Szegedi Tudományegyetem
Informatikai Tanszékcsoport

Kimutatások készítése NoSQL alapon orvosi döntéstámogató rendszerhez

Szakdolgozat

Készítette:
Hajas Tamás
gazdaságinformatikus BSc
szakos hallgató

Témavezető:
Dr. Bilicki Vilmos
egyetemi adjunktus

Szeged
2015

Feladatkiírás

A feladat egy orvosi költségelszámoló és döntéstámogató szoftver részegységének fejlesztése, ami a Szoftverfejlesztés Tanszéken készül a Szegedi Tudományegyetem Általános Orvostudományi kar Egészségbiztosítási Igazgatóság részére. A feladat része a probléma pontos megismerése, a lehetséges megoldási módszerek kutatása és maga a megvalósítás. A technológiai eszközkészlet a manapság egyik legelterjedtebb JavaScript környezetből, NodeJS és AngularJS-ből, valamint MongoDB-ből és az ezekhez tartozó modulokból áll. A részegység több összetett lekérdezésből és adatbázis műveletből áll, amelyet informatív grafikonokon jelenítünk meg a felhasználói oldalon. A feladat részét képezte egy olyan NoSQL adatbázis-modell kialakítása, amely megfelelő alapot biztosít a későbbiekben fejlesztésre kerülő vezetői információs rendszer kiszolgálására.

Tartalmi összefoglaló

- ***A téma megnevezése:***
  Kimutatások készítése NoSQL alapon orvosi döntéstámogató rendszerhez

- ***A megadott feladat megfogalmazása:***
  Adatbázis műveletek, több, összetett lekérdezés írása és az eredmények megjelenítése grafikonon

- ***A megoldási mód:***
  Szerver és kliens oldal közötti megfelelő adatok átküldésével, amik a map-elési módszerrel lettek kinyerve, a Highcharts grafikon készítő könyvtárral kirajzolásra kerül

- ***Alkalmazott eszközök, módszerek:***
  AngularJS, Node.js, MongoDB, Mongoose, HashMap, WebStrom fejlesztői környezet, Scrum módszer

- ***Elért eredmények:***
  A létrehozott adatbázis sémák alapján, a CSV fájlok feltöltése után a helyes táblák létrejönnek. Megfelelő tábla feltöltésekor létrejön az összegző táblázat, mely adatainak felhasználásával havi szinten kimutatható az osztályok közötti pénzmozgás a grafikonokon

- ***Kulcsszavak:***
  AngularJS, Node.js, MongoDB, Highcharts, HashMap, gazdasági lekérdezés

Tartalomjegyzék

BEVEZETÉS

Egyetemi tanulmányaim végén a Szegedi Tudományegyetem Szoftverfejlesztés Tanszékre mentem céltudatosan, hogy itt szeretném szakdolgozatomat megírni. Itt kerültem Bilicki Vilmos M2M nevű szoftverfejlesztő csapatába, ahol az első hónapok betanulási időszaka után egy AngularJS keretrendszert használó projektbe kerültem.

A projekt neve EBIG – Orvosi Gazdasági Döntéstámogatás, a célja pedig a Szegedi Tudományegyetem Általános Orvostudományi kar Egészségbiztosítási Igazgatóság (EBIG) részére egy olyan szoftvert létrehozni, ami a meglévő MS Access alapú rendszert váltja le. A fő probléma, ami miatt ez a szoftver szükséges, az, hogy az igazgatóságról elment a munkatárs, aki értette a rendszer működését. A rendszer megléte és hibátlan futása azért szükséges, mert a klinika teljes belső elszámolását ezen keresztül monitorozzák (konzíliumok, diagnosztika, labor), valamint éves szinten több milliárd forintnyi elszámolás kerül az OEP felé.

A szűrés alapja az, hogy egy adott klinika nem küldhet át egyik osztályból a másikba pénzt konzíliumért, mert egy az intézetvezető, tehát egy intézetbe is tartozik a két osztály. Ezek a pénzmozgások viszont a medsol-ban (klinikai nyilvántartó rendszer) megjelennek, de nem számít az elszámolásba.

A feladat jelenleg összetett, medsol-ból exportált adatok szűrése, betöltése az Access adatbázisba, ott további ellenőrzés és plusz adatok felvétele és kimutatások készítése. A sok lépés sok hibát is eredményezhet, pláne hogy kívülről is belenyúlnak időnként a rendszerbe, ahol a rendszerben lévő statikus adattáblákat változtatják, OEP elszámolási tételek változása miatt. Ez egy mindig fennálló probléma lehetőség, de jelenleg nincs verziózva hogy ki, mikor és mit változtatott, nincsenek kezelve az esetleges adatütközések vagy duplikációk. Erre szeretnénk megoldást találni egy közös rendszerrel, amely kiváltja a kézi adatszűrést és tisztítást és egy rendszeren belül kezeli a vezetői információkat, kimutatásokat, kézi adatfeltöltést. Mindezt úgy, hogy minden esemény felhasználóhoz és időhöz kötött, így visszakövethető hogy ki mikor és mit változtatott.

Személyes feladataim között voltak táblázat sémák létrehozása, összegző táblázat létrehozó logika kialakítása és a már összeállított táblázatokból történő gazdasági lekérdezések írása, továbbá a lekérdezésekből kapott eredmények megjelenítése.

1. Agilis fejlesztési módszerek

Szakdolgozatomban projekt munkán dolgoztam, egy csapat részeként. A csapat létszáma kisebb volt, a feladat jól szeparálható és egy viszonylag hosszabb projekt határidő állt rendelkezésünkre, így minden adott volt egy jól működő Scrum-hoz. Ezzel a módszerrel jól ellenőrizhetővé vált az összes tag munkája és hatékonyabbá, gyorsabbá vált a fejlesztés.

1.1. Scrum

A Scrum egy módszertan, amely magában foglal bizonyos tevékenységeket és meghatározott szerepeket.

Takeuchi Hirotaka és Nonaka Ikujiro 1986-ban leírtak egy módszert, ami gyorsabbá és rugalmasabbá teszi a fejlesztést. Váltófutáshoz hasonlítják a vízesés modellt, ahol egyszerre csak egy futó fut és egymásnak adják a stafétát. Az új módszerre úgy tekintenek, mint a rögbire, ahol az egész csapat fut és passzolgatják a labdát.

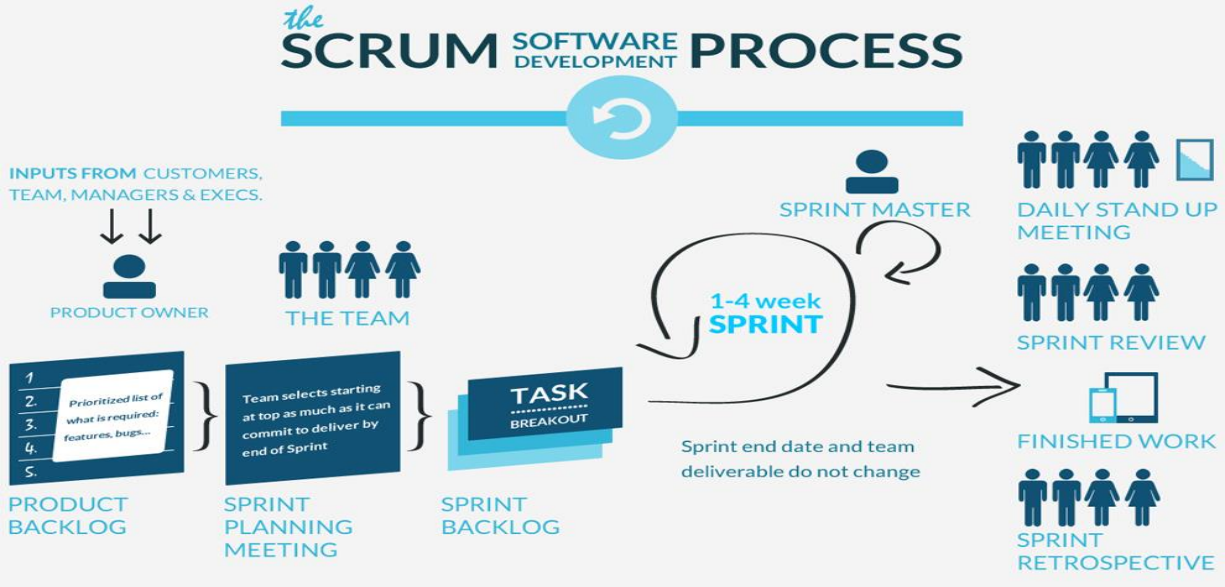
A scrum alapelvei közé tartozik:

- A csapat kis létszámú, önjáró és keresztfunkcionális, tehát nem kell másokra várni bizonyos technikák elvégzésére, hanem van a csapatban minden feladatra hozzáértő személy
- A nagy feladatokat próbáljuk meg a lehető legjobban felszeletelni, minél kisebb részekre
- A rendelkezésre álló időt osszuk fel kisebb részekre (helyzetünkben ez 2-2 hét volt)
- A csapatnak saját magának kell megbecsülnie, mennyi idő alatt képest elvégezni a feladatot
- A csapat folyamatosan monitorozza a tevékenységét

További scrum kulcsszavak:

- Sprint – Futam, iteráció, ameddig egy-egy fejlesztési hullám tart, 2 és 4 hét közötti időtartamot jelent
- Story – elvégzendő feladat
- Product backlog – PO által felvitt story-k összessége
- Sprint backlog – sprint során elvállalt story-k összessége
- Burndown chart – ennek segítségével lekövethető a csapat produktivitása, két fajtája van, az egyik az aktuális sprinthez tartozik, míg a másik a teljes projekthez

Az 1.1. ábrán a scrum módszer fejlesztési folyamatának mérföldköveit és a feladatokhoz tartozó tagokat lehet megtekinteni.



1.1. – Scrum fejlesztési folyamat [21]

1.1.2. Scrum szerepkörök

Role	Ceremony	Document
PO	Sprint planning	Product backlog
SM	Daily stand-up	Sprint backlog
Team	Sprint review	Burndown chart

1.2. – Szerepkörök táblázata

- Product Owner (PO) – Ő adja át a megrendelő ötleteit a csapatnak, kapcsolat fenntartó a két fél között.

  Minden fejlesztési igény hozzá fut be, majd ő ezek után felméri és priorizálja azokat. Az ő feladata összeállítani a product backlogot, amibe más nem nyúlhat bele, legfeljebb megjegyzéseket tehet.
- Scrum Master (SM) – A csapat érdekeit szem előtt tartó csapattag, aki azért dolgozik, hogy a csapat zökkenőmentesen tudjon haladni. Ügyel arra, hogy a scrum folyamatot megfelelően alkalmazza a csapat és annak szabályait be is tartsák. A ceremóniákat vezeti, és azok ütemterveit betarttatja.
- Csapat (Team) – Ők felelősek a munkák elkészüléséért. Feladatkiosztásnál figyelni kell a keresztfunkcionalitás fejlődésére, tehát minden tagot ki kell emelni a saját

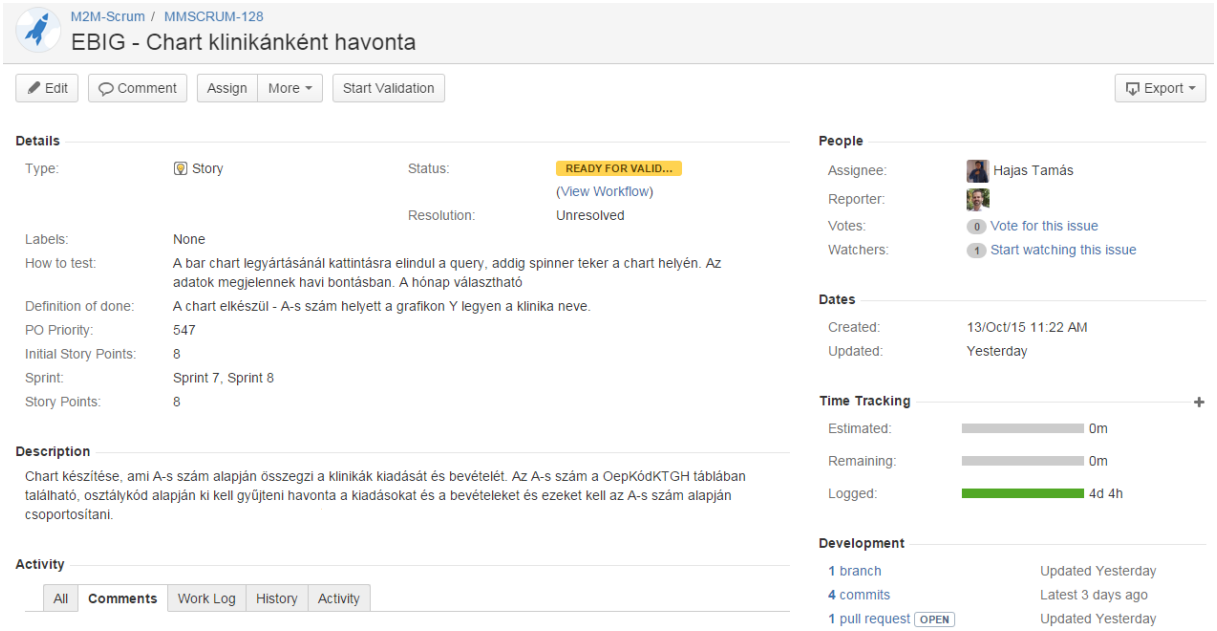
komfort zónájából és olyan feladatokban is részt kell venniük, amiben más jó, de ezt a sprint elején be kell kalkulálni.

Az 1.2. ábrán láthatóak a szerepek és azok feladatai.

1.1.3. Story felépítése

- Rövid, tömör cím (a példában: EBIG – Chart klinikánként havonta)
- Leírás (story hosszabb kifejtése)
- Hogyan teszteljük
- DOD (definition of done) mire van szükség ahhoz, hogy a storyt elfogadjuk
- Fontosság (PO Priority) adott munka prioritása, nem lehet két egyforma prioritású story. 1-től végtelenig mehet a pontozás és érdemes nagy léptéket használni, mellékelt képen 547, hogy az új feladatokat is tudja a PO értékelni.
- Kezdeti becslés, PO teszi ezt a becslést, hogy szerinte mennyi story pontot ér az adott story
- Későbbi értékelés, amit már a csapat együtt tesz
- Tényleges ráfordított idő, hány órát foglalkozott a csapattag a feladattal, a későbbi becslésekhez fontos ez az adat

A story-k életútja az 1.1.6. pontban lesz kifejtve. Az 1.3. ábrán egy story példa látható a projektből.



