

**Sapientia Erdélyi Magyar Tudományegyetem  
Műszaki és Humántudományok Kar,  
Marosvásárhely  
Szoftverfejlesztés Szak**

# **M E S T E R I   D I S S Z E R T Á C I Ó**

**Témavezető:  
Dr. ing. Johann Stan**

**Végzős hallgató:  
Aszalos Zoltán**

**2016**

**UNIVERSITATEA SAPIENTIA  
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE, TÂRGU-MUREȘ  
SPECIALIZAREA DEZVOLTAREA APLICAȚIILOR SOFTWARE**

## **Sistem pentru procesarea textelor medicale**

### **Lucrare de disertație**

**Coordonator științific:  
Dr. ing. Johann Stan**

**Absolvent:  
Aszalos Zoltán**

**2016**

UNIVERSITATEA SAPIENTIA  
FACULTATEA DE ȘTIINȚE TEHNICE ȘI UMANISTE DIN TÂRGU -MUREȘ  
SECȚIA DEZVOLTAREA APLICAȚIILOR SOFTWARE

VIZAT DECAN  
Dr. Kelemen András

VIZAT ȘEF CATEDRĂ  
Dr. Kátai Zoltán

### LUCRARE DE DISERTAȚIE

Conducătorul temei:  
Dr. Johann Stan

Candidat: Aszalos Zoltán  
Anul absolvirii: 2016

#### 1. Conținutul proiectului:

a) **Tema proiectului de diplomă:** Sistem pentru procesarea textelor medicale

b) **Problemele principale care vor fi tratate în proiect:**

- prezentarea problemei de identificare a interacțiunilor dintre medicamente
- introducerea noțiunilor de bază relevante implementării sistemului propus
- descrierea algoritmului pentru realizarea sistemului informatic
- efectuarea testării și evaluării sistemului

c) **Desene obligatorii:**

- schema bloc al aplicației
- figuri descriptive
- figuri cu rezultate experimentale

d) **Softuri obligatorii:**

- aplicație de tip server pentru prelucrarea datelor de intrare
- aplicație frontend pentru vizualizarea termenelor medicale

#### Bibliografie recomandată:

1. Daniel Jurafsky, James H. Martin, Speech and Language Processing, 2nd edition, Pearson Prentice Hall, 2008
2. Alan R. Aronson: MetaMap: Mapping Text to the UMLS Metathesaurus, 2006
3. Alan Gates, Programming Pig, O'Reilly Media, 2011

#### 2. Termene obligatorii de consultații: săptămânal

#### 3. Locul practicii: Universitatea Sapientia

Primit la data de:

Termen de predare:

Semnătura șefului de catedră:

Semnătura îndrumătorului științific:

Semnătura candidatului:

## Declarație

Subsemnata/Subsemnatul.....,  
funcția....., titlul științific..... declar pe propria răspundere  
că absolventul specializării de ..... a  
întocmit prezenta lucrare cu îndrumarea mea.

Forma finală a lucrării a fost verificată de mine și acesta corespunde cu  
cerințele de formă și conținut precizate Consiliului Facultății de  
..... în baza reglementărilor Universității Sapiientia.  
Lucrarea/proiectul corespunde și cerințelor de Legea Educației Naționale 1/2011 cu  
modificările ulterioare, Codului de etică și de ontologie profesională a Universității  
Sapiientia referitoare la furt intelectual.

Sunt de acord cu susținerea lucrării în fața comisiei de examen de disertație.

Localitatea,  
Data:

Semnătura îndrumătorului,

## Declarație

Subsemnata/Subsemnatul....., absolvent(ă)  
al/a specializării....., promoția ....., cunoscând  
prevederile Legii Educației Naționale 1/2011 și a Codului de etică și de ontologie  
profesională a Universității Sapiientia cu privire la furt intelectual, declar pe propria  
răspundere că prezenta lucrare de disertație se bazează pe activitatea personală,  
cercetarea/proiectare este efectuată de mine, informațiile și datele preluate din  
literatura de specialitate sunt citate în mod corespunzător.

Localitatea,  
Data:

Semnătura îndrumătorului,

## EXTRAS

Identificarea interacțiunii dintre medicamente are un rol important pentru medici, farmaciști deoarece acestea au un conținut informativ relevant pentru a preveni efectele adverse. În această lucrare pe de o parte am prezentat interacțiunea dintre medicamente utilizând setul de date publicat de serviciul DailyMed, pe de altă parte am căutat o soluție optimă de a procesa un volum mare de date prin intermediul platformei Hadoop. În procesul proiectării aplicației am considerat un factor important utilizarea celor mai bune practici aplicate în programarea web actuală pentru a menține un grad ridicat de calitate al codului, care asigură o bază solidă pentru dezvoltarea ulterioară și mentenanță.

DailyMed este un portal oficial în Statele Unite care oferă informații actualizate și detaliate despre medicamente comercializate. În această lucrare am folosit setul de date publicat de acest portal pentru asigurarea unei structuri care permite procesarea medicamentelor prin intermediul unui sistem informatic. Aplicația care aparține acestei lucrări permite căutarea eficientă a medicamentelor și al altor entități, și asigură consultarea efectelor adverse, a simptomelor și a interacțiunilor cauzate.

Dat fiind faptul că afecțiunile, simptomele și posibilele interacțiuni apar în text nestructurat folosind limbajul natural, a prezentat o problema cheie identificarea, procesarea și stocarea acestor termene pentru consultarea ulterioară. Pentru identificarea termenelor medicale apărute în aceste texte nestructurate am utilizat aplicația MetaMap care la rândul lui este un proiect sprijinit de guvernul american și permite recunoașterea elementelor UMLS (Unified Medical Language System).

Aplicația MetaMap folosește procesarea limbajului natural pentru textul de intrare și selectează termenele identificate, sugestiile sau evidențiază negațiile dintre expresii. Pentru a identifica interacțiunile dintre medicamente am folosit structurile de fraze marcate de MetaMap, apoi cu ajutorul expresiilor regulate predefinite am clasificat rezultatele pentru a decide care reprezintă interacțiuni negative.

Prin intermediul aplicației am realizat căutări semantice în baza de date DBPedia în scopul de a vizualiza o scurtă descriere al afecțiunilor, simptomelor identificate în prospectul medicamentelor. Cu rezultatele obținute prin intermediul interogării serviciului DBPedia am aplicat o căutare minimă pentru a selecta termenele cele mai potrivite.

Pentru procesarea cantității mari de date publicat de DailyMed, am folosit sistemul distribuit Hadoop care aplică algoritmul MapReduce pentru prelucrarea paralelă a datelor de intrare. Algoritmul MapReduce divide setul de date în unități mai mici al căror procesare se execută paralel prin intermediul mai multor noduri separate, iar în final îmbină aceste rezultate. În cazul construirii bazei de date a fost necesar stocarea eficientă a entităților aparținătoare (producător, substanțe, afecțiuni, simptome) care a fost asigurat de baza de date MongoDB de tip document. Rezultatul operațiunii executate pe platforma Hadoop a fost stocat în această bază de date.

### Rezultate

Am măsurat performanța sistemului din mai multe aspecte. În primul rând, am testat eficiența sistemului folosind 10 medicamente care au cuprins mai multe interacțiuni. Din perspectiva identificării interacțiunilor dintre medicamente, rata preciziei selective (precision) a fost de 82%, iar rata corectitudinii (recall) a sistemului a ajuns la 96%. Rata preciziei selective a fost influențat negativ de faptul că MetaMap a identificat incorect termene care nu au reprezentat elemente reale din textul precizat (de exemplu „Pharmaceutical Preparations” sau „Pharmacologic Substance”).

Am verificat eficiența interogării serviciilor DBPedia și MedlinePlus: sistemul a reușit să atingă rate de 56% și respectiv 82%. În cazul interogării bazei de date DBPedia valoarea scăzută este

cauzată de faptul că expresia folosită nu cuprinde toate termenele căutate.

Dintr-o altă perspectivă, am adunat numărul termenelor care reprezintă afecțiuni sau simptome în setul de date publicat de DailyMed în ianuarie 2016 care a inclus 2875 medicamente. Am selectat 20 dintre cele mai frecvente medicamente din acest set identificat de MetaMap. Statisticile obținute pot fi consultate în ultimul capitol al lucrării.

În final, am examinat calitatea codului sursă al aplicației frontend prin măsurarea ratei de acoperire a testelor. 71 de unit teste și 3 teste de integrare au asigurat menținerea unui grad ridicat de calitate al codului sursă: în cazul criteriilor aplicate testele au atins o acoperire de aproximativ 75% (din care rata de acoperire a expresiilor a fost de 84.29%).

### **Cuvinte cheie**

UMLS, MetaMap, Hadoop, Pig Latin, MapReduce, DBPedia, SPARQL, Javascript, Node.js, Angular.js, MongoDB, code coverage, unit test, integration test

**SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM  
MŰSZAKI ÉS HUMÁNTUDOMÁNYOK KAR, MAROSVÁSÁRHELY  
SZOFTVERFEJLESZTÉS SZAK**

## **Orvosi szövegértelmező rendszer**

### **Mesteri disszertáció**

**Témavezető:  
Dr. ing. Johann Stan**

**Végzős hallgató:  
Aszalos Zoltán**

**2016**



# KIVONAT

Gyógyszerek közötti kölcsönhatások megállapítása fontos szerepet játszik orvosok, gyógyszerészek vagy más gyógyászati szakmával foglalkozók körében, mivel ezek releváns információt szolgáltatnak, hogy elkerülhetővé tegyék a súlyos mellékhatások kiváltását, megelőzését. Ebben a munkában egyrészt gyógyszerek közötti kölcsönhatások kimutatását céloztam meg a DailyMed szolgáltatás által nyilvánossá tett gyógyszer adathalmazon, másrészt megoldást kerestem a nagy mennyiségű adat kezelésére az osztott rendszeren alapuló Hadoop platformon keresztül. A dolgozathoz készült alkalmazás tervezése során fontos szempontnak tekintettem a modern web alapú programozásban ismert bevált gyakorlatok használatát a magasszintű kódminőség elérése végett, amely egy jó alapot biztosít a későbbi továbbfejlesztésre és karbantartásra.

A DailyMed egy hivatalos portál az Egyesült Államokban, amely részletes címkeinformációkat szolgáltat a forgalomba hozott gyógyszerekről, és napirendre kész adatokat tartalmaz a legújabban megjelenő termékekről. A munkában ezen portál által publikált gyógyszer adathalmazt használtam fel ahhoz, hogy gépi úton kezelhető struktúrát biztosítsak ezen gyógyszereknek és a hozzájuk kapcsolódó adatoknak. A dolgozathoz készült alkalmazás lehetővé teszi a gyógyszerek és hozzájuk kapcsolódó adatok keresését egy könnyen kezelhető interfészen keresztül, amely biztosítja a keresett gyógyszer által okozott mellékhatások, tünetek és kölcsönhatások megjelenítését.

Mivel a betegségek, tünetek és fennálló kölcsönhatások struktúrátlan, természetes nyelvi szövegben jelennek meg a forrásban, kulcsprobléma volt ezen kifejezések felismerése, feldolgozása és eltárolása a későbbi megjelenítés céljából. Ezen struktúrátlan szövegrészletekben megjelenő orvosi kifejezéseket a MetaMap eszköz segítségével végeztem, amely szintén egy amerikai kormány által támogatott projekt, és lehetővé teszi az ún. UMLS (Unified Medical Language System) kifejezések felismerését.

A MetaMap természetes nyelvi feldolgozást alkalmaz a bemeneti szövegen, és megjelöli a felismert kifejezéseket, javaslatokat határoz meg vagy kifejezések közötti tagadásokra mutat rá. A kölcsönhatások megállapítása érdekében felhasználtam a MetaMap által megjelölt mondatstruktúrákat és gyógyszer kifejezéseket, majd előre definiált reguláris kifejezések segítségével osztályoztam a találatokat, hogy eldöntsem melyek vonatkoznak negatív gyógyszer-kölcsönhatásra.

Az alkalmazáson belül a gyógyszerek útmutatóiból kinyert betegségek és tünetek rövid meghatározására a DBPedia adatbázisában végeztem szemantikus kereséseket, majd az eredményekkel minimumkeresést alkalmaztam a leginkább illeszkedő kulcsszó azonosítására.

Mivel a DailyMed portál nagy mennyiségű gyógyszer adathalmazt tartalmaz, ennek hatékony előfeldolgozását a Hadoop osztott rendszerrel végeztem, amely a MapReduce algoritmust alkalmazza a bemenő adatok párhuzamos kezelésére. A MapReduce algoritmus a bemeneti adatok halmazát kisebb egységekre osztja szét, amelyek feldolgozását párhuzamosan, több számítási csomóponton keresztül végzi el, majd az egyes részmegoldásokat egyesíti. A felépített gyógyszer adatbázis esetében szükség volt a kapcsolódó adatok (gyártó, tartalmazott alapanyagok, felismert betegségek, tünetek) hatékony eltárolására, amelyet a dokumentum alapú MongoDB adatbázis biztosított. A Hadoop platformon végrehajtott feladatok eredménye ebben az adatbázisrendszerben került eltárolásra.

## Eredmények

A rendszer teljesítményét több kritérium szerint vizsgáltam meg. Először, 10 gyógyszer esetében vizsgáltam meg a rendszer hatékonyságát, amelyek számos kölcsönhatásban vettek részt. A felismert gyógyszer-kölcsönhatások szempontjából a rendszer szelektív precizitási (precision) aránya 82%-os volt, míg a rendszer helyességi (recall) aránya elérte a 96%-ot. A szelektív precizitás értékét negatívan befolyásolta, hogy a MetaMap olyan kulcsszavakat is visszatérített, amelyek valójában

csak gyűjtőnevek („Pharmaceutical Preparations” vagy „Pharmacologic Substance) és nem valódi találatok.

Megvizsgáltam a szemantikus keresések hatékonyságát a DBPedia majd a MedlinePlus webes szolgáltatás esetében: a rendszer 56%-os illetve 82%-os precizitást ért el. A DBPedia esetében az alacsonyabb számérték azzal magyarázható, hogy a felhasznált lekérdezés átlagban a találatok felére adott vissza helyes eredményt.

Egy újabb felmérés keretén belül összesítettem a betegségek és tünetek találatainak számát a DailyMed 2016 januárjában publikált csomagjára, amely 2875 gyógyszert tartalmazott. Kiválasztottam a 20 leggyakoribb betegség és tünet nevét, amelyet a MetaMap ezen gyógyszerekre azonosított. A számadatok a zárófejezetben tekinthetők meg.

Végül, megvizsgáltam a dolgozathoz készült alkalmazás kliens oldali komponensének kódminőségét lefedettségi arányok meghatározásával. 71 egységteszt és 3 integrációs teszt biztosította a kódminőség magas szintjének elérését: a megvizsgált kritériumok esetében a tesztesetek átlagban 75%-os lefedettséget eredményeztek (amelyből a kifejezések lefedettsége elérte a 84.29%-ot).

### **Kulcsszavak**

UMLS, MetaMap, Hadoop, Pig Latin, MapReduce, DBPedia, SPARQL, Javascript, Node.js, Angular.js, MongoDB, code coverage, unit test, integration test

**SAPIENTIA HUNGARIAN UNIVERSITY OF TRANSYLVANIA  
FACULTY OF TECHNICAL AND HUMAN SCIENCES  
DEPARTMENT OF MATHEMATICS-INFORMATICS  
SPECIALIZATION SOFTWARE DEVELOPMENT**

## **System for medical text processing**

### **Master's thesis**

**Supervisor:  
Dr. ing. Johann Stan**

**Student:  
Aszalos Zoltán**

**2016**

# ABSTRACT

Identifying interactions between drugs plays an important role for physicians, pharmacists or other health care professionals, because these contain relevant information to prevent serious side effects. In this work my aim was to detect such interactions using the drug data set provided by the DailyMed service and to find a solution for handling big amounts of data using the Hadoop platform. During the design of the accompanying application I favored the use of best practices in modern web development to maintain a high quality of the code base, which serves a good foundation for future work and maintenance.

DailyMed is an official portal in the United States which provides detailed label information for commercialized drugs and contains up to date information for new products on the market. This work uses the data published by this portal to create a data structure for these drugs and related entities usable by machine processing. The application enables the search for these drugs and entities using a user friendly interface which facilitates reporting the possible side effects, caused diseases and interactions.

Given the fact that diseases, symptoms and possible drug interactions appear in unstructured texts written in natural language, a key problem was to identify these expressions, process them and store them for later retrieval. To identify these medical terms in unstructured texts I used the MetaMap tool which is a project funded by the American government and facilitates the annotation of UMLS (Unified Medical Language System) terms.

MetaMap uses natural language processing on the given input and annotates detected terms, defines possible candidates and identifies negations between terms. To identify interactions between drugs I used the terms and phrases delimited by MetaMap, then I classified the findings to decide which of them relate to negative interactions between drugs.

In the application I used the diseases and symptoms identified from the additional labels to provide a short definition for them by querying the DBpedia database using semantic search methods. On the returned results I applied minimum edit distance search between terms to find the most appropriate ones.

The big amount of the data provided by the DailyMed portal necessitates the use of the Hadoop distributed system which applies the MapReduce algorithm to process input data in parallel. The MapReduce algorithm distributes the input data into smaller chunks whose processing is done in parallel using many nodes and in the end, the multiple results are merged together. In case of the used database I needed to store the related data (producers, ingredients, detected diseases, symptoms) efficiently using the document based MongoDB database. The result of the jobs executed on the Hadoop system were stored in this database.

## Results

The performance of the system was measured by several criteria. First, I measured the system's effectiveness by using 10 drugs which contained many interactions. Based on the detected interactions the system's precision was 82% and the system's recall was 96%. The precision of the system was negatively influenced by the fact that MetaMap identified such terms which were group names ("Pharmaceutical Preparations" or "Pharmacologic Substance") and were not relevant for the given input.

I measured the effectiveness of the semantic searches used for DBpedia and MedlinePlus web service: the precision of finding the correct definitions for terms were 56% and 82%, respectively. The lower precision in case of DBpedia is caused by the fact that the ontology is rather diverse and the used queries don't cover all cases.

By another benchmark I measured the number of appearances of the diseases and symptoms identified in case of 2875 drugs from the package published by DailyMed in January 2016. I collected the 20 most frequent disease and symptom names which were identified by MetaMap. The resulting statistics are located in the final chapter of this work.

Lastly, I analyzed the code quality of the accompanying application's front end component by measuring code coverage information. 71 unit tests and 3 integration tests were used to assure a high quality of the product: in case of the applied measurements the test cases on average resulted in 75% code coverage (code coverage of statements were 84.29%).

### **Keywords**

UMLS, MetaMap, Hadoop, Pig Latin, MapReduce, DBPedia, SPARQL, Javascript, Node.js, Angular.js, MongoDB, code coverage, unit test, integration test

## Tartalomjegyzék

1. Bevezetés.....	16
2. Kapcsolódó munkák.....	18
2.1. NegEx algoritmus.....	18
2.2. NegFinder algoritmus.....	19
2.3. Gyógyszerkölcsonhatások felismerése a DrugDDI adathalmazban.....	20
3. Elméleti alapok.....	20
3.1. Természetes nyelv elemzés.....	20
3.2. MetaMap.....	22
3.3. Hadoop és Pig lekérdezések.....	23
3.4. SPARQL.....	24
3.5. Reguláris kifejezések.....	25
4. Az algoritmus működési elve.....	26
5. A rendszer megvalósítása.....	29
5.1. Követelmények.....	29
5.2. Az alkalmazás felépítése, architektúrája.....	30
5.3. A szerver komponens.....	37
5.3.1. A server.js modul.....	38
5.3.2. A routes komponens.....	39
5.3.3. A passport komponens.....	40
5.3.4. A logger komponens.....	41
5.3.5. A db komponens.....	41
5.3.6. Egyéb szerver oldali komponensek.....	43
5.4. Az adatelemző komponens.....	45
5.4.1. A data.import komponens.....	45
5.4.2. Pig szkriptek használata.....	49
5.5. A kliens komponens.....	49
5.5.1. A common modul.....	52
5.5.2. A widgets modul.....	53
5.5.3. A users modul.....	53
5.5.4. A main modul.....	56
5.5.5. Az admin modul.....	62
6. Tesztelés és eredmények.....	63
6.1. Gyógyszerkölcsonhatások tesztelése.....	63
6.2. Betegségek és tünetek felismerésének tesztelése.....	65
6.3. Betegségek és tünetek előfordulása.....	66
6.4. Forráskód minőségének tesztelése.....	68
6.5. Következtetések és továbbfejlesztési lehetőségek.....	70
7. Irodalomjegyzék.....	71

## Táblák jegyzéke

3.1. Táblázat: Találatok minősítése.....	21
3.2. Táblázat: Példa RDF hármasokra a DBPedia adatbázisában.....	24
5.1. Táblázat: Az alkalmazás függőségei.....	35
5.2. Táblázat: Az alkalmazás által támogatott URL azonosítók listája.....	40
5.3. Táblázat: DailyMed XML dokumentum elemzésének XPath kifejezései.....	46
5.4. Táblázat: Az alkalmazás által használt szemantikus típusok.....	48
5.5. Táblázat: A users modul által támogatott URL azonosítók listája.....	54
5.6. Táblázat: A SearchController osztály függőségei.....	57

6.1. Táblázat: Aspirin kölcsönhatása Arava és Multivitamin találatok esetén.....	64
6.2. Táblázat: Találatok minősítése 10 kivélasztott gyógyszer esetében.....	64
6.3. Táblázat: Találatok minősítése.....	64
6.4. Táblázat: A rendszer teljesítményének összefoglalása.....	64
6.5. Táblázat: Betegségek és tünetek definícióinak detektált száma.....	65
6.6. Táblázat: A rendszer teljesítménye a detektált betegségek és tünetek számára nézve.....	66
6.7. Táblázat: 20 leggyakoribb betegség és tünet neve a DailyMed 2016. januárjában kiadott gyógyszer adathalmazra.....	67
6.8. Táblázat: Kód lefedettségi szintjének összesítése.....	68
6.9. Táblázat: Kifejezések lefedettségi szintje a users és a main modulok esetében.....	69

## Ábrák jegyzéke

4.1. Ábra: Az alkalmazást alkotó komponensek és kapcsolataik.....	26
4.1. Ábra: Felhasználó által történő keresést megvalósító komponensek.....	28
5.1. Ábra: Az alkalmazás Road map diagramja.....	31
5.2. Ábra: Az alkalmazás egyszerű Use Case diagramja.....	31
5.3. Ábra: Az alkalmazás szekvenciadiagramja.....	32
5.4. Ábra: Az alkalmazás futtatása terminálból.....	37
5.5. Ábra: Szerver komponensek Road map diagramja.....	38
5.6. Ábra: Routes modul komponens diagramja.....	40
5.7. Ábra: A db komponens osztálydiagramja.....	42
5.8. Ábra: Az adatelemző komponens függőségei.....	45
5.9. Ábra: A data.import komponens modul diagramja.....	46
5.10. Ábra: Kliens komponens modul diagramja.....	52
5.11. Ábra: A common modul osztálydiagramja.....	52
5.12. Ábra: A users modul osztálydiagramja.....	53
5.13. Ábra: A main modul osztálydiagramja.....	56
5.14. Ábra: Az alkalmazás kereső oldala.....	58
5.15. Ábra: Gyógyszeradatokat megjelenítő oldal az alkalmazásban.....	60
5.16. Ábra: SPARQL végpont által visszatérített JSON objektum.....	61
5.17. Ábra: Az alkalmazás előzményeit megjelenítő oldal.....	62
5.18. Ábra: Adatelemzés elindítása az alkalmazáson belül.....	62
6.1. Ábra: 20 leggyakoribb betegség és tünet előfordulása a DailyMed 2016. januárjában kiadott gyógyszer adathalmazra.....	67
6.2. Ábra: Integrációs tesztek futtatásának eredménye.....	69

# 1. Bevezetés

Az információ korát éljük: hatalmas mennyiségű adat áll rendelkezésünkre, amelynek feldolgozása komoly nehézségeket jelent az informatikai rendszerek számára. Az orvostudomány területén is észlelhető ez a jelenség, hogy a szakemberek által használt rendszerek egyre nagyobb mennyiségű adat kezelését kell lehetővé tegyék. Nagy mennyiségű gyógyszer áll az egészségügyi alkalmazott rendelkezésére, amelyeket különböző betegségek kezelésére írnak fel, viszont ezen termékek nem megfelelő használata súlyos mellékhatásokat vagy akár tragikus kimenetelű eseteket is eredményezhet.

Évente másfél millió ember szenved a nem megfelelően használt gyógyszerek szedése által. [1] Gyógyszer-gyógyszer kölcsönhatások (DDI drug-drug interaction) súlyos klinikai tüneteket válthatnak ki az által, hogy vagy gyengítik a gyógyszer hatásfokát vagy pedig erősítik a gyógyszer által okozott káros mellékhatásokat [2]. Ezért olyan megoldásra van szükség, amely könnyebbé teszi az egészségügyi alkalmazott számára ezen információkhoz való hozzáférést ahhoz, hogy megfelelő módon tudjon dönteni a megfelelő gyógyszer felírása végett. Ezen információk közé tartozik, hogy az egyes gyógyszereket milyen más gyógyszerekkel lehet együtt használni az illető beteg egészségi állapotának kockáztatása nélkül.

Ahhoz, hogy csökkentsük a előfordulandó hibák számát, fontos szerepe volna egy olyan informatikai rendszernek, amely könnyen és gyorsan képes kimutatni a lehetséges kölcsönhatásokat a különböző gyógyszerek között, vagy legalább feltüntetné az orvos számára a lehetséges gyógyszereket, amelyek kapcsolatban állnak egymással.

A DailyMed portál egy megbízható eszköz az amerikai piacon levő gyógyszerek használati utasításainak, figyelmeztetéseinek megfigyeltetésére, viszont ezt stuktúrálatlan szövegrészeket által biztosítja, amely praktikai szempontból megnehezíti a egészségügyi alkalmazott munkáját. A gyógyszerek leírását XML dokumentumok által biztosítja, amelyek struktúrált formátumban tárolnak különböző információkat az illető gyógyszerről, mint például az összetevők, alapanyagok, gyártó neve, stb. Ezeket az adatokat böngészhetővé tehetjük, ha egy olyan elmozzó (parser) programot írunk, amely képes kibányászni egy egyes dokumentumfájlokból a releváns információkat, és képes arra, hogy összefésülje a többi fájlban szereplő adatokkal. Ez egy nehéz feladat, mivel nagy mennyiségű adat áll rendelkezésünkre, és ennek mennyisége folytonosan növekedik, mivel újabb gyógyszerek jelennek meg napi rendszerességgel a piacon. Ezért egy olyan megoldásra van szükségünk, amely a hatalmas mennyiségű adatot optimálisan képes kezelni, és ugyanakkor biztosítja, hogy a folyton bővülő adathalmazt frissítse a legújabb adatokkal.

