

Pannon Egyetem
Műszaki Informatikai Kar
Villamosmérnöki és Információs Rendszerek Tanszék
Mérnökinformatikus BSc

SZAKDOLGOZAT

Életmódnapló-adatbázis migrálását támogató alkalmazás tervezése és megvalósítása

Lénárt Bálint

Témavezető: Dr. Vassányi István
2016

Szakdolgozat témakiírás

Nyilatkozat

Alulírott Lénárt Bálint hallgató, kijelentem, hogy a dolgozatot a Pannon Egyetem Villamosmérnöki és Információs Rendszerek tanszékén készítettem a mérnökinformatikus végzettség megszerzése érdekében.

Kijelentem, hogy a dolgozatban lévő érdemi rész saját munkám eredménye, az érdemi részen kívül csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy a dolgozatban foglalt eredményeket a Pannon Egyetem, valamint a feladatot kiíró szervezeti egység saját céljaira szabadon felhasználhatja.

Veszprém, 2016. november 21.

aláírás

Alulírott dr. Vassányi István témavezető kijelentem, hogy a dolgozatot Lénárt Bálint a Pannon Egyetem Villamosmérnöki és Információs Rendszerek tanszékén készítette a mérnökinformatikus végzettség megszerzése érdekében.

Kijelentem, hogy a dolgozat védeésre bocsátását engedélyezem.

Veszprém, 2016. november 21.

aláírás

Köszönetnyilvánítás

Ezúton szeretném megköszönni témavezetőmnek, dr. Vassányi Istvánnak, hogy tapasztalatával, tanácsaival és tudásával támogatta és segítette szakdolgozatom elkészítését.

Hálával tartozom továbbá családtagjaimnak, akik nélkül ez a dolgozat nem jöhetett volna létre. Köszönöm nekik, hogy tanulmányaim során türelemmel és megértéssel támogattak, és minden helyzetben mellettem álltak.

TARTALMI ÖSSZEFOGLALÓ

A Pannon Egyetem Villamosmérnöki és Információs Rendszerek Tanszékén működő Egészségügyi Informatikai Kutató-Fejlesztő Központ már hosszú ideje fejleszt egy életmód-tükör alkalmazást, a Laviniát. Segítségével könnyen lehet naplózni különböző eseményeket táplálkozásunkkal és egészségünkkel kapcsolatban. 2013 és 2015 között klinikai kísérletek folytak a Honvédkorház Balatonfüredi Kardiológiai Rehabilitációs Intézetén, ahol 4 klinikai kísérlet keretében összesen 80 cukorbeteg személy részvételével tesztelték a Laviniát.

A kísérletek többek között azt is kimutatták, hogy a program használatának hajlandósága nem csökkent. Vannak alanyok, akik a mai napig is rendszeresen használják a szoftvert. A felhasználók száma eközben csak nőtt, mert bárki bekapcsolódhat a tesztelésbe. Egyértelmű következtetés, hogy a rendszer apró módosítások mellett piacképes és hatékonyan segíti az emberek táplálkozásainak naplózását és nyomon követését.

A kísérletben minden felhasználóhoz egy eszköz tartozott, ami egyértelműen azonosította az általa rögzített megfigyeléseket. Az eszköz alapú megfigyelés helyett a piaci szoftver adatbázisa esemény alapú. Ez sokkal jobban kezeli az egyes eseményeket, valamint az ehhez tartozó számításokat, kimutatásokat. A séma váltás mellett adatbáziskezelő rendszerváltozás is történt. Az eddig használt NoSQL (MongoDB) adatbázis helyett már relációs adatbázis kezelő rendszer, a PostgreSQL lesz az alkalmazás mögött. Szakdolgozatom témája, hogy a jelenlegi MongoDB-ben található dokumentumokat az új séma alapján migráló alkalmazás segítségével átültessem a leendő adatbázis tábláiba. Ehhez olyan szoftvert sikerült készíteni, ami képes nagy mennyiségű adatok kezelésére. Sok esetben az adatokat vagy kapcsolatukat át is kellett alakítani, így jelentős mennyiségű adatfeldolgozás is történt.

Kulcsszavak: életmód, migráció, Java, NoSQL, relációs adatbázis

ABSTRACT

The Medical Informatics Research and Development Center at the Department of Electrical Engineering and Information Systems, University of Pannonia has been developing a lifestyle mirror application, the so-called Lavinia. With the help of the program it will be easier to logging the different processes in connection with our nutrition and health. Between 2013 and 2015 clinical experimentations were conducted in the Cardiological Rehabilitation Institute of Balatonfüred, where they tested the Lavinia with the participation of 40 diabetic person during 4 trials.

The experimentations have shown that the willingness of the using the program did not decrease. There are several people who regularly use the software. Meanwhile the number of the users has increased, because anyone can participate in the testing process. Obvious conclusion that the system is marketable with some minor modification and effectively helps to audit and examine people's nutrition.

In the experimentation process, each person had a device, which apparently identified the recorded observations. Instead of the the asset-based examination, the new software's database is event-based. As a result, it is much better in handling the certain events, as well as the related calculations and accounts. In addition to changing the schema, there is a difference in the database management system too. Instead of the previously used NoSQL (MongoDB) database the background of the application will be a relational database management system, the PostgreSQL. The theme of my thesis is to transpose the current MongoDB documents to the prospective tables of the database with the help of the application based on the new scheme. For that I could accomplish a software, which is able to manage large amount of data. In many cases it was needed to convert the data or its connection, thus significant amount of data processing has happened.

Keywords: lifestyle, migration, Java, NoSQL, relational database

Tartalomjegyzék

Szakdolgozat témakiírás.....	II
Nyilatkozat.....	III
Köszönetnyilvánítás.....	IV
TARTALMI ÖSSZEFOGLALÓ	V
ABSTRACT	VI
1. Bevezetés	1
2. Felhasznált technológiák	5
2.1. Java.....	5
2.1.1. JavaFX	5
2.2. Apache Maven	6
2.2.1. Mongo Java Driver	6
2.2.2. PostgreSQL Driver	6
2.2.3. Project Lombok	7
2.2.4. Log4j.....	7
2.2.5. Google Guava.....	7
2.2.6. Apache Commons IO	8
2.2.7. Apache Commons Lang 3.....	8
2.3. Git.....	8
2.4. JetBrains IntelliJ IDEA	8
3. Környezet ismertetése.....	9
3.1. Entitások.....	9
3.1.1. Felhasználók, epizódok.....	9
3.1.2. Eszközök.....	10
3.1.3. Megfigyelések	10
3.2. Adatbázisok ismertetése	12

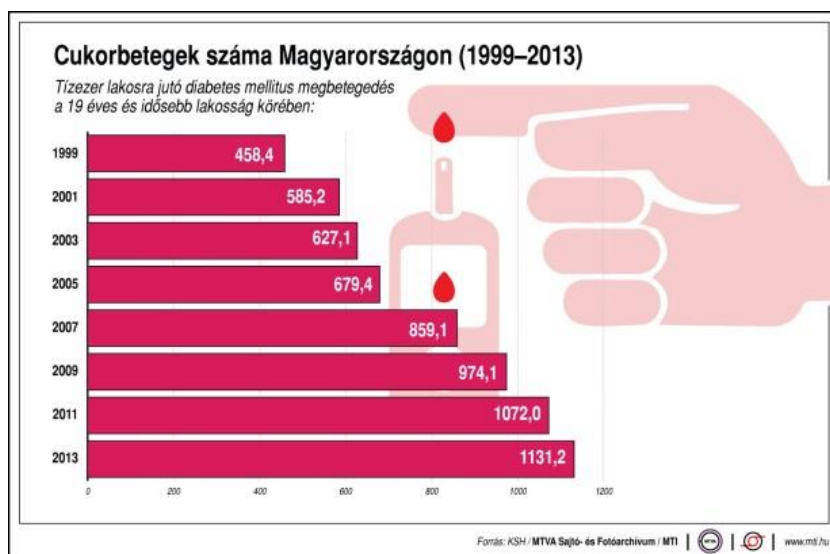
Tartalomjegyzék

3.2.1. Relációs adatbázis	12
3.2.2. Dokumentumtároló adatbázis.....	14
3.3. Lokális teszt környezet kialakítása	19
3.4. Dokumentumtár tartalma.....	21
4. Specifikáció	24
5. Rendszerterv	26
5.1. Megvalósítási lehetőségek.....	27
5.2. Migrációs folyamat tervezése	29
5.3. Adatbázissal való kommunikáció	31
5.4. Forrásfájlok könyvtárszerkezete	32
6. Implementálás.....	34
6.1. Bejelentkezés	34
6.2. Inicializálás	35
6.3. Felhasználói felület	36
6.4. Migrációs folyamat	38
6.5. On-line nyomon követés.....	41
6.6. PostgreSQL adatbázisba való betekintés.....	44
6.7. Biztonsági mentés	44
6.8. Naplózás	45
6.9. Beállítások	47
7. Tesztelés	49
8. Összefoglalás	53
<i>Irodalomjegyzék.....</i>	54
Mellékletek.....	56

1. Bevezetés

Az egészséges életmód egyre fontosabbá válik az emberek körében. Ennek egy meghatározó része a tudatos táplálkozás. Sokaknak az egészségügyi szempontok jutnak először eszébe, mint például a cukorbetegség, elhízás, hiánybetegségek, stb. Ezeknek a kóroknak a következményei és szövődményei súlyosak és sokszor visszafordíthatatlanok.

Az 1.1. ábrán látható a cukorbetegek számának folyamatos növekedése Magyarországon.



1.1. ábra Cukorbetegek száma Magyarországon

A helytelen táplálkozásból fakadó betegségek mellett az informatikai eszközök is robbanásszerűen hódítanak teret. A ma kapható mobiltelefonok legnagyobb része már az okostelefonok kategóriájába tartozik. Segítségükkel szinte bárhol internetezhetünk, valamint a fejlesztőknek köszönhetően rengeteg játék, illetve számos más applikáció jelent meg és jelenik meg folyamatosan, szinte minden nap. Az egyik legnépszerűbb mobiltelefonokkal foglalkozó internetes oldal becslései alapján öt év alatt közel harmincszorosára nőtt az okostelefonok száma csak Magyarországon. A növekedés az 1.2. ábrán látható.



1.2. ábra Okostelefonok száma Magyarországon

Az interneten rengeteg magyar és idegen nyelvű cikk született már a tudatos táplálkozás és az egészséges életmód jegyében. Orvosoktól, szakértőktől, betegektől rengeteg hasznos információt kapunk, arról, hogyan tartsuk kezeink között egészségünket. Ennél egy bizonyos szempontból jobbak azok a szoftverek, melyek adataink alapján valamilyen index vagy egyéb számítást végeznek, amik indoklás mellett tanácsot is adnak.

Talán legegyszerűbb példa erre BMI index, azaz a testtömeg-index számítás. Ez az algoritmus magasságunkat, nemünket és életkorunkat figyelembe véve egy index-et számol, amiből megtudhatjuk, hogy testsúlyunk ideális-e. Ez az algoritmus egyszerű, viszont kevés paramétert vesz figyelembe, így az általa számolt eredmény nem ad komplex leírást, csak bizonyos egészségügyi paraméterünket osztályozza.

Vannak ennél bonyolultabb számítások, amiknek a kimenete sokkal több és összetettebb információt hordoz. Könnyen belátható, hogy ez gyorsítható számítógépes támogatással. Olyan szoftverek is léteznek, melyek számon tartják napi étkezéseinket, fizikai aktivitásainkat, gyógyszerbeviteleinket, stb. Ezek alapján már egy komplex számítást is lehet végezni, valamint becsülni rövid távon előre. Sokkal több előnyt hordoz a szoftveres integrálás. Számítógépek, okostelefonok, okosórák és egyéb szenzorok az internet vagy helyi hálózatok segítségével szinte azonnal tudnak kommunikálni egymással, így a folyamat egy

része automatizálható is. Egy jó szoftver tudástárát könnyű bővíteni, így könnyen és gyorsan taníthatunk meg programunknak új recepteket vagy fizikai aktivitási típusokat.

Látható, hogy egyre több ember rendelkezik valamilyen okostelefonnal. Egy jól tervezett mobil alkalmazás, ami képes az ismertetett adatokat tárolni, majd azokon különböző számításokat végezni, akár éleket is menthet. Sokkal könnyebb áttérni egészséges életmódra úgy, hogy szinte azonnal elkönnyvelhetjük egészségügyi paramétereinket és erről már abban a pillanatban visszajelzéseket olvashatunk.

A Pannon Egyetem Műszaki Informatikai Karán működő Egészségügyi Informatikai Kutató-Fejlesztő Központ készítette el a Lavinia életmód-tükör alkalmazás első verzióját. A mai napig tartó fejlesztést számos hazai és európai uniós projekt támogatja. Az alkalmazás felhasználói a fogyni, hízni vagy éppen egészséges életmódot megtartani kívánt személyek, akik kezelni tudják okostelefonjaikat. Az alkalmazásban található napló alapja a magyar kultúrából és szokásokból álló adatbázis, melyet szakértő segítségével állítottak össze.



1.3. ábra Lavinia alkalmazás kezdőképernyőre

Az 1.3. ábrán a Lavinia felhasználói felülete látható rögtön az alkalmazás indítása után. Innen könnyedén és gyorsan elérhetjük a kívánt funkciókat, így az egyes tételek naplózás nem vesz el sok időt a felhasználótól. A klinikai tesztek kimutatták, hogy a naplózási időtartam a mobil technológiákban nem jártas idős

betegeknél is napi 5 percre csökkenthető. A cukorbetegek részére az egyik legfontosabb szolgáltatás az azonnali visszajelzés.



1.4. ábra Naplózó felület

A naplóba felvitt új tételek alapján a rendszer automatikusan kiszámolja, hogy az utoljára rögzített étel milyen mértékben növeli a napi ajánlott zsír-, energia-, illetve szénhidrát-keretet (1.4. ábra). A glikémiás index mutatja a cukor felszívódásának gyorsaságát. Ennek köszönhetően azok a betegek, akiknél fontos ez az információ könnyen össze tudnak állítani maguknak menüket a napi étkezésekhez. Az inzulin bevitel naplózásával a Lavinia képes matematikai modellek alapján rövid távon kiszámolni előre a vércukorszint várható értékét.

A Laviniát a balatonfüredi kórházban öt – egyenként húsz fős – kísérletben tesztelték. A cukorbetegek tekintetében a vércukorszint mérések eredményeinek szórása (azaz a veszélyesen alacsony és veszélyesen magas értékek előfordulási gyakorisága) számottevően csökkent. Továbbá csökkent az átlagos vércukorszint és testsúly is.

2. Felhasznált technológiák

Dolgozatom ezen részében azokat a technológiákat szeretném bemutatni, amiket felhasználtam a program megírása során. Már tervezési fázisban kerestem és ismerkedtem azokkal a modulokkal, függőségekkel, amik megkönnyítik a fejlesztést. Az általam felhasznált ingyenes könyvtárak nemcsak nagyban segítettek a munkámat, hanem csökkentették a kód méretét is, ami a karbantartást nagymértékben javítja. A modulok megismerését és elsajátítását egyéb területeken is hasznosan tudom majd felhasználni.

2.1. Java

A Java ma az egyik legelterjedtebb programozási nyelv, melynek fejlődése folyamatos és töretlen. Egy általános, objektumorientált programozási nyelvről van szó, ami az 1990-es évek elejétől a mai napig fejlődik. Választásom azért esett erre a nyelvre, mert rengeteg könyvtár található hozzá, általános célú, valamint a programozási ismeretemet is legjobban és legkönnyebben Java-ban tudom fejleszteni.

