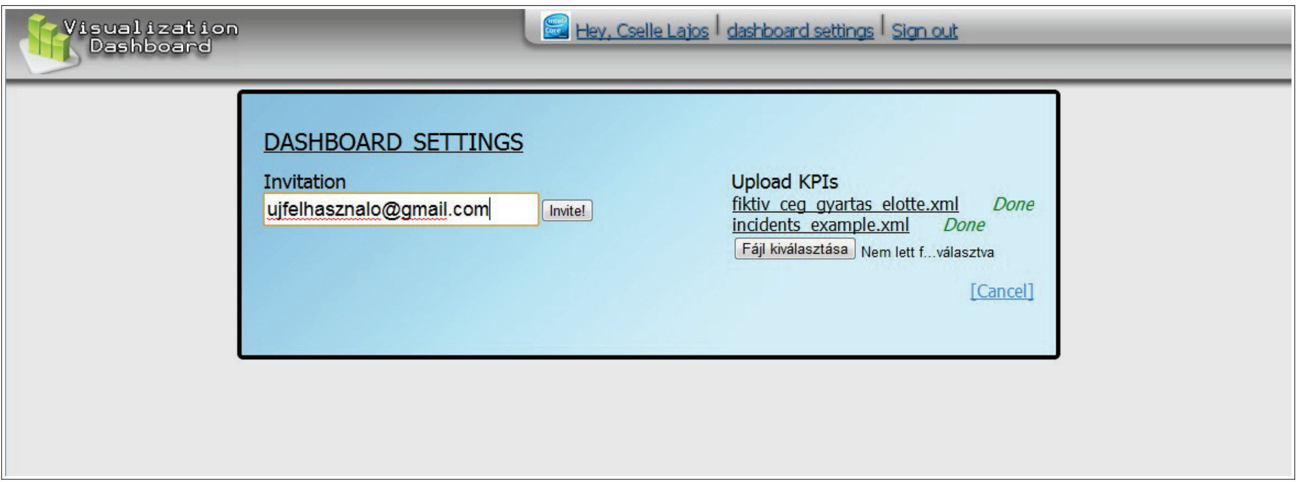
(4. ábra) A háromdimenziós vonaldiagram használati esete

A diagramon látszik, hogy nem áprilisban volt a leglassúbb a probléma megoldásokkal foglalkozó csapat, csupán több feladatuk volt akkor, mint a többi hónapban.

4.5. Dashboard beállítások

Ezen a felületen két funkciót találunk:

- KPI-k feltöltése: Az adatok bevitele egy előre definiált formátumú XML fájl feltöltésével lehetséges melyet a „dashboard settings” menüpont alatt végezhetünk el. A pontos formátumról a későbbiekben még szót ejtek.
- Új felhasználó meghívása e-mail cím beírásával.



(5. ábra) Dashboard beállítások oldala

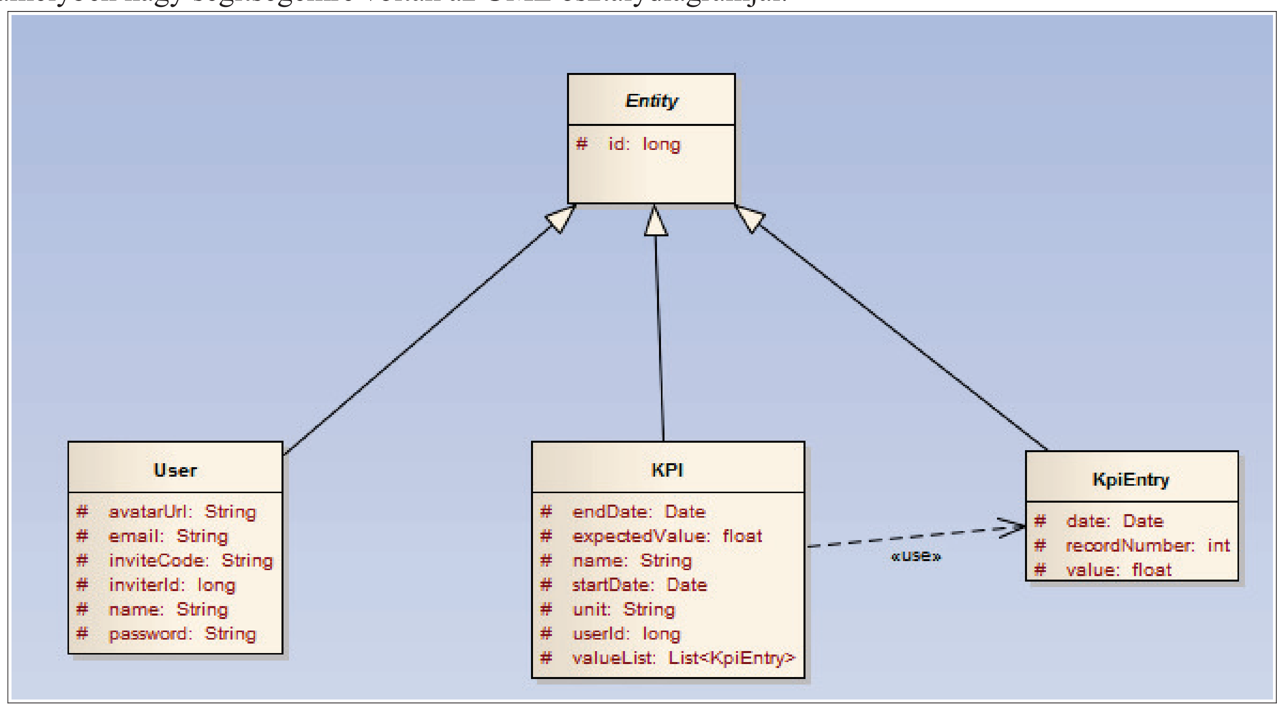
4.6. Adatmódosítás

A regisztrációkor megadott adatok (név, e-mail cím, jelszó) ezen a felületen módosíthatók.

(6. ábra) Adatmódosítás oldala

5. Tervezés

Az alkalmazás első víziója egy táblára rajzolt KPI chart volt, amely a külső témavezetőmmel való eszmecsere alatt született meg. Az ezt követő hetekben számos rajz és skicc készült, melyek nagy része a kukában landolt, mondván ennél letisztultabb dizájn kell. Végül sikerült egy a mai trendekhez hasonló egyszerű stílust eltalálni. A tervezés következő lépése az objektum-hierarchia felvázolása volt, amelyben nagy segítségemre voltak az UML osztálydiagramjai.

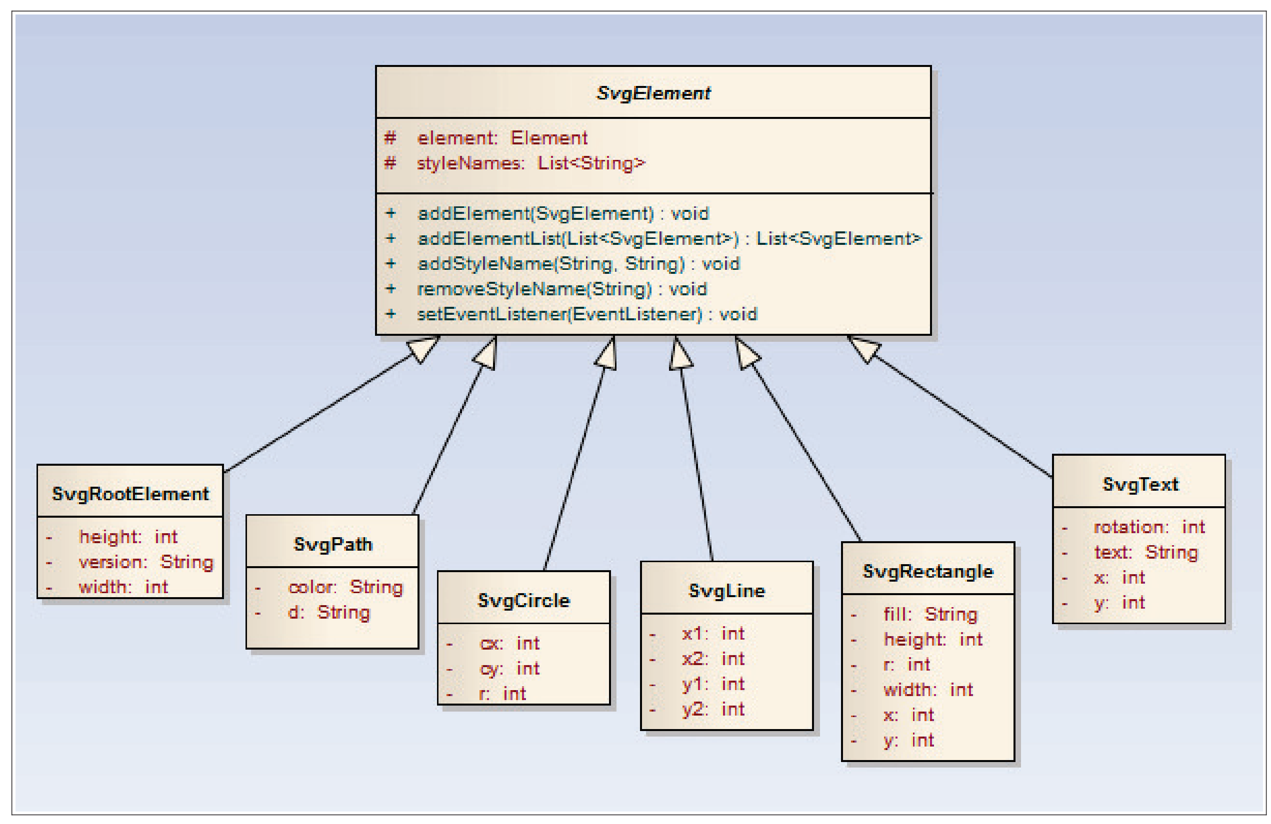


(7. ábra) Az entitások osztálydiagramjai UML-ben

Ezek alapján már egyszerű volt elkészíteni a relációs adatbázis ER sémáját is.

A rajzolgatás után elkezdődött az architektúra tervezése. Miután a **Java Enterprise Edition** platformot választottam, tisztán látszott hogy szükségem lesz néhány Enterprise Java Bean-re, amelyek az alkalmazás logikáját adják (ezekről később).

