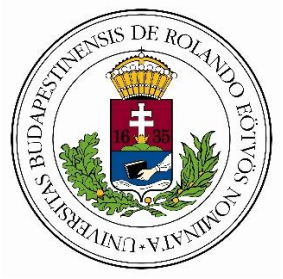


Eötvös Loránd Tudományegyetem

Informatikai Kar

Média- és Oktatásinformatika Tanszék

Webes alkalmazás társasházak közüzemi fogyasztásainak adminisztrálására

Dr. Horváth Győző

egyetemi adjunktus

Soós Bálint

programtervező informatikus Bsc

Budapest, 2018

Tartalomjegyzék

1. fejezet - Bevezetés

1.1 Motiváció

Újépítésű társasházakban egyre jellemzőbb a központi kazán, amellyel a lakások fűtését és melegvízellátását biztosítják. Ilyen rendszer mellett a lakók fogyasztásainak adminisztrálása a közös képviselő feladata lesz, azonban ennek kivitelezése a lakóknak és a közös képviselőnek is rengeteg odafigyelést igényel és komoly terhet jelent.

Saját helyzetemből kiindulva, az én társasházamban - mely 96 lakásból áll - minden hónap végén egy papírlapot függesztenek ki a lépcsőházban, amelyre minden lakónak kézzel kell beírni a hőmennyiségmérő és a vízórák állását. Ezt a papírlapot a bejelentési időszak végeztével a közös képviselő elviszi és kézzel viszi fel az adatokat egy Excel táblába, ahol kiszámolja a havi fűtés és vízmelegítés díját. A számlák kitöltése szintén kézzel történik, amiket ezután a postaládába dobva kézbesítenek a lakóknak. Ez a rendszer lassú, és a legtöbb lépésben ott van az emberi hibalehetőség is. Ennek a rendszernek a kiváltására szeretnék tervezni egy webes alkalmazást, amely a lépések automatizálásával kiküszöbölheti az emberi tényezőt és kényelmes felületet nyújthat a társasház lakóinak és a közös képviselőnek.

1.2 Felhasználási igények

Az alkalmazásnak két fő felhasználási igényt kell kielégíteni. A lakók számára egy könnyen átlátható felületet kell biztosítani, ahol feltölthetik a fogyasztásmérők állását, illetve megtekinthetik a kapott számlákat. A közös képviselőnek egy adminisztrációs felületet kell biztosítani, ahol láthatja a fogyasztási adatokat, és ezek alapján számlákat tud kiállítani, majd ezeket elküldeni a lakóknak.

2. fejezet - Felhasználói dokumentáció

2.1 Követelmények

Az alkalmazás elérhető a https://app-rezsi.herokuapp.com/ címen. Használatához szükség van egy JavaScript futtatására alkalmas webböngészőre, internet hozzáférésre és email címre. Az alkalmazás csak angol nyelven érhető el, így szükség van minimális angol nyelvtudásra is.

2.1.1 Támogatott böngészők

- Google Chrome (v62.x és későbbi verziók)
- Google Chrome for Android (v62.x és későbbi verziók)

2.2 Alkalmazás használata

2.2.1 Alkalmazás használata közös képviselők számára

2.2.1.1 Regisztráció

Az alkalmazás használatához rendelkeznünk kell egy regisztrált fiókkal, amit a Sign Up (Regisztrálás) aloldalon hozhatunk létre. Meg kell adnunk az email címünket, felhasználó nevünket és a jelszavunkat. Sikeres regisztráció esetén a megadott email címre egy visszaigazoló email fog érkezni. Az emailben található gombra kattintva erősítsük meg, hogy valódi címet adtunk meg.

Sign Up screenshot

2.2.1.2 Bejelentkezés

Az alkalmazásba való belépéshez navigáljunk a Login (Bejelentkezés) aloldalra és írjuk be a regisztrációkor megadott email címet és jelszót.

Login screenshot

2.2.1.3 Kijelentkezés

Az alkalmazás jobb felső sarkában található menü ikonra kattintva a felugró menüben láthatjuk az aktuálisan bejelentkezett felhasználó nevét és email címét. A Sign Out (Kijelentkezés) gombra kattintva kijelentkezhetünk az alkalmazásból.