Platformfüggetlen, ami azt jelenti, hogy egy jól megírt kód egyszeri fordítás után minden operációs rendszeren ugyanúgy fog futni. Ez azért lehetséges, mert a fordítás alatt egy úgynevezett Java bájtkód jön létre. Ez még nem gépi kód, azaz futtatása nem lehetséges. Ahhoz, hogy ezt futtatni tudjuk szükségünk lesz egy Java virtuális gépre. Ez lesz az, ami értelmezi a bájtkódot és natív kódot készít belőle. A megoldás jellegéből adódóan lassabb, mint egy azonnal natív kódra forduló nyelv. Ez a romlás azonban ilyen feladatnál nem szignifikáns, ezért bátran merem használni.

2.1.1. JavaFX

A JavaFX egy 2006-ban megjelent könyvtár Java alkalmazásokhoz. Segítségével könnyebbé tehető a grafikus felhasználói interfésszel rendelkező alkalmazások fejlesztése. Könnyen lehet vele implementálni animációt, audió és vizuális effekteket. Kiadása után akkora érdeklődést keltett, hogy a Java 8-as verziójában már beépítve megtalálható (korábbi kiadások mellé függőségként kellett letölteni).

A keretrendszer teljes mértékben támogatja az úgynevezett MVC (Model, View, Controller- magyarul: Modell, Nézet, Vezérlő) tervezési mintát. A „Nézet” felület nem más, mint egy speciális XML formátumú fájl, ami leírja a komponensek elhelyezkedését és egyedi tulajdonságait. Ezeket lehet módosítani a Vezérlőből. Az ilyen osztályokban egyedi azonosítók alapján tudunk az egyes mezőkre hivatkozni. A JavaFX a kódban lévő változót és az XML-ben definiált komponenst automatikusan rendeli össze injection segítségével. A modell réteg nem más, mint egyszerű POJO (Plain Old Java Object) osztályok halmaza.

2.2. Apache Maven

Az Apache Maven a szoftveres projektek hatékony menedzselésére és a build (magyarul: építés) folyamat automatizálására jött létre a 2000-es évek elején. Használatával egy egyszerű XML fájlban definiálni tudjuk a projekt összes függőségét és tulajdonságát.

Használatát azért választottam, mert az XML-ben megadott projekt függőségeit automatikusan tölti le a központi szerverről (Maven Central Repository). Az általam írt szoftvernek pár függősége van, amit röviden be is mutatok.

2.2.1. Mongo Java Driver

Hivatalos interfész implementáció az adatbázisszerver, valamint a program között. Támogatja a szinkron, illetve aszinkron hívásokat is. A könyvtár tartalmazza a régebbi komponenseket is a kapcsolat létrehozásához, de szakdolgozatom során mindig a legfrissebb és a MongoDB fejlesztői által javasolt osztályokat és függvényeket használtam.

2.2.2. PostgreSQL Driver

Ez a könyvtár hasonló tudással bír, mint a MongoDB-s társa. A Java nyelv tartalmaz egy interfészt, ami segítségével könnyen lehet relációs adatbáziskezelő rendszerekkel kapcsolatot létrehozni és ezeken műveleteket végrehajtani. Ez a JDBC (Java Database Connectivity). Ennek az egyik megvalósítása a PostgreSQL

fejlesztői által kiadott driver. Feladata, hogy az interfészen keresztül biztosítsa a kapcsolatot és az adatáramlást a Java program és az adatbázis között.

2.2.3. Project Lombok

A Project Lombok könyvtár segítségével sokat szépíthetünk kódjainkon, miközben időt takarítunk meg. Példaként lehet említeni egy egyszerű POJO osztály felépítését. Az objektumorientált programozási elveket figyelembe véve az osztályváltozókat kívülről nem lehet elérni, csak azokat lekérdező és módosító függvényekkel. Ezek a függvények általában nem tartalmaznak semmit, csak az érték beállítását vagy visszaadását. A Project Lombok használatával automatikusan lehet generálni az osztályba ezeket a függvényeket. Ez fordítási időben történik, így a kód tiszta marad. További funkcionálisai közé tartozik, hogy egyetlen annotáció segítségével automatikusan Builder (magyarul: építő) tervezési mintához igazítja az osztályokat. Legtöbb helyen setter és getter függvényekhez, valamint az alapértelmezett és paraméteres konstruktorok létrehozásához használtam ezt a hasznos könyvtárat.

2.2.4. Log4j

A Log4j egy Java alapú naplózási keretrendszer, amit jelenleg az Apache Software Foundation tart karban. E könyvtár segítségével készítettem el a szoftver naplózási modulját. Használata igen egyszerű és gyors. Rengeteg naplózási szintet, illesztőt és formátumot lehet beállítani, azaz nagyon könnyen testreszabható. Létezik hozzá rengeteg kiegészítő osztály és beállítás, de ezek használatától eltekintettem, mivel az alap keretrendszer is képes előállítani azokat a kimeneteket, amikre szükségem volt.

2.2.5. Google Guava

A Guava egy nyílt forráskódú könyvtár gyűjtemény, amit a Google mérnökei fejlesztenek. Mivel több korábbi könyvtárból áll össze, ezért a funkcionalitása szerteágazó. Segítséget adott a kollekciók létrehozásában és a rajta végzett műveletek végrehajtásában.

2.2.6. Apache Commons IO

Az Apache Commons IO az Apache Software Foundation által jelenleg is fejlesztett könyvtár egyike. Ez a keretrendszer a fájlműveletekre helyezi a hangsúlyt. Bár a feladatomban viszonylag kevés fájlművelet található, az egyes helyeken e könyvtárban található osztályok függvényeit használtam. Fájlok tartamának beolvasására, valamint a becsomagolt JAR állományból való fájl másolásnál alkalmaztam.

2.2.7. Apache Commons Lang 3

Ez a függőség nagyon hasonlít az előbb bemutatott keretrendszerre. Az osztályok és a metódusok, leginkább a Java *lang* nevű csomagjaiban található objektumok manipulálást segítik. Ide tartoznak a különböző szöveg, szám, környezeti változó, stb. műveletek. Mivel sokszor kellett szövegeket feldolgoznom, így ezzel a könyvtárral sok időt takarítottam meg.

2.3. Git

A Git egy verziókezelő rendszer, melynek feladata a különböző fájlok verzióinak tárolása, kezelése és megosztása. Rendkívül hasznosnak bizonyult egy-egy komponens fejlesztése közben. Az általa fenntartott kód adatbázisban mindig egy működő, tesztelt kódhalmaz volt. Ennek előnye, hogy egy esetlegesen rosszul implementált verzióról egyszerű a visszaállítás az utolsó jóváhagyottra. Segítségével áttekinthetőbbé vált a fejlesztés üteme is.

2.4. JetBrains IntelliJ IDEA

Az egyik legnépszerűbb integrált fejlesztői környezet Java nyelvhez. Kereskedelmi és közösségi kiadásban is egyaránt megszerezhető. Használatával rendkívül egyszerűsödik a fejlesztése a kód navigációnak, automatikus kódkiegészítőnek és egyéb hasznos funkciónak köszönhetően. Magas fokon támogatja a kód átalakításához szükséges kiegészítőket: automatikus formázás, függőség felismerés. Beépített moduljai kezelik az előbb említett technológiákat, mint például a Git és a Maven.

3. Környezet ismertetése

A tervezés első lépése a már kialakított és jól működő környezet megismerése. Ebben a fejezetben bemutatom a migrálásban részt vevő entitásokat, valamint az adatbázisok szerkezetét és tartalmát. Egészségügyi adatbázis révén szükségszerű volt megismerkednem alapvető orvosi és egészségügyi kifejezésekkel, paraméterekkel és ezek lehetséges értékeivel.

3.1. Entitások

A következőkben leírt modellek alkotják a Lavinia rendszer fő részének egy darabját. Ezek definiálják az elvégzett megfigyeléseket, azok tulajdonosát, mérőeszközét. Az összetett adatszerkezetek mellett az egymás közötti kapcsolatokat is részletesen meg kell ismerni ahhoz, hogy a céladatbázisban a hivatkozási integritások megfeleljenek a követelményeknek.

3.1.1. Felhasználók, epizódok

Az adatbázis egyik központi szereplője a felhasználók. A szoftver használatához regisztráció szükséges. A Lavinia rendszer ekkor csak a legalapvetőbb adatokat kéri el, mint például a név és az email cím.

Az alkalmazást igénybe vevő személyeknek több típusa van. A legtöbben egyszerű felhasználók, akik saját méréseiket viszik fel a rendszerbe. Rajtuk kívül lehetnek még orvosok, dietetikusok is. Az ő szerepkörük, hogy a hozzárendelt felhasználók mérései és a rendszer kiértékelései alapján döntéseket hozzanak, ezzel segítve a pácienseik helyes életmódját. Az adminisztrátorok felügyelik a rendszer hibátlan működését, folyamatosan tesztelik a funkciókat és fejlesztik a szoftvert. Természetesen az orvosok mellett ők is használhatják a Laviniát úgy, mint az egyszerű felhasználók.

A felhasználók egy cél elérése érdekében epizódokat hozhatnak létre. Ilyen lehet például 20 kg leadása, 70 kg elérése, stb. A cél bármi lehet, amit a felhasználó definiál. Ezekhez az epizódokhoz lehet eseményeket rögzíteni.

3.1.2. Eszközök

A felhasználók az egyes méréseket eszközök segítségével tudják felvinni a rendszerbe, amik általában okostelefonok. Az eszközök nem a felhasználókhoz, hanem az epizódokhoz vannak rendelve. Az adatmodell segítségével egy epizódhoz bármennyi eszköz kapcsolható, valamint egy eszköz egyszerre több epizódban is lehet egy időben.

Több kiegészítő adatot is lehet egy eszközhöz rendelni, de migráció szempontjából ez nem releváns, adathiány miatt. Ezek az értékek segítik a különböző egészségügyi mutatók kiszámítását.

3.1.3. Megfigyelések

A felhasználók a Laviniában különböző egészségügyi megfigyelések sokaságát töltik fel. Ezek alapján történik a döntéstámogatás, valamint a felhasználóhoz tartozó indexek számítása. Az összes megfigyelés közös adatai a különböző időpontok (történés, felvétel, adatbázisba kerülés, módosítás, törlés), valamint az adott megfigyelést rögzítő eszköz azonosítója.

A megfigyelések között található a testsúlymérés. A felhasználó a mérlegről leolvasott értéket könnyen rögzítheti az alkalmazásban. A rendszeres mérés segítségével könnyen lehet visszakeresni vagy elemezni az epizódban történt testsúlyváltozást.

Továbbá gyakori a vércukorszint-mérés. A legfontosabb megfigyelések között szerepel a cukorbetegknél. A speciális mérőeszköz a vérben található szőlőcukor (glükóz) mennyiségét határozza meg.

Vérnyomásmérésnél rögzíthetjük a szisztolés, diasztolés és pulzus értékeket is. A szisztolés érték azt a nyomást mutatja, ami a szív összehúzódásakor alakul ki, míg a diasztolés érték a szívverések között mérhető. A pulzus a szívverés miatt kialakuló lüktetés az artériában. Méréskor a szív percenkénti összehúzódásainak számát határozzuk meg.

Rengeteg fizikai aktivitás létezik már, ami nem csak a testre, hanem a lélekre is pozitív hatással van. Segítik a fejlődést, izmok és csontok erősödését, valamint segítségével a felesleges energiát is le lehet vezetni. A fizikai aktivitás

típusa, valamint az ezzel eltöltött idő együttesével a rendszer kiszámolja a tevékenység során leadott energiát is.

Az egyes betegségek velejárói a gyógyszerek. A gyógyszerek összetevői különböző mellékhatásokat is kiválthatnak az emberekben, amik befolyásolhatják a Lavinia által kiszámított egészségügyi indexeket. Rögzíthetünk gyógyszeres beviteket is, ahol ki lehet választani a bevett gyógyszereket, azok mennyiségét, valamint beviteli módjait. Ezeket az eseményeket étkezésekhez is párosíthatjuk.

Az anamnézis görög eredetű szó, melynek jelentése: kórelőzmény. Célja, a páciens kórtörténetének megismerése, valamint a jelenlegi panaszok rögzítése. Ezt az orvosok is rendszeresen jegyzetelik betegek esetén. A Lavinia-ban rögzíthető a magasság, a testsúly, a születési idő, a nem, a sportolási szokások, az elérni kívánt testsúly és annak eléréséhez szükséges időtartam, valamint az ismert betegségek. A pontosabb meghatározás mellé pár laboreredményt is rögzíteni lehet. Az eGFR érték a vesebetegségekre hívhatja fel a figyelmet. Cukorbetegségeknél fontos az inzulinszint, hiszen testünk csak e hormon jelenlétében képes felvenni a glükózt a vérből. Sok kezelésben vesznek részt a szteroidok, amiknek hatása nagyon különböző lehet. A pontos kimutatások miatt célszerű a páciensnek megadni azt, hogy áll-e ilyen kezelés alatt. Ezeket az értékeket általában csak laboreredményekről ismerjük.

A legtöbbet használt és az egyik legfontosabb esemény az étel- és ital fogyasztás. Az egyes ételek, italok, élelmiszerek összetevői más és más módon hathatnak egészségi állapotunkra rövid és hosszú távon is. A rendszeres és egészséges étkezés egészségünk egyik legfontosabb szükséglete. A Lavinia rendelkezik ételek és azok összetevőit tároló adatbázissal. A felhasználónak csak ki kell választania az elfogyasztott ételt és mennyiségét. Mentés után rögtön frissülnek az aktuális napra vonatkozó egészségügyi mutatók. Ehhez a Lavinia az ételek összetevőit és azok glikémiás indexeit is figyelembe veszi számításakor.

Az említett élelmiszer adatbázis bővítésébe a felhasználók is bekapcsolódhatnak egy adott módon. Felvehetnek olyan ételeket, élelmiszereket vagy recepteket, amik nem szerepelnek ebben az adattárházban. Ekkor a Lavinia adminisztrátorai értesítést kapnak és felvehetik a hiányzó élelmiszert az adatbázisba.

Az egy csoportban lévő felhasználók látják egymás naplóbejegyzéseit. Ehhez egyszerű módon kommentet is tudnak írni. Megjegyzést lehet fűzni egy étkezési megfigyelésre, egy teljes napra vagy egyéb naplóban lévő rekordokra. Ezzel a felhasználók segíthetik egymást, tanácsokat és ötleteket adhatnak mások céljának elérése érdekében.

3.2. Adatbázisok ismertetése

Az adatbázisok (DB: Database) elsődleges feladata a többnyire strukturált adatok tárolása. Ezeket a kollekciókat menedzselő, általában rendszerszintű folyamatokat hívjuk adatbáziskezelő rendszereknek (DBMS: Database Management System). Együttük céljuk az adatok hosszú távú tárolása, védelme, gyors visszakereshetősége és azok módosítása.

Az adatbázisokat sokféleképpen csoportosíthatjuk. Egy szervezési mód az adatmodell tárolásán alapul. Ez alapján megkülönböztetünk relációs-, dokumentum-, kulcs-érték tároló, hálós-, gráf- vagy objektum-adatbázisokat. Dolgozatomban az első két típussal foglalkoztam, így azokat részletezem a továbbiakban.

3.2.1. Relációs adatbázis

A relációs adatbáziskezelő rendszerek (angol rövidítéssel: RDBMS – Relational Database Management System) a matematikai reláció definícióján alapulnak. Kidolgozása Edgar F. Codd nevéhez fűződik, aki 1970-ben jelentette meg modelljét és szabályait egy akkori újságban. Elmélete azóta is fontos szerepet játszik az adatbáziskezelők alkalmazásában. Előnye a többi rendszerhez képest, hogy az adatmodell egyszerű és könnyen megérthető, letisztult tervezési metodika tartozik hozzá, nagyfokú logikai függetlenséget biztosít, és matematikai alapokon nyugszik. Nagyban támogatja az ACID (Atomicity, Consistency, Isolation, Durability – magyarul: atomicitás, konzisztencia, izoláció, tartósság) tulajdonságokat, melyek segítségével garantálható az adatbázis konzisztenciája. Az elmélet szerint a tárolni kívánt adatokat relációkban kell elhelyezni. A felhasználók számára ezek táblázatos formában jelennek meg. Ez a legnagyobb adatszerkezet, ami egy jól definiált tárgyat képes leírni. A táblának egyedi neve van egy adatbázison belül. Ezt a

struktúrát oszlopok építik fel. A táblázatban egy sort rekordnak neveznek, ami egy adott entitás adatait írja le. A legkisebb elem a mező, ami egy entitás egyetlen atomi tulajdonságát tárolja. Az adatok visszakereséséhez és manipulálásához az SQL (Structured Query Language – magyarul: strukturált lekérdezőnyelv) áll rendelkezésre. Ismertebb relációs adatbáziskezelő rendszerek: Oracle, IBM DB2, Microsoft SQL Server, MySQL, SQLite.