A platformfüggetlen tervezés utáni első lépések, próbálkozások voltak egy sima Java SE projektben. A próbálkozások egy SVG fájl létrehozására irányultak, amely egy tömbben tárolt egész adatokat jelenít meg egy egyszerű vonaldiagramon. Az SVG kód egyszerű *String* konkatenációk sorozatával állt elő, ami mellé natív JavaScript kódot írtam, amely segítségével némi dinamizmust vihettem a chart életébe. Ez szemmel láthatóan túl egyszerű és naiv hozzáállás volt részemről. Nagyon hamar átláthatatlan és kezelhetetlen eljárás kerekedett a chart rajzoló metódusból.



(8. ábra) Az Svg osztályok osztálydiagramjai UML-ben

Újabb tervezésbe kezdtem, amely az SVG osztályok és közöttük lévő szülő-gyermek viszonyok felvázolását jelentette UML segítségével.

Tudván, hogy szükségem lesz a megjelenésért felelős eszközre is, kutakodni kezdtem milyen lehetőségeim vannak. Kézenfekvő és hatékony megoldás lett volna a JSP (Java Server Pages) használata, amely nagy hátránya a statikusság. Felmerült a JSF keretrendszer használata, amely jó választásnak tűnt, de végül mégis a **GWT (Google Web Toolkit)** mellett döntöttem.

Az agilis szoftverfejlesztés jegyében hozzá is láttam a fejlesztéshez. Megterveztem, milyen lapokra lesz szükségem, aztán elkezdtem a felhasználói felület implementálását, amelyet az SVG projekttel való sikeres integráció követett. A felület időközben teljes ráncfelvarráson ment keresztül a folyamatos refactorálásnak és CSS módosításoknak köszönhetően. A felhasználói felület működőképes volt a kódba teszt-jelleggel (mock) épített adatokkal. Ezután következett a háttér-projekt megvalósítása. Idővel a fejlesztés és a tervezés szinte összemosódott, s végül a kész alkalmazás tesztelése következett. Most lássuk a fejlesztés konkrét, néhol implementációs mélységekig menő ismertetését.

6. Megvalósítás

Az alkalmazás megvalósításához igyekeztem ingyenes, nyílt forráskódú eszközöket és technológiákat felhasználni. Az eszközök tárháza végeláthatatlan, a technológiák kombinálhatóak egymással emiatt elképesztően sokszínű alkalmazásokkal találkozhatunk.

6.1. Az eszközök

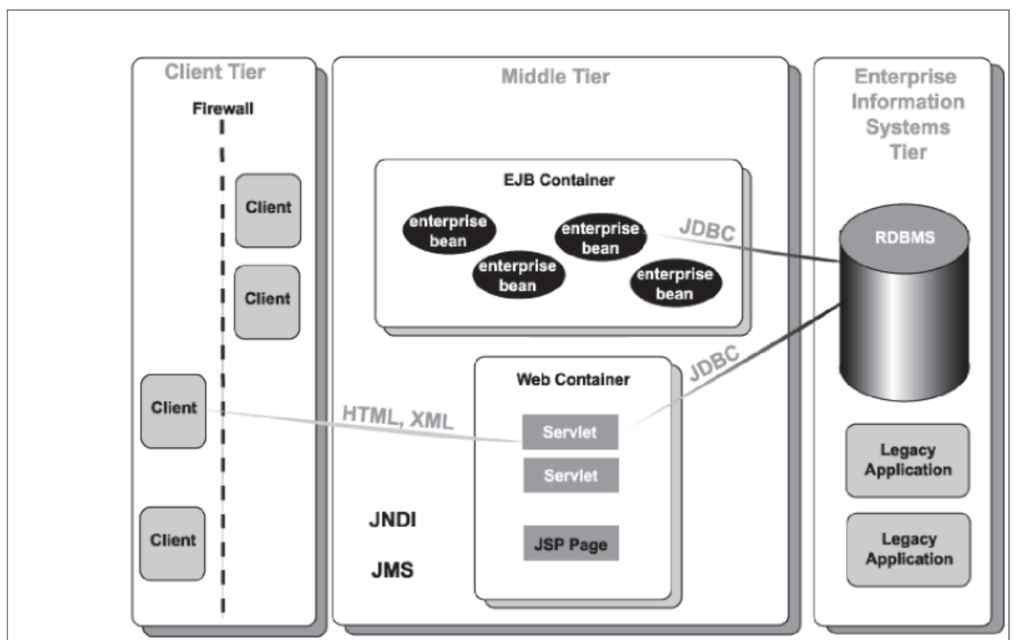
Vizualizációs szoftverről lévén szó fontos momentum az adatbázis-kezelő rendszer kiválasztása. Én a **MySQL** mellett tettem le a voksomat mely ingyenes, kiforrott és bevált adatbázis-kezelő rendszer a ma használt webalkalmazásoknál.
A szoftver **Java** nyelven íródott mely az objektum-orientált paradigma mentén épül fel, ami tökéletes választás ezen alkalmazáshoz. A Java futtatásához webes környezetben számos eszköz áll rendelkezésre. A webszerverek tömkelege közül, a méltán híres **Glassfish 3** (Oracle) alkalmazás szerver tűnt elérhető és ideális megoldásnak. Az adatbázissal való kommunikációra a Glassfish eszközeit használtam fel, mellyel megvalósítható az adatok biztonságos és gyors perzisztációja, erről a későbbiekben ejtek még néhány szót.
Mint említettem, a felhasználói felülethez szintén sok lehetőség közül választhattam, de végül a **GWT** (Google Web Toolkit) mellett döntöttem, melynek előnyeit a későbbiekben kiemelném.
A vizualizált adatok **SVG** (Scalable Vector Graphics) segítségével jelennek meg a böngészőben.
A fejlesztés során Netbeans 6.9 IDE-t használtam, a buildelést és deployozást Ant scriptek végzik, az adatbázis kezeléséhez pedig MySQL Workbench 5.2-t használtam.

6.2. A platform

Mint említettem az alkalmazás Java nyelven íródott és a J2EE (Enterprise Edition) platformra épül, amely jól skálázható, rétegzett, robusztus és biztonságos alapot nyújt az alkalmazás számára.
A Java EE szerver rétegzett architektúrája 3 fő konténerből áll. A konténerek biztosítják a komponensek és az alacsonyabb szintű eszközök közötti kommunikációt.
Íme a 3 réteg:

- Web konténer (Web Container), amely a webszerver és a web-komponensek közötti interfészt valósítja meg. Web-komponens lehet egy JSP vagy JSF lap vagy esetleg egy Servlet. A mi esetünkben a GWT kód helyezkedik el a web-konténerben.

- Alkalmazás kliens konténere (Aplication Client Container). Ez biztosít összeköttetést az alkalmazás kliensei és a Java EE szerver között.
- EJB konténer (EJB Container). Egy Java EE alkalmazás EJB (Enterprise Java Bean)-k használatával valósítja meg az üzleti logikát (Business Logic), melyek ebben a konténerben helyezkednek el és egy-egy szolgáltatást biztosítanak a kliensei számára. Az EJB konténer, tehát biztosítja az alkalmazás és a Java EE szerver közötti kapcsolatot.



(9. ábra) A J2EE architektúra

Ezen platform egy megvalósítása a Glassfish, amelyet a Sun Microsystems fejlesztett, és immár az Oracle tart kézben.

Az alkalmazás két projektből áll. Az hátsó réteg, a Backend, amely biztosítja az adatbázis elérést a felsőbb rétegek felé, illetve az alkalmazás logikáját (Business Logic) adja, ez egy EJB projekt. A másik projekt az erre ráépülő Frontend, amely egy GWT projekt és a felhasználó felé prezentálja az adatokat, illetve feldolgozza a felhasználói interakciókat.

Most bemutatom a két projektet, felvázolom, milyen kapcsolatban állnak egymással, milyen problémákba ütköztem a fejlesztés során és hogyan oldottam meg azokat.

6.3. A backend

6.3.1. Az adatbázis

Az alkalmazás entitásainak tárolása relációs adatbázisba történik. Minden tábla elsődleges kulcsa az id oszlop, amelynek egyedi, pozitív egész értékeit 1-1 MySQL által biztosított szekvencia biztosítja.

A felhasználó tábla:

A *User* entitás tárolására alkalmas tábla, az id mellett tárolja a felhasználó meghívójának azonosítóját. Erre azért van szükség, mert nyilván kell tartanunk a felhasználói hierarchiát (ki kinek a meghívottja) hiszen közvetlen meghívó-meghívott viszonyban lévő egyének megtekinthetik egymás KPI-jeit. Itt tárolódik a felhasználó neve és e-mail címe. Az e-mail cím az egyedi (unique) megszorítással rendelkezik. A következő tárolt adat a felhasználó jelszava, amelynek SHA1 algoritmussal való titkosítását szintén a MySQL-re bíztam. Tárolva van továbbá egy a meghíváskor generált egyedi meghívókód, amely az e-mailben küldött URL-ben szereplő kóddal hasonlítódik majd össze a regisztrációkor. Minden felhasználóhoz tárolódik egy opcionálisan megadható avatar url, amely a felhasználói felületen profilképként jelenik meg, amennyiben az URL egy képre mutat.

A KPI tábla:

Ez a tábla a *KPI* entitások tárolására szolgál. Az elsődleges kulcsa mellett (id), létezik egy külső kulcsa amely a felhasználó táblával köti össze, ezáltal biztosítva az egy-a-többhez (one-to-many) kapcsolatot a két tábla között (egy felhasználóhoz több KPI tartozhat). Tárolja a KPI nevét, elvárt értékét, kezdő és vég-dátumát, illetve a KPI értékeinek mértékegységét.

A KpiEntry tábla:

A *KpiEntry* entitás adatbázisbeli megfelelője, tárolja a KPI-ben foglalt értékeket. Egy ilyen kpi-bejegyzés egy KPI-hez kell, hogy kapcsolódjon, ezt a szintén több-az-egyhez kapcsolatot a kpiId külső kulcs biztosítja (egy KPI-hez tartozhat akárhány bejegyzés, de egy bejegyzés csak egy KPI-hez tartozhat). Itt tárolódik a bejegyzés értéke, a rögzítés dátuma, és egy opcionálisan megadható rekord-szám mező, amely a korábban említett érték nyomatékosítására szolgál.