Logout screenshot

2.2.1.4 Új csoport létrehozása

Jövőbeli felhasználóinkat (lakóinkat) saját preferenciáink szerint csoportokba oszthatjuk. A csoportosítás módja a közös képviselőre van bízva, ez lehet például épület, lépcsőház, vagy akár szint alapján. Egy lakó csak egy csoport tagja lehet. Új csoport létrehozásához a csoport nevét kell megadni.

Groups screenshot

Create group screenshot

2.2.1.5 Csoport szerkesztése és törlése

A már létező csoportok nevét átírhatjuk, vagy akár törölhetjük az egész csoportot a hozzá tartozó felhasználókkal együtt. Ebben az esetben a törölt felhasználók többé nem lesznek képesek bejelentkezni az alkalmazásba.

Group options screenshot

Edit group screenshot

Delete group screenshot

2.2.1.6 Lakók meghívása a csoportba

Új lakók meghívásához kattintsunk az adott csoport lakóinak listájánál az Invite (Meghívás) gombra. A megjelenő párbeszédablakban található linken keresztül tudnak regisztrálni a lakók a csoportunkba. Ez a link teljesen publikus, így fokozottan figyeljünk oda, hogy kinek küldjük el.

Invite dialog screenshot

2.2.1.7 Lakó törlése

A csoportba felvett lakókat egyesével törölhetjük. Ehhez kattintsunk a lakók listájában az adott lakó mellett található opciók ikonra, majd a Delete (Törlés) gombra. A törölt lakók többé már nem lesznek képesek bejelentkezni az alkalmazásba. Újrafelvételükhöz újra regisztrálni kell.

2.2.1.8 Csoportos számla kiállítása

A csoport összes tagja számára egyszerre tudunk kiállítani számlát. A csoport számláinak listájánál kattintsunk a Create bill (Számla készítése) gombra. A felugró párbeszédablakban meg kell adnunk a számlázási időszak kezdetének dátumát, a számlázási időszak végének dátumát, a pénznemet, a fűtés egységárát pénznem/kWh-ban, a melegvíz egységárát pénznem/köbméterben, és a hidegvíz egységárát pénznem/köbméterben. A számla készítéséről minden érintett felhasználó értesítést kap.

2.2.1.9 Csoportos számla adatainak letöltése

A kiállított csoportos számla adatait letölthetjük CSV formátumban. Ehhez a csoport számláinak listájában a számla alatti Download as CSV (Letöltés CSV-be) gombra kell kattintani. A fájl egy sora tartalmazza a csoporthoz tartozó lakó nevét, email címét, az adott időszakban felhasznált fűtés mennyiségét kWh-ban, a fűtés egységárát pénznem/kWh-ban, fűtés összdíját az adott pénznemben, melegvíz fogyasztást köbméterben, melegvíz egységárát pénznem/köbméterben, melegvíz összdíját, hidegvíz fogyasztást köbméterben, hidegvíz egységárát pénznem/köbméterben, hidegvíz összdíját, a megadott pénznemet és a végösszeget.

2.2.1.10 Lakó fogyasztási adatainak és számláinak megtekintése

A csoport lakóinak listájánál, ha rákattintunk egy adott lakóra, akkor megtekinthetjük az adott lakó összes fogyasztási bejelentését és az összes kiállított számláját. A bejelentések tartalmazzák a bejelentés napjának dátumát, és a fogyasztásmérők állását: a hőmennyiséget kWh-ban és a meleg-, illetve hidegvíz mennyiségét köbméterben. A számlák tartalmazzák a kiállítás dátumát, a

számlázási időszak kezdetének és végének dátumát, illetve egy táblázatban lebontva a fogyasztások díjait a végösszeggel együtt.

2.2.1.11 Lakó számlájának letöltése

A lakók kiállított számláit szabadon letölthetjük PDF formátumban. Ehhez kattintsunk a lakó számláinak listájában a számla alatt található Download as PDF (Letöltés PDF-be) gombra. A letöltött fájl tartalmazza a számla azonosítószámát, kiállításának dátumát, a számlázási időszak kezdetének és végének dátumát, illetve egy táblázatban lebontva a fogyasztások díjait a végösszeggel együtt.