1.3. – Egy story

1.1.4. Megbeszélések

- Scrum meeting/Stand-up:

  Minden nap egy pár perces (5-10 perc), nyílt, rövid beszélgetést tartanak a csapattagok a Scrum Master levezetésével. Három alapvető kérdésre kell az összes fejlesztőnek válaszolnia:

  - Mit csináltál a tegnapi megbeszélés óta?
  - Mit fogsz csinálni a következő megbeszélésig?
  - Van-e valami, ami blokkol, akadályoz a munkádban?

- Sprint planning:

  Minden új sprint/futam előtt futamtervező megbeszélést tartanak, ezzel megtervezésre kerül a következő sprint. Feltétele a megfelelő számú storyt tartalmazó backlog, ami elég a következő sprinthez. Mindig a következő sprinthez kell csak tervezni. Tervezéskor jelen van a PO, a SM és a Team is.

  A PO azért, mert ő tisztázza a csapat számára az összes storyt. A SM vezeti a megbeszélést. A TEAM pedig azért jön, hogy minden fontos információt megtudjon a következő sprintről. Mindig a PO kezd, bemutatja, hogy melyik story mit takar. Bemutatás után a Team kérdezhet.

- Demo:

  Ezen az áttekintésen azzal foglalkoznak, hogy melyik feladatok készültek el, és melyikek nem. Az elkészült munkát a terméktulajdonosnak és a fejlesztőknek mutatja be a készítője (Demo). A demót követi a retrospective, tehát a visszatekintés.

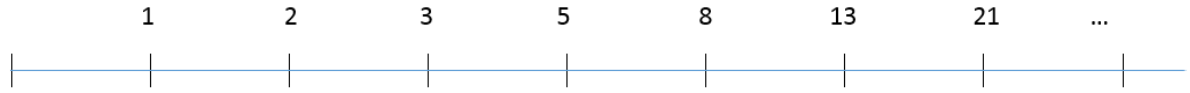
- Retrospective (Visszatekintés):

  A csapattagok véleményt alkotnak az előző sprintről, összegyűjtik a pozitív és negatív véleményeket azzal kapcsolatban. Javaslatokat tesznek, hogyan lehetne a következő fordulót javítani.

- Planning coffe:

  Ez egy extra megbeszélés, a csapattagoknak nem kötelező részt venni rajta, opcionális. Ebéd után szokták tartani és szigorúan 30 perc. A nap során felmerült kérdéseket lehet megbeszélni itt. Érdemes egy dokumentumot vezetni arról, mi lesz a téma, hogy akit érdekel, az részt tudjon venni rajta [20].

1.1.5. Story Pontok



1.4. – Story pontok

A story pontok a Fibonacci számok szerint alakulnak.

Becslésük úgy történik, hogy az előző fordulókból vesz példát a csapat, és egy számegyenesre teszik, ezzel segítve az újabb feladatok mérlegelését. Ez a számegyenes az 1.4. ábrán látható.

Két extra jelzés áll még a felek rendelkezésére, a kérdőjel (?) és a felkiáltójel (!).
A kérdőjel akkor kell, ha nem lehet felbecsülni a feladatot, hiányosnak véli a csapat.
A felkiáltójel pedig azt jelzi, hogy felbecsülhetetlenül nagy a feladat, kisebb részekre kell bontani, vagy nincs meg a kellő kompetencia.

A pontok kiosztása egy úgynevezett Planning poker eseményen történik, ennek a lényege az, hogy minden csapattag kap értékekkel ellátott kártyákat, majd egy kis gondolkozási idő után egyszerre felmutatnak egy értéket. Ha a csapattagok pontjai eltérőek, akkor a Scrum Master vezénylésében átbeszélik, miért nincs egyetértés. Az átbeszélés után újra szavaznak a pontról.

A pontok kiosztása után meg kell határozni, hogy a csapat az adott fordulón mennyi feladatot tud elvégezni. Ezek után a prioritási sorrend alapján kiválasztásra kerülnek azok a feladatok, amik bekerülnek a sprint backlog-ba és a csapat jóváhagyja ezt, itt még lehetőség van vétózni, vagy kérdezni, de az elfogadás után nincs lehetőség módosításra.

1.1.6. Story életútja

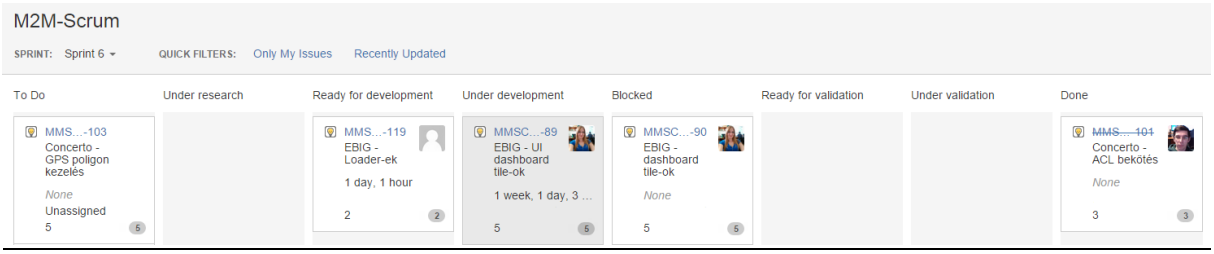
A JIRA rendszert használtuk a megkapott feladatok (story) életútjának kezelésére.
8 állapot fordulhat elő egy story elkezdésétől a befejezésig:

- To Do: minden új feladat innen indul, innen választanak a csapattagok, mindig a legmagasabb prioritásút kell választani, ha végzett a csapattag az előző feladatával, ezt az elvet csak a Scrum Master másíthatja meg
- Under research: Egy feladat akkor kerül ebbe az állapotba, ha kutató munkát igényel, vagy teljes egészében kutatásról szól
- Ready for development: Feladatok, amelyek készen állnak a fejlesztésre, főleg akkor kerülhet ide a story, ha a kutatómunka befejeződött, ledokumentálták azt, de a fejlesztés később következik

- Under development: Amikor dolgoznak a feladaton, akkor kerül ebbe az állapotba
- Blocked: Ha valamilyen oknál fogva a fejlesztő nem tudja folytatni a munkáját, akkor a feladat ebbe az állapotba kerül, illik ilyenkor kommenttel ellátni, hogy mások is tudják, miért akadt el
- Ready for validation: Ez egy jelzés a tesztelők felé, ami azt jelenti, hogy a fejlesztő szerint készen van a feladat, ellenőrizhető állapotban van
- Under validation: Amint elkezdődik a story ellenőrzése, akkor kerül ebbe az állapotba
- Done: A story elkészül, megfelel a leírtaknak, teljesül a Definition of Done.

Sprintforduló után a fejlesztőhöz kiosztott feladatok Ready for development státuszba kerülnek, innentől kezdve a hozzárendelt fejlesztő kezeli az életútját a Done-on kívül végig.

Az 1.5. ábrán 5 feladat látható különböző állapotokban.



1.5. Story-k életútja

1.2. Atlassian Jira

A JIRA egy feladat és projektkezelő rendszer, egy issue-tracker. Az Atlassian cég eredetileg hibajegyek és ügyfélpanaszok kezelésére találta ki, de az elmúlt több, mint egy évtizedben annyira kinőtték magukat, hogy mára már több saját fejlesztésű szoftverük és plugin-jeik is létezik, amiket széles körben használnak is. Mint például: hiba- és eseménykezelés, projekt folyamat követés (mi erre használtuk), elemzések, tesztelés menedzsment.

Atlassian termékek:

- JIRA: feladat és projektkezelő rendszer
- Confluence: alap rendszer, ami a belső kommunikációt hivatott erősíteni, dokumentum kezelő
- Stash: GIT repository feladatokat lát el

  Csapatunk ezt a három szoftvert használta a fejlesztés során [3].

1.3. Verziókezelők

Akkor beszélünk verziókezelésről, amikor egy adat több verzióval rendelkezik, és azokat kezeljük. A szoftverfejlesztésben a fejlesztés alatt álló forráskódok verzióinak kezelésére

használható, amiken több ember dolgozik egyszerre, egyidejűleg. A különböző változtatásokat verziószámokkal és betűkkel követik nyomon.

A hagyományos verziókezelők központosított modellel dolgoznak, ahol minden művelet egy közös szerveren történik. Ha két fejlesztő egyazon időben próbál meg változtatni egy fájlt, akkor el kell kerülni, hogy a két fejlesztő felülírja egymás munkáját. Az ilyen centralizált rendszerek kétféleképpen hidalják át ezt a problémát, vagy zárolással, vagy összefűzéssel.

- Zárolás (lock): A konkurens hozzáférés kezelésének legegyszerűbb módja, ha megtiltjuk azt, tehát ha valaki elkezd módosítani egy fájlt, akkor azt más felhasználó nem nyithatja meg szerkesztésre. Ezt hívják lock-olásnak, a magyarosabb zárolás helyett, az angol elnevezés elterjedtebb, így ez az általános elnevezése ennek a műveletnek. Ha egy felhasználó kivesz (checkout-ol) egy fájlt, akkor a többi felhasználó már csak olvashatja azt, amíg az első vissza nem teszi, vagy el nem veti a módosítását.

  Ez a megközelítés előnyökkel és hátrányokkal is jár. Nagyobb, több fájlt érintő változtatásnál célszerűbb ezt választani, mert ezzel bonyolult merge-elési műveleteket lehet megtakarítani. Viszont, ha a változtatás túl sokáig tart, ezzel a fájl túl sokáig lesz lezárt állapotban, akkor a többi fejlesztő csak a lokális fájljait módosíthatja, ami nagy problémákhoz vezethet.
- Összefésülés (merge): Ebben az esetben is az angol szóhasználat az elterjedtebb. A legtöbb verziókezelővel lehetséges, hogy több felhasználó dolgozzon egyidejűleg egy fájlon. Ilyenkor, aki először tölti vissza a változtatásait, az mindenképp sikerrel jár. A többi felhasználó lehetőséget kap az összefésülésre, a felülírást ezzel kiküszöbölve, mellyel a különböző változtatások összeolvasztásra kerülnek. Kézzel és automatikusan is történhet a merge [23].

1.3.1. Nyitott rendszerek

A nyitott, elosztott verziókezelők támogatják különböző branch-ek létezését és erősen függnek a merge művelettől. Ilyen rendszerekben minden munkamásolat egy ág, ezek összefésülése patch-ek küldésével történik. Választhatunk az egyes változtatások között és új tagok bármikor csatlakozhatnak a rendszerhez. Szabadon használható, nyitott verziókezelő rendszer például a Git és a Mercurial [23].

1.3.2. Verziókezelőknél használt fogalmak

- Baseline: fájl jóváhagyott verziója, ehhez viszonyítják a következő változásokat
- Check-out: lokális másolat készítése egy veziókezelt fájlról. Általában a legfrissebb verziót kapja a felhasználó, de lehetőség van konkrét verzió kikérésére is.
- Check-in, Commit: Művelet, amivel a lokális példány változatásai bekerülnek a szerveren tároltba.
- Conflict: Ha ketten akarnak megváltoztatni egy fájlt és a rendszer nem tudja merge-elni a változtatásokat. A felhasználóknak ekkor fel kell oldania a konfliktust, ami vagy a változtatások összekombinálása, vagy csak az egyik változtatást választja és használja.
- Repository: Az a hely, általában szerver, ahol az aktuális és a korábbi verziók tárolódnak [23]

1.3.3. Branch

Másik nevén ág. Verziókezelt fájlok egy részhalmaza elágazhat, ilyenkor egyidejűleg több verziójuk is létezhet. Ezeket külön csapatok, külön gyorsasággal, de még különböző irányokban is fejleszthetik.

Amikor valaki elkezd új story-n dolgozni, akkor egy branch-et kellett létrehoznia. Ezzel egy saját hibajavítást (Bugfix), új funkciót (Feature), gyorsjavítást (Hotfix), vagy végső megjelenést (Release) kezd el fejleszteni. Stash-ben kiválasztható, hogy melyik meglévő ágból szeretne a fejlesztő saját branch ágat létrehozni a „Branch from” opcióval.

Amikor elkészül egy story, akkor a fejlesztő egy pull request-et küld, hogy a munkája a Development ágba bekerülhessen. Pull request a Stash-ben hozható létre. Meg kell itt adni, hogy honnan, melyik branchből, hova, melyik branchbe szeretnénk átmozgatni a kódot. Ezt a kérést nálunk a Scrum Master fogadta el, vagy mozgatta vissza további fejlesztésre, de a legtöbb esetben a csapattagok egymás kódját ellenőrzik és fogadhatják el.

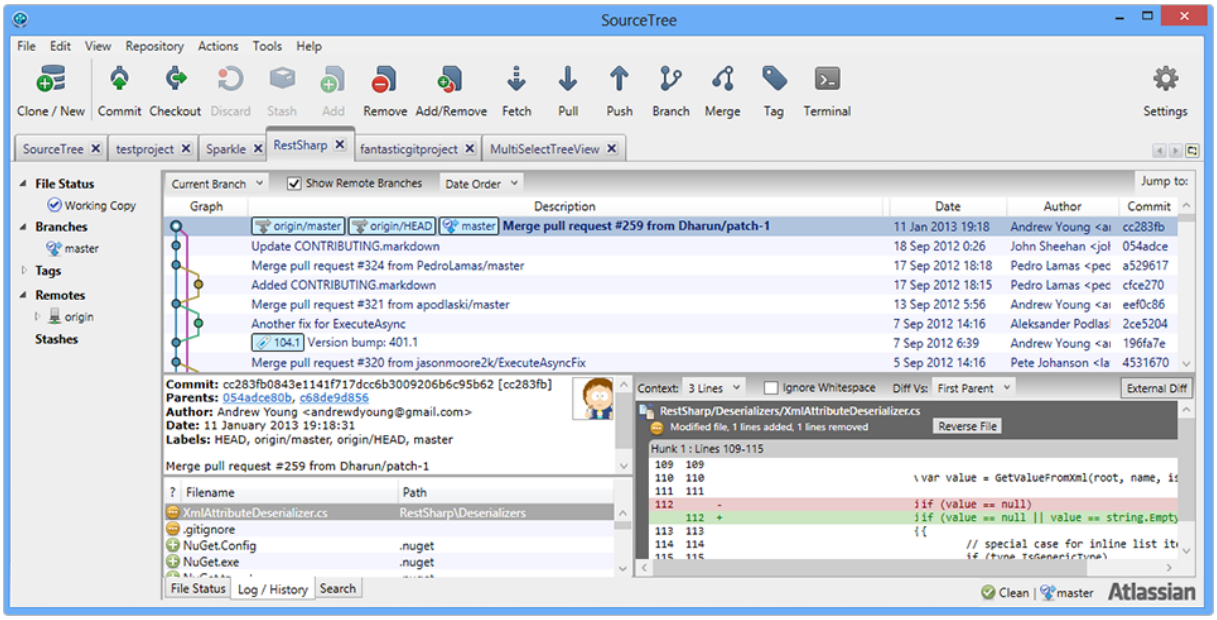
A csapattagok különböző branch-eit és azok változásait a végső változatig verziókezelő szoftverrel tudjuk nyomon követni. A mi csapatunk a SoureTree nevű szoftvert használta erre a feladatra. Egy branch lokálisan elérhetővé válik, ha a Stash-ben a „Check out in SourceTree” opciót választjuk. Ezután felugrik egy ablak, ahol klónozzuk a számítógépre lokálisan a repository-t [23].