A legnagyobb problémát az jelenti, hogy a számunkra legfontosabb információ struktúrálatlan formában jelenik meg a dokumentumokban, és nincsen semmilyen indíték arra, hogy ezt, hogyan dolgozzuk fel, és indexeljük a könnyebb és gyorsabb elérés érdekében. Sok esetben csak egy név áll rendelkezésünkre, amely természetesen az egészségügyi alkalmazottnak nem biztos, hogy sokat jelent. Ezért szükség van arra, hogy egy hivatkozást biztosítsunk a felhasználónak, amelyet követve eljut a gyógyszer részletes leírásához. Ez a gyógyszer szintén szerepel a DailyMed adatbázisában valamelyik XML dokumentum formájában, viszont ezt meg kell találni, és optimálisan el kell tárolni a későbbi megjelenítés céljából.

Egy olyan rendszer tervezésére törekedtem, amely képes feldolgozni a folyamatosan bővülő adathalmazt az említett portál által közzétett adatok függvényében. Meg kell jegyeznem, hogy havi,



heti vagy akár napi rendszerességgel jelennek meg új adatok a portálon XML dokumentumok formájában, ezért szükség van egy olyan szolgáltatás alapú megoldásra, amely állandó időközönként megvizsgálja az illető forrást az új adatok megjelenéséről. Amennyiben friss adatokat talál, letölti azokat, és feldolgozza offline módban, összefésülve a létező adatbázisban eltárolt adatokkal. Lehetőséget biztosítottam a felhasználónak, hogy meghatározza, amennyiben igénybe szeretné venni ezt a szolgáltatást vagy használja a létező adatbázist a feltöltött adatokkal, és csak akkor frissítse a rendszert, ha ennek szükségét érzi vagy pedig lehetősége nyílik rá (pl. Internet kapcsolat hiánya miatt vagy egyéb megfontolásokból).

Az offline mód alatt azt értjük, hogy nem egy gyors feladat (job) elvégzéséről van szó, hanem egy időigényesebb munka végrehajtásáról beszélünk, amely több percig vagy akár óráig is eltarthat. Természetesen, ezt a feladatot az alkalmazás használata előtt kell beütemezni és végrehajtani ahhoz, hogy használható legyen előben, amikor már kereséseket lehet végezni valós időben, elfogadható precizitással és válaszidővel.

A nagy adat (big data) témaköre egyre fontosabb szerepet kap napjainkban, mivel rengeteg új, eddig kihasználatlan lehetőség rejlik az adatok feldolgozásában és értelmezésében. Egyre több és változatosabb megoldás jelenik meg a piacon, amely képes rendszerezni és elemezni a nagy adatban rejlő összefüggéseket. Említettem, hogy a megoldandó probléma szembesül ezzel a feladattal, mivel relatív értelemben véve nagy mennyiségű adatról van szó, amely feldolgozásra vár. Ebben a feldolgozásban is (amely jelen esetben offline módban fog történni) törekednem kellett arra, hogy minél gyorsabb megoldást válasszam, és optimálisan osszam szét a feladatot kisebb egységekre, amely feldolgozása könnyebben végezhető el több csomópont (node) segítségével. Nyilvánvalóvá vált, hogy nem használhatam hagyományosan alkalmazott eszközöket, ahhoz az adatokat eltároljam és feldolgozzam, mivel több tízezer nagyságrendű gyógyszer feldolgozása volt a célom. Megállapítottam, hogy offline módban fogom végezni a feldolgozást, amely az adatokat úgy fogja rendszerezni, hogy az kezelhetővé váljon az alkalmazás számára, és így, valós időben biztosítsa a felhasználó számára a használatot.

A MapReduce egy olyan modell, amely biztosítja egy összetett, bonyolult feladat megoldását úgy, hogy a bemenő adathalmazt több részre osztja szét egy párhuzamosan végrehajtott osztott rendszer segítségével. Az algoritmust egy számítógép halmazon (computer cluster) hajtja végre, amely több egyedi csomópontból (node) áll. Több könyvtár megvalósítás létezik, amely a MapReduce-on alapul, viszont az Apache Hadoop az egyik legnépszerűbb termék, amely támogatja ezen feladatok megoldását. A nagy adat feldolgozása azért nehéz, mert az egyes számítógépekben működő hardver lemezek írása és olvasása lassú a rendelkezésre álló adathoz képest. Abban az esetben is, ha több csomópontot használunk, a összetett feladatot kisebb egységekre kell szétosztani, majd a részeredményeket össze kell fésülni egymással, amely ismét problémákat okoz. A Hadoop által biztosított MapReduce algoritmus segít elvonatkoztatni a fizikai értelemben vett hardver eszközök kezelésétől, és egy könnyen kezelhető modellt biztosít a programozó számára. Röviden összefoglalva, az algoritmus két részből áll: a map fázisban kiválasztjuk a számunkra releváns adatokat (például gyógyszer gyártója vagy alapanyagok kódja) és a reduce fázisban összesítjük az eredményeket az egyes egységekből és előállítjuk a kimenetet (például az összes gyógyszer bejegyzés, amely tartalmazza az etil alkohol összetevőt vagy az összes gyógyszert, amelyet egy gyártó állít elő). A MapReduce algoritmus önmagában véve egy batch alapú rendszer, amely nem valós időben kerül végrehajtásra, hanem hosszabb ideig tart a futtatása (ez akár több óráig is eltarthat).

A gyógyszerek közötti kölcsönhatások megállapításában és felismerésében fontos szerepet játszott a MetaMap rendszer, amely a gyógyászati kifejezések felismerését teszi lehetővé teljesen struktúrátlan szövegben. A jelen probléma esetében a DailyMed által eltárolt XML dokumentumokból kinyert figyelmeztető szövegrészleteket használtam fel, és bemenetként adtam

meg a MetaMap rendszernek ahhoz, hogy felismerjem a releváns gyógyszereket, kölcsönhatásokat, az okozott tüneteket vagy betegségek neveit. Ez a rendszer egy tudás alapú eszköz, amelynek adatbázisában különböző gyógyászati kifejezések szerepelnek különböző szemantikus típusokba csoportosítva. A rendszer magában foglalja az ún. UMLS Metathesaurus adathalmazt, amely egy egységes gyógyászati nyelvrendszer (unified medical language system), és arra szolgál, hogy kapcsolatot teremtsen gyógyászati szövekben használt kulcsszavak, kifejezések és a tudásbázisában eltárolt fogalmak között. [3] A felhasznált algoritmus lépései között szerepel a tokenizálás, mondatvégek felismerése, lexikális keresés, szintaktikai elemzés, lehetséges találatok kiválasztása, szavak megkülönböztetése, stb. A rendszer különböző szűrésüket tesz lehetővé az által, hogy leszűkíthetjük a keresett kifejezések osztályát néhány típusra, amely érdekel bennünket. Jelen esetben felhasználtam a sosy (Sign or symptom), dsyn (Disease or syndrome), orch (Organic Chemical) és phsu (Pharmacologic Substance) szemantikus típusokat a MetaMap megfelelő kapcsolójának értékeként. Ezen szemantikus típusok teljes listája megtekinthető online. [4]

## 2. Kapcsolódó munkák

### 2.1. NegEx algoritmus

Több munka is íródott ebben a témában, amelyek megoldásokat keresnek klinikai szövegekben szereplő kulcsszavak felismerésére: például annak megállapítása, hogy az illető beteghez társított-e a szöveg szerzője valamely adott betegséget (pl. HIV pozitív). Chapman et al. arra kerestek megoldást, hogy kimutassák klinikai szövegekben megjelenő betegségek vagy tünetek szerepét a páciensre nézve tagadásra épülő reguláris kifejezések által. [5] A kifejlesztett NegEx algoritmus felhasznál néhány reguláris kifejezést arra, hogy felismerje a tagadásokat a szövegben és kiszűrje azokat a betegségeket és tüneteket, amelyek a tagadás hatókörén belül jelennek meg. Klinikai szövegekre jellemző, hogy a szerző (orvos) sok esetben azt fejezi ki, hogy milyen betegségeket lehet kizárni, elvetni a jelen ismeretek, vizsgálatok alapján. Ezeket a kijelentéseket tagadások formájában jegyzi fel a jelentésben:

(1) „A betegnek nem voltak gyomorfájdalmai.” vagy

(2) „A beteg nem utalt fejfájásra, viszont a sugárvizsgálat súlyos idegkárosodást mutatott”

Az (1) esetben az említett betegség hiányát állapították meg a betegen, ami azt jelenti, hogy a tünet (gyomorfájdalom) nem releváns rá nézve, ezért az algoritmus nem kell feltüntesse a tünetet, mint találatot. A (2) esetben viszont a tagadás a fejfájásra vonatkozik, nem pedig az idegkárosodásra, ezért az algoritmusnak ki kell mutatnia az utóbbit (az idegkárosodás jellemző a betegre). Itt problematikus annak megállapítása, hogy a tagadás mely klinikai kifejezésekre vonatkozik (az egyik kulcsszót elutasítjuk a találatok listájából, a másikat viszont nem).

A találatok megkereséséhez az ún. UMLS rendszert használták, amely segítséget nyújtott orvosi kifejezések felismeréséhez. [6] A NegEx algoritmus bemenete egy mondat, amelyben meg vannak jelölve az egyes klinikai kifejezések (betegségek, tünetek), a kimenetben pedig fel vannak tüntetve, hogy mely találatok pozitívak és melyek negatívak a betegre nézve. Az algoritmus kiértékelését, teljesítményét úgy vizsgálták meg, hogy összehasonlították egy egyszerű alapalgoritmussal (baseline algorithm), amelyet a University of Pittsburgh fejlesztett ki betegek csoportosítására, amely szintén tagadások felismerésén keresztül szűri ki a hamis találatokat (false positives), azonban nem tesz különbséget a tagadások hatókörei között (ha tagadást észlel a mondatban, akkor az összes orvosi találatot elveti abban a mondatban). Nyilvánvaló, hogy ez csökkenteni az

algoritmus precizitását.

Az algoritmus megvalósításához 35 tagadásos kifejezést használtak fel, amelyet két csoportra osztottak szét: az első csoportba tartoztak a dupla tagadások (például „nem zárták ki”), kétértelmű megfogalmazások („jelentéktelen”), a másodikba sorolták a valódi tagadásokat. Az alkalmazott két reguláris kifejezés a következő volt:

- (1) <tagadásos kifejezés> \* <UMLS kifejezés>
- (2) <UMLS kifejezés> \* <tagadásos kifejezés>

Mindkét esetben a „\*”-gal jelölt helyre 0-5 szó helyettesíthető be.

A kiértékeléshez 1000 darab orvosi dokumentumot vizsgáltak meg mindkét algoritmussal, és arra a következtetésre jutottak, hogy a NegEx magas precizitása (94%) miatt a kiválasztott tagadások aránya nagyobb, mint a alapalgoritmus esetében (85%), viszont a tervezése miatt alacsonyabb az érzékenysége (77% - NegEx és 88% - alapalgoritmus), mivel nem ismeri fel a valódi tagadásokat ugyanazon mondat esetében (az UMLS kifejezéshez viszonyított távolsága miatt).

## 2.2. NegFinder algoritmus

A Mutalik et al. által kifejlesztett algoritmus egy más típusú tagadásra épül, mint az előző fejezetben bemutatott algoritmus. [7] Ők egy lexikai elemzőt használtak fel, amely többnyire formális nyelvek esetében alkalmazható. A rendszer egy csővezetéken áthaladó (pipeline) komponenseken alapul, amelyek rendre hozzájárulnak a megoldás meghatározásához. Minden lépésben az eredeti dokumentum egy átalakuláson halad keresztül, amelynek eredményeként az algoritmus megjelöli a tagadásokat. A következő 4 komponenst definiálták:

- (1) Kulcsszó keresése: ide tartozik az UMLS kifejezés detektálása és feltüntetése a kimenetben, a mondatban előforduló kifejezéssel és szövegbeli pozícióval együtt.
- (2) Bemenet átalakítása: az eredeti dokumentumban megjelenő UMLS kulcsszót tartalmazó kifejezés felcserélődik a neki megfelelő UMLS kulcsszó azonosítójával.
- (3) Elemzés: az előző lépés eredménye átadódik egy lexikális elemzőnek, amely egy nagy mennyiségű tagadási kulcsszó halmazt tartalmaz. Ez az elemző meg tudja állapítani, hogy a tagadás alanya a kulcsszó előtt vagy után helyezkedik el és hány alanyra vonatkozik. Az elemző alkalmazza a nyelvtani szabályokat ahhoz, hogy a tagadási kulcsszavakat összekösse a kifejezésekkel, amelyekre vonatkoznak. A lépés kimenete tartalmazza a tagadásra vonatkozó adatokat.
- (4) Ellenőrzés: az eredeti dokumentum megjelölésre kerül különböző színek által, amelyet később egy személy vizsgál meg.

A tesztelést 60 orvosi dokumentummal végezték el először a tervezett algoritmus futtatásával, majd egy független szakembert bíztak meg a dokumentumok kiértékelésével. Az algoritmus érzékenysége és specifikussága nagyon közel állt a személyes megfigyelőjével összehasonlítva (érzékenység 95,3% illetve 95,7%, specifikusság 97,7% illetve 91,8%), amely azt igazolja, hogy a már megjelölt dokumentumok kissé eltérítik az elemző meglátását. A NegFinder algoritmust alkalmazták a MediClass tudásbázis rendszerben, amely akár szabadszöveg, akár struktúrált szöveg feldolgozását is lehetővé teszi. [8]

## 2.3. Gyógyszerkölsönhatások felismerése a DrugDDI adathalmazban

Ebben a munkában a szerzők gyógyászati szövegekben megjelenő gyógyszerek közötti kölcsönhatások kimutatását célozták meg. [9] Rámutattak a rendszer szükségszerűségére, mivel több gyógyszeradathalmaz létezik a piacon, azonban ezeknek a frissítése, egyesítése nehéz feladat a szakemberek számára. Amint azt az előző munkák is kiemelték, a tagadások fontos szerepet töltenek be, mert a gyógyászati szakemberek gyakran tagadásokat foglalnak bele a orvosi szövegekbe a valós gyógyszerek feltüntetése mellett. Ezeknek a figyelmen kívül hagyása hamis találatok (false positive) detektálásához vezethet.

A munkában a szerzők felhasználták a DrugDDI adathalmazt, amelyet elsőként fejlesztettek ki gyógyszerkölsönhatások kimutatására. [10] Ennek az adatforrása a DrugBank adatbázis volt, amely struktúrátlan formában tartalmaz információt gyógyszer-gyógyszer kapcsolatokról és kölcsönhatásokról. A DrugDDI adathalmazban a gyógyszerek hivatkozásokat tartalmaznak az eredeti szövegben megjelenő mondatra. Minden lehetséges kapcsolatpár fel van tüntetve a mellékelt XML dokumentumban a megfelelő attribútummal együtt, amely jelzi, hogy a kölcsönhatós valós-e vagy sem.

A DrugDDI adathalmazban a szerzők megjelölték a tagadásokat: először bizonyos tagadási indítékokra kerestek (például „no” vagy „not”), majd meghatározták a tagadásos kifejezések hatókörét. Felhasználták az összes 5806 darab mondatot, amelyből 1340 tartalmazott tagadást. A mondatokat elemző személyek is végigvizsgálták, és 350-et találtak, amelyeket az algoritmus hamisan sorolt a tagadások csoportjába.

A kölcsönhatások megállapítása folyamán a Stanford elemzőt használták tokenizálásra [11], a tanulási motort a Weka SVM (support vector machine) szolgáltatta. Az eredmények azt bizonyították, hogy a tagadási funkciók alkalmazása javította az algoritmus teljesítményét.

## 3. Elméleti alapok

### 3.1. Természetes nyelv elemzés

Mivel a dolgozat kulcsfeladata struktúrátlan szövegben előforduló gyógyszerkifejezések és betegségek, tünetek megállapítása, ezért szükség volt olyan módszereket használnom, amelyek a természetes nyelv elemzésének témakörébe tartoznak (Natural Language Processing, NLP). Ez a szakterület a természetes, emberi nyelv gépi úton történő feldolgozásával foglalkozik, és magába foglalja nyelvi kifejezések, struktúrák értelmezését, két vagy több kifejezés egymáshoz viszonyított kapcsolatának meghatározását, stb. Különböző módszerek léteznek a struktúrátlan szöveg elemzésére, értelmezésére, viszont általában minden egyes módszer alapja valamilyen előfeldolgozás, mint a tokenizálás vagy normalizálás (mondatok szétválasztása, szótő meghatározása, szótárbeli szótípusok és tokenek meghatározása, stb.). Normalizálás alatt értjük ekvivalencia osztályok definiálását, amely során például eltávolítjuk a fölösleges írásjeleket (pont, vessző, idézőjel, stb.), nagybetűket kisbetűkkel helyettesítjük. Ide tartozhat ugyanakkor a szavak végének törlése (stemming), amely esetén minden szónak csak a szótővét őrizzük meg, és így alkalmassá tesszük a keresett szavakat az adatbázisrendszerben eltárolt megfelelőivel történő összehasonlításra.

A minimális változási távolság (minimum edit distance) meghatározása abban az esetben

indokolt, amikor két kifejezést szeretnénk egymással összehasonlítani. Jelen dolgozat esetében szükség volt egy ilyen mérték kiszámítására, mivel a DBPedia által szolgáltatott eredmények közül ki kellett választani a keresett kifejezéshez legközelebb állót (például az „urticaria” kifejezés keresése esetén a visszatérített értékek között szerepelt a „cold urticaria” és az „urticaria” kifejezés is, én pedig csak az utóbbit akartam megjeleníteni). Legrövidebb távolság meghatározására a Levenshtein algoritmust használtam fel, amely dinamikus programozásra alapozva határozza meg a két kifejezés közötti legrövidebb utat. Egy betű törlésének, illetve beszúrásának költsége 1 egység, viszont egy betűnek valamely másra cserélésének költsége 2 egységet jelent. Az algoritmus úgy működik, hogy az egyik szó minden részegységétől meghatározza a másik szó minden részegységéig a legrövidebb utat, majd az újabb távolság meghatározásához felhasználja az előző részfeladatok eredményét.

Osztályozási algoritmusok kiértékeléséhez gyakran használnak valamilyen eszközt, amely kifejezi annak teljesítményét. Találatok esetében meg kell különböztetnünk a helyes találatokat a helytelen találatoktól:

True positive	Helyes találat (létezik valós gyógyszerkölcsonhatás, az algoritmus helyesen észleli)
True negative	Helyes elutasítás (nem létezik valós gyógyszerkölcsonhatás, az algoritmus helyesen észleli)
False positive	Hamis találat (nem létezik valós gyógyszerkölcsonhatás, az algoritmus helytelenül téríti vissza a találatot)
False negative	Hamis elutasítás (létezik valós gyógyszerkölcsonhatás, az algoritmus helytelenül utasítja el a találatot)

### 3.1. Táblázat: Találatok minősítése

Ahhoz, hogy az algoritmus teljesítményét megállapítsuk, két mértéket használhatunk fel:

$$(1) \text{ precizitás (precision) } = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalsePositive}}$$

$$(2) \text{ érzékenység (recall) } = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}}$$

Mondattani elemzés a Stanford elemzővel végezhető, amelynek segítségével kimutathatók a mondatban szereplő releváns kifejezések (gyógyszernevek, betegségek, tünetek) közötti kapcsolatok. [11] Ez az eszköz több lépést biztosít a szöveg feldolgozására: tokenizálás, mondatok szétválasztása, szövegrészek megjelölése (Part-of-speech Tagging), morfológiai elemzés, stb. Használata rendkívül egyszerű: egy egységes interfészt biztosít (API) natív Java programozási nyelvben, illetve terminálon keresztül is alkalmazhatjuk szövegelemzésre. Egy csővezeték típusú (pipeline) architektúrára épül, amely több komponensből áll, és az elemzendő szöveget lépésenként értelmezi. A megfelelő komponenseket (annotators) a felhasználó paraméterként definiálhatja (például tokenize, cleanxml, ssplit, stb.) használva a Java nyelv által támogatott mezőket vagy a terminálon keresztül kapcsolókat határozhat meg a feldolgozásra.

## 3.2. MetaMap

A MetaMap egy népszerű és rendkívül hasznos eszköz orvosi UMLS (unified medical language system) kifejezések felismeréséhez. [12] Az UMLS egy olyan rendszer, amely magában foglal több egészségügyi és orvosi szótárt (vocabulary) ahhoz, hogy megkönnyítse a különböző informatikai rendszerek közötti kommunikációt. Ebben a rendszerben megtalálhatók különböző orvosi kifejezések, gyógyszernevek, betegségek, tünetek nevei, amelyeket keresésekre, adatbányászatra vagy akár egészségügyi statisztikák előállítására is felhasználhatunk.

Az eszköz orvosi szövegek elemzése céljából jött létre azért, hogy kapcsolatot biztosítson a bemeneti szöveg és tudásbázis között. Akár a Stanford elemző, a MetaMap esetében is az értelmezendő szöveg több lépésen halad keresztül mint a tokenizálás, szótó meghatározása, szótárbeli keresés vagy ellentétek felismerése.

Három egységből áll:

1. Metathesaurus: kifejezéseket és kódokat tartalmaz több szótárból,
2. Szemantikus hálózat: csoportokat (szemantikus típusokat) és a közöttük fennálló kapcsolatokat tartalmazza,
3. Specialist lexikon: természetes nyelv feldolgozó eszközöket foglal magában.

Az elemzés alatt a megtalált kifejezés több lépésen keresztül értékelődik ki, amelyek során a kifejezés összehasonlításra kerül az adatbázisban tárolt értékekkel, a rendszer kiválasztja a lehetséges kandidátusok listáját, majd végül azok a kifejezések kerülnek kiírásra, amelyek szemantikailag összeférhetők egymással.

Az eszköz használható a Java nyelvben támogatott fejlesztői interfészen (API) vagy terminálon keresztül. Az alkalmazás implementálása során ez utóbbit használtam fel, amely több kapcsoló által biztosítja a elemzés testreszabását. A futtatás a következő paranccsal történik:

```
echo cancer | metamap [ options ]
```

Ebben az esetben az echo program segítségével a „lung cancer” kifejezést adjuk a MetaMap programnak, majd ez után különböző opciók közül választhatunk. Az előbbi parancs eredménye a következő volt:

Phrase: lung cancer

Meta Mapping (1000):

1000 LUNG CANCER (Carcinoma of lung) [Neoplastic Process]

Meta Mapping (1000):

1000 Lung Cancer (Malignant neoplasm of lung) [Neoplastic Process]

Az alkalmazásban a következő kapcsolókat használtam fel:

-J sosy,dsyn,orch,phsu,inpo

Ezen kapcsoló által megadhatók azon szemantikus típusok, amelyeket fel szeretnénk ismerni a bementi szövegben. A keresett szemantikus típusoknak megfelelő rövidítések listája megtalálható online [12].

--negex

Ez a kapcsoló lehetővé teszi, hogy a kimenetben fel legyenek tüntetve a tagadásos UMLS kulcsszóhoz fűződő kifejezések. Ezen opció meghatározásával, az eszköz használja a NegEx algoritmust ahhoz, hogy felismerje a tagadásokat a klinikai szövegben.

### 3.3. Hadoop és Pig lekérdezések

A Hadoop egy összetett osztott rendszer, amely nagy mennyiségű adathalmaz kezelését és feldolgozását teszi lehetővé. Legfontosabb komponensei közé tartozik a HDFS (Hadoop Distributed File System), amely egy önálló osztott operációs rendszer. Ezt úgy tervezték meg, hogy nagyon nagy mennyiségű adat kezelését tegye lehetővé, és mindezt folyamszerűen használja (streaming access), ami azt jelenti, hogy előnyben részesíti a teljes adat olvasását a legelső adategység eléréséhez viszonyítva. Az adathalmaz egyszer íródik be a rendszerbe, majd ezen különböző feladatokat hajthatunk végre. Meg kell jegyezni, hogy a HDFS rendszeren végrehajtott műveletek futtatása kötegekben (batches) történik, így ezek hosszabb ideig is eltarthatnak. A rendszer magasfokú elérhetőséget biztosít (High Availability), csomópont meghibásodása esetén a másodlagos csomópont veszi át az előző szerepét.

Egy Hadoop rendszerben 3 típusú csomópontot különböztetünk meg: kliens csomópont (client node), mester csomópont (master node) és szolga csomópont (slave node). A mester csomópontok feladata az, hogy felügyeljék az adat tárolást a HDFS fájlrendszerben és koordinálják a MapReduce feladatok ütemezését a szolga csomópontok által. A cluster nagy részét a szolga csomópontok teszik ki, amelyek a valódi adattárolást és feladatok végrehajtását végzik. A feladat (job) elindítása a kliens csomópontban történik, amely blokkokra osztja szét a nagy mennyiségű adathalmazt, és egy névcsomópont konzultálásával továbbküldi a szolga csomópontoknak. Ezek a maguk rendjén replikálják az adatblokkot, így ugyanazon adatblokk több csomóponton fog megisméltódni a hibatűrés biztosítása miatt (alapértelmezésben a replikált blokkok száma 3).

A Hadoop keretrendszer legfontosabb komponensei a következők:

1. Hadoop Common: közös függvénykönyvtárakat és eszközöket tartalmaz, amelyeket a többi modul használ fel,
2. Hadoop Distributed File System: osztott rendszer, amely közönséges hardvereket használ, és nagy mennyiségű adatátvitelt biztosít az összetett egységen (cluster),
3. Hadoop YARN: erőforráskezelő platform, amely rendszerezi a cluster-t alkotó számítógépeket, és ütemezi a különböző feladatok végrehajtását,
4. Hadoop MapReduce: programozási modell nagy mennyiségű adat kezelésére. [13]

A MapReduce egy párhuzamos programozási modell, amely nagy mennyiségű adat feldolgozását biztosítja a cluster-t alkotó számítógépeken. Az algoritmus két fő részből áll:

- a. Map: ebben a fázisban kiválasztjuk a bemeneti adathalmaz minden egységéből (sorából) a számunkra releváns információt,
- b. Reduce: miután a keretrendszer elvégezte a Map fázis végén az összesítést (csoportosítás, rendezés) minden egyes kulcsra, meghatározzuk, hogy milyen műveletet szeretnénk végezni a közös kulcsú értékekkel (például minimum kiválasztása).

Mindkét fázis bemenete és kimenete egy kulcs-érték páros, amelynek típusait a programozó határozhatja meg.