A táblák relációsémája az alábbi:

User(id, inviterId, name, email, password, inviteCode, avatarUrl)

Kpi(id, userId, name, expectedValue, startDate, endDate, unit)

KpiEntry(id, kpiId, value, date, recordNumber)

6.3.2. A perzisztenciaréteg

A Java platformon különféle perzisztációs eszközök alakultak ki. A perzisztálás az a folyamat, amely során az alkalmazásban előállt adatokat egy olyan helyre mentjük, ahol azok tartósan megmaradnak. Ez a hely tipikusan egy adatbázis. A tradicionális adatbázis-modell mai napig a relációs adatmodellre épülő relációs adatbázis. A relációs adatbázis táblái sorokból és oszlopokból állnak, ahol az oszlopok tulajdonságokat, a sorok pedig egy-egy rekordot reprezentálnak. Ahhoz, hogy ezeket a rekordokat az objektum-orientált világ egy objektumának meg tudjunk feleltetni, egy technológiára van szükségünk, melyet objektum-relációs megfeleltetésnek, vagy ahogy mindenki ismeri ORM (Object Relational Mapping)-nak nevezzük. A Java világban erre a problémára született a JPA (Java Persistence API), amely objektum/relációs megfeleltetéshez ad eszközöket a kezünkbe. Ezek megvalósításai a Hibernate és a JDO. Ezen keretrendszerek (framework) használata gyorsítja az alkalmazásfejlesztést, hiszen a mi feladatunk csak annyi, hogy a rekordokat reprezentáló osztályainkat elkészítsük és vagy annotációkkal vagy egy külső xml fájl (pl Hibernate esetén *.hbm.xml) segítségével megadjuk a map-oláshoz szükséges információkat. Ide tartoznak az adattagok megfeleltetése a rekordok oszlopaihoz, a táblák közötti kapcsolatok megadása, kulcsok definiálása stb. Ezek után a választott keretrendszer saját SQL-szerű (pl: HQL) lekérdezéseivel vezérelhetjük a tranzakciókat az adatbázis felé, amelyeket ő maga fog natív SQL lekérdezésre fordítani. Ezen eszközök használata jó eséllyel adatbázis-függetlenné teszi alkalmazásunkat, hiszen nem írunk natív MySQL, ORACLE vagy PostrgeSQL lekérdezéseket, hanem egy köztes nyelven fogalmazzuk meg azokat. Így egy kevés konfigurációs fájlban való módosítás után lecserélhetjük egy akár más gyártótól származó adatbázis-kezelő rendszerre a jelenlegit, a forráskód módosítása nélkül.

Mint általában minden keretrendszernél, itt is gondunk támadhat a nem elegendő testre-szabhatóságból eredő hibákkal. Mivel nem mi írtuk a keretrendszert, és gyakran nem is nyílt forráskódú, hogy megnézhessük, valószínűleg fogalmunk sincs miként kaptuk az adatokat. Persze a legtöbbször nem is érdekel minket, kivéve egy esetben, ha teljesítmény problémával állunk szemben. Egy sok felhasználós, sok táblából és sok adatból dolgozó szoftver, mint például egy ilyen üzleti vizualizációs szoftver (több gigabájt esetenként néhány terrabájt adat), lényeges a teljesítmény, hiszen az alkalmazás teljes mértékben az adatbázisra támaszkodik miközben működik. Továbbá könnyen a verzió-függőség

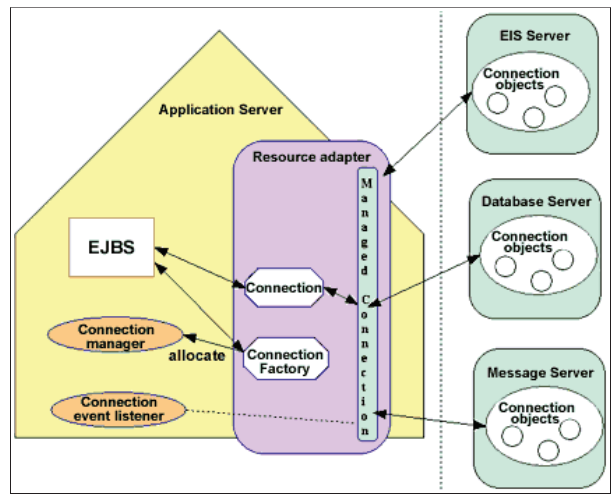
áldozatává válhat az alkalmazás, ami annyit tesz, hogy az a keretrendszer verzió amellyel a projektet elkezdtük, már elavultnak számít a fejlesztés végéhez közeledve, és nincs fölösleges erőforrás a működő alkalmazás által használt keretrendszer kicserélésére. Nyilván ez nagyobb projekteknél okozhat problémát.

A fenti szempontokat szem előtt tartva kell döntenünk a gyorsaság és a kézben-tarthatóság/testre-szabhatóság között. Hosszas mérlegelés után a manuális ORM mellett döntöttem, amely a Glassfish alkalmazás-szerver lehetőségeivel kombinálva egy egyedi és remélhetőleg gyors perzisztenciaréteg elkészítését eredményezte.

6.3.3. Connection poolozás manuális ORM-el

Röviden így nevezhetjük azt a megoldást, amit megvalósítottam az adatbázis kapcsolatok kezeléséhez, illetve az objektum/relációs megfeleltetéshez. Menjünk végig a megoldáson.

A legalsóbb szinten a JDBC kapcsolatok kezelését a Glassfish alkalmazás-szerverre bízzuk, ez a megoldás kihasználja a Glassfish által biztosított adatbázis-kapcsolat gyorsítótárazást, avagy a connection poolozást, továbbá lehetővé teszi az adatbázis-kapcsolatok dinamikus és újrafelhasználható kezelését. A kapcsolatok számát egy minimum és egy maximum korlát közé szorítjuk, ahol a kapcsolatok száma biztosan a két határ között lesz minden pillanatban.



(10. ábra) Connection pooling a J2EE architektúrán

Egy ilyen gyorsítótárat a Glassfish adminisztrátori-felületén egyszerűen létrehozhatunk néhány beállítás megadásával:

1. Belépünk az admin felületre (http://localhost:4848/common/index.jsf)
2. Resourecs/JDBC/JDBC Connection Pools menüpont
3. A „New" gombra kattintás után az alábbiakat kell beállítanunk:
    - Pool name: a gyorsítótár neve amivel később hivatkozzuk (visualizationPool)
    - Resource type: javax.sql.ConnectionPoolDataSource
    - Database Driver Vendor: az adatbázis típusa (MySQL)
    - Additional Properties fülön szükséges megadni az adatbázis kapcsolódáshoz a felhasználónevet (user), jelszót (password) és az adatbázis nevét (databaseName)

Konfigurálhatjuk emellett a minimum pool méretet (Initial and Minimum Pool Size), a maximum méretet (Maximum Pool Size) továbbá az időtúllépés idejét és a kapcsolat kérése utáni maximális várakozási időt. Egy érdekes beállítás van még, amiről nem beszéltem: amikor elfogy az összes élő kapcsolat, de a maximum értéket még nem érte el a számuk, a Glassfish újabb kapcsolatokat fog rendelkezésünkre bocsátani. Azt, hogy hány új kapcsolatot hozzon létre egyszerre, a Pool Resize Quantity paraméter módosításával állíthatjuk be. Ugyanennyivel fogja csökkenteni a kapcsolatok számát, amennyiben nincs szükség annyi kapcsolatra. Ez a megoldás egy jól skálázható adatbázis-kapcsolati réteget biztosít az alkalmazás számára.

Ahhoz, hogy az alkalmazásunkból használni tudjuk ezt az intelligens adatbázis-kezelési mechanizmust, egy erőforrás-kezelőt, JDBC Resourcet (DataSource) is létre kell hoznunk, amit szintén a Glassfish admin felületén tehetünk meg. Ez a resource lesz a kapcsolat a connection pool és az EJB-k között.

Amit meg kell adnink létrehozáskor, a JNDI Name, az erőforrás neve (visualizationDS) és egy Connection Pool-t, amely az adatbázis elérését biztosítja (visualizationPool).

6.3.4. Az ORM

A manuális ORM annyit tesz, hogy az objektum-relációs leképzéseket kézzel írjuk meg az egyes entitásokhoz. Ez nyilvánvalóan több munkát igényel mint egy Hibernate mapping beállítása, de sokkal kezelhetőbb lesz a perzisztenciaréteg, és biztosan azt fogja csinálni, amit mi akarunk. Minden entitás osztály-szinten megvalósít egy „public Entity retrieve(ResultSet set)" metódust amely egy SQL lekérdezés eredményhalmazát várja paraméterül és felépít belőle egy saját példányt. A mapping viszonylag egyszerű, hiszen a MySQL típusai könnyedén megfeleltethetőek a Java típusainak.

6.3.5. Az EJB-k

Az EJB-k avagy az Enterprise JavaBean-ek valósítják meg az üzleti logikát az alkalmazásban. Mint már említettem, az EJB-k is egy konténerben helyezkednek el, ami felel az EJB-k példányosításáért, törléséért és egymással való kommunikációjukért.
Az EJB-knek két alapvető típusa van:

- Session Bean: 3 típusa van, bármelyik típushoz definiálhatunk Locale interfészt ha ugyanabból a JVM-ből szeretnénk használni, illetve Remote interfészt ha egy távoli JVM-ből.
    1. Állapotmegőrző vagy Stateful Beanek, amelyek állapotát megőrzik két hívás között, ha a kliens bizonyos ideig nem fordul az objektumhoz, automatikusan perzisztálódik, ezzel is memóriát takarítva meg. Használhatjuk egy munkamenet alatti adatok tárolására például kosár funkció egy webshopnál.
    2. Állapot mentes vagy Stateless Beanek, akkor használjuk, ha nem kell, hogy a hívások között az állapotok megőrződjenek, csak egy szolgáltatást szeretnénk elérni, például egy adatbázis-kérés.
    3. Egyke azaz Singleton Beanről beszélünk ha a program globális egészére egy példányt szeretnénk biztosítani
- Message Driven Bean: JMS (Java Message Service) üzeneteket feldolgozó bean, hasonló elven működik az eseménykezelőkhöz annyi különbséggel, hogy események helyett üzeneteket fogadnak. Alapvető különbség a Session Bean és az Message Driven Bean között, hogy az utóbbiakat a kliens nem tudja interfészeken keresztül elérni.