A cél adatbázis és annak sémája a feladatom kiírása előtt definiálva lett, így az elkészült szoftvernek ehhez a sémához kell alkalmazkodnia. A sémáról készült ábrát az [1] számú melléklet tartalmazza.

Az adatbázis szerkezete esemény alapú, azaz előtérbe helyezi a felhasználók által rögzített megfigyeléseket. Minden típus egy specializáció az *ep_event* táblából. Ebben a relációban a különböző időpontok (mikor mérték, mikor mentették el, mikor került be az adatbázisba, mikor módosították utoljára, mikor törölték) és a mérőeszközre való hivatkozás található. A könnyebb azonosítás érdekében bekerült egy jelző érték is, ami pontosan beazonosítja, hogy ezek az adatok milyen típusú méréshez tartoznak.

A séma egy különálló része a kódszótár, mely funkciója, hogy a más helyeken elhelyezett kódokra magyarázatot adjon. Két táblából áll össze: a kód típusokat tartalmazó *code_type*, valamint az egyes kódokat tároló *code_item* relációból. Például az anamnézis eseményeket tároló táblában található *gender_code* mező értéke lehet 1 vagy 2. A mezőn található megjegyzés jelzi, hogy a 7-es kódkészlet vonatkozik a mezőre. Ilyenkor a kódszótárból kikereshető, hogy a 7-es csoportban mit jelent az 1-es és 2-es érték. Jelen esetben az első a férfit, a második a nőt definiálja. Mivel az adatok nem változnak – csak időként bővülnek – így célszerű egy view-t létrehozni. A view nem más, mint egy nevesített lekérdezés, mely tartalma, más forrásból tevődik össze. A következő parancs segítségével könnyen létrehozhatunk egy ilyen adatszerkezetet:

```
CREATE MATERIALIZED VIEW log.code_dict AS SELECT
ct.type_id, ct.type_name, ci.item_id, ci.item_name FROM
log.code_type ct JOIN log.code_item ci ON ct.type_id =
ci.type_id WITH DATA;
```

Piaci szoftver révén fontos a licencek kezelése, azaz engedélyek gyűjteménye az alkalmazás használására. Ezeket felhasználókhoz lettek rendelve, ami alapján pontosan azonosítani lehet, hogy az egyes személyek milyen jogosultságok birtokában állnak. A jelenlegi adatbázis nem rendelkezik ilyen információval, ezért a migráció szempontjából ezeket a relációkat figyelmen kívül hagytam.

3.2.2. Dokumentumtároló adatbázis

Az internet elterjedésével az információ és az adatmennyiség robbanásszerűen nőtt és nő a mai napig is. Becslések szerint naponta két és fél exabit (kb. két és fél trillió bit) adatot állítunk elő. Az adatok mennyisége és a jelenkori webes technológiák már feszegetik az RDBMS adatbázisok határait vagy éppen át is lépik azt. Olyan adatbáziskezelő rendszerekre mutatkozik igény, aminél az írás és olvasás művelet sokkal fejlettebb. A 2010-es évek elején a nagy informatikai cégek (pl. Google, Yahoo, Apache Alapítvány) sorra jelentették be saját NoSQL adatbázis rendszereiket. A NoSQL az angol Not Only SQL (magyarul: nem csak SQL) kifejezésből származik, ami utal arra, hogy ezek az adatbázisok szakítanak az SQL kifejezésekkel. Céljuk az írási és olvasási műveletek gyorsítása az RDBMS társaikhoz képest. Hátrányként lehet említeni, hogy sokkal kevesebb műveletet támogatnak (nagy általánosságban hiányoznak a megszorítások, join műveletek, tranzakciók, stb.). A szűk funkcionalitást sebességnöveléssel és nagyobb skálázhatósággal kompenzálják.

A NoSQL adatbázisokat csoportosíthatjuk gráf-, objektum-, kulcs-érték tároló és dokumentumtároló adatbázisokra. Többnyire az adattárolási módjukban különböznek egymástól. A szakdolgozatomban egy dokumentumtároló adatbázissal is kellett foglalkoznom. A legismertebb ilyen adatbáziskezelő rendszer a MongoDB. Ez a legelterjedtebb NoSQL adatbázisszerver, ami az adatokat bináris JSON (JavaScript Object Notation), azaz BSON formátumban tárolja, ezeket hívjuk dokumentumoknak.

A kapott minta adatbázis három kollekcióból épül fel: User (felhasználókat tárolja), Device (eszközöket tárolja) valamint Observation (az összes megfigyelést tárolja). Az első két említett kollekció sémájában található dokumentumok

szerkezete állandó. Az alábbi információt az előbb említett adatbázis User nevű gyűjteményéből emeltem ki:

```
{
  "_id" : ObjectId("5486ea7fe4b0fd1846deb9a9"),
  "className" : "model.User",
  "email" : "test.lavinia.32@gmail.com",
  "fullname" : "test.lavinia.32@gmail.com",
  "active" : true,
  "comment" : "",
  "password" : "",
  "type" : "user"
}
```

A 3.1. táblázat az előbb látható dokumentum adattagjait ismerteti.

Adattag neve	Típus	Leírás
className	Szöveg	Java osztály neve az alkalmazás kódjában
email	Szöveg	A felhasználó e-mail címe
fullname	Szöveg	A felhasználó neve vagy e-mail címe
active	Logikai	Aktív-e a felhasználó
comment	Szöveg	Felhasználóhoz tartozó leírás
password	Szöveg	A felhasználó jelszava
type	Szöveg	A felhasználó típusa

3.1. táblázat Felhasználók adattagjai

Egy eszköz leírását az alábbi dokumentum szemlélteti:

```
{
  "_id" : ObjectId("5486ea7fe4b0fd1846deb9aa"),
  "className" : "model.Device",
  "name" : "test.lavinia.32@gmail.com",
  "description" : "",
  "owner" : {
    "$ref" : "User",
    "$id":ObjectId("5486ea7fe4b0fd1846deb9a9")
  },
  "hwId":"UUID:4ee6b895-eef3-45ef-bed0-bd027ac0ba07",
  "acls" : [ {
    "user" : {
      "$ref" : "User",
```

```

    "$id":
      ObjectId("52f37af3e4b09a5f6ed287b1")
  },
  "level" : "ro"
}]
}

```

Az eszközök adattagjait a 3.2. táblázat ismerteti.

Adattag neve	Típus	Leírás
className	Szöveg	Java osztály neve az alkalmazás kódjában
name	Szöveg	Az eszköz neve
description	Szöveg	Az eszközhöz tartozó leírás
owner	Referencia	Hivatkozás a tulajdonsora
hwId	Szöveg	Egyedi azonosító
acls	Tömb	Hozzáférési lista

3.2. táblázat Eszközök adattagjai

Az eszközöket tároló dokumentumban két említésre méltóbb adattag is jelen van. Az egyik a tulajdonos, melynek típusa referencia. A MongoDB lehetőséget ad, hogy egyik dokumentumból hivatkozzunk egy másikra. Ez egy beágyazott dokumentum, melynek két adattagja van: a hivatkozott dokumentum kollekciója (*\$ref*) és a hivatkozott dokumentum egyedi azonosítója (*\$id*). Azonban mindig gondolni kell arra is, hogy nem léteznek kényszerek, melyek biztosítják, hogy mindig érvényes hivatkozás szerepel a referenciában. MongoDB adatbázisoknál a kliens oldalon kell ellenőrizni, hogy a referencia tényleg valós dokumentumra mutat-e. Másik érdekesség egy tömb, ami referenciákat és hozzáférési szinteket tárol. Ez szolgált a régi modellben arra, hogy különböző személyeknek megengedjük (például: saját orvosunknak vagy dietetikusunknak, esetleg fogyótársunknak), hogy bejegyzéseinket lássák, esetleg megjegyzéseket írjanak hozzá. Ezt a PostgreSQL adatbázis nem ismeri, így a migráció szempontjából elhanyagolható.

Az egyes megfigyeléseket leíró dokumentumokat ilyen részletesen nem mutatom be, azok nagysága és darabszáma miatt. Az egyes típusokban található adatokat az Entitások című alfejezetben ismertettem. Az összes naplóbejegyzésben

található számos adattag, aminek halmazát nevezhetjük a különböző megfigyelések ösosztályának is.

```
{
  "_id" : ObjectId("5293792387bb696e1214081f"),
  "className" : "model.Observation",
  "extId" : "1152",
  "device" : {
    "$ref" : "Device",
    "$id" : ObjectId("5293785d87bb696e12140439")
  },
  "timestampIn" :
    ISODate("2013-04-14T19:35:09.203Z"),
  "timestampRecorded" :
    ISODate("2013-01-30T20:31:15.224Z"),
  "timestampUpdated" :
    ISODate("2013-04-14T19:35:09.203Z"),
  "type" :
    "hu.uni_pannon.mhealth.dsapi.datatype.MealLogRecord",
  "deleted" : false
}
```

A 3.3. táblázat tartalmazza az adattagok típusát és jelentését.

Adattag neve	Típus	Leírás
className	Szöveg	Java osztály neve az alkalmazás kódjában
extId	Szöveg	Külső azonosító
device	Referencia	Hivatkozás az eszközre
timestampIn	Dátum	Adatbázisba kerülés ideje
timestampRecorded	Dátum	Mérés időpontja
timestampUpdated	Dátum	Utolsó módosítás dátuma
type	Szöveg	Megfigyelés típusa
deleted	Logikai	Törlésre került-e?

3.3. táblázat Megfigyelések ösosztályának adattagjai

A kapott adatbázis sémáján és tartalmán nem változtathattam, hiszen az alkalmazásnak működnie kell az eredeti adathalmazon is. Viszont a gyorsabb futás érdekében index-eket helyeztem el a saját adatbázisomon. Index készítésével az

adathalmazt rendezzük valamilyen adattag vagy adattagok szerint. A rendezett adathalmazon a keresés sokkal gyorsabb művelet. Hátránya viszont, hogy a beszűrés művelet lassulhat. A MongoDB adatbázisban az összes megfigyelés egyetlen kollekcióba kerül, így ennek az adathalmaznak van a legnagyobb mérete. A migrációs folyamat implementálása után tanulmányoztam, hogy milyen lekérdezések vannak elküldve a MongoDB szervernek. Lehetőségünk van tanulmányozni, hogy egy adott lekérdezési műveletet a szerver miképp hajtana végre. Az alábbi példában lekérdezzük a végrehajtási tervet arra a lekérdezésre, ami visszaadná az 5293785d87bb696e12140438 eszköz azonosítóhoz tartozó megfigyeléseket:

```
db.Observation.find({"owner" : {
    "$ref" : "Device", "$id" :
    ObjectId("5293786087bb696e12140442") }})
.explain()
```

A lekérdezés eredménye egy JSON dokumentum. Ez tartalmazza a végrehajtási tervet (melyik kollekcióban, milyen feltételek mellett, stb.), a szerver információt (melyik IP címen található a hoszt, annak melyik portján fut a szerver, verziószám), valamint az alábbi dokumentum részletet:

```
"winningPlan" : {
  "stage" : "COLLSCAN",
  "filter" : {
    "owner" : {
      "$eq" : {
        "$ref" : "Device",
        "$id" : ObjectId(
          "5293786087bb696e12140442" )
      }
    }
  },
  "direction" : "forward"
}
```

A *stage* adattagban a COLLSCAN érték figyelhető meg. Ennek jelentése, hogy a lekérdezés az egész adathalmaz elemeit megvizsgálja. Könnyen belátható, hogy ez a kollekció bővülésével egyre lassabb művelet lesz. A lekérdezések tanulmányozása után azt tapasztaltam, hogy ezt a kollekciót rengetegszer fésüli át

a szerver feleslegesen. A megfigyeléseket mindig a hozzá kapcsolódó eszközök referenciája segítségével kérdezzük le, ezért célszerű lenne egy index elhelyezése a mező alapján. Ezt az alábbi utasítással hoztam létre:

```
db.collection.createIndex( { owner: 1 } )
```

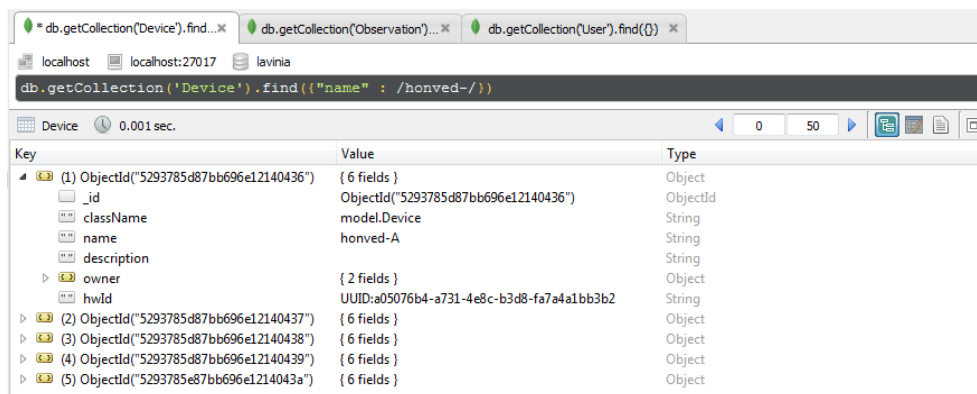
A parancs hatására egy új adatszerkezet jön létre a háttérben, ahol a megfigyelések rendezve vannak az *owner* adattag szerint növekvő sorrendben (ezt az 1-es értékből tudjuk). Amennyiben a végrehajtási tervet ismét lekérjük a szervertől, látható lesz, hogy a *stage* adattag értéke megváltozott, azaz a lekérdezés már nem fésüli át az egész adathalmazt.

3.3. Lokális teszt környezet kialakítása

A tervezési fázisban is már sok tesztadat állt a rendelkezésemre. Rengeteg dokumentációt, adatbázis séma leírást, ábrát kaptam. A szoftver teszteléséhez két adatbázis mentés is elérhető volt számomra.

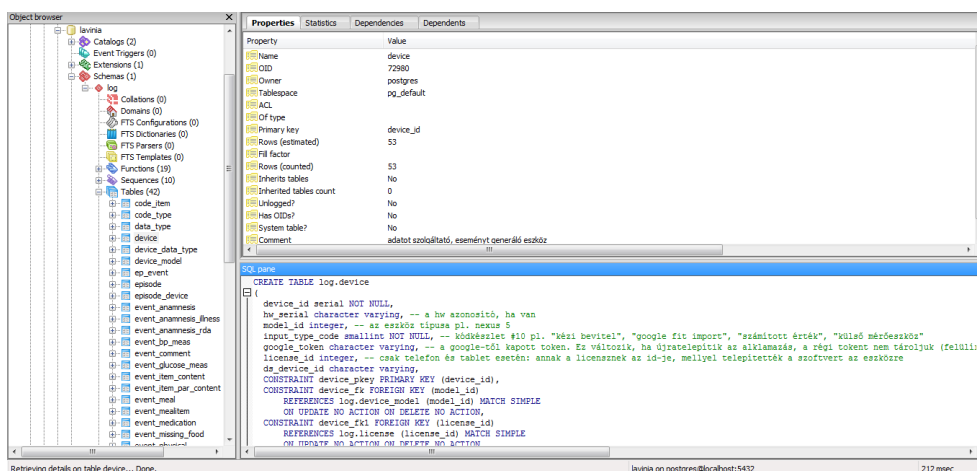
Első feladatomban a két adatbáziskezelő rendszer telepítése volt. Mindkét szoftver ingyenesen letölthető a fejlesztői oldalakról. A telepítés során törekedtem olyan környezetet létrehozni, ami hasonlít a két éles adatbáziséhoz. Telepítés után a két adatbázis tartalmát megfelelő parancsok segítségével parancssorból tudtam böngészni. A kényelmesebb munka érdekében különböző grafikus szoftvereket alkalmaztam az adatok visszakereséséhez. A MongoDB adatbázisban lévő adatok könnyebb kinyeréséhez a Robomongo nevű szoftvert használtam. Segítségével könnyebben lehetett tanulmányozni és visszakeresni az egyes dokumentumokat. A grafikus felületnek köszönhetően a lekérdezések és azok eredményei is sokkal jobban olvashatóak és tagolhatóak.