A Pig egy eszköz, amely a Hadoop rendszeren fut, és párhuzamos adatfeldolgozást valósít meg a HDFS és MapReduce feladatok segítségével. A Pig Latin nyelv által biztosítja a MapReduce műveletek definiálását úgy, hogy közben a programozót mentesíti a bonyolult adatfolyamok meghatározásától. A háttérben lefordítja MapReduce feladatokra (jobs) a programozó által megadott műveleteket. Az adatbázisok világában ismert SQL nyelvre hasonlít, de ugyanakkor el is tér tőle több szempontból, mivel a Pig Latin-ban meg tudjuk határozni, hogy hogyan szeretnénk feldolgozni az adatokat. Az SQL nyelvben deklaratív módon írjuk le a lekérdezéseinket anélkül, hogy meghatároznánk a feldolgozás lépéseit, a Pig Latin viszont megköveteli a lépések pontos meghatározását.

A gyógyszerek feldolgozása során a csoportosítanom kellett a gyógyszereket minden egyes összetevőre nézve. Ebben az esetben ez a csoportosító feladat ideálisnak bizonyult a MapReduce algoritmus által, amely képes több számítógépet igénybe venni a feladat megoldásához.

A mongo-hadoop meghajtó segítségével adatokat olvashatunk és írhatunk MongoDB adatbázisból Pig szkriptek segítségével. Meghatározhatunk külső függvényeket is, amelyek elsődleges feldolgozást végeznek az adatbázisból beolvasott sorokon. Ezen függvényeket Javascript nyelvben implementáltam, amelyben a Java nyelvben biztosított programozási interfészen (API) keresztül végeztem átalakításokat a bemeneti sorokon. A Pig szkriptnek szüksége volt a bemeneti sorok egyes mezőinek típusára ahhoz, hogy további műveletet hajtson végre rajtuk (például GROUP BY vagy FLATTEN).

### 3.4. SPARQL

A SPARQL (SPARQL Protocol and RDF Query Language) magában foglalja a SPARQL protokollt és az RDF lekérdező nyelvet. A SPARQL protokoll meghatározza, hogy a kliens alkalmazás hogyan kommunikál a SPARQL szerverrel. Az RDF egy modell, amely adatok tárolását teszi lehetővé adathármasok (alany, állítmány, objektum) formájában.

Alany	Állítmány	Objektum
< <a href="http://dbpedia.org/page/Urticaria">http://dbpedia.org/page/Urticaria</a> >	dbp:field	dbr:Dermatology
< <a href="http://dbpedia.org/page/Urticaria">http://dbpedia.org/page/Urticaria</a> >	foaf:name	Urticaria (en)

3.2. Táblázat: Példa RDF hármasokra a DBPedia adatbázisában

A SPARQL része a szemantikus webnek, amely az adatok könnyű és hatékony megosztását, értelmezését célozza meg gépi feldolgozás által. Ezen modell alapján struktúráltan tárolhatjuk el az adatokat, és biztosíthatjuk azok megosztását és feldolgozását különböző funkciójú rendszerek között.

Az implementált alkalmazásban a MetaMap eszköz segít a gyógyászati kulcsszavak (betegségek, tünetek) felismerésében, viszont nem biztosít egy rövid leírást róluk. Ezért felhasználtam a DBPedia [14] által fenntartott SPARQL végpontot (<http://dbpedia.org/snorql>), amely SPARQL lekérdezések végrehajtására ad lehetőséget egy REST interfészen keresztül. A DBPedia projekt a Wikipedia portál által használt adatbázist teszi elérhetővé struktúrált RDF hármasok formájában.

Az alkalmazás a lekérdezéseket kliens oldalon küldi el a SPARQL végponthoz valós időben,



amikor a MetaMap által talált eredmények megérkeznek a böngészőbe. Ez által nincs szükség ezen adatok lokális eltárolására.

### 3.5. Reguláris kifejezések

A reguláris kifejezéseket intenzíven használják természetes nyelven írt szövegekben előforduló szabályos minták felismerésére. Ezek a kifejezések egy speciális kifejezést tartalmaznak, amelyek egy mintát írnak le, és a bemeneti szöveg több részegységére illeszkedhetnek.

Megjegyeztem, hogy a NegEx algoritmus két reguláris kifejezésen alapul, amelyek tagadásos mondatrészekre illeszkednek a bemeneti szövegben:

(1) <tagadásos kifejezés> \* <UMLS kifejezés>

(2) <UMLS kifejezés> \* <tagadásos kifejezés>

Ezekben a mintákban kétféle kifejezésre keresünk: egyrészt egy tagadásos kifejezést lokalizálunk a bemeneti szövegben (például a „**nem** okozott **fejfájást** az **aszpirin** szedése” szövegben a tagadásos kifejezés a „nem”), majd megkeressük az összes UMLS kifejezést, amely utána következik vagy előtte szerepel (az előbbi példa esetében a „fejfájás” illetve „aszpirin” az UMLS kifejezésre illeszkedik a reguláris mintában). A „\*” kifejezés több különálló szóra illeszkedhet, amely nem UMLS kifejezést jelöl. A NegEx algoritmus esetében ezen szavak száma legfeljebb 5 lehet.

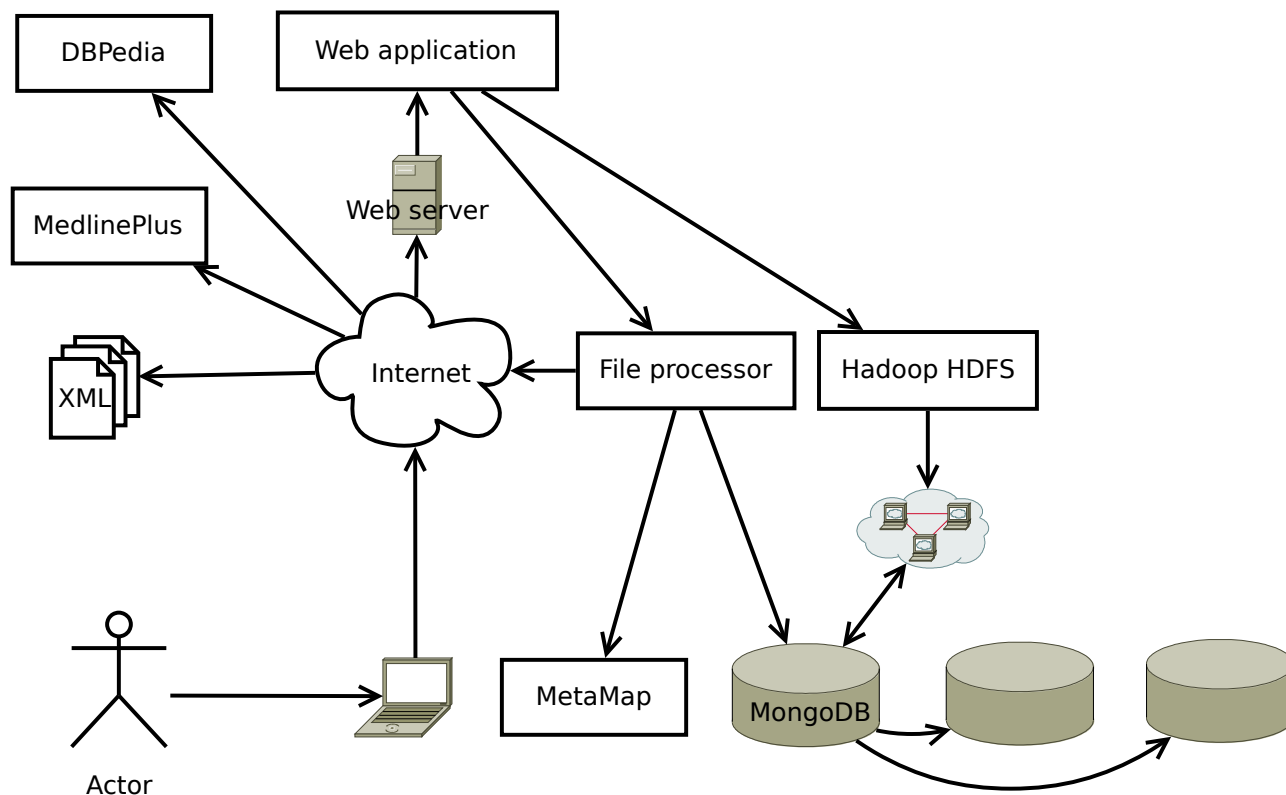
Az alkalmazásban felhasználtam azt a heurisztikus függvényt, amely bizonyos előre meghatározott szavakra keres a bemeneti szövegben. Ilyenek voltak a következő kifejezések:

- Coadministration (együttes használat)
- Concomitant (együttes használat vagy valami követő használat)

Amennyiben a bemeneti mondatban a felismert gyógyszer neve mellett előfordul az előző szavak valamelyike, akkor a talált gyógyszert eltároltam mint lehetséges kölcsönhatásban álló példányt, mivel ezek jelentése az együttes használatra utal.

## 4. Az algoritmus működési elve

Az alkalmazás több különböző rendszert foglal magában, amelyek integrálása és közreműködése együtt szolgáltatja a probléma megoldását. A következő ábra egy átfogó képet nyújt az alkalmazáson belül használt technológiákra és komponensekre.



4.1. Ábra: Az alkalmazást alkotó komponensek és kapcsolataik

A fenti ábra szemlélteti a rendszert alkotó komponenseket és a közöttük fennálló kapcsolatokat. A rendszer alapjában véve két részből áll:

- adatfeldolgozó komponens
- megjelenítő komponens

A megjelenítő komponenst a következő fejezetben részletezem, amikor az implementációs sajátosságokra kerül sor. Az adatfeldolgozó komponens futtatása teljes mértékben a szerveren történik, és a következő lépéseket foglalja magában:

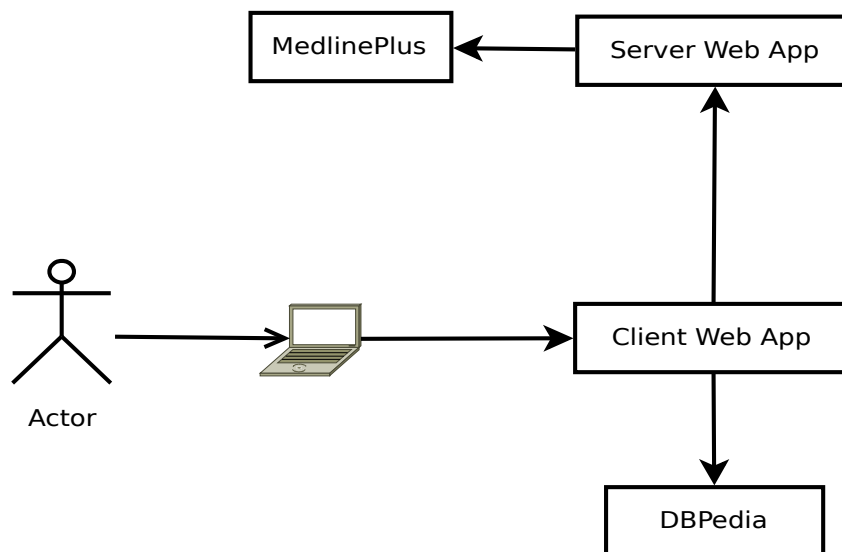
1. beolvassuk a bemeneti könyvtár tartalmát és kicsomagoljuk a tömörített fájlokat
2. minden azonosított XML fájl elemzésre kerül: a releváns adatokat (gyógyszer neve, gyártója, alapanyagai, használati útmutatók, megjegyzések, stb.) eltároljuk az adatbázisban
3. miután az összes XML dokumentum elsődleges elemzése és adatbázisba történő beszállása megtörtént, megvizsgáljuk egyenként az összes gyógyszer entitást. Minden gyógyszerre végrehajtjuk a következő lépéseket:
  - 3.1. kiolvassuk a gyógyszerrel eltárolt útmutatókat, megjegyzéseket, amelyek struktúrátlan szöveggént jelennek meg

- 3.2. minden egyes struktúrálatlan bejegyzést átadunk a MetaMap eszköznek, amely feldolgozza a szöveget, és XML dokumentum formájában szolgáltatja az eredményt
- 3.3. elemezzük az eredményt, és frissítjük a gyógyszer tároló sort az adatbázisban
4. az elemzés után, végrehajtjuk rendre a PIG lekérdezéseket, amelyek előállítanak rendre egy egy táblát a gyártóknak, alapanyagoknak és betegség vagy tünet találatoknak

Egy tipikus gyógyszer leíró XML fájl, amely a DailyMed portálról tölthető le, tartalmazza a gyógyszer nevét, gyártóját, összetevőit, útmutatókat, figyelmeztetéseket, stb.

```
<?xml version="1.0" encoding="UTF-8"?><?xml-stylesheet
href="http://www.accessdata.fda.gov/spl/stylesheet/spl.xml" type="text/xsl"?>
<document xmlns="urn:hl7-org:v3" ...>
  <id root="eb1f77e0-a70f-470b-9199-1b9b8d6cc4fa"/>
  <code code="34391-3" displayName="HUMAN PRESCRIPTION DRUG LABEL"
codeSystem="2.16.840.1.113883.6.1"/>
  <title>
    <br/>These highlights do not include all the information needed to use Warfarin Sodium safely and effectively.
See full prescribing information for Warfarin Sodium.<br/>
    <br/>Warfarin Sodium (Warfarin) TABLET for ORAL use.<br/>Initial U.S. Approval: 1954
  </title>
  ...
  <ingredient classCode="ACTIB">
    <quantity>
      <numerator value="3" unit="[hp_X]"/>
      <denominator value="1" unit="mL"/>
    </quantity>
    <ingredientSubstance>
      <code code="5EF0HWI5WU" codeSystem="2.16.840.1.113883.4.9"/>
      <name>BAPTISIA TINCTORIA ROOT</name>
      <activeMoiety>
        <activeMoiety>
          <code code="5EF0HWI5WU" codeSystem="2.16.840.1.113883.4.9"/>
          <name>BAPTISIA TINCTORIA ROOT</name>
        </activeMoiety>
      </activeMoiety>
    </ingredientSubstance>
  </ingredient>
  ...
  <component>
    <section ID="ID_71cd2e00-0c94-4b47-8f16-96ef2f99fb53">
      <id root="71cd2e00-0c94-4b47-8f16-96ef2f99fb53"/>
      <code code="34071-1" codeSystem="2.16.840.1.113883.6.1" displayName="WARNINGS SECTION"/>
      <title>Warnings</title>
      <text>
        <paragraph>
          <content styleCode="bold">Allergy alert</content>: Ibuprofen may cause a severe allergic reaction,
especially in people allergic to aspirin. Symptoms may include: ■ hives ■ facial swelling ■ asthma (wheezing) ■
shock<content styleCode="bold"/>■ skin reddening <br/>■ rash ■ blisters<br/>If an allergic reaction occurs, stop use
and seek medical help right away.<br/>
          <content styleCode="bold">Stomach bleeding warning:</content>This product contains a
nonsteroidal anti-inflammatory drug (NSAID), which may cause severe stomach bleeding. The chances are higher if
you: ■ are age 60 or older ■ have had stomach ulcers or bleeding problems ■ take a blood thinning (anticoagulant) or
steroid drug ■ take other drugs containing prescription or nonprescription NSAIDs (aspirin, ibuprofen, naproxen, or
others) ■ have 3 or more alcoholic drinks every day while using this product ■ take more or for a longer time than
directed.</paragraph>
        </text>
      </text>
    </section>
  </component>
  ...
```

Az alábbi ábrán megjelennek a DBPedia és MedlinePlus komponensek, amelyek webes szolgáltatások formájában biztosítanak hozzáférést gyógyászati kifejezések értelmezéséhez.



4.1. Ábra: Felhasználó által történő keresést megvalósító komponensek

A felhasználó választásának függvényében ezen két szolgáltató vehető igénybe a gyógyászati kifejezések (betegségek és tünetek) értelmezésére. A DBPedia által támogatott SPARQL végponthoz a böngészőalkalmazás küld lekérdezéseket, amelyek eredménye valós időben kerül kiértékelésre és megjelenítésre a felhasználó böngészőjében. A MedlinePlus egy webes szolgáltatás keretén belül biztosít hozzáférést gyógyászati kifejezésekhez: a lekérdezéseket szintén valós időben végezzük a böngésző alkalmazáson belül az által, hogy elküldjük rendre a keresett kifejezéseket a webszerviznek.

A felhasználó bejelentkezése után, valamely kifejezésre történő keresés a következő lépéseket eredményezi:

1. elküldjük a keresett kifejezést a web szerverhez
2. a válasz megérkezésekor megjelenítjük a gyógyszer adatait a web oldalon
3. a DBPedia szolgáltatás kiválasztása esetén a következő lépéseket hajtjuk végre:
  - 3.1 dinamikusan előállítjuk a SPARQL lekérdezést
  - 3.2 elküldjük a SPARQL végponthoz
  - 3.3 a válasz megérkezése esetén kiválasztjuk a visszatérített listából a keresett kifejezéshez legközelebb állót, és megjelenítjük a weboldalon
4. a MedlinePlus szolgáltatás esetén, kérést küldünk a szerver alkalmazásnak a keresett kifejezéssel
  - 4.1 a szerver alkalmazás elküldi a lekérdezést a MedlinePlus webszerviznek
  - 4.2 a válasz megérkezése esetén továbbítjuk a választ a böngésző alkalmazásnak
  - 4.3 a böngésző alkalmazás frissíti a web oldalt a találatokkal

## 5. A rendszer megvalósítása

Ebben a fejezetben részletes leírást nyújtok az implementálás sajátosságairól. Mivel több technológiát használtam fel a megvalósítás során, ezért több lépésben fogom ismertetni a programot. Meg kell jegyezmem, hogy fontos szempontnak tekintettem a moduláris programozás elveit, és törekedtem arra, hogy egy könnyen áttekinthető kódbázist hozzak létre, amely továbbfejleszthető és karbantartható. Fontos szempontnak tartottam, hogy betarttam az ún. SOLID programozási elveket, amelyek megkövetelik a program komponenseinek jól meghatározott szétválasztását és elhatárolását egymástól, a komponensek egyedi felelősségeinek érvényesítését, a könnyű bővíthetőség lehetőségét, a magasabb rendű modulok függetlenségét az alacsonyabb rendű moduloktól. Fontos szerepet kapott a tesztelhetőség az alkalmazás megírása során: úgy építettem fel az egyes komponenseket, hogy azok könnyen tesztelhetők legyenek, és így növeljék a kód minőségét és megbízhatósági szintjét.

Első lépésben megfogalmaztam a követelményeket, amelyeket az alkalmazásnak ki kell elégítenie.

### 5.1. Követelmények

A bevezetésben illetve az algoritmus leírásának fejezeteiben kitértem a megoldandó probléma megfogalmazására, most viszont formálisan rögzíttem az elvárásokat és követelményeket:

- Mivel az Internet korában élünk, egy olyan alkalmazást szeretnénk megvalósítani, amely könnyen elérhető a felhasználó számára, legyen az személyi számítógép (PC), táblagép vagy akár mobiltelefon. Azt állíthatjuk, hogy manapság mindenki rendelkezik hordozható számítási funkcióval rendelkező eszközökkel, ezért ez egy alapkövetelmény, mivel a programot nem csak a gyógyászati szakemberek számára szeretnénk biztosítani, hanem az átlagfelhasználót, beteget is megcélozzuk ezzel a termékkel. Az alkalmazás felületét (User Interface) webes böngészővel érhetjük el, amelynek célja, hogy könnyű kezelhetőséget biztosítson eszköztől függetlenül.

- Az alkalmazás felhasználónév és jelszó által védett kell legyen, és erre egy külön oldalt kell biztosítani, amely lehetővé teszi a bejelentkezést. A bejelentkezés történhet feliratkozás által vagy a felhasználó Facebook profiljának a segítségével.

- A feliratkozás egy külön oldal által kell történnjen, ahol a felhasználó meg kell határozza a kívánt nevet, jelszót, keresztnévet, családnévet, születési évet, e-mail címet és szemantikus keresési modult (lásd a következő követelményekben megfogalmazva, hogy mire szolgál ez).

- Két típusú felhasználót kell támogasson a rendszer: rendszergazda (admin) és egyszerű felhasználó. Az admin mindenre képes, amire az egyszerű felhasználó képes, viszont lehetősége van a DailyMed portál által közzétett adatok beimportálására, amely egy újabb weboldal által kell elérhető legyen.

- A rendszergazda által elérhető oldal biztosít egy nyomógombot, amelyet használva a rendszer elindítja a fájlok feldolgozását egy meghatározott könyvtárból. Ez egy batch alapú művelet lesz, amelyet a Hadoop rendszer által fogunk elvégezni, és több ideig is eltartathat a végrehajtása, tehát nem kapunk egy azonnali visszajelzést az eredményről, viszont biztosítani kell egy haladási sávot (progressbar) az oldalon, amely folyamatosan frissül a művelet végrehajtásának következtében. Ha a felhasználó egy más oldalra navigál, biztosítani kell, hogy értesítést kapjon a művelet eredményéről. Ez történhet úgy, hogy valamilyen buborékos megjelenítési eszközt használunk, amely előbukkan az oldalon, és a felhasználó rá kattinthat.

– Az alkalmazásnak biztosítania kell egy keresési oldalt, ahol megadható a keresendő minta és keresési típus: gyógyszer neve, alapanyag neve vagy gyártó neve. A keresendő minta egy szöveges doboz (textbox) által kell megjelenjen, és törölhető kell legyen gombnyomásra. Amennyiben a keresés gyógyszerre irányul, a megjelenítendő mezők között kell szerepeljen: gyógyszer neve, gyártó neve, alapanyagok listája (vesszővel elválasztva vízszintes vagy függőleges irányban). Ezeket az adatokat egy táblázatban foglaljuk össze, és minden sorban biztosítunk egy nyomógombot, amelyet megkattintva egy dialog típusú ablakot jelenítünk meg, amely részletes leírást ad a gyógyszerről. Itt megjelenítjük az említett mezők mellett a figyelmeztető tájékoztatást az egyes gyógyszerekről. Ezek alatt pedig egy listában foglaljuk össze a szövegben felismert betegségeket, tüneteket, és a kölcsönhatásban álló egyéb gyógyszereket. Minden kulcsszó biztosítson egy linket, amelyet kiválasztva részletes leírást kaphatunk egy külső weboldal által. Gyógyszerek esetében biztosítani kell, hogy amennyiben rákattint a felhasználó, egy keresést végzünk az adatbázisban, és az eredményeket az említett keresési oldalon jelenítjük meg.

– Biztosítsuk, hogy a keresések során használni tudjuk a böngésző előre, illetve vissza gombjait (Back és Forward) azért, hogy visszatérjünk egy előző keresésre.

– Alapanyagra keresve, listázzuk ki az összes olyan gyógyszert, amely tartalmazza a keresett alapanyagot; gyártóra keresve, listázzuk ki a gyártó által előállított gyógyszer termékeket. A gyógyszer, alapanyag és gyártó nevek jelenjenek meg követhető linkek formájában, amelyeket követve újabb keresést tudunk végezni.

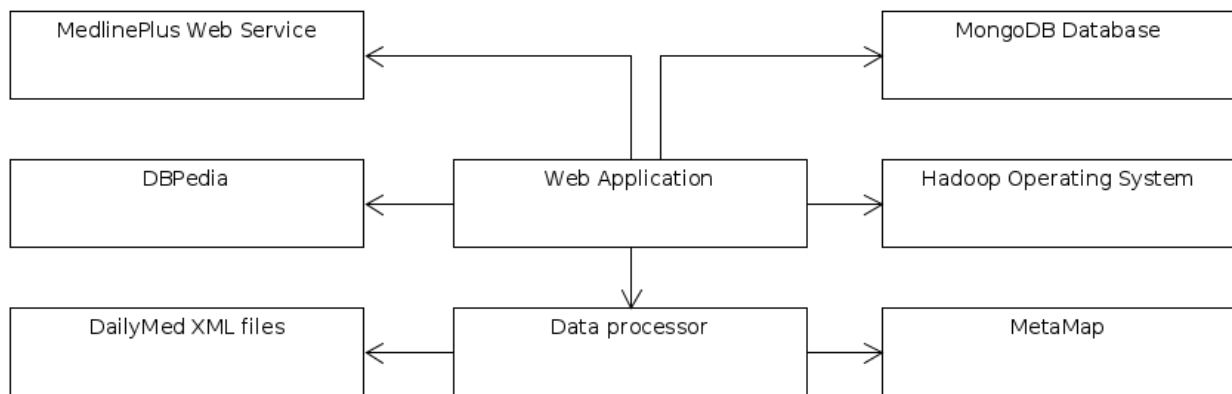
– A MetaMap által felismert kulcsszavak esetében végezzünk keresést egy külső tudásbázisban (MedlinePlus vagy DBPedia) a felhasználó választása szerint. Ezt a választási lehetőséget a felhasználó profiljának oldalán tegyük lehetővé. Legyen lehetőség a felhasználó adatainak megváltoztatására (email cím vagy jelszó megváltoztatása, illetve keresési motor kiválasztása). A tudásbázisban történő keresést valós időben végezzük a böngészőben, amikor a keresési eredmények megérkeznek.

– Biztosítsunk egy oldalt, ahol a felhasználó megtekintheti a keresési előzményeket és újra végrehajtja őket. Jelenítsük meg a keresés típusát, kulcsszót, illetve dátumot. Legyen lehetőség a bejegyzést törölni.

– Biztosítsunk a képernyőn egy nyomógombot, amellyel kijelentkezhetünk az alkalmazásból. Védjük le az alkalmazást felhasználónév és jelszó által. Feliratkozás esetén tároljuk el a felhasználó bejelentkezési adatait a lokális adatbázisban, Facebook profil esetében vegyük igénybe a Facebook bejelentkezési mechanizmusát. Legyen lehetőség bejelentkezve maradni megszakított munkamenet esetében (például lezárjuk a böngészőt majd ha később újraindítjuk, ne kelljen a bejelentkezési adatokat újra megadni).

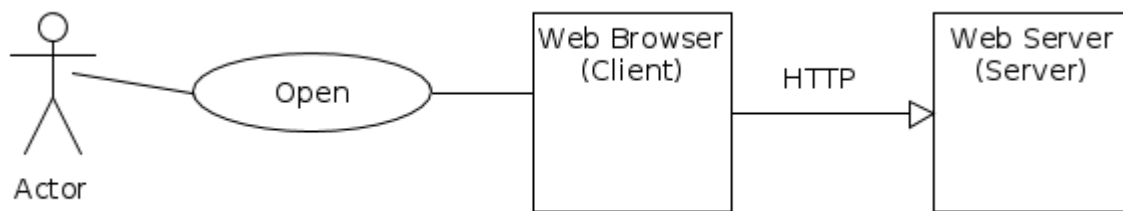
## 5.2. Az alkalmazás felépítése, architektúrája

Az előző részben összefoglaltam az alkalmazás követelményeit, most rátérek a különböző eszközök, programnyelvek, programkönyvtárak ismertetésére, amelyekre szükség volt a megvalósítás során, megindokolva az egyes termékek megválasztását és szükségszerűségét. Még mielőtt az eszközöket ismertetném, tekintsünk meg egy átfogó képet az alkalmazás komponenseiről egy ún. Road map diagram formájában.