2.2.2 Alkalmazás használata lakók számára

2.2.2.1 Regisztráció

Az alkalmazás használatához rendelkeznünk kell egy regisztrált fiókkal, amit a közös képviselőtől kapott linken keresztül tehetünk meg. Meg kell adnunk az email címünket, felhasználó nevünket és a jelszónkat. Ajánlott a valódi nevünket használni, hiszen a közös képviselő így tud a legegyszerűbben beazonosítani minket. Sikeres regisztráció esetén a megadott email címre egy visszaigazoló email fog érkezni. Az emailben található gombra kattintva erősítsük meg, hogy valódi címet adtunk meg.

Sign Up screenshot

2.2.2.2 Bejelentkezés

Az alkalmazásba való belépéshez navigáljunk a Login (Bejelentkezés) aloldalra és írjuk be a regisztrációkor megadott email címet és jelszót.

Login screenshot

2.2.2.3 Kijelentkezés

Az alkalmazás jobb felső sarkában található menü ikonra kattintva a felugró menüben láthatjuk az aktuálisan bejelentkezett felhasználó nevét és email címét. A Sign Out (Kijelentkezés) gombra kattintva kijelentkezhetünk az alkalmazásból.

Logout screenshot

2.2.2.4 Új bejelentés készítése

Új bejelentés készítéséhez kattintsunk a bejelentések listájánál a Create report (Bejelentés készítése) gombra. A felugró párbeszédablakban adjuk meg a hőmennyiségmérő állását kWh-ban és a meleg-, illetve hidegvíz mérőórájának állását köbméterben.

2.2.2.5 Értesítés számla kiállításáról

Minden új számla kiállításáról a felhasználó email-ben és alkalmazáson belül is értesítést kap. Az alkalmazás jobb felső sarkában található harang ikonra kattintva megtekinthetjük új értesítéseinket, amelyeknek számát a harang mellett található buborékban is jelezzük. A számla adatainak megtekintéséhez kattintsunk az értesítésre.

2.2.2.6 Számla letöltése

A kiállított számlákat szabadon letölthetjük PDF formátumban a számla alatt található Download as PDF (Letöltés PDF-be) gombra kattintva. A letöltött fájl tartalmazza a számla azonosítószámát, kiállításának dátumát, a számlázási időszak kezdetének és végének dátumát, illetve egy táblázatban lebontva a fogyasztások díjait a végösszeggel együtt.

3. fejezet - Fejlesztői dokumentáció

3.1 Probléma elemzése

Az alkalmazás tervezését az alapprobléma megfogalmazásával érdemes kezdeni, melyet a Motiváció pontban fejtettem ki részletesen. A papír alapú folyamat időigényes, és túl sok hibatényező áll fenn. Mi történik, ha valaki hiányosan tölti ki a papírt, vagy ha a közös képviselő hibásan rögzíti az adatokat. Ezeket mind ki tudnánk küszöbölni egy programmal. A mostani folyamat túl lassú, több nap, akár egy hét is eltelhet mire az adatokat feldolgozzák, és újabb több nap, amíg mindenkihez eljut a kiállított számla. Ez egy száz lakásból álló társasház esetén logisztikailag is megterhelő folyamat, és a közös képviselőnek akár több ilyen társasház ügyeit is intéznie kell párhuzamosan. Egy program esetén ez az egész folyamat automatizálható, és a lakókhoz a kiállítás után pillanatok alatt megérkezhet a számla.