Környezet ismertetése



3.1. ábra Robomongo felhasználói felülete

A PostgreSQL adatbázisba átültetett adatok visszakereséséhez a PostgreSQL fejlesztői által kiadott ingyenes pgAdmin III nevű programot használtam. Mivel a relációs adatbáziskezelők sokkal régebb óta vannak a piacon, mint NoSQL társaik, így számos fizetős, illetve ingyenes szoftver közül választhattam. Azért döntöttem az említett alkalmazás mellett, mert egyszerű, felülete letisztult, valamint minden olyan funkciót ismer, ami számomra szükséges.



3.2. ábra pgAdmin III felhasználói felülete

A MongoDB adatbázis biztonsági mentése nem más, mint egy JSON dokumentumokat tároló szöveges állomány. Az adatbázisba való visszaállításhoz az alábbi utasítást kell kiadni parancssorból:

```
mongoimport --db lavinia --collection User
--file User.json
```

A fentebb említett parancs a mongoimport.exe folyamatot indítja el, ami a lavinia nevű adatbázis User kollekciójába másolja a User.json

állományban található dokumentumokat. A parancs kiegészíthető további paraméterekkel, például autentikáció, fájlformátum, adatbázis elérése, stb. Kényszerek híján a fájlban található adatszerkezeteket vagy a sémát nem kellett átalakítani. A további két kollekció importjához hasonló parancsot kellett használnom.

A relációs adatbázis sémájának létrehozásához egy DDL (Data Definition Language – magyarul: SQL adatdefiníciós nyelv) utasításokból álló szöveges állományt kaptam. A parancsok szekvenciális futtatása során különböző hibákat észleltem, így kisebb módosításokat kellett végrehajtanom a kapott állományon. Először azokat az idegen kulcs hivatkozásokat létrehozó utasításokat kellett törölnöm, amik más sémában található táblák mezőire hivatkoztak. Ennek oka, hogy a relációs adatbázisnak csak azt a sémáját kaptam meg tesztelésre, ami a dolgozatom témája. Utána az egyes táblák összerendeléseit töröltem azokkal a felhasználókkal, akik nem szerepelnek az én adatbázisomban. Ezek a módosítások nem befolyásolják a migrációs folyamatot, csupán az éles rendszerben játszanak szerepet.

3.4. Dokumentumtár tartalma

Forrásként kaptam egy 2015. márciusában készült mentést az éles MongoDB adatbázis tartalmáról. Ezen adatok között nem szerepelt a klinika kísérletekben részt vevő személyek azonosítására alkalmas adat.

A sémák megismerése után feltérképeztem az adatbázisban található adatokat. Első körben az egyes típusok darabszámát vizsgáltam. A 3.4. táblázatban látható a dokumentumok darabszáma típusonként.

Entitás	Darabszám
Felhasználó	197
Eszköz	194
Megfigyelés	54 352

3.4. táblázat Dokumentumok darabszáma kollekciónként

Ahogy a fentiekben ismertettem az összes megfigyelés egy kollekcióban szerepel a MongoDB adatbázisban. Erre utal a táblázatban kiugró nagy szám. Az

alábbi MapReduce parancs futtatása után megkaphatjuk, hogy pontosan milyen megfigyelésből hány darab van:

```
db.Observation.mapReduce(
  function() {
    emit(this.type, 1);
  },
  function(typeid, values) {
    return Array.sum(values);
  },
  {
    out: "observation_count"
  }
);
```

A MapReduce technika előnye, hogy nagy adathalmazon párhuzamosan tudunk adatot feldolgozni, ami gyorsítja a működést. Két része van: map, valamint a reduce. Az első funkció a szűrést és a rendezést végzi el, míg az utóbbi összegzi ezek az eredményeit. Az előbbi parancs lefuttatása után az aktuális adatbázisban kapunk egy új kollekciót *observation_count* néven. Az ebben található dokumentumok *_id* mezőinek értéke az egyes megfigyelések típusa. Minden dokumentumban található még egy *value* mező is, melynek értéke az aktuális megfigyelés darabszáma az adatbázisban. A teszt adatbázison lefuttatva a 3.5. táblázatban látható eredményeket kaptam.

Megfigyelés neve	Darabszám
Vércukorszint mérés	4 844
Vérnyomás mérés	706
Dietetikai anamnézis	2 832
Laboreredmény	216
Étkezés	37 508
Gyógyszerezés	3 930
Hiányzó élelmiszer	57
Fizikai aktivitás	3 402
Testsúlymérés	857

3.5. táblázat Különböző megfigyelések száma

Az itt megismert számok a tesztelésnél hasznosak, hiszen a migrációs folyamat futtatása után a PostgreSQL adatbázisban pontosan meg kell egyezni az egyes entitások darabszámának.

Az anamnézis megfigyelést egyszer kell tárolni felhasználónként. A 3.5. táblázat alapján látható, hogy sokkal több ilyen entitás van, mint regisztrált felhasználó. A hiba oka, hogy amikor a felhasználó módosítja ezen adatok valamelyikét, akkor új bejegyzés készül róla. A migrációs folyamatban le kell kezelni, hogy amennyiben köthető az adott felhasználóhoz ilyen esemény, úgy csak időben az utolsót kell másolni.

4. Specifikáció

A következőkben a szoftverrel szemben támasztott követelményeket mutatom be. Röviden ismertetem az egyes feladatokat, majd a következő fejezetekben részletesen kifejtem és megindokolom a kész rendszerbe került funkcionálisok működését és tulajdonságait.

A már bemutatott entitások másolását egyenként meg kell tervezni. Az eltérő nevű, de funkcionálisban ugyanazt a szerepet betöltő bejegyzéseket össze kell párosítani. Ugyanez a feladat az egyes dokumentumok adattagjaival. Nem minden esetben egyezik meg a forrás és cél adatbázis adott adattagjának az értékkészlete, ebből fakadóan az adatok tartalmát, típusát vagy felépítését szükség esetén módosítani kell. Ezeket a problémákat figyelembe véve kell minden entitásra a másolási folyamatot megtervezni és implementálni.

A migrálási folyamat a különböző entitások másolásának összessége. A folyamatot számos más funkcióval is bővíteni kell. Gondolni kell a migrációs folyamat közben fellépő hibákra. Ilyenek lehetnek például a hivatkozási integritások vagy egyéb megszorítások megsértése. Elengedhetetlen olyan mechanizmust kidolgozni, mely képes a lehető legtöbb hibát kezelni. Lehetőség szerint vonjuk be a felhasználót, aki befolyásolhatja a migrációs folyamat további műveleteit.

Az említett hibákról és fontosabb eseményekről készüljenek naplók, melyek segítségével a hiba pontos oka, típusa és ideje könnyen visszakereshető. Egy naplón belül csak egy futtatás eredménye szerepeljen. Ez tárolódjon a fájlrendszeren egy szöveges állományban, melynek neve utal a folyamat elindításának idejére. A naplókban belül található bejegyzések tartalmazzanak minél több információt a bejegyzés keletkezésének okairól. Ezek segítségével egy esemény gyorsan és könnyen beazonosítható és visszakereshető.

A könnyebb felhasználói interakciók kezelése érdekében a programnak legyen grafikus felhasználói interfésze. Ezen keresztül a migrálási folyamatot elindító felhasználó könnyebben be tudja konfigurálni a másolási folyamatot, valamint áttekinthetőbb összképet kaphat az eredményről. Ennek első lépése a vázlatos felület-tervek elkészítése. Egy jól megtervezett és elkészített grafikus felület növeli a program használhatóságát.

Amennyiben a program már rendelkezik grafikus felhasználói felülettel, úgy célszerű a migrációs folyamatot valós időben közvetíteni a felhasználó felé. Legyen látható a folyamat aktuális állása, a hátralévő adatmennyiség nagysága, valamint a fontosabb események. A folyamatot lehessen biztonságosan leállítani is. Ilyenkor gondolni kell az inkonzisztens állapot elkerülésére, ezért a folyamatot gondosan meg kell tervezni.

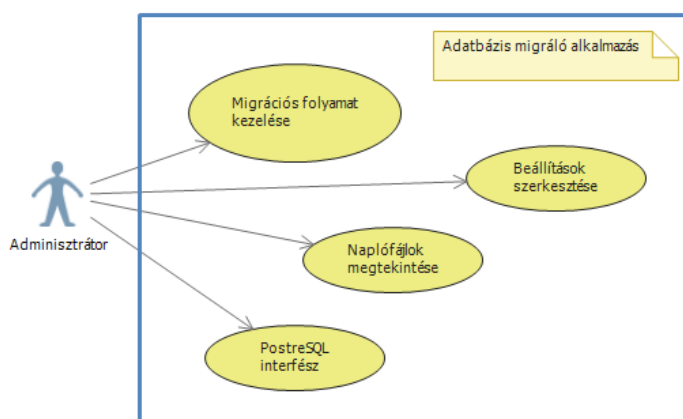
Mivel a program többször is elindítható, azért a felhasználó képes a migrációs folyamatot többször is futtatni. Gondoskodni kell, hogy egy adott entitást csak az első migrációs alkalomkor kell csak másolni, azaz ne történjen újraírás. Ennek hiányában egy felhasználó többször is szerepelhet a cél adatbázisban, ami futás idejű hibákat okozhat a rendszerben.

A folyamat ellenőrzése képpen a rendszert tesztek sorozatának kell alávetni. Ehhez egy adatbázismentés áll rendelkezésre, mely elegendő információt tárol egy kisebb migrációs folyamat szimulálására, mivel minden entitásból több példányt is tartalmaz. Amennyiben ezen adathalmazt sikeresen dolgozza fel a folyamat, ki kell próbálni az éles adatbázisokon is a folyamat működését.

5. Rendszerterv

A tervezés kihagyhatatlan lépés a szoftver implementálása előtt. Meg kellett tervezni a migrációs folyamatot és az azt vezérlő felhasználói felületet, alfolyamatokat, funkciókat.

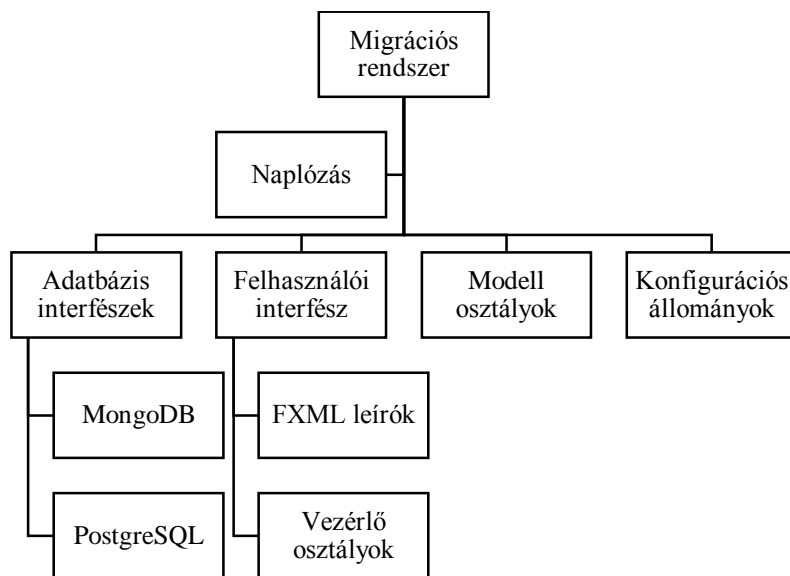
A migrációs alkalmazás kifejezetten az adatbázis adminisztrátoroknak készült. Más aktor nem kap szerepet a rendszerben. Az 5.1. ábrán látható UML (Unified Modelling Language) használati eset diagram a rendszer főbb használati eseteit mutatja be.



5.1. ábra Használati eset diagram

Az 5.1. ábrán az alkalmazás fő komponensei láthatók. Ezek írják le a rendszer által végrehajtandó funkcionálisokat.

Figyelembe véve az objektumorientált elveket, olyan komponenseket kellett tervezni, melyek később könnyedén újra fel lehet használni. Fontosnak tartottam, hogy minden komponens jól különöljön el egymástól, ne legyenek függőségek közöttük. Az 5.2. ábra a komponensek hierarchiáját mutatja be.



5.2. ábra Komponens hierarchia

5.1. Megvalósítási lehetőségek

Egyik mód az adatbáziskezelő rendszerek által nyújtott export és import lehetőség használata lett volna. Amennyiben ezen az úton indultam volna el, úgy első lépésben egy export-ot vagy egy biztonsági másolatot kellett volna készítenem a MongoDB megfelelő adatbázisának aktuális állapotáról. Ez lehet egy JSON (JavaScript Object Notation) formátumú szöveges fájl vagy egy ugyanezeket az információkat tároló bináris állomány. Ezt egy szoftver vagy egy hosszabb script segítségével át lehet alakítani SQL utasításokká. Az átalakított állományt a relációs adatbáziskezelők fel tudják már dolgozni, így a PostgreSQL import lehetőségét kellett volna kihasználnom. Az import/export funkciók könnyen kezelhetők pár soros parancssori hívásokkal. Ennél a módszernél hátrányként értékeltem, hogy nem élhetek az adatbázisok CRUD (Create, Read, Update, Delete – magyarul: Létrehoz, Olvas, Módosít, Töröl) műveleteivel, hanem egy adott, szekvenciális fájlfeldolgozást kellett volna implementálnom. A fájlnek a feldolgozása sem lett volna egyszerű. Adatbázis tervezésnél egy relációs adatbáziskezelő rendszerrel normalizáljuk a relációkat, addig MongoDB adatbázis tervezésnél éppen ellenkezőleg, denormalizáljuk, azaz egymásba ágyazzuk az egyes entitásokat. Előnye lett volna a gyorsaság, valamint az, hogy egy ilyen fájlstruktúra átalakítás egy egyszerűbb script-el is implementálható lett volna.

Másik megoldási lehetőség lett volna egy ETL folyamat implementálása. Az ETL rövidítés a következőt jelenti: Extract, Transform, Load (magyarul: Kinyerés, Átalakítás, Betöltés). A három szó arra utal, hogy az ilyen folyamatokat három fő részre bontjuk. Az elején egy adott erőforrásból ki kell olvasni és szelektálni kell a számunkra szükséges adatokat, ez a kinyerés. Az így kinyert adatokat át kell alakítani szabályokon keresztül, hogy megfeleljenek az üzleti logikának, igényeknek. Az így átalakított adathalmazt be kell tölteni a kívánt adatbázisba, adattárházba. ETL folyamatok létrehozására is számos szoftver áll rendelkezésünkre, mint például: Oracle Warehouse Builder, SAP Data Services, IBM Infosphere Information Server, SQL Server Integration Services, Talend, stb. A listából kiemelkedik a Talend nevű szoftver, mert beépített lehetőség van MongoDB adatbázisokhoz való kapcsolódásra. ETL folyamatokat általában könnyen és gyorsan lehet létrehozni, ami elég nagy pozitívum a többi lehetőséggel szemben. Azonban nem ezt a módszert választottam, mert az adatok szerkezete elég komplex, valamint a forrás és cél struktúrák jelentősen eltérnek egymástól. Az ekkora méretű feladatokhoz az ingyenes szerkesztők már nem felelnek meg, a fizetős rendszerek pedig drágák. Az ár mellett negatívum, hogy általában nem tartalmaznak automata hibakezelést és visszaállítást.

