Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Sallai Tamás

BANKI ALRENDSZER TERVEZÉSE, MEGVALÓSÍTÁSA ÉS INTEGRÁCIÓJA SZOLGÁLTATÁSORIENTÁLT KÖRNYEZETBEN

KONZULENSEK

Berényi Zsolt

Imre Gábor

HALLGATÓI NYILATKOZAT

Alulírott **Sallai Tamás**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök, stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Tudomásul veszem, hogy az elkészült szakdolgozatban található eredményeket a Budapesti Műszaki és Gazdaságtudományi Egyetem, a feladatot kiíró egyetemi intézmény saját céljaira felhasználhatja.

Kelt: Budapest, 2010. 10. 30.

.....

Sallai Tamás

Tartalomjegyzék

Összefoglaló

Az internet elterjedésével a cégek működésük mind nagyobb részét helyezték informatikai alapokra. Ezeket a szolgáltatásokat a felhasználók kényelmesen el tudják érni a weben keresztül, ugyanakkor ez a felhasználás az eddigiektől eltérő rendszerek szükségességét idézte elő. A nagyvállalati IT rendszerek jellegzetessége, hogy több kiszolgálóból állnak, magas rendelkezésreállást kell biztosítaniuk, az üzleti folyamatok könnyen áttekinthetőek és módosíthatóak legyenek, valamint hogy működés közben is lehessen őket megfigyelni.

Ezen igények kielégítésére számos technológia és szabvány született, melyek eltérő módon felelnek meg ezen követelményeknek, valamint a már meglévő rendszereket is különbözően támogatták. Federált környezetben kulcsfontosságú egy, minden rendszer által érthető és támogatott, kommunikációs protokoll. Bonyolult működés esetén felmerült az igény az átlátható üzleti folyamatok tervezésére is, mely a felhasználható szolgáltatások implementációját elrejtve csak a működésre helyezi a hangsúlyt.

Szakdolgozatomban egy bemutató banki befektetési és bejelentkező modult mutatok be és valósítok meg. Ennek során leírom a kész alrendszer elvárt viselkedését, kitérek a rétegek tervezésére, valamint az implementáció során felmerült problémákra. Ezeken keresztül az Olvasó egy átfogó képet kaphat arra vonatkozóan, hogy a bemutatott eszközökkel egy valós problémát miként lehet megoldani.

Abstract

With the progressive increase in the Internet penetration, corporations more and more based their operation on the IT. The clients can easily reach these services through the web, although this requires different infrastructure and systems. Some characteristics of the enterprise IT applications are that they consist of multiple servers, they must provide high availability, the business processes must be easily readable and modifiable, and they must be monitorable in runtime.

Numerous new technologies and standards emerged, that fit these requirements differently, and they don't necessarily support legacy systems. In a federated environment it is vital to have a communication protocol, that all participants understand and support. For complex operations, the need has risen to develop easily readable processes, that conceal the implementation of services and emphasise functionality.

In my final thesis I introduce and realize a demo banking investment and authentication module. In the course of the process, I describe the specification of the final system, the design of the layers and the challenges throughout the implementation. After these, the Reader should have an actual insight about the usage of the presented technologies to solve a real-world problem.

1 Bevezető

A szakdolgozatom feladata egy banki alrendszer megvalósítása. Annak ellenére, hogy ez csak prezentációs célokat szolgál, olyan előfeltevésekből indulok ki, és olyan technológiákat használok fel az elkészítése során, melyek éles környezetben használt programok esetén is kielégítenék az igényeket. Ezzel a feladat megoldása közben megismerem és bemutatom a nagyvállalati IT környezetben történő felhasználásra készült szabványokat és megvalósításokat.

Az alaprendszer implementációjához Java EE-t választottam, mivel ez széles körben elterjedt, a legnagyobb ipari szereplők részvételével folyamatosan továbbfejlesztett, valamint kielégíti a vállalati igényeket.

A modulok közötti kommunikációhoz a webszolgáltatásokat választottam. Mivel ez szöveges protokoll, ezért rendszer-, és platformfüggetlen módon tudja összekötni a kommunikáló feleket, valamint széles körben támogatott.

Az üzleti folyamatokat BPEL-ben valósítottam meg, ezáltal azok könnyen karbantarthatóak és megbízhatóan működnek, valamint támogatja a webszolgáltatások kezelését, így könnyen integrálhatóak az elkészített modulhoz.

A szakdolgozatom első felében részletesen vázolom az alkalmazott technológiákat, az alaprendszertől felfelé haladva a kommunikáción keresztül a folyamatokig. Második felében az implementált alkalmazást mutatom be. A feladat leírása után a tervezési lépéseket, szintenként felfelé haladva, utána a fejlesztés során felmerült problémákat és megoldandó feladatokat emelem ki az általam adott megoldásokkal együtt. Végül vázolom az alkalmazásom kiterjesztési lehetőségeit, valamint összefoglalom a tapasztalataimat.

2 Technológiai áttekintő

2.1 Java EE

2.1.1 Általános bemutatás

A Java EE[1] (Java Enterprise Edition) az SE (Standard Edition)-re épül, kiegészítve vállalati igényeket kielégítő egyéb szabványokkal. Tervezésekor elsősorban a skálázhatóságra, a stabilitásra és a biztonságra helyezték a hangsúlyt, ennek megfelelően a futtatási környezet nagyobb és erőforrásigényesebb lett. A jelentős többletszolgáltatás miatt nem elég hozzá az alap JVM (Java Virtual Machine), hanem egy alkalmazásszerveren belül futnak a programok. A szabványos Java EE alkalmazások ezen szerverek között hordozhatóak, erre a TCK (Technology Compatibility Kit) ad garanciát. Azonban az egyes alkalmazásszerverek fejlesztői a szabványosokon kívül egyéb gyártóspecifikus jellemzőket is elérhetővé tehetnek, melyeket felhasználva esetleg jobb megoldásokat készíthetünk a hordozhatóság árán. A Java EE tehát egy szabványgyűjtemény, melyet az alkalmazások használhatnak.

A legtöbb webes kiszolgáló-oldali rendszer felépítése 3 rétegű. Legalul található az adatréteg, melyet egy relációs adatbáziskezelő rendszer nyújt. E fölött helyezkedik el az üzleti logikai szint, mely az adatok feldolgozásáért és az üzleti folyamatok megvalósításáért felel. Végül legfelül található a megjelenítési réteg, mely a szolgáltatásokat teszi elérhetővé a felhasználó felé webes technológiák segítségével. A Java EE az utóbbi két réteget, valamint egy adatelérési szintet foglal magában, mely az adat-, valamint az üzleti logika réteg összekötését hivatott megoldani. Ezt objektum-relációs leképzéssel valósítja meg, melynek során a java objektumokat relációs adatbázisba lehet leképezni. Vannak rétegfüggetlen szolgáltatásai is, ide tartozik a biztonság kezelése, valamint a tranzakciókezelés. A konténeren belüli három szintet, valamint a réteghez nem köthető technológiák közül a JTA[2]-t részletesen bemutatom a következő fejezetekben.

Mint heterogén környezetbe szánt technológia, támogatja az elterjedt kommunikációs protokollokat. Ezek közül a webszolgáltatásokat fogom részletesen bemutatni.

2.1.2 Adatelérési réteg: JPA

A JPA[3] (Java Persistence API) felelős a java objektumok relációs adatbázisban való elmentéséért, lekéréséért valamint módosításáért. Ehhez felhasználja a JDBC[4] (Java Database Connectivity)-t, melynek segítségével el tudja érni az adatréteget. A JPA az objektumrelációs leképzés szabványosítása, tehát egy konkrét API (Application Programming Interface), mely mögött az implementáció szabadon változtatható. Sok esetben felmerülő probléma volt, hogy különböző adatbáziskezelők eltértek az SQL[5] szabványtól, emiatt az egyikhez írt program nem volt hordozható közöttük. Ezt a legtöbb megvalósítás az ún. SQL dialektusokkal oldotta meg, mely szerint a JPA által generált tényleges SQL utasítások a beállított környezetnek megfelelően generálódnak le.

A perzisztens objektumok egyszerű java osztályok példányai, nincsen szükség technológiaspecifikus interfész implementálására. Az entitások konfigurálása deklaratívan történik, annotációk segítségével, ugyanakkor az orm.xml fájlban felüldefiniálható. Ennek megfelelően csak meg kell jelölnünk az osztályt entitásként és ezáltal az osztály perzisztencia-képes lesz. Ilyenkor minden entitás általában egy-egy tábla lesz az adatbázisban, ahol az oszlopai az osztály tulajdonságai, sorai pedig a példányai lesznek. Alapértelmezés szerint a tábla neve az osztály egyszerű neve (a teljes java nevének az utolsó pontja utáni rész), az oszlopok nevei pedig az attribútumok nevei lesznek, de ez felüldefiniálható. Minden entitásnak szükséges, hogy legyen egy egyedi azonosítója, tehát az egyik mezőt meg kell jelölnünk *@id*-vel. Legtöbb esetben nincs szükség arra, hogy minden egyednek magunk állítsuk be ezt az azonosítót, ezért megadhatunk generációs stratégiát, mely egy megfelelőt fog rendelni hozzá minden létrehozáskor. Ennek a legegyszerűbb este, amikor szám típusú a tulajdonság és egy növekvő szekvencia következő eleme lesz az értéke.

Lehetőségünk van kapcsolatok kialakítására is, melyek lehetnek egy-egy, egy-több, több-egy, valamint több-több kardinalitásúak. Az egy-egy esetben az egyik entitás egyik attribútuma lesz a másik entitás, az adatbázisban pedig az adott oszlop a másik tábla megfelelő sora azonosítójának az értékét fogja felvenni. Egyetlen célentitáshoz csak egyetlen másik entitás tartozhat. A JPA nem generál idegen kulcsokat az adatbázisba,

azonban azzal azonos funkcionalitást biztosít, saját magán belül lekezeli a műveletek kaszkádosítását. Ennek beállításához a kapcsolatra megadhatjuk a kaszkádosítás fokát, melynek segítségével elérhetjük, hogy ne maradjanak érvénytelen hivatkozások az adatbázisban. Az egy-több és a több-egy kapcsolatok megadása párban történik. Hatása megegyezik az adatbázisban az idegen kulcséval. Az ‘egy’ oldalra meg kell adnunk, hogy a másik entitás melyik attribútumával áll párban, valamint ez egy kollekció lesz. A ‘több’ oldalon elég csak annotációval megjelölnünk a tulajdonságot. Az adatbázisban a kapcsolat az egy-egy esettel azonos módon jelenik meg, azonban itt a JPA engedi a ‘több’ oldalon több azonos entitásra referáló entitást felvenni. Több-több kardinalitás esetén mindkét oldalon meg kell adnunk a másik oldalon használt attribútum nevét, valamint mindkét oldalon kollekció lesz. Adatbázis szinten egy kapcsolótábla jön létre, amely oszlopai a két tábla azonosítói, ennek segítségével tudja összekötni az entitásokat.

Az osztályok közötti öröklődést is lehetőségünk van leképezni. Ezt háromféle módon tehetjük meg. Egyrészt megtehetjük, hogy egyetlen táblát hozunk létre, mely a teljes osztályhierarchia összes attribútumával rendelkezik, ezen kívül pedig egy diszkriminátorral, amely megadja, hogy az adott sor éppen melyik osztály példányát tárolja. Ehhez a megoldáshoz a szülőosztályt meg kell jelölnünk mint egy olyan entitás, amelynek lehetnek leszármazottai, valamint öröklési stratégiának meg kell adnunk, hogy egyetlen táblába szeretnénk leképezni. Ezen kívül meg kell adnunk a hierarchia összes osztályában, hogy milyen értéket vegyen fel a diszkriminátor oszlop az adott entitástípus esetén. Természetesen ennek egyedinek kell lennie a hierarchián belül. A másik megoldás az öröklés leképezésére, hogy osztályonként hozunk létre egy-egy táblát, azok között pedig idegen kulcsokkal teremtünk kapcsolatot. Ez esetben is szükséges használni a diszkriminátor tulajdonságot. Harmadik megoldásként minden konkrét osztály 1-1 tábla lesz, mely az összes attribútummal rendelkezik.

Az entitások lekérésekor azon mezők, amelyek kapcsolatokat fejeznek ki más entitások felé még nem feltétlenül töltődtek be. Ez minden kapcsolatra egyesével szabályozható. Alkalmazható lusta betöltés, ebben az esetben a kérdéses tulajdonság csak hozzáféréskor lesz beemelve az adatbázisból. Ugyanakkor lehet korai betöltés, ekkor az első entitással együtt lesz inicializálva az attribútum. Az előbbi esetben kevesebb

memóriára lesz szükség, valamint az adatbáziskapcsolaton is kevesebb adat érkezik, ugyanakkor többször kell lekérdezéseket futtatni mint a második esetben. Utóbbi esetben azonban nem árt figyelni arra, hogy ha a korán betöltött entitások is betölthetnek ilyenkor másokat, ezzel egy-egy lekéréskor méretes objektumgráf is létrejöhet.