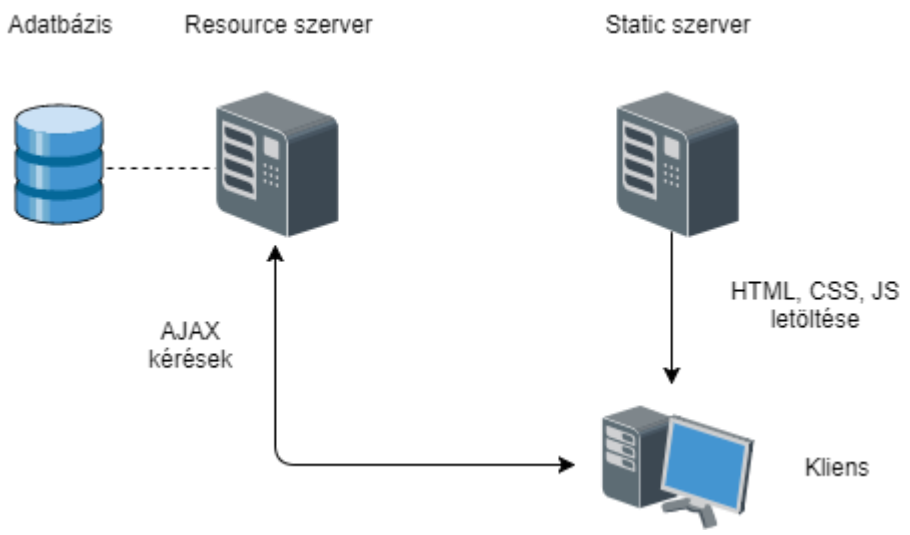
3.2 Specifikáció

Olyan alkalmazásra van szükségünk, ahova két típusú felhasználó tud bejelentkezni. A felületet és a funkcionalitást a felhasználó típusától függően el kell különíteni. A közös képviselő adminként fog funkcionálni, az alá tartozó felhasználók összes adatát láthatja. Csoportokat hozhat létre a felhasználók könnyebb áttekinthetőségének érdekében. A csoportnak nevet kell tudni adni, amit később szerkeszteni is lehessen. Lehetőséget kell adni csoportok törlésére is. A közös képviselő egy adott csoportba tudja meghívni a lakókat, illetve törölhet is a csoportba felvett lakók közül. Egy lakó csak egy csoport tagja lehet. A lakók bejelentett óraállásainak tartalmaznia kell a dátumot, a hőmennyiségmérő állását kWh-ban és a meleg-, illetve hidegvízóra állását köbméterben. A bejelentett óraállásokról összesítést kell tudni készíteni. Az összesítés készítéséhez meg kell adni a kezdő és záró dátumot, hogy melyik intervallumban vegyük az óraállásokat. Az összesítéseket letölthetővé kell tenni CSV formátumban. Az összesítés folyamatát össze lehet kötni a számlák kiállításával, ehhez ismernünk kell a hő, melegvíz és hidegvíz egységdíjait és a pénznemet. Az alkalmazásnak kell

kiszámolni a részösszegeket és a fizetendő végösszeget. Ezeket meg kell jeleníteni az alkalmazás felületén és elérhetővé kell tenni PDF formátumban is. A lakóknak értesítéseket kell küldeni új számla kiállításáról. Ezek az értesítések történhetnek email formában, vagy az alkalmazáson belüli értesítési felületen keresztül is.

3.3 Tervezés

A specifikációban megfogalmazott igényeket remekül lefedheti egy webes alkalmazás. Az alkalmazás architektúrája 3 komponensből épül fel. A resource szerver egy REST (representational state transfer) elveire épített Web API (Application Programming Interface), ami az adatbázishoz kapcsolódva a rendszer backend felületét nyújtja. A kliens alkalmazás egy Single Page Application, ami AJAX hívásokkal fog kapcsolódni a resource szerverhez. Szükségünk lesz tehát egy MVVM architektúrát megvalósító keretrendszerre. A kliens programot egy külön static szerver fogja kiszolgálni.



3-1. ábra: Az alkalmazás hálózati terve

A lakók kényelme érdekében az oldalt mobil nézettel is el kell látni, mivel a mérőórák mellett állva így egyből beírhatják az adatokat a mobilkészülékükön keresztül. A mobil felületet lehetőleg minél inkább a natív mobil alkalmazások mintájára célszerű elkészíteni, hogy a felhasználóknak ismerős legyen, és könnyebben tudják megszokni a használatát. Szükség lesz egy SMTP szerverre is, ami az email értesítések kiküldéséért lesz felelős, illetve egy valós idejű értesítési

rendszer is kell, ami az alkalmazáson belüli értesítéseket fogja kezelni. Modern és megbízható authentikációs technológiát kell választani, ugyanis itt két különböző szintű felhasználóval van dolgunk, a legtöbb funkció engedélyezése az authentikált felhasználó típusától függ.