5.1. Ábra: Az alkalmazás Road map diagramja

Ez a diagram magában foglalja az alkalmazás minden komponensét, és megmutatja az egyes modulok közötti kapcsolatokat is. A következőkben ezen diagram komponenseit fogom ismertetni, helyenként hivatkozva erre az ábrára. Az alkalmazás követelményeiből kiderült már, hogy egy webes alkalmazást szeretnénk megvalósítani, amely könnyen elérhető különböző eszközökről. A webes alkalmazás két részből áll: egy szerver oldali, illetve egy kliens oldali rész. Ezt a következő ábra szemlélteti:



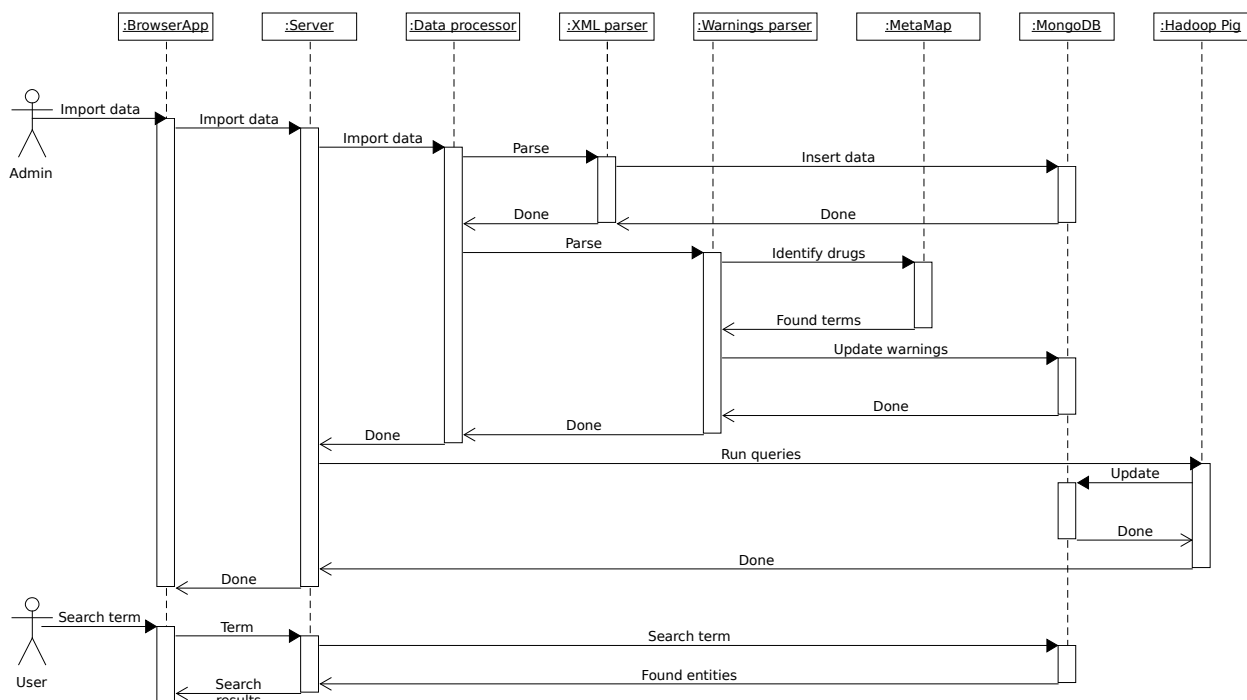
5.2. Ábra: Az alkalmazás egyszerű Use Case diagramja

A felhasználó elindítja a böngészőjét, megadja az alkalmazás elérési címét (URL-jét), amely HTTP protokollon keresztül kommunikál az alkalmazást szolgáltató webszerverrel. A webszerver első lépésben visszatéríti a szükséges HTML, Javascript, illetve CSS fájlokat, majd a kommunikáció REST [15] alapú csomópontokon keresztül fog megvalósulni JSON típusú adat formájában, amelyet a böngészőben felépített Javascript keretrendszer kezel.

Még mielőtt az egyes modulok működését megvizsgálnánk, tekintsük meg átfogó képet a rendszerről, amely egy szekvencia diagram formájában tünteti fel az összes komponenst, a közöttük fennálló adatcserét és a műveletek időrendi sorrendjét. Az egyes komponensek részletes leírása a következő fejezetekben kerül bemutatásra.

Az alkalmazásnak elsősorban két funkciót kell betölteni:

- adatok feldolgozása
- adatok megjelenítése



5.3. Ábra: Az alkalmazás szekvenciadiagramja

Mivel egy gyors válaszidejű alkalmazás a cél, ezért a Node.js platformot választottam az alkalmazás szerver oldali megvalósítására, amely lehetőséget biztosít nemcsak a böngészőből érkező kérések kiszolgálására, hanem az alkalmazás egyéb részfeladatának a megoldására is mint az adatok feldolgozása és betöltése az adatbázisba, vagy a MetaMap rendszer használata. A Node.js egy Javascript alapú platform, amely szerver oldali alkalmazások futtatását teszi lehetővé. A Chrome webes böngésző V8-as motorját használja, viszont ellentétben a böngészőben futó megoldástól, használhatja a fájlrendszert, illetve az operációs rendszer által hozzáfér egyéb erőforrásokhoz is. Jelen esetben, szükségünk lesz feldolgozni XML fájlokat, illetve kommunikálnunk kell a MetaMap eszközzel is ahhoz, hogy felismerjük a gyógyszereket és más gyógyászati kifejezéseket. Webes alkalmazásokra jellemző, hogy bemenet-kimenet korlátoltak (IO-bound), mivel a bemeneti adatok mennyisége véges, nem állandó, ezért sok esetben nem az erőforrások mennyisége határozza meg az alkalmazás teljesítményét, hanem a feldolgozás ütemezése játszik fontos szerepet ebben. A Node.js platform megfelel a követelményeinknek, mivel esemény orientált stílusban hajtja végre a kódot abban a pillanatban, amikor az esemény bekövetkezik (például a felhasználó beüt egy gyógyszernevet vagy eredményt szolgáltat a MetaMap rendszer); egy esemény ciklust követ, amely nem blokkolja az alkalmazást, amikor eredményre vár, hanem visszakérül készenléti állapotba, és más események lekezelését (callback függvények által megadott műveletek) teszi lehetővé. A Javascript környezet egy szálon hajtja végre a programkódot, sohasem fog egyidőben két különböző kódrészlet végrehajtódni, viszont abban a pillanatban, amikor kimenetre vagy bemenetre várunk, az eseményciklus (event loop) tovább pörög, és a szál (thread) nem blokkolja a programot, hanem végrehajtja az esetleges események lekezelését.

Egyre népszerűbbé kezdenek válni a böngészőben futtatott egyoldalú alkalmazások (ún. SAP vagy Single Page Applications), amelyekre az jellemző, hogy a HTML oldal betöltése után AJAX kéréseket kezdeményeznek a webserververhez, így részlegesen frissítik a böngészőben megjelenített weboldalt anélkül, hogy teljesen újratöltenék azt minden egyes lépésre, amelyet a felhasználó végez



a böngészőben (például HTML form kitöltése és elküldése vagy egy új menüpontra történő kattintás). A böngészők nyelve a Javascript programnyelv, egy olyan szkriptnyelv, amely dinamikus, funkcionális és a függvényeket elsőszámú komponensnek tekinti (first class functions). Ez utóbbi kifejezésen azt értjük, hogy a függvényeket könnyen átadhatjuk paraméterként más függvényeknek, adatstruktúrákban tárolhatjuk el, illetve visszatérítési értéknek is használhatjuk őket. A legfontosabb különbség a Javascript és más programnyelvek mint a Java vagy C++ között az, hogy a Javascriptben nincs típus ellenőrzés kódírás közben, így a lehetséges hibák csak futásidőben bukkannak fel. Ez a negatívum mára ellensúlyozódott a legújabb tesztelési eszközök megjelenésével, amelyek biztosítják a kód minőségének fenntartását, és növelik a programozók termelékenységét. Az alkalmazás jelentős része a böngészőben fog futni, ezért indokolt egy jól teljesítő, gyors keretrendszer megválasztása, amely segít a kód struktúrázásában és megírásában. Nyilvánvaló, hogy a kliensoldali rész több modult fog magában foglalni, ezért az AngularJS keretrendszert választottam, mivel nyílt forrású projekt, nagy közösségi háttérrel rendelkezik, és egy gyors fejlesztési modellt biztosít. Segít a kód struktúrázásában, támogatja a tesztelhetőséget és a moduláris programozást. Vissza fogok térni a tesztelésre (tesztelési könyvtárak, tesztek futtató környezetek, stb.), amikor részletezem az alkalmazás felépítését.

Az eddig felsorolt korszerű technológiák, eszközök ismertetésével nyilvánvalóvá válik, hogy a Javascript egyre nagyobb teret hódít a webes alkalmazások területén, nemcsak ami a böngészőt illeti, hanem ami a szerver oldali feldolgozást is jelenti.

A webes alkalmazáshoz szorosan kapcsolódik az adatbázis, mivel itt fogjuk eltárolni a gyógyszereket, alapanyagokat, gyártókat, stb. Adatbázis kezelő rendszernek a MongoDB-t választottam, mivel megfelel a követelményeknek, és jól kapcsolódik a már említett technológiákhoz (AngularJS, Node.js), mivel szintén támogatja a Javascript programnyelvet. Könnyűszerrel kommunikálhatunk az adatbázissal a Node.js alkalmazásból, mivel mindkettő ugyanazt a nyelvet beszéli. A MongoDB egy dokumentum orientált adatbázis rendszer, amely abban különbözik elsősorban a relációs adatbázis rendszerektől, hogy a tárolandó adathalmaznak nincsen egy rögzített sémája. Az adat teljességét az alkalmazásnak kell megvizsgálni és leellenőrizni, mielőtt az beszúrásra kerül, az esetleges hibákat szintén az alkalmazásnak kell orvosolni, mivel az adatbázis ezzel nem foglalkozik. Mindennek az az előnye, hogy az adatbázis rendkívül gyors, az adatok elérhetőségét replikahalmazok segítségével biztosítja, amelyek ugyanazt az adatot több, különálló csomópontokban teszi elérhetővé. A skálázhatóságot vízszintesen oldja meg, ami azt jelenti, hogy az adatokat nem egyetlen nagy csomópontban tárolja, hanem több, különálló egységre osztja szét.

Meg szeretném itt jegyezni, hogy felhasználtam az NPM (Node Package Manager) rendszert az egyes Node.js modulok kezelésére. Az NPM lehetővé teszi, hogy különböző komponenseket használjunk a Node.js alkalmazáson belül úgy, hogy terminálból megadjuk a megfelelő utasítást a telepítésre. A felhasznált komponensek listája a package.json fájlban kerül eltárolásra, amely által bármikor újra építhetjük a függő komponenseket. A függő komponensek a node\_modules könyvtárban találhatók meg. Egy tipikus package.json fájl a következő alakú:

```
"name": "DrugsDB",
"version": "1.0.0",
"description": "",
"main": "server.js",
"author": "",
"license": "ISC",
"devDependencies": {
  "connect-livereload": "^0.5.4",
  "grunt": "^1.0.0",
```

```

...
},
"dependencies": {
  "adm-zip": "^0.4.7",
  ...
}
}

```

Ez a fájl tárolja az alkalmazás összes függőségeit. A dependencies nevű rész tárolja az összes olyan függőségeket, amelyre az alkalmazásnak termelés közben szüksége van, a devDependencies rész olyan függőségeket tárol, amelyeket fejlesztés során használunk fel (például a grunt modul automatizálási folyamatokra használják). Nézzük meg, hogy mely külső komponensekre (third party tools, libraries) volt szükség a webes alkalmazáson belül:

devDependencies	
	connect-livereload
	grunt
	grunt-concurrent
	grunt-contrib-less
	grunt-contrib-watch
	grunt-express-server
	grunt-nodemon
	jasmine-core
	karma
	karma-chrome-launcher
	karma-requirejs
	karma-jasmine
	load-grunt-tasks
dependencies	
	adm-zip
	bcrypt-nodejs
	body-parser
	cookie-parser
	ejs
	express
	express-session
	mongoose
	passport

	q
	request
	socket.io
	winston
	winston-mongodb
	xmlDOM
	xpath

### 5.1. Táblázat: Az alkalmazás függőségei

Az előző táblázatban számos komponens található, amely megkönnyíti a különböző funkcionalitások implementálását (például jelszavak titkosítása a bcrypt függvénykönyvtár segítségével történik vagy a callback függvények végrehajtásának ütemezését a q modul biztosítja). Ezen komponensek használatát a megfelelő fejezetekben részletezem, ahogy haladok az alkalmazás felépítésének bemutatásával.

Még mielőtt rátérnék az egyes komponensek részletezésére, kulcsfontosságúnak tartom, hogy kihangsúlyozzam a fejlesztés során felhasznált eszközök szerepét, amelyek egy modern webes alkalmazásban segítik a programozó munkáját. [16] Ez a munka kiterjed az automatizálási eszközök nyújtotta előnyökre szoftverprogramok esetében: automata tesztek segítségével növelhetjük a kód biztonságát, csökkenthetjük a rizikófaktort újabb funkcionalitás beépítése esetén, és könnyebben megtalálhatjuk a hibákat még a fejlesztési fázis ideje alatt. Fontosnak tartottam automatizálási és tesztelési eszközök használatát, mert ezek lényegesen hozzájárulnak a kód minőségének fenntartásához, és lerövidítik a fejlesztési folyamatot. A Node.js platform egy fejlett környezet, amely támogatja a legújabb agilis folyamatok implementálásához szükséges eszközök használatát.

Az alkalmazáson belül a grunt függvénykönyvtárt választottam a különböző folyamatok automatizálásához, amely feladatok futtatását teszi lehetővé (task runner). A grunt eszközhöz szintén nagyon sok bővítmény létezik az NPM modul kezelő rendszeren belül; ezek letöltéséhez a package.json-t használjuk. Nézzük meg a beállított grunt.js fájlt, amely meghatározza a különböző automatizálási feladatokat.

```

1. module.exports = function (grunt) {
2.     require("load-grunt-tasks")(grunt);
3.
4.     grunt.initConfig({
5.         express: {
6.             options: {
7.             },
8.             web: {
9.                 options: {
10.                    script: "server.js",
11.                }
12.            }
13.        },
14.        less: {
15.            dev: {
16.                options: {
17.                    compress: true,
18.                    optimization: 2
19.                },
20.                files: {

```

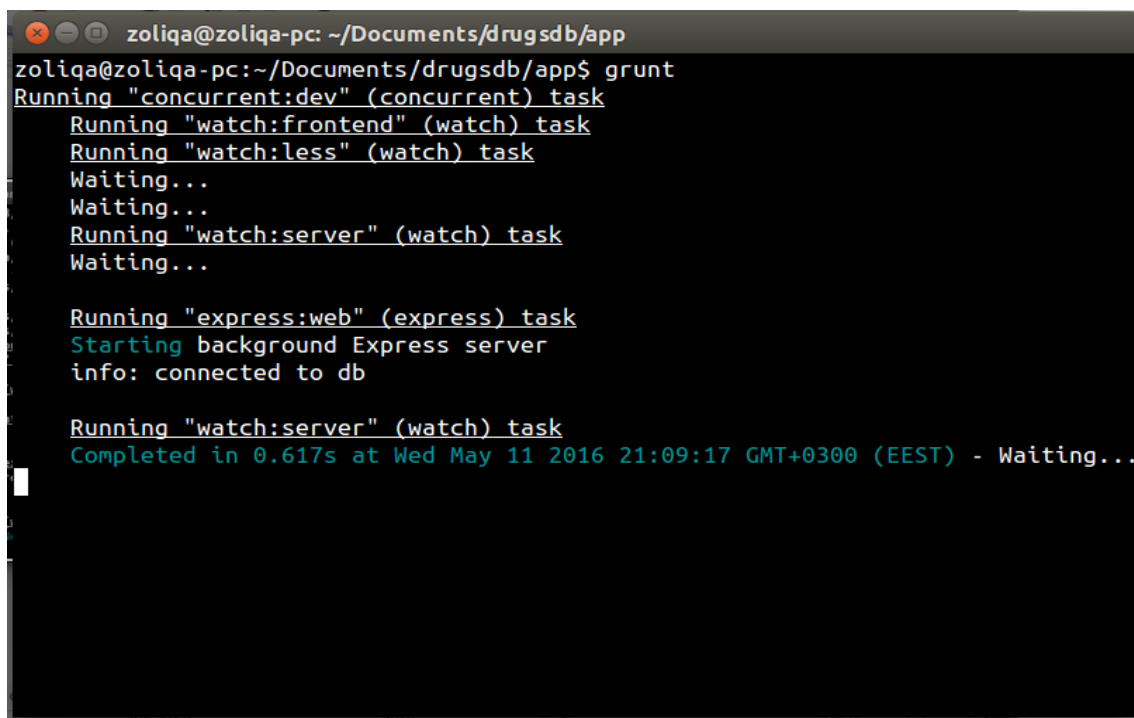
```

21.             "public/css/index.css": "public/css/less/index.less"
22.         }
23.     }
24. },
25.     watch: {
26.         frontend: {
27.             options: {
28.                 livereload: true
29.             },
30.             "public/app/**/*.js",
31.             files: [
32.                 "public/css/**/*.css",
33.                 "views/index.ejs"
34.             ]
35.         },
36.         less: {
37.             files: ["public/css/**/*.less"],
38.             tasks: ["less:dev"],
39.             options: {
40.                 nospawn: true
41.             }
42.         },
43.         server: {
44.             files: [
45.                 "data.import/**/*.js",
46.                 "db/**/*.js",
47.                 "logger/**/*.js",
48.                 "passport/**/*.js",
49.                 "routes/**/*.js",
50.                 "gruntfile.js",
51.                 "server.js"
52.             ],
53.             tasks: [
54.                 "express:web"
55.             ],
56.             options: {
57.                 nospawn: true,
58.                 atBegin: true
59.             }
60.         }
61.     },
62.     concurrent: {
63.         dev: {
64.             tasks: ["watch:frontend", "watch:less", "watch:server"],
65.             options: {
66.                 logConcurrentOutput: true
67.             }
68.         }
69.     }
70. });
71.
72.     grunt.registerInitTask("default", "concurrent:dev");
73. };

```

Az „express” komponens (5. sor) meghatározza, hogy az alkalmazás kiindulópontja a server.js modul, a „less” komponens (14. sor) meghatározza a less kiterjesztésű fájlokból létrehozott és tömörített index.css fájlt, a „watch” komponens (25. sor) pedig definiálja, hogy az előző feladatok milyen események bekövetkezése esetén fognak végrehajtódni (például a less fájlok mentése maga után vonja a végső css fájl kigenerálását vagy a szerver oldali „.js” kiterjesztésű fájlok mentése a

szerver újraindítását eredményezi). Az alkalmazás elindítása terminálból történik. Miután az aktuális könyvtárt kiválasztjuk, hogy mutasson a projekt helyére, beütjük a „grunt” kulcsszót, amely az előbb definiált fájlfigyelők elindítását eredményezi. A következő eredményt kapjuk:



```
zoliqa@zoliqa-pc: ~/Documents/drugsdb/app
zoliqa@zoliqa-pc:~/Documents/drugsdb/app$ grunt
Running "concurrent:dev" (concurrent) task
  Running "watch:frontend" (watch) task
  Running "watch:less" (watch) task
  Waiting...
  Waiting...
  Running "watch:server" (watch) task
  Waiting...

  Running "express:web" (express) task
  Starting background Express server
  info: connected to db

  Running "watch:server" (watch) task
  Completed in 0.617s at Wed May 11 2016 21:09:17 GMT+0300 (EEST) - Waiting...
```

5.4. Ábra: Az alkalmazás futtatása terminálból

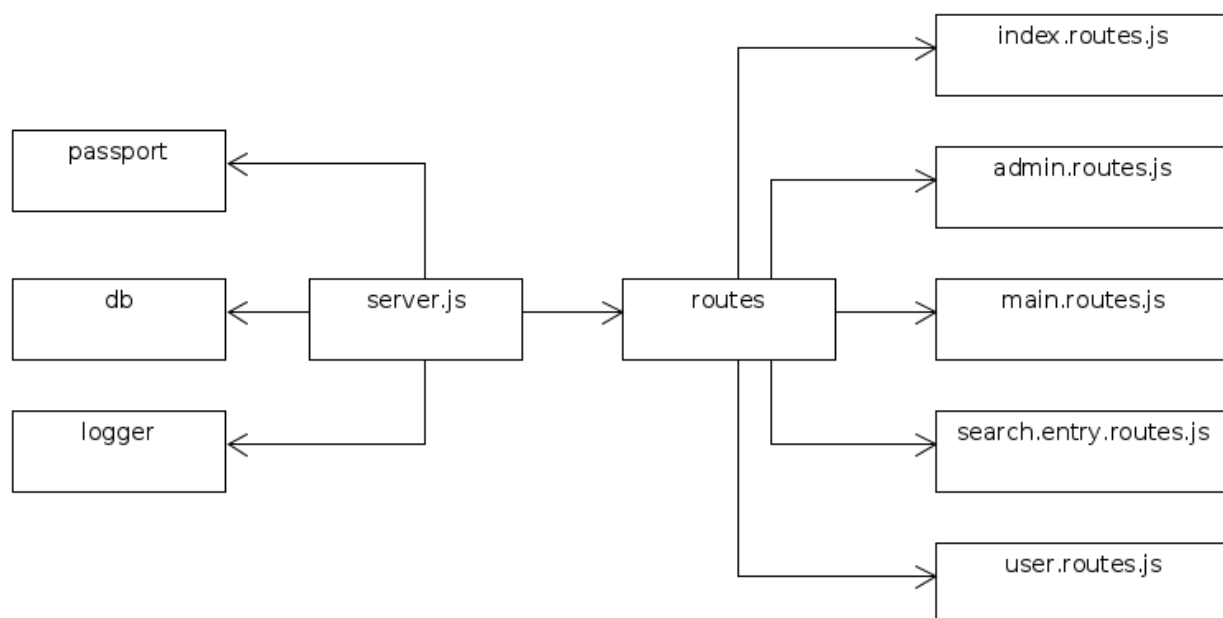
Az alkalmazást úgy építettem fel, hogy lehetővé tegyem a különböző modulok könnyű tesztelését, legyen az szerver oldali kód vagy kliensoldali kód. Mivel a Javascript egy szkriptnyelv, amely nem követeli meg a típusok definiálását, ezért elengedhetetlen az egység tesztek (unit test) illetve integrációs tesztek (integration test) használata. A unit tesztek kisebb egységeket tesztelését teszik lehetővé, és gyors visszajelzést adnak a tesztek eredményéről. A projektben a Karma.js könyvtárt használtam a unit tesztek futtatására [17], ennek beüzemeltetése a karma.conf.js fájl segítségével történik, amelyben megadhatók különböző paraméterek a tesztek futtatásához (melyik böngészőben történjen a tesztelés – a mi esetünkben ez a Chrome volt, de megadható Internet Explorer vagy Firefox böngésző is egyaránt vagy akár a Phantom.js könyvtárral böngésző nélkül is futtathatjuk tesztjeinket). A konfigurációs fájlban megadható, hogy a test runner automatikusan futtassa le az egyes teszteket abban a pillanatban, amikor valamely kódfájl módosul („autowatch: true” kapcsoló segítségével). Teszt esetek bemutatására a 6.4. fejezetben („Tesztelés és eredmények: Forráskód minőségének tesztelése” fejezet keretén belül) kerül sor.

Integrációs tesztek segítségével összetett lépéssorozatokot tesztelhetünk (például Ibuprofen gyógyszer keresése és eredmények kilistázása, majd a kimeneti adatok összehasonlítása egy meghatározott lista elemeivel). Integrációs tesztek futtatását a Nightwatch.js függvénykönyvtárral valósítottam meg, amely egy egyszerű interfészen keresztül biztosítja az alkalmazások tesztelését CSS lekérdezésekhez hasonló szintaxisal.

A következőkben részletezni fogom a bemutatott road map diagramon szereplő egyes komponenseket, kitérve az implementációs sajátosságokra.

## 5.3. A szerver komponens

Az előző fejezetben említettem, hogy Node.js-t használtam a szerver oldali komponens implementálására. A technológiát röviden bemutattam az előző részben, most kitérek a implementálási sajátosságokra. Node.js-ben modulokat hozunk létre, amelyek különböző funkcionalitásokat valósítanak meg, mint például a szerver elindítása, különféle REST csomópontok meghatározása, adatbázis kapcsolat létrehozása, stb. A böngészőben futó Javascript motorral ellentétben, ahol nem létezik natív módon egy modul kezelő rendszer (a kódfájlok egy globális névtérben hoznak létre kódstruktúrákat (objektumokat), ahol bármely kódfájlból elérhetünk bármely más kódfájlban definiált struktúrákat vagy objektumokat), a Node.js platform támogatja és megköveteli a modulokban történő fejlesztést, amely egy robusztus, átlátható kód elrendezést tesz lehetővé. A `require()` függvényhívás segítségével hivatkozhatunk más függvénykönyvtárakra vagy az általunk definiált modulokra a projektkönyvtárban belül. Ahhoz, hogy egy átfogó képet kapjunk a szerver oldali rész felépítéséről, tekintsük meg következő komponensdiagramot:



5.5. Ábra: Szerver komponensek Road map diagramja

### 5.3.1. A server.js modul

A `server.js` modul tartalmazza a webes alkalmazás kiindulópontját, ennek a futtatása indítja el az élő HTTP szerveret, amely a böngészőből érkező kéréseket kiszolgálja. Itt vannak megadva a projekten belüli (`passport`, `db`, `logger`, stb.) és a külső függőségek (`third party library`). Jelen esetben a Node.js platform v5.11.0 verzióját használtam, amely a Chrome V8-as motorja által több új programozási funkciót tartalmaz az ECMAScript 2015-ös specifikációjából [18], mint például a `const`, `let` kulcsszavak vagy `lambda` kifejezések használata. Ezek a szintaktikai újdonságok (`syntactic sugar`) hozzájárulnak a Javascript kód könnyebb írásához és karbantartásához. Itt megjegyzem, hogy ezeket az újdonságokat csak a szerver oldali kódrészekben használtam ki, mivel nem minden böngésző támogatja őket, és tervem az volt, hogy minél szélesebb körű felhasználó közösséget célozzak meg az alkalmazással.

A server.js modulban felhasználtam az Express.js webes keretrendszert (framework), amely megkönnyíti a webes alkalmazások írását Node.js programokban. Az Express.js modul segítségével oldottam meg a felhasználó hitelesítését, amely a passport komponensben (könyvtárban) található két modul által történik. Az Express.js modul lehetővé teszi, hogy middleware komponenseket használjunk fel egy HTTP kérés kiszolgálása során, amelyek különféle átalakításokat és feldolgozásokat végeznek a request objektumon (JSON csomagok elemzése, HTML form értékek dekódolása, cookie értékek elemzése, stb.), mielőtt a végső Express.js callback függvényekhez kerülnek.