A JPA az entitások lekérdezéséhez a JPQL (Java Persistence Query Language)-t használja, mely az SQL-hez hasonló, de objektum-orientált lekérdező. Ennek segítségével az általános kifejezőrőn túl hatékonyan fogalmazhatunk meg kifejezéseket az entitásaink közötti kapcsolatok bejárásához. A JPQL kifejezések végrehajtódáskor a kiválasztott dialektusnak megfelelő SQL kifejezésekre fordulnak le, és azt küldi le az adatbáziskezelőnek.

2.1.3 Üzleti logika réteg: EJB

A háromrétegű architektúra második szintje az üzleti logika réteg. Ez a szint felel a tényleges alkalmazáslogikáért, itt kapnak helyet az üzleti folyamatok, valamint ez köti össze a megjelenítést az adatokkal. A Java EE-ben ez az EJB[6] (Enterprise Java Beans) technológiával van megoldva, mely alapvetően java beanek felhasználására épül. Ezen beanek olyan eldobható java osztályok, melyek üres konstruktorral példányosíthatóak, valamint minden tulajdonságukhoz van beállító és lekérdező (getter és setter) függvény definiálva. Ezáltal ezen komponensek egyszerűen lementhetőek például adatbázisba, hogy onnan szükség esetén bármikor azonosan visszaállíthatóak legyenek. Ezáltal az operatív memóriában elférőnél több beant is létrehozhatunk, valamint egy terheléselosztási fürt egyik tagja által létrehozott komponenseket egyszerűen átvihetjük egy másik tagra. Ezáltal az EJB jól skálázódik több kiszolgálóra is, így beleillik a vállalati IT környezetbe.

Háromféle beantípust különböztetünk meg. Vannak az entitás beanek, melyek helyét az újabb szabványokban a JPA által is kezelt szabványos entitások veszik át. Ezek azok a perzisztens objektumok, melyek tulajdonságai relációs adatbázisba mentődnek, és onnan is állíthatóak vissza. Ezután vannak az MDB (Message-Driven Bean)-k, melyek

JMS[7] (Java Message Service) üzeneteket dolgoznak fel aszinkron módon, ezáltal nem is példányosíthatóak és hívhatóak közvetlenül.

Végül vannak a session beanek, melyeket a konténer példányosít, valamint az életciklusukat is kezeli, ezek valósítják meg a program hívható logikáját. Ezen belül megkülönböztetünk állapottal rendelkező és állapotmentes session beant. Az előbbi esetében ha szereztünk egyre referenciát, akkor az egymás utáni metódushívások garantáltan ugyanazt az objektumot, más referenciák pedig garantáltan másik objektumot fognak hívni. Ezáltal a bean állapota függvényhívások között megmarad, viszont a konténernek meg kell őrizni ezt. Az állapotmentes beanek hívásakor ezzel szemben semmilyen garanciát nem kapunk arra nézve, hogy melyik példányt is fogjuk hívni, ezáltal az objektum állapota nem ismert, ugyanakkor a futtató környezet csak a szükséges minimális mennyiségű beant példányosítja, valamint ha már nincs rá szükség, akkor következmények nélkül eldobja. Ezen kívül a konténer az állapotmentes beanek metódusait futtathatja egyszerre több szálon is. Tipikusan a jellemzőkkel rendelkező beanek állapottal rendelkezők, e nélküliek pedig állapotmentesek lesznek.

Egyik lehetőség beanekre referenciát szerezni a JNDI (Java Naming and Directory Interface)-n keresztül. Ez egy olyan szolgáltatása a java rendszereknek, amivel objektumokat lehet névhez kötni, valamint ezen név alapján bárhonnan elérhető. A JNDI a neveket könyvtárakba rendezi, ezen belül a neveknek egyedieknek kell lenniük. A könyvtárak egymásba is ágyazhatóak, ezáltal egy hierarchia jön létre, melyben minden csomópont lekérhető, valamint a gyerekeinek a nevei is. Minden session bean típus egyedi névvel bekerül ebbe a struktúrába, így a konténertől ezen név ismeretében kérhetünk példányokat.

Másik lehetőség referencia szerzésére a függőséginjektálás felhasználása. Ez a bean egy megjelölt tulajdonsága, melyet a futtató környezet használat előtt inicializál a megfelelő típusú objektummal. Ennek egyik előnye, hogy nem kell tudnunk előre, hogy a JNDI hierarhiában hol fog elhelyezkedni az adott bean típus, mely esetenként a gyártók között eltérő is lehet, hanem csak a típusát ismerve is tudunk egy példányra referenciát szerezni.

A session bean implementációt kétféle interfészen keresztül lehet hívhatóvá tenni. Az egyik a helyi, a másik a távoli nézet. Előbbi esetében a metódusok paraméterei referenciaként adódnak át, ezáltal gyors lesz, ugyanakkor csak azonos futtatási környezetben alkalmazható. Távoli nézeten keresztüli hívás esetén az argumentumok érték szerint adódnak át, valamint átesnek egy szerializációs/deszerializációs lépésen, melynek során karakterfolyammá lesznek alakítva, a túloldalon pedig vissza. Ennek jelentős számításigénye van, ugyanakkor ennek a használatával elosztott környezetben is hívható lesz az implementáció. Természetesen mindkétféle nézetet létrehozhatjuk, és majd a referencia szerzésekor döntjük el, hogy az elosztottság vagy a sebesség a fontos az adott esetben. Tipikusan a mindig együtt előforduló beaneknek éri meg a helyi nézetet használni, míg máshol a távolit, egészen addig, amíg nem fordulnak elő teljesítményproblémák.

A session bean létrejövése és megsemmisülése között lehet aktív vagy passzív állapotban. Előbbiben kész metódushívásokat kiszolgálni, utóbbiban a futtató környezet a tulajdonságait háttértárra írta, ahonnan bármikor visszahozható, de ebben az állapotban nem hívható. Az aktiválás/passziválás műveletét a konténer kezeli, melybe közvetlen beleszólásunk nincs, azonban definiálhatunk olyan metódusokat, melyek meghívásra kerülnek állapotváltás esetén. Ezek megjelölt, argumentum nélküli függvények, melyet a környezet az elvárt megfelelő időben meghív, ezáltal bizonyos műveleteket végrehajthatunk. Az alábbi jelölések alkalmazhatóak az életciklus kezelésére:

- PostConstruct: a bean létrehozása után hívódik meg
- PreDestroy: megsemmisülés előtt hívódik
- PostActivate: aktiválás után
- PrePassivate: passziválás előtt

Fontos megjegyezni, hogy ezen metódusok hívásai között a bean állapota megőrződik, ugyanis mindegyik ugyanarra a példányra vonatkozik. Ezeken kívül felvehetünk egy speciális metódust, amely minden más hívás körül fog lefutni, ezzel lehet befolyásolni az eredményt, vagy pl. naplózni.

Az entitásokat, általuk pedig az adatbázist az entitás menedzseren keresztül lehet elérni. Ez tartja a kapcsolatot a perzisztencia kontextussal. Ezen környezeten belül léteznek az entitások, és ez felel az életciklusukért is, ezáltal képes az egyediséget biztosítani, valamint olyan többletszolgáltatásokra, mint a gyorsítótárazás. A menedzser rendelkezik azokkal a metódusokkal, melyekkel az entitásokat lehet lekérdezni, létrehozni, törölni vagy módosítani. Az elérhető entitások halmazát a perzisztencia egység adja meg, mely az egy adatbázisba kerülő osztályokat tartalmazza. Ez névvel van megadva, és minden entitás menedzserhez pontosan egy van rendelve. A perzisztencia szolgáltató implementációját is a perzisztencia egységben kell megadni. Az entitás menedzser objektumra referenciát függőség injektálással lehet szerezni, ahol meg kell adnunk hozzá a használt egység nevét.

A session beanek metódusait le lehet védeni, hogy az RBAC[8] (Role Based Access Control) szerint csak adott szereppel rendelkező felhasználó tudja csak meghívni. Ennek eléréséhez a Java EE biztonsági megvalósítását, a JAAS (Java Authentication and Authorization Service)-t használja. Ennek megfelelően a felhasználókhoz szerepek vannak rendelve, és a biztonsági szabályok ezekre vannak létrehozva. Amennyiben a hívó nem rendelkezik a szükséges szereppel, akkor kivételt vált ki a metódushívás, ezáltal meghiúsul.

A beanek metódusai tranzakciókba rendelhetőek, melyek a JTA-ról szóló fejezetben lesznek részletesen ismertetve.

2.1.4 Megjelenítési réteg: Szervlet, JSP, JSF

Szervlet

A szervlet[9] olyan java komponens, amely képes kérésekre dinamikus választ adni, legtöbbször HTML formában, ezáltal képes az alkalmazásoknak webes megjelenítési felületet adni. Futtatása egy szervlet konténerben történik, mely szabályozza az életciklusát, valamint a bejövő HTTP kéréseket a megfelelő formában továbbítja a szervletnek, valamint visszaküldi a kliensnek a választ. Egy konténeren belül több szervlet is futhat, ezek jellemzőinek a beállítására egy web.xml nevű konfigurációs leíró szolgál. Itt állíthatjuk be, hogy mely URL címekre melyik szervlet generálja a választ, valamint itt adhatunk kezdeti paramétereket is.

A szervletek képesek a kliens állapotának szerver oldali megőrzésére, így a felhasználókhoz munkamenetet (session) rendelhetünk. Ezt HTTP sütikkel, vagy URL átírással éri el, a fejlesztő felé transzparens módon.

A szervletek mellett felvehetünk szűrőket, melyek a bejövő kérést elkaphatják, valamint a generált választ is utófeldolgozhatják. Ezeket is URL mintára kell felvenni, és a kiszolgáló szervlet előtt fogják megkapni a vezérlést.

Lehetőségünk van bizonyos oldalakat levédeni, melyeket csak bizonyos szereppel rendelkező felhasználók látogathatnak. Ezt is URL mintára vehetjük fel, és beállítható, hogy be nem jelentkezett látogatót mely oldalra irányítson. Amennyiben a kliens már be van jelentkezve, de mégsem rendelkezik a szükséges csoporttagsággal, akkor egy hibaoldalra kerül átirányításra.

JSP

A JSP[10] (JavaServer Pages) egy olyan komponens, amely első eléréskor szervletté fordul, ezáltal a lehetőségei megegyeznek vele. Elsősorban a szerveroldali programkód átláthatóságát hivatott fokozni. Tartalmazhat java kódot, valamint a nem értelmezett szöveget egyszerűen bemásolja a kimenetre. Ezen kívül tartalmazhat scriptleteket,

valamint van saját leíró nyelve. Az előbbi java kód, mely a generált szervletbe belekerül, tehát a válasz előállításakor le fog futni.

A leíró nyelve XML-hez hasonló tagek, melyek előre meghatározott utasításokká fordulnak. Ezek lehetnek direktívák, illetve akcióelemek. Az előbbi segítségével beilleszthetünk egy fájlt az oldal elejére, taglibet tudunk deklarálni, vagy az egész oldalra állíthatunk be jellemzőket. Az akcióelemek segítségével beilleszthetünk weberőforrást, továbbirányíthatjuk az oldalt, valamint java beaneket deklarálhatunk, illetve a tulajdonságait állíthatjuk és lekérhetjük. Ezen beaneknek meg kell adni egy láthatóságot, mely szabályozza, hogy honnan és meddig érhető el. Ez az alábbi értékekből lehet:

- Page: az elem az adott JSP oldalon érhető el kizárólag
- Request: az adott kérésen belül mindenhol elérhető
- Session: az adott felhasználó összes kérésein belül elérhető
- Application: az alkalmazáson belül mindenhol elérhető

Az értelmezett akcióelemek listáját kiegészíthetjük saját megoldásokkal is, erre a taglibek szolgálnak. Egy saját tag definiálásához szükségünk van arra a java kódra, amelyik feldolgozza az adott elemet. Ez használhat paramétereket, valamint gyerekelemeket is. A használni kívánt tageket egy XML leíróban fel kell vennünk, végül a JSP oldalon egy taglib direktívával elérni, hogy a szervletbe bekerüljön. Ezzel a megoldással tetszőlegesen bonyolult viselkedést tudunk leírni egyetlen taggel, mely nagyban segíti az elkészült oldal átláthatóságát.

A JSTL (JavaServer Pages Standard Tag Library)-ben megtalálhatóak a legtöbbször használt tagek. Ennek segítségével tudunk vezérlést vinni JSP oldalba, így használhatunk ciklusokat és elágazásokat is. A könyvtárban találunk még sokszor előforduló problémákra kész megoldást, mint például az XML feldolgozás, vagy a többnyelvűsítés.