3.4 Szerver oldali architektúra

3.4.1 RESTful API

A REST, azaz Representational State Transfer [1] egy internetes architektúra típus, amelyben a hálózatot szerverek és kliensek alkotják. A kliensek kéréseket indítanak a szerver felé, amelyre a szerver valamilyen webes erőforrással (HTML, XML, JSON) válaszol. A REST architektúra hat megkötést fogalmaz meg, ami biztosítja az alkalmazások teljesítményét, skálázhatóságát és megbízhatóságát. Ezek a megkötések a következők:

1. Kliens-szerver architektúra: A kliensek és szerverek feladatköreit el kell különíteni. Például a szerverek nem foglalkozhatnak felhasználói felülettel, vagy a kliens állapotával, és a kliensek nem foglalkozhatnak adattárolással. Ez az elv biztosítja a kliens hordozhatóságát és a szerver skálázhatóságát. A fejlesztést teljesen szétválaszthatjuk, amíg az interfész nem változik.

2. Állapotmentesség: A kommunikáció további megkötése, hogy a szerver nem tárolhatja a kliens állapotát kérések között. Minden kérésnek tartalmaznia kell elégséges információt, hogy a szerver képes legyen azt végrehajtani. A session állapotot a kliensnek kell megőrizni. Ez kifejezetten érdekes elmélet authentikációs szempontból.

3. Gyorsítótárazhatóság (Cacheability): A kliensek gyorsítótárazhatnak bizonyos válaszokat. A válaszoknak tartalmazniuk kell, hogy tárazhatóak-e, vagy sem. Egy jól felépített tárazási stratégia megnövelheti a szerver skálázhatóságát, mivel egyes kéréseket nem kell újra kiszolgálni, és így terhet vehetünk le a szerverről.

4. Rétegelt felépítés: A kliensek nem képesek megmondani, hogy direkt kapcsolódott a szerverhez, vagy közvetítő szervereken keresztül, így

terheléselosztó szerverek (load balancers) közbeiktatásával növelhetjük a skálázhatóságot.

5. Code on Demand (igényelhető kód): A szerverek ideiglenes kibővíthetik a kliens funkcionalitását futtatható programrészek elküldésével. Ezt a módszert alkalmazták a Java applet-ek, vagy a kliensoldali JavaScript szkriptek.

6. Egységes interfész: Egyszerűsíti és szétválasztja az architektúrát, ami megkönnyíti a kliensek és szerverek független fejlesztését.

A REST elveit követő webes szolgáltatásokat röviden RESTful Web Services-nek nevezzük. A kérések típusai a HTTP szabvány metódusainak felelnek meg, azaz GET, POST, PUT, PATCH, DELETE, stb. A legelterjedtebb interfész a RESTful szolgáltatásoknál a JSON (JavaScript Object Notation).

3.4.1.1 Felhasznált technológiák

Node.js

A Node.js [2] egy nyílt forráskódú, platformfüggetlen JavaScript futtatókörnyezet. A JavaScript-et történetileg a böngészők elsődleges nyelvének fejlesztették ki, de a webes szolgáltatások elterjedésével ma már az egyik legnépszerűbb programozási nyelvvé nőtte ki magát. A Node.js a Chrome böngésző JavaScript futtatómotorjára, a V8-ra épül, amely interpretálás helyett natív gépi kódra fordítja le a JavaScript forráskódot. Eseményvezérelt architektúrája és aszinkron I/O műveletek támogatása miatt jó választás lehet valós idejű, többfelhasználós alkalmazások fejlesztésére. Ebben a kategóriában népszerű példa a chat programok, amik általában nagyforgalmú, adat-intenzív, de kis számításigényű rendszerek. Hatékonysága és könnyen skálázhatósága miatt a Node.js különösen népszerű eszköz lett data API-ok implementálására is. Funkcionalitásának kiterjesztésére létrehozták az NPM (Node Package Manager) nevű csomagkezelőt, amely több mint 350 ezer csomaggal a világ legnagyobbjának számít. Fejlesztésének felügyelésére és piaci adoptálásának felgyorsítására létrehozták a Node.js Foundation-t, melynek tagjai között ott találjuk a Google, Microsoft, IBM, SAP, Intel, Redhat és a PayPal delegáltjait is.

Express.js

Az Express [3] a Node.js beépített HTTP moduljára épülő, fejlesztőbarát API, amely remek absztrakciós réteget ad webszolgáltatások fejlesztéséhez.