A harmadik választásra esett a döntésem. Ez egy migráló alkalmazás írása valamilyen programozási nyelven. A lehetőségek közül talán ez a leghosszabb, valamint a legtöbb energiát követelő megoldás. Ugyanakkor ez a legjobban testreszabható módszer, hiszen bármilyen funkcionalitást a forráskódból vezérelni lehet. Az adatokat könnyen ki lehet nyerni egyszerű lekérdezésekkel. A struktúra ismeretében ki lehet válogatni az adatokat, ami után egy rendszerezés történik. Az adatok átalakítása után a céladatbázisba is nagyon egyszerű menteni egy kapcsolat segítségével. Előnyei közé tartozik, hogy saját hibakezelés is definiálható a folyamat mögé. Ez számos esetben jól jöhet, például amikor különböző entitásokhoz különböző hibakezelés tartozik. A folyamat naplózása is teljes mértékben konfigurálható. Lehetőségünk van mindenről naplóbejegyzést készíteni, de természetesen szűkíthetjük a kört csak fontos tranzakciókra is. Hátrányként említettem már, hogy ehhez a megoldáshoz kell a legtöbb idő és energiabefektetés.

Fontos megemlíteni, hogy egy MongoDB-ben tárolt egyszerű dokumentum is lehet összetett. Tartalmazhat listákat, aldokumentumokat, de akár listát aldokumentumokról is. Az ilyen adatszerkezeteket egy relációs adatbáziskezelő rendszer nem tudja értelmezni. Ehhez az ilyen adatszerkezeteket szét kell bontatni, hogy megfeleljen a relációs adatbáziskezelő rendszerek tervezésénél megfogalmazott harmadik normálformának. Az ilyen szintű adatfeldolgozásra már célszerű létrehozni egy transzformációs réteget, ami az adatok szétbontását végzi. Ezt legegyszerűbben az általam választott, saját kód írásában lehet implementálni.

5.2. Migrációs folyamat tervezése

A folyamatot le lehet egyszerűsíteni három különböző egyed típusra: felhasználók, eszközök, megfigyelések. A jelenlegi adatbázisban referenciával hivatkoznak egymásra a dokumentumok, ami egy laza kötés, hiszen a referencia mögött nem biztos, hogy létezik dokumentum.

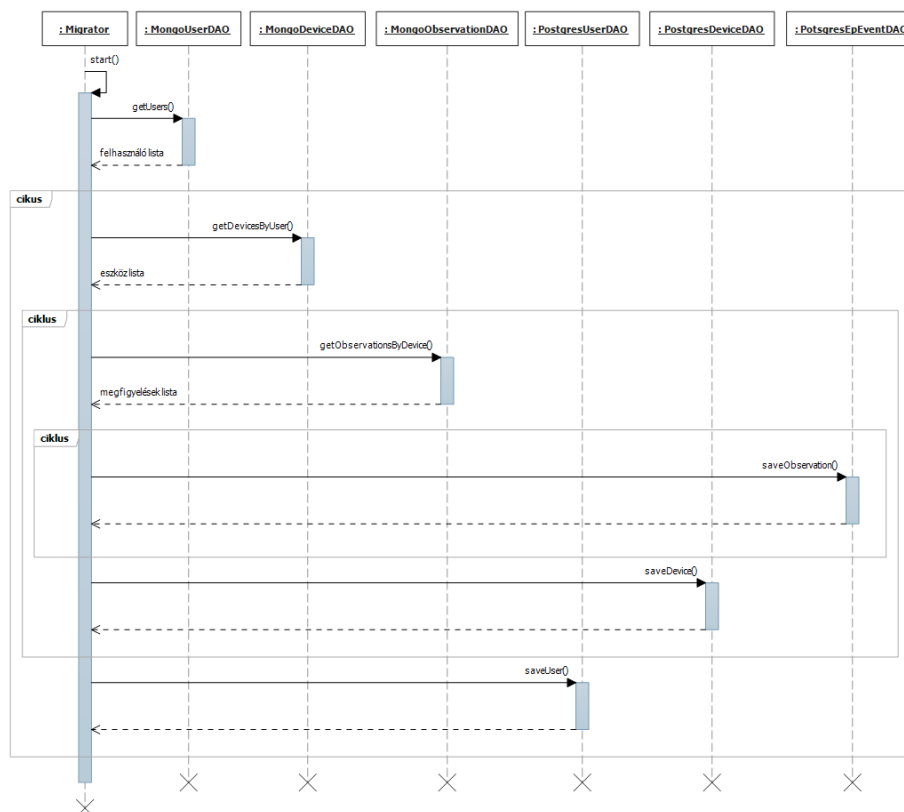
Első feladatomban az adatok sorrendjének megtervezése volt. Az adathalmaz nagysága miatt már ebben a szakaszban is gondolnom kellett az idő és a memória tényezőkre. Első ötletem a szekvenciális másolás volt, azaz a három típus entitásait egymástól függetlenül másoljuk át a forrás adatbázisból a cél adatbázisba. Ez azt jelenti, hogy a memóriában egyszerre csak egy adott típus egyedei szerepelnek, melyek a megfigyelések kivételével csekély méretűek. Utóbbinál gond lehet, hiszen ebből van a legtöbb elemszámú dokumentum. Az idő tekintetében az egyik leggyorsabb, valamint minimális MongoDB utasítást használ fel. Hibakezelés és tranzakcionális szempontból rossz. Az egyes megfigyeléseknél nincs rá garancia, hogy a hozzá tartozó eszköz vagy felhasználó sikeresen volt-e másolva. Ennek lekérdezéséhez egy PostgreSQL utasítás szükséges. Könnyen belátható, hogy pontosan annyi ilyen ellenőrzés kell, ahány darab megfigyelést át szeretnénk másolni. Ez drasztikusan lelassítaná a migrációs folyamatot.

A kész rendszerben egy másik modellt alkalmaztam. A specifikációnak eleget téve különböző szintű tranzakciókat kell megfogalmazni. Ez úgy érhető el, hogy a másolás nem típusonként, hanem entitásonként hajtódik végre. A program a MongoDB szervertől lekérdezi az összes felhasználót, majd ezen iterál végig a következő képpen: mindegyik felhasználóhoz lekérdezi a hozzá tartozó eszközöket.

Az eszközök listájának birtokában a hozzá tartozó megfigyeléseket kell összegyűjteni. Ezeket már lehet másolni a cél adatbázisba. Az utolsó megfigyelés után az adott eszközt is véglegesíteni kell, valamint az utolsó eszköz után a felhasználót is. A memóriában mindig egy felhasználóhoz tartozó adatok (eszközök és azok megfigyelései) találhatók. Idő tekintetében hasonló eredményeket ér el, mint az első modell. Összefoglalva a migrációs folyamat vázát az alábbi pszeudókód mutatja be:

```
ciklus a felhasználók listáján
  tranzakció létrehozása
  eszközök := felhasználó eszközeinek lekérdezése
  ciklus az eszközök listáján
    megfigyelések := eszközökhöz megfigyelései
    ciklus a megfigyelések listáján
      megfigyelése mentése
    ciklus vége
    eszköz_mentés := eszköz mentése
  ciklus vége
  felhasználó_mentés := felhasználó mentése
  tranzakció véglegesítése
ciklus vége
```

Ugyanezt a folyamatot az 5.3. ábrán látható UML folyamatábrán is megtekinthetjük.



5.3. ábra Migrációs folyamat szekvencia diagramja

5.3. Adatbázissal való kommunikáció

Az egyik kritikus pont a két adatbázissal való kapcsolat felvétele, valamint az adatok mozgatása ezeken keresztül. Az egyik lehetőség egy ORM (Object-Relational Mapping – magyarul: objektum-relációs leképezés) réteg bevezetése lett volna. Ennek fő feladata, hogy az adatbázisban található entitásokból objektumokat készít, amit az aktuális programozási nyelvben könnyebben lehet kezelni. Előnye, hogy csökkenti a megírandó kód mennyiségét, valamint az egyszerűbb manipulációk is könnyebben kivitelezhetők. Hátrányként meg kell említeni, hogy kevésbé teljesít jól nagyobb adathalmaz és több tábla összekapcsolása esetén. Mivel már kész sémát kaptam, illetve sok adatot tároló adatbázissal kellett dolgoznom, így ezt a lehetőséget korán elvetettem.

A PostgreSQL adatbázis kommunikációhoz a Java nyelv által nyújtott JDBC API-t (Java Database Connectivity Application Programming Interface) használtam. Ez a nyelv által definiált felület, ami segít felvenni a kapcsolatot bármilyen relációs adatbáziskezelő rendszerrel, valamint segít a parancsok elküldésében és a visszaérkező adatok fogadásában és feldolgozásában. Fontos

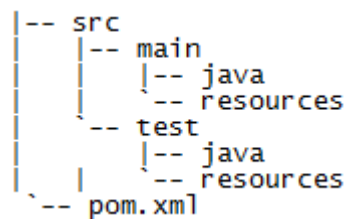
megemlíteni, hogy ez nagy részben interfészek halmaza, ezeknek a megvalósítása az aktuális adatbázishoz tartozó driver függőség. A kapcsolódás módjának megtervezése után már csak egy dolgot kellett eldönteni: SQL parancsok sorozatát küldöm el az adatbázis szervernek vagy a szerveren tárolt eljárásokat hívom meg paraméterekkel. Mindkét lehetőség számos előnnyel jár, így a döntés nem volt egyértelmű. Azért döntöttem az SQL parancsok küldése mellett, mert az említett JDBC API könnyen testreszabható, így az adatbázisszerver által visszaadott eredményhalmaz metaadataiból sok információ kiolvasható és felhasználható (például a beszűrt elem elsődleges kulcsa, amennyiben azt a szerver generálta). Hátránya, hogy szinte mindegyik függvényhívás ellenőrzött kivételt válthat ki, amit kötelezően kezelnem kell vagy tovább kell dobnom. Az utolsó megoldás nem használható, hiszen a program is leállhat tőle. Viszont ezeket sikerült előnyösen felhasználnom, mégpedig a hibakezelésben. Ezt az Implementáció című fejezetben mutatom be részletesen.

Mivel a MongoDB nem relációs adatbázis, így ezen az oldalon a JDBC API nem kompatibilis. Bár a MongoDB-hez is létezik ORM réteg, mégsem azt alkalmaztam, hanem a már régen kiadott hivatalos API-t. Ennek oka, hogy sokkal stabilabb, valamint használata sokkal könnyebb azok számára, akik ismerik a MongoDB-s parancsokat.

5.4. Forrásfájlok könyvtárszerkezete

A kódok könnyebb fejlesztése és megértése érdekében célszerű az egyes forrásfájlokat funkcionalitás szempontjából csoportosítani és könyvtárakba sorolni. A Java ezeket a mappákat package-nek (magyarul: csomagnak) hívja. Jól szervezett struktúránál egy mappában belül található állományok egy funkcióhoz tartoznak.

A felhasznált technológiában röviden ismertettem az Apache Maven szoftvert. Egyik hasznos funkcionalitása, hogy automatikusan generál egy könyvtárszerkezetet, amibe a szoftver működéséhez szükséges állományokat kell elhelyezni. Üres projekt létrehozása esetén az 5.4. ábrán található könyvtárszerkezetet kapjuk meg.



5.4. ábra Maven könyvtárszerkezet

Az *src* könyvárba kerülnek a forrásfájlok, azaz a kódok és a statikus erőforrások (fájlok, képek, stb.). A *pom.xml* fájl írja le a szoftvert, paramétereit, függőségeit. Ez mindig a generált könyvtár gyökerében található. A *src/main/java* könyvtár a Java forráskódot tartalmazza. Természetesen ezen belül is lehetnek újabb könyvtárak. A következő funkciók szerint válogattam szét a forráskódot: adatbázis interfész, felületet vezérlő osztályok, migrációhoz szükséges osztályok, segédfüggvényeket tároló osztályok, egyéb. Az ezekhez tartozó forrásállományok mind-mind külön mappában szerepelnek. A *src/main/resources* könyvárban a statikus fájlok kaptak helyet. Ilyenek a felhasználói felületet leíró XML dokumentumok, képek, különböző keretrendszereket konfiguráló állományok, valamint az alapértelmezett beállításokat tartalmazó fájl. Az *src/test* könyvtár hasonló szerkezettel rendelkezik, mint az előbb bemutatott main mappa. Itt az automata tesztek kódjai, valamint az ehhez szükséges erőforrások kapnak helyet.

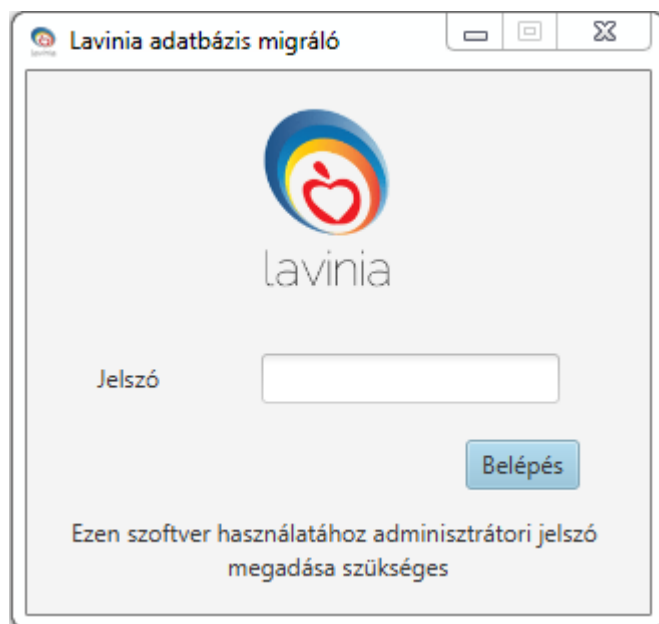
6. Implementálás

A tervezés után a következő lépés az implementálás. Ebben a fejezetben a főbb funkciókat fogom ismertetni, valamint az okokat, hogy miért éppen így oldottam meg a felmerülő problémákat.

6.1. Bejelentkezés

Az alkalmazás indulása után felépíti a kapcsolatokat az adatbáziskezelő rendszerekkel. Az elérhető funkciók között van egy felület, mely bármilyen SQL utasítást lefuttat a PostgreSQL adatbázison. Ehhez korlátlan hozzáférésre van szükség, ezért az alkalmazást éppen futtató felhasználó hozzáfér a teljes aktuális adatbázishoz. Biztonsági szempontból egy alapszintű védelmi mechanizmust építettem ki annak érdekében, hogy a szoftvert csak egy meghatározott jelszó ismeretében lehessen használni.

Az alkalmazás indulása után a 6.1. ábrán látható felület jelenik meg. Itt kell megadni a továbblépéshez szükséges jelszót. Helyes érték megadása esetén a felhasználói felületen az adminisztrátoroknak szánt funkciók jelennek meg, míg hibás érték esetén hibaüzenetet kapunk és naplóbejegyzés generálódik.



6.1. ábra Bejelentkezési felület

A jelszót az alkalmazás forráskódjában helyeztem el. Ennek előnye, hogy rendkívül könnyen hozzáférhető, valamint minden esetben rendelkezésre áll.