Az UEL (Unified Expression Language) lehetővé teszi, hogy az oldalon létező változókhoz egyszerű módon hozzáférhessünk. Ezen változók rendszerint java beanek,

melyek tulajdonságaihoz léteznek lekérdező (getter) függvények, ezért ezeken keresztül tudunk navigálni az objektumok között. Ezen kívül rendelkezik még beépített műveletekkel listák vagy asszociatív tömbökhöz kezeléséhez. A kiértékelése kétféle lehet, mely szintakszisban is különbözik egymástól. Lehet azonnali kiértékelésű, mely esetben az oldal generálásakor hajtódnak végre, mindössze egyszer. Ezen kívül lehet késleltetett a kiértékelés, ekkor a kérés életciklusához alkalmazkodik, valamint a tulajdonságot képes módosítani is.

JSF

A JSF[11] (JavaServer Faces) olyan, a JSP-re épülő, webes keretrendszer, mely leegyszerűsíti a javás webalkalmazások fejlesztését azáltal, hogy beépített megoldást nyújt a leggyakrabban előforduló problémákra. Leírása a JSP-vel megegyező, azt egészíti ki kettő taglibbel, melyek a JSF specifikus elemeket deklarálják.

A nézet egy olyan objektumhierarchia, mely az oldal lekérésekor áll össze az oldalon lévő JSF elemekből. Ez egy fastruktúra, melynek a felépítése megegyezik a JSP oldalon lévő elemekével, valamint a látogató által válaszként visszakapott HTML oldaléval. A generált nézet a szerveroldalon elmentődik, így ha a felhasználó az előzőleg meglátogatott oldalra nyit kérést, például mert elküldött egy űrlapot, akkor az újra előállítás helyett helyreállítja a letárolt példányt.

Fontos része a kérésfeldolgozás folyamata. Ez a következő lépésekből áll:

- Nézet visszaállítása: ekkor visszaállítja a lementett nézetet, ha viszont még nincs lementve, akkor az utolsó fázisra ugrik.
- A kérés paramétereinek rögzítése: ekkor a kérésből kiolvasott paramétereket a nézetben a hozzájuk tartozó objektumokhoz rendeli.
- Validáció: konvertálja és validálja az előzőleg beállított értékeket. Amennyiben ez nem sikerül, akkor hibával az utolsó fázisra ugrik.
- Modell aktualizálása: a konvertált és validált értékek beíródnak az objektumok jellemzőibe.

- Alkalmazás meghívása: minden űrlapelküldéshez tartozik egy akció paraméter, mely rámutat egy metódusra, mely ebben a fázisban hívódik meg.
- Nézet generálása: előállítja a nézetet, majd lementi és előállítja belőle a választ.

Mivel a felhasználótól jövő űrlapadatokban nem biztonságos megbízni, ezért minden bemeneti elemhez felvehetünk validátorokat, melyek nem engednek meg nem megfelelő értéket beírni. Ehhez szükség van egy java osztályra, mely az ellenőrzés folyamatában kerül meghívásra a kapott értékkel. Ezt fel kell vennünk a faces-config.xml konfigurációs állományban, majd az oldalon az elemhez a validátort elhelyezhetjük.

Az oldalak közötti navigáció is a konfigurációban szabályozható. Minden felhasználói művelethez rendelhetünk egy akciót, mely a kérésfeldolgozás közben meg fog hívódni. Ennek a metódusnak a visszatérési értéke lesz az akció kimenetele. A konfigurációban felvehetjük, hogy melyik oldal mely kimenetei esetén mely másik oldalra legyen átirányítva a látogató.

A JSP oldal mögötti adatokat a menedzselt beanek szolgáltatják. Ezek is a megszokott eldobható komponensek, melyek attribútumaihoz köthetünk kimeneti és bemeneti elemeket, valamint metódusaikhoz felhasználói akciókat. Ezáltal egyszerűen lehet kezelni minden interaktivitást, valamint ezen belül elérhetjük az üzleti logikai réteget, ugyanis függőség injektálással tudunk referenciát szerezni session beanekre.

2.1.5 Rétegfüggetlen szolgáltatások

JTA

A Java EE-ben a tranzakciókezelés megvalósítására a JTA (Java Transaction API) ad szabványos megoldást. Legtöbbször az adatbázis-módosításokat akarjuk összevonni, azonban ennél többre is használhatjuk.

Egy eszközkezelő XA kompatibilis, amennyiben képes együttműködni a globális tranzakciókezelővel. Ezen kívül támogatnia kell a két fázisú kommittálást, tehát a kommitpontnál meg kell állnia, és itt még lehetőség van a visszagörgetésre.

A JTA tehát egy tranzakción belül a globális tranzakciókezelő, mely a tranzakcióban részt vevő lokális eszközkezelőket oly módon irányítja, hogy a teljes tranzakció rendelkezzen az ACID (atomicitás, konzisztencia, izoláció, tartósság) tulajdonságokkal. Ennek eléréséhez minden eszközkezelő eljut a kommitpontig, majd amennyiben ez sikeresen megtörtént, csak akkor fognak érvényre jutni. Amennyiben valamelyik eszközkezelő hibát észlel, az összes változtatás visszagörgetésre kerül.

Java EE-vel kliens és szerveroldalon tudunk tranzakciót kezelni. Előbbit csak programozottan, utóbbit azonban deklaratívan annotációkkal avagy XML konfigurációval is megoldhatjuk. Ezen szemlélet lényege, hogy a metódusoknak előírhatjuk, hogyan viselkedjenek tranzakcionálisan. Ennek értelmében az alábbi lehetséges viselkedést adhatunk meg:

- Required: ha van futó tranzakció, akkor csatlakozik, egyébként indít egyet.
- RequiresNew: mindenképpen indít újat, és az összes művelete ehhez lesz kötve.
- Supports: futó tranzakció esetén csatlakozik, egyébként nem tranzakcionáltan fut le.
- Mandatory: amennyiben nincs futó tranzakció, akkor kivétel keletkezik.
- NotSupported: amennyiben van futó tranzakció, akkor is azon kívül fog lefutni.
- Never: amennyiben van futó tranzakció, akkor kivétel keletkezik.

A másik lehetséges mód a tranzakciók kezelésére a UserTransaction interfészen keresztül van. Ennek segítségével tudunk tranzakciót indítani, valamint kommittálni vagy visszagörgetni.

Egyéb rétegfüggetlen szolgáltatás is vannak még, például a biztonsági szabályok betartásáért felelős modul, valamint a JNDI könyvtárszolgáltatást nyújtó.

2.2 Webszolgáltatások

2.2.1 Általános áttekintő

A webszolgáltatás egy általában HTTP[12] protokoll fölött működő kommunikációs szabvány, mely számítógépek közötti együttműködést tesz lehetővé platformfüggetlen módon. Megvalósítása 3 szereplős, de a mediátor opcionális. Először van a szolgáltatás

nyújtó fél, mely távolról hívható műveleteket ajánl ki, és regisztrál egy brókerhez, mely tárolja ezeket. Végül van a szolgáltatást igénybe vevő szereplő, aki ettől a brókertől kéri el a hívható metódusokat, így képes lesz megtalálni a kiajánlott szolgáltatásokat. Amennyiben csak 2 szereplő van, akkor a felhasználónak ismernie kell a szolgáltatások elérhetőségét.

A kommunikációs protokollok XML[13] (eXtensible Markup Language)-re épülnek, mely szöveges volta miatt platformfüggetlen módon képesek működni. Ennek eredményeképpen a kommunikációban nincsen szerepe annak, hogy milyen fizikai számítógép vagy programozási nyelv áll a másik oldalon egészen addig, amíg a szabványt helyesen kezeli. A leíró dokumentumokhoz tartoznak sémák (XSD: XML Schema Definition), melyek az üzenet szintaktikai helyességét biztosítják.

A szolgáltatásleíró dokumentum a WSDL[14] (Web Services Description Language). Ebben találhatóak az elérhető szolgáltatások, valamint az értelmezett üzeneteknek a leírásai, melyekkel a szolgáltatásokat igénybe lehet venni. Webszolgáltatás meghívása SOAP[15] (Simple Object Access Protocol) üzenetekkel történik, mely leírja a kért szolgáltatást, valamint a hívásához szükséges paramétereket. Amennyiben erre a szerver választ kíván küldeni, azt egy hasonló üzenettel tudja megtenni.

A protokoll HTTP fölött általában a 80-as porton kommunikál, ezáltal a legtöbb tűzfalon akadálytalanul jut át. Ez, valamint az implementációfüggetlensége alkalmassá teszi federált IT rendszerekben való használatra, így beleillik a nagyvállalati szabványok körébe.

A WSDL leírót és a SOAP üzeneteket részletesebben bemutatom a következő fejezetben.

2.2.2 Webszolgáltatás protokollok: WSDL, SOAP

WSDL

A WSDL a nyújtott szolgáltatások leírására szolgál, mely alapján igénybe lehet venni azokat. A leírása XML formátumban történik, és kiajánlható egy brókernek, valamint le is kérhetőattől. A WSDL leírónak 5 része van:

- Típusok: egy XSD, mely leírja, hogy az egyes elemeknek milyen típusai értelmezettek, például az üzeneteket. A leíró típusa miatt lehet definiálni összetett típusokat is.
- Üzenetek: leírja, hogy milyen üzenetek értelmezettek. Egy művelet bemenő-, és kimenő üzenetei innen kerülnek ki.
- Port típusok: a port egy interfész, mely hívható szolgáltatásokat fog össze. Ennek lehet 1 vagy több művelete, melyekhez tartozik pontosan egy bemeneti-, és opcionálisan kimeneti üzenet, valamint hibakezelő ág, mely nem várt működés esetén ad visszajelzést a hiba okáról.
- Kötések: a kötések a portokat kötik konkrét protokollokhoz, melyeken a szolgáltatást igénybe vevő fél majd hívni tudja. Ennek segítségével nem csak SOAP-on lesz hívható a szolgáltatásunk, hanem a támogatott többi protokollon is elérhetővé tudjuk tenni.
- Szolgáltatások: végpontokat definiálnak, melyek több portot tartalmazhatnak. A kliensek ezen végpontokat tudják elérni, és a rajtuk értelmezett műveleteket elvégezni.

SOAP

A SOAP egy platformfüggetlen protokoll, melyet használhatunk webszolgáltatások hívásának lebonyolításához. XML alapú, ez biztosítja az implementációfüggetlenséget. Az üzenet paraméterei érték szerint adódnak át. Egy üzenet gyökéreleme a boríték. Ennek van egy fejrésze, amely kiegészítő információkat tartalmazhat az üzenet feldolgozásával kapcsolatban, valamint ki is terjesztheti a protokollt. A boríték másik

eleme a törzsrész, melyben az üzenet konkrét tartalmát, a törzsbejegyzéseket, valamint a hibára utaló hibabejegyzéseket tartalmazza.

2.3. SOA

2.3.1 Áttekintő

A SOA (Service-Oriented Architecture) egy rendszerfejlesztési paradigma, amelyben elemi szolgáltatásokat valamint azokra épülő üzleti folyamatokat definiálunk, és ezek alkotják az folyamatlogikát. A komponensek interfészeken keresztül érik el egymást, melyek mögötti implementáció szabadon lecserélhető, ezáltal az elemek lazán csatoltak. A tényleges alkalmazásimplementáció az elemi szolgáltatásokba csomagolva található elfedve, kifelé csak a szolgáltatásinterfész látható. A folyamatok és a szolgáltatások közötti kommunikáció többféle protokollon keresztül történhet, például webszolgáltatásokon és SOAP-on keresztül. A komponensek belső állapottal rendelkezhetnek, valamint hívás nélkül is futhatnak.

A komponensek összekötését egy ESB (Enterprise Service Bus) végzi, mely biztosítja a kommunikációs protokollok megértését és egymás általi használhatóságát. Tehát ez egy integrációs eszköz, amely az elemeknek egy közös elérést nyújt a többi felé. Ezen keresztül érhetőek el az erőforrások, például az adatbázisok, valamint külső rendszerekhez (B2B: Business to Business) is kapcsolódhat. Valamint ez végzi el a szolgáltatások nyilvántartását, ezáltal tudják megtalálni egymást a komponensek. Ez történhet adaptív módon, szabályrendszer, házirend, vagy tartalom alapon. Egymással inkompatibilis komponensek között segíthet a mediációs szolgáltatás, amely a kommunikációt átfordítja a fogadó oldal által elvárt felépítésűvé.

SOA környezetben fontosak az üzleti folyamatok, melyek elemi szolgáltatásokat felhasználva valósítanak meg komplex logikát. Ezen folyamatok maguk is szolgáltatásként hívhatóak, tehát a többi komponenshez hasonlóak lesznek. Futtatásukat a folyamatszerver végzi, mely legtöbb esetben tartalmazza az ESB-t. Legelterjedtebb szabványa a BPEL (Business Process Execution Language), melyet a következő fejezetben részletesebben bemutatok.