Ez a modul egy felsőszintű komponens, amely felhasználja többi komponenst: passport, routes, connection, logger. Itt vannak definiálva az egyes middleware komponensek is:

```
app.use(session({
  secret: "secret",
  resave: true,
  saveUninitialized: true
}));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));
app.use(passport.initialize());
app.use(cookieParser());
app.use(passport.session());
```

### 5.3.2. A routes komponens

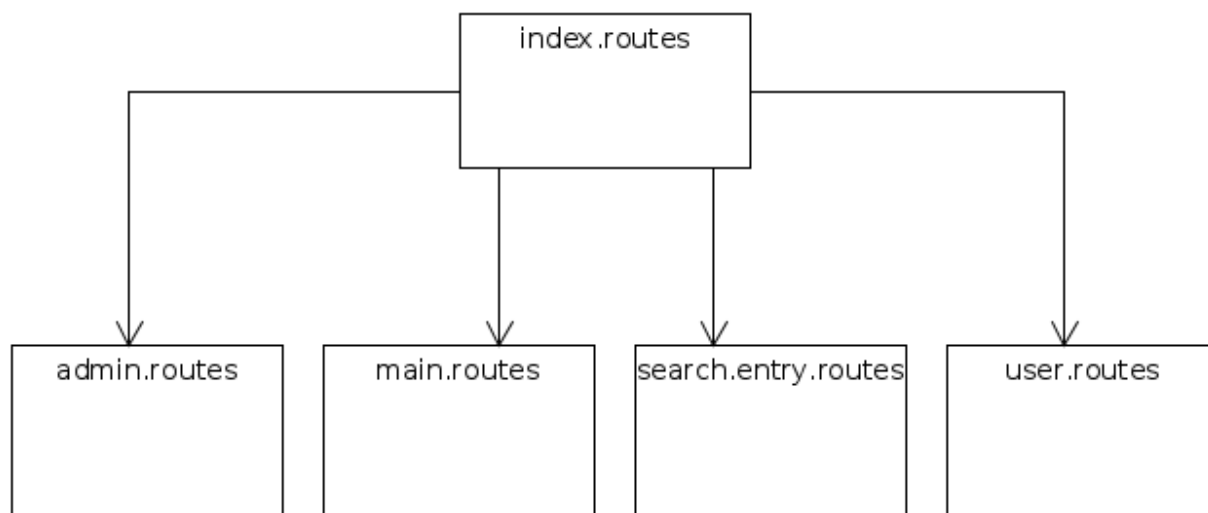
Ez a komponens magában foglalja az egyes végpontokat (endpoints), amelyeket a Node.js szerver biztosít: különböző formájú URL-ek, amelyeket a kliens oldali (frontend) Javascript kód felhasznál. Ezeket az URL-eket több csoportba foglaltam bele annak függvényében, hogy milyen erőforrás elérést céloznak meg. A routes könyvtárban belül 5 modul kapott helyet:

admin.routes.js:	Admin felhasználó által elért erőforrások: POST /admin/import (gyógyszerek feldolgozása és beszúrása az adatbázisba, illetve Pig query alapú feldolgozás elindítása)
index.routes.js	A route komponensek összegyűjtése (admin, main, search.entry és user), és az URL első részének meghatározása (lásd következő diagramon) GET / (kezdő HTML fájl lekérése)
main.routes.js	Alapműveletek elvégzése: POST /main/searchall (kritérium nélküli keresés) POST /main/searchdrug (keresés gyógyszernév szerint) POST /main/searchproducer (keresés gyártónév szerint) POST /main/searchsubstance (keresés alapanyag szerint) POST /main/searchmedterm (gyógyászati kifejezés keresése a MedlinePlus szolgáltatásban)

search.entry.routes.js	POST /searchentry (új keresés bejegyzés beszúrása az adatbázisba) GET /searchentry (összes keresés bejegyzés lekérdezése) DELETE /searchentry/:id (id-val megadott keresés bejegyzés törlése)
user.routes.js	POST /user/login (felhasználó bejelentkezése) GET /user/logout (felhasználó kijelentkezése) GET /user/:username (adott nevű felhasználó lekérdezése) GET /user (bejelentkezett felhasználó lekérdezése) POST /user (új felhasználó regisztrálása a rendszerbe) PUT /user (bejelentkezett felhasználó adatainak frissítése) DELETE /user (bejelentkezett felhasználó törlése a rendszerből)

5.2. Táblázat: Az alkalmazás által támogatott URL azonosítók listája

A következő diagram szemlélteti a routes komponenst:



5.6. Ábra: Routes modul komponens diagramja

Az előző modulok REST csomópontjai által definiált callback függvények rendre az adatbázissal lépnek kapcsolatba a megfelelő erőforrások lekérdezése illetve módosítása végett. Az adatbázis elérését és kezelését az 5.3.5 fejezetben ismertetem.

Amint a előző diagramon is észrevehető, a fájlok neveiben kisbetűt és pontot használtam; ezáltal könnyen behatárolható a keresett kódfájl a név alapján egyszerű billentyűkombinációkkal. Törekedtem arra, hogy szemléletes nevet adjak a fájloknak a könnyű lokalizálás érdekében, ami fontos szempont egy relatív nagyobb méretű projekt esetében. A kliens oldali komponens felépítésében is megőriztem ezt a fájlstruktúrát, amely egyszerűsíti a keresett kód lokalizálását.



### 5.3.3. A passport komponens

Ez egy middleware komponens az Express.js keretrendszer számára, amely biztosítja a bejelentkezett felhasználó nyomon követését. Ezt egy böngészőben tárolt HTTP süti (cookie) segítségével valósítja meg. A passport könyvtárban két modul található, amelyek arra szolgálnak, hogy a bejelentkezett felhasználót (user) identitását érvényesítsék. Mindez úgy történik, hogy minden egyes kérés (web request) esetén, amely egy levédett erőforráshoz próbál hozzáférni, megvizsgáljuk a felhasználó kilétét, vagyis megbizonyosodunk, hogy a felhasználó előzőleg valóban bejelentkezett az alkalmazásba vagy sem. Amennyiben a felhasználónak nincsen joga elérni az erőforrást, 401-es hibát térítünk vissza „Unauthorized” üzenettel.

Ez a komponens két modulból áll:

- passport.init.js. Két függvényt tartalmaz: a `serializeUser()` meghatározza, hogy mely adat kerül eltárolásra a sütin belül (a mi esetünkben a user sor id mezőjét használjuk), illetve a `deserializeUser()` definiálja, hogy hogyan építjük újra a user sort a sütiből kinyert id érték alapján. Természetesen az adatbázist fogjuk használni ennek a megvalósítására.

- passport.utilities.js. Két függvényt tartalmaz: az `isAuthenticated()` eldönti, hogy az adott felhasználó bejelentkezett-e már a rendszerbe vagy sem, az `isAdminUser()` visszatéríti, hogy az adott felhasználó rendelkezik-e rendszergazda (admin) jogokkal vagy sem.

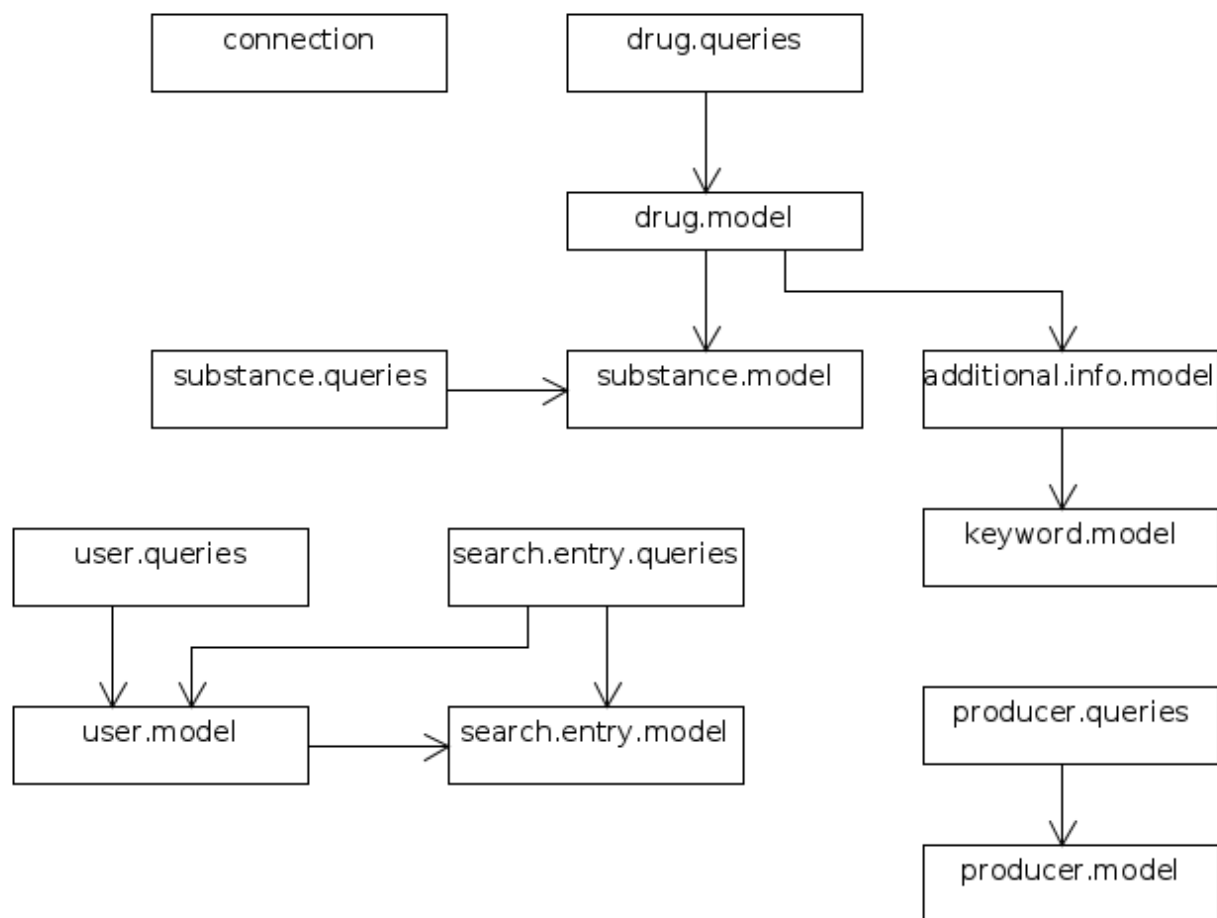
### 5.3.4. A logger komponens

A logger könyvtárban található logger.js modul log bejegyzések tárolását teszi lehetővé, amely felhasználja a winston, illetve winston-mongodb npm-es modulokat. A winston egy olyan modul, amely biztosítja különböző prioritású log bejegyzések tárolását az alkalmazás robusztus működése végett. Ez által utólag megtekinthetjük az egyes műveletek eredményét, illetve hiba esetén részleteket tudhatunk meg a program viselkedéséről. A winston-mongodb támogatja a bejegyzések MongoDB adatbázisban történő eltárolását. Kétféle típusú üzenetet tárolhatunk el a logger által, amelyeket az `info()` illetve az `error()` függvények valósítanak meg. Az `info()` függvényt általános események rögzítésére használjuk, mint például az adatbázisba került adatok beszúrásának eredménye, időponja. Az `error()` függvényt akkor hívjuk meg, ha valamilyen váratlan hiba lépett fel egy feladat végrehajtása során (például nem sikerült egy XML dokumentum elemzése vagy a megfelelő figyelmeztetést tároló elemek tartalmának a feldolgozása. Ebben az esetben eltároljuk a fájl nevét és a hiba okát, felüntetve a stacktrace loggokat is a visszafejtés céljából).

### 5.3.5. A db komponens

A db könyvtárban elhelyezett modulok rögzítik az adatbázisban tárolt modellek meghatározását. Mivel a MongoDB adatbázis nem követeli meg a beszúrandó adatok teljességét (data integrity), ezért előnyös az alkalmazásban megbizonyosodni, kiértékelni a beszúrandó adatok helyességét. Erre a célra a Mongoose.js modult használtam, amely egy objektum modellező rendszer Node.js alkalmazások számára. A Mongoose.js modul lehetővé teszi, hogy modelleket definiáljunk az alkalmazás számára, amelyeket az adatbázisban kívánunk eltárolni. Ugyanakkor biztosítja az adatok teljességének megvizsgálását (data validation). Úgy építettem fel ezt a komponenst, hogy minden különálló entitásnak (mint például felhasználó, gyógyszer vagy gyártó) egy új modult határoztam meg egy fájlban belül az `[entitásnév].model.js` sablon szerint. Az entitáshoz kapcsolódó lekérdezéseket az `[entitásnév].queries.js` sablon szerint megnevezett fájlban tároltam el, így

könnyen lehet lokalizálni a keresett kódészleteket. A Mongoose.js modul támogatja a modellek definícióin keresztül az adatbázis kezelő műveletek (lekérdezések, beszúrások, törlések) végrehajtását.



5.7. Ábra: A db komponens osztálydiagramja

Az adatbáziskapcsolat kiépítése a connection.js fájlban belül történik, ahol meghatároztam a futó adatbázis példány URL-jét és beállítottam a q modul használatát promise típusú objektumok kezelésére. A q modulnak fontos szerepe van az alkalmazásban, mivel a Javascript nyelv aszinkron tulajdonsága miatt callback függvényeket határoztunk meg különböző események lekezelésére (eseményfolyamat meghatározására). Ezeknek az egymásba ágyazása egyrészt a kódot nehezen követhetővé, másrészt pedig az eseményfolyamatok végrehajtását függővé teszi külső könyvtárak (third party library) által, ami a program futásának, viselkedésének kiszámíthatatlanná válásához vezethet. [19] Ezeket a hátrányokat kívántam kiküszöbölni a q modul segítségével, amely elérhető egyaránt a szerver oldali Node.js rendszerben mint a kliens oldali Angular.js rendszerben.

A drug modell definiálása a drug.model.js fájlban történik. A diagramon látható, hogy ez a modell hivatkozik a substance, illetve az additionalInfo modellekre, ezért ezekenek a sémáit rendre be kell tölteni. A Mongoose.js modul megköveteli, hogy definiáljunk egy séma, illetve egy modell objektumot minden adatbázis entitás számára. A sémában meghatározzuk a mezők típusait, esetleg indexet is definiálhatunk rajta. A drug sémája a következő alakú:

1. const mongoose = require("mongoose"),
2. Schema = mongoose.Schema,

```

3.     substanceModel      = require("./substance.model"),
4.     additionalInfoModel = require("./additional.info.model");
5.
6.     const DrugSchema = new Schema({
7.         name: String,
8.         producerId: String,
9.         producerName: String,
10.        ingredients: [substanceModel.SubstanceSchema],
11.        additionalInfos: [additionalInfoModel.AdditionalInfoSchema],
12.        interactionDrugs: [String]
13.    });

```

Látható, hogy az ingredients mező meghatározásakor (10. sor) felhasználtam a substance.model.js fájlban definiált sémát. Ez azt jelenti, hogy az adatbázisban a drugs halmaz minden eleme tartalmazni fog egy beágyazott listát az összetevők számára. A 12. sorban feltüntetett interactionDrugs mező az illető gyógyszerrel kölcsönhatásban álló gyógyszerneveket fogja tartalmazni ugyancsak egy lista formájában, azonban ennek típusa karakterlánc típusú lesz (String) az előbb említett objektum struktúra helyett.

A drugs halmaz lekérdezéseit a drug.queries.js fájl tartalmazza. Itt egyetlen lekérdezés található, amely a gyógyszernév szerinti keresést valósítja meg:

```

1. function search(term, next) {
2.     Drug.find({
3.         name: {
4.             $regex: new RegExp(".*" + term + ".*", "i")
5.         }
6.     }, (err, drugs) => {
7.         if (err)
8.             return next(err);
9.
10.        return next(null, drugs);
11.    });
12. }

```

A search() függvény két paramétert fogad: a term a keresett kifejezést tárolja, míg a next változó egy függvényobjektum, amelyet arra használunk, hogy értesítsük a hívó objektumot az eredményről (10. sor) vagy, hiba esetén, jelezzük azt (8. sor). A 4. sorban definiált reguláris kifejezéssel határoztuk meg a term kifejezés összehasonlítását a gyógyszer nevével.

A search.entry.queries.js fájlban három függvényt határoztam meg: findAll, create illetve remove, amelyek a keresési bejegyzések listájának lekérését, létrehozását illetve törlését valósítják meg. Itt meg kell jegyeznem, hogy a adatbázisban nem használtam egy új halmazt ezen adatok tárolására, hanem ezeket a users halmazban a user sor megfelelő mezőjében tároltam el. Így nincsen szükség két halmaz adatainak összevonására ahhoz, hogy az összetartozó sorokat előállítsuk. Ugyanez a stratégia áll a háttérben a producers illetve a substances halmazok felépítésében is, ahol lényegében véve ugyanaz az adathalmaz kerül eltárolásra, amely a drugs halmazban már szerepel. Ezekben az esetekben csupán más kritériumok szerint történik az adatok csoportosítása: a producers esetében minden egyes gyártó sor tartalmazza az általa előállított gyógyszerek listáját, míg a substances halmaz az alapanyagok szerint csoportosítja az egyes gyógyszerek listáját.

### 5.3.6. Egyéb szerver oldali komponensek

A szerver oldali rész egyéb komponensei:

- data.import
- public
- views

A data.import könyvtár tartalmazza a DailyMed oldaláról letöltött XML dokumentumok feldolgozását. Ide tartozik a tömörített fájlok kicsomagolása, elemzése (parsing) előre meghatározott szabályok szerint, a kinyert részek MetaMap által történő feldolgozása, stb. Ennek a modulnak a részletes leírása az 5.4.1 alfejezetben található meg.

A public könyvtár tartalmazza az összes olyan Javascript kódfájlt, amelyhez a böngésző hozzáfér. Itt található a teljes kódbázis, amely a böngészőben fut, beleértve az egyes függőségeket. Az Express.js által tesszük nyilvánossá ennek a könyvtárnak a tartalmát (a \_\_dirname kulcsszó egy beépített globális változó, amely a futó Node.js alkalmazás aktuális könyvtárának nevét tárolja) a server.js fájlban:

```
app.use("/public", express.static(__dirname + "/public"));
```

A server.js fájlban található a kezdő HTML fájl útvonalának meghatározása egyben az oldal meghajtó eszköz (view engine) kiválasztásával:

```
app.set("views", "./views");  
app.set("view engine", "ejs");
```

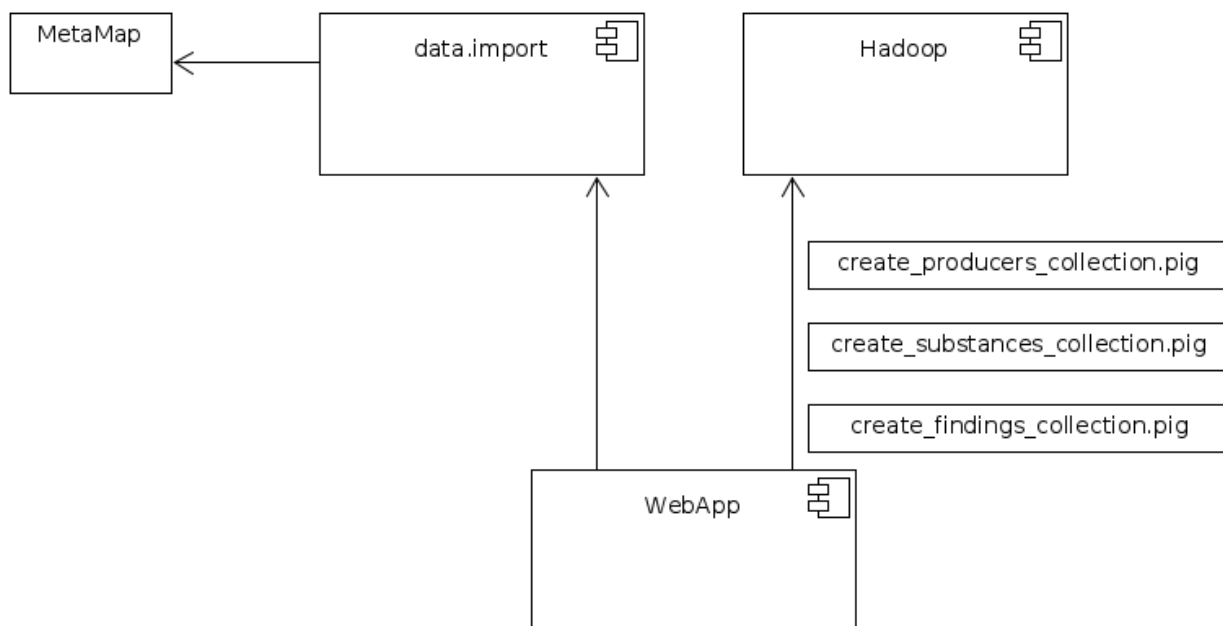
A views könyvtárban található az index.ejs fájl, amely az alkalmazás kiindulópontja abban az értelemben, hogy ez tárolja az elsődleges HTML fájlt, amelyet a Node.js szerver szolgáltat a böngésző számára. Az előző kódrészlet meghatározza, hogy a HTML oldalak (view page) a views könyvtárban találhatók, és megjelenítő motor az ejs lesz. Az ejs egy népszerű megjelenítő motor Node.js alkalmazások számára, amely lehetővé teszi HTML oldalak szerver oldali kigenerálását. A routes komponens URL végpontjainak részletezésekor feltüntettem, hogy a GET / kérés hatására a szerver komponens visszatéríti az alap HTML fájlt:

```
router.get("/", function (req, res) {  
    res.render("index", { title: "DrugsDB", NODE_ENV: process.env.NODE_ENV });  
});
```

Az index.ejs fájl felépítését és a betöltött fájlok meghatározását a 5.5. alfejezetben ismertetem a kliens oldali komponens bemutatása során.

Az app könyvtárban találhatók még a következő fájlok: .bowerrc, bower.json, gruntfile.js, package.json és server.js. A bower.json olyan függőségeket tárol, amelyekre a kliens oldali kódnak van szüksége (pl. jQuery, AngularJS, stb.). A Bower szintén egy Node.js modul, és hasonló az NPM rendszerhez, összesített helyen tárolja a különböző Javascript könyvtárakat.

## 5.4. Az adatelemző komponens



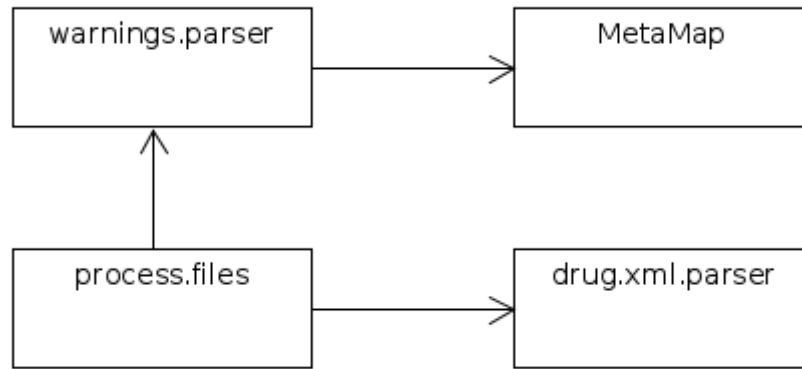
5.8. Ábra: Az adatelemző komponens függőségei

Ezen komponens feladata az adatok elmezése, kibányászása, feldolgozása és eltárolása az alkalmazás számára. A komponens szintén a szerver oldalon kerül felhasználásra a server modul által, viszont egy jól elkülönített egységként működik, és nem kapcsolódik szorosan a webes alkalmazáshoz. A komponenst alkotó kódfájlok és Pig szkriptek a data.import könyvtárban találhatók.

A data.import komponens szétválasztása a webes alkalmazástól azért is indokolt, mivel ebben történik az adatok feldolgozása, amely egy időigényes folyamat. Ha ugyanabban a processzben hajtánánk végre ezen műveleteket, mint amelyekben a szerveret futtatjuk, leblokkolnánk a szálát, amely a HTTP kérések kiszolgálását hajtja végre. Ezért az adatfelgozó modult egy új processz létrehozásával indítottam el a webes alkalmazáson belül, és egy callback függvény segítségével értesültem a feladat eredményéről. Ezen feladat sikeres végrehajtása után, végrehajtottam a Hadoop rendszer által az ábrán feltüntetett három Pig szkript fájlt (create\_producers\_collection.pig, create\_substances\_collection.pig és create\_findings\_collection.pig). Node.js alkalmazáson belül nagyon könnyen létrehozhatunk egy új gyerek Node.js processzt a fork függvény segítségével, amely egy kommunikációs csatornát biztosít a két processz között az adatok küldésére illetve fogadására. A spawn függvény segítségével egy tetszőleges processzt indíthatunk el az operációs rendszeren belül. Erre szükség volt a Pig szkriptek futtatása esetén, illetve a MetaMap eszköz elindításánál.

### 5.4.1. A data.import komponens

Ez a komponens végzi el az DailyMed portálról letöltött XML dokumentumok feldolgozását és adatbázisban való eltárolását.