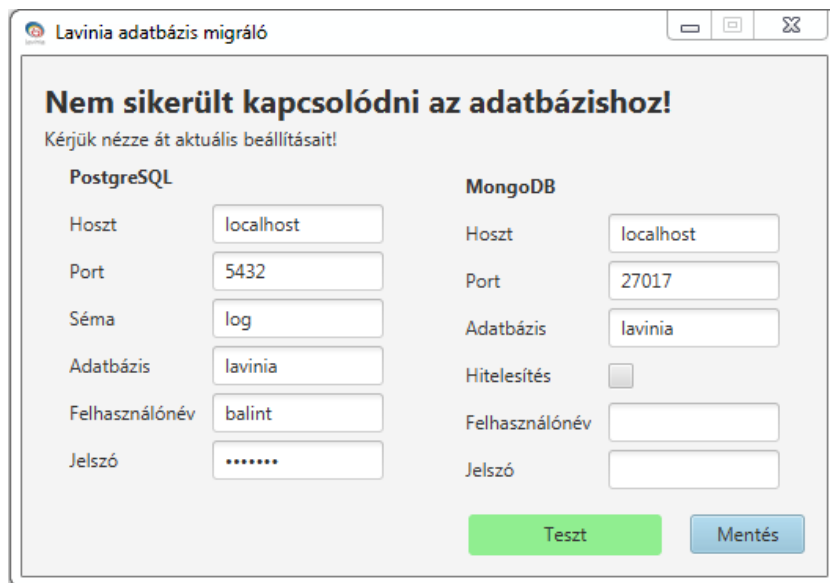
Hátránya, hogy megváltoztatni a kódban kell, ami az aktuális kód újrafordítását is maga után vonja.

6.2. Inicializálás

A program minden elindulásnál ellenőrzi, hogy rendelkezésre állnak-e a neki szükséges erőforrások. Amennyiben valamelyik hiányzik, azt megpróbálja elérhetővé tenni. E szerint két csoportot lehet megkülönböztetni.

Az egyik azok az erőforrások halmaza, amelyeket saját maga valamilyen szinten képes pótolni. Jó példa erre a beállításokat tartalmazó szöveges állomány, ami a programmal egy könyvtárban található. Ha ez az állomány nem létezik, esetleg nem hozzáférhető, akkor a program egy alapértelmezett beállításokat tartalmazó fájlal helyettesíti, amit létre is hoz az adott könyvtárban. Ezek az előre megírt beállítások a csomagolt JAR fájlban találhatóak, ezért ezen módosítani nem lehet. Természetesen a másolata már nem csomagolt, így bármikor változtathatunk a tartalmán. Ebbe a kategóriába tartoznak még a naplókat és biztonsági mentéseket tartalmazó könyvtárak. Amennyiben ezek nem állnak rendelkezésre, úgy a program az inicializálás során automatikusan létrehozza.

Az erőforrások hiányának másik felét a program már nem képes automatikusan pótolni. Ez nem más, mint az adatbázisokhoz való kapcsolódási paraméterek. Amennyiben az utoljára beállított értékekkel nem sikerül felvenni a kapcsolatot az adatbázis szerverrel, úgy a program induláskor elkéri a paramétereket, amit a 6.2. ábrán látható.



6.2. ábra Inicializáló felület sikertelen kapcsolódás esetén

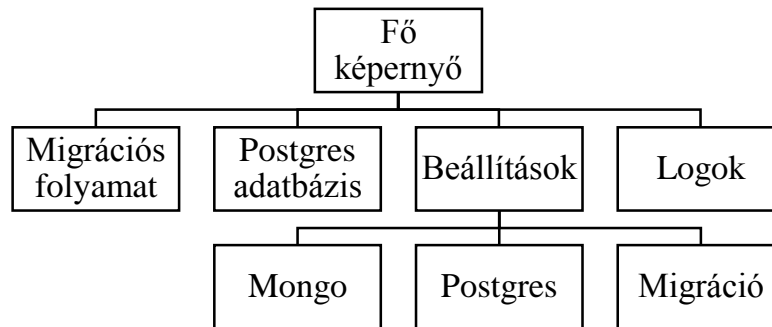
A felületről csak akkor léphetünk tovább, ha olyan értékeket adunk meg, melyek segítségével a két adatbázis elérhető. A „Teszt” feliratú zöld gomb megnyomásával lehetőségünk van a mezőkben található paraméterek tesztelésére. A „Mentés” gomb megnyomása után a rendszer automatikusan leteszteli a megadott értékeket, majd amennyiben eléri az adatbázisokat, úgy továbblépteti a felhasználót a grafikus felületen.

6.3. Felhasználói felület

A felhasználói felület teljes egészében a JavaFX keretrendszeren alapul. Előnye, hogy a grafikus felületeket egyszerű XML dokumentumban is le lehet írni. Ilyen dokumentumok szövegszerkesztő segítségével is készíthetők, nem szükséges hozzájuk Java programozási ismeret. A Scene Builder program segítségével Drag & Drop módszerrel lehet akár komplex GUI-kat (Graphical User Interface, magyarul: grafikus felhasználói felület) is létrehozni. Ezeket hívjuk FXML fájloknak, és ez a programunk „Nézet” része az MVC tervezési mintának megfelelően.

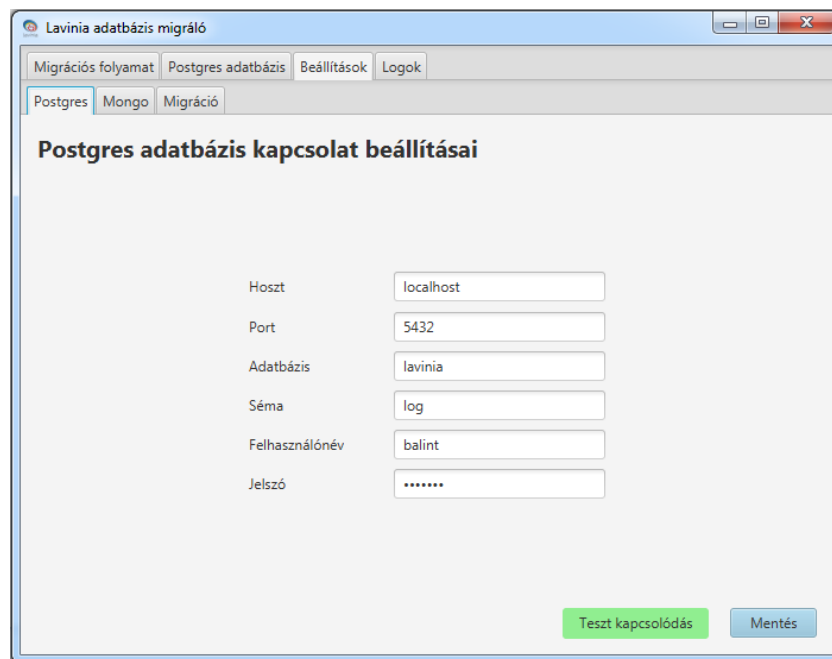
Fontos a felhasználó felület kialakítása, hiszen a felhasználó ez alapján fogja megítélni először a szoftvert. A felületen tabok, azaz fülek között lehet választani. Mindegyik fül egy-egy funkcionálisitást takar. Az összetettebb funkcióknál a felület

további fülre oszlik fel a könnyebb navigálás érdekében. Az elrendezést a 6.3. ábra mutatja.



6.3. ábra Funkciók elhelyezése a felhasználói felületen

Látható, hogy a beállítások felület további három fülre oszlik. A grafikus felületen az elrendezést a 6.4. ábra szemlélteti.



6.4. ábra PostgreSQL adatbázis kapcsolat paramétereinek beállítása

A hierarchikus elrendezésnek köszönhetően nem csak könnyen értelmezhető a felület, hanem programozási szempontból is van egy pozitívuma. Mindegyik felülethez rendeltem egy controller osztályt, ami a felület vezérlését bonyolítja le. Mivel mindegyik controller ismeri saját komponenseit, ezért inicializáláskor elegendő volt csak a fő képernyő inicializáló függvényét meghívni. A többi controller és nézet e függvényét a szülő osztály hívja meg, így egy rekurzív inicializálás történik. Ennek előnye, hogy sokkal kevesebb kódra van szükség,

valamint új felület hozzáadása esetén csak a szülő osztályban egyetlen sorral kell bővíteni a meglévő kódot.

6.4. Migrációs folyamat

A szoftver központi eleme a migrációs folyamat, így ezt részletesen is ismertetni szeretném. Erre a problémára való megoldás megtalálása tette ki a tervezési fázis túlnyomó részét. A hatékony működéshez kulcsfontosságú, hogy ez a folyamat a lehető legjobban legyen megtervezve. Gondolni kell az adathalmaz nagyságára, ami nagy befolyással lehet a memóriára és a processzorra, ezek által pedig az egész szoftverre. A megoldásom nem biztos, hogy optimális, de érvekkel alá fogom támasztani, hogy miért ezt az utat választottam.

A folyamat implementációja teljesen elkülönül a többi komponenstől. Ennek előnye, hogy a szoftveren belül nincs más modullal semmilyen függősége, így kisebb módosítások után teljesen kiemelhető és beilleszthető más rendszerbe. A kisebb módosítások alatt a naplózást és a hozzá kapcsolódó hívásokat értem. Az implementációban egy tervezési minta figyelhető meg, nevezetesen a Singleton (magyarul: Egyke). Ennek tulajdonsága, hogy az így tervezett osztályokból a program futása közben maximum egy objektum létezhet. Biztonsági okok miatt terveztem így, hogy ne lehessen egymással párhuzamos migrációs folyamatokat elindítani. A csatlakozás ezen tervezési mintát követő osztályokon keresztül valósult meg. Ez biztosított arról, hogy a program és az egyes adatbáziskezelő rendszerek között pontosan csak egy kapcsolat lehessen.

Amikor a felhasználó a migrációt elindító gombra kattint, akkor a program lekéri az egyetlen példányt, ami a migrációs folyamatot kezeli. Az egész folyamat egy új szálon indul el. Többszálú programoknak nagyon sok előnyük van, viszont ezek megvalósítása és tesztelése sokkal nehezebb. Előnye, hogy így a folyamat kihasználhatja a rendelkezésre álló processzor több magját is, ami hatékonyabb, gyorsabb futást eredményez. Egy másik látható eredménye is van a többszálú programnak. Amíg a migrációs folyamat fut, addig a felhasználói felület is elérhető. Ezzel ellentétben az egyszálú programoknál ilyenkor a felhasználói felület nem reagál további akciókra. A szálkezeléshez a JavaFX Concurrency csomag által kínált lehetőségeket használtam. Ennek egyik legfőbb előnye, hogy a folyamatokat

és a felhasználói felületen elhelyezett komponenseket könnyen össze tudom kapcsolni.

Szót kell ejteni a szálak egymás közötti kommunikációjáról is. A felhasználó valós időben visszajelzést kap az egyes entitások átvitelének sikerességéről, valamint az esetlegesen fellépő hibákról. Belátható tehát, hogy a migrálást végző szál nagyon sok adatot küld a program fő szálának. Visszafele már nem kell adatot küldeni, hiszen a folyamat csak akkor indul el, ha rendelkezésre áll az összes szükséges információ. A kapott adatokat már csak el kell helyezni a felhasználói felületen. A JavaFX szabálya (és ez sok egyéb felhasználói felületet tartalmazó keretrendszerre igaz), hogy a felhasználói felületet módosítani csak az alkalmazás fő szálán (amin a JavaFX alkalmazás fut) lehet. Vannak olyan grafikus komponensek, amiknek egyes változóit össze lehet kötni a migrálást végző szál belső változóival. Ilyen például a folyamat állapotát jelző komponens. A többi adat megjelenítésére viszont más technikát kellett választanom. A JavaFX bevezette *Observable* interfészt és azok különböző implementációt. Az ebből származtatott osztályok példányaira lehetőségünk van megfigyelőket tenni. Ez az Observer (magyarul: Megfigyelő) tervezési mintához hasonlít. A tárolni kívánt adatot egy ilyen objektumba csomagoljuk. Ha a csomagolt adat értéke megváltozik, akkor a feliratkozott megfigyelők erről értesítést kapnak. Ezzel a lehetőséggel élve már tudok adatot tárolni és változásokról üzenetet küldeni a migrálást végző szálról a JavaFX szálaknak. Mivel a változások a másik szálon történnek, ezért egyszerűen a felhasználó felületre kitenni nem lehet az ismertett szabály miatt. A keretrendszer lehetőséget ad, hogy kódrészleteket a program fő szálán hajtsuk végre. Ehhez a *Platform.runLater* nevű metódust kell használni, ami paraméterként egy *Runnable* típust vár (ez lehet lambda kifejezés, metódus referencia vagy egy ezt implementált osztály is). A felület módosítása innentől már könnyű. A megfelelő változóhoz egy megfigyelőt kell hozzárendelni, ami a változást a JavaFX szálon – az előbb említett függvény segítségével – a felületre továbbítja.

A folyamat központi része a másolás, ami tranzakciók segítségével történik. A Rendszerterv című fejezetben bemutatott modellben a fellépő hibát a felhasználókon iteráló ciklusban kell kezelni. Megfigyelhető, hogy a mentések sorrendje ellentétes a tényleges beszúrásokkal. Azaz, ahhoz, hogy egy megfigyelést

be tudunk szűrni a relációs adatbázisba, annak már tartalmaznia kell az eszközt. A probléma egyik megoldási módja a tranzakció-kezelés. A JDBC API alapértelmezett tranzakció típusa mellett a kiadott SQL parancsokat azonnal végrehajtja az adatbázison. Lehetőségünk van manuálisra állítani, ekkor kézzel kell meghívni a *commit* függvényt, ami az addig bejegyzett SQL utasításokat véglegesíti az adatbázison. A kész verzióban nem csak felhasználói szintű, hanem eszköz és megfigyelés szintű tulajdonságot is be tudunk állítani. Ehhez a JDBC API-ban található *Savepoint* funkcionalitást használtam ki. Röviden összefoglalva lehetőségünk van úgynevezett *Savepoint*-okat elhelyezni a kódban, amire hiba esetén a tranzakciót vissza tudjuk gördíteni. Ezt nem minden adatbáziskezelő-rendszer támogatja, de a PostgreSQL igen. A tranzakción meghívott *commit* függvény hatására ezek a mentési pontok eltűnnek. Mivel ez a hibakezelés egyik központi eleme, így ezt példával is ki szeretném emelni:

```
Connection postgresConnection =
    getActiveConnection();
postgresConnection.executeQuery(
    "INSERT INTO log.user(firstname, family_name)
    VALUES ('User, 'Test')");
Savepoint savepoint1 =
    postgresConnection.savePoint();
// többi SQL INSERT utasítás
postgresConnection.rollback(savepoint1);
postgresConnection.commit();
```

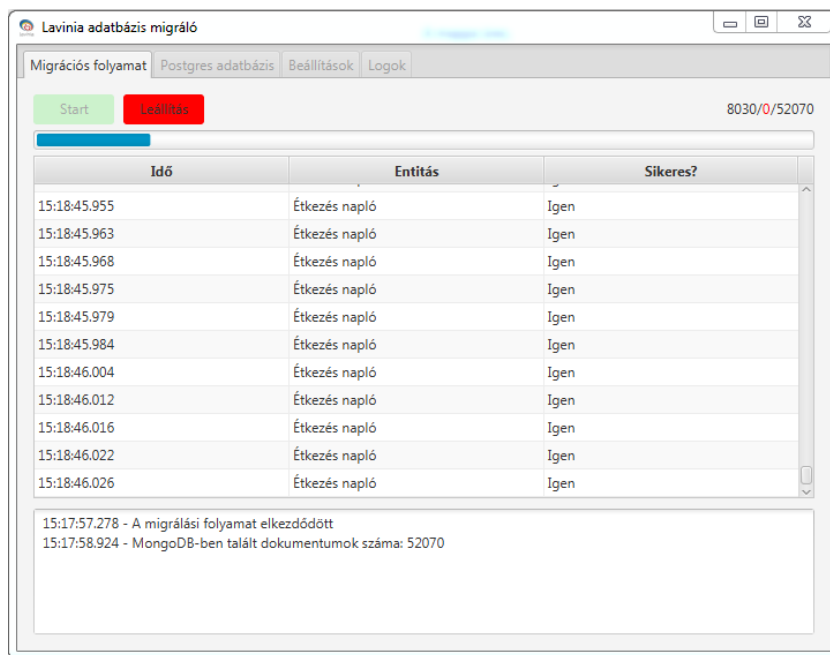
A példában lekérjük az aktuális kapcsolatot, amit a PostgreSQL szerverrel tartunk fenn. Az első beszűrő művelet kiadása után elhelyezünk az említett *Savepoint*-ból egyet, valamint több *INSERT* utasítást is kiadhatunk ezek után. A *rollback* függvény paramétere az előbb létrehozott mentési pont aktuális példánya, így a tranzakciót a JDBC API erre a pontra gördíti vissza. A véglegesítésnél csak egyetlen utasítás fog a tranzakcióban szerepelni, mégpedig az első, amint a *Savepoint* előtt adtuk ki.