2.3.2 BPEL

A BPEL[16] egy folyamatleíró nyelv, amivel komplex folyamatlogikát valósíthatunk meg egyéb szolgáltatásokat felhasználva. A felépítése XML, mely a hozzá tartozó XSD-vel akár szerkesztés közben is validálható. A folyamat is egy szolgáltatásként fog megjelenni, ezáltal a többihez hasonlóan hívható, és egyszerűen egymásba is ágyazhatóak. Maga a folyamat egy irányított gráffal írható le, melyben egy forrás van, ez a folyamat kiindulópontja. A végrehajtás egy üzenet hatására indul meg, majd a befejeződésekor opcionálisan küldhet vissza válaszüzenetet is. Amennyiben ezt megteszi, akkor kérés-válasz (request-response) típusú, amennyiben pedig nem, akkor egyirányú (one-way) a folyamat.

A folyamatból kifelé menő, vagy befelé jövő üzenetek a folyamattal partner link-en keresztül kapcsolódnak. Ez meghatározza a másik oldal interfészét, ezáltal nem kell ismernünk a konkrét implementációt a tervezés közben, erre a kompozit alkalmazások szolgálnak, amelyek a BPEL folyamatok partnereit összekötik a konkrét implementációval.

A folyamat lépései a taszkok, ezek a típustól függően egy műveletet végeznek el. A következő taszkok értelmezettek:

- Receive: a folyamat elindulása a hívó üzenet beérkezésével.
- Reply: a folyamat válasza.
- Invoke: egy partnerszolgáltatás meghívása.
- Assign: változókhoz értékek rendelése.
- Wait: várakozás egy bizonyos ideig.
- Throw/Rethrow: kivétel dobása/továbbdobása.
- Exit: kilépés a folyamatból.
- Validate: validál változókat.
- Javascript: dinamikus kód hívása.

A taszkokon kívül strukturált futás meghatározására is van lehetőségünk. Erre az alábbi eszközöket használhatjuk:

- If: elágazás, egy feltételtől függ, hogy melyik ágon folyik tovább a feldolgozás.
- While/RepeatUntil: addig fut a a benne levő rész, amíg a feltétel igaz/hamis.
- ForEach: struktúrán iterál végig.
- Pick: egy üzenet megérkezésére vár.
- Flow/Sequence: párhuzamosan/sorosan végzi a benne levő taszkokat.

A folyamat lehet atomi vagy hosszan futó. Előbbi esetben az összes lépése egy tranzakcióban fut, mely biztosítja az atomiságot, és a módosítások visszagörgetését hiba esetén, ugyanakkor fenntart bizonyos zárakat, melyek más folyamatokat várakozásra kényszeríthetik. Ezért a hosszú ideig tartó folyamatok nem tranzakcionáltak, ezáltal nem tudnak várakoztatni. Ebben az esetben azonban nem lehet a visszagörgetést automatizáltan megoldani. Erre való a kompenzáció, melyet taszkok egy összefüggő halmazához vehetünk fel, egy kompenzációs kezelőt felvéve hozzájuk. Ez a kezelő meghívásra kerül, amennyiben nem kezelt kivétel történik, vagy explicit meghívjuk. Létezik kivételkezelés is, mely hasonló módon működik, azonban a céljuk alapvetően más. A kivételkezelés a nem elvárt működés esetén próbálja a hibát lekezelni, hogy a folyamat futhasson tovább, azonban a kompenzáció a változtatások visszagörgetésére szolgál elsősorban, mivel ezt a tranzakciókezelő ebben az esetben nem tudja megtenni magától.

Lehetőségünk van futó folyamatnak üzenetet küldeni, amivel befolyásolhatjuk a futását. Ennek az elérésére szolgál a korreláció, mely a folyamat bemeneti paramétereinek egy részhalmaza alapján a bejövő üzenetet hozzá tudja kötni egy már futó, azonos korrelált paraméterekkel meghívott folyamathoz. A futó folyamat a *Pick* struktúra segítségével tudja ezen üzeneteket fogadni és feldolgozni, valamint a *Reply* taszkkal tud rá válaszolni is, így küldhet az állapotáról visszaigazolást.

A folyamat során felvehetünk változókat, melyeket használat előtt inicializálni kell. A típusuk a folyamathoz felvett XSD-k által meghatározott típusok egyikének kell lennie. A változók XML-ben leírt adatstruktúrák, és felvehetünk hozzájuk predikátumokat,

melyek a változóra értelmezett XPath (XML Path Language) kifejezések, melyek maguk is változók, értékékük pedig a kifejezés kiértékelésének az eredménye. Lehetőségünk van láthatósági határokat megadni, melyek tartalmazhatnak taszkokat, valamint változókat és kezelőket is, ezáltal strukturálhatjuk a hibakezelést és a változók életciklusát. Ezek egymásba is ágyazhatóak.

A kész folyamatok, mint webszolgáltatások leírását a folyamathoz mellékelt WSDL dokumentummal tudjuk leírni.

3 A feladat megvalósítása

3.1 Specifikáció

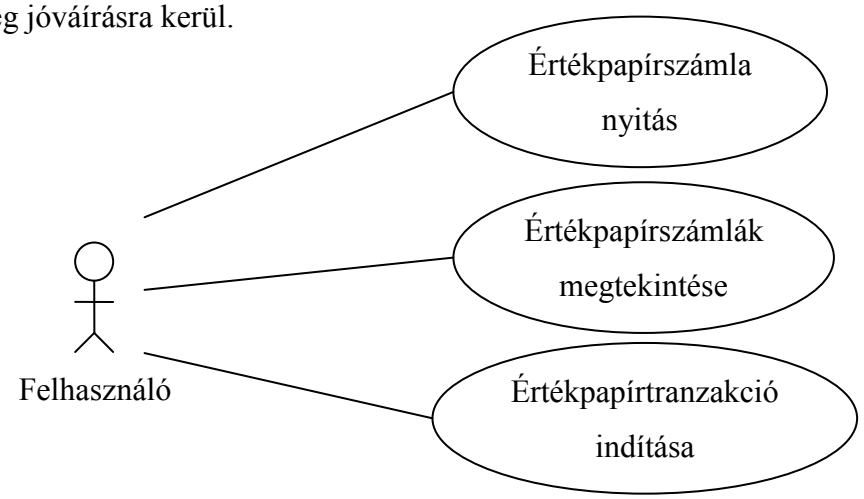
3.1.1 Befektetések modul

A befektetési modul a felhasználók értékpapírszámláinak kezelésére szolgál. Minden ilyen számla mögötti pénzmennyiséget pontosan egy pénzalap számla biztosítja. Értékpapírszámlákat a felhasználók szabadon létrehozhatnak, melynek során ki kell választaniuk az egyik pénzszámlájukat, valamint a számla típusát. Ezen típustól függően csak meghatározott instrumentumok halmazával kereskedhetnek ezen a számlán. A számlák a típusuktól függően rendelkeznek pénznemmel is, mely a kompatibilis instrumentumok árának a mértékegységét jelenti. A pénzalap is rendelkezik ilyen pénznemmel, mely el is térhet az értékpapírszámláétól, ezért a pénzmozgásoknál konverziót kell végrehajtani.

Az értékpapírszámlák segítségével a felhasználók értékpapírokkal kereskedhetnek. Ennek során tranzakciókat indítanak, melyeket egy külső rendszer fogja elbírálni és visszaigazolni, majd ezt az ügyfél a webes felületen megtekintheti. Tranzakció indítása során meg kell adnia a felhasználónak, hogy mely számlához köti a tranzakciót, azon belül melyik, a számlával kompatibilis, instrumentummal kíván kereskedni, hány darabot szeretne venni, milyen áron, valamint milyen típusú legyen a tranzakció. Ezen típus a külső rendszernek ad irányelvet, hogy a megadotthoz képest mely árakon elfogadható a kötés. A tranzakció feladásakor az ellenérték a pénzszámlán zárolva lesz, majd a külső rendszerhez kerül. Ez visszatérhet időtúllépéssel, vagy pedig a tényleges árral, melyen a vétel megtörtént. Ezt a tényleges ellenértéket a pénzalapból le kell vonni, és az értékpapírszámlán jóváírható a vett instrumentum.

Háromféle instrumentumtípus van. Mindegyiket a fent ismertetett módon lehet venni és eladni, különbség a tartásuk során van. A rendes értékpapír, mely egyik másik csoportba sem tartozik, birtoklása semmilyen következménnyel sem jár. Vannak a határidős derivatívák, melyekkel szabadon lehet kereskedni, azonban egy konkrét jövőbeli időpontban lejárnak, ekkor pedig az alaptermék aktuális árával automatikusan el lesznek

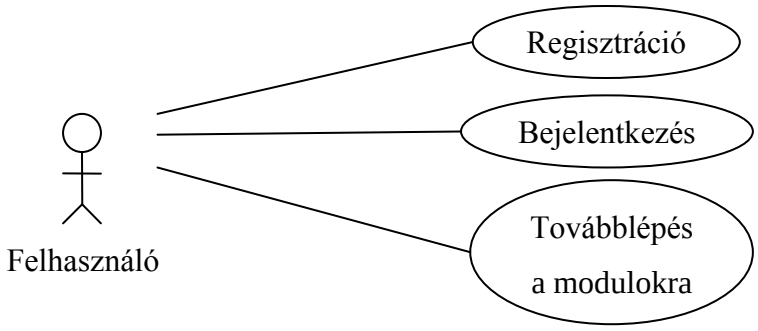
adva. Végül vannak az államkötvények, melyeknek van lejárati idejük, ismétlődési periódusuk, kamatuk, valamint névértékük. Tartásuk során minden olyan időpontban, melytől a lejárati idő a periódusidő egész számú többszörösére van, kamatot fizetnek. Ennek mennyisége a névérték és a kamat szorzata, instrumentumonként. Végül a lejárati idejükkor az értékpapírszámláról levonódnak, ugyanakkor a névértéküknek megfelelő pénzmennyiség jóváírásra kerül.



Kép 1: A befektetések modul use case-e

3.1.2 Közös bejelentkező modul

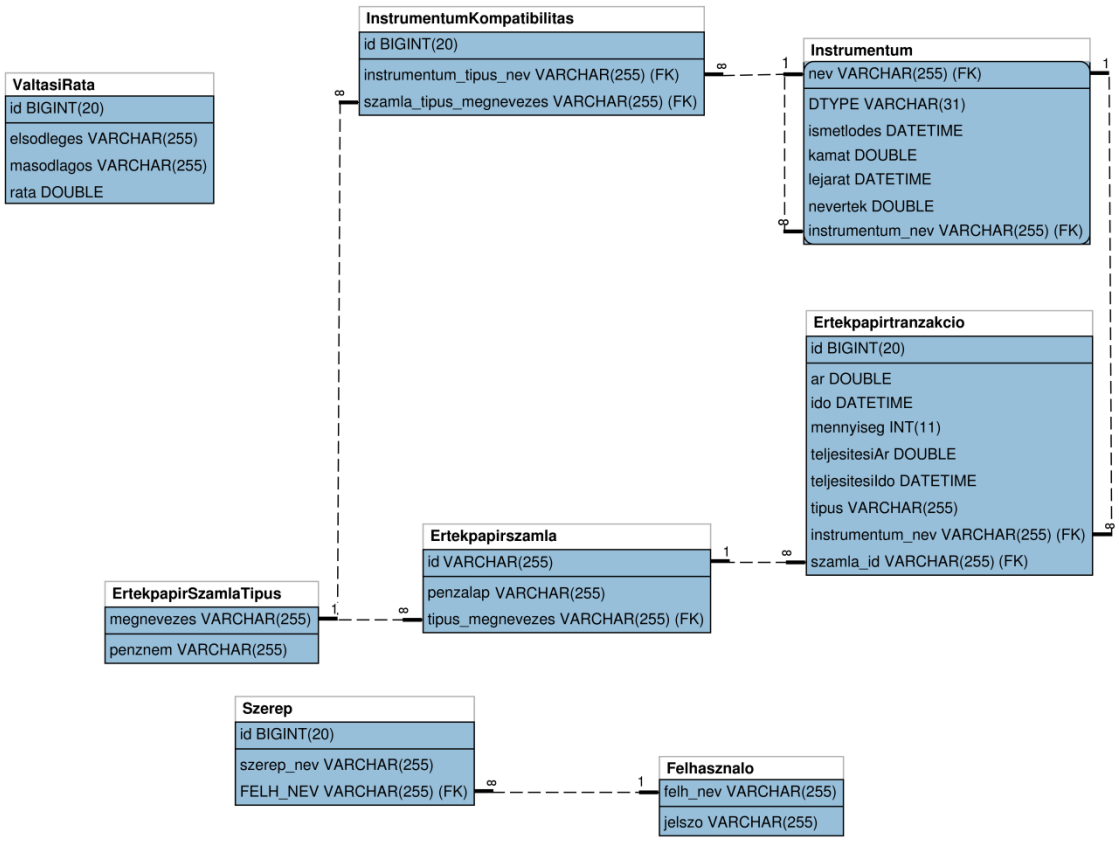
A banki rendszerben a befektetések modul mellett van egy folyószámlakezelő is, melyet nem én fejlesztettem. Az ezek közötti integráció egyik lépése volt, hogy össze kellett hangolni a felhasználókezelést, hogy a bejelentkezési információk csak egyetlen helyen tárolódjanak. Ennek a megvalósítása egy harmadik modulba, egy közös felhasználókezelőbe került. Az azonos rendszerhez tartozó különböző alrendszerek esetén jogos az a felhasználói igény, hogy ezek használatához elég legyen egyszer bejelentkezni. Ezért szükséges egy központi felhasználókezelői modul, mely segítségével új felhasználók regisztrálhatnak, valamint bejelentkezhetnek a banki rendszerbe. Az autentikált felhasználók ezután eldönthetik, hogy melyik tényleges modult szeretnék használni, természetesen ide visszalépve át tudnak lépni a másikba is. Ezen bejelentkezési megoldásra a többi alrendszert is fel kell készíteni, mivel ha nincs bejelentkezve a látogató, akkor át kell irányítaniuk erre az oldalra.