1.3.4. Atlassian SourceTree

A SourceTree egy Atlassian Windows és Mac alatt futtatható, asztali alkalmazás, ami egy ingyenes Git és Mercurial kliens, egyszerű, könnyen kezelhető felülettel.

A megfigyelt branchek minden hivatalos változását figyelemmel követhetjük benne. A felületen az első opció a git gráf, itt lehet nyomon követni a branchek mozgását a development ágban. Ezt követi a leírás, amit a fejlesztő ad, rákattintva a változtatott fájlokat nézhetjük meg, és magukat a változásokat. Majd a feltöltés dátuma jön, a létrehozó és egy egyedi commit azonosító. Az általunk használt műveletek a commit, a push és a pull voltak [22].

Az 1.6. ábrán a SourceTree alap felhasználói felületéről egy minta látható az alkalmazás főoldaláról.



1.6 .- SourceTree felhasználói felülete [22]

2. Felhasznált technológiák

Dinamikus weblapok és webes programok készítésére egy ingyenes és nyílt forráskódú szoftver csomagot szoktak használni, és használtuk mi is, ennek a neve az angol szakkifejezésből átvett MEAN mozaikszó, ami a felhasznált négy technológia nevéből ered:

- MongoDB, NoSQL adatbázis
- Express.js, webes alkalmazás backend framework
- Angular.js, JavaScript MVC framework webalkalmazásoknak, ez a frontend rész
- Node.js, szoftver platform skálázható szerver oldali és hálózati alkalmazásokhoz

A MEAN a frontend, valamint a backend fejlesztését is lehetővé teszi [12]. A MEAN felépítése 2.1. ábrán látható.

2.1. MEAN moduljai illusztrálva [11]

2.1 Fejlesztői környezetek

A már említett szoftvercsomagon kívül egy fejlesztői környezetre is szükség van, amiben a fejlesztést lehet végezni. Az integrált fejlesztői környezet (angolul IDE, Integrated Development Environment) a neve a fejlesztők munkáját megkönnyítő, programozást jelentősen egyszerűsítő, részben automatizáló programoknak.

Az IDE-k tartalmaznak egy szövegszerkesztőt a program forráskódjának szerkeszthetősége érdekében, egy fordítóprogramot, fordításautomatizáló eszközöket, grafikusfelület szerkesztőket és beépített verziókezelőket is. A nagyobb környezetekhez további rengeteg kiegészítő (plugin) is elérhető, amelyek a fejlesztés más szakaszaiban, például tesztelésben nyújtanak segítséget.

Főbb lépések, amit egy IDE támogat:

- program megtervezése
- kódírás
- fordítás
- tesztelés
- hibakeresés (debug)

A fejlesztői környezettel szemben elvárás, hogy a forráskódot automatikusan tudja formázni az adott nyelvnek megfelelően és a kódban segítse a tájékozódást színkiemeléssel. A fordítás tudjon végrehajtható programot készíteni, megfelelően kezelje a hibákat, érthető legyen, hol van a gond. Ha nem sikerül a fordítás, ugorjon az első hibás sorhoz. A debug tegye lehetővé a program lépésenkénti futtatását.

A legjobb IDE-k a kód írása közben ellenőrzik is azt, valamint a forráskódban történő keresésben fejlett lehetőséget nyújtanak [8].

2.1.1. WebStorm

A JetBrains kiadó terméke a WebStorm IDE. Ez egy fejlesztői környezet, ami tökéletesen felszerelt éppúgy a kliens oldali, mint a szerver oldali fejlesztéshez is. A WebStorm okos kódszerkesztője első osztályú JavaScript, Node.js, HTML és CSS támogatást biztosít. További előnyei közé tartozik még az automatikus kódkiegészítés, hiba detektálás és kód refaktor.

A hibákat on-the-fly, tehát futás közben azonnal detektálja. A refaktor automatizált, de mégis biztonságos a használata.

Rengeteg időt megtakarít a beépített verziókezelő, ami a Git-et, az SVN-t és a Mercurialt egyesíti.

Rengeteg bővítmény (plugin) letölthető a WebStormhoz, amikkel az alapvető funkciókat terjesztik ki.

Mi a Mongo Plugin (Mongo Explorer) nevű plugint használtuk. Ezzel nem kell a mongo shell-t használni, helyette egy felhasználói felületet kapunk, ami az adatbázissal kapcsolatos műveleteket könnyíti meg. Saját query szűrővel is rendelkezik, hogy a kollekciókból megkapjuk a kívánt adatot, valamint szerkeszthetőek is a már feltöltött adatok ezen a felületen keresztül. Használata egyszerű, megnyitjuk a beállításai, megadjuk a mongo elérési útvonalát, majd hozzáadunk egy szervert nekünk tetsző névvel, és innentől ha fut a MongoDB-t használó projekt, meg is jelenít minden táblát az adott adatbázishoz [24].

2.2. NoSQL

A NoSQL (not only SQL) egy gyűjtőnév, magában foglalja az összes adatbázist, ami eltér a relációs adatbázisoktól – RDMS – (elhagyják a JOIN műveleteket, tábláik között nincs kapcsolat).

A relációs adatbázisok hiányosságait áthidalni jöttek létre. Ilyenek például:

- rugalmatlanság (az inkrementális fejlesztések folyamatosan változó követelményeiből kifolyólag nem tudnák megfelelően követni a változást),
- relációktól való erős függés (ha például nagyon sok mezőt kell felvenni és sajátot is vehet fel egy felhasználó, akkor nem optimális a relációs adatbázisok használata, és le is lassul a keresés),
- hosszú válaszidők (nagy adatbázisoknál a sok relációs kapcsolat túl hosszú válaszidőt eredményez).

Összességében a NoSQL adatbázisoknak sokkal nagyobb a szabadságfoka a relációs adatbázisokhoz képest.

A legtöbb NoSQL adatbázis erősen optimalizál a CRUD (Create, Read, Update, Delete) alapvető adatbázis műveletekre, ezeken kívül viszont nem támogatnak sokkal több műveletet. Ezt a korlátozást viszont ellensúlyozza jobb sebességével, skálázhatóságával. Ismert NoSQL adatbázisok: Apache Cassandra, MongoDB (erről a következő pontban bővebben), CouchDB.

Ha egy meglévő NoSQL adatbázist szeretnénk kiegészíteni egy új mezővel (az adatbázis módosítása nélkül), akkor semmi probléma nem merül fel, mivel az ilyen típusú adatbázisoknál nincs előre megszabva, hogy hány oszlopból kell állnia egy sornak, továbbá itt az sem annyira fontos. Relációs adatbázisoknál mindezt folyamatosan mozgatni kéne, de ezt az oda-vissza mozgást nehéz lenne megoldani, egy cégnél mindig jelen kéne lennie egy adatbázis szakértőnek, ezzel viszont a kiadások is sokszorosan megnőnének.

A NoSQL rendszereket gyakran nevezik „Not only SQL”-nek, hogy kihangsúlyozzák, támogatják az SQL szerű lekérdező nyelveket.

Néhány főbb csoportra szét lehet bontani az adatbázisokat, ezek főleg adatstruktúrában és a más-más kitűzött célban térnek el egymástól. Az ilyen adatbázisok szegregációját jól szemlélteti, hogy nosql-database.org-on már több, mint a kétszerese létezik a kezdeti négy csoportnak.

- Kulcs-érték tárolók (Key-Value databases/stores): A kulcs-érték tárolók a legegyszerűbbek a négy kezdeti típusból. Az adott információt tartalmazó rekeszt egy

kulcs segítségével lehet elérni. Választhatja ezt a kulcsot a felhasználó is, de le is generáltathatja azt.

Egy átlagos kulcs-érték tároló string kulcsokkal és string értékekkel dolgozik, ezeket byte tömbökké alakítják. Az adatokat a memóriában tárolja el a kiszolgálás gyorsasága érdekében.

Jelentősebb adatbázisok ebben a típusban: Amazon SimpleDB, Microsoft Azure Table Storage

- Dokumentumtárolók (Document stores): Ennél az adatbázis típusnál is hash alapján történik az adatok tárolása az adatbázisba, de itt a letárolt adatnak van szerkezete, tehát az adatbázis képes kereséseket és gyűjtéseket végezni az adatstruktúra alapján. A dokumentumtárolók a legnépszerűbb adatstruktúra. A fejlesztés kiváltó oka az, hogy a fejlesztők gyakran az adatbázisokban dokumentumokat tárolnak, amiket egy kulccsal érnek el.

  Leíró nyelve a JSON, ami egy olyan struktúrát biztosít, ami az emberek számára könnyen értelmezhető, a számítógépek pedig könnyedén fel tudják dolgozni azt.

  Jelentősebb dokumentumtárolók: CouchDB, MongoDB.

  A MongoDB-t részletesebben a következő pontokban fogom kifejteni, mivel a fejlesztés során a szoftverfejlesztő csapatunk ezt a tároló technikát használta.
- Oszloptárolók (Column stores): Sorokban tárolás helyett ez az adatbázis típus oszlopokban tárolja az adatokat. Ha csak egy mezőre van szükség, akkor először a megfelelő sort keresi ki, majd a sorban a megfelelő oszlopot.

  Jelentősebb oszloptárolók: Cassandra, Hadoop.
- Gráftárolók (Graph stores): Gráf technológián alapul a működésük, olyan adatokat tárolnak, amelyek gráfként jól modellezhetők, azaz határozatlan számú kapcsolat van az adatok között.

  Jelentősebb gráf tárlók: közösségi oldalak, Neo4J, InfoGrid [18].

2.2.1. MongoDB

A MongoDB egy nyílt forráskódú, cross-platform adatbázis program, ami a NoSQL adatbázisok dokumentumtároló típusába tartozik és a MongoDB Inc. fejleszti. Neve az angol humongous szóból ered, ami óriásit jelent. Az adatokat ODM (Object Document Mapping) technológiával tárolja.

Ami a legfontosabb, nincs adatbázis-séma, dokumentumonként más mezők helyezhetőek el benne. A benne eltárolt adatok JSON-szerű (BSON) formában találhatóak meg az adatbázisban. SQL utasítások helyett itt függvényhívások vannak.
Rengeteg olyan funkcióval rendelkezik, mint a hagyományos relációs adatbázis kezelő rendszerek, ilyenek például a másodlagos indexek, gazdag módosítási lehetőségek, upsertek (update, ha a dokumentum létezik, egyébként insert). Funkcionalitásban elérhetjük a relációs adatbázisok szintjét, de azzal a rugalmassággal és skálával, amit a nem-relációs modell megenged.

2007-ben egy Platform as a Service (PaaS – Szolgáltatásként kínált platform, ami egy felhőszolgáltatásként kínált rendszer, az előfizetőnek számítógépes platformot kínál a szolgáltató) részének tervezték, a vállalat utána váltott 2009-ben open-source fejlesztési modellre.

2015-ös közvélemény kutatás alapján a legnépszerűbb NoSQL és a negyedik legnépszerűbb adatbázis kezelő rendszer. Olyan nagyobb internetes oldalak és szolgáltatások is használják, mint a FourSquare, eBay, MTV Networks, FIFA (videó játék), MetLife, SAP, McAfee, LinkedIn.

Főbb jellemzői:

- Ad-hoc lekérdezések: A MongoDB támogatja mind a mező érték, intervallum, reguláris kifejezésekkel való keresést is. A lekérdezések visszatérhetnek a dokumentum szükséges mezőjével, vagy akár a felhasználó által definiált JavaScript funkciókkal is.
- Indexelés: Minden egyes mező a dokumentumokban indexelhető elsődlegesen és másodlagosan is.
- Replikáció: Támogatja a master-slave replikációt (replica sets). Írási műveleteket csak a master hajthat végre, a slave szerverek másolják az adatokat. A slave adatbázisok képesek új master adatbázist választani, ha a master meghibásodik (automatikus szerepváltás).
- Terhelés elosztás: A mongo horizontális skálázhatóságához a sharding módszert alkalmazza. Ez lehetőséget nyújt adatok tárolására több számítógép között. A szilánk egy mester és egy, vagy több szolgából áll. A kollekciók sharding mezői minden dokumentumban legyenek benne.
- Fájltároló: Köszönhetően a terhelés elosztásnak és a replikációnak, a mongo-t lehet fájlrendszerként is használni. Ezt Grid File System-nek, vagy röviden GridFS-nek

nevezik. Ahelyett, hogy a fájlt egy egyszerű dokumentumban tárolná, részekre bontja azt, és minden egyes darabot külön dokumentumként tárol el.

- Aggregáció: A MapReduce egy adatfeldolgozó paradigma, ami nagy adatkötegeket sűrít hasznos eredményhalmazokba. Használható kötegelt feldolgozáshoz és aggregációra. JavaScript nyelven kell írni.

Adatmodell:

1. Egy MongoDB példány több adatbázist is tartalmazhat, amik magas szintű tárolókként viselkednek.
2. Egy adatbázis több gyűjteményt (collection) tartalmazhat.
3. Egy gyűjtemények dokumentumokból áll, amikre tekinthetünk sorokként.
4. Egy dokumentum pedig több mezőből áll (BSON objektumok).

Ha egy új sor kerül beszúrásra, és az adatbázis vagy a gyűjtemény nem létezik még, akkor az automatikusan legenerálódik a konfigurációban megadott paraméterekkel, sőt, automatikus indexeléssel.

Fontosabb korlátozások:

- BSON dokumentumok maximum mérete 16MB lehet. Ez a maximum méret biztosítja, hogy egy szimpla dokumentum nem használhat túl nagy mennyiséget a RAM-ból, vagy küldés közben nem foglal túl nagy sávszélességet. A maximum méretnél nagyobb dokumentumok eltárolására a MongoDB a GridFS API-val biztosítja.
- Linuxon egy mongo példány maximum 64 TB adatot tartalmazhat, míg Windowson a maximum mindössze 4 TB.
- A maximum beágyazás egy dokumentumnál 100 lehet (az első dokumentumba ágyazva a második, a másodikba a harmadik és így tovább).
- Egy namespace mérete 120 byte lehet (Például: <database>.<collection>).
- Egy indexelt mező nem tartalmazhat 1024 byte-nál több adatot.
- Egy gyűjteményben 64 index lehet legfeljebb.