5.9. Ábra: A data.import komponens modul diagramja

A process.files modul paraméterül fogadja a könyvtár nevét, amely tartalmazza az XML fájlok tömörített halmazát, majd rendre meghívja a drug.xml.parser.js fájlban tárolt modul egyetlen függvényét. A parseXml() függvény elemzi a paraméterül fogadott XML szöveget: megkeresi a gyógyszer nevét, gyártóját, összetevőit, és beszúrja az adatbázisba a figyelmeztető szövegrészletekkel együtt. Az XML dokumentumban történő keresést az xmldom, illetve az xpath modulokkal valósítottam meg. A parseXml() függvényben felhasznált XPath kifejezések a következők voltak:

//manufacturedProduct/name	Gyógyszerek neve
document/author/assignedEntity/representedOrganization/id/@extension	Gyártó azonosítója
document/author/assignedEntity/representedOrganization/name/text()	Gyártó neve
//ingredientSubstance	Alapanyagokat tároló elemek
./code/@code	Alapanyag kódja
./name	Alapanyag neve
//section[code[contains('34071-1 50570-1 50569-3 50568-5 34073-7', @code)]]	Figyelmeztetéseket tároló elemek
./code/@code	Figyelmeztetési elem kódja
./code/@displayName	Figyelmeztetési elem neve
./title	Figyelmeztetési elem címe
./text	Figyelmeztetési elem szövege

5.3. Táblázat: DailyMed XML dokumentum elemzésének XPath kifejezései

Az egyszerűség kedvéért, a feltüntetett XPath kifejezések nem tartalmazzák a forráskódban használt névtérváltozókat. A ponttal kezdődő lekérdezések relatívak az előző sorban használt abszolút útvonalak lekérdezéseihez viszonyítva. A figyelmeztetéseket tároló elemeket a következő kód kérdezi le:

```
let warningSectionElements = select("//x:section[x:code[contains('34071-1 50570-1 50569-3 50568-5 34073-7',
```

@code)]]", doc);

Ebben az utasításban olyan XML elemeket kerestem, amelyek tartalmazzák a felsorolt kombinációk valamelyikét. Ezután végighaladtam a talált elemeken, és létrehoztam az egyes találatoknak egy AdditionalInfo objektumot, amely eltárolja a kódot, nevet, címet és szöveget.

```
<section ID="ID_230e4d62-cb0c-411f-8a8b-c99112ad756c">
  <id root="2070a386-7825-46ef-90bd-e4369a381724"/>
  <code code="34071-1" codeSystem="2.16.840.1.113883.6.1" displayName="WARNINGS SECTION"/>
  <title>WARNINGS</title>
  <text>
    <paragraph>
      <content styleCode="bold">If patient has been given ipecac syrup, DO NOT give
      ACTIDOSE<sup>®</sup> WITH SORBITOL or ACTIDOSE<sup>®</sup>-AQUA until after last vomiting episode.
      Do not use in persons who are not fully conscious. Do not use this product unless directed by a health professional, if
      turpentine, corrosives, such as </content>
      <content styleCode="bold">alkalies</content>
      <content styleCode="bold"> (lye) and strong acids, or petroleum distillates, such as kerosene,
      gasoline, paint thinner, cleaning fluid or furniture polish, have been ingested. This product is not recommended for use
      in children weighing less than 32 kg, or during multiple dose activated charcoal therapy since excessive catharsis and
      significant fluid and electrolyte abnormalities may occur.</content>
    </paragraph>
  </text>
  ...
</section>
```

A fenti XML részletben, a „section” elem bekerül a találatok közé, mivel a „code” attribútuma tartalmazza a „34071-1” kifejezést.

Miután az összes fájl elemzése befejeződött, a process.files.js modulban a promise objektumok segítségével ütemeztem a gyógyszerek végigjárását, és megvizsgáltam a figyelmeztető struktúrálatlan szövegrészleteket a MetaMap eszköz segítségével. Itt fontos megjegyezni, hogy egyidőben csak egyetlen processzt ütemeztem, amely a MetaMap eszközt használja, mivel párhuzamos futtatás esetén hibák léptek fel, ami azt igazolja, hogy nem támogatott a párhuzamos végrehajtás a MetaMap eszköz által.

A gyógyszernevek, amelyek a kölcsönhatásokban vesznek részt, illetve a betegségek és tünetek felismerését a warnings.parser.js modul valósítja meg, amely a parseWarnings() nevű nyilvános függvényt definiálja. Ez a függvény egy újabb gyerek processzt indít el, amely a MetaMap végrehajtható állományt futtatja le, bemenetként szolgáltatva a struktúrálatlan szöveget.

A MetaMap eszköz telepítése után, el kell indítanunk két szerver szolgáltatást, amely az elemzést és a kifejezések megkülönböztetését végzi (skrmedpostctl és wdsrverctl). Ezután terminálból futtathatjuk a MetaMap eszközt. A parseWarnings() függvényen belül, megadtam a MetaMap elérési útvonalát és a felhasznált kapcsolókat. Egy tipikus parancs a futtatásra:

```
./echo "lung cancer" | ./metamap14 -J sosy,dsyn,orch,phsu,inpo -g --XMLf
```

Ebben az esetben az echo parancs segítségével átirányítottam a „lung cancer” kifejezést a MetaMap bemenetére. A „-J” kapcsoló segítségével meghatároztam a szemantikus típusokat, amelyre keresni kívántam, majd végül a „--XMLf” kapcsoló által megadtam, hogy XML formátumban vártam az eredményeket. A „-g” kapcsoló lehetővé teszi, hogy az eszköz Metathesaurus találatokat adjon vissza részekkel.

sosy	Sign or Symptom
dsyn	Disease or Syndrome
orch	Organic Chemical
phsu	Pharmacologic Substance
inpo	Injury or Poisoning

#### 5.4. Táblázat: Az alkalmazás által használt szemantikus típusok

A MetaMap által visszatérített XML dokumentumban ismét elemzést végeztem az említett Node.js eszközökkel. Több mondatból álló szövegrészlet esetén a MetaMap különálló mondatokra bontotta szét azt, és minden egyes mondatot egy „Utterance” elembe tárolt el, majd az egyes összetartozó kifejezéseket „Phrases” elemekben szolgáltatta vissza. Nézzünk egy tipikus példát:

```
<Utterances Count="1">
  <Utterance>
    <PMID>00000000</PMID>
    <UttSection>tx</UttSection>
    <UttNum>1</UttNum>
    <UttText>Coadministration of Plavix and NSAIDs increases the risk of gastrointestinal bleeding.</UttText>
    <UttStartPos>0</UttStartPos>
    <UttLength>86</UttLength>
    <Phrases Count="5">
      <Phrase>
        <PhraseText>Coadministration of Plavix</PhraseText>
      ...
    <CandidateMatched>Plavix</CandidateMatched>
      <CandidatePreferred>Plavix</CandidatePreferred>
      <MatchedWords Count="1">
        <MatchedWord>plavix</MatchedWord>
      </MatchedWords>
      <SemTypes Count="2">
        <SemType>orch</SemType>
        <SemType>phsu</SemType>
      </SemTypes>
    </Phrases>
  </Utterance>
</Utterances>
```

Ebben az esetben a felismert mondat az „UttText” elembe jelenik meg, majd alatta a „PhraseText” elem tárolja az egyes mondatrészeket. A számomra fontos információt a „CandidateMatched” elem jelentette, amely ez esetben tartalmazza a gyógyszer nevét (Plavix), majd alatta megjelenik a fogalom szemantikus típusa is („orch” és „phsu”). A felismert betegségeket és tüneteket eltároltam a megfelelő gyógyszer sor additionalInfos listájának a mezőjében, a kölcsönhatásban álló gyógyszerneveket akkor regisztráltam egy listában, ha az egy olyan mondatban jelenik meg, amely valamilyen betegségnevet vagy tünetet is tartalmaz (ha a felismert fogalom a következő szemantikus típusok valamelyikét tartalmazza: sosy, dsyn, inpo), vagy betegség és tünet hiányában megvizsgáltam a mondaton belüli szókombinációkat reguláris kifejezések segítségével. (Például „Coadministration of Plavix and NSAIDs increases the risk of



gastrointestinal bleeding.” esetében a MetaMap nem talált betegséget vagy tünetet a mondaton belül, viszont indokolt volt a két gyógyszer típus felismerése: Plavix és NSAID).

## 5.4.2. Pig szkriptek használata

Két Pig szkriptet használtam fel arra, hogy csoportosítsam a gyógyszer sorokat a MongoDB adatbázisban gyors lekérdezések végett. A szkript fájlok a pig.queries könyvtárban találhatók. A create\_producers\_collection.pig feladata az, hogy beolvassa a gyógyszer sorokat a drugsdb.drugs halmazból és csoportosítsa az összes gyógyszert gyártó szerint (pontosabban gyártó azonosító és név szerint). Ez által, a webes interfészről könnyen kereshetünk gyártónévre, az adatbázisban a drugsdb.producers táblát használhatjuk fel a lekérdezésre. A producer.model.js fájlban található meg a gyártó modell kódja:

```
const ProducerSchema = new Schema({
  producerId: String,
  producerName: String,
  drugs: [{
    name: String,
    ingredients: Array
  }]
}, {
  collection: "producers"
});
```

A create\_substances\_collection.pig szkript hasonló az előzőhöz, ebben az esetben viszont alapanyag szerint csoportosítottam a gyógyszereket, mivel az is nagyon gyakori, amikor azt szeretnénk megtudni, hogy mely gyógyszerek tartalmazzák az adott összetevőt (például Ibuprofen).

## 5.5. A kliens komponens

Ebben a részben ismertetem a böngészőben futtatott alkalmazás felépítését és működését. Amint azt már említettem, a böngésző számára fenntartott fájlok halmaza a public könyvtárban belül található. Innen bármely típusú fájl elérhető a böngésző számára. Az alkalmazás úgy indul el, hogy a felhasználó beüti a böngésző cím mezőjébe a http://localhost:4000/ URL-t, amely egy HTTP kérést küld a Node.js alkalmazás szerver oldali komponensének. Válaszként a szerver visszatéríti az index.ejs sablon (template) alapján kigenerált HTML fájlt:

```
1. <!DOCTYPE html>
2.
3. <head>
4. <html xmlns="http://www.w3.org/1999/xhtml">
5.   <meta charset="utf-8" />
6.   <title><%= title %></title>
7.
8.   <link href="/public/lib/components-font-awesome/css/font-awesome.css" rel="stylesheet" />
9.   <link href="/public/lib/jquery-ui/themes/base/all.css" rel="stylesheet" />
10.  <link href="/public/lib/bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
11.  <link href="/public/lib/bootstrap-social/bootstrap-social.css" rel="stylesheet" />
```

```

12. <link href="/public/lib/offline/themes/offline-theme-dark-indicator.css" rel="stylesheet" />
13. <link href="/public/css/index.css" rel="stylesheet" />
14. </head>
15. <body>
16. <ng-include src="/public/app/main/page.header.html"></ng-include>
17.
18. <div class="container main-content">
19.   <div ng-view></div>
20. </div>
21.
22. <script src="/public/lib/requirejs/require.js" data-main="/public/app/require.main.js"></script>
23.
24. <% if (NODE_ENV === "development") { %>
25.   <script src="//localhost:35729/livereload.js"></script>
26. <% } %>
27. </body>
28. </html>

```

Ebben a HTML fájlban határoztam meg a kliens oldali alkalmazásban felhasznált CSS fájlokat (8-13 sorok) illetve a Javascript kód bázis kiindulópontját (require.main.js fájl). Mivel a Javascript nyelv nem támogat natívan egy modul kezelő rendszert, amellyel strukturálni lehet a kódfájlokat, felhasználtam a Require.js függvénykönyvtárt. Ennek segítségével elkülöníthetjük egymástól az egyes fájlokban definiált objektumokat, nem kell beszennyezzük a globális névteret különböző függvények definíciójával, és egy jól strukturált, átlátható kód bázist hozhatunk létre. A megfelelő függőségeket csak akkor töltjük le a szerverről, ha azokra szükségünk van, így nem kell minden egyes, az alkalmazás által felhasznált Javascript fájlt az alap HTML fájlban betölteni. A 25. sorban található szkript betöltés feltételhez kötött (csak fejlesztés során használandó), és arra szolgál, hogy lehetővé tegye a böngészőben futtatott alkalmazás újratöltését minden egyes fájlmentés alkalmával. Ez nagyon hasznos, mert csökkenti a fejlesztési időt, gyakorlatilag csak átváltunk a böngészőre, és máris a legfrissebb verzióval dolgozunk. Ez a grunt-express-server segítségével valósul meg, amely egy szerveret indít el a 35729-es porton, és a WebSocket protokoll által értesíti a böngészőt, hogy módosultak a Javascript vagy más fájlok az alkalmazáson belül, és szükség van a böngésző frissítésére a legújabb verzió letöltése végett.

A kliens oldali függőségek meghatározására a Bower csomagkezelő rendszert használtam, amely funkcionalításban hasonló az NPM-hez azzal a különbséggel, hogy a böngésző számára tartalmaz komponenseket. A felhasznált komponensek listája a bower.json fájlban található az alapkönyvtáron belül:

```

{
  "name": "DrugsDB",
  "description": "",
  "main": "server.js",
  "dependencies": {
    "jquery": "~2.1.4",
    "jquery-ui": "jquery.ui#~1.11.4",
    "angular": "~1.4.7",
    "angular-route": "~1.4.7",
    "angular-resource": "~1.4.7",
    "underscore": "~1.8.3",
    "bootstrap": "~3.3.5",
    "angular-bootstrap": "~0.14.3",
    "bcryptjs": "~2.3.0",

```

```

    "dexie": "~1.2.0",
    "offline": "~0.7.15"
  },
  "devDependencies": {
    "angular-mocks": "~1.4.9",
    "offlinejs-simulate-ui": "~0.1.2",
    "requirejs": "requirejs-bower#^2.2.0"
  }
}

```

Ebben a fájlban is két fontos komponenst határozhatunk meg: dependencies (tartalmazza a termelési fázisban használt komponsek listáját) és devDependencies (tartalmazza a fejlesztés alatt felhasznált komponenseket).

A kliens oldali alkalmazás az Angular.js keretrendszerben íródott, amely az MVC (model-view-controller) paradigmára épül, és szorgalmazza a moduláris fejlesztést, lehetőséget biztosítva a modulok könnyű tesztelhetőségére. Az Angular.js magában foglal egy függőség kezelő rendszert (Dependency Injection – DI), amely által a modulok függőségeinek példányosítását futásidőben határozza meg. Így a modulok könnyen szétválaszthatók, nincsenek szorosan egymáshoz kötve (loose coupling), amely lehetővé teszi a tesztelést, mivel könnyen kicserélhetők az egyes függőségek hamis objektumokkal. A keretrendszer viszont nem támogatja a Javascript fájlok feltétel szerinti betöltését (például csak akkor van szükségem a b.js fájlra, ha az a.js fájlt használom vagy meg akarom határozni, hogy az c.js fájl függ d.js fájltól). Ebben segít a Require.js, amely ezeket a követelményeket teljesíti.

A require.main.js fájlban definiáltam a külső függőségeket (third party libraries), amelyre az alkalmazásnak szüksége lehet. Ezeknek a külső könyvtáraknak az abszolút elérési útvonalait egy-egy rövid névvel helyettesítettem (például az „angular” a „public/lib/angular/angular” útvonalat helyettesíti). A fájl utolsó utasítása require(["app/app.init"]) meghatározza, hogy az app.init.js fájl lesz az kliens alkalmazás kiindulópontja. Ebben a pillanatban nincsen még semmilyen külső függőség betöltve a böngészőbe ezen a fájlon kívül. Tekintsük meg az app.init.js fájl definícióját:

```

1.  define([
2.      "angular",
3.      "uiBootstrapTpls",
4.      "bootstrap",
5.      "offline",
6.      "offlineSimulateUI",
7.      "app/main/main.module",
8.      "app/users/users.module",
9.      "app/admin/admin.module",
10.     "app/widgets/widgets.module",
11.     "app/common/common.module"],
12.     function (angular,
13.         uiBootstrapTpls,
14.         bootstrap,
15.         offline,
16.         offlineSimulateUI,
17.         mainModule,
18.         usersModule,
19.         adminModule,
20.         widgetsModule,
21.         commonModule) {

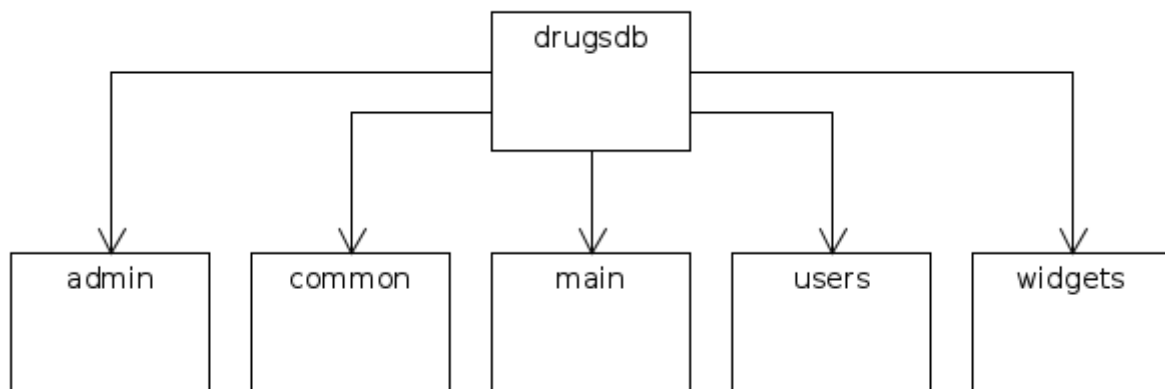
```

```

22.
23.         var drugsdb = angular.module("drugsdb", ["ngRoute", "ui.bootstrap", "users", "main",
"admin", "widgets", "common"]);
24.
25.         angular.bootstrap(document, ["drugsdb"]);
26.     }
27. );

```

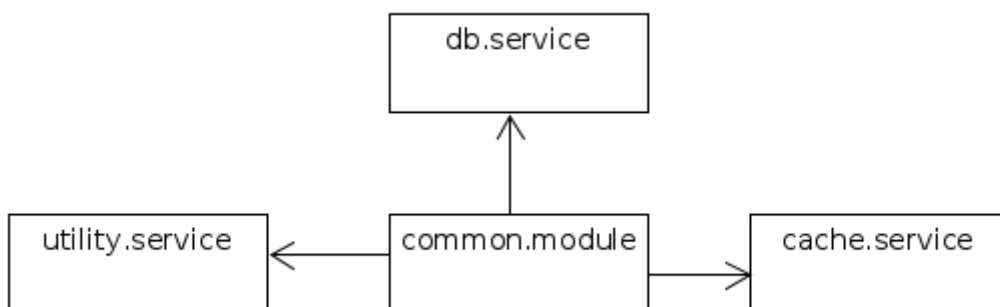
Ebben a fájlban határoztam meg az összes modult, amelyre Angular.js alkalmazásnak szüksége van. Ezen függőségek között vannak külső függőségek (ui.bootstrap vagy offline) és az alkalmazásban használt belső modulok (user, main, stb.). A 25. sorban történik a létrehozott drugsdb modell hozzárendelése a DOM-hoz. Tekintsük meg a következő diagramot, amely magában foglalja a felhasznált modulokat:



5.10. Ábra: Kliens komponens modul diagramja

Ezen modulok felépítését a következő alfejezetekben részletezem.

### 5.5.1. A common modul



5.11. Ábra: A common modul osztálydiagramja

Ez a modul tartalmazza a közös komponenseket, amelyekre szükség lehet bármely más modul esetében (admin, users vagy main). Olyan funkciókat tartalmaz, amelyek gyakori

segédműveleteket valósítanak meg. A következő komponenseket foglalja magában:

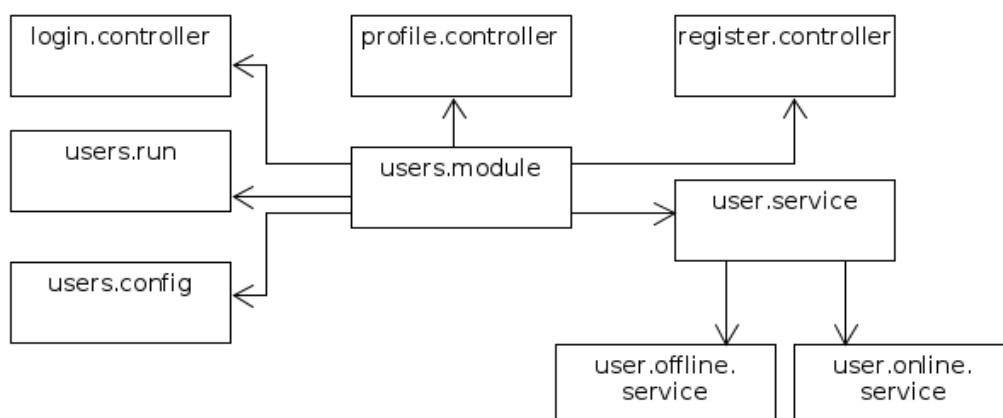
- `cacheService`: ez a modul arra szolgál, hogy az Angular.js rendszer \$http komponense által eltárolt adatokat kezelje. Például egy felhasználó lekérése a `GET /user` típusú URL segítségével történik. Ha azt szeretnénk, hogy a visszatérített JSON adatot rögzítsük az Angular.js cache tárában későbbi elérés céljából anélkül, hogy ismét egy újabb kérést küldenénk a szerverhez, akkor felhasználhatjuk a `cacheService` által definiált `store()` függvényt. Amennyiben törölni szeretnénk ezt az adatot, meghívhatjuk az `invalidate()` függvényt.
- `dbService`: ez a modul egy proxy mintán alapul; felhasználja a Dexie külső függvénykönyvtárt ahhoz, hogy létrehozza az IndexedDB adatbázist a böngészőben, és műveletek végrehajtását biztosítsa. Erre a modulra hivatkozik a `users` modul ahhoz, hogy offline módban elérje az előzőleg bejelentkezett felhasználó személyes adatait és keresési listáját.
- `utilityService`: segédfüggvényeket tartalmaz mint a `generateGuid()`, amely egyedi azonosítókat (ID) állít elő

### 5.5.2. A widgets modul

Ebben a modulban a megjelenítendő UI (user interface) komponensek találhatók Angular.js direktívák formájában. Ezek a direktívák kapcsolatba lépnek a DOM-mal (document object model) ahhoz, hogy kiegészítsék a HTML által támogatott komponensek viselkedését, megjelenítését. Olyan modulokat határoztam meg itt, amelyek újrahasználhatók a teljes alkalmazásban.

### 5.5.3. A users modul

Ez a modul az alkalmazás felhasználóinak kezelését valósítja meg. Ide tartozik a felhasználók bejelentkezése, feliratkozása, profil adatainak a megváltoztatása. A modul definíciója a `users.module.js` fájlban történik, amelyben meghatározom, hogy milyen Angular.js komponenseket fog magában foglalni ezen modul. Tekintsük meg a következő diagramot:



5.12. Ábra: A users modul osztálydiagramja

A modulok definíciójakor arra törekedtem, hogy minél jobban szétválasszam őket egymástól azért, hogy betartsam a egyedi felelősségű komponensek elvét (single responsibility principle). Ez hozzájárul a kód könnyű tesztelhetőségéhez és karbantartásához. A modul definíciókra jellemző, hogy (ez esetben user.module.js fájl) meghatározom a felhasznált komponenseket (controller, factory, constant). A Require.js biztosítja a kódfájlok betöltését a böngészőbe, viszont ezeket még regisztrálni kell az Angular.js függőség kezelő rendszerének segítségével ahhoz, hogy ezek futásidőben elérhetők és használhatók legyenek.

A users.config modulban meghatároztam a különböző URL-eket, amelyeket a users modul használ. Mivel a böngészőben futó alkalmazás egy oldalú alkalmazás (Single Page Application), ezért a különböző oldalak közötti navigációt az Angular.js keretrendszer fogja kezelni. Amikor egyik oldalról egy másikra navigál a felhasználó, a böngészőben megjelenő URL-t fogjuk módosítani anélkül, hogy újratöltenénk a teljes weboldalt. Mivel célom volt a régebbi böngészők támogatása, ezért a böngésző URL-jében a „#” után tároltam el az oldal útvonalát, ugyanis a HTML 5-öt nem támogató böngészők esetében nem alkalmazható az oldal URL-jének változtatása a weboldal teljes újratöltése nélkül. A „#” karakter utáni rész módosítása nem vonja maga után a weboldal teljes újratöltését, és az új böngészőkben is támogatott. A következő útvonalakat határoztam meg:

/user/login	A bejelentkezési oldal által lehet belépni a rendszerbe; ezen megadható a felhasználónév és jelszó, és követhető a link a feliratkozás oldalra
/user/register	A feliratkozási oldalon megadhatóak a felhasználó adatai, és beállítható a keresési eszköz típusa: MedlinePlus vagy DBPedia
/user/profile	Ezen a oldalon módosíthatja a felhasználó az adatait; megváltoztathatja a jelszavát, email címét és a keresési eszközt

5.5. Táblázat: A users modul által támogatott URL azonosítók listája

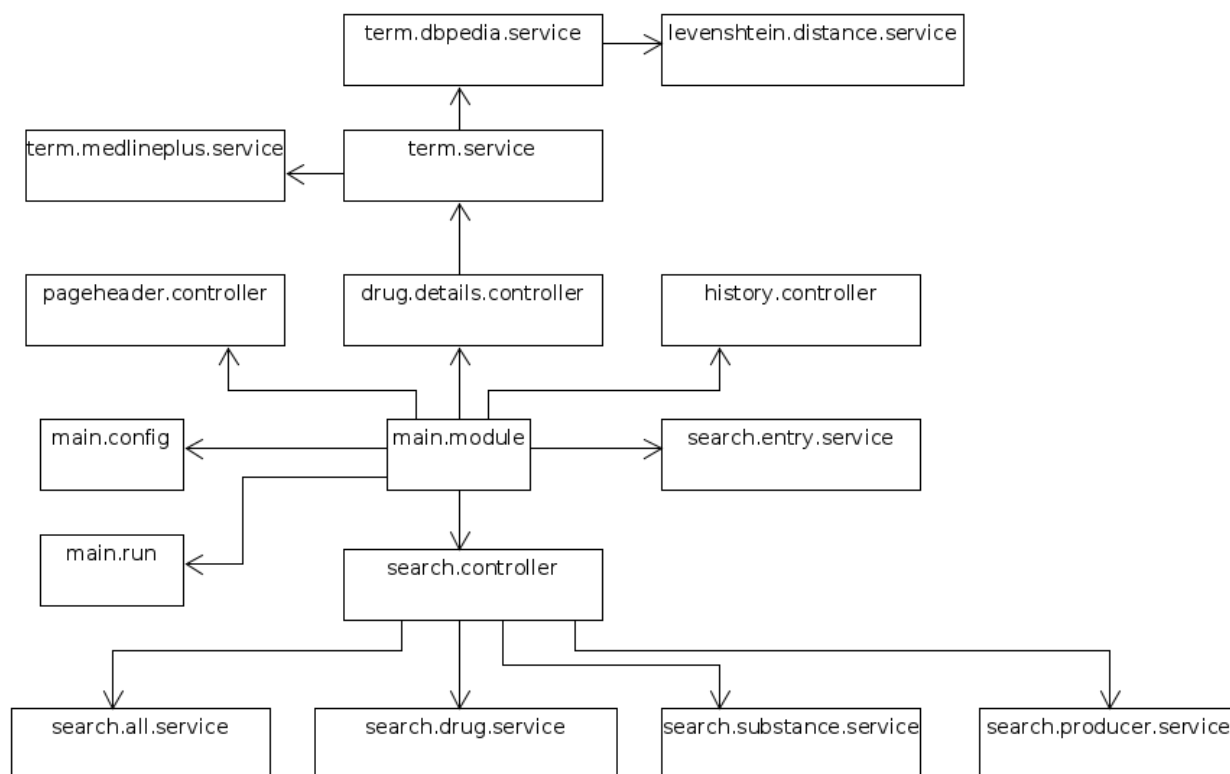
A userService modul a stratégia tervezési mintára épül, felhasználja a userOnlineService és userOfflineService modulokat ahhoz, hogy a felhasználóval kapcsolatos műveleteket elvégezze. Amennyiben az alkalmazás online módban fut, a userService modul a userOnlineService modulhoz továbbítja a kéréseket, ha pedig nincsen internetes kapcsolat, akkor a kéréseket a userOfflineService modul szolgálja ki. Az internetes kapcsolat állapotának (online vagy offline) megállapítását az Offline.js külső komponens segítségével valósítottam meg.