Kép 2: A közös bejelentkező modul use case-e

3.2 Tervezés

3.2.1 Adatréteg tervezése



Kép 3: Az adatréteg felépítése

Az értékpapírszámlákhoz tartozik értékpapírszámla, valamint van típusa és egyedi azonosítója. A számlatípus a típusok neveihez tartalmazza a pénznemüket. Az instrumentumok kompatibilitását tartalmazó tábla a számlatípusokhoz rendeli az instrumentumtípusokat. Az instrumentumok típusát (hagyományos, határidős derivatíva vagy államkötvény) a DTYPE diszkriminátor oszlop tárolja. Ezen táblán belül neve mindnek van, lejárata a határidősnek és az államkötvénynek, alap instrumentuma csak a derivatívának, ismétlődése, kamata, lejárata és névértéke pedig csak az államkötvénynek. Ezek alapján ez az öröklési hierarchia egyetlen táblába van leképezve. Az értékpapír-tranzakciónak van azonosítója, ára, feladási ideje, mennyisége, típusa, meghatároz egy instrumentumot, valamint kötve van egy értékpapírszámlához. Ezeken kívül, amennyiben már teljesítődött, akkor van teljesítési ideje, valamint teljesítési ára. Azt, hogy melyik pénznem milyen másikra váltható, valamint az milyen váltási rátával tehető meg, a ValtasiRata táblába kerül.

A bejelentkező modulhoz tartozik a Felhasznalo és a Szerep tábla. Az előbbi a rendszerben létező felhasználókat kezeli nevükkel és jelszavukkal, az utóbbi pedig a felhasználókhoz rendel szerepeket, akár többet is.

3.2.2 Üzleti logika tervezése

Az üzleti logika réteg fő építőelemei a session beanek, ezek halmazából épül fel a hívható alkalmazáslogika. A funkcionalitás nem kívánja meg állapottal rendelkező bean használatát, ezért kizárólag állapotmenteset használok. Ugyan a szükséges metódusokat akár egyetlen osztályban is fel lehetne sorolni, ez jelentősen rontaná az áttekinthetőséget, ezért feladatkör szerint 4 beanre van szükség:

- ManagerBean: ez felel a felhasználók kezeléséért, valamint a határidős és az állampapírok felügyeletét időről időre ellátó időzítő elindításáért, és kezeléséért.
- PénzváltásBean: ez a bean felel a pénzkonverzióhoz szükséges ráta megállapításáért.
- ÉrtékpapírszámlaBean: ez a bean felel az értékpapírszámlákkal kapcsolatos összes műveletért. Ez alapján az alábbi műveleteket lehet elvégezni vele:
    - Értékpapírszámlát lehet nyitni.
    - Le lehet kérdezni, hogy létezik-e számla adott pénzalappal.

- Meg lehet tudni, hogy az adott instrumentum kompatibilis-e az adott típusú számlával.
- Le lehet kérni adott értékpapírszámla típusát.
- Le lehet kérni, hogy az adott államkötvény le van-e járva.
- Vissza tudja adni az államkötvény kamatát.
- Le lehet kérni az adott számlához tartozó pénzalap számlaszámát.
- Meg lehet tudni, hogy az adott számlán adott típusú instrumentumból hány darab van.
- Le lehet kérni az összes periódusnapon lévő államkötvényt, a hozzájuk tartozó számla számát, valamint a darabszámukat.
- Le lehet kérni az összes lejárt határidős instrumentumot, a hozzájuk tartozó számlával és darabszámukkal.
- Vissza tudja adni az összes értékpapírszámlát.
- Vissza tudja adni az összes pénzszámlát.
- Vissza tudja adni az összes számlatípust.

- ÉrtékpapírTranzakcióBean: ez a bean kezeli a tranzakciókat, tehát el tudja indítani a vásárlás folyamatát, valamint az adatbázisba tudja írni a tranzakció kezdetét, valamint a befejeződését.

3.2.3 Üzleti folyamatok tervezése

A specifikáció alapján 3 üzleti folyamatot határoztam meg. Ezek az értékpapír vásárlás, az államkötvények perióduskezelése, valamint a határidős derivatívák lejáratának a kezelése. Mindegyik folyamat egyirányú, tehát csak bemeneti paraméterei lehetnek, kimeneti nem. A megtervezett folyamatokról készült képek megtalálhatóak a függelékben.

3.2.3.1 Az értékpapírok vásárlásának a folyamata

A folyamat[Függelék B] bemenő paraméterként várja a megadott árat, a számla számát, a vételi mennyiséget, a tranzakció típusát, valamint a vásárolni kívánt instrumentum megnevezését. Első lépésként meg kell határozni azt a pénzmennyiséget, amelyet a pénzalapon zárolni kell. Ehhez szükséges meghatározni a pénzalap és a számla

pénznemét, majd le kell kérni az ezek közötti aktuális váltási rátát. Szükséges ellenőrízni, hogy a megvenni kívánt instrumentum kompatibilis-e a számlával. Amennyiben nem kompatibilis, vagy a zárolás sikertelen, akkor a tranzakciót megszakítjuk. Amennyiben minden rendben ment eddig, akkor egy hívással az adatbázisba írjuk ezt, hogy a felhasználó is lássa a tranzakció elindulását. Ezután következhet a külső rendszer hívása, mely idővel visszatér a tényleges vételi árral, vagy -1–el, ha sikertelen. Előbbi esetben a zárolt pénzmennyiséget felszabadítjuk, majd a tényleges vételi árral számolt mennyiséget levonjuk a pénzalapból, valamint lezárjuk a tranzakciót a kapott adatokkal, ezáltal a felhasználót tudatjuk a sikerességről. Amennyiben sikertelen az értékpapír vétele, akkor is felszabadítjuk a zárolt pénzt, és töröljük a nyitott tranzakciót.

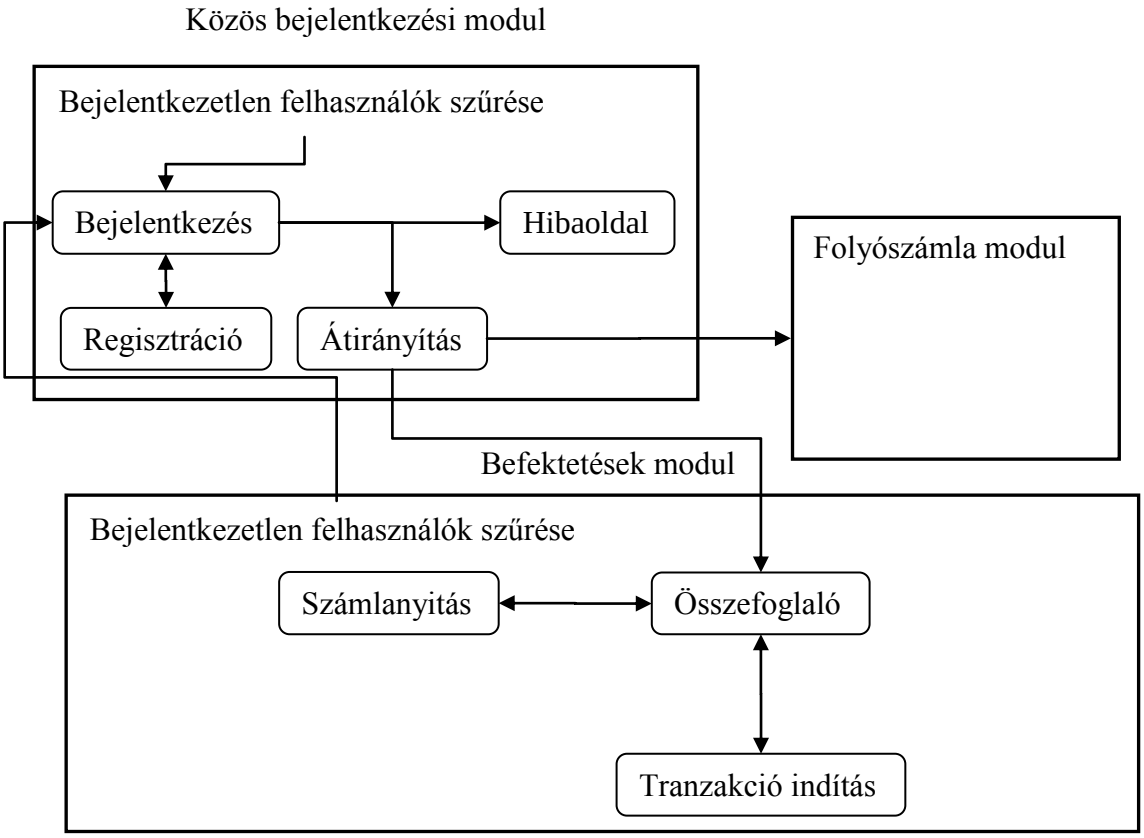
3.2.3.2 Az államkötvények periódusának kezelése

Ez a folyamat[Függelék C] automatikusan hívódik minden nap, nincs bemeneti paramétere. Első lépésként lekérjük az összes államkötvényt, a hozzájuk tartozó számlát, és a mennyiségét, amelynek most van a periódusa. Ezen hármasokon egyesével végigiterálunk, és mindegyikre ellenőrizzük, hogy lejártak-e. Amennyiben igen, akkor az ügyfélnek jóvá kell írnunk a névértéket, ehhez indítunk és befejezünk egy eladási tranzakciót, melyben a teljes mennyiséget sikeresen eladjuk a névértékért. Ha az adott államkötvény még nem járt le, akkor csak kamatot fizetünk, ehhez meg kell tudnunk a számla és a pénzalap pénznemét, ebből pedig a váltási rátát, majd a kamattal szorzott névértéket minden egyes instrumentumra jóváírjuk a pénzalapon.

3.2.3.3 A határidős derivatívák lejáratának kezelése

Az államkötvényekhez hasonlóan ez a folyamat[Függelék A] is minden nap ellenőrződik, és bemeneti paramétere sincs. Az első lépés a lejárt határidősek, a hozzájuk tartozó számla, valamint a darabszámuk lekérése, majd ezen hármasokon való végigiterálás. Ezután a derivatívához tartozó instrumentum aktuális árának a meghatározása jön, melyhez szükséges egy külső rendszer igénybevétele. Ezután indítunk és befejezünk egy olyan tranzakciót, mely az összes birtokolt terméket eladja a meghatározott aktuális áron.

3.2.4 A webes megjelenítési réteg tervezése



Kép 4: Az oldalak és a navigáció felépítése

A látogató először a bejelentkező modulhoz jut, ahol a felhasználónevét és jelszavát megadva be tud lépni a rendszerbe. Amennyiben ezt rosszul teszi meg, egy hibaoldalra kerül, ahonnan visszalépve újra próbálkozhat. Ha új felhasználót akar létrehozni, azt egy regisztrációs oldalon lévő űrlapot kitöltve teheti meg, mely elküldése után a főoldalra kerül, ahol a megadott adatokkal be tud lépni. Ha sikeresen azonosította magát a rendszernek, egy átirányító oldalra kerül, ahol kiválaszthatja, hogy a folyószámla, vagy a befektetések modulba akar továbblépni. A bejelentkező modul oldalaihoz bejövő kérések először egy szűrőhöz kerülnek, amely a még be nem jelentkezett felhasználókat nem engedi az átirányító, a bejelentkezetteket pedig a többi oldalra belépni.