3.4.1.2 Végpontok

A resource szerver elérhető a https://api-rezsi.herokuapp.com/ címen. Amelyik végpontnál külön nincs megemlítve, ott JSON formátumban fogadhatunk és küldhetünk adatot. Hiba esetén az API HTTP hibakóddal válaszol. Verziózás céljából az összes végpont az `/api/v1/` előtaggal érhető el. Ennek segítségével párhuzamosan több verzió is elérhető lehet az éles környezetben, így függetlenül fejleszthetjük az új interfészeket és az arra támaszkodó klienseket.

`GET /health`: Sikeres kérés esetén a szerver HTTP 200 kóddal válaszol. Erre a végpontra egy automatizált felügyeleti szolgáltatást köthetünk, ami folyamatosan ellenőrzi az API elérhetőségét és terheltségét. Ha a szerver elérhetetlenné válik, HTTP 503 hibakóddal tér vissza.

`GET /auth`: Authentikált felhasználó adatainak lekérdezése. Sikertelen kérés esetén HTTP 401 (unauthorized) hibakóddal válaszol.

`POST /auth`: Felhasználó authentikálása email címmel és jelszóval. Sikeres kérésnél a felhasználó adatait és az authorizációs token-t küldi vissza.

`POST /users`: Új felhasználó regisztrálása. A kérésben el kell küldeni az email címet, a felhasználónevet és a jelszót.

`GET /users/:id/confirm`: Felhasználó email címének visszaigazolása. Sikeres kérés esetén a szerver átirányít a bejelentkezési felületre.

`GET /groups`: Authentikált felhasználó összes csoportjának lekérdezése. Csak közös képviselőknek elérhető.

`POST /groups`: Új csoport létrehozása az authentikált felhasználónak. Csak közös képviselőknek elérhető.

`GET /groups/:id`: Authentikált felhasználó egy csoportjának lekérdezése. Csak a csoport tulajdonosának elérhető.

`PATCH /groups/:id`: Authentikált felhasználó egy csoportjának adatainak módosítása. Csak a csoport tulajdonosának elérhető.

`DELETE /groups/:id`: Authentikált felhasználó egy csoportjának törlése. Csak a csoport tulajdonosának elérhető.

`GET /groups/:id/summaries`: Csoport összesítéseinek lekérdezése. Csak a csoport tulajdonosának elérhető.

`POST /groups/:id/summaries`: Összesítés létrehozása. Csak a csoport tulajdonosának elérhető.

`GET /groups/:id/summaries/:summaryId/csv`: Összesítés letöltése CSV formátumban. Csak a csoport tulajdonosának elérhető.

`GET /groups/:id/users`: Csoport összes felhasználójának lekérdezése. Csak a csoport tulajdonosának elérhető.

`GET /groups/:id/users/:userId`: Csoport egy felhasználójának lekérdezése. Csak a csoport tulajdonosának elérhető.

`DELETE /groups/:id/users/:userId`: Csoport egy felhasználójának törlése. Csak a csoport tulajdonosának elérhető.

`GET /groups/:id/users/:userId/reports`: Felhasználó óraállásainak lekérdezése. Csak a csoport tulajdonosának és az adott felhasználónak érhető el, feltéve, ha a csoport tagja.

`POST /groups/:id/users/:userId/reports`: Óraállás létrehozása. Csak az adott felhasználónak érhető el, feltéve, ha a csoport tagja.

`GET /groups/:id/users/:userId/bills`: Felhasználó számláinak lekérdezése. Csak a csoport tulajdonosának és az adott felhasználónak érhető el, feltéve, ha a csoport tagja.

`GET /groups/:id/users/:userId/bills/pdf`: Felhasználó számláinak letöltése PDF formátumban. Csak a csoport tulajdonosának és az adott felhasználónak érhető el, feltéve, ha a csoport tagja.

3.4.2 Adatbázis

3.4.2.1 Felhasznált technológiák

MongoDB

A MongoDB [4] egy nyílt forráskódú, NoSQL, dokumentumorientált adatbázis. Az adatokat BSON (JSON-höz hasonló) formátumban tárolja. A dokumentumok szerkezeti felépítését Schema-k segítségével definiálhatjuk. Lekérdezésekben és aggregációs függvényekben natívan használhatunk JavaScript kifejezéseket, emiatt igazán elterjedt Node.js alapú rendszerekben.

Mongoose

A Mongoose [5] egy NPM-en keresztül elérhető csomag, ami megkönnyíti a MongoDB Schema-k definiálását, validálását, és egy magasabb szintű interfészt biztosít a MongoDB dokumentumom kezeléséhez.