Használata:

MongoDB telepítése a következő paranccsal lehetséges: ”npm install mongodb”. Ez a parancs kiadható a fejlesztői környezet saját termináljában, vagy az operációs rendszer termináljában is. A ”--global” kapcsolóval globálisan telepítjük a számítógépre, így bárhol elérhetjük a mongo-t, nem csak abból a mappából, amelyikben kiadtuk a telepítés parancsát. Ahová telepítettük a MongoDB-t ott data\db mappát létre kell hozni, mert a mongo ott dolgozik.

Adattárolás:

2^n méretű elosztás: A MongoDB 3.0 alap stratégiaként a kettő hatványai elosztást használja. Ezzel a stratégiával minden rekordnak van egy mérete bájtokban, ami kettő hatványa (32, 64, 128… 2MB). 2MB fölötti dokumentumok a méretükhöz legközelebbi hatványát kapják a kettőnek, maximum 2 GB lehet egy dokumentum.

Ezzel a stratégiával hatásosan újrahasználhatóak a felszabadított rekordok, ezzel a töredezettség szintje csökken.

A /data/db könyvtárban kerülnek tárolásra az adatok. Két típusú fájl található gyűjteményenként itt, az egyik a ”gyűjtemény.0” (és ahány adatbázis szelet, annyiszor növekszik ez a számozás a gyűjtemény neve után), ami magát az adatot tárolja és a ”gyűjtemény.ns”, ami a namespace metaadatokat tartalmazza a gyűjteményekhez [17].

2.2.2. MongooseDB

A Mongoose a MongoDB-nél egy magasabb szintű, könnyen kezelhető, séma alapú lehetőséget kínál az adatok modellezésére.

A mongohoz képest funkciói annyival változtak, hogy sémákat lehet benne létrehozni, ez meggátolja, hogy struktúra nélkül kerüljenek adatok az adatbázisba. Sémák feletti metódusokat enged definiálni. (A séma nem más, mint egy objektum, ami definiálja egy dokumentum felépítését, ami a mongo egyik gyűjteményében lesz majd később tárolva. Előredefiniálhatóak típusok a későbbi adatok számára.)

MongoDB-ben nincsenek kapcsolatok (JOIN), de néha mégis szükség van más gyűjtemények dokumentumaira hivatkozni, ennek a problémának az áthidalására jött létre a Population folyamat. Automatikusan felülírja a megadott útvonalakat a dokumentumokban a hozzákapcsoltból.

Telepítése az ”npm install mongoosedb” paranccsal történik [13].

2.3. Node.js

Szerver oldali (backend) fejlesztéshez Node.js-t használtunk. Ez egy nyílt forráskódú környezet, ami a Google V8 JavaScript motorja felett íródott. Ez a rendszer egy JavaScriptet alkalmazó, esemény vezérelt, nem blokkoló I/O modellt használ, ami könnyen felhasználhatóvá és hatásossá teszi. Talán a legfontosabb előnye az, hogy egy esemény hurokban (event loop) fut a program. Architektúrája egyszálas, minden futtatás aszinkron. Csomagkezelője, az npm, a világ legnagyobb nyílt forráskódú könyvtár nyilvántartó rendszere. Mára használatuk annyira összenőtt, hogy együtt települ a két szoftver.

Története:
A node története összeforr a V8 történetével. A V8 egy nyílt forráskódú Google projekt, ami a Google Chrome böngésző alapját adja. 2008 szeptember 2-án adták ki. Hatalmas lépés volt megjelenése az internet világában, új szintre emelve a böngészők technológiáját. C++-ban íródott és a legnagyobb újítása az volt, hogy előre lefordítja a JS forráskódot gépi kódra, ahelyett, hogy előbb értelmezné, majd azután használná rajta a JIT műveletet, ezzel fejlesztve a dinamikus kód futtatást.

Majd 2009-ben Ryan Dahl létrehozta a node.js-t, munkáját a Joynet szponzorálta, Ryan munkáltatója. Az eredeti probléma, ami miatt elkezdte a fejlesztést az volt, hogy a böngészők nem jelezték sehogy sem, mennyi idő van még hátra egy feltöltésből. Végül a JavaScript-et választotta I/O API hiánya, eseményvezérelt és nem blokkoló felépítése miatt.

A csomagkezelő által kezelhető modulokon kívül vannak alapértelmezetten beépített moduljai is, amik számos tulajdonsággal ruházzák fel már kezdéskor is. A modulok a CommonJS specifikáció alapján készültek, így az alkalmazásokba a ”require” paranccsal kerülhetnek be. Beépített csomagok:

- fs (filesystem): Az egyik leggyakoribb modul, fájlműveletek végrehajtására használható. A ”var fs = require('fs');” paranccsal hozzuk létre
- http: Alap http szerverek elkészítéséhez használható. var http = require('http');
- url: URL-ek kezeléséhez

Aszinkron eseményvezérelt keretrendszer lévén, úgy tervezték, hogy képes legyen skálázható hálózati alkalmazásokat létrehozni. Mivel minden eseménye aszinkron, a futó program soha nem blokkolódik, az események párhuzamosan futnak, nem várnak arra, hogy egy másik esemény befejezze a munkát előttük. Ez a tulajdonság segíti az optimális, folyamatosabb működést, valamint több program párhuzamosítását.
Természetesen előfordulnak esetek, amikor pont ez az aszinkronitás okoz gondot a fejlesztés során, főleg a visszahívásoknál (callback). Több könyvtár is létezik ennek áthidalására, a fejlesztés menetében erről bővebben írok majd [16].

2.3.1. Express Framework

A node http modulja megfelelően működik, de sajnos az alapvető funkcionalitásokat a fejlesztőnek magának kell megvalósítania. Ennek a problémának a megoldására jött létre az Express keretrendszer, kiváltva a routing és a statikus fájlok kiszolgálását. A node alapjait felhasználva segíti a fejlesztést.

Az Express telepítése is úgy történik, mint a többi modulé. A ”npm install express” parancs kiadása után települ a node_modules mappába.

Egy új webes alkalmazás létrehozása ezzel a keretrendszerrel pár lépésből áll. Parancssorban először is telepítjük a generáló eszközt (npm install express-generator -g), majd kiadjuk az express ”alkalmazás_neve” parancsot és már létre is jön a megadott néven a kívánt alkalmazás. Telepít is egyből minden függőségét, létrehozza a package.json fájlt, amiben a szükséges függőségek találhatóak meg. Majd az npm install parancs telepíti a szükséges modulokat. Egy könyvtárszerkezetet is legenerál, ami a projekt vázát adja. Ezek a könyvtárak:

- public: ide kell rakni az összes statikusan kiszolgálható fájlt, például a stílusleíró fájlokat, JS fájlokat és a képeket
- routes: ide kerül az index.js, ez tartalmazza az url-ekhez tartozó JS fájlokat, amik a különböző kérésekre reagálnak (ez a node kézi működtetésének javítása)
- views: html template fájlok kerülnek ebben tárolásra

Egyszerűbb weblap példa:

```
//modulok behúzásával kezdjük mindig (require parancs)
var express = require("express");
var http = require("http");

//az app létrehozása, ez az alkalmazás fő JS fájlja
var app = express();

app.use(function(request, response) {
  response.writeHead(200);
  response.end("Hello world!\n");
});

//app elindítása és az 5000-es porton hallgat
//(localhost:5000 url be is tölti ezt egy böngészőben)
http.createServer(app).listen(5000);
```

Routing

Lehetőségünk van a routing használatára, amivel meghatározható, hogy az alkalmazás hogyan reagáljon egy ügyfél kérésére egy adott végponton, ami egy url és egy speciális http kérés (POST, GET, DELETE, PUT).

Példák a hivatalos weblapról:

```
// válasz "Hello World!"-del a kezdőoldalon
app.get('/', function (req, res) {
  res.send('Hello World!');
});
// elfogadja a POST felkérést a kezdőoldalon
 app.post('/', function (req, res) {
  res.send('Got a POST request');
});
// elfogadja a PUT felkérést a /user url-nél
app.put('/user', function (req, res) {
  res.send('Got a PUT request at /user');
});
// elfogadja a DELETE felkérést a /user url-nél
app.delete('/user', function (req, res) {
  res.send('Got a DELETE request at /user');
}); [5]
```

2.4. NPM

Az npm azaz Node Package Manager (Node csomagkezelő) node programok telepítésére és karbantartására használható. Könnyebbé teszi a JavaScript fejlesztők számára a kódjaik megosztását és újrafelhasználását. A rendszerben jelenleg (2015. november 25.) 209153 csomag van, átlagosan napi 349 új modullal bővül ez a lista és naponta több mint 60 milliót installálnak programozók. Az npm használata a node-dal annyira összenőtt, hogy együtt is települ vele, nem kell külön telepíteni, kezelése egyszerű.

Egy csomagot kétféleképpen lehet telepíteni, az egyik a helyi telepítés, a másik pedig a globális.

Ahhoz, hogy egy csomagot a helyi mappába telepítsünk, nem kell mást tenni, mint kiadni a következő parancsot:

”npm install <csomag_neve>” (hosszabb parancsok lerövidíthetőek, például az ”install” helyett elég ”i” betűt írni).

A parancs lefuttatása után a node létrehoz (ha még nem létezett) egy node_modules mappát, benne a telepített csomag nevével megegyező mappával.

Globálisan olyan csomagokat szokás telepíteni, amiket terminálból akarunk futtatni és később is szükségünk van még rájuk, más projektekben is. A parancs ugyan az, mint a lokális változatnál, csak az ”install” után és a modul neve előtt egy ”–g” (global) kapcsolót kell használni: npm install -g mongodb

Csomagok törlésére az ”npm uninstall <csomag_neve>”, míg frissítésére az ”npm update <csomag_neve>” paranccsal történik.

Lehetőség van több csomagot egyszerre telepíteni egy konfigurációs fájllal, npm-nél ennek package.json a neve, npm init paranccsal hozzuk létre, amit a ”--save” kapcsolóval bővíthetünk. A benne lévő modulokat az ”npm install” paranccsal telepíthetjük egyszerre [19].

2.5. JavaScript

A Javascript (sűrűn használt rövidítése a JS) egy 1995-ben megjelent, gyengén típusos, interpretált szkript nyelv.

A gyengén típusosság azt jelenti, hogy változói ugyan vannak, de a benne lévő értékek határozzák meg azok tényleges típusát (változót a var kulcsszóval lehet létrehozni, például ”var peldavaltozo = 0”, ez egy szám típusú változó lesz).

A JS egy szkript nyelv, a böngészőbe beépített JavaScript interpreter (értelmező) értelmezi soronként. Ezeknél a nyelveknél nincs fordítási fázis, nem a lefordított kód fut, hanem az értelmező azonnal elkezdi a program végrehajtását. Azt, hogy van-e a programban hiba, csak akkor derül ki, ha az értelmező ráfut.

A böngésző a HTML oldalt sorfolytonosan tölti be és jeleníti meg. Ha script blokk következik, akkor annak JS kódját futtatja. Ennek befejezése után folytatja a HTML kód betöltését. Ha egy script blokk kódja hiba, végtelen ciklus vagy más hatására megáll, akkor az oldal betöltése is az akkori helyzetben marad.

A JavaScript nyelv megalkotásához több nyelvből merítették az ötleteket.

- a C nyelvcsaládtól vették át a szintaxisát
- a Scheme funkcionális programozási nyelvtől kapta a funkcionális aspektusát
- a Self nyelv OOP hibáit hivatott kijavítani úgy, hogy az osztályok és az objektumok közötti különbséget megszüntették benne, helyettük dinamikus objektumok vannak
- a Hypercard rendszerből pedig az eseménykezelési filozófiát vette át

Szkript nyelveknek azokat a programozási nyelveket nevezzük, amelyek egy adott alkalmazás vagy működési környezet vezérlését teszik lehetővé, hiszen a böngésző és a betöltött dokumentum állapotát változtatja meg [9].

2.5.1. AngularJS

Az AngularJS egy 2009-ben megjelent, nyílt forráskódú, deklaratív, kliensoldali UI keretrendszer, amit a Google fejlesztett, dinamikus webes alkalmazások készítéséhez

használható. Használata lényegesen megkönnyíti a frontend fejlesztést. Az AngularJS egy MVC alapú keretrendszer, ennek felépítése a következő címpontban kerül kifejtésre (2.5.2.)

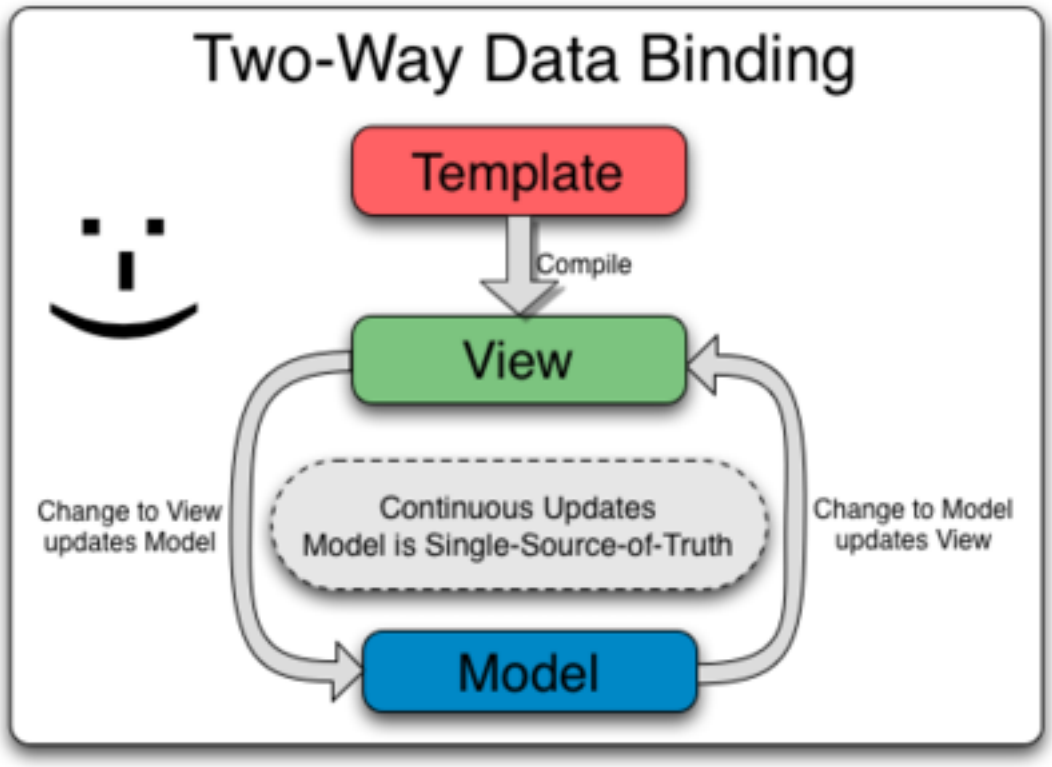
A keretrendszer főbb célkitűzése az alkalmazás kliens, és szerver oldalának teljes különválasztása.