A JavaBean-ek tehát szolgáltatásokat nyújtanak a Frontend projekt felé. Az általam használt bean-ek mind a Local interfészüket valósítják meg, mivel a kliens projekttel azonos JVM-en futnak. Néhány szó az alkalmazás enterprise bean-jeiről:

- AuthenticationBean (Stateful): A felhasználó be/ki-jelentkezése illetve a regisztrációért felelős, továbbá megkérdezhetjük tőle hogy van-e bejelentkezett felhasználó.
- UserManagerBean (Stateless): A felhasználóval kapcsolatos műveleteket valósítja meg, regisztráció, meghívás, illetve adatmódosítás.
- KpiServiceBean (Stateless): A KPI-k kezelését teszi lehetővé. Lekérdezhetjük egy adott felhasználó által megtekinthető KPI-ket, elérhetjük ID alapján és újat hozhatunk létre az adatbázisba.

AuthenticationBean

Ez az állapotmegőrző session bean tárol egy *User* példányt, amely a bejelentkezett felhasználóhoz tartozik. Lekérdezni a bejelentkezett felhasználót a *getLoggedInUser()* metódussal lehet, amely a tárolt *User* példánnyal tér vissza. Amennyiben ez NULL, nincs bejelentkezett felhasználó. A bejelentkezés is itt zajlik, a *login(String email, String password)* metódus hívásakor a bean a megfelelő SQL lekérdezést futtatva megpróbál a megadott e-mail címmel és jelszóval (SHA1 titkosítás után) egyezést találni. Amennyiben ez sikeres, felépít egy *User* példányt, eltárolja azt, és vissza is tér vele a hívó fél részére. Ha a lekérdezés sikertelen, tehát nincs ilyen adatokkal rendelkező felhasználó az adatbázisban, NULL-al tér vissza, amit a kliens a megfelelő hibaüzenet kiírásával jelez a felhasználó felé. A *logout()* metódus a bejelentkezett felhasználót eltávolítja a sessionből, amely kliens oldalon a felhasználó bejelentkező képernyőre való átirányítását vonja maga után.

UserManagerBean

A felhasználó kezeléséért felelős osztály. Míg az *AuthenticationBean* a felhasználó be és kijelentkezéséért felel, ez az osztály a felhasználók regisztrációját és adatainak módosítását biztosítja. A bean biztosít egy *update(User user)* metódust, amely az átadott felhasználó megváltozott adatait frissíti az adatbázisban. A visszatérési értéke igaz vagy hamis attól függően, hogy a perzisztálás sikeres volt-e. Amikor egy bejelentkezett felhasználó a meghívás funkciót használja, ezen bean *invitation(long inviterId, String email)* metódusa hívódik meg. A metódus generál egy egyedi, véletlenszerű karaktersorozatot, amelyet a meghívó id-jével és a megadott e-mail címmel együtt az adatbázisban eltárol, mint újonnan meghívott felhasználót. Egy e-mail küldő osztály segítségével pedig egy előre

definiált szöveget küld az e-mail címre, amelyben szerepel a generált aktivációs kód egy URL formájában. Az e-mail küldésről később még ejtek néhány szót.
Amikor a meghívót kapó egyén az aktivációs linkre kattint, akkor a *userActivation(String activationCode)* metódus hívódik meg, amely az egyedi aktivációs kód alapján nyeri ki az előzetesen elmentett, de még nem aktivált felhasználó adatait, és az ezekből az adatokból felépített *User* objektummal tér vissza. Amennyiben nincs ilyen aktivációs kóddal meghívott felhasználó eltárolva, NULL-al tér vissza és a hívó környezet jelzi a felhasználó felé, hogy hibás vagy már aktivált kóddal próbálkozik regisztrálni.

KpiServiceBean

Ez az állapotmentes session bean felel a KPI-kkel végezhető műveletekért. A legalapvetőbb lekérés a *getKpiById(long id)* amely ID alapján ad vissza egy KPI példányt. A másik gyakran használt lekérdező metódus a *getKpisForUser(long userId)* amely a megadott felhasználó által megtekinthető összes KPI-t adja vissza. Nem csak azokat amelyeket ő hozott létre, hanem azokat is amelyeket az őt meghívó felhasználó. Amennyiben nincs a feltételeknek megfelelő KPI, üres listával tér vissza a metódus.
Ez a bean nem csak a lekérdezésekért, hanem a KPI-k létrehozásáért is felel. A kliens oldalról érkező XML fájlt egy segédosztály validálja, parszolja, majd az adatbázisba menti.

6.3.6. Segédosztályok

Két segédosztályt hoztam létre, amelyek csak egy-egy osztályszintű metódust biztosítanak az őket használó osztályok számára. Az egyik az email küldést, a másik az XML-ben feltöltött KPI adatok feldolgozását valósítja meg.

EmailSender

Az email küldéshez a *javax.mail* csomag osztályait használtam, amelyek az SMTP-n keresztüli e-mailküldést tették lehetővé. A tesztelés során a Google e-mail szolgáltatását (smtp.gmail.com) vettem igénybe. Az e-mail szövege külön HTML fájlban tárolódik, amelyben az aktivációs kód helyett, egy helykijelölő szerepel, melyet a beolvasás után helyettesítek be az aktuális meghívott kódjával. Egy meghívó URL a következőképpen néz ki:

http://localhost:8080/VisualizationDashboard-war/#REGISTRATION;7gNdYyLuT9TCBEhq3qR7

Az alkalmazás elérési útja után a #REGISTRATION lapra való hivatkozás történik. Ez után pontosvesszővel elválasztva szerepel a regisztrációs kód.

XmlParser

Ezen osztály *parse(File xml)* statikus (osztály szintű) metódusa felel a felhasználói felületről érkező importált XML fájl feldolgozásáért. Első lépésben egy XML validációt hajt végre az általam előre definiált XML-Scema alapján. A program számára valid XML formátum egy <kpi> nevű gyökérelemből és legalább egy (legfeljebb akárhány) <entry> gyermekelemből kell, hogy álljon. A kpi elem kötelezően kell, hogy tartalmazza a *title, startDate, endDate, expectedValue* és *unit* attribútumokat. Az entry elemek kötelező attribútuma a *date* és opcionálisan megadható egy *recordNumber* attribútum is. Az entry tag-ek között megadott értékek lesznek a KPI értékei. Ha a dokumentum a validáláson nem bukott meg, a parszolás vagyis a feldolgozás következik. Az XML feldolgozáshoz a *java.xml* csomag által biztosított feldolgozót használtam. A feldolgozó metódus egy KPI példánnyal tér vissza amennyiben sikeres volt. Ellenkező esetben kivételt dobva leáll, és a hívó környezet tájékoztatja a kliens oldalt az importálás sikertelenségéről.

Íme egy példa az importálható XML fájlra:

```
<?xml version="1.0" encoding="UTF-8"?>
<kpi title="BMW Incidents" startDate="2011-01-01" endDate="2011-01-20" unit="hour"
expectedValue="230">
        <entry recordNumber="12" date="2011-01-01">100</entry>
        <entry recordNumber="42" date="2011-01-10">200</entry>
        <entry date="2011-01-20">300</entry>
</kpi>
```

6.4. A Frontend

6.4.1. A Google Web Toolkit (GWT)

A GWT egy, a Google fejlesztői eszköztárai közül. Segítségével AJAX alapú webalkalmazásokat fejleszthetünk, gyorsan és hatékonyan. A program teljes kódja Java nyelven íródik, így egy sokkal átláthatóbb kódot kapunk, mint amikor a szerver és a kliens oldali nyelvek különbözőek (például php-JavaScript). A GWT kereszt-fordítója (cross-compiler) minden böngészőre külön optimalizált JavaScript-re fordítja a kódot, így a kliens böngészőjébe mindig a megfelelően optimalizált alkalmazás töltődik be. A kliens böngészőjében futó JavaScript intéz asszinkron kéréseket a szerver oldal felé, ezzel egy dinamikus és látványos felhasználói felületet biztosít a webalkalmazásunk számára.

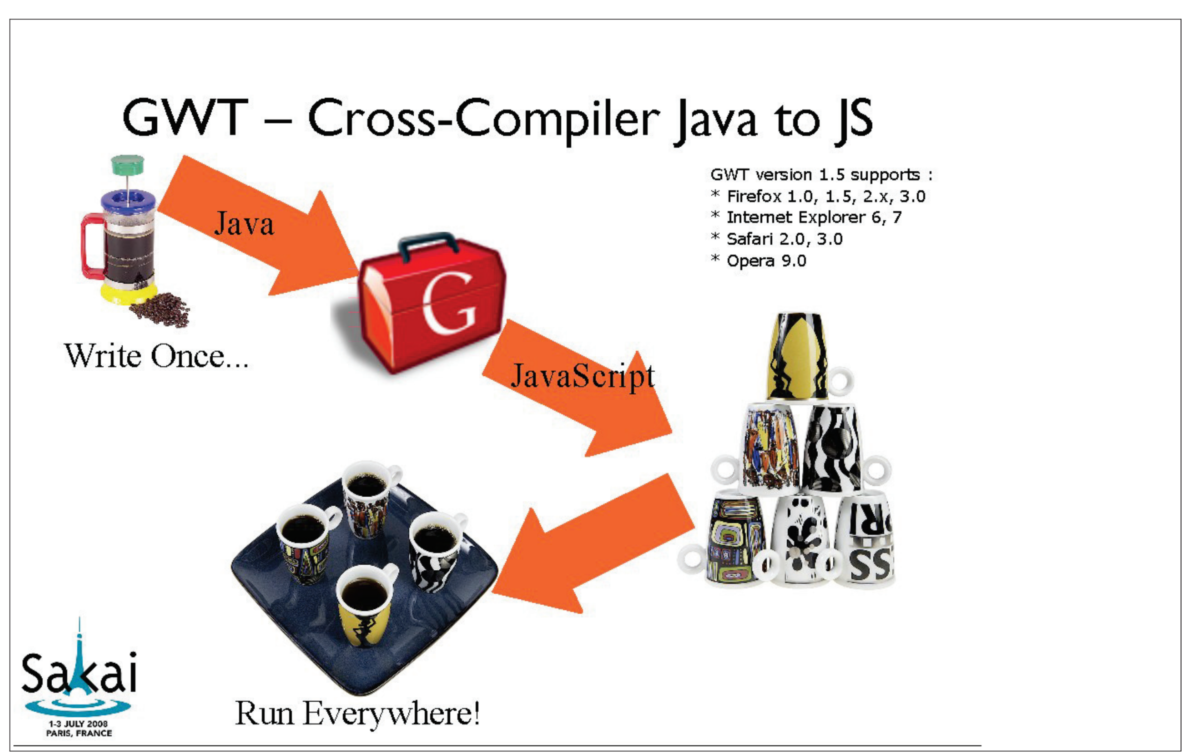
6.4.2. A GWT szerver és kliens oldala

A GWT-s alkalmazás szerver oldali kódja a web-szerver JVM-én fut, a mi esetünkben a Glassfish web-szerverén. A kliens kód JavaScriptre fordul, és a böngészőben fut. Minden GWT projekthez tartozik egy *project_neve.gwt.xml* fájl, amelyben definiálnunk kell az alkalmazás belépési pontját, amelynek egy olyan osztálynak kell lennie, amely implementálja az *EntryPoint* interfészt. Itt kell megadnunk továbbá azt is, hogy mely java csomagok (packages) forduljanak JavaScriptre. Egy tipikus GWT projekt 3 csomagra oszlik fel: client, server és shared.