3.4.2.2 Modellek

User:

- email
- password
- displayName
- role
- group
- confirmed
- disabled

Group:

- name
- leader
- disabled

Report:

- hotWater
- coldWater
- heat
- user

Summary:

- from
- to
- hotWaterPrice
- coldWaterPrice
- heatPrice
- currency
- group

Bill:

- hotWaterConsumption
- coldWaterConsumption
- heatConsumption
- summary
- user

Notification:

- type
- seen
- user

- bill

3.5 Kliens oldali architektúra

3.5.1 Felhasznált technológiák

3.5.1.1 React

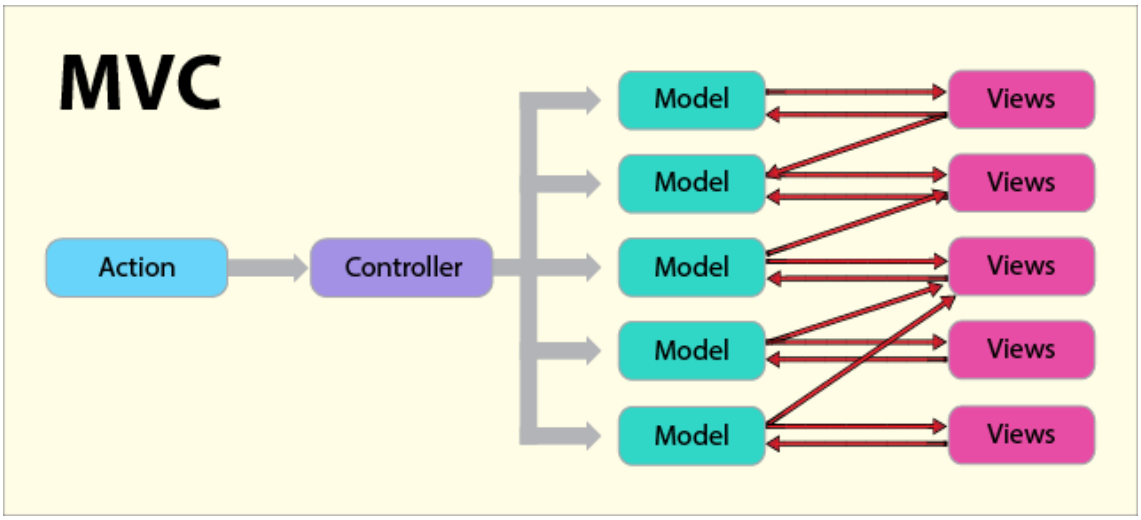
A React [6] egy nyílt forráskódú, kliens oldali JavaScript könyvtár, amelyet felhasználói felületek készítésére hoztak létre. A Facebook fejlesztette ki 2013-ban, és azóta az egyik legnépszerűbb Single Page Application könyvtárrá vált. Célja egyszerűen karbantartható, újrafelhasználható, enkapszulált komponensek létrehozása és ezekkel skálázható, gyors alkalmazások építése.

Hatékonyságát a Virtual Document Object Model-nek, azaz a virtuális DOM-nak köszönheti. Egy kliens oldali alkalmazásban az egyik legköltségesebb művelet a DOM manipulációja, azaz a képernyőn látható felület frissítése. Minél több újrarajzolási folyamatot kell végrehajtani a böngészőnek egységnyi idő alatt, annál lassabb lesz az alkalmazásunk. Ha a gyorsaságra akarunk törekedni, akkor minimalizálnunk kell a felesleges újrarajzolások számát. A virtuális DOM ezt úgy oldja meg, hogy amikor a felületen valamit dinamikusan változtatni szeretnénk, akkor a változtatás először csak a virtuális DOM-ban hajtódik végre. A React összehasonlítja a virtuális és a valódi DOM-ot és csak azokat az ágakat frissíti a valódi DOM-ban, amik különböznek. Ezzel a módszerrel komoly performancia növekedést lehet elérni.