A scope-ok, direktívák és a kétirányú kötések adják az alapját.

JS-ben a HTML a scope, Angularban viszont scope hierarchia van.

A direktívák a HTML tag-eknek adnak extra tulajdonságokat. Használatuknak hála nincs szükség közvetlen DOM változtatásra. Minden direktíva parancs ”ng-”-vel kezdődik. Leggyakrabban használt elemei az ”ng-app”, ”ng-model”, ”ng-controller”, ”ng-show/hide”, ”ng-if”.

Kétirányú kötések: A fordítás eredménye nem statikus lesz. Bármilyen változás a nézetben azonnal megjelenik a modellben és fordítva. Ennek köszönhetően az állapot egyedül a modellben tárolódik, tehát a nézet vehető a modell egy kivetülésének. Ennek köszönhetően a kontroller teljesen elkülönített, így megkönnyíti a tesztelhetőséget is [2]. Ennek a folyamatát mutatja be a 2.2. ábra.



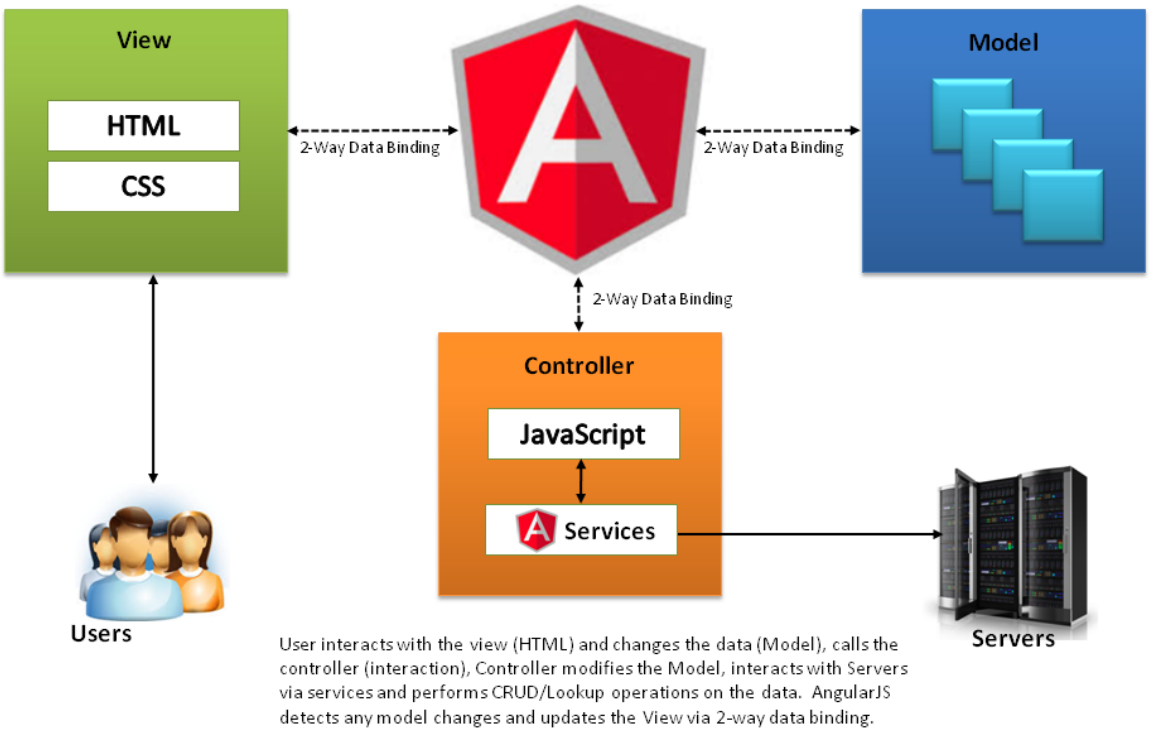
2.2. – Kétirányú kötés illusztrációja [10]

2.5.2. MVC

A Model View Controller (MVC) egy szoftverfejlesztési minta webes alkalmazások fejlesztéséhez. Három részből áll:

- Model − Ez a legalacsonyabb szint, az adattartalom fenntartásáért felelős
- View − Ez a rész felelős a megjelenítésért, hogy mit lássanak a felhasználók
- Controller − A köztes réteg, ami a model és a view közötti kölcsönhatásokat irányítja.

A 2.3. ábra jól illusztrálja a rétegek kapcsolatát, a kérések és válaszok irányát minden résztvevő szemszögéből [15].



2.3. – Az MVC modell [14]

2.6. HashMap

A HashMap egy kulcs/érték párokat létrehozó JavaScript osztály. A mappelés egy megfeleltetés két adat között. Eltérően a hagyományos objektumoktól, a kulcsok nem lesznek sztringgé alakítva. A számok és betűk külön használhatóak (példa lentebb), tovább adhatunk dátumokat, reguláris kifejezéseket, DOM (Dokumentum Objektum Model) elemeket, sőt, még a null és undefined értékekkel sincs gondja.

Telepítése az ”npm install hashmap” parancsal történik.

Először is importálni kell magát a HashMap-et: ”var HashMap = require('hashmap');” Ezután már elérhetőek a metódusai. Fontosabb metódusok például a get, set, has, keys és values. Ezek részletes működését a Fejlesztés menete című fejezet második részében fogom kifejteni.

Első példa az alap működésére:

```
map.set("pelda_kulcs", " pelda_ertek ");
map.get("pelda_kulcs "); // --> "some value"
```

Az első parancsban ”set”-tel beállítjuk a ”pelda_kulcs” értékét, ami ”peda_ertek” lesz. Majd a másodikban a ”get” visszaadja ”pelda_kulcs” értékét, ami tényleg a ”pelda_ertek”.
A második példa a sztringgé alakítás hiányát szemlélteti:

```
map.set("1", "string_1");
map.set(1, "szam_1");
map.get("1"); // --> "string_1"
```

Abban az esetben, ha egy objektumot használunk mapben, akkor a ”szam_1” érték került volna visszaadásra [6].

2.7. Highcharts

A Highcharts egy JavaScriptben írt, grafikonmegjelenítő könyvtár, ami egy könnyű és gyors megoldást kínál interaktív grafikonok hozzáadására a webes projektekhez. Szintaxisa egyszerűvé teszi használatát. Amennyiben valaki nem profitszerzés céljából használja fel, akkor teljesen ingyenesen elérhető.

Támogatja az oszlop, kör, vonal, torta, buborék, tölcsér, vízesés és rengeteg más típusú grafikont is. Több ezer fejlesztő és a világ 100 legnagyobb cégéből 61 használja. Minden modern mobil és számítógépes böngészővel fut, mobil eszközökön még a multitouch támogatás is elérhető, hogy segítse a zökkenőmentes munkát a felhasználók számára.

Az egyik, ha nem a legfontosabb tulajdonsága az opensource beállítottság, mivel a forráskód szabadon letölthető és bárki kiegészítheti a számára szükséges változtatásokkal.

Kizárólag natív böngésző technológiákon alapul, nincs szükséges kliens oldali bővítményekre, mint a Flash vagy a Java. Mindössze két JS fájlra van szükség hozzá, az egyik a Highcharts.js alap fájlja, a másik pedig egy keretrendszer (jQuery, MooTools vagy Prototype) [7].

2.8. Bower

A bower egy csomagkezelő modul frontend fejlesztéshez. Összességében a csomagkezelők feladata nem más, mint más programok kezelése, tehát telepítés, frissítés és törlés. Rengeteg plugin áll rendelkezésre a frontend fejlesztéshez, és előfordulhat, hogy egy plugin egy, vagy

több további plugint használ, erős függőségeket létrehozva ezzel, amely helyzeteket a csomagkezelők tudnak kezelni.

Egyszerű használata mellett előnye még, hogy a JavaScript mellett több CSS keretrendszerhez is felhasználható. Továbbá a már elkészült fejlesztéseket segít megtalálni, felgyorsítva ezzel a fejlesztők munkáját, hogy ne kelljen még egyszer elkészíteni, ami már készen van. A megtalált modulokat függőségeikkel együtt telepíti és figyelmeztet, ha frissebb verzió érhető el valamelyik bővítményéhez.

Csak akkor telepíthető a Bower, ha már a NodeJS a számítógépen van, ezután terminálban egy egyszerű parancs kiadásával telepíthető: ”npm install bower”.
Miután megtaláltuk a számunkra szükséges csomagot, telepíthetjük azt az egyszerű ”bower install <csomag_neve>(#<version>)” paranccsal. A ”#<version>” kapcsolóval variálhatunk a verziók között, ha nem az alapértelmezettet szeretnénk telepíteni. A már telepített csomagok a bower_components mappában találhatóak, ezeket HTML script tag-ekkel érhetjük el: <script type="text/javascript" src="bower_components/modul"></script>

Ahogy az npm-nél, úgy itt is létre lehet hozni a projekt mappájában egy konfigurációs fájlt, aminek a neve bower.json, a ”bower init” parancs hozza ezt lére, és a ”bower install” parancs telepíti az összes benne található modult. Bővíteni pedig a ”--save” kapcsolóval lehet egy csomag telepítésekor. Például ”bower install --save <csomag_neve>” [4].

2.9. ACL

Weboldalak és alkalmazások fejlesztésekor gyorsan egyértelművé válik, hogy a session-ök nem biztosítanak kellő védelmet az adatainknak, az illetéktelen behatolókkal szemben viszont nehezebb a védekezés, mint először gondolná egy fejlesztő. Erre nyújt rugalmas megoldást az ACL (Access Control Lists, Hozzáférés jogosultsági lista), az egyik legismertebb hozzáférés szerepkör kezelő függvénykönyvtár. Ez egy lista, ami tartalmazza, hogy melyik objektumhoz melyik felhasználónak milyen jogosultsága van, és mit tehetnek az adott objektummal. Példák hozzáférési jogokra: az olvasás (read), az írás (write) és a futtatás (execute). A felhasználók egyéni szerepeket kaphatnak vagy szerepkörökbe sorolhatóak többen egyszerre.
Ezeket a jogosultságokat többféleképpen is tárolhatjuk, memóriában, vagy köztes rétegen, mi az utóbbi típus egyikén, a mongoDB-n tároltuk.

```
acl = new acl(new acl.mongodbBackend(mongoose.connection.db, 'acl'));
```

A konstruktor két paramétert vár, az első egy adatbázis példány, a másik pedig egy prefix, mely az ACL által definiált táblák nevének lesz az előtagja. Ez a paraméter opcionális.

Segítségével definiálhatjuk, ahogy az egyes szerepkörök, adott objektumokon milyen eljárásokat végezhetnek.

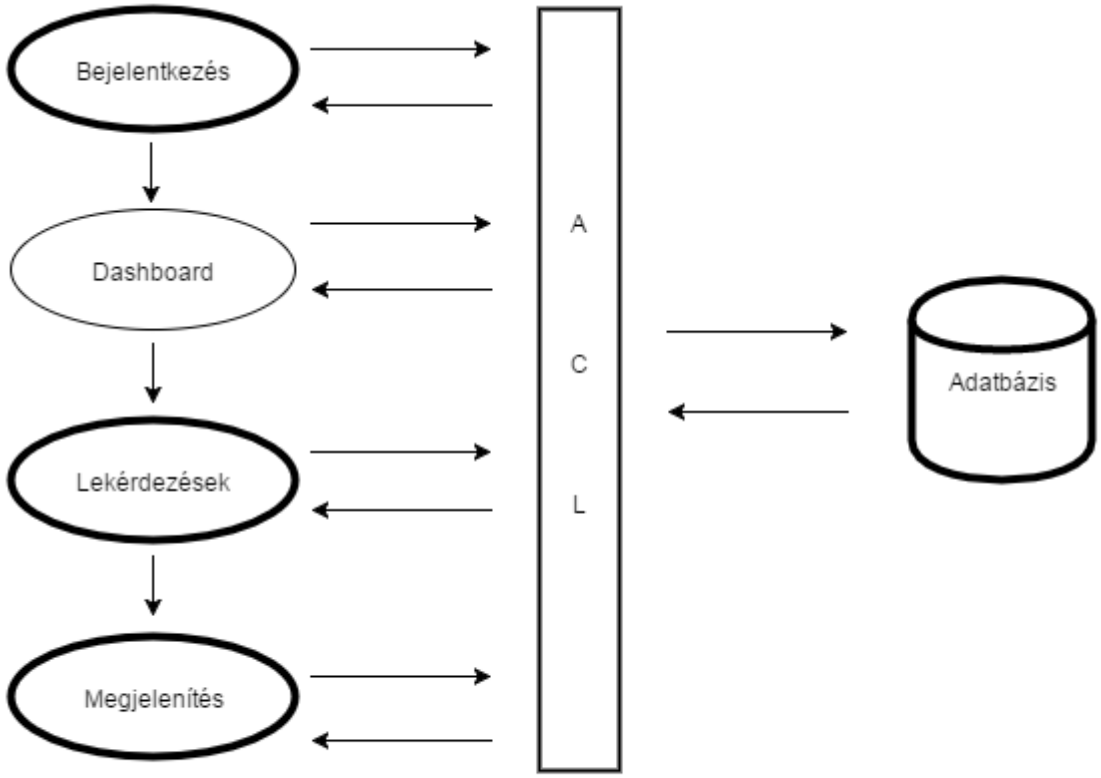
Az ACL modult importálni kell a ”require” paranccsal: ”var acl = require('acl');”. Impliciten is létre lehet hozni jogosultságokat engedélyek megadásával: ”acl.allow('member', 'blogs', ['edit','view', 'delete']);” Az allow metódus, mint a szó angol értelme is (enged), megengedi a member, tehát tag jogosultságú felhasználóknak, hogy a blogokat szerkessze, olvassa és törölje is. Tömböket is elfogad paraméterként, ebben a példában több jogot rendeltünk egy tag-hoz, de több taghoz is lehet akár egy, akár több jogot rendelni és mindezt akár több objektumon is tehetik. Ugyan így működik a felhasználók létrehozása is: ”acl.addUserRoles('tamas', 'development')” [1]

3. Fejlesztés menete

3.1. Az elkészítés célja

A feladat egy orvosi költségelszámoló és döntéstámogató szoftver részegységének fejlesztése, ami az Orvostudományi kar Egészségbiztosítási Igazgatóság részére készül. A részegység több összetett lekérdezésből és adatbázis műveletből áll, amelyet informatív grafikonokon jelenítünk meg a felhasználói oldalon. A feladat részét képezte egy olyan NoSQL adatbázis-modell kialakítása, amely megfelelő alapot biztosít a későbbiekben fejlesztésre kerülő vezetői információs rendszer kiszolgálására.