A befektetések modul főoldala egy összesítő oldal, ahol a felhasználó egy táblázatban látja az értékpapírszámláit, azok pénzalapján lévő pénzmennyiséget, az értékpapírjait,

valamint a számlához tartozó tranzakciókat. Innen továbbléphet a számlanyitó oldalra, ahol egy űrlapot kitöltve létrehozhat új értékpapírszámát, majd visszakerül az összefoglalóra. Indíthat új tranzakciót is, melynek során kiválasztja a számlát és az instrumentumot, megadja a mennyiséget, az árat, valamint a tranzakció típusát, majd visszakerül az összefoglaló oldalra, ahol figyelemmel tudja kísérni a tranzakció életútját.

3.3 Implementáció

3.3.1 Az adatréteg implementációja

3.3.1.1 Összetett adatlekérdezések

A JPA lehetőséget ad az entitásokhoz nevesített JPQL kifejezések rendelésére, melyekkel összetett lekérdezéseket tudunk megfogalmazni az adatok elérésére. Habár akármelyik osztályhoz felvehetjük bármelyiket, célszerű az érintett entitáshoz rendelni őket, valamint az elnevezésekbe is belevonni a helyüket. Az implementáció során az alábbi nevesített lekérdezéseket fogalmaztam meg:

- *Ertekpapirszamla.getAllForPenzszamla*: *select e from Ertekpapirszamla e where penzalap = :szamlaszam*
  az adott pénzszámlához tartozó értékpapírszámlák listáját kérdezi le.
- *Ertekpapirszamla.getAll*: *select e from Ertekpapirszamla e*
  az összes értékpapírszámlát kéri le.
- *ErtekpapirSzamlaTipus.getAllSzamlatipus*: az összes értelmezett számlatípust kéri le.
- *Ertekpapirtranzakcio.getAllForErtekpapirszamla*: *select e from ErtekpapirSzamlaTipus e*
  az adott értékpapírszámlához tartozó összes értékpapírtranzakciót kéri le.
- *InstrumentumKompatibilitas.getKompatibilitas*: *select e from InstrumentumKompatibilitas e where szamla_tipus.megnevezes = :szamlatipus and instrumentum_tipus.nev= :instrumentum_tipus*

lekéri az adott számlatípus és instrumentumhoz tartozó bejegyzést, tehát amennyiben kompatibilisek, akkor ad vissza eredményt, ellenkező esetben nem.

- *InstrumentumKompatibilitas.getAllKompatibilisInstrumentum*: *select e.instrumentum_tipus.nev from InstrumentumKompatibilitas e where szamla_tipus.megnevezes = :szamlatipus*
  lekéri az adott értékpapírszámlatípussal kompatibilis instrumentumokat.
- *ValtasiRata.getRata*: *select e from ValtasiRata e where elsodleges = :mit and masodlagos= :mire*
  lekéri a megadott forrás-, és célvaluta közötti váltási rátát.

3.3.2 Az üzleti logika réteg implementációja

A létrehozott session beanek és metódusaik:

- *ErtekpapirszamlaBean*:
  - *void ertekpapirszamlaNyitas(String, String)*: Adott pénzalappal adott típusú értékpapírszámlát hoz létre.
  - *void ertekpapirszamlaTorles(long)*: Kitörli a megadott azonosítóval rendelkező értékpapírszámlát.
  - *boolean vanErtekpapirszamlaPenzszamlahoz(String)*: Igaz, ha az adott pénzszámlához létezik értékpapírszámla.
  - *boolean isKompatibilis(String, String)*: Igaz, ha az adott típusú értékpapírszámla kompatibilis az adott instrumentummal.
  - *String getErtekpapirszamlaTipus(long)*: Visszaadja az adott azonosítóval rendelkező értékpapírszámla típusát.
  - *List getAllAllamkotveny()*: Visszaadja az összes államkötvényt.
  - *boolean kamatnapE(String)*: Igaz, ha az adott típusú államkötvénynek kamatnapja van.
  - *boolean lejartE(String)*: Igaz, ha az adott instrumentum lejárt.
  - *double getKamat(String, int)*: Visszaadja az adott típusú és mennyiségű államkötvény kamatát.
  - *String getErtekpapirszamlaPenznem(long)*: Visszaadja az értékpapírszámla pénznemét.

- *String getPenzalapSzamlaszam(long)*: Visszaadja az értékpapírszámlához tartozó pénzalap azonosítóját.
- *int getErtekpapirMennyiseg(long, String)*: Visszaadja, hogy az adott értékpapírszámlán az adott típusú instrumentumból hány darab van.
- *List getAllAllamkotvenyAdatForThisKamatnap(int)*: Visszaadja a kamatozó államkötvényeket, a számlaszámukkal és mennyiségükkel egy listában.
- *List getAllLejartHataridos(int)*: Visszaadja a lejárt határidős derivatívákat számlaszámukkal és mennyiségükkel egy listában.
- *List getAllErtekpapirszamla(String)*: Visszaadja az adott felhasználó összes értékpapírszámláját.
- *List getSzamlaTipusok()*: Visszaadja az összes számlatípust.
- *List getAllPenszamla()*: Visszaadja a bejelentkezett felhasználó összes pénzszámláját.
- *List getKompatibilisInstrumentumok(String)*: Visszaadja az adott típusú értékpapírszámlával kompatibilis összes instrumentumot.

- *ErtekpapirTranzakcioBean*:
  - *long createErtekpapirTranzakcio(double, String, int, long, String)*: Létrehoz egy értékpapírtranzakciót, melyhez meg kell adni az árat, az instrumentum típusát, a mennyiséget, a számlaszámot, valamint a tranzakció típusát, és visszaadja a tranzakció azonosítóját.
  - *void deleteErtekpapirTranzakcio(long)*: Letörli az adott azonosítóval rendelkező tranzakciót.
  - *void finishErtekpapirTranzakcio(double, long)*: Befejezi az adott tranzakciót az adott áron.
  - *void startEPVasarlas(double, String, int, long, String)*: Elindítja az értékpapír vásárlásának folyamatát.
- *ManagerBean*:
  - *void createUser(String, String)*: Létrehoz egy felhasználót adott néven, adott jelszóval.
  - *void handleTimeout(Timer)*: Kezeli az államkötvények és határidős derivatívák periódusait ellenőrző folyamatokat.

- *boolean isUserExists(String)*: Igaz, ha adott nevű felhasználó már létezik.
- *void startTimer()*: Elindítja az időzítőt.

- *PenzValtasBean*:
  - *double getRata(String, String, double)*: Visszaadja az árfolyamkonverzió eredményét, ahol adott a forrás-, és a célpénznem, és a pénz mennyisége.

3.3.2.1 Webszolgáltatások fejlesztése

A webszolgáltatások fejlesztése alapvetően kétféle megközelítésben történhet. Először fentről lefele, melynek során először a WSDL leírót kell megvalósítani, majd ebből célprogrammal generálni az üres függvényeket a használt programozási nyelvre, ezután pedig kitölteni a metódustörzseket. Másik hozzáállás lentről felfele irányú. Ennek során a kész függvényeket ajánljuk ki mint webszolgáltatások, majd egy másik célprogram generálja a WSDL leírót. Én az utóbbi utat választottam, tehát az elérhető szolgáltatásokat *@WebService* annotációval láttam el, így telepítés után az alkalmazásszerveren elérhetővé vált a kész leíró, valamint a típusokat definiáló XSD dokumentum is.

3.3.3 A webréteg implementálása

Az alábbi weberőforrásokat hoztam létre a fejlesztés során a megvalósított modulokba:

- SSO modul:
  - Menedzselt beanek:
    - *RegistrationBean*: Kezeli a kijelentkezést és a regisztrációt.
  - Szűrők:
    - *AuthFilter*: A felhasználó bejelentkezési állapotától függően átirányítja bizonyos oldalakról.
  - JSP oldalak:
    - *Login.jsp*: A bejelentkező oldal, valamint innen juthat tovább a felhasználó a regisztrációhoz.

        - *Error.jsp*: Sikertelen bejelentkezés esetén ez az oldal közli ezt a felhasználóval.
        - *Redirector.jsp*: A többi modulba léphet tovább a felhasználó.
        - *Registration.jsp*: A regisztrációs űrlap található itt.
- Befektetések modul:
    - Menedzselt beanek:
        - *ErtekpapirszamlaBean*: Az értékpapírszámlákat kezelő oldalakhoz biztosítja az adatokat és műveleteket.
        - *ErtekpapirTranzakcioBean*: A tranzakciókat kezelő oldalakhoz biztosítja az adatokat és műveleteket.
    - Szűrők:
        - *RedirectorFilter*: Amennyiben nincs bejelentkezve, akkor a bejelentkező modulba irányítja a felhasználót.
    - Életciklus figyelők:
        - *ServletContextListener*: Elindítja az időzítőt induláskor.
    - Szervletek:
        - *InstrumentumKompatibilitasServlet*: Lekérhető tőle az adott értékpapírszámlával kompatibilis instrumentumok listája.
    - JSP oldalak:
        - *Summary.jsp*: Az összesítő oldal, ahol a felhasználó láthatja az összes számláját és a hozzájuk tartozó tranzakciókat.
        - *Szamlanyitas.jsp*: A számlanyitó oldal.
        - *Tranzakciokezdes.jsp*: Tranzakciót lehet innen elindítani.

3.3.3.1 Közös bejelentkezés megoldása

Minden Java EE-s alkalmazás saját maga kezeli a bejelentkezéseket, ezáltal minden felhasználónak külön-külön be kell lépnie minden modulba. Ez azonban kényelmetlen egy szolgáltatáson belül, ezért a felhasználókezelés egy közös bejelentkezési modulba került, míg a folyószámla, valamint a befektetések modulba ezért felelős programrészek nincsenek.

Az egyszeri bejelentkezésre a Glassfish lehetőséget biztosít. Ehhez az alkalmazásokat közös virtuális szerverbe kell helyezni, be kell kapcsolni az SSO-t, valamint be kell állítani az *sso-max-inactive-seconds*, és az *sso-reap-interval-seconds* paramétereket. Fontos még felvenni az összes web modul konfigurációs leírójába egy *login-config* bejegyzést, különben az a modul kimarad a központi felhasználókezelésből.

Ezek eredményeképpen amikor a felhasználó belép a közös bejelentkezési modulban, akkor automatikusan jelentkezik az azonos virtuális szerveren lévő többi modulba is.

3.3.3.2 Bejelentkezés és biztonság

A felhasználókezelés a JAAS (Java Authentication and Authorization Service)-re épül, mely ez esetben egy JDBCLoginModulon keresztül azonosítja a felhasználókat és rendel hozzájuk csoportokat az adatbázisból. Mivel a nevükön kívül nincsenek megkülönböztetve egymástól a felhasználók, ezért mindegyikükhöz egy ‘client’ szerepet rendeltem, ezzel együtt a védett erőforrásokhoz mindannyian hozzáférhetnek. A biztonság kérdését érdemes két részben tárgyalni. Egyrészt a web rétegben elérhetetlenné kell tenni azon oldalakat, amelyek csak bejelentkezés után elérhetőek. Azonban a rétegek elválaszthatóak egymástól, az üzleti logika rétegben is ellenőrizni kell a hozzáférési kérelmeket. A bejelentkezés a JAAS űrlap alapú megvalósításával történik.

A web rétegben a JAAS is kínál megoldást az oldalak védelmére, ugyanakkor kevésbé testreszabható az, hogy hova legyen irányítva a felhasználó, ha esetleg nincs bejelentkezve, vagy hogy megtekintheti-e a bejelentkező oldalt a már autentikált látogató. Ezért a felhasználói kényelmet figyelembe véve ezt egy szűrővel oldottam meg, mely a bejelentkező-, a hiba-, valamint a regisztrációs oldalról az átirányítóra dobja a bejelentkezett felhasználót, az utóbbiról pedig a bejelentkezőre az ismeretlen látogatót. Ezzel sikerült elkerülnöm, hogy esetleg többször is be lehessen jelentkezni, valamint hogy illetéktelenek megtekinthessék a védett oldalt. A befektetések modul összes oldalára egy másik szűrőt alkalmaztam, mely minden azonosítatlan látogatót a bejelentkező modul bejelentkezési oldalára irányít.

Az üzleti logika réteg védelme a metódusok elérésekor történő ellenőrzésből áll. Az RBAC-nak megfelelően minden elérhető metódus csak a ‘client’ szereppel rendelkező felhasználók érhetnek el, amennyiben ezzel nem rendelkeznek, akkor egy kivétel kíséretében a hívás meghiúsul.