A client a kliens kód osztályait tartalmazza, főként a felhasználói felület (UI) komponenseit, illetve az aszinkron kommunikációhoz szükséges interfészeket.

A server csomagban az alkalmazás szerver oldali kódja helyezkedik el, a szolgáltatások (services) és az esetleges EJB hívások. Az általam létrehozott service-k az EJB-k szolgáltatásait publikálják a GWT kliens oldala felé, tehát hasonlóképp az EJB-khez, léteznek *AuthenticationService, KpiService* és *UserManagerService* implementációk, amelyekhez tartoznak aszinkron (async) interfészek is.

A shared csomag tipikusan olyan osztályok helye, amelyeket használni kívánunk mind szerver, mind kliens oldalon. Ilyenek az esetleges helper osztályok, illetve a DTO-k. A shared és client csomagok tartalma fordul JavaScriptre.



(11. ábra) A GWT keresztfordítójának szemléletes ábrázolása

6.4.3. Szerver-kliens kommunikáció, avagy AJAX a GWT-vel

Mint tudjuk, a JavaScript nyelvre fordított kliens kód az alkalmazás felhasználói (a kliensek) böngészőjében fut. Ennek a kliens kódnak azonban el kell érnie a szervert ahhoz, hogy naprakész adatokat tudjon megjeleníteni. Ehhez egy szimpla alkalmazásnak HTTP kéréseket kell elküldenie a szervernek minden gombnyomáskor vagy adatok frissítésekor. Ez a webalkalmazás teljes egészének újratöltődését eredményezi. Ezt nevezzük szinkron kommunikációnak, mivel az alkalmazás vár a válaszra, amíg az meg nem érkezik és közben nem fogad semmilyen interakciót. Ez a felhasználói élmény szempontjából igen kellemetlen, főleg hosszan tartó szerver oldali műveletek elvégzése esetén. Ennek a problémának a kezelésére született egy technológia, amely AJAX néven vált hírhedté.

AJAX (Asynchronous JavaScript and XML)

Az AJAX egy technológia, amely gyors és dinamikus weboldalak készítéséhez nyújt hatékony eszközt. Lehetővé teszi a webalkalmazás számára, hogy anélkül, hogy a teljes oldal tartalmát újra letöltené, annak egy darabját frissítse asszinkron módon. Ezt a JavaScript által biztosított közvetlen DOM manipuláció és az *XMLHttpRequest* nevű objektum együttes használata teszi lehetővé. A kérés a háttérben asszinkron módon megy végbe, és amikor készen van, egy általunk megadott függvényt hív meg, amelynek paramétere a szerver által adott válasz (ezt a metódust nevezzük callback method-nak).

Lássuk a GWT hogyan valósítja meg az asszinkron kommunikációt.

GWT AsyncCallback és a Service-k

Egy GWT-s alkalmazás szerver által biztosított szolgáltatásai használatához definiálnunk kell egy kliens oldali interfészt, amelyet a szerver oldalon implementálunk. Ezen kliens oldali interfész mellé létre kell hozni egy asszinkron interfészt is. A két interfész annyiban különbözik, hogy az utóbbiban minden metódusnak szüksége van egy plusz paraméterre amely egy AsyncCallback<E> generikus típusú paraméter, ahol E az eredeti interfészben definiált metódus visszatérési értéke. Ezt egy egyszerű kódrészleten keresztül szemléltetném ami az alkalmazásból származik:

```
@RemoteServiceRelativePath("kpi_service")
public interface KpiService extends RemoteService {
        public KpiDTO getKpiById(long id);
}
```

A fenti kódrészlet a kliens oldali interfész a szolgáltatáshoz. A *RemoteServiceRelativePath* annotációval megadott név lesz a szolgáltatás URL-ben használt név a távoli híváskor. Az ehhez tartozó asszinkron interfész így néz ki:

```
public interface KpiServiceAsync {
        void getKpiById(long id, AsyncCallback<KpiDTO> callback);
}
```

A GWT dependency injection segítségével bocsát rendelkezésünkre egy példányt a service-ből, amelyet kliens oldalon használni szeretnénk:

```
KpiServiceAsync kpiService = GWT.create(KpiService.class);
```

Azt, hogy milyen szerver oldali implementációt kapjunk ezen hívás helyén, a web.xml projekt-leíró XML fájlban kell megadni. A fenti service-t az alábbi módon tudjuk asszinkron módon igénybe venni:

```
kpiService.getKpiById(kpiId, new AsyncCallback<KpiDTO>() {
                @Override
                public void onSuccess(KpiDTO result) {
                        //feldolgozzuk a szerver válaszát
                }
                @Override
                public void onFailure(Throwable caught) {
```

```
                //kezeljük a kivételt a sikertelen kérés esetén
            }
    });
```

Ennyi az asszinkron kommunikáció GWT-vel. Vegyük azonban észre, hogy nem a szerver oldalon definiált *KPI* osztály példánya küldődik a kliens oldalra hanem valami más.

A Data Transfer Object (DTO)

Egy ilyen architektúrájú alkalmazásnál meg kell oldanunk a szerver-kliens közötti kommunikációt. A szerver és kliens oldal közötti adatküldésre jól bevált módszer a DTO-k használata. A DTO-k a szerver oldali entitások „másolatai", amelyek a kommunikáció során az adatokat szállítják. A DTO-k a függőségi viszonyok kezelésében is szerepet játszik, hiszen a kliens kód nem ismeri a szerver oldali EJB projektben definiált entitásokat. A DTO-k gyakran nem tartalmazzák az entitások összes attribútumát, tipikusan csak annyit, amennyit feltétlenül szükséges a kliens tudtára hozni. Így amellett, hogy a kliens és a szerver a minimálisan szükséges adatmodellen operál, a hálózaton küldött adatok mennyisége is optimalizálható. Minden entitáshoz definiálnunk kell egy DTO-t, amit a szerver-kliens adatátvitel során használni fogunk. Az entitásokat és a DTO-kat meg kell feleltetni egymásnak, amire ugyan vannak eszközök (pl: dozer), de nem láttam szükségét a használatának, hiszen azt a néhány entitást, amit az alkalmazásom használ egy egyszerű metódussal átalakítom DTO-vá. A *Mapper* osztály implementálása után egyszerűen az alábbi módon megfeleltethetőek egymásnak az entitások és a DTO-k:

```
    public class KpiServiceImpl extends RemoteServiceServlet implements KpiService {
        @EJB
        KpiServiceBeanLocal kpiBean;

        public KpiDTO getKpiById(long id) {
            return (KpiDTO)Mapper.map(kpiBean.getKpiById(id));
        }
    }
```

A fenti példa az előbb bemutatott *KpiService* interfész egy szerver oldali megvalósítása. Mint látszik ez GWT Service az EJB projektben futó *KpiServiceBean* metódusát hívja az adatok beszerzéséhez. Mivel az EJB projekt az entitásokkal dolgozik, így az egy *KPI* példánnyal tér majd vissza, de a kliens felé egy *KpiDTO* példányt szeretnénk küldeni. Ezen a ponton történik az entititás-DTO megfeleltetés. A

Mapper osztályban kihasználva a Java reflexív (reflection) mivoltát és azt, hogy minden entitás az *Entity* absztrakt osztályból, és minden szállító objektum a *DTO* ős-osztályból származik, könnyedén implementáltam a *public DTO map(Entity entity)* metódust, amely bármilyen általam létrehozott entitást képes a kliens számára küldhető objektummá alakítani.

6.4.4. A Model-View-Presenter minta

A GWT a kliens oldali tervezést egy magas szintű tervezési minta segítségével valósítja meg, ez az MVP (Model-View-Presenter). Ebben a mintában, amint ez a nevéből is kiderül, 3 részre bontjuk a funkcionalitást.

A **Model** az adatokat reprezentáló objektumok (jelen esetben DTO-k), melyeket a felhasználói felületen valamilyen formában meg szeretnénk jeleníteni, illetve a felhasználói interakciók során állapotukat megváltoztatni.

A **View** (avagy nézet) a felhasználói felület tényleges megjelenéséért felel, itt helyezhetjük el a különböző beviteli mezőket, gombokat és egyéb UI komponenseket, melyekből kedvünkre válogathatunk az API-ban, illetve sajátokat is készíthetünk.