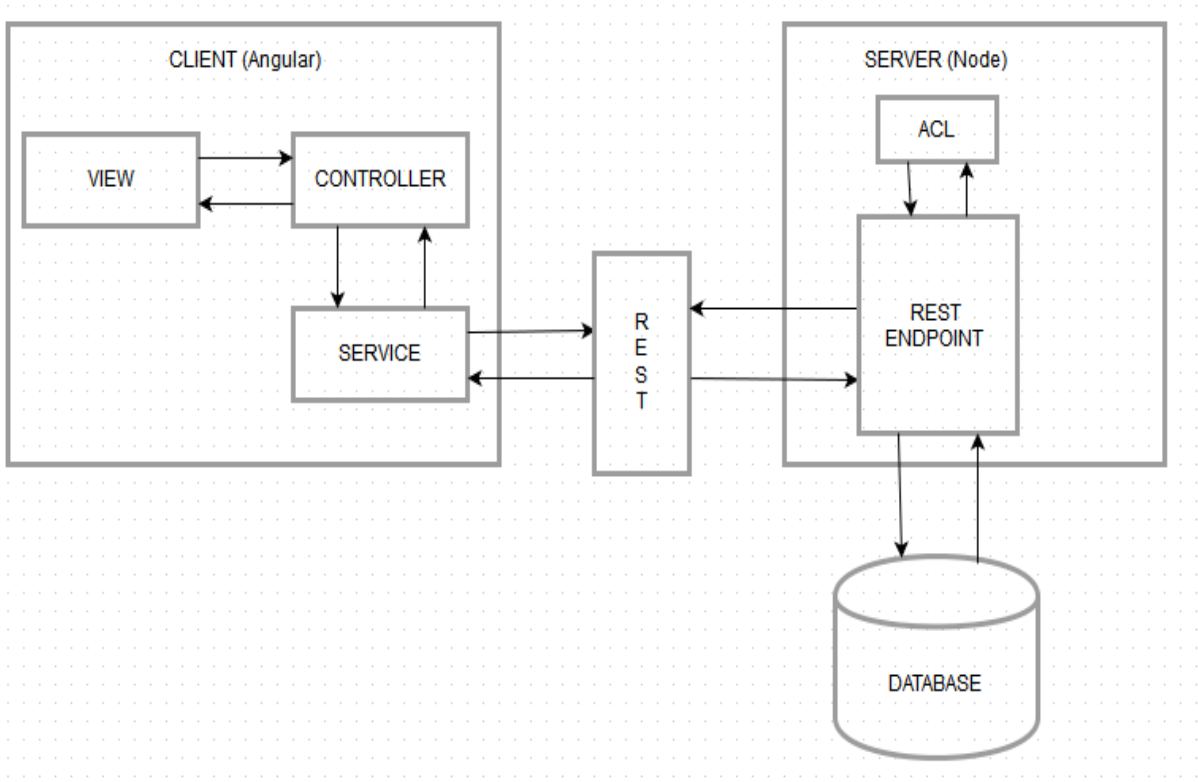
3.2. A rendszer felépítése



3.1. Az alkalmazás felépítése

A 3.1. ábrán jól látható a rendszer hat fő része, ezek közül a vastagabb kerettel jelöltek mindegyikében dolgoztam (csak a dashboardon nem).

3.3. Rendszer architektúra



3.1. A rendszer architektúrája

A 3.1. képen látható a rendszer architektúrája, jól látszik, hogy a kliens és a szerver oldal jól elkülönül.

A kliens oldali fejlesztések a nyelve az Angular, itt a ”view” és a ”service” között a ”controller” tartja a kapcsolatot.

Szerver oldalon pedig Node-ot használunk. Itt találhatóak a REST végpontok, amik az ACL jogosultságkezelővel vannak kapcsolatban.

A két réteg között helyezkednek el a REST hívások, ezek felelősek a kommunikációért. Klienstől a szerver felé haladnak ezek a hívások.

A szerver oldallal pedig az adatbázis van összeköttetésben, ami jelen projektben a MongoDB, innen töltődnek be a kért adatok.

Már a projekt mappaszerkezetén is jól megfigyelhető ez az elosztás. A gyökér könyvtárban található a kliens oldalhoz tartozó ”public” mappa, ebben van elhelyezve minden kliens oldalon megjelenítendő felület és a hozzájuk tartozó logika, tehát az általam készített, már említett grafikonok is. Az ehhez az oldalhoz tartozó, már telepített Bower modulokat tartalmazó ”bower_components” mappa, ami szintén a gyökérkönyvtárban található. A ”server” mappaszerkezet tartozik a szerver oldalhoz.

3.4. Login

A login felületet a Bootsnipp login sémái közül választottam. A Bootsnipp egy weblap, ahol a Bootstrap keretrendszerhez találhatóak kódrészletek. Ez az oldal segít abban, hogy ne kelljen mindig előröl kezdeni egy design feladatot, hanem a dizájn részt kiválthatjuk és lehet egyből a logikájával foglalkozni.

Az alkalmazás olyanok számára készül, akik eddig kézzel oldottak meg minden feladatot, így célszerű volt egyszerűbb felépítésű felületet választani. Két ikonnal is jelezve van a felhasználónév és a jelszó szövegdoboza, valamint alattuk egy bejelentkezési gomb található.

Mivel ezt a felületet template-nek készítettem, hogy bármikor, bármelyik másik projektben használható legyen, így egy külön mappaszerkezetben helyeztem el. Ennek a ”public\login” az elérési útja. Itt található a html, a controller, a modul és a service.

A controllerben vannak a ki és belépéshez szükséges funkciók, amik state providerekkel biztosítják a megfelelő oldalra lépést.

3.5. Hibakezelés

Az express hibakezelő modulját használtunk. Minden hívásnak három paramétere van, a request, a response és a next. Amennyiben egy REST hívásban hiba van, akkor a ”next” paraméterrel tovább küldjük azt. Majd meghívódik a ”server\server.js” fájlból a hibakezelő modul, ami visszaküldi a hibaüzenetet és egy 500-as státuszkódot.

Egy példa hívás, ami tartalmazza a ”next” paramétert:

```
router.get('/topicNumberIO', function (req, res, next) {
    ...
    });
```

A server.js-ben lévő modul kódja:

```
app.use(function (error, req, res, next) {
    if (res.headersSent) {
        return next(error);
    }
    console.log(error);

    res.status(500).send({message: error.message});
});
```

3.6. ACL elemei

A már említett módon a mongoDB backendbe mentés öt táblát hoz létre, ezek a köetkezők:

users

_id, key , role-neve

A tábla kulcs érték párokat tartalmaz, melyek a felhasználókat kötik össze a szerepkörökkel. Minden sor egy key-t tartalmaz, mely a user azonosítója, a második attribútum pedig a felhasználóhoz tartozó szerepkör neve, mely értéke mindig true.

roles

A roles tábla nagyban hasonlít a users táblához, annyi különbséggel, hogy a key mező a role nevét tartalmazza, míg a következő sorok a role-hoz tartozó felhasználó azonosítóit tartalmazzák

resources

A táblában a role-ok vannak párosítva az általuk használható objektumokhoz

meta

A tábla két sort tartalmaz, az elsőben a role-ok vannak felsorolva, még a második sorban a rendszerben lévő felhasználók

allows_*

Speciálisabb tábla, mert minden egyes „tárgy” objektumhoz létrejön egy, és itt van összekötve, melyik felhasználó, melyik objektumhoz milyen jogkört kapott.

3.7. ACL megvalósítása

Lehetőségünk van kiválasztani, hogy az ACL milyen módon mentse el a modelljeit. Választhatjuk a memóriában, vagy MongoDB-ben történő tárolást. Mi itt az architektúra felépítése miatt a mongodbBackend-et használtuk. A későbbi felhasználás érdekében az ACL ki lett szervezve egy külön fájlba. Ez a fájl a következő pár sorból áll: Mongoose és ACL betöltés a ”node_modules” mappából, majd maga a mentési művelet.

```
var mongoose = require('mongoose');
var acl = require('acl');
acl = new acl(new acl.mongodbBackend(mongoose.connection.db, 'acl'));
module.exports = acl;
```

Mivel a module.export miatt az ACL innentől behúzható más fájlokba.

```
var acl = require('../config/acl');
```

A ”server\utils\authenticate” felhasználó hitelesítő modul használja ezt. Belépéskor világos, hogy tudni szeretné a rendszer a megadható funkciók miatt, hogy admin lépett-e a

rendszerbe, vagy sem. Erre lett írva az ”isAdmin”, ami egy összeágyazott funkció. Ezzel ellenőrizzük, hogy beléphet-e az adott felhasználó, valamint, hogy admin-e. Itt az ACL ”hasRole” metódusát kellett igénybe venni, ez egy azonosítót vár, valamint egy szerepkört. A lekérdezés megvizsgálja, hogy az adott azonosítóhoz hozzá-e van rendelve az adott szerepkör, ha igen, akkor igazzal tér vissza, ellenkező esetben hamis értéket ad vissza. Ennek függvényében az ellenőrző metódusunk, vagy továbbhív, a next callback segítségével, a tényleges REST hívás implementációjára, vagy 403-as hibakóddal visszatér, és ezzel jelzi, hogy a felhasználónak nincs joga az adott dolgokhoz.

Annak érdekében, hogy ne kelljen minden egyes adminhoz tartozó REST híváshoz külön definiálni az ellenőrzést, készítettünk egy URL prefix-et, mely hívása során mindig leellenőrizzük, hogy van-e joga a felhasználónak meghívni az adott URL-t.

app.use('/rest/admin/', authModule.isAdmin);

```
function authenticate(req, res, next) {
    if (req.isAuthenticated()) {
        return next();
    }
    res.status(401).send({message: 'Unauthorized'});
}

function isAdmin(req, res, next) {
    authenticate(req, res, function () {
        acl.hasRole(req.user.id, 'admin', function (error, hasRole)
{
            if (error) {
                return next(error);
            }

            if (hasRole) {
                return next();
            }
            res.status(403).send({message: 'Forbidden'});
        });
    });
}
```

3.8. Passport

A Passport egy Node alapú hitelesítő middlewear. Egyszerű a célt szolgál, autentikálja a REST kéréseket. A ”server\config\passport.js” útvonalon található. Függőségeit definiálni kell, ezek a passport, a mongoose és a passport-local. Valamint mivel felhasználókat belépését kell elbírálni, hogy helyes-e a jelszavuk, amit csak az adatbázisból kaphatunk meg, be kell húzni a user modellt.

```
var passport = require('passport');
var mongoose = require('mongoose');
var LocalStrategy = require('passport-local');
var userModel = mongoose.model(global.USERS_SCHEMA_NAME);
```

3.9. Fő adatbázis fájl

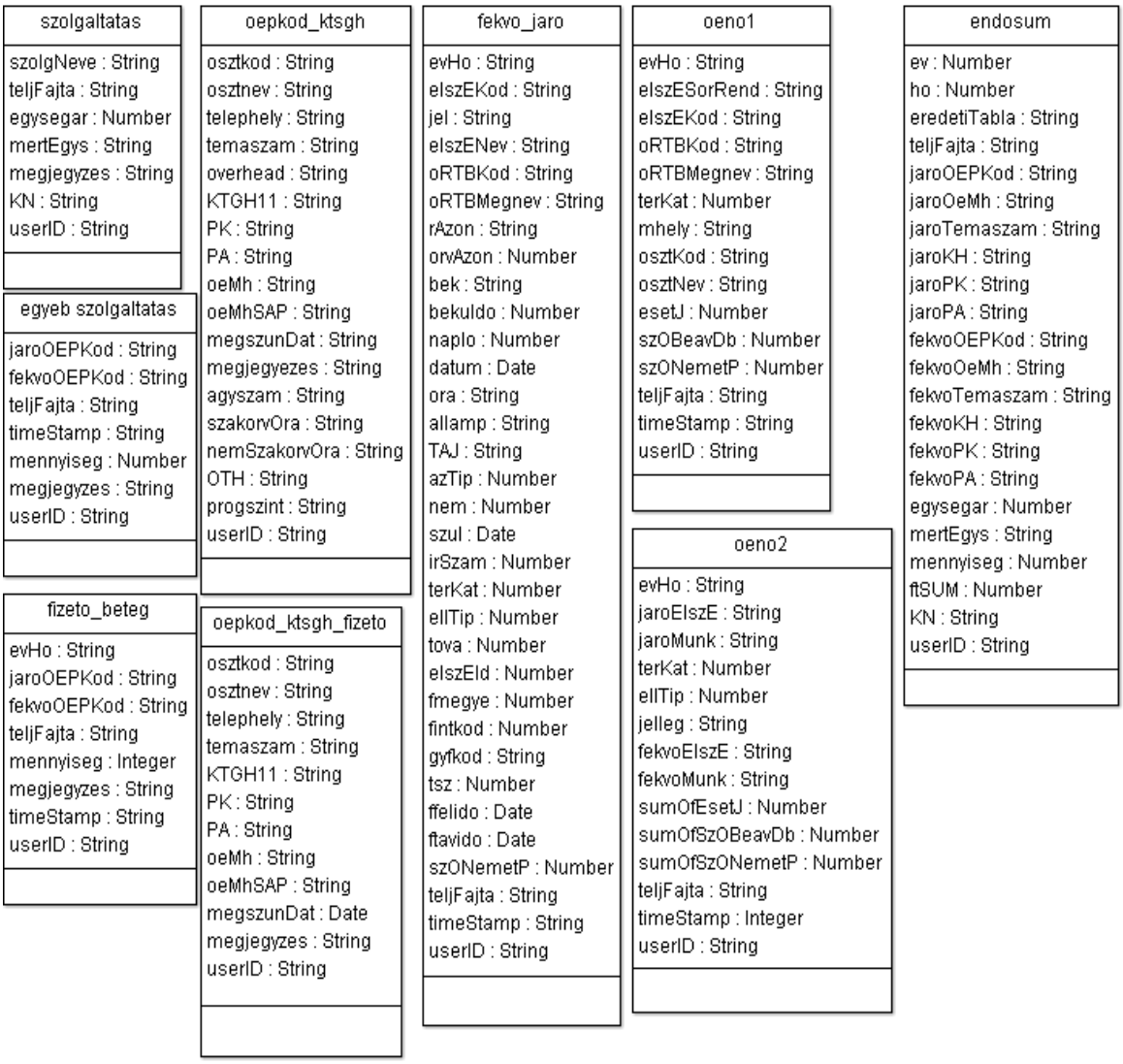
A fő adatbázis fájl a ”server\config\database.js” útvonalon található. Ez a fájl a felelős az adatbázis kapcsolódásért, az állapotok felhasználóknak szánt üzeneteiért, a modellek globális név definíciójáért és a sémák összegyűjtéséért.