A userOnlineService egy Angular.js által támogatott factory típusú modul, amely a beépített \$resource modul segítségével definiál egy REST csomóponthoz való hozzáférést. Alapértelmezésben támogatja a megszokott HTTP műveleteket: erőforrás lekérése (GET), létrehozása (POST), módosítása (PUT), törlése (DELETE). Csak abban az esetben kell újakat létrehozni, ha azok különböznek az említett műveletektől (például a getByUsername nevű metódus egy GET kérést kezdeményez a „/user/:id/:username” sablon alapján).

A userOfflineService támogatja az alkalmazás használatát offline módban. Ebben az esetben, ha a böngésző tartalmazza az alkalmazást a cache tárában, és előzőleg a felhasználó már bejelentkezett, akkor ez a modul lehetővé teszi, hogy a felhasználó bejelentkezzen, és megtekintse az előző kereséseinek eredményét. Az adat eltárolása az IndexedDB segítségével történik, amely egy böngészőben futó adatbázisrendszer [20]. Az IndexedDB egy tranzakciós adatbázisrendszer, amely egy Javascript alapú interfészen keresztül biztosít tárolási lehetőséget struktúrált adathalmaz

számára. Az adatbázis műveletek aszinkron módon hajthatók végre azért, hogy ne blokkolják a futó alkalmazást. Az offline műveleteket úgy végezzük el, hogy online módban eltároljuk a bejelentkezett felhasználó adatait az IndexedDB adatbázisban a gyógyszerkeresésekkel együtt, majd amikor az alkalmazás offline módba kerül, a userOfflineService modulon keresztül kommunikálunk az adatbázissal. Mindezt úgy valósítottam meg, hogy a controller modul számára mindez átlátszó maradjon, vagyis a műveleteket (például bejelentkezés) a közös userService modul által biztosított interfészen keresztül végeztem.

#### 5.5.4. A main modul



5.13. Ábra: A main modul osztálydiagramja

Ez a modul tartalmazza az oldal fejlécének a definiálását, gyógyszerek keresését, kölcsönhatások listázását, előzmények tárolását.

A `PageHeaderController` osztály az oldal fejlécének a kezelésére szolgál. A `page.header.html` fájl deklaratív módon definiálja ezen osztályt, amely irányítja ennek működését (ez tölti be a controller szerepét az MVC paradigmában).

```
<nav class="navbar navbar-inverse navbar-fixed-top" role="navigation" ng-controller="PageHeaderController as vm" ng-class="{ 'disabled': vm.isDisabled }"> ...</nav>
```

A következő linkeket találjuk meg a fájl definíciójában:

- `#/admin/import`: az admin modul import oldalára küldi a felhasználót,
- `#/search`: a main modul kereső oldalára küldi a felhasználót,
- `#/history`: a main modul keresési előzmények oldalára küldi a felhasználót,
- `#/user/profile`: a user modulnak a felhasználó profiljának a megváltoztatását megvalósító oldalára helyezi át a böngészőt
- `#/contact`: main modul kapcsolat oldalára helyezi át a böngészőt.

Mivel két típusú felhasználót kell a rendszer megkülönböztessen egymástól, és hitelesíteni kell a bejelentkezett felhasználót, ezért szükség van az egyes oldalak levédésre (a rendszernek irányítania



kell a felhasználók hozzáférési jogait – access control). A main modul esetében ez a feladat a main.config.js fájlban történik, ahol meg vannak határozva az előbb említett URL-ek leképezései a valódi controller típusú osztályokra. A „/search” URL a SearchController osztályra képződik le, míg a „/history” a HistoryController osztályt használja. Az auth() függvény valahányszor lehívódik, amikor a felhasználó az említett URL-k egyikét megpróbálja elérni (legyen az a fejléc valamelyik nyomógombjának megkattintása által vagy pedig a böngészőbeli webcím módosítása által), és leellenőrzi, hogy a felhasználó előzőleg bejelentkezett-e a rendszerbe. Ez a vizsgálat a userService segítségével történik (a userService osztály a user modulban került definiálásra, amely visszatéríti a bejelentkezett felhasználó objektumot, ha a bejelentkezés előzőleg sikeres volt, ellenkező esetben a \$q szerviz objektum által visszautasítja a kérést). Amennyiben a kérés visszautasításra kerül, a böngésző a bejelentkezési oldalra ugrik (ennek URL-je: „/user/login”).

A SearchController osztály tartalmazza a gyógyszerek keresésének megvalósítását. Az osztály konstruktora magában foglalja a függőségeket, amelyek különböző paraméterek szerinti keresést, illetve az előző keresésekre való visszatérést valósítja meg. Nézzük, milyen függőségekre van szükségünk:

\$scope	Ezen szerviz segítségével figyelni tudjuk böngésző webcím mezőjének a változását.
\$location	Ezen szerviz által elérjük a böngésző webcím mezőjében tárolt paramétereket (például „type=searchDrugService”).
\$injector	Ez a szerviz valósítja meg, hogy futásidőben hozzuk létre a keresési típus szöveges értéke alapján a megfelelő szerviz objektumot: például searchDrugService vagy searchProducerService.
\$uibModal	Dialógus típusú ablakok megjelenítését ezen szerviz segítségével végezzük.
searchEntryService	Előzmények eltárolását valósítja meg.
searchAllService	Típustól független keresést valósít meg.
searchDrugService	Gyógyszernév szerinti keresést valósít meg.
searchProducerService	Gyártónév szerinti keresést valósít meg.
searchSubstanceService	Alapanyagnév szerinti keresést valósít meg.

5.6. Táblázat: A SearchController osztály függőségei

A „\$” karakterrel kezdődő változónevek az Angular.js keretrendszer által biztosított szerviz típusú osztályok példányai, a többi változó az általam definiált szerviz osztályok példányai. Jelen esetben ezen komponensek a services könyvtárban belül vannak definiálva a main modulban. Ezek betöltése a függőségkezelő rendszer (dependency injection – DI) segítségével történik futásidőben; a rendszer név szerint azonosítja a megfelelő objektumtípust, és a singleton tervezési minta alapján szolgáltatja azt a felhasználó osztályok számára (egyetlen példány jön létre minden szerviz típusú komponensből).

A keresés funkció által a felhasználó megad egy kifejezést, amelyre keresni szeretne (ez lehet egy része a valódi gyógyszernévnek, például a „dragon” kifejezésre keresve a rendszer meg fogja találni a „DragonTabs” nevű gyógyszert), majd kiválasztja a keresés típusát (gyógyszernév, gyártónév, alapanyagnév vagy bármi ezek közül), és megnyomja a „Search” nyomógombot.

Az ábra egy gyógyszernév szerinti keresést mutat; a szöveges doboz bal oldalán a gyógyszernév volt kiválasztva. A keresés funkció az említett 4 típusú szerviz komponens segítségével történik.

The screenshot shows the DrugsDB website with a search bar containing 'dragon'. Below the search bar, a table displays the search results for 'DragonTabs' by Genomma Lab USA. The table lists various ingredients including BAPTISIA TINCTORIA ROOT, IBUPROFEN, SILICON DIOXIDE, STARCH, CORN, HYPROMELLOSES, FERRIC OXIDE RED, MAGNESIUM STEARATE, CELLULOSE, MICROCRYSTALLINE, SODIUM STARCH GLYCOLATE TYPE A CORN, TITANIUM DIOXIDE, LACTOSE, POLYDEXTROSE, and POLYETHYLENE GLYCOLS.

#	Name	Producer	Ingredients
1	DragonTabs	Genomma Lab USA	BAPTISIA TINCTORIA ROOT IBUPROFEN SILICON DIOXIDE STARCH, CORN HYPROMELLOSES FERRIC OXIDE RED MAGNESIUM STEARATE CELLULOSE, MICROCRYSTALLINE SODIUM STARCH GLYCOLATE TYPE A CORN TITANIUM DIOXIDE LACTOSE POLYDEXTROSE POLYETHYLENE GLYCOLS

#### 5.14. Ábra: Az alkalmazás kereső oldala

Ezen komponensek egy közös interfészt valósítanak meg, mindegyik tartalmaz egy search() nevű metódust, amelyet a SearchController osztály meghívhat. Mivel Javascript nyelvben interfész típusú komponensek nem léteznek, ezért ennek megvalósítása nem foglalja magában ezeknek a kapcsolatoknak a kompilálás időben történő leellenőrzését, viszont ezt a feladatot helyettesítheti az egységteszt (unit tests) használata. A megfelelő szerviz objektum kiválasztása futásidőben történik a következő kódrészlet segítségével (ebben az esetben a gyógyszernév szerinti keresést használtuk):

```
function searchByDrug() {
    vm.results = null;
    vm.searchService = searchDrugService;
}
```

Mivel az alkalmazásnak támogatnia kell a böngésző előre illetve hátra (back, forward buttons) nyomógombjainak a használatát, ezért figyelni kell az URL cím változását (ez történhet a a nyomógombok kattintásával vagy pedig az URL cím átírásával is). Ezt a következő kódrészlet valósítja meg:

```
function checkSearchParams() {
    try {
        var name = $location.search().type;

        vm.searchTerm = $location.search().term;
        vm.searchService = $injector.get(name);

        search();
    }
    catch(e) {
        $location.search("type", "");
    }
}
```

```

        searchByDrug();
    }
}

```

Ebben az esetben az előző függvény minden egyes URL változásra meg fog hívódni. A \$location szerviz segítségével lekérdezzük az aktuális URL címből a „type” kifejezés értékét (például a „#/search?type=searchDrugService&term=dragon” esetben a „type” kulcs értéke a „searchDrugService”), majd lekérdezzük a keresett kifejezés értékét is (az említett példában a „term” kulcs értéke a „dragon” kifejezés). Felhasználtam az \$injector szervizt ahhoz, hogy a valódi objektumot elérjem a szerviz neve alapján. Miután ezen értékeket eltároltam, meghívtam a search() metódust, amely elküldi a kérést a szerverhez, és lekezeli a visszakapott eredményeket.

Az eredmények megjelenítése a results könyvtárban definiált HTML sablonok által történik. Itt három HTML sablon (template) található (drug.html, producer.html és substance.html), mivel az eredmények szemantikája változó, más HTML szerkezetű komponenseket jelenítünk meg. A megfelelő HTML sablon kiválasztása a SearchController osztályon belül történik a getResultsTemplateName() metódus által, amely a kiválasztott szerviz template mezője függvényében kerül kiértékelésre a search.html fájlban belül:

```
<ng-include src="vm.getResultsTemplateName()" ng-if="vm.results.length > 0"></ng-include>
```

Amennyiben a gyógyszerek közötti kölcsönhatásokat kívánjuk megvizsgálni, a keresést gyógyszernév szerint kell végezzük, majd a kilistázott eredmények valamelyikét kiválasztva (megnyomjuk a táblázat sorainak jobb oldalán megjelenő nyomógombot), egy dialógus típusú ablak segítségével jelenítjük meg a részleteket:

DrugsDB
Admin
Search
History
Profile
Contact
a
Log out

Drug details

Name
DragonTabs

Producer
Genomma Lab USA

Ingredients
BAPTISIA TINCTORIA ROOT; IBUPROFEN; SILICON DIOXIDE; STARCH, CORN; HYDROXYMETHYLCELLULOSE; FERRIC OXIDE RED; MAGNESIUM STEARATE; CELLULOSE, MICROCRYSTALLINE; SODIUM STARCH GLYCOLATE TYPE A CORN; TITANIUM DIOXIDE; LACTOSE; POLYDEXTROSE; POLYETHYLENE GLYCOLS;

Interactions
Ibuprofen; Aspirin; Anti-Inflammatory Agents, Non-Steroidal; Nonsteroidal Anti-inflammatory Drug [EPC]; Anticoagulants; Anti-coagulant [EPC]; Steroid containing root canal medicament; Pharmaceutical Preparations; Drugs, Non-Prescription; Naproxen; Diuretics; Pharmacologic Substance;

Additional information

Warnings
Do not use
Ask a doctor before use if
Ask a doctor or pharmacist before use if
When using this product
Stop use and ask a doctor if

If pregnant or breast-feeding
Do not use
Ask a doctor before use if
Ask a doctor or pharmacist before use if

Allergy alert Ibuprofen may cause a severe allergic reaction, especially in people allergic to aspirin. Symptoms may include hives facial swelling asthma (wheezing) shock skin reddening rash blisters If an allergic reaction occurs, stop use and seek medical help right away. Stomach bleeding warning This product contains a nonsteroidal anti-inflammatory drug (NSAID), which may cause severe stomach bleeding. The chances are higher if you are age 60 or older have had stomach ulcers or bleeding problems take a blood thinning (anticoagulant) or steroid drug take other drugs containing prescription or nonprescription NSAIDs (aspirin, ibuprofen, naproxen, or others) have 3 or more alcoholic drinks every day while using this product take more or for a longer time than directed.

#	Keyword	Preferred keyword	Type	Description
1	Allergy	Allergic disposition	disease or syndrome	
2	Allergic drug	Drug Allergy	disease	A drug allergy is an allergy to a drug, most commonly a medication. Medical attention should be sought immediately if an allergic reaction is suspected. An allergic reaction will not occur on the first exposure to a substance. The first exposure allows the body to create antibodies

5.15. Ábra: Gyógyszeradatokat megjelenítő oldal az alkalmazásban

A megjelenített dialógus típusú ablakban feltüntettem a gyógyszer nevét, gyártóját, összetevőit, a kölcsönhatásban álló gyógyszerek és alapanyagok neveit, illetve a figyelmeztető útmutatókat a gyógyszer használatáról. A kölcsönhatásban álló gyógyszerek és alapanyagok nevei linkek formájában jelennek meg, és arra szolgálnak, hogy rákattintva elérjük a megfelelő gyógyszer adatait. Alább a figyelmeztetéseket jelenítettem meg a Bootstrap.UI könyvtár által biztosított komponens által, amely különálló lapozást (tab control) tesz lehetővé. Az összes lap fejléce egyidőben látható, viszont csak egyetlen lap tartalma jeleníthető meg egyszerre. Minden lapon belül, megjelenítettem az eredeti XML fájlból kinyert struktúrátlan szöveget, majd alatta táblázat formájában látható az egyes betegségek és tünetek megnevezése egy rövid magyarázat kíséretében. Ennek a magyarázatnak a forrása vagy a DBPedia vagy pedig a MedlinePlus portálok. Ennek a kiválasztása a felhasználó profiljának az oldalán határozható meg.

A termService nevű szerviz komponens szolgál a figyelmeztetéseket tartalmazó táblázat magyarázatainak a meghatározására. A stratégia tervezési mintára épül egy közös interfészt biztosítva a DrugDetailsController osztály számára. Ez a komponens a userService szerviz komponens által visszatérített keresési szolgáltatás által végzi el a keresést.

```
function search(term) {
    var deferred = $q.defer();

    userService.current.get().$promise.then(function (user) {
        var serviceImpl;
        try {
            serviceImpl = $injector.get(user.termServiceProvider)
        }
        catch (e) {
            serviceImpl = termDbpediaService;
        }
        serviceImpl.search(term).then(deferred.resolve, deferred.reject);
    });
    return deferred.promise;
};
```

Két osztályt definiáltam, amely implementálja a termService interfész által meghatározott search() metódust. A termDbpedia szerviz a DBPedia portál adatbázisát használja fel a figyelmeztetési útmutatókban megjelenő betegségek és tünetek fogalmainak a keresésére. A kereséseket futásidőben végeztem a felhasználó böngészőjében a gyógyszerkeresés eredményének megérkezése után. A DBPedia.org portál által szabadon hozzáférhetővé tett SPARQL végpontot (<http://dbpedia.org/sparql>) használtam fel a REST alapú lekérdezések végrehajtására, amely JSON formában szolgáltatja a választ. Egy tipikus ilyen eredmény volt a következő, amely az „Urticaria” kifejezésre érkezett vissza a böngészőbe:

```
▼ {head: {link: [], vars: ["s", "s2", "label", "comment", "s3"]}, ...}
▶ head: {link: [], vars: ["s", "s2", "label", "comment", "s3"]}
▼ results: {distinct: false, ordered: true, ...}
▼ bindings: [{s: {type: "uri", value: "http://dbpedia.org/resource/Urticaria"}, ...}, ...]
▼ 0: {s: {type: "uri", value: "http://dbpedia.org/resource/Urticaria"}, ...}
▼ comment: {type: "literal", xml:lang: "en", ...}
type: "literal"
value: "Urticaria (from the Latin urtica, "nettle" from urere, "to burn"), commonly referred to as hives, is a kind of skin rash notable for pale red, raised, itchy bumps. It is usually caused by an allergic reaction to something in the environment."
xml:lang: "en"
▼ label: {type: "literal", xml:lang: "en", value: "Urticaria"}
type: "literal"
value: "Urticaria"
xml:lang: "en"
▼ s: {type: "uri", value: "http://dbpedia.org/resource/Urticaria"}
type: "uri"
value: "http://dbpedia.org/resource/Urticaria"
▼ 1: {s: {type: "uri", value: "http://dbpedia.org/resource/Cold_urticaria"}, ...}
▶ comment: {type: "literal", xml:lang: "en", ...}
type: "literal"
value: "Cold urticaria"
xml:lang: "en"
▶ s: {type: "uri", value: "http://dbpedia.org/resource/Cold_urticaria"}
type: "uri"
value: "http://dbpedia.org/resource/Cold_urticaria"
▶ 2: {s: {type: "uri", value: "http://dbpedia.org/resource/Pressure_urticaria"}, ...}
▶ 3: {s: {type: "uri", value: "http://dbpedia.org/resource/Drug-induced_urticaria"}, ...}
▶ 4: {s: {type: "uri", value: "http://dbpedia.org/resource/Aquagenic_urticaria"}, ...}
▶ 5: {s: {type: "uri", value: "http://dbpedia.org/resource/Cholinergic_urticaria"}, ...}
```

5.16. Ábra: SPARQL végpont által visszatérített JSON objektum

Megtörténhet, hogy a SPARQL végpont által visszatérített eredmények listája túlságosan nagy, és olyan kifejezéseket is tartalmaz, amelyek nem relevánsak a bemeneti paraméterek esetében. Ezért felhasználtam a levenhsteinDistanceService szerviz komponensét, amely arra szolgál, hogy összehasonlítsa a két kifejezés értékét egy súlyfüggvény által. Ennek a komponensnek a segítségével összehasonlítottam az összes DBPedia által visszatérített kifejezést az eredeti keresett kifejezéssel, és csak azt jelenítettem meg a oldalon, amelynek a távolsága a legkisebbnek bizonyult.

A termMedlinePlusService osztály szintén a termService interfészt implementálja. Ebben az esetben a MedlinePlus portál szolgáltatja az eredményeket, amelyet egy webszerviz által tesz elérhetővé (<https://wsearch.nlm.nih.gov/ws/query>). A webszerviz URL-jében a „query” kulcsszó helyébe a megfelelő keresendő kifejezést kell beírni, majd elküldeni. A visszatérített válasz XML formátumban érkezik meg, és a következőképpen alakult a <http://localhost:4000/main/searchmedterm?term=Urticaria> URL esetében:

```
<nlmSearchResult>
<term>Urticaria</term>
<file>viv_RPacuC</file>
<server>pvlbsrch16</server>
<count>12</count>
<retstart>0</retstart>
<retmax>10</retmax>
<list num="12" start="0" per="10">
<document rank="0" url="https://www.nlm.nih.gov/medlineplus/hives.html">
<content name="title"><span class="qt1">Hives</span></content>
<content name="organizationName">National Library of Medicine</content>
<content name="altTitle"><span class="qt0">Urticaria</span></content>
<content name="FullSummary"><p><span class="qt1">Hives</span> are red and sometimes itchy bumps on your skin. An allergic reaction to a drug or food usually causes them. Allergic reactions cause your body to release chemicals that can make your skin swell up in <span class="qt1">hives</span>. People who have other allergies are more likely to get <span class="qt1">hives</span> than other people. Other causes include infections and stress.</p><p><span class="qt1">Hives</span> are very common. They usually go away on their own, but if you
```

have a serious case, you might need medicine or a shot. In rare cases,

...

Ezen XML dokumentum elemzése és a megfelelő meghatározás kinyerése (a `<content name="FullSummary">...</content>` elem) a jQuery függvénykönyvtár segítségével történik, amely a HTML oldalbeli DOM struktúrához hasonlóan támogatja az XML dokumentumokban való keresést.

A `HistoryController` támogatja a keresési előzmények megtekintését; az oldalt a History linkre kattintva a „`#/history`” URL segítségével érhetjük el. Itt megtaláljuk az összes keresést csökkenő időrendi sorrendben, amelyet a felhasználó végzett. Megjelenítettem a keresés típusát, a keresett kifejezést és a dátumot. Lehetőség van törölni a listából a bejegyzést az X link kiválasztásával.

#	Criteria	Term	Date	
1	drug	dragon	5/16/16 1:27 PM	✕
2	drug	dragon	5/16/16 12:15 PM	✕
3	drug	dragon	5/16/16 12:14 PM	✕
4	drug	dragon	5/16/16 12:13 PM	✕
5	drug	dragon	5/16/16 11:25 AM	✕
6	drug	10 person	5/16/16 11:25 AM	✕
7	drug	10 person	5/15/16 11:42 PM	✕
8	drug	10 person	5/15/16 11:40 PM	✕

5.17. Ábra: Az alkalmazás előzményeit megjelenítő oldal

Ez a controller típusú osztály a `searchEntryService` szervízen keresztül kommunikál a szerverrel az adatok megjelenítése és törlése végett. A `searchEntryService` az Angular.js által biztosított `$resource` szervíz segítségével valósítja meg a szerveroldali REST végponttal történő kapcsolat kiépítését (ezt a „`/searchentry/:id`” minta megadásával értem el).

### 5.5.5. Az admin modul

Az admin modul arra szolgál, hogy a rendszergazda típusú felhasználók számára biztosítsa a gyógyszer adathalmaz feldolgozását (releváns információk elemzése, Pig műveletek végrehajtása, adatok beszúrása az adatbázisba, stb.).

DrugsDB	Admin	Search	History	Profile	Contact	a	Log out
---------	-------	--------	---------	---------	---------	---	---------

Import data into the database

Start

5.18. Ábra: Adatelemzés elindítása az alkalmazáson belül

A feldolgozás elindítása egy egyszerű interfész segítségével valósítható meg, amely egyetlen nyomógombot tartalmaz. A nyomógomb aktiválásával az adminService modulon keresztül meghívódik az „/admin/import” POST típusú kérés, amely elindítja a szerveren az adatok feldolgozását. Mivel ez egy hosszabb folyamat, így nem tudunk rögtön visszatéríteni a böngészőnek egy választ az eredményről. A Socket.IO függvénykönyvtár segítségével lekezelhetjük a valós időben érkező üzeneteket a szerverről. Ez a függvénykönyvtár a WebSocket protokollon alapul, amely kétirányú (full-duplex) kommunikációt valósít meg a böngésző és a szerver között.

A felhasználó eltávozhat erről az oldalról akár, de mégis értesítést kap a befejezett szerveroldali munkafolyamat eredményéről. Az ImportController modulban található a socket kapcsolat kiépítése, ahol egy eseményfigyelőn keresztül kezeljük le a szerverről érkező adatokat, és frissítjük az oldalon megjelenített folyamatkijelzőt (progressbar).

## 6. Tesztelés és eredmények

### 6.1. Gyógyszerkölsönhatások tesztelése

Az alkalmazás tesztelését több lépésben végeztem el. Először kiválasztottam 10 darab XML dokumentumot, amelyek mindegyike egy-egy gyógyszer adatait tartalmazta különböző kölcsönhatásokkal együtt. A dokumentumokat manuálisan vizsgáltam meg, és feljegyeztem a következő értékeket:

- **TP (true positive):** helyes találatok száma,
- **TN (true negative):** helyes mellőzött találatok száma (olyan gyógyszernevek, amelyeket nem kívánunk megjeleníteni, mivel nem állnak kölcsönhatásban: erre példa lehet az aktuális gyógyszer neve, amelyet nem kell a kölcsönhatások listájában feltüntetni),
- **FP (false positive):** hamisan kiválasztott találatok száma (olyan gyógyszernevek, amelyeket a rendszer kiválasztott, mint kölcsönhatásban álló példányok, viszont valójában nem létezik kölcsönhatás közöttük és az aktuális gyógyszer között),
- **FN (false negative):** hamisan nem kiválasztott találatok száma (olyan gyógyszernevek, amelyeket a rendszer nem választott ki, viszont kölcsönhatásban állnak az aktuális gyógyszerrel).

Tekintsük példának a következő kijelentést, amely az Aspirin gyógyszert leíró XML dokumentumból származik:

„Aspirin has adverse effects if administered in combination with Arava but it is safe to take it with multivitamins.” (Az Aspirin káros mellékhatásokat okoz az Arava gyógyszerrel, viszont biztonságos multivitaminnal kombinálni.)

Ebben az esetben az Aspirin az alany, és azt kívánjuk eldönteni, hogy mely gyógyszerrel szedhető egyszerre és mely gyógyszerrel okozhat káros mellékhatásokat. A következő táblázatban összefoglaltam az előzőleg definiált fogalmak értékét az előző kijelentésre nézve:

	Arava	Multivitamin
Kiválasztott	TP	FP
Nem kiválasztott	FN	TN

6.1. Táblázat: Aspirin kölcsönhatása Arava és Multivitamin találatok esetén

A teszteléshez felhasznált 10 gyógyszer esetében a következő eredményekhez jutottam:

Gyógyszer	TP	TN	FP	FN
DragonTabs	6	0	3	0
Actidose	4	2	3	4
Arava	26	0	5	0
10 Person	9	0	5	1
Plavix	11	1	4	0
Warfarin	133	1	13	1
Absorica	18	0	5	1
Ambien	5	4	3	1
Naproxen	8	0	4	0
Activase	18	0	4	0

6.2. Táblázat: Találatok minősítése 10 kiválasztott gyógyszer esetében

A következő táblázat egy átfogó képet nyújt ezen mértékekről:

	Helyes	Helytelen
Kiválasztott gyógyszerek	TP	FP
Kihagyott gyógyszerek	FN	TN

6.3. Táblázat: Találatok minősítése

A rendszer kiértékeléséhez a következő képleteket használtam fel:

Típus	Képlet	Érték
Teljes pontosság (accuracy)	$\frac{TP+TN}{TP+TN+FP+FN}$	81%
Szelektív precizitás (precision)	$\frac{TP}{TP+FP}$	82%
Helyesség (recall)	$\frac{TP}{TP+FN}$	96%

6.4. Táblázat: A rendszer teljesítményének összefoglalása

Nézzük meg, hogy mit is jelképeznek a fenti számértékek. A rendszer teljes pontossága



(accuracy) meghatározza a helyes találatok arányát az összes találathoz képest, a rendszer szelektív precizitása (precision) kifejezi a kiválasztott helyes találatok arányát az összes kiválasztott találatokhoz képest, míg a rendszer helyessége (recall) tükrözi a helyesen kiválasztott találatok számát az összes helyes találathoz (valós kölcsönhatás és valós kölcsönhatás hiánya) képest.