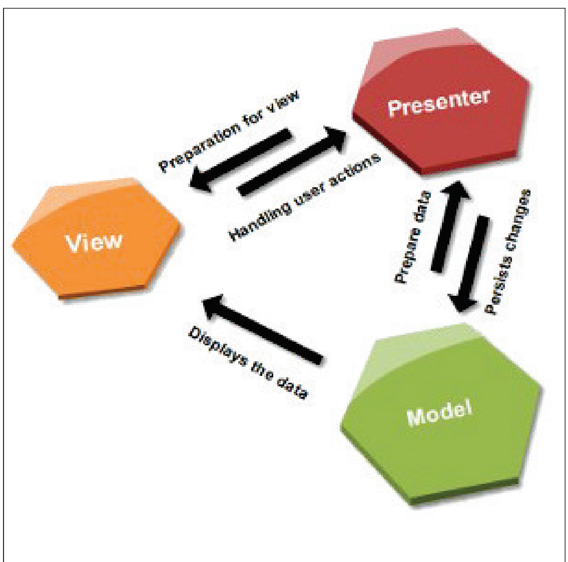
A **Presenter** végzi a különböző interakciók feldolgozását, kezeli a felhasználói eseményeket és elvégzi az adatok perzisztálását, amelyeket XML-RPC szerver-hívások segítségével old meg.

Ezen három réteg összehangolásával és megfelelő használatával jól fejleszthető és áttekinthető kódot kapunk.

Van még egy eszköz, amely a GWT 2.0 egyik nagy újítása volt, az úgynevezett UiBinder, amely lehetővé teszi a felhasználói felület deklaratív megfogalmazását. Ez annyit tesz, hogy az alapvető elrendezéseket és UI komponenseket egy XML fájlban állítjuk össze, ezzel is egyszerűsítve a View osztályokat. A viewOszályNeve.ui.xml fájlokban használhatjuk az összes beépített GWT UI komponenst, és import után a sajátjainkat is. Ezek után az egyes View osztályokba csak a dinamikus tartalmakkal kell feltöltenünk nézetünket, illetve az esetleges eseménykezelőket kell hozzáadni a komponensekhez (például gombokhoz).

Egy ilyen MVP minta implementálásához szükségem volt két generikus interfészre (View és Presenter), és két őket megvalósító generikus, absztrakt osztályra (ViewImpl és PresenterImpl). Minden további lap az oldalon ezek egy-egy megvalósításai. Erre azért volt szükség, hogy a Presenter-View kapcsolatot egyszer kelljen megfogalmazni, és ne minden új lap létrehozásakor. A generikus típus kikényszeríti, hogy minden View implementációhoz tartozzon egy Presenter implementáció, továbbá

egymást el tudják érni, hiszen nyilván tartanak egymásról egy-egy referenciát.



(12. ábra) Az MVP tervezési minta

Példaként bemutatom az egyik legegyszerűbb oldalt, a bejelentkezés oldalát:

- *LoginPresenterImpl* osztály: A *PresenterImpl*-ből származik, és implementálja a *LoginPresenter* interfészt. Ebben az interfészben van definiálva egy *login(String email, String password)* metódus amely az *AuthenticationService login* metódusát fogja hívni aszinkron módon. Ha sikeres a bejelentkezés, megkéri az *ApplicationController*-t hogy lépjen át a Dashboard-ra. Amennyiben nem sikerült, hibaüzenetet dob a felhasználói felületre.
- *LoginViewImpl.gwt.xml* fájl: A bejelentkezéshez szükséges két beviteli-mezőt (e-mail cím és jelszó) illetve egy belépés gombot tartalmaz. Itt kerülnek beállításra a megfelelő stílusok (amelyek a css fájlban vannak definiálva).
- *LoginViewImpl* osztály: A *ViewImpl*-ből származik és implementálja a *LoginView* interfészt. Itt történik a fenti xml-fájlal való kötés (binding), és a bejelentkezés gombra beállítódik egy eseménykezelő, amely a prezentere (*LoginPresenter*) *login* metódusát fogja hívni, a két beviteli mezőbe beírt paraméterrel. Továbbá definiáltam még egy eseménykezelőt, amely azt hivatott figyelni, hogy nyomtunk-e entert a jelszó beviteli mezőn. Ha igen, ugyanazt a hatást váltja ki, mintha a belépés gombra kattintottunk volna, ez mindössze a felhasználó könnyebb bejelentkezését segíti, hiszen nem kell az egérhez nyúlnia és a gombra kattintania.

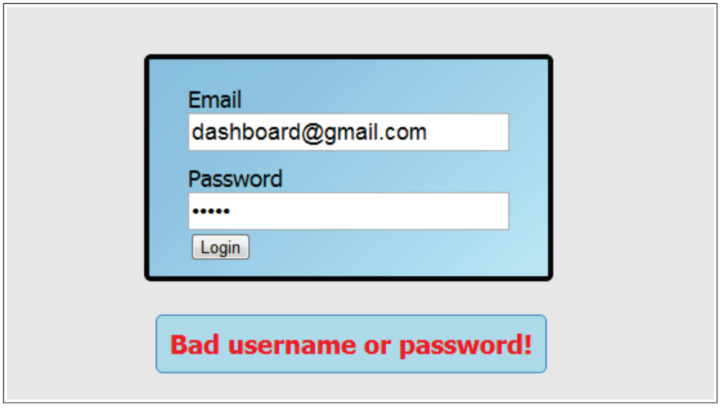
Ehhez hasonlóan épül fel az összes oldal az alkalmazásban.

Említettem egy osztályt, amiről eddig még nem ejtettem szót, az *ApplicationController-t*. Ez az osztály felel az alkalmazáson belüli navigációért. A *navigateTo(ApplicationPlaces placeEnum)* metódus

meghívásával lépünk az egyik lapról a másikra, de ezt a későbbiekben kifejtem. Mint látható egy enumot (felsorolásos típus) vár paraméterül amellyel kijelöljük hova szeretnénk átirányítani a felhasználót (pl: *ApplicationEnum.DASHBOARD*). A Controller minden lapváltáskor authorizálja a felhasználót, azaz megvizsgálja, valóban be van-e még jelentkezve, és van-e joga a kért oldal megtekintéséhez. Ha az authorizáció sikertelen, a kért oldal helyett a bejelentkezés oldala fog megjelenni. Az *ApplicationController* birtokol egy *ApplicationPresenterImpl* példányt, amely az egész alkalmazás fő-prezentere. Ez az osztály fogja példányosítani az összes többi megjelenítendő oldalt, és a Controller által vezérelt módon megjeleníteni azokat. Az ehhez az osztályhoz tartozó View-on helyezkedik el a dashboard fejléce, amely tartalmazza az alkalmazás logo-ját, a menüsort, és a bejelentkezett felhasználó nevét. Ez a fejléc a bejelentkezési képernyő kivételével minden esetben látszik.

6.4.5. Kommunikáció a felhasználóval

A felhasználóval való kommunikáció akkor teljes értékű, ha az alkalmazás is képes közölni a felhasználó irányába bizonyos események sikerességét vagy épp sikertelenségét. Ezt a felhasználói felületre kijelzett szöveges üzenetek megjenítésével szokás megvalósítani. Készítettem egy saját osztályt, amely a GWT *PopupPanel* osztályából származik, és a neve *AlertBox*. Egyetlen publikus, osztály-szintű metódusa van, amelyet a program bármelyik prezentere képes meghívni egy *String* paraméterrel. Ez az üzenet a képernyő alsó részén jelenik meg 3 másodpercig. Ez a kis felugró címke a felhasználóval való rövid szövegek közlésére született, mint a „Sikeres bejelentkezés", „KPI feltöltve", „Hibás e-mail cím vagy jelszó" üzenetek.



(13. ábra) Üzenet a felhasználónak

6.4.6. Böngésző előzmények és navigáció a lapok között

Egy AJAX alapú alkalmazásnál mindig probléma a böngészési előzmények, azaz a history kezelése. Mivel a böngésző előzményként csak az új lap letöltéseket kezeli automatikusan, így az asszinkron lapváltásokat a fejlesztők dolga kezelni. A GWT használata esetén persze kapunk eszközt ezen probléma megoldására is. Miután az alábbi iframe-t elhelyezzük a projekt kezdőlapján, a GWT a kódhoz csatolja a megfelelő előzmény-kezelő könyvtárat:

```
<iframe src="javascript:''" id="__gwt_historyFrame" tabIndex='-1'
        style="position: absolute; width: 0; height: 0; border: 0"></iframe>
```

Ez után az alkalmazásban lehetőség nyílik új histroy bejegyzéseket létrehozni, amelyek az URL-ben is megjelennek az oldal frissülése nélkül.

```
public static void navigateTo(ApplicationPlaces viewEnum) {
        History.newItem(viewEnum.name());
}
```

A fenti metódus meghívásával navigálhatunk az alkalmazás oldalai között. A *History* osztályra definiált eseménykezelő minden új newItem() híváskor frissíti az URL-t, aztán betölti a megfelelő oldalt.
Az alábbi módon épül fel minden oldal URL-je:

```
http://localhost:8080/VisualizationDashboard-war/#LOGIN
http://localhost:8080/VisualizationDashboard-war/#DASHBOARD
```

Ezután a böngésző vissza és előre gombja is használható. Ha bejelentkezés nélkül próbáljuk elérni valamelyik oldalt, a program a bejelentkező képernyőre irányít át minket.

6.4.7. SVG (Scalable Vector Graphics)

Saját vizualizációs eszköztárat készítettem amely vektorgrafikus módon állít elő különböző típusú diagramokat. Miért volt erre szükség? Hiszen rengeteg ingyenes szolgáltatás áll rendelkezésünkre (például a Google Chartok a Visualization API-ban), amelyek jelentősen felgyorsítják a fejlesztést. A válasz egyszerű: a biztonság. Egy vállalat belső alkalmazásáról lévén szó, nagy jelentősége van az adatok biztonságának, így ezeket nem szívesen adják ki külső szolgáltatások számára.
Sokféleképpen kísérelhetjük meg diagramok kirajzolását a böngészőben. Például előállíthatunk egy képet a programunkkal, amit aztán a böngészőben egyszerűen megjelenítünk. Ezzel a nyilvánvaló probléma a felbontás, illetve a nagyítás során bekövetkező óriási minőségromlás, hiszen a képet raszteresen állítottuk elő. Próbálkozhatunk HTML5-ös Canvas-ra való rajzolással, ami sajnos jelen

állás szerint nem sok böngészőben jelenne meg megfelelően. Kézenfekvő megoldás tehát a vektorgrafikus előállítás amire kiváló az SVG.
Az SVG egy XML formátumú leíró nyelv melyet HTML kódba ágyazva jeleníthetünk meg a böngészőben. Nem veszít a minőségéből a nagyítás során, hiszen skálázható, továbbá JavaScript-el minden eleme és attribútuma manipulálható, illetve animálható.