Minden PostgreSQL adatbázison való művelet az ismertetett JDBC API-n megy keresztül függvényhívások segítségével. Ezek a metódusok ellenőrzött kivételeket dobhatnak hiba esetén. Ez azt jelenti, hogy a függvényhívás helyén le

kell kezelnem azt az esetet is, ha hiba lép fel. Másik választásom lehet a kivétel továbbdobása. A sok függvényhívás miatt rengeteg helyen kell megoldást találni a problémára. A tervezés során sikerült olyan eredményt elérnem, ami nemcsak jól és könnyen kezeli a hibákat, de még a folyamat hibakezelés előnyére is válik. Fontos megemlíteni, hogy a folyamat milyen hibákat képes felismerni és kezelni. Ezek túlnyomóan a PostgreSQL adatbázison keletkeznek, melyet Java oldalon kivétel formájában kapunk meg. Legtöbb esetben valamilyen hivatkozási integritás vagy kényszer (például értékkészlet, kötelező mezőkbe való *NULL* érték beszúrása) megsértése. A folyamat az adatot vagy adatokat nem képes magától pótolni, ezért a Specifikációban megfogalmazott hibakezelés lép életbe. A keletkezett kivétel tartalmazza a függvényhívások sorrendjét, így a folyamat képes meghatározni, hogy melyik szinten történt a hiba (felhasználó, eszköz vagy megfigyelés másolásánál). Mivel minden kivételt valamely szinten kezelni kell, így el kell dönteni, hogy pontosan hol reagáljuk le hibát. Ezt a felhasználó tudja megadni a beállításokban. Amennyiben felhasználói szintű tranzakciót állított be, úgy minden hiba miatt keletkezett kivételt a legfelső szinten, azaz a felhasználó másolásánál kell kezelni. Ez azt eredményezi, hogy az éppen migrált felhasználó csak abban az esetben kerül másolásra, amennyiben nem történt az alsóbb szinteken semmilyen hiba. Analóg módon lehet a többi tranzakció szintre levezetni a példát. A mechanizmus lényege, hogy a kivételt a felhasználó által definiált szinten kell kezelni. Ez változtatja meg a migráció kimenetelét.

6.5. On-line nyomon követés

Lehetőségünk van a migrációs folyamatot valós időben nyomon követni. Ez azt jelenti, hogy egy entitás migrációja után azonnali visszajelzést kapunk annak sikerességéről vagy sikertelenségéről. A 6.5. ábrán látható a felhasználói felület a migrációs folyamat futása közben.



6.5. ábra Valós idejű nyomon követés felülete

A folyamat futása közben az összes elérhető funkció le van tiltva. Ennek biztonsági oka van, például migráció közben ne lehessen a beállításokban a paraméterezést megváltoztatni, ami futás idejű hibákat váltana ki.

A „Start” és a „Leállítás” gomb funkciója a folyamat elindítása, valamint leállítása. A folyamat elindítása után a szoftver a beállításokban megadott paramétereket figyelembe véve kiszámolja, hogy hány darab dokumentum érintett a migrációban. Ez a szám a valós idejű nyomon követéshez szükséges. A leállítás gomb csak akkor aktív, amikor a migrációs folyamat fut. Az adatbázis konzisztenciáját figyelembe véve a megszakítás nem azonnali. A gombra való kattintás után a program még befejezi az utolsó megkezdett felhasználóhoz tartozó adatok másolását, de a soron következő felhasználót már nem fogja érinteni a folyamat.

A migráció aktuális állásához tartozó statisztikát az előbb említett gombokkal egy sorba, az ablak jobb oldalán találjuk. A három szám az alábbi szisztematika szerint épül fel: az első szám a sikeresen migrált dokumentumok darabszáma. Ezt követi pirossal kiemelve a sikertelenül migrált entitások száma. Az utolsó szám a már említett migrációban résztvevő összes dokumentum darabszáma. Az kurzort a számokon tartva tooltip jelenik meg, ami tájékoztatja a felhasználót a szám funkciójáról.

Az alatta lévő sorban egy folyamatjelző csíkot találunk. Ez szemlélteti a folyamat állapotát. Segítségével könnyebben meg lehet határozni a migráció aktuális állását. Ez nem más, mint az előbb ismertetett számok vizuális megjelenése.

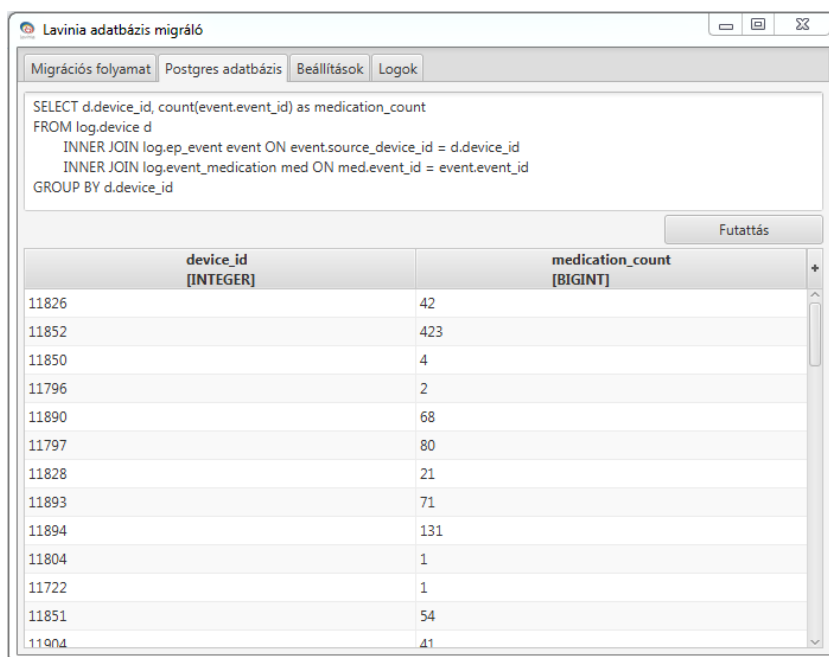
Az alatta lévő táblázat a migrációs folyamat során töltődik ki. Egy sor a migrációban résztvevő egy entitásnak feleltethető meg. Feladata közölni a felhasználóval, hogy az aktuális dokumentum másolása sikeres vagy sikertelen volt-e. Emellett megtudhatjuk a másolás pontos idejét ezredmásodpercre pontosan, valamint a dokumentum típusát (felhasználó, eszköz vagy a pontos megfigyelés neve). A 6.5. ábrán látható „Étkezési napló” bejegyzéseket a rendszer sikeresen átmásolta egyik adatbázisból a másikba. Hiba esetén az utolsó oszlopban „Nem” szöveg olvasható, valamint a kurzort e szövegen tartva tooltip-ben megjelenik a hibát kiváltó kivétel szövege. A tervezésnél felmerült, hogy a „Nem” szöveg mellett figyelemfelhívó színnel is ki legyen emelve azon sorok, ahol hiba lépett fel. Azonban a rendkívül gyors folyamatnak köszönhetően a felhasználói felület nem tudott lépést tartani a módosítások eseményeivel, így a helyesen átvitt dokumentumok sorait is gyakran kiszínezte pirossal. A problémára teljes értékű megoldást nem találtam így ez nem került bele a kész rendszerbe. A rendszer valós időben tervezési döntés következtében nem közöl részletesebb információt, azaz nem tudjuk pontosan, hogy mely entitásnál lépett fel a hiba. Ennek memóriakezelési döntés az oka, miszerint minimalizáljuk a migráción áthaladó adatok melletti felesleges információk tárolását. Természetesen a hibát okozó entitást könnyen ki tudjuk deríteni, hiszen a migrációt naplózó bejegyzésekben a hiba mellett fel van tüntetve a hibát okozó dokumentum MongoDB-ben lévő egyedi azonosítója.

A táblázat alatt egy nem szerkeszthető szöveges mező található. A migrációs folyamat ide naplóz minden olyan információt, amelyet a fenti táblázatba nem tud beilleszteni. Ilyen naplózási üzenet lehet például az esetlegesen fellépő kivételek típusa és azok üzenetei is. A 6.5. ábrán látható, hogy nem csak hibaüzeneteket, hanem egyéb kiegészítő információkat is tartalmaz. Ez tekinthető a migrációról készült napló lényegesen leegyszerűsített és valós idejű másolatának.

6.6. PostgreSQL adatbázisba való betekintés

Az elkészült szoftver tartalmaz egy olyan funkcionalitást, mely segítségével könnyedén, egyéb program futtatása nélkül betekintheünk a PostgreSQL adatbázisba. A felület felső részén egy szerkeszthető szövegmező található. Ide bármilyen érvényes SQL utasítás beírható. A „Futtatás” gombra kattintva a beírt parancsot a PostgreSQL adatbázison a szoftver végrehajtja.

A 6.6. ábrán egy SQL lekérdezést láthatunk, mely eszközönként megszámolja a hozzá tartozó gyógyszerelés típusú események darabszámát:



The screenshot shows a window titled 'Lavinia adatbázis migráló' with tabs for 'Migrációs folyamat', 'Postgres adatbázis', 'Beállítások', and 'Logok'. The 'Postgres adatbázis' tab is active, displaying a SQL query in a text area. Below the query is a 'Futtatás' button. The results are shown in a table with two columns: 'device_id [INTEGER]' and 'medication_count [BIGINT]'. The table contains 15 rows of data.

device_id [INTEGER]	medication_count [BIGINT]
11826	42
11852	423
11850	4
11796	2
11890	68
11797	80
11828	21
11893	71
11894	131
11804	1
11722	1
11851	54
11904	41

6.6. ábra Dinamikus felépülő táblázat

A visszakapott adathalmazt feldolgozva a szerkeszthető szövegmező alatti táblázat magától kitöltődik. A kapott metaadatokból az oszlopok számát, azok neveit, valamint típusait is kiolvastva rajzolódnak ki az oszlopok. A sorok pedig a kapott adathalmaz egyes elemei.

6.7. Biztonsági mentés

A szoftver lehetőséget kínál automatikus biztonsági mentések létrehozására az egyes migrációs folyamatok elindítása előtt. Ezt a beállítások között adhatjuk meg. A funkció csak a PostgreSQL adatbázis megadott sémáját menti el szöveges állományba, hiszen csak az ebben található adatszerkezetben keletkezik változás.

A program a *pg_dump* nevű alkalmazást használja annak érdekében, hogy az elkészült adatbázis mentést bármely PostgreSQL szerver értelmezni tudja. Ezen program eléréséhez szükséges, hogy a rendszer *PATH* nevű környezeti változó értékei között szerepeljen a *pg_dump* alkalmazást tartalmazó mappa elérési útvonala.

A mentés elkészítéséhez a rendszer generál egy parancsot, amit az operációs rendszer parancssora segítségével lefuttat. Az ilyen utasításra láthatunk példát:

```
pg_dump
--dbname=postgresql://user:pswd@localhost/lavinia
--schema log
```

A parancs a már említett *pg_dump* alkalmazást indítja el a megadott paraméterekkel. A *dbname* paraméterben az adatbázis elérési útvonala található a következő formában: *postgresql://[felhasználó]:[jelszó]@[hoszt]/[adatbázis]*. A táblákat tartalmazó sémát a *schema* paraméter definiálja.

A mentés tartalmának fájlba írásához a parancssori átirányító operátort alkalmaztam, így a Java kódban nem kellett fájlműveleteket használnom. Az állomány neve az alábbi sémát követi: *backup-log-schema-[év]-[hó]-[nap]-[óra]-[perc]-[másodperc].sql*.

6.8. Naplózás

A program minden elindítása után az összes fontosabb eseményről naplóbejegyzés készül. Segítségével könnyedén vissza lehet keresni a váratlan hibákat, valamint vissza lehet követni a program futásának eredményét.

A naplózást egy előre elkészített könyvtár segítségével oldottam meg. A függőség neve az Apache Log4j. Használta egyszerű, két lépés szükséges ehhez. Az első egy konfigurációs állomány, amely leírja a naplóbejegyzések tulajdonságait. Ez egy kulcs-érték párokat tároló szöveges dokumentum, ami az *src/main/resources* könyvtárban található meg *log4j.properties* néven. Naplóbejegyzések készítéséhez az adott osztályban példányosítani kell egy statikus *org.apache.log4j.Logger* objektumot, mely metódusai segítségével egyszerűen lehet generálni bejegyzéseket.

Nehézséget jelentett, hogy minden indításnál új fájlba kerüljenek a bejegyzések, valamint a fájl neve tartalmazza a program elindulásának időpontját is. Ezt úgy oldottam meg, hogy a beépített *FileAppender* osztályból (aminek a feladata fájlokba írni az egyes bejegyzéseket) származtattam egy saját osztályt, aminek az *activeOptions* függvényét felülírva új fájlt hozok létre a megadott néven, amit ezek után beállítok alapértelmezett kimenetnek. Amennyiben ez nem sikerülne, úgy a program inicializálásakor hibát kapunk, melynek pontos üzenetét a konzolról tudhatjuk meg.

A naplófájlok a program mellett található *logs* nevű mappában találhatók. Ha nincs ilyen mappa, akkor a program még nem volt elindítva. A naplófájlok nevében megtalálható a program elindításának ideje. Például a *log-migrator-2016-10-29-12-46-52.log* nevű napló 2016. október 29-én 12 óra 46 perc 52 másodperckor indult folyamatról készített bejegyzéseket tartalmazza. A kiterjesztése log, de ez csak egy szöveges dokumentum.

A naplóbejegyzésre egy példát láthatunk:

```
[TRACE]    2016-10-29    12:47:22,046    MigrationLogger  
startProcess - JDBC tranzakció szintje: manuális
```

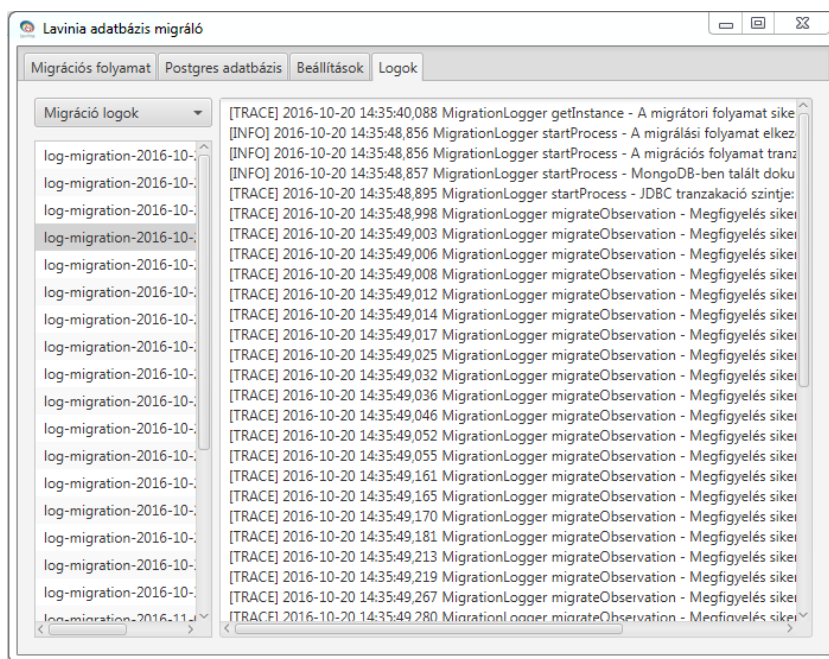
A szögletes zárójelben lévő sztring a naplóbejegyzés szintjét mutatja. Ez a különböző naplózó keretrendszerekben eltérhet, itt most egy közepes szintű üzenetről van szó. Utána jön a bejegyzés keletkezésének ideje ezredmásodperce pontosan. A MigrationLogger kifejezés annak a naplózónak a nevét adja meg, amely elkönyvelte ezt a bejegyzést. Egy programban több naplózó is lehet egyszerre, ami egy fájlba írja ki a történéseket, így fontos tudni, hogy éppen melyik naplózótól kaptuk ezt meg. A kötőjel előtti szó annak a metódusnak a neve, amiben a bejegyzés keletkezett. A kötőjel után a bejegyzés szövege található, amit gyakran a fejlesztő fogalmaz meg. A kivételeket a bejegyzések között hamar észre lehet venni. Ezek a sorok ugyanúgy kezdődnek, mint a többi, csak WARN szintű a bejegyzés. A kivételeket tartalmazó sorok után a kivétel típusa, üzenete, pontos helye és a hívások (stack trace) sorai találhatók.