A fenti táblázatból kiderül, hogy a rendszer szelektív precizitása (precision) elérte a 82%-ot. Ez kielégítő teljesítménynek számít; ezt a eredményt negatívan befolyásolta az, hogy a MetaMap bizonyos esetekben olyan találatokat is visszatérített, amelyek valójában csak gyűjtőnevek (például „Pharmaceutical Preparations” vagy „Pharmacologic Substance”) vagy tévesen illeszkednek a forrásszövegben megjelenő kifejezésekre. A rendszer helyessége (recall) viszont rendkívül magas (96%), mivel a kölcsönhatásban álló gyógyszerek száma jóval nagyobb, mint a a kölcsönhatásban nem álló gyógyszerek száma, és az előbbi csoportba tartozó gyógyszereket a rendszer majdnem teljes mértékben megtalálta.

## 6.2. Betegségek és tünetek felismerésének tesztelése

Ebben az esetben megvizsgáltam, hogy a rendszer milyen mértékben ismerte fel a betegségeket és tüneteket. Bemeneti adatként az előző fejezetben felsorolt gyógyszereket leíró XML dokumentumokat használtam fel.

A fogalmak felismerése két forrás szerint történhet: a felhasználó meghatározhatja, hogy melyik szolgáltatást szeretné igénybe venni (DBPedia portál vagy MedlinePlus webszerviz). Mindkét szolgáltató esetében megvizsgáltam a rendszer hatékonyságát, ami abban állt, hogy megszámláltam a valós találatok és helytelen találatok számát. Az eredményeket a következő táblázatban összesítettem:

Gyógyszer	MedlinePlus		DBPedia	
	Helyes	Hiányzik	Helyes	Hiányzik
DragonTabs	20	4	17	7
Actidose	4	0	3	1
Arava	51	20	31	40
10 Person	74	7	41	41
Plavix	25	3	20	8
Warfarin	65	8	38	35
Absorica	85	45	80	50
Ambien	35	8	22	21
Naproxen	48	2	28	22
Activase	58	4	38	24

6.5. Táblázat: Betegségek és tünetek definícióinak detektált száma

Ebben az esetben a rendszer szelektív precizitását számítottam ki mind a két típusú szolgáltatásra nézve a következő képlet szerint:

$$\frac{\text{helyes találatok száma}}{\text{helyes találatok száma} + \text{hiányzó találatok száma}}$$

A következő eredményekhez juttottam:

Típus	Precizitás
MedlinePlus	82%
DBPedia	56%

6.6. Táblázat: A rendszer teljesítménye a detektált betegségek és tünetek számára nézve

A MedlinePlus online szolgáltatás esetében viszonylag sok kifejezés magyarázatát megtaláltam, a DBPedia esetében ez az érték alacsonyabb volt (56%). A rendszer egy összetett struktúrájú lekérdezést tartalmaz a betegség vagy gyógyszer nevének megtalálására, viszont a DBPedia ontológiája nem tartalmaz egy egységes struktúrát ezen kifejezések tárolására, így léteznek olyan kulcsszavak, amelyekre a lekérdezés nem ad vissza eredményt.

### 6.3. Betegségek és tünetek előfordulása

Ebben a részben azt vizsgáltam meg, hogy milyen gyakorisággal fordulnak elő a feldolgozott adatbázisban a különféle betegségek és tünetek. Ezen információ hasznos lehet egészségügyi alkalmazottak vagy gyógyszer gyártók számára, akik felmérhetik a gyógyszerpiacon forgalmazott termékek számát a betegségek és tünetek függvényében.

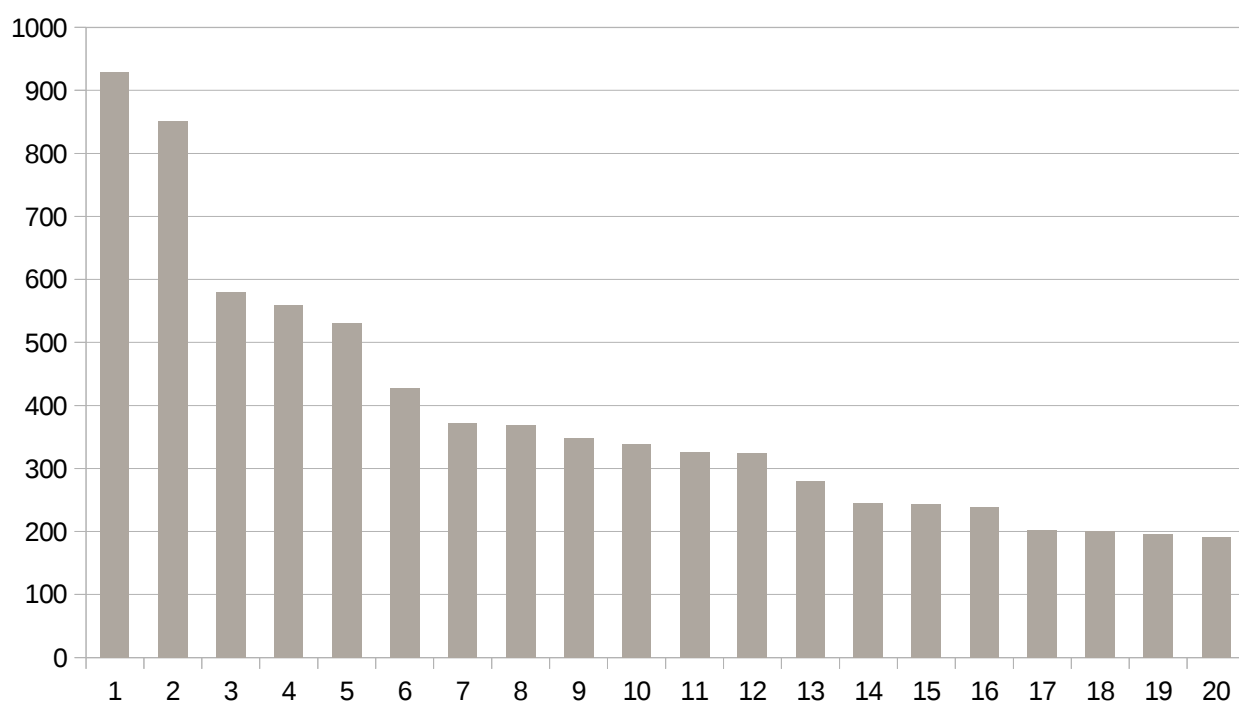
A következő diagramon feltüntettem a 20 leggyakoribb betegséget és tünetet a DailyMed 2016. januárjában kiadott gyógyszer csomagra nézve, amely 2875 darab gyógyszert tartalmazott.

A statisztikák előállításához felhasználtam a Hadoop rendszer által támogatott Pig lekérdezések futtatását. A lekérdezésben felhasználtam az XML dokumentumokból felépített MongoDB adatbázis drugs tábláját, amelynek tartalmát beolvastam a Hadoop rendszerbe, majd csoportosítottam a betegség és tünet neveket úgy, hogy minden csoport tartalmazza az összes előforduló gyógyszert, amely az illető betegségre vagy tünetre vonatkozik. Az eredményt kiírtattam a MongoDB adatbázis findings táblájába a gyógyszerelőfordulások száma szerint csökkenő sorrendben.

Sorszám	Betegség, tünet találat neve (candidate matched)	Betegség, tünet ajánlott neve (candidate preferred)	Típus (semantic type)
1	Ophthalmia	Endophthalmitis	dsyn
2	Rash	Exanthema	dsyn
3	Ill	Malaise	sosy
4	Vomiting	Vomiting	sosy
5	Nausea	Nausea	sosy
6	Swelling	Edema	patf
7	Asthma	Asthma	dsyn

8	Diabetes	Diabetes mellitus	dsyn
9	Diarrhea	Diarrhea	sosy
10	Arrest	Cardiac Arrest	dsyn
11	Dizziness	Dizziness	sosy
12	Headache	Headache	sosy
13	Heart disease	Heart disease	dsyn
14	Liver disease	Liver disease	dsyn
15	Sore Throat	Sore Throat	sosy
16	Emphysema	Pulmonary Emphysema	patf
17	Bronchitis, Chronic	Bronchitis, Chronic	dsyn
18	Bleeding	Hemorrhage	patf
19	Seizures	Seizures	sosy
20	Stroke	Cerebrovascular accident	dsyn

6.7. Táblázat: 20 leggyakoribb betegség és tünet neve a DailyMed 2016. januárjában kiadott gyógyszer adathalmazra



6.1. Ábra: 20 leggyakoribb betegség és tünet előfordulása a DailyMed 2016. januárjában kiadott gyógyszer adathalmazra

## 6.4. Forráskód minőségének tesztelése

Mivel egy összetett, rétegesen felépülő rendszer implementálását valósítottam meg, fontos szempontnak tekintettem a forráskód magas minőségi szintjének biztosítását. Ez abból a szempontból is indokolt, hogy a Javascript egy dinamikus, típus ellenőrzés nélküli programnyelv, amely esetében nem történik meg a változók típusának ellenőrzése vagy az adott függvények létezésének megvizsgálása kompilálási időben. Ezért egy magas szintű kódminőség elérése csak úgy érhető el, ha teszteseteket definiálunk, és biztosítjuk a valós kódbázis lefedettségét a tesztek által.

A kódbázis tesztelését a kliens oldali komponensen végeztem, amely a felhasználó böngészőjében fut. Az alkalmazás tesztelését 71 egységteszt (unit test) és 3 integrációs teszt valósítja meg. Nézzük néhány előnyét ezen teszteknek:

- újabb funkcionalitás bevezetése esetén, gyors visszajelzést kapunk az alkalmazás integrálásáról (megbizonyosodunk, hogy nem romlott-e el valamilyen létező funkcionalitás)
- a tesztesetek leírják az alkalmazás különböző működési elveit, dokumentálják a megírt kódot
- lehetőség van a kód lefedettségi szintjének kimutatására, amely egy jó mérték az esetleges fennálló hibák kiküszöbölésére

Az egységtesztek írását a Jasmine függvénykönyvtár segítségével valósítottam meg, amely biztosítja a tesztek rendszerezését és számos kész függvényt tartalmaz a kiemenetek, elvárások megvizsgálására. A tesztek automatizált futtatását a Karma függvénykönyvtár tette lehetővé. A tesztek esetében az AAA (Arrange-Act-Assert) tervezési mintát alkalmaztam, amely 3 lépésből áll: a bemenet meghatározása, a tesztelendő függvény meghívása, és végül a kimenet összehasonlítása a elvárt értékekkel. A kód lefedettségi szintjének meghatározását az Istanbul függvénykönyvtár biztosította, és a következő eredményeket állapította meg az alkalmazásra nézve:

Kifejezések	84.29%	118/140
Elágazások	70.59%	24/34
Függvények	72.73%	32/44
Sorok	84.29%	118/140

6.8. Táblázat: Kód lefedettségi szintjének összesítése

Az összesített értékek alapján látható, hogy átlagban minden kritérium esetében elértem a  $\frac{3}{4}$ -es lefedettségi szintet. Megjegyzem, hogy az értékek azért nem közelítik meg a 100%-os arányt, mivel nem végeztem tesztelést a szerviz osztályok esetében. Azonban a hibák előfordulási aránya valószínűtlenebb ezen esetekben, mivel a szerviz típusú osztályok nagyrészt egyszerű logikára alapoznak: például adatok lekérése a szerverről a REST csomópontok által vagy cache tár ürítése. Kritikusabb volt a controller típusú osztályok tesztelése, amelyeknek esetében biztosítottam a kód majdnem teljesfokú lefedettségi szintjét. A main modul esetében a következő eredményeket értem el (csak a kifejezések lefedettségi arányát tüntettem fel):

Modul	Fájl	Kifejezések lefedettségi aránya	Lefedett kifejezés száma / összes kifejezés száma
users	login.controller.js	100%	25/25
users	profile.controller.js	96.77%	30/31
users	register.controller.js	100%	26/26
users	user.offline.service.js	38.46%	10/26
users	user.online.service.js	83.33%	5/6
users	user.service.js	77.78%	7/9
users	users.config.js	100%	9/9
users	users.module.js	100%	2/2
users	users.run.js	66.67%	4/6
main	drug.details.controller.js	97.37%	37/38
main	history.controller.js	100%	21/21
main	main.config.js	84.62%	11/13
main	main.module.js	100%	2/2
main	main.run.js	100%	16/16
main	pageheader.controller.js	100%	20/20
main	search.controller.js	98.77%	80/81

6.9. Táblázat: Kifejezések lefedettségi szintje a users és a main modulok esetében

Az integrációs teszteket a Nightwatch függvénykönyvtárral valósítottam meg, amely felhasználja a Selenium webes meghajtót a tesztek futtatásához. Több típusú böngésző használata is lehetséges (Firefox, Chrome, Internet Explorer). Jelen esetben a teszteket a Chrome és PhantomJS böngészőkkel végeztem. A tesztek futtatása terminálból indítható el az „npm run int-test” parancs segítségével, és egy tipikus sikeres futtatás eredménye a következő volt:

```
[Login Tests] Test Suite
=====
Running: Test login and logout
✓ Element <#login> was visible after 587 milliseconds.
✓ Testing if element <#search> is present.
✓ Testing if element <.navbar .navbar-right a> is present.
✓ Testing if element <#login> is present.
OK. 4 assertions passed. (4.91s)

[Search Tests] Test Suite
=====
Running: Login
✓ Element <#login> was visible after 1115 milliseconds.
OK. 1 assertions passed. (3.757s)

Running: Search for DragonTabs and check if results are found
✓ Element <#search> was visible after 58 milliseconds.
✓ Element <#search table> was visible after 575 milliseconds.
OK. 2 assertions passed. (890ms)

OK. 7 total assertions passed. (9.623s)
zoliqa@zoliqa-pc:~/Documents/drugsdb/app$ npm run int-test
```

6.2. Ábra: Integrációs tesztek futtatásának eredménye

## 6.5. Következtetések és továbbfejlesztési lehetőségek

A munkának a célja az volt, hogy gyógyászati kifejezéseket ismerjünk fel struktúrálatlan szövegben, és ezen belül kimutassuk a gyógyszerek közötti kölcsönhatásokat egy olyan online rendszer segítségével, amely könnyen elérhető bármely számítógépes eszközről, legyen az személyi számítógép, táblagép vagy telefon. A bemeneti adatokat a DailyMed portál által nyilvánosan közzétett XML dokumentumok szolgáltatták, amelyeket a rendszer rendre feldolgoz és adatbázisban tárol el. Az adatfájlok feldolgozása több lépésben történt, amelynek végrehajtásához több különböző funkcionalitású rendszert kapcsoltam össze.

A Node.js platform v5.11.1-es verziója biztosította elsősorban a teljes rendszer működését és irányítását, másodsorban ez foglalja magában a rendszer web szerveren történő futtatását, a REST hívások kiszolgálását, stb.

Az alkalmazás fejlesztését és tesztelését az Ubuntu 14.04-es Linux disztribúción végeztem, amely megfelelőnek és alkalmasnak bizonyult a felhasznált rendszerek integrálásának céljából. Fontos szempont volt egy Linux rendszer használata, mivel a Hadoop keretrendszer csak ezt a platformot támogatja, de előnyös volt a Node.js platform szempontjából is, amely könnyű hozzáférhetőséget biztosított az operációs rendszer különböző programjainak futtatásához, a fájlrendszer kezeléséhez valamint a MetaMap program és Pig program futtatásához.

Felhasználtam a Hadoop 2.6.4 csomagját ahhoz, hogy egy egycsomópontos Hadoop rendszert telepítsek fel az operációs rendszerre. Ez által lehetővé vált a Pig lekérdezések futtatása, amely felhasználja a telepített HDFS rendszert az adatok feldolgozásához és a szkriptfájlok végrehajtásához. A Pig szkriptek futtatásához szükség volt különböző meghajtókra, amelyek segítségével hozzáférhettem a lokális MongoDB adatbázisban tárolt táblákhoz. Ezen meghajtók Java nyelvben megírt JAR típusú fájlok formájában érhetők el. Mivel a MongoDB adatbázisból beolvasott gyógyszereket tároló táblának tömb típusú oszlopa is volt, ezért szükség volt egy külső függvényre, amely a tömb típusú adatstruktúrát átalakítja ún. bag típusú struktúrára. Ennek a külső függvénynek (UDF – user defined function) a segítségével átalakítottam a beolvasott a tömb struktúrát, és meghatároztam a típusát a Pig számára ahhoz, hogy tovább tudja kezelni annak elemeit. Pig szkriptek segítségével sikerült új táblákat létrehozni az adatbázisban, amelyek gyors hozzáférést biztosítanak a gyógyszerekhez különböző keresési feltételek szerint (gyártó, alapanyag). Az előző fejezetben ismertetett statisztikák előállításához szintén egy Pig szkriptet használtam fel, amely rendszerezte az adatokat az előállítandó struktúrák szerint.

A MongoDB adatbázis 3.2.6 verzióját alkalmaztam az adatok tárolására, amely megfelelő választásnak bizonyult a Hadoop rendszerrel történő integráláshoz, és könnyen kezelhető a Node.js keretrendszer által, mivel mindkettő interfésze támogatja a Javascript nyelvet.

A MetaMap 14-es verziója biztosította a struktúrálatlan szöveg elemzését, illetve a gyógyászati kifejezések és gyógyszernevek felismerését. Az alkalmazás adatfeldolgozó Node.js modulja az operációs rendszer által biztosított parancsok segítségével futtatta a MetaMap eszközt, amelynek eredménye minden egyes végrehajtás esetén egy XML dokumentum volt. Ez tartalmazta a felismert kulcsszavakat és kifejezéseket, amelyeket az alkalmazás az adatbázisban tárolt el a webes interfészen keresztül történő megjelenítésre. Meg kell jegyeznem, hogy egy időben csak egyetlen MetaMap végrehajtást tudtam futtatni, mivel párhuzamos hívások esetén hibák álltak elő a feldolgozás alatt. Ez bizonyos szinten hátrányt jelent, viszont a feldolgozási lépés mindenképpen offline módon történik, így ez nem befolyásolja a rendszer használatát. A MetaMap ígéretes választásnak bizonyult, mivel jó eredményt értem el a gyógyszerkölcsönhatások felismerése során (a rendszer helyessége 96%-os volt). A rendszer szelektív precízitása 82%-os volt, amely értéket negatívan befolyásolta a MetaMap által visszatérített fölösleges találatok (nem voltak relevánsak a

bemeneti szövegre nézve). Ezen a teljesítményen javítani lehetne úgy, hogy egy fekete lista segítségével eltárolnánk azon kifejezéseket, amelyeket nem kívánunk figyelembe venni az eredmények megjelenítésénél.

A külső szemantikus kereső szolgáltatás használatának precizitása csak 56%-os volt a DBPedia esetén, ami azzal magyarázható, hogy nem volt teljes a SPARQL lekérdezés, nem foglalta magában az összes RDF kapcsolatokat, amelyek betegségeket és tüneteket írnak le. Ezen javítani lehet, ha finomítjuk a lekérdezést úgy, hogy több releváns RDF kapcsolatot érintsen, amely a betegségek és tünetek találatait tartalmazza. A MedlinePlus web szolgáltatás elég jó eredményt biztosított, viszont megszorításokat is tartalmaz, amely meggátolja az alkalmazás kereskedelmi használatát, mivel egy korlátot szab a lekérdezések számára egy bizonyos időintervallumon belül. Lehetséges kiegészíteni az alkalmazást újabb keresőmotorok által, amelyek más adatbázist vagy szolgáltatást használnak a találatok meghatározására.

Az alkalmazás webes interfésze az Angular.js keretrendszerben íródott, amely egy moduláris struktúra kialakítását tette lehetővé. Így a rendszer könnyen karbantartható, és hatékony továbbfejlesztési lehetőségeket biztosít. Több külső Javascript könyvtárt használtam fel az alkalmazás kliens oldali (Require.js, Bootstrap, jQuery, stb.) illetve szerver oldali részének implementálásához (Grunt.js, Express.js, xpath, stb.), amelyek mind egy modern fejlesztési környezetet és kódstruktúra kialakítását valósították meg. Továbbfejlesztési lehetőség lehet az alkalmazás offline módbeli támogatása, amely által a felhasználó akár Internet kapcsolat nélkül is elérheti a keresések előzményeit vagy naplót vezethet a használt gyógyszerek listájáról.

A kliens alkalmazás kódbázisának átfogó tesztelése biztosította a magasszintű kódminőség fenntartását: 71 egységteszt és 3 integrációs teszt íródott a böngészőalkalmazás esetében, amelyek elősegítik az alkalmazás funkcionalitásának dokumentálását, és biztosítják a karbantarthatóságot. A kód  $\frac{3}{4}$ -ének lefedése biztosítja az alkalmazás megbízhatóságát és csökkenti az esetleges hibák megjelenését. Kiegészíthetők a tesztesetek a szerver oldali rész bevonásával, amely ugyancsak növelné az alkalmazás minőségét és dokumentálná a szerver oldali komponensek működési elvét.

A rendszer megvalósítása során egy átfogó képet kaptam a különböző funkcionális rendszerekről, amelyeket mind a nagy mennyiségű adat kezelésére használnak. Rájöttem, hogy a Linux operációs rendszer rendkívül hasznos, és elősegíti ezen rendszerek között az egyszerű adatcserét és adatfeldolgozást. Az általam használt technológiák közös tulajdonsága, hogy mind vagy nyílt forráskódú vagy szabadon felhasználható termékek, amelyek mind aktív közösséggel rendelkeznek (egyszerű volt megoldást találni a fennálló problémákra, mivel nagy számú technikai cikk, könyv vagy fórum áll a fejlesztő rendelkezésére). A természetes nyelven írt szövegek (jelen munka esetében orvosi szövegek) teljes gépi feldolgozása még nem elég kiforrott, viszont léteznek eszközök, amelyekkel elég jó eredményeket érhetünk el. A webfejlesztői eszközök szempontjából, megbizonyosodtam, hogy a Javascript nyelv rendkívül sokat fejlődött, és a megfelelő könyvtárakkal nagyon gyorsan megbízható, stabil kód írható, amely a fejlesztett terméket alkotó rétegek minden szintjén támogatott (kliens – frontend, szerver, adatbázis vagy akár a Hadoop rendszer által támogatott Javascript UDF-ek).

## 7. Irodalomjegyzék

- [1] Stockley, I. H., Stockley's Drug Interaction, Pharmaceutical Press, 2007
- [2] Thomayant Prueksaritanont, Xiaoyan Chu, Christopher Gibson, Donghui Cui, Ka Lai Yee, Jeanine Ballard, Tamara Cabalu, and Jerome Hochman, Drug–Drug Interaction Studies:

[3] R. Kleinsorge, J. Willis, Unified Medical Language System (UMLS) Basics, National Library of Medicine 1–157 <[https://www.nlm.nih.gov/research/umls/pdf/UMLS\\_Basics.pdf](https://www.nlm.nih.gov/research/umls/pdf/UMLS_Basics.pdf)>

[4] <[https://metamap.nlm.nih.gov/Docs/SemanticTypes\\_2013AA.txt](https://metamap.nlm.nih.gov/Docs/SemanticTypes_2013AA.txt)>

[5] Chapman, W.W., Bridewell, W., Hanbury, P., Cooper, G.F., Buchanan, B.G., A Simple Algorithm for Identifying Negated Findings and Diseases in Discharge Summaries. Journal of Biomedical Informatics, 2001

[6] Lindberg DA, Humphreys BL, McCray AT, The Unified Medical Language System, Methods Inf Med., 1993

[7] Mutalik PG, Deshpande A, Nadkarni PM, Use of general-purpose negation detection to augment concept indexing of medical documents: a quantitative study using the UMLS, J Am Med Inform Assoc., 2001

[8] Brian Hazlehurst, H. Robert Frost, Dean F. Sittig, Victor J. Stevens, MediClass: A System for Detecting and Classifying Encounter-based Clinical Events in Any Electronic Medical Record. J Am Med. Inform. Assoc., 2005

[9] B Bokharaeian, A Díaz Esteban, M Ballesteros Martínez, Extracting Drug-Drug interaction from text using negation features, Sociedad Española para el Procesamiento del Lenguaje Natural, 2013

[10] Segura-Bedmar, I., P. Martinez y C. de Pablo Sanchez, Using a shallow linguistic kernel for drug-drug interaction extraction, Journal of Biomedical Informatics, 2011

[11] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, David McClosky, The Stanford CoreNLP Natural Language Processing Toolkit

[12] Alan R Aronson, Francois-Michel Lang, An overview of MetaMap: historical perspective and recent advances, May 4, 2010

[13] J. Dean and S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, in Proceedings of OSDI '04: 6th Symposium on Operating System Design and Implementation, San Francisco, CA, Dec. 2004

[14] Auer, S., Bizer, C., Lehmann, J., Kobilarov, G., Cyganiak, R., Ives, Z.: DBpedia: A Nucleus for a Web of Open Data, The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 ASWC 2007, Busan, Korea, November 11-15, 2007

[15] Roy Fielding, Architectural Styles and the Design of Network-based Software Architectures, University of California, Irvine, 2000

[16] <<https://scholarsbank.uoregon.edu/xmlui/bitstream/handle/1794/7817/2005-grater.pdf?sequence=1>>

[17] <<https://github.com/karma-runner/karma/blob/master/thesis.pdf>>

[18] <<http://www.ecma-international.org/publications/standards/Ecma-262.htm>>

[19] <[http://exploringjs.com/es6/ch\\_promises.html](http://exploringjs.com/es6/ch_promises.html)>

[20] Stefan Kimak, The role of HTML5 IndexedDB, the past, present and future, 2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)

[21] Tom White, Hadoop: The Definitive Guide, 4th Edition, O'Reilly Media, 2015



- [22] Bob DuCharme, Learning SPARQL, 2nd Edition, O'Reilly Media, 2013
- [23] Kristina Chodorow, MongoDB: The Definitive Guide, 2nd Edition, O'Reilly Media, 2013
- [24] Jim Wilson, Node.js the Right Way, Pragmatic Bookshelf , 2013
- [25] Alan Gates, Programming Pig, O'Reilly Media, 2011
- [26] Robert C. Martin, Agile Principles, Patterns, and Practices in C#, Prentice Hall; 2006