A kapcsolódás a következőképp néz ki:

```
var mongoose = require('mongoose');
global.DATABASE_ADDRESS = 'mongodb://localhost/EBIG';
mongoose.connect(global.DATABASE_ADDRESS);
```

Szükséges behúzni a mongoose-t, hogy a műveleteit elérjük. A ”connect” parancs segítségével lehet kapcsolódni az adatbázishoz. A dinamikusság megtartása érdekében az adatbázis címét külön változóban tároljuk, így annak a változtatása nem igényel sok műveletet. A kódban jól látható, hogy az adatbázis neve ”EBIG” lesz.

3.10. EBIG Domain



3.2. EBIG Domain

A 3.2. ábrán látható az EBIG domain. Ezek a modellek találhatóak a projekt szerver oldalán, belőlük dolgozik minden művelet. MongoDB adatbázis révén nincsenek közöttük kapcsolatok. Az endosum tábla egy összesítő, ez alapján készülnek a fontosabb lekérdezések és kimutatások. Fontos megjegyezni, hogy a fekvő és a járó felosztás jelenti azt, hogy melyik osztály fizet melyiknek és mennyit. Az ”ev” és a ”ho” mezői segítik a grafikonok havi lebontásához szükséges szűrést.

3.11. Adatbázisban tábla sémák létrehozása

A Mongoose-ban minden a sémák elkészítésével kezdődik, minden séma MongoDB gyűjteményekben tárol és definiálja a dokumentumok formáját az adott gyűjteményben.

Az adatbázis sémákat a rendszer szerver oldalán hoztam létre, ez a kliens oldali létrehozással szemben nagyobb védelmet nyújt. Az alkalmazáson belül a ”server\belsoelszamolas\endosumdata” mappaszerkezetben van egy ”static” és egy ”dynamic” mappa. Ebben a két mappában feltöltési logika szerint vannak elrendezve a sémák. A statikus sémák közé tartoznak az oepkod_ktsgh, az oepkod_ktsgh_fizeto és a szolgaltatas. Ezek a sémák, hála a MongoDB skálázhatóságának, több ezer soros CSV fájlokból fogják adataikat kapni és feltölteni a modelljeikbe. Azért tartalmaznak ilyen sok adatok, mert itt a rendszer több éves adatokat kap visszamenőleg, amit nem kell szerkeszteni. A dinamikus sémák értelem szerűen a többi, még nem említett fájl. Ez külön dialógusablakokat kaptak, amivel adataikat szerkeszteni és bővíteni lehet.

A fizeto_beteg sémán bemutatom, milyen lépések is szükségesek ezeknek a fájloknak a létrehozásához. Először is egy mongoose példányt kellett létrehozni, ebbe importáltam magát a mongoose modult.

Másodszor a fizetőbeteg séma változót hoztam létre, ez segít majd a modell elkészítésében, amiben egy új sémát definiáltam a feladatban kívánt adatokkal, egyszerű szám és szöveg típusú adatok kerülnek bele. Egyedül a teljFajta változónál volt az a kikötés, hogy amennyiben a későbbiekben nem kerülne oda adat, úgy ’Fizetőbeteg’ szöveg kerüljön alapértelmezetten abba a mezőbe. Ez könnyen megoldható a következő módon: {type: vasztott_tipus, default: ’default_ertek’}. Fontos megjegyezni, hogy csakis azok az adatok kerülnek elmentésre a Mongoose-ban, amik ráillenek a séma formájára, a mongo-tól eltérően itt nem lehet csak úgy egy új mezőt beszúrni, ez a szigorúbb séma felhasználás egy bizonyos védelmet nyújt az adatbázis számára a kéretlen adatokkal szemben.

Harmadszor a Mongoose alapértelmezésben a modell nevét adja a gyűjtemény nevének az utils.toCollectionName metódusnak átadva azt, de meg is lehet változtatni, amennyiben mást használnánk. Mi így is tettünk, egy globális változót definiáltunk ennek a névnek, ami az adatbázis JS fájljában (database.js) kap értéket. Ezen a néven fog létrejönni a későbbi feltöltéskor a gyűjtemény, jelen példában a `{collection: global.FIZETO_BETEG_SCHEMA_NAME}` által behelyettesített név a ”fizeto_beteg” lesz

Végül a modellek létrehozása következett. A modellek a mi sémadefiníciónkból létrehozott konstruktorok. Ezek példányai ábrázolják a dokumentumokat, amikkel adatbázis műveleteket végezhetünk.

A séma létrehozásának forráskódja:

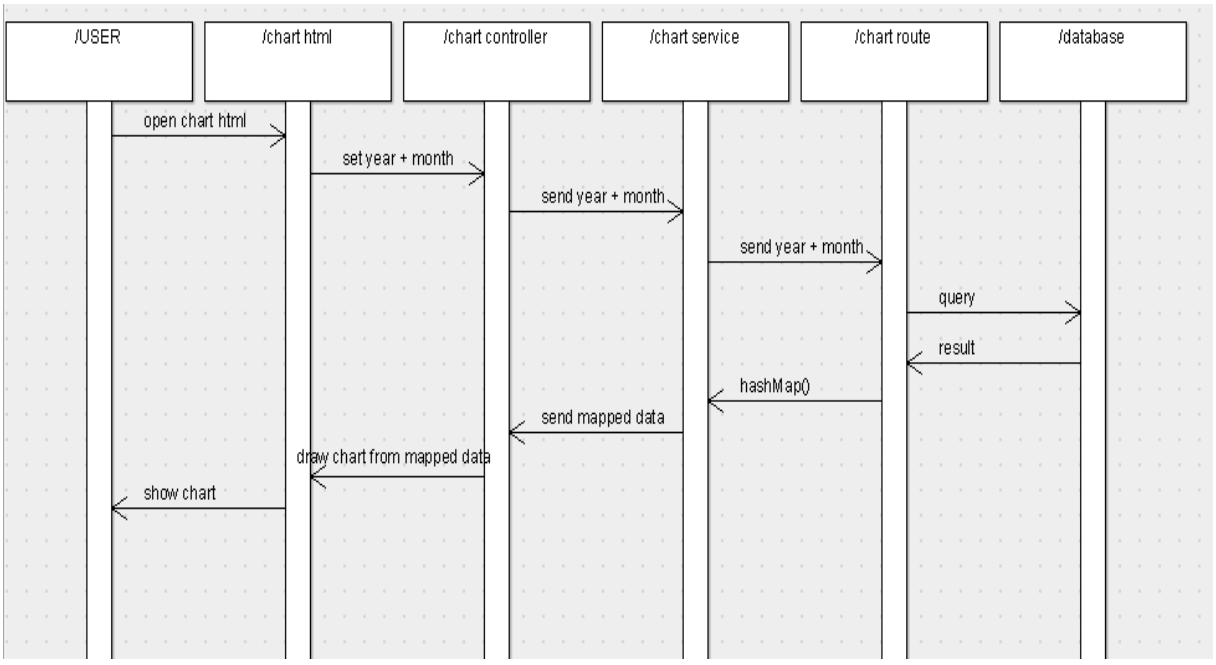
```
var mongoose = require('mongoose');
```

```
var payPatientSchema = new mongoose.Schema({
    evHo: String,
    jaroOEPKod: String,
    fekvoOEPKod: String,
    teljFajta: {type: String, default: 'Fizetőbeteg'},
    mennyiseg: Number,
    megjegyzes: String,
    timeStamp: String,
    userID: String
}, {collection: global.FIZETO_BETEG_SCHEMA_NAME});

mongoose.model(global.FIZETO_BETEG_SCHEMA_NAME, payPatientSchema);
```

Mongoose-al könnyen lehet dokumentumokban keresni, mivel támogatja a MongoDB gazdag lekérdezési szintaxisával. A find, findOne, findById és a where metódusokat használva lekérdezhetünk bármit egy dokumentumból. Fontos megjegyezni, hogy a find és a findOne között nem csak a találat mennyiség különbözik, hanem a visszaadott érték típusa is. Míg a find tömböt ad vissza, addig a findOne egy objektumot.

A többi sémát is ugyan ezen a módon hoztam létre. Ezekkel a később beszúrandó adatok kerete elkészült, a gyűjteményeket parse-olás segítségével töltöttük fel, amihez a szükséges adatokat a feltöltött CSV fájlokból kapunk meg.

3.12. Grafikon elkészítés szekvencia diagramon ábrázolva



3.4. ábra Szekvencia diagram a grafikon létrejöveteléről

A 3.4. ábrán, szekvencia diagramon ábrázoltam, milyen lépések is kellenek addig, amíg a felhasználó megtekintheti a kiválasztott év és hónap alapján az összesítő grafikonokat. Kliens oldalon található a ”public\charts” mappa.

3.13. Endosum adatainak hashmappelése

A két legkomplexebb feladatom volt a diagramok készítése. Ezek a ”chartdata.route.js” és ”chart2data.route.js”-ben találhatóak.

Ezek közül az első az ”Endosum kiadás/bevétel grafikon” előállítása (”chartdata.route.js”). Itt témaszám alapján kellett ábrázolni a kiadásokat és a bevételeket. Egy olyan oszlop diagram kirajzolása volt a feladat, ahol az X tengelyen a témaszámok, az Y a forintértékek láthatóak választható havi lebontásban, a rendszeres kimutatások támogatására. Egy témaszámhoz tartozó osztály adhat és kaphat is pénzt (tehát lehet bevétele is és kiadása is), attól függően, hogy a száma fekvőként vagy járóként szerepel az Endosum-ban. Itt összegezni kellett a témaszámokat (fekvő és járó is) és a fekvő lesz a bevétel, a járó a kiadás.

A diagram megjelenítése előtt az adatok helyes csoportosításával kell foglalkozni. Ehhez volt segítségemre a HashMap, hogy kulcs/érték párokat alkossak. A kulcsok a csoportosítandó témaszámok, hiszen ehhez tartozik több, jelen esetben két érték, az adott témaszámhoz tartozó összes bevétel és kiadás.

A függőségek behúzása után elkezdtem a fejlesztést. Első lépésként egy router példány GET hívását valósítottam meg, hogy ha erre az url-re lépünk, akkor menjen végbe egy adatbázis szűrés az Endosum modellben a kért évvel és hónappal. Ez az eredmény a data változóba kerül. Továbbá a következő funkció eredményét is majd ez a hívás fogja tovább küldeni.

```
router.get('/topicNumberIO', function (req, res, next) {
    var year = req.query.year;
    var month = req.query.month;

    endoSumModel.find({ev: year, ho: month})
        .exec(function (error, data) {
            if (error) {
                return next(error);
            }
            hashMap(data, function (map) {
                res.status(200).send({map: map});
            });
```

```
        });
});
```

A hashmappelő funkciónak a beszédes hashMap nevet adtam, ami paraméterként megkapja az előző lekérdezés eredményét a data változóban. A kapott változóban indexelhető objektumok vannak, így egy ciklussal a kulcsokat azonnal lehet is a map-be helyezni és értékét növelni az azonos objektum összegével. Amennyiben a map-ben már létezett egy témaszám, majd újra az következett, abban az esetben a kulcsot nem adta újra a map-hez, csak a hozzá tartozó értéket növelte.

```
for (i = 0; i < length; i++) {
    tmpIncome = 0;
    tmpOutcome = 0;

    var jaroTemaszam = data[i].jaroTemaszam;
    if (outcomeMap.has(jaroTemaszam)) {
        tmpOutcome += outcomeMap.get(jaroTemaszam);
    }
    outcomeMap.set(jaroTemaszam, tmpOutcome + data[i].ftSUM);

    var fekvoTemaszam = data[i].fekvoTemaszam;
    if (incomeMap.has(fekvoTemaszam)) {
        tmpIncome += incomeMap.get(fekvoTemaszam);
    }
    incomeMap.set(fekvoTemaszam, tmpIncome + data[i].ftSUM);
}
```

Végül, hogy minden kulcs a megfelelő helyen legyen és a hozzájuk tartozó értékek ne keveredjenek össze, a kulcsokat egy objektumba tettem, majd egy újabb ciklussal a már biztosan helyes sorrendű kulcsokkal az objektumhoz rendeltem az értékeket is, és ha egy adott indexen nincs bevétel, vagy kiadás, akkor nullát szúrok a helyére a hibák megelőzése érdekében. Finomításként, mivel a JavaScript a számokat lebegőpontosan jeleníti meg, beszúrás előtt egészre kerekítettem a Math.round függvénnyel.

```
for (var key in keys) {
    if (keys.hasOwnProperty(key)) {
        categories.push(key);
        if (outcomeMap.has(key)) {
            outcome.push(Math.round(outcomeMap.get(key)));
        } else {
            outcome.push(0);
        }
```

```
            if (incomeMap.has(key)) {
                income.push(Math.round(incomeMap.get(key)));
            } else {
                income.push(0);
            }
        }
    }
```

Ezzel elkészült a struktúra, egy result objektumban callback-ként visszaadom a már fentebb említett GET hívásnak.

```
result= {
    categories: categories,
    outcome: outcome,
    income: income
};
callback(result, data);
```

A második grafikon logikája is hasonlóképpen épül fel (”chart2data.route”), csak itt ”oeMh_SAP” alapján összegzi a klinikák kiadását és bevételét. Az ”oeMh_SAP” az ”oepkod_ktsgh” táblában található, osztálykód alapján ki kell gyűjteni havonta a kiadásokat és a bevételeket és ezeket kell az ”oeMh_SAP” alapján csoportosítani. Tehát itt kétszer kell rendszerezni az adatokat, dupla map-et hozok létre.

Az első mappelés pontosan az előzővel egyezik meg, csak témaszám helyett osztálykód alapján lesznek a kulcsok és az értékek szétosztva.

Ezután a funkciók futási sorrendjének fontossága miatt egy syncloop nevű vezérlésfolyamot vezetek be, ami lényegében úgy működik, mint az async waterfall.

Újabb lekérdezésekre volt szükség, hogy a már megkapott osztálykódokkal az Endosum táblából az oepkód költséghely táblának a megfelelő objektumait nyerjem ki. Az osztálykód csak része az oep költséghelyben lévő kódnak, így újra reguláris kifejezéssel kellett bővíteni a stringet. Továbbá itt már a kulcsok értékei nem csak két számot tartalmaznak

Ez a példakód a syncLoop kezdetét tartalmazza, egy lekérdezést, és a loop.next parancsot, ami a következő funkcióba dobja a futást.

```
var categoriesLength = categories.length;
syncLoop(categoriesLength, function (loop) {

    var i = loop.iteration();

    ktsghModel.findOne({osztkod: {$regex: new RegExp('.*' +
categories[i] + '.*')}}, function (error, ktghTalalat) {
```

```
        if (ktghTalalat) {
            var osztnev = ktghTalalat.osztnev;
            var oemhSAPnev = ktghTalalat.oeMhSAP;

            if (oepMapKtgh.has(oemhSAPnev)) {

oepMapKtgh.get(oemhSAPnev).sumOsztkod.push(categories[i]);
            } else {
                oepMapKtgh.set(oemhSAPnev, {osztnev: osztnev,
sumOsztkod: [categories[i]], income: 0, outcome: 0});
            }
            loop.next();
        }
```

Végül ennél a műveletnél is megcsinálom az előző feladat lépéseit az új adatokkal.