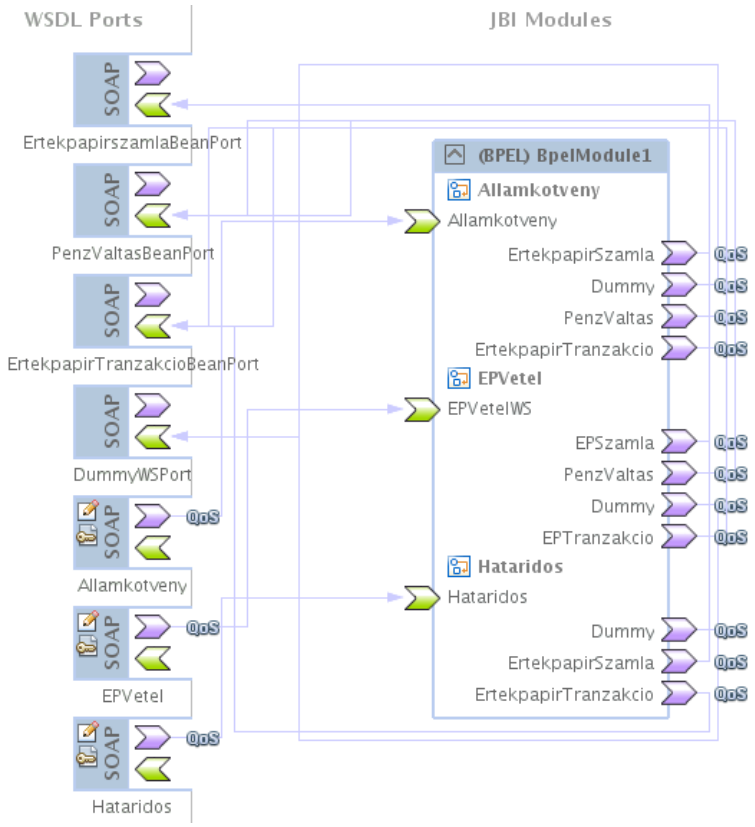
3.3.4 Az üzleti folyamat réteg implementálása

3.3.4.1 Iteráció megvalósítása BPEL-ben

Több folyamatomban is szükséges volt egy visszakapott lista minden elemén egyesével végighaladni és rajtuk műveleteket végezni. Ennek elérésére egy ciklusváltozót és egy *while* ciklust használtam. A BPEL adatstruktúrái XML alapúak, és felvehetőek hozzájuk XPath predikátumok. Ezt felhasználva le lehet kérni az adott adatstruktúra meghatározott pozícióján álló elemet. Ezzel már egyszerűen lehet iterálni, a lekért pozíció a ciklusváltozó aktuális értéke, melyet minden elem feldolgozása után növelünk, egészen addig, amíg az utolsót is bejártuk. Fontos megjegyezni, hogy az XPath predikátum miatt az első elem sorszáma 1.

3.3.4.2 A kompozíciós alkalmazás

A BPEL folyamatok a többi komponenst partner kapcsolatokon keresztül érik el, ezek azonban csak egy interfészt határoznak meg, konkrét implementációt nem. A kompozíciós alkalmazás feladata az, hogy ezeket a végpontokat és az implementációkat összekösse, valamint a kívülről elérhető pontokat kivezesse egy kifelé mutató nézetbe.



Kép 5: A kompozíciós alkalmazás

Az alkalmazás 4 kimenő SOAP protokollt használó portból áll, ezek mind a Java EE-s implementációval kötik össze. Ezek sorban:

- ErtekpapirszamlaBeanPort: az értékpapírszámlával kapcsolatos műveleteket és adatokat szolgáltatja.
- PenzValtasBeanPort: az aktuális árfolyamadatokat lehet lekérdezni.
- ErtekpapirTranzakcioBeanPort: tranzakciókat lehet indítani, befejezni, valamint törölni vele.
- DummyWSPort: a külső rendszerekhez biztosít látszólagos (dummy) implementációt.

Az alkalmazás 3 portot tesz elérhetővé, melyek ugyancsak SOAP segítségével hívhatóak. Ezek a következők:

- Allamkotveny: az államkötvények periódusát és lejáratát kezelő folyamat indítása hívható rajta keresztül.
- EPVetel: az értékpapírok vásárlásának a folyamata indítható el.
- Hataridos: a határidős derivatívák lejáratkezelő folyamata indítható el.

3.3.5 Fejlesztőkörnyezet

Az alkalmazás futtatásának legfőbb követelménye egy alkalmazásszerver. Erre a feladatra Glassfish[17] ESB-t használtam, mely nem csak egy Java EE alkalmazásokat futtatni képes konténer, de a BPEL folyamatokat is kezeli. Így egyetlen program szükséges csak az üzleti logika és a megjelenítési réteg futtathatóságához.

Az adatbázis a nyílt forráskódú MySQL relációs adatbáziskezelő rendszer, mely jól karbantartott meghajtóprogrammal rendelkezik az alkalmazásszerverhez. Fejlesztéshez Eclipse és Netbeans[18] IDE-t használtam, előbbit a Java EE programrészek írására, utóbbit a beépített folyamatszerkesztője miatt.

Segédprogramként phpmyadmin-t, valamint a hozzá szükséges Apache webszervert használtam az adatbázis grafikus szerkesztéséhez. Természetesen szükséges volt egy webböngésző, ami esetemben a Firefox.

A folyószámla modulból érkező WSDL-ekből a java osztályok generálására a Java SDK-ban is megtalálható wsimport-ot használtam.

Mindezek alatti operációs rendszer egy Ubuntu GNU/Linux, melyet egy Sun xVM VirtualBox virtualizációs alkalmazásban futtattam.

3.4 A teszteléshez bevezetett változtatások

Az implementációba több helyen is kénytelen voltam változtatásokat bevezetni annak érdekében, hogy az elkészített program önállóan is működni tudjon, valamint hogy a futása praktikusan vizsgálható legyen. Az alábbi fejezetekben ezeket ismertetem részletesen.

3.4.1 Külső rendszer megvalósítása

Valós környezetben az értékpapír vásárlások egy brókercéghez futnának be, mely a tőzsdén kereskedve beszerezné a kívánt instrumentumokat, majd erről visszaigazolást küldene. Mivel a szakdolgozatom témáján kívülre mutat egy ilyen rendszer kifejlesztése, ezért ezt egy látszólagos megvalósítással helyettesítettem. Ennek megfelelően létrehoztam egy DummyWS-t, mely olyan webszolgáltatásokat ajánl ki, melyeket a külső rendszer is megtenne, és ez tartalmazza az implementációt is. Két ilyen metódusra volt szükségem.

Egyrészt az értékpapírok megvételéhez szükséges egy olyan szolgáltatás, mely megkapja az instrumentum típusát, a feladási árat, a tranzakció típusát, valamint a vételi mennyiséget, majd a teljesítéskor visszatér a tényleges kötési árral. A saját megvalósítás annyit csinál, hogy néhány másodperc múlva visszatér egy kicsit eltérő árral. Ennek megfelelően a tranzakció típusa értelmét veszíti, ugyanis annak csak a tőzsdei kereskedőnél van befolyása, jelen esetben nincs felhasználva.

Másik szolgáltatás a határidős derivatívák lejárásakor szükséges aktuális piaci ára egy instrumentumnak. Egy tőzsdén jelenlévő rendszer ezt minden pillanatban ismeri, de jelen esetben csak egy konkrét értéket ad vissza minden termékre.

3.4.2 Államkötvények periódusának és lejáratának kezelése

Ezen speciális instrumentum időjellemzőinek mérőegysége rendszerint egy nap, ez azonban jelen esetben nem praktikus. Ezért bevezettem azt, hogy néhány másodpercenként kamatoznak az államkötvények, és a lejáratuk véletlenszerűen néhány periódus után van. Ezzel a változtatással már jól megfigyelhető a kezelő folyamat működésének mindkét végrehajtási ága, azonban az eladás időpontja nem ismert előre.

3.4.3 Árfolyamadatok megadása

Egy éles banki rendszer esetében folyamatosan érkeznek a friss devizaadatok, ezért az átváltási ráták minden időpillanatban elérhetőek. Jelen esetben azonban ez nem működik így, ezért az adatbázisba kézzel kell felvenni őket, statikusak lesznek.

4 Kitekintés

A program működő állapotba került, a specifikációtól csak az előző fejezetben ismertetett pontokban tér el.

Az elkészített alkalmazás csak bemutató jellegű, több ponton is kínál továbbfejlesztési lehetőséget. A teljesség igénye nélkül néhány lehetséges pont:

- Jelenleg a webszolgáltatások nincsenek levédve semmilyen biztonsági megoldással. Erre a célra lehetne használni például SAML (Security Assertion Markup Language)-t, mellyel a különböző domének közötti bejelentkezési információk megoszthatóak lennének.
- Jelenleg a felhasználók egy adatbázisba vannak felvéve, erre azonban univerzálisabb megoldás lenne egy címtár.
- Nincs naplózás.
- A távoli megfigyelhetőséget és beavatkozást JMX (Java Management Extensions)-el lehetne megoldani.
- A BPEL folyamatok tranzakcionálása és visszagörgetése nincs megoldva.
- A külső rendszerhez menő hívás jobb lenne JMS (Java Message Service) üzenetekkel megoldani webszolgáltatás helyett.
- A webszolgáltatások kibővíthetőek megbízhatóságot növelő, valamint titkosított kiterjesztések használatával.
- A hibakezelés sok helyen hiányos, a felhasználókhoz kijutnak a hibaüzenetek.
- A bemeneti mezőkre szükséges lenne felvenni validátorokat.
- A felhasználók nem kapnak visszajelzéseket a műveleteik során.

5 Összefoglalás

A szakdolgozatom során elkészítettem egy bemutató rendszert nagyvállalati környezetre specializált technológiákkal. Ennek során megismerkedtem a Java EE több szabványával, a webszolgáltatások fejlesztésének lépéseivel, valamint üzleti folyamatok megvalósításának szükségességével és előnyeivel.

A Java EE a web rétegre összetett, de általánosan használható megoldást nyújt a JSF-el, mely beépített támogatással rendelkezik a legtöbb webalkalmazásokban felmerült igényre, amelyre pedig nem, a nyitottságának köszönhetően kevés utánajárással egyszerűen megoldható. Kevés nehezen megoldható problémába ütköztem a fejlesztés során, ezek egyike a nem menedzselt erőforrásba függőséget injektálni. Az üzleti logika réteg fejlesztése az 5-ös verzióval sokat egyszerűsödött, ehhez nagyban hozzájárul a telepítési leírók teljes mellőzhetősége, amely nagyban növeli a karbantarthatóságot és az átláthatóságot. A JPA kényelmes és egyszerűen felépített megoldást biztosít az adatok eléréséhez. Problémát okozhat azonban, hogy az ear-ba csomagolt alkalmazásban ha több ugyanolyan java nevű osztály is létezik, akkor azok összekeveredhetnek.

A webszolgáltatások fejlesztése és használata során a teljes folyamat magától értetődő. Bár a protokoll a szerkeszthetőség miatt XML-ben van ábrázolva, a feladat közben nem volt szükségem a generált leírókat kézzel szerkeszteni. Egyetlen probléma a paraméter nélküli függvények hívása BPEL-ből.

A BPEL fejlesztéséhez átlátható és hatékony eszközök állnak rendelkezésre. Bár kiváltható lenne, mégis létjogosultsága van a kritikus és/vagy sűrűn változó üzleti folyamatok leírásában.

Összességében a tapasztalatom az, hogy ezen technológiák rendkívül jól skálázható, valamint megfelelő tervezés mellett nagy megbízhatóságú rendszerek fejlesztését teszik lehetővé, azonban szerteágazódásuk miatt hosszú tanulási görbével rendelkeznek, valamint a robusztusság miatti járulékos erőforrásigény is magas.