Az általam elkészített program két naplózót is tartalmaz. A migrációs folyamatnak van egy saját naplózó objektuma is, ami duplikálja a bejegyzéseket.

Implementálás

Ez azért jó, mert így a migrációs folyamatról készült értesítések más szöveges állományban is megjelennek, így megkülönböztethető a többi bejegyzésekről. A *logs* nevű könyvtárban található fájlokban az egész programra kivetítve találhatóak a naplók, valamint az ebben található *migrations* alkönyvtárában csak a migrációs folyamatok naplói szerepelnek.

Lehetőségünk van az elkészített alkalmazás segítségével is megtekinteni az egyes naplófájlokat. Ezt a felületen a „Logok” fül alatt lehet elérni, amit a 6.7. ábra szemléltet.



6.7. ábra Napló tartalmának megtekintése

A legördülő menü segítségével megadhatjuk, hogy a migrációhoz vagy éppen az alkalmazáshoz köthető naplók tartalmát szeretnénk látni. Az program beolvassa az összes olyan naplófájlt, amit eddig létrehozott. Az ismeretet könyvtárakból összegyűjti az összes fájlt, ami naplófájl lehet (azaz a fájl neve és kiterjesztése is egy mintát követ). A bal oldalon kiválaszthatjuk az általunk megtekinteni kívánt fájlt, aminek a tartalma a jobb oldali nem szerkeszthető szövegmezőben jelenik meg.

6.9. Beállítások

A program számos beállítási lehetőségekkel rendelkezik. Ezeknek kisebb része a kódba van beégetve, azaz ezeket az értékeket nem lehet megváltoztatni. Ilyen a

felhasználói felület minimális méretei, vagy a komponenseken található feliratok. A migrációs folyamathoz tartozó paramétereket viszont a „Beállítások” fülön meg lehet adni. Ilyenek a PostgreSQL vagy a MongoDB adatbázishoz tartozó kapcsolatok szükséges értékei.

A kapcsolat mellett a migrációs folyamat néhány tényezőit is be lehet állítani. Ilyen a migrációs szint, ami a hibakezelésnél és az adatok másolásánál játszik fontos szerepet. Az adatbázisban található entitás típusokat is lehet szűkíteni. A migráció folyamán csak a kiválasztott típusokat kezeli a folyamat. Mivel a relációs adatbázisban a szoftver nagy módosítást hajt végre, ezért szükségeszerű a migrációs folyamat előtti biztonsági mentés. Bármilyen nagyobb módosítás előtt az adatbázis adminisztrátorok az adatbázis tartalmáról mentést készítenek, hogy hiba esetén vissza tudják állítani az eredeti állapotot. Lehetőségünk van a folyamatot automatizálni úgy, hogy a migráció előtt a programra bizzuk a biztonsági mentés létrehozását. Ehhez csak annyit kell biztosítanunk, hogy a két adatbázis biztonsági mentést végrehajtó folyamata a környezeti változók között legyen.

A beállítások aktuális értékeit egy szöveges állomány tárolja, ami a program mellett található *settings.properties* néven. Ez a fájl kulcs-érték párokat tárol. A program inicializálásakor feldolgozza a fájlt, aminek a tartalma egy *Map<String, String>* kollekcióba kerül.

7. Tesztelés

Ebben a fejezetben az elkészült migrátor alkalmazás teszteredményeit ismertetem. Lehetőségem volt két adatbázison is tesztelni a működést: a már említett adatbázis mentés, valamint annak egy szűkített változata, amiben minden entitásból pontosan csak egyet tartalmaz. A következőkben ismertetett eredményeket a rendelkezésemre álló adatbázison végeztem el.

Tesztelés céljából véletlenszerűen kiválasztottam, a MongoDB-ben található felhasználók közül egyet. Ezután egyeztettem a kiválasztott felhasználóhoz tartozó eszközök és a különféle megfigyelések darabszámát.

A kiválasztott dokumentum egyedi azonosítója: 5422b1dce4b0a769abd2b2cc. Az entitás többi mezőjének az értéke ebből a szempontból lényegtelen.

Migrálás után a PostgreSQL adatbázis *user* táblájában megnézhetjük, hogy sikeresen átmásoltuk-e a kiválasztott entitást.

```
SELECT COUNT(*) FROM log.user
WHERE ds_id = '5422b1dce4b0a769abd2b2cc'
```

Eredményként egyet kaptam, azaz a felhasználót sikeresen átmásolta a folyamat az egyik adatbázisból a másikba. A következő lépés megnézni, hogy az adott felhasználóhoz az összes eszköz hozzá lett-e rendelve az eredmény adatbázisban. A következő lekérdezés visszaadja, hogy a MongoDB adatbázisban hány eszköz tartozik a felhasználóhoz:

```
db.Device.find({"owner.$id" :
  ObjectId("5422b1dce4b0a769abd2b2cc")}).count()
```

Ugyanez SQL kifejezésben a relációs adatbázis sémájához igazítva:

```
SELECT COUNT(*)
FROM log.user u
      INNER JOIN log.episode ep
      ON ep.user_id = u.user_id
      INNER JOIN log.episode_device ed
      ON ed.episode_id = ep.episode_id
      INNER JOIN log.device device
      ON device.device_id = ed.device_id
WHERE u.ds_id = '5422b1dce4b0a769abd2b2cc'
```

Mindkét lekérdezés eredménye egy, így az eszköz és a hozzárendelés is sikeresen megtörtént. Fontos megjegyezni, hogy a MongoDB adatbázis nem ismeri az epizód fogalmát, míg a PostgreSQL igen, így a felhasználókhöz rendelt eszközök lekérdezése többszörös join művelettel valósítható csak meg.

Utolsó részben a megfigyeléseket kell ellenőrizni. Először azok összesített darabszámát (Mennyi megfigyelés van összesen?), majd típusonként lebontva (Hány darab megfigyelés van az adott típusból?) kell megvizsgálni. A kiválasztott felhasználóhoz a hozzá tartozó megfigyelések darabszámát az alábbi lekérdezéssel tudhatjuk meg:

```
db.Observation.find({"device.$id" :
    ObjectId("5422b1dce4b0a769abd2b2cd")}).count()
```

Ennek a következő SQL lekérdezés feleltethető meg:

```
SELECT COUNT(*)
FROM log.device dev
    INNER JOIN log.ep_event ev
        ON ev.source_device_id = dev.device_id
WHERE
    dev.ds_device_id = '5422b1dce4b0a769abd2b2cd'
```

A lekérdezésekben felhasználtam az eszköz egyedi azonosítóját, hiszen az az előbbi lekérdezésből ismertté vált. A lekérdezések eredménye 177.

A típusokra való lebontáshoz ismét a MapReduce technikát alkalmaztam. A fent ismertetett parancsot kiegészítettem egy feltétellel, hogy csak az adott eszközhöz számolja a különböző esemény típusok darabszámát. Az eredmény a 7.1. táblázatban látható.

Esemény típus	Darabszám
Vércukorszint mérés	9
Vérnyomás mérés	14
Laboreredmény	1
Étkezés	72
Gyógyszerezés	6
Testsúlymérés	1
Dietetikai anamnézis	3
Összesen	106

7.1. táblázat A megadott eszközhöz tartozó megfigyelések száma

Erre a kérdésre a következő SQL utasítás adhat választ:

```
SELECT e.event_type_code, COUNT(*)
FROM log.ep_event e
      INNER JOIN log.device d ON
            e.source_device_id = d.device_id
WHERE d.ds_device_id = '5422b1dce4b0a769abd2b2cd'
GROUP BY e.event_type_code
```

A lekérdezés eredményét a 7.2. táblázat tartalmazza:

Esemény típus	Darabszám
Vércukorszint mérés	9
Vérnyomás mérés	14
Laboreredmény	1
Étkezés	72
Gyógyszerezés	6
Testsúlymérés	1
Dietetikai anamnézis	3
Összesen	106

7.2. táblázat A migrált eszközhöz tartozó megfigyelések száma

Látható, hogy a két táblázat tartalma megegyezik, azaz a különböző típusú megfigyeléseket a migrációs folyamat helyesen ismerte fel és másolta át.

A darabszámok mellett természetesen fontosak az egyes entitásokon belüli adattagok értékeinek helyes másolása. A dokumentumok darabszáma miatt

manuálisan nem lehet belátható időn belül leellenőrizni a másolás helyességét. A problémára megoldást jelenthetnek az automata tesztek. Ennek kidolgozása több időt vesz igénybe, ezért ehelyett szűrőpróba szerűen választottam ki felhasználókat a rendszerben. Az entitások, valamint a hozzá tartozó eszközök és megfigyelések összes adattagját és azok értékeit megvizsgáltam. A teszt alatt nem leltem semmilyen különbséget a forrás és a cél adatbázisban lévő tulajdonságok értékei között.

8. Összefoglalás

Ahogy már említettem, az elkészült szoftver célja, hogy a Lavnia alkalmazás használatával strukturálatlan vagy félig strukturált MongoDB-ben lévő adathalmaz másolása egy új, megszorításokat és kényszereket is tartalmazó sémába. A migrációs folyamat nagy mennyiségű adatfeldolgozással jár, hiszen 3 kollekció tartalmát kellett 31 táblába szétválogatni.

A könnyebb kezelés érdekében a program grafikus felhasználói interfésszel rendelkezik, mely könnyen áttekinthető és könnyen kezelhető. Segítségével a migrálási paraméterek is egyszerűen testreszabhatóak. Ilyenek a tranzakciók kezelési szintjei, egyes entitások figyelembevételének ki- és bekapcsolása, stb. A folyamatot valós időben is követni lehet, mialatt a program a lehető legtöbb adatot közöl a felhasználóval. Folyamatjelző és számszerűsített adatok mutatják az aktuális állást. Táblázatban entitásra lebontva kapunk információt a másolás idejéről, sikerességéről, valamint az estelegesen fellépő hibák típusáról és pontos helyéről. A folyamat ellenőrzése és a hibák esetleges visszakeresése érdekében minden fontosabb eseményről naplóbejegyzés készül egy szöveges állományba. A felhasználói felületről lehetőség van SQL parancsok futtatására, valamint ezek eredményeinek grafikus megjelenítésére.

A probléma méretét mutatja, hogy a kapott teszt adatbázisban több, mint ötszázezer dokumentum található, mely egy 2015. tavaszán készült adatbázis mentés. Azóta ez a szám csak nőtt, így hatékony módszert kellett implementálnom. A sikerességet mutatja, hogy a teszt adatbázis tartalmát közel három perc alatt képes a szoftver átmásolni. Ezt az eredményt a másolási módszer részletes megtervezésével, valamint száakezeléssel sikerült megoldanom.

Irodalomjegyzék

[1] Brad Dayley (2015). Sams Teach Yourself NoSQL with MongoDB in 24 Hours. Sams.

[2] <http://docs.oracle.com/javafx/2/binding/jfxpub-binding.htm> Using JavaFX Properties and Binding (letöltés dátuma 2016.11.23.)

[3] <http://docs.oracle.com/javafx/2/threads/jfxpub-threads.htm> Concurrency in JavaFX (letöltés dátuma 2016.11.23.)

[4] <http://lavinia.hu/documents/lavinia215.pdf> Lavinia Életmód-tükör 2.1.5 Felhasználói kézikönyv (letöltés dátuma 2016.11.23.)

[5] http://mongodb.github.io/mongo-java-driver/3.2/driver/?_ga=1.205820309.729931641.1478510126 MongoDB Driver Documentation (letöltés dátuma 2016.11.23.)

[6] <http://napidroid.hu/harom-millional-is-tobb-okostelefon-lehet-a-hazai-piacon/> Három milliónál is több okostelefon lehet a hazai piacon (letöltés dátuma 2016.11.23.)

[7] <http://oroscafe.hu/2016/04/13/fekevesztetten-tor-elore-a-diabetesz-oroshazan-is/cikk/kezisrac> Fékevesztetten tör előre a diabétesz – Orosházán is (letöltés dátuma 2016.11.23.)

[8] <https://docs.mongodb.com/v3.2/crud/> MongoDB CRUD Operations (letöltés dátuma 2016.11.23.)

[9] <https://github.com/google/guava/wiki/CollectionUtilitiesExplained> Google Guava - CollectionUtilitiesExplained (letöltés dátuma 2016.11.23.)

[10] <https://projectlombok.org/features/Data.html> ProjectLombok @Data (letöltés dátuma 2016.11.23.)

[11] <https://www.postgresql.org/docs/9.6/static/index.html> PostgreSQL 9.6.1 Documentation (letöltés dátuma 2016.11.23.)

[12] https://www.tutorialspoint.com/log4j/log4j_logging_files.htm log4j - Logging in Files (letöltés dátuma 2016.11.23.)

[13] Korry Douglas, Susan Douglas (2003). PostgreSQL. Sams.

[14] Nemes Márta, Vassányi István, Kósa István, Pintér Balázs (2014). Okostelefon alapú diéta-naplózó rendszer diabeteses betegek számára. Új diéta: 15-17.

Mellékletek

```

|| Szakdolgozat Lénárt Bálint.docx
|| Szakdolgozat Lénárt Bálint.pdf
+---Forráskód
|   Lavinia_migration_process.zip
+---Hivatkozások
|   Concurrency_in_JavaFX.zip
|   Fekevesztetten_tor_elo_re_a_diabatesz.zip
|   Guava_CollectionUtilitiesExplained.zip
|   Harom_millional_is_tobb_okostelefon_lehet_a_hazai_piacon.zip
|   lavinia215.pdf
|   log4j_logging_in_files.zip
|   MongoDB_CRUD_Operations.zip
|   MongoDB_Driver_Documentation.zip
|   Postgres_documentation.zip
|   ProjectLombok_Data.zip
|   Using_JavaFX_Properties_and_Binding.zip
\---Ábrák
    1.1. ábra Cukorbetegség száma Magyarországon.jpg
    1.2. ábra Okostelefonok száma Magyarországon.jpg
    1.3. ábra Lavinia alkalmazás kezdőképernyője.png
    1.4. ábra Naplózó felület.png
    3.1. ábra Robomongo felhasználói felülete.png
    3.2. ábra pgAdmin III felhasználói felülete.png
    5.1. ábra Használati eset diagram.png
    5.2. ábra Komponens hierarchia.JPG
    5.3. ábra Migrációs folyamat szekvencia diagramja.png
    5.4. ábra Maven könyvtárszerkezet.png
    6.1. ábra Bejelentkezési felület.png
    6.2. ábra Funkciók elhelyezkedése a felhasználói felületen.JPG
    6.3. ábra Inicializáló felület sikertelen kapcsolódás esetén.png
    6.4. ábra PostgreSQL adatbázis kapcsolat paramétereinek beállítása.png
    6.5. ábra Dinamikus felépülő táblázat.png
    6.6. ábra Napló tartalmának megtekintése.png
    6.7. ábra Valós idejű nyomon követés.png
    Lavinia log séma.JPG

```

1. melléklet CD melléklet tartalma