Az első és legfontosabb kérdés az SVG-vel kapcsolatban, talán az volt, hogy miként integrálható a GWT felhasználói felületébe. Külső nyílt forráskódú könyvtár keresgetése helyett készítettem egy egyszerű API-t az SVG fájl kezeléséhez. Az API felépítése végtelenül egyszerű.
Minden SVG elem (<svg>, <circle>, <rect>) egy absztrakt osztályból, az *SvgElement*-ből származik. Ezen osztály minden példánya magába burkol egy a GWT által rendelkezésünkre bocsátott *Element* példányt, ami egy DOM (Document Object Model) elemet reprezentál. Az *SvgElement*-ek fa szerűen építhetők egymásba amelyek azonos struktúrában képződnek le a DOM fába a futáskor. Az *SvgElement* osztály funkcionalitásához tartozik az elem attribútumainak manipulálása. Egy egyszerű osztály, az *SvgAttribute* reprezentálja magát az attribútumot és az értékét. Adhatunk style neveket az attribútumok sorához, melyeket a CSS fájlban kell definiálunk, illetve egy *EventListener*-t amely a böngészőben lezajló, az adott elemre vonatkozó eseményekre lesz feliratkozva.
Minden további osztály az *SvgElement*-ből származik és valósít meg valamilyen SVG elemet. Kiemelt szerepe van az *SvgRootElement*-nek amely az SVG fájl gyökérelemét reprezentálja, annak méreteivel. Van továbbá *SvgCircle, SvgRectangle, SvgText,* stb… melyek a további SVG elemeket valósítják meg. Ezen osztályok segítségével létrehozható egy SVG fájl amit a különböző chart-ok a nekik megfelelő módon építenek fel és jelenítenek meg.

6.4.8. A chartok felépítése és algoritmusa

A chartok típusát már ismertettem, most a megvalósításukat fejteném ki bővebben. Minden chart az *AbstractChart* osztályból származik. Ez egy absztrakt osztály, amely konstruktorában 3 paramétert vár: szélességet, magasságot, és egy *KpiDTO* példányt, melyet meg szeretnénk jeleníteni. Létrehoz továbbá egy *SvgRootElement* példányt, amelyhez leszármazottai fogják hozzáfűzni a megfelelő SVG elemeket. Ez az osztály tartalmaz egy absztrakt metódust, melynek neve *draw().* Az egyes konkrét chart osztályok különbözőképpen implementálják ezt a metódust. Alapvetően két típusú diagram van a programban, a vonal és a tortadiagram.
Mint említettem két típusú vonaldiagramot készítettem (két és három dimenziós), melyek közös

tulajdonságait egy absztrakt osztályba emeltem ki, melynek neve *AbstractLineChart*. Ebből származnak a Linechart2D, illetve a *LineChart3D* konkrét osztályok. Az egyetlen különbség a két típus között, az értékeket reprezentáló körök mérete, amely a három dimenziós verziónál súlyozva jelenik meg, tehát dinamikusan változik a kör mérete. Ez felvet néhány problémát, amelyről később beszámolok.
A másik típus a tortadiagram, amely a programban a *PieChart* nevet viseli.
Lássuk mire volt szükségem ezek megvalósításakor.

A vonal diagram

A vonaldiagram rajzolásának algoritmusa meglehetősen egyszerű: pontokat kell kirajzolni és ezeket vonalakkal összekötni. A diagram rajzolásakor ismerjük a rendelkezésre álló hely magasságát és szélességét. Ezekből a méretekből előre definiált szélességű darabokat vágtam le (padding), ezzel hagyva helyet az alsó és oldalsó diagram feliratoknak. A megadott padding-ek egy új belső négyzetet határoznak meg, amelybe a diagram rajzolása fog történni.

```
viewPortWidth = width - rightPadding - leftPadding;
viewPortHeight = height - topPadding - bottomPadding;
```

Ezen behúzások használata nem csak esztétikai szempontból hasznos, a háromdimenziós változatnál ennél nagyobb szerepet kap majd. Szélesség és magasság alatt ezentúl az újonnan kiszámított, belső négyzet méretét értem.
Egy pontot a síkon két koordináta határoz meg (x,y). Az x koordináták meghatározásához a rendelkezésre álló hely szélességét kell elosztani a KPI értékek számával. Így megkapjuk a pontok x tengelyen egymástól való távolságát, amellyel inkrementálva az első pont x koordinátáját (az első pont x koordinátája *leftPadding* értékével egyenlő), sorra egyforma távolságra tudunk pontokat rajzolni.
A nagyobb problémát a kör függőleges elhelyezése, az y koordináta kiszámolása okozta. A magasság függvényében ki kellett számolni egy skála-értéket abban a tudatban, hogy a legnagyobb KPI értéket a diagram legmagasabb pontjára szeretnénk elhelyezni. A skála érték a magasság és a legmagasabb KPI érték hányadosa. Ezen értékkel és az alábbi skálázó metódussal, minden KPI értéket el tudunk helyezni a diagram y tengelyén.

```
protected int scalePoint(double value) {
        return (int) (viewPortHeight - (value * scale) + topPadding);
}
```

A skálázás a KPI értékek és a képernyő pixelei közötti egyértelmű megfeleltetést teszi lehetővé. Ezután a KPI értékeken végig-iterálva berajzolhatóak a megfelelő pontok és a köztük lévő vonalak.

A diagramhoz tartozik még egy vízszintes rácsozás amelyeken az értékek szerepelnek, illetve a diagram alján elhelyezkedő dátumok tájékoztatnak az egyes értékek időpontjáról amelyek 60°-os dőlésszögben jelennek meg. Az értékeket reprezentáló körökre definiáltam továbbá egy eseményfigyelőt, amely az egérrel való rámutatás esetén (onMouseOver esemény) egy kis felugró buborékban (tooltip) ad bővebb tájékoztatást a kör által reprezentált értékről.

Ettől egy kicsivel bonyolultabb az értékek súlyozása, azaz a háromdimenziós vonaldiagram. A körök rajzolása azonos módon történik a kétdimenziós társával, annyi különbséggel, hogy ebben az esetben a kör átmérője dinamikusan változik. Az alábbi metódus számítja ki a rajzolandó kör méretét, a bemenő *recordNumber* paraméter függvényében.

```
private int circleScale(int recordNumber) {
        int maxRecordNumber = kpi.getMaxRecordNumber().getRecordNumber();
        circleScale = circleMaxR / maxRecordNumber;
        int scaledCircleR = (int) (circleScale * recordNumber);
        if(scaledCircleR < circleMinR){
                return circleMinR;
        }
        return scaledCircleR;
}
```

A *circleMinR* és *circleMaxR* a rendelkezésre álló hely figyelembevételével kiszámolt maximális és minimális körméretet jelenti.

Felmerült egy probléma a változó méretű körök megjelenítésénél. Ha a diagram széleire nagy méretű kört kellett rajzolni, az lelógott a diagramról, elrontva ezzel a megjelenést. Ezen probléma kezelésére született a fent említett padding-ek bevezetése. A háromdimenziós diagramnál a paddingek dinamikusan, a várható legnagyobb kör mérete szerint változnak, így elérve, hogy minden kör teljes egészében látható maradjon a diagramon.

A kör diagram

A kör diagram annyi darab körcikkből épül fel, ahány KPI értéket ábrázol. Egy körcikket egy ív (amely a körvonal egy szakasza) és két sugár határol. Először is ki kell számolni, hogy az értékekhez viszonyítva, egy egység (egy forint vagy egy óra, attól függ mi a mértékegység) hány fokot jelent a körben. Mivel a kör 360 fokos, ezt egyszerűen az alábbi képlettel számolhatjuk:

```
angle = (kpi.getKpiEntryList().size() / kpi.getSumPrice()) * 360
```

ahol a *getSumPrice()* metódus végig-iterál a KPI elemeken összeadva azok értékeit. Ezután a fokot radiánba átszámolva az alábbi két képlettel kapjuk meg a következő ívet meghatározó pont koordinátáit:

```
rad = Math.radian(angle)
nextx = (int)(Math.cos(rad) * R)
nexty = (int)(Math.sin(rad)*R)
```

ahol R a kör sugara. Ezután egy SVG <path> elememmel leírható a két egyenes és a körív, amely meghatározza a cikkelyt. A KPI értékeken végigérve megkapjuk a teljes kört, amely cikkenként más színű. Az egyes cikkekre úgyszintén definiáltam eseménykezelőt, amely a rámutatáskor buborékban tájékoztat az aktuális cikkelyről. Tartozik továbbá egy jelmagyarázat (legend) a diagramhoz, amely a kördiagram mellett jobbra helyezkedik el, és felsorolás-szerűen jelzi a színekhez rendelt értékeket. Ha ezekre a színes négyzetekre mutatunk az egérrel, a diagramban a hozzá tartozó torta-cikk át-színeződik, jelezvén melyik szeletről van szó.

7. Szoftverevolúció

Miután a bemutatott alkalmazásom elérte első verziójának végleges formáját, a szoftverfejlesztés életciklusának a kiadás utáni részéről szólnék néhány szót. Egy szoftver soha nem lesz tökéletes, befejezett, és az azt használó teljes felhasználói bázis igényeinek kielégítője. A kiadás után, a tesztelésen átcsúszott hibák javítása mellett, további feladatunk is van a szoftverrel. A felmerülő újabb igények miatt, és hogy a szoftver ne avuljon el, szükséges a további fejlesztés. Az új funkciókat verziókba szedve adjuk ki, ezzel biztosítva a folyamatos fejlődést. A felmerülő új elvárások priorizálása után el kell döntenünk, melyek a legfontosabbak az egyes verziókba. Ahhoz, hogy ténylegesen jól fejleszthető kódot írjunk sok mindenre oda kell figyelni.

7.1. A jó kód fejleszthető

Törekednünk kell a szép és jól olvasható kód írására, amely nagyban megkönnyíti a későbbi fejlesztést. Nem is beszélve arról, ha nem ugyanaz az ember végzi a későbbi fejlesztést, mint aki elkezdte a projektet. Mivel ez nagyvállalati környezetben igen gyakran előfordul, a nagy cégek különös hangsúlyt fektetnek az alábbi konvenciók betartására.