6 Irodalomjegyzék

[1] Balogh Péter, Berényi Zsolt, Dévai István, Imre Gábor, Soós István, Tóthfalussy Balázs: Szoftverfejlesztés Java EE platformon

[2] JTA specifikáció: http://jcp.org/en/jsr/detail?id=907

[3] JPA specifikáció: http://jcp.org/en/jsr/detail?id=317

[4] JDBC specifikáció: http://jcp.org/en/jsr/detail?id=54

[5] SQL: http://en.wikipedia.org/wiki/SQL

[6] EJB 3.0 specifikáció: http://jcp.org/en/jsr/detail?id=220

[7] JMS specifikáció: http://jcp.org/en/jsr/detail?id=914

[8] RBAC: http://en.wikipedia.org/wiki/Rbac

[9] Szervlet specifikáció: http://jcp.org/en/jsr/detail?id=315

[10] JSP specifikáció: http://jcp.org/en/jsr/detail?id=245

[11] JSF specifikáció: http://jcp.org/en/jsr/detail?id=252

[12] HTTP specifikáció: http://www.ietf.org/rfc/rfc2616.txt

[13] XML specifikáció: http://www.w3.org/XML/

[14] WSDL specifikáció: http://www.w3.org/TR/wsdl

[15] SOAP specifikáció: http://www.w3.org/TR/soap/

[16] BPEL specifikáció:
http://www.ibm.com/developerworks/library/specification/ws-bpel/

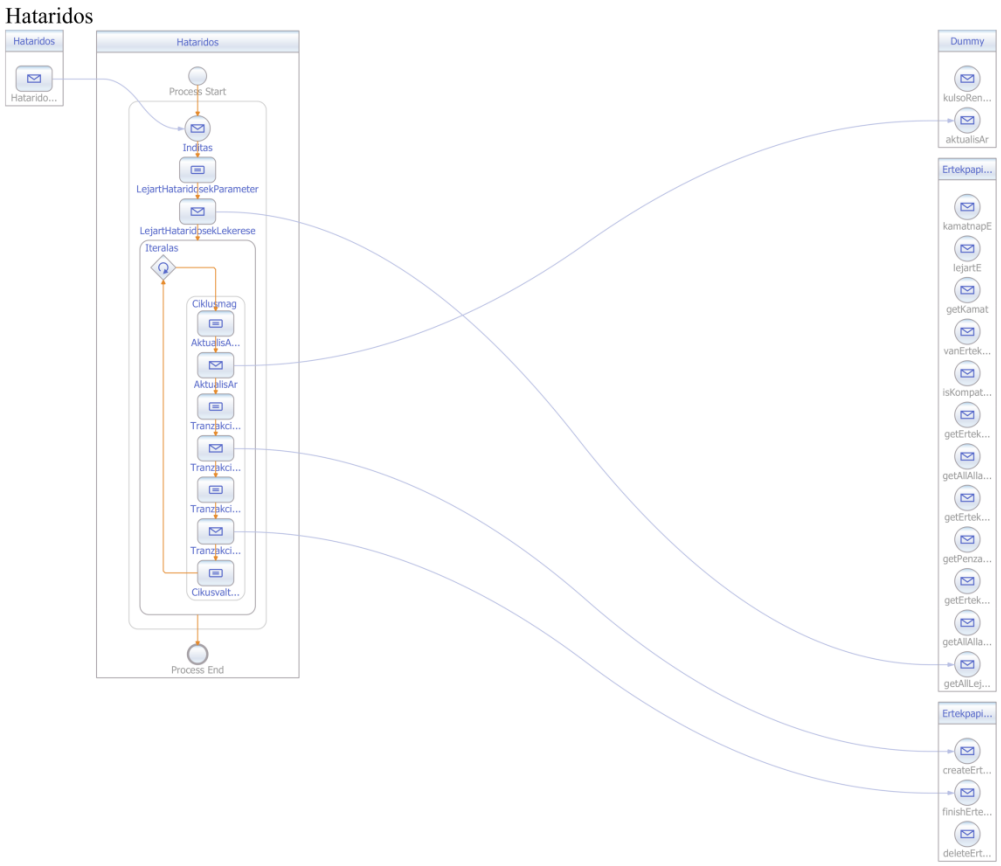
[17] Glassfish Administration Guide:
http://dlc.sun.com/pdf/821-0185/821-0185.pdf

[18] David Heffelfinger: Java EE 5 Development with NetBeans 6

7 Képjegyzék

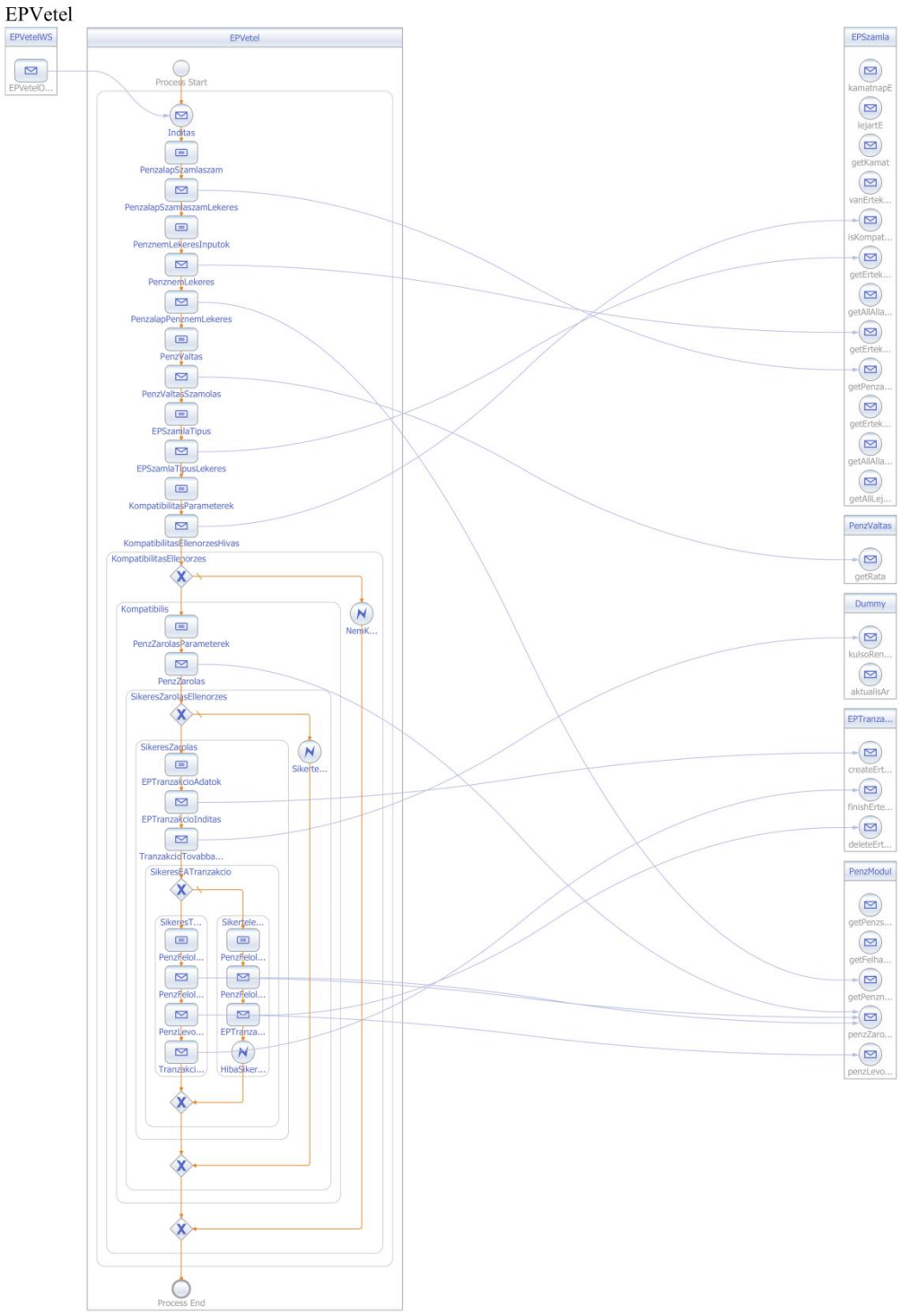
8 Függelék

8.1 Függelék A



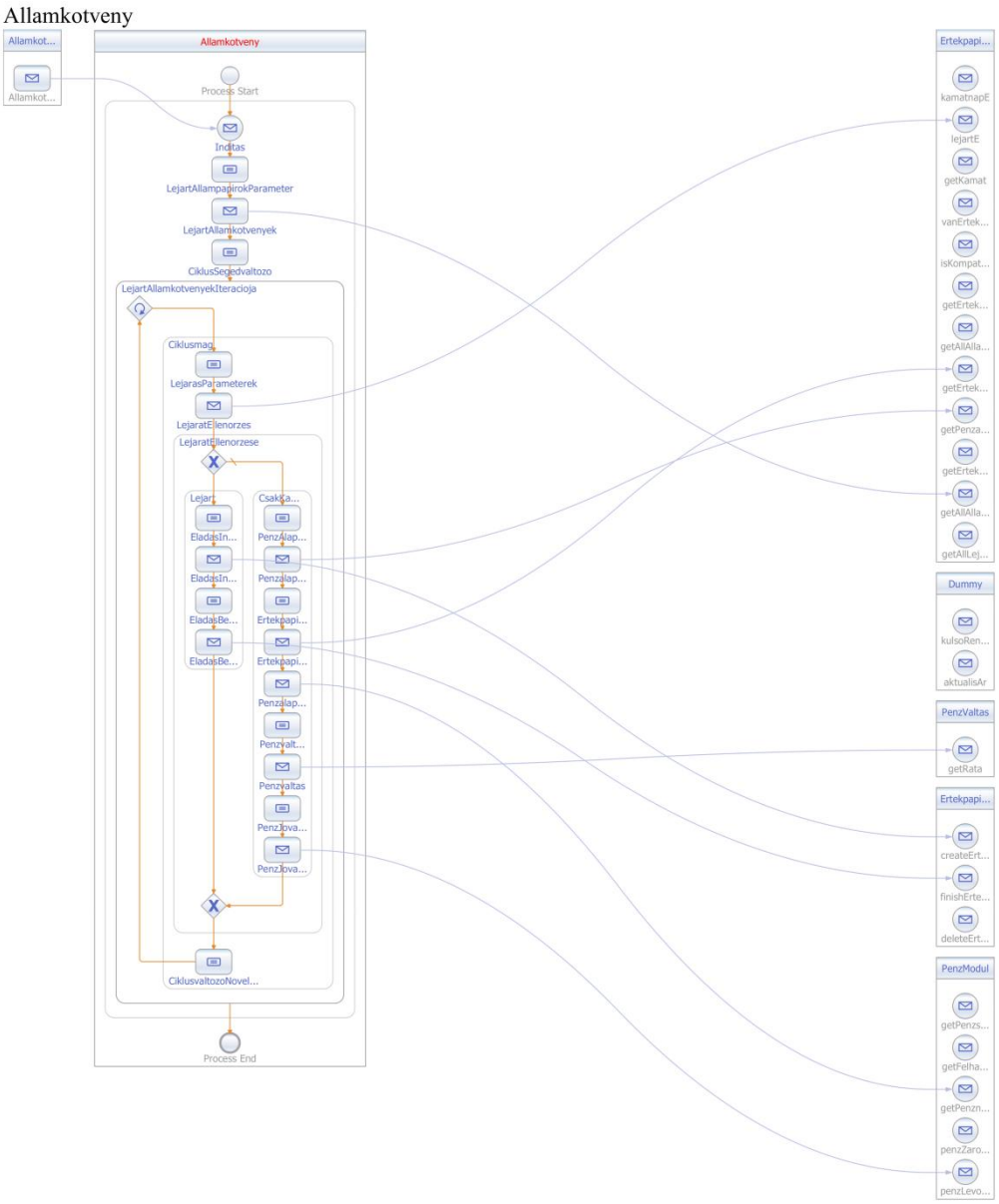
Kép 6: A határidős derivatívák lejáratát kezelő folyamat

8.2 Függelék B



Kép 7: Az értékpapírok vételét/eladását kezelő folyamat

8.3 Függelék C



Kép 8: Az államkötvények kamatozását és lejáratát kezelő folyamat

8.4 Függelék D

A mellékelt virtualizált környezet használata:

A DVD-n lévő vdi állomány egy merevlemez, ezt először írható helyre kell másolni, majd létre kell hozni egy virtuális gépet a Sun xVM VirtualBox programmal. Ajánlatos legalább 512 megabájt memóriát a rendelkezésére bocsátani. A gép indulás után elindítja az adatbázist és az alkalmazásszervert, ez eltarhat néhány percig. Ezután a Firefox webböngészőt elindítva és a http://localhost:8080-as címet beírva jutunk el a bejelentkező oldalra. Amennyiben itt fehér oldal jön be, akkor valószínűleg még nem indult el a futtatókörnyezet.

A bejelentkező képernyő a főoldal. Itt a c/c nevű/jelszavú felhasználóval be lehet lépni. Ezután lépjünk tovább a befektetések modulba, itt az összefoglaló táblázat jön be, jelenleg még üres. Hozzunk létre új értékpapírszámlát, melynek a típusa legyen BUX. Több pénzszámla közül is választhatunk, esetünkben mindegy, hogy melyiket választjuk. Miután a számla készen van, az összesítőben megjelenik. Ezután indítsunk el egy tranzakciót. Ki van választva az imént létrehozott számla, majd válasszuk ki, hogy rendes értékpapírt, államkötvényt vagy határidős instrumentumot akarunk venni. Az árhoz írjunk egy nem túl nagy számot, mondjuk 10-et, a mennyiséghez egy egészet, a típushoz bármit, majd kattintsunk a létrehozásra. Ekkor visszakerülünk az összefoglalóra, de az oldalbetöltés és a folyamat versenyhelyzete miatt nem biztos, hogy a tranzakció megjelenik. Ha nem látjuk, frissítsük az oldalt. Amennyiben még ekkor sem jelenik meg, akkor valószínűleg nincs elég pénz a számlán. Ha sikerült létrehozni, akkor 10 másodperc múlva kitöltődik a teljesítési idő, ezzel a tranzakció befejeződött. Amennyiben államkötvényt vettünk, akkor másodpercenként fog kamatozni, és egyszercsak el fog tűnni, ahogy a határidős is. Ekkor egy új tranzakció fog létrejönni, mely lenullázza a lejárt instrumentum darabszámát.