3.14. Grafikonok megjelenítése

A 2.7. pontban már kifejtésre került Highcharts grafikonmegjelenítő könyvtárra esett a választás ezeknek a GET hívással tovább küldött adatok megjelenítéséhez.

A public/charts mappában találhatóak a szükséges fájlok ehhez a művelethez. Mind a kettő grafikon egy-egy almappával rendelkezik, mindkettőben megtalálható a megjelenítő html és a logikát tartalmazó js fájl, valamit itt helyezkedik el még a ”chart.module.js” és a ”chart.service.js”.

A ”chart.service.js” fogadja a chartdata.route.js és a chart2data.route.js result objektumait. Tartalmaz egy ”initChart” funkciót, ami a diagramok keretét tartalmazzák, ebbe csak a saját ”controller.js” fájljukban behelyettesítem a megkapott eredményeket és a diagramok kirajzolódnak. A html oldalra bekerült egy dátum választó is, aminek az értékei alapján történik az egész szűrés, melyik hónap kimutatása jelenjen meg. Ezek választott értékeit localstorage-ben tárolom, hogy az oldalról való ellépés következtében ne újra az aktuális hónapból töltődjenek vissza a kimutatások.

```
var selectedYear =
localStorageService.get(localStorageSettings.incomeoutcomePrefix +
localStorageSettings.selectedYear);
if (selectedYear) {
    vm.selectedYear = selectedYear;
}

var selectedMonth =
localStorageService.get(localStorageSettings.incomeoutcomePrefix +
```

```
localStorageSettings.selectedMonth);
if (selectedMonth) {
    vm.selectedMonth = selectedMonth;
}
```

Ezt a két változót egy ”watchGroup” nevű scope-pal figyelem meg, és amennyiben valamelyik érték változik, úgy újra töltődik az egész oldalt. Ilyenkor a szekvencia diagramon ábrázolt műveletsor újra lefut, és az új dátumnak megfelelő kimutatás kerül ki a weblapra.

```
$scope.$watchGroup(['vm.selectedYear', 'vm.selectedMonth'],
function () {
    setLocalStorageData();
    endosumIncomeOutcome();
});
```

A chartObj objektumba a minden diagramnál megegyező tulajdonságok kerültek be, amíg az alatta lévő sorok csak specifikusan az oszlop diagramnál használandóak.

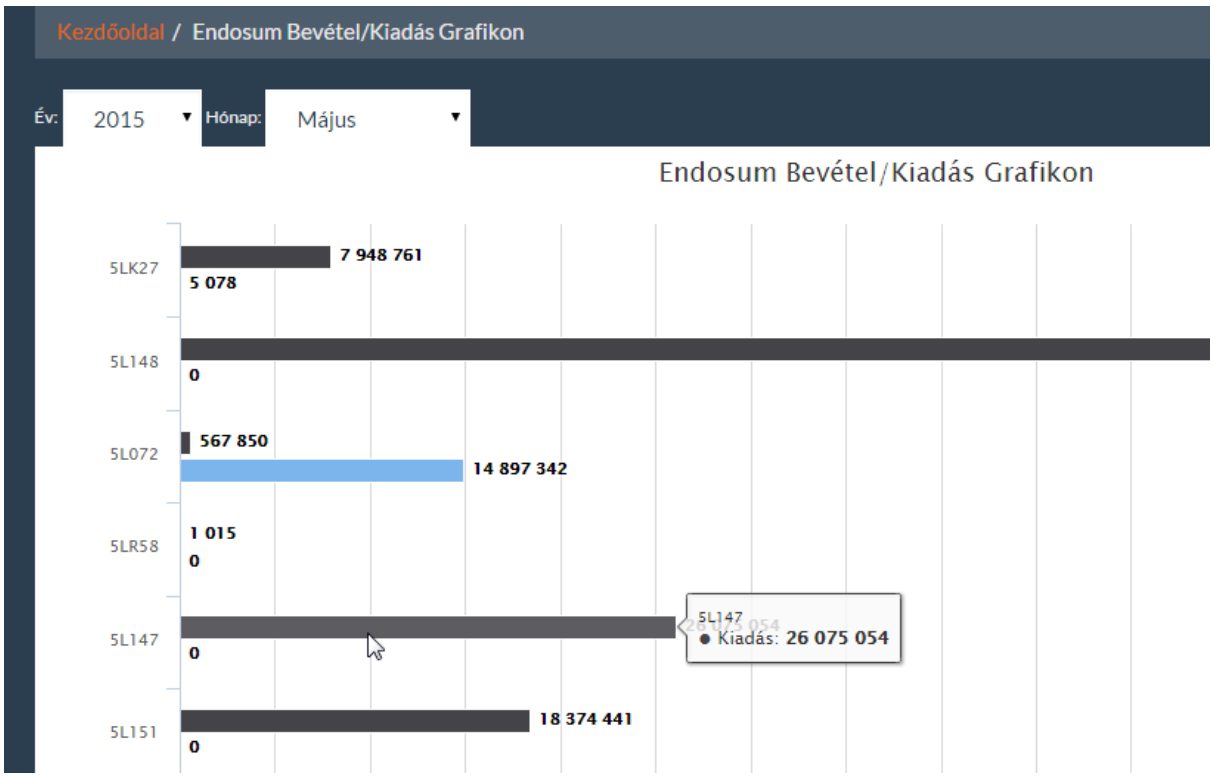
```
function endosumIncomeOutcome() {
    chartService.endosumIncomeOutcome(vm.selectedYear,
vm.selectedMonth)
        .then(function (data) {

            var chartObj = {
                type: 'bar',
                title: 'Endosum Bevétel/Kiadás Grafikon',
                xAxis: {
                    title: 'OEPTémaszám'
                },
                yAxis: {
                    title: 'Bevételek és Kiadások'
                },
                series: [{
                    name: 'Bevétel',
                    data: data.map.income
                }, {
                    name: 'Kiadás',
                    data: data.map.outcome
                }]
            };
            vm.chartConfig = chartService.initChart(chartObj);
            vm.chartConfig.xAxis.categories = data.map.categories;
            vm.chartConfig.options.chart.height =
65*data.map.categories.length;
```

```
            vm.chartConfig.options.plotOptions = {series:
{dataLabels: {align: 'left', enabled: true}}};
```

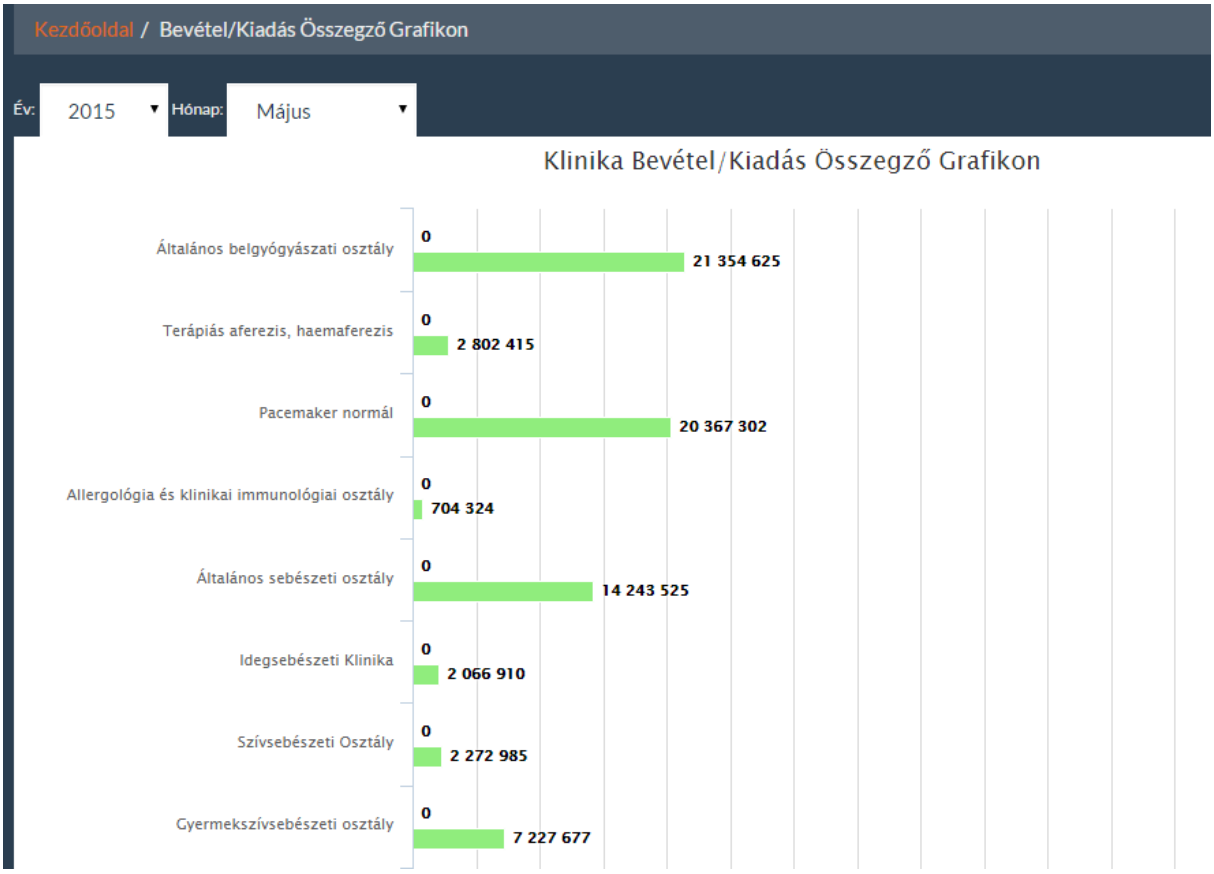
Az index.html-be be kell húzni a szükséges JS blokkokat, hogy megjelenhessenek a grafikonok:

```
<script src="../bower_components/highcharts-
release/highcharts.src.js"></script>
<script src="../bower_components/highcharts-ng/dist/highcharts-
ng.js"></script>
<script
src="http://code.highcharts.com/modules/exporting.js"></script>
```

A két végleges grafikon az éles rendszerben a 3.5. és 3.6. ábrán látható módon készült el:



3.5. – Első grafikon



3.6. – Második grafikon

4. Összefoglaló

4.1.Eddig elért eredmények

A döntéstámogató szoftver elkészült, a megrendelő által kért feladatok mind bekerültek. Egy modern webalkalmazáshoz ma már szükségszerű a reszponzív dizájn kialakítása, melyet mi sikeresen implementáltunk. Könnyen kezelhető, szép arculatot kapott az alkalmazás, aminek a működését a klinika munkatársai gyorsan el tudják sajátítani. A tesztek alapján a lekérdezések helyesek, a grafikonokon a megfelelő adatok jelennek meg és hála a megjelenítő modul informatív kialakításnak, könnyűszerrel megtudhatunk bármilyen szükséges adatot a kimutatásokhoz. Az adatbázis-modell megfelelően működik, skálázhatósága miatt a robosztus adatmennyiséggel rendelkező fájlok sem okoznak gondot.

4.2. Továbbfejlesztési lehetőségek

A MEAN szoftver csomag egy könnyen elsajátítható, rugalmas környezetet biztosít minden fejlesztő számára, ami elősegíti az eredményes munkát. Az AngularJS és a NodeJS rengeteg hasznos modullal rendelkeznek. A MongoDB skálázhatóságának köszönhetően nagy mennyiségű új adattal lehet bővíteni az alkalmazást minden gond nélkül. Amennyiben a megrendelőnek további ötletei vannak, mivel bővítené még az alkalmazását, úgy minden gond nélkül folytatódhat a fejlesztés, beilleszthetőek a változtatásaik.

Irodalomjegyzék

[1] ACL
https://www.npmjs.com/package/acl

[2] AngularJS
https://www.npmjs.com/package/angular/

[3] Atlassian JIRA
https://www.atlassian.com/software/jira

[4] Bower
http://bower.io/

[5] Express
http://expressjs.com/

[6] HashMap
https://www.npmjs.com/package/hashmap

[7] Highcharts
http://www.highcharts.com/products/highcharts

[8] Integrált fejlesztői környezet
https://hu.wikipedia.org/wiki/Integr%C3%A1lt_fejleszt%C5%91i_k%C3%B6rnyezet

[9] JavaScript
http://www.w3schools.com/js/js_intro.asp

[10] Kétirányú kötés illusztráció
https://docs.angularjs.org/guide/databinding

[11] MEAN illusztráció
http://garywoodfine.com/what-is-the-mean-stack/

[12] MEAN
https://en.wikipedia.org/wiki/MEAN_(software_bundle)

[13] MongooseDB
http://mongoosejs.com/docs

[14] MVC illusztráció
http://imgarcade.com/1/angular-js-mvc/

[15] MVC
http://www.tutorialspoint.com/angularjs/angularjs_mvc_architecture.htm

[16] Node.js
https://nodejs.org/en/

[17] NoSQL – MongoDB
https://docs.mongodb.org/

[18] NoSQL
https://en.wikipedia.org/wiki/NoSQL

[19] NPM
https://www.npmjs.com/

[20] Scrum
https://hu.wikipedia.org/wiki/Scrum

[21] Scrum illusztráció
https://www.emaze.com/@AIOTOOQL/Agile-Software-Development

[22] SourceTree
https://www.sourcetreeapp.com/

[23] Verziókezelők
https://hu.wikipedia.org/wiki/Verzi%C3%B3kezel%C3%A9s

[24] WebStorm
https://www.jetbrains.com/webstorm/

Nyilatkozat

Alulírott Hajas Tamás gazdaságinformatikus BSc. szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Tanszékcsoport Szoftverfejlesztés Tanszékén készítettem, gazdaságinformatikus BSc. diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Tanszékcsoport könyvtárában, a helyben olvasható könyvek között helyezik el.

2015. november 3.

Aláírás

Köszönetnyilvánítás

Köszönöm Dr. Bilicki Vilmosnak a szakdolgozat elkészítéséhez adott lehetőséget.
Sárosi Árpádnak és Füle Győzőnek a munkám folyamatos támogatását és figyelemmel követését.
Szénási Anitának, Hunyadi Zsombornak, Szabó Zoltánnak, Kiss Dánielnek és Tóth Péternek a munka elejétől a legvégéig történő segítséget és jó kedvet.