A Java nyelvnek is mint minden másnak, vannak jól bevált konvenciói melyek egyfajta íratlan törvényekké fejlődve beleégnek minden Java programozó tudatába. Ilyen alapvető szabály a „CamelCase”, mely egy elnevezési konvenció és azt írja elő hogy az azonosítók (metódus, osztály, stb… nevek) szavanként nagybetűvel kezdődjenek (metódusok és objektum példányok nevei csak a második betűtől). Egy másik, hogy a konstansok nevei csupa nyomtatottal írottak és „_”-el elválasztva szerepelnek benne a szavak.

Fontos az elnevezések átgondolása. A jó forráskódban egyértelműen kiderül mindenről, hogy mire gondolt a fejlesztő amikor implementálta azt. Mennyivel egyértelműbb például a „*getIncrementedCounterForResource*” mint a „*get_incCount_for_rsc*”. Megoszlanak a vélemények azok között, akik ragaszkodnak a forráskódba írt kommentek használatához, és akik nem. Véleményem szerint, a fenti módon elkeresztelt eszközök használata esetén sok esetben fölöslegessé válik a kommentek írása minden metódushoz. Egy metódus specifikációnak önleírónak kell lennie, ha elolvassuk, mintha egy mondatot olvasnánk, érthetővé válik mit is valósít meg, így nincs szükség további magyarázatra. Kivétel ez alól persze ha publikus API-t készítünk, vagy ismertetni szeretnénk valamilyen implementációs részletet a kód felhasználójával.

A metódusok ne legyenek túl hosszúak Sokkal emészthetőbb, ha kiemelünk bizonyos funkcionalitást sok rövid metódusba, ezzel javítva az olvashatóságot. Általában ezzel nő a kód mérete, de az érthetőség nagyban javul, és a mai modern fejlesztői környezetek leveszik a vállunkról a néha hosszúra sikeredett metódusnevek begépelésének terhét.

Emellett vannak cégek által definiált konvenciók, mint például a formázás (formatting). Ahhoz, hogy egységes kinézetű kódot készítsen minden fejlesztő, a cégek gyakran definiálnak egy formát, amelyet az IDE-ben könnyedén mindenki egy gombnyomásra használatba vehet. Ez biztosítja, hogy a megfelelő helyekre odakerüljenek a szóközök, a túl hosszú sorok tördelve jelenjenek meg, és a fájl végi fölösleges üres sorokat is eltávolítja.

Nagyon fontos dolog a kód kézben tartása. A legjobb eszköz az újratervezés, azaz a refactoring. A kódot folyamatosan refactorálni kell, ezzel javítva a minőségét. Ha valami két helyen szerepel, biztos hogy refactorálásra van szükség. A kódrészlet megfelelő helyre mozgatásával tisztul a kód és javul a program szerkezete. A fejlesztés során folyamatosan használtam a refactoring-ot, hiszen véleményem szerint ez az agilis szoftverfejlesztés egyik elengedhetetlen eszköze.

7.2. A következő verzió

A Visualization Dashboard-al kapcsolatban rengeteg további funkció vár megvalósításra, melyek az idő szűkössége miatt nem készültek el. A következő verzióba mindenképp bekerül egy az XML importálásnál intelligensebb, webszolgáltatáson (WebService) keresztüli adatbetöltés, amely a külső szoftverekkel való közvetlen kommunikációt valósítja meg. A következő lépés a diagramtípusok bővítése összetettebb diagramokkal. A későbbi fejlesztések során bekerülhet egy a jelenlegi PUSH adatbevitel helyett az úgynevezett PULL módszer, amikor az adatokat nem külső alkalmazás tölti a vizualizációs szoftver adatbázisába, hanem az alkalmazás nyúl ki különböző adatbázisokhoz, és megjeleníti azok adatait. Hasznosnak tartom az alkalmazás mobil platformokra (például Android) való elkészítését, amellyel a mobilitás igen magas szintje érhető el.

7.3. A jövő

Véleményem szerint a döntéstámogatási rendszerek a jövőben még nagyobb szerepet kapnak majd, hiszen az egyre bonyolódó üzleti folyamatok megértését és kezelését nagyban megkönnyítik. Az igények növekedésével egyre intelligensebb és hasznosabb eszközök jönnek majd létre a vizualizáció terén is, hiszen az adatbázisok egyre csak nőnek és a nagy adathalmazok megértése továbbra is nagy

kihívást jelent számunkra.
A jövőben a jól ismert átlagos diagramokat (mint a vonal, oszlop vagy torta-diagram) minden bizonnyal sokkal beszédesebb és látványosabb társaik váltják fel, amelyek még szemléletesebben tudják prezentálni az adatokat. A különböző típusú, de egymással összefüggő adatok egy diagramon való ábrázolása még-nagyobb hangsúlyt nyerhet, hiszen addig még nem látott viselkedéseket deríthetünk ki vele. Az ilyen összefüggések felderítése mindenképpen nagy hatással van az emberiség fejlődésére, így erre talán nagyobb hangsúlyt kellene fektetnünk.
Én személy szerint a jövőt az eszközök integrációjában látom minden téren. Gondolok itt sokkal rugalmasabb, és intelligensebb vizualizációs eszközökre, melyek bármilyen adatbázisból képesek a megfelelő adatok aggregációja után érdekes kimutatásokat készíteni.

8. Befejezés

Dolgozatom végéhez közeledve összefoglalnám a dolgozatban leírtakat, visszatekintve a kezdeti célkitűzésekre az elkészült alkalmazás birtokában.

Megpróbáltam bemutatni a vizualizációt mint üzleti-intelligenciát és érzékeltetni jelentőségét és hatékonyságát számos példán keresztül.

Ezután bemutattam az elkészített vizualizációs szoftvert, amely egy webes alkalmazás. Bemutattam a használt platformot, a Java Enterprise Edition-t, és az általa biztosított technológiák egy részét, amelyeket használtam. Ilyenek az EJB technológia, az adatbázis-kapcsolatok gyorsítótárazása (Connection Pooling), és a többi. Ismertettem a GWT asszinkron alkalmazásfejlesztéshez szükséges eszközeit, mint a history-kezelés vagy az XML-RPC használata. Illetve felhasználói felületének MVP minta alapján való elkészítését. Bemutattam a vektorgrafikus rajzolás előnyeit és beszámoltam néhány, a fejlesztés során felmerült problémáról és azok megoldásairól.

Az elkészült alkalmazással elégedett vagyok, a kezdetekkor kitűzött célokat teljesíti, és úgy vélem az éles használat során is megállná a helyét. Mindemellett ismerem a korlátait, azokat a funkciókat, amelyekkel bővíteni szeretném, és amelyek még finomításra szorulnak.

A fejlesztés alatti munkát az IT-Services Hungary debreceni székhelyének kellemes légköre és az ott dolgozók barátságos fogadtatása kikapcsolódássá változtatta.

Összességében tehát megérte a fáradtság, hiszen a fejlesztés alatt újabb tapasztalatokkal lettem gazdagabb. Elmélyítettem GWT és SQL ismereteimet, felelevenítettem az XML-ről és XSD-ről tanultakat, és újakat szereztem főként SVG és J2EE területen.

9. Köszöntenyílvánítás

Köszönetet szeretnék mondani

- Vágner Anikónak, tanáromnak és témavezetőmnek a támogatásáért és tanácsaiért, amelyek sokat segítettek e dolgozat elkészülésében
- Bettenbuk Zoltánnak, külső konzulensemnek, aki idejét nem sajnálva fogadott munkahelyén, hogy segítsen. Köszönet az ötleteiért, melyek segítettek elindulni, és folyamatos szakmai támogatásáért, amiért mindvégig bátran fordulhattam hozzá
- Vajda Tibornak és Tóth Csabának, akik pozitív irányba mozdították programozói szemléletemet és olyan hasznos tudást adtak át, melyeket a fejlesztés alatt folyamatosan használni tudtam
- családomnak, akik mindvégig támogattak elképzeléseim megvalósításában, és az egyetemi tanulmányaim alatt felmerülő problémáim megoldásában
- barátaimnak és csoporttársaimnak, akik vagy szakmai tudásukkal vagy biztatásukkal támogattak a három éves egyetemi tanulmányaim alatt

10. Irodalomjegyzék

Könyvek

[1] Ryan Dewsbury (2008): Google Web Toolkit alkalmazások

[2] Henrik Kniberg & Mattias Skarin (2009): Kanban and Scrum – making the most of both

[3] Martin Flower (2006): Refactoring

Internet

[4] http://www.biprojekt.hu/Uzleti-intelligencia-Business-Intelligence-BI.htm

[5] http://www.ted.com/talks/lang/eng/david_mccandless_the_beauty_of_data_visualization.html

[6] http://www.koloknet.hu/?354-tanulsi-stlusok

[7] http://en.wikipedia.org/wiki/Performance_indicator

[8] http://download.oracle.com/javaee/6/firstcup/doc/gcrky.html

[9] http://www.w3schools.com/svg/

[10] http://wiki.scribus.net/canvas/Making_a_Pie_Chart

11.Függelék

11.1. Képek jegyzéke

(1. ábra) Fiktív cég fejlesztési és támogatási idejei (órában)

(2. ábra) A bejelentkezési képernyő

(3. ábra) Visualisation Dashboard

(4. ábra) A háromdimenziós vonaldiagram használati esete

(5. ábra) Dashboard beállítások oldala

(6. ábra) Adatmódosítás oldala

(7. ábra) Az entitások osztálydiagramjai UML-ben

(8. ábra) Az Svg osztályok osztálydiagramjai UML-ben

(9. ábra) A J2EE architektúra

(http://download.oracle.com/docs/cd/B31017_01/migrate.1013/b25219/img/j2ee.gif)

(10. ábra) Connection pooling a J2EE architektúrán

(http://java.sun.com/developer/technicalArticles/J2EE/pooling/j2eepool_fig3.gif)

(11. ábra) A GWT keresztfordítójának szemléletes ábrázolása

(https://confluence.sakaiproject.org/download/attachments/23330834/GWTCrossCompiler.PNG)

(12. ábra) Az MVP tervezési minta

(http://acris.googlecode.com/svn/wiki/images/mvp_small.png)

(13. ábra) Üzenet a felhasználónak