3.5.1.2 Flux architektúra

A tipikus MVC architektúrából a React csak a View, azaz a megjelenítés rétegéért felelős, ezért egy teljes alkalmazáshoz szükségünk lesz még a Model és Controller feladatait ellátó komponensekre. Az eddig megszokott architektúrák mind magukban hordozzák azt az alapproblémát, hogy az adat rétegek nehezen skálázódnak az alkalmazásunk megjelenítési rétegével. Az utóbbi pár évben teret nyerő komponens alapú fejlesztés eredményeként egyszerre több nézet is megjelenhet a felületen és gyakori elvárás, hogy egy nézet egy másik modell változására is reagáljon. Ez láthatatlan függőségeket eredményez a komponensek

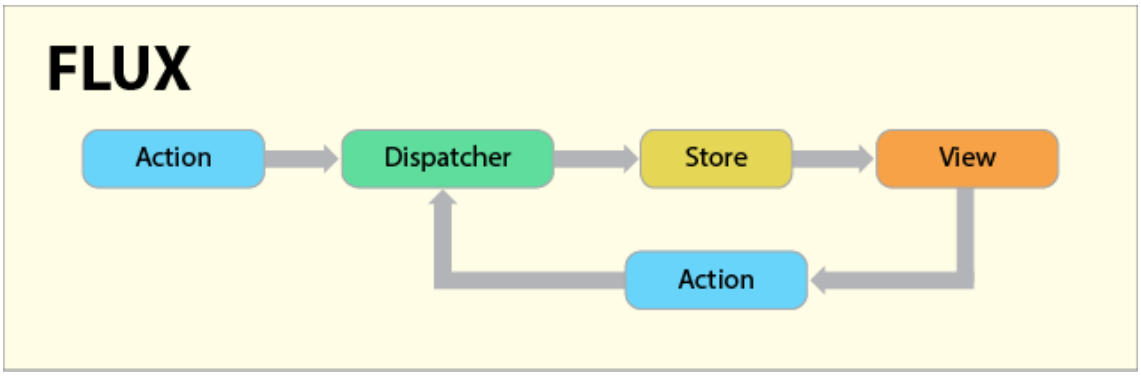
között, így nagyon hamar egy nehezen átlátható, karbantartható és tesztelhető kódbázisban találjuk magunkat.



3-2. ábra: Az MVC architektúra vázlata (Forrás: [7])

A Facebook egyre növekedő alkalmazásában egyszerűen már nem működött az MVC, ezért új megoldás után kellett nézni. Ennek eredményeképp tervezték meg a Flux-ot [7], amit inkább nevezhetünk design pattern-nek, mint kész keretrendszernek. Amíg az MVC-k általában a kétirányú adatfolyamot részesítik előnyben, addig a Flux az egyirányú adatfolyam koncepcióját követi, megkönnyítve ezzel az alkalmazás működésének átláthatóságát és a problémák észlelését. Négy fő egységből épül fel:

- Actions
- Dispatcher
- Stores
- Views (React components)



3-3. ábra: A FLUX architektúra vázlata (Forrás: [7])

Az Action egy egyszerű JavaScript objektum, amit a View, vagy egy külső hatás vált ki. Típussal szokás ellátni, ami azonosítja az Action eredetét. A Dispatcher megkapja az Action-t és típusa alapján továbbítja a Store megfelelő callback függvényének. A Store tartalmazza az alkalmazás állapotát és logikáját. Figyel a Dispatcher által továbbított Action-ökre, és módosítja az alkalmazás állapotát, ha szükséges, illetve frissíti a View-t. A View valójában a nézetet alkotó React komponensek összessége. Megkapják a Store-ból az alkalmazás aktuális állapotát, ami alapján megjelenítik a felületet és felhasználói interakcióra kiváltanak újabb Action-t. Ezzel a módszerrel teljesen eltűnnek a komponensek közötti keresztfüggőségek, minden adat egy jól átlátható, egyirányú adatfolyamban halad.

3.5.1.3 Redux

A Flux implementálására rengeteg keretrendszer jött létre, de ezek közül a Redux terjedt el. Több, különálló Store helyett egy globális, változtathatatlan (immutable) Store objektumunk van, amelynek állapottereit egy-egy, úgynevezett Reducer-hez társíthatjuk. A Reducerek a funkcionális programozási paradigmából átvett tiszta függvények, amelyek egy Action hatására a Store objektumból egy új példányt hoznak létre, végrehajtva rajta az Action típusának megfelelő műveletet. Az alkalmazás logikája így a Store-ból átkerül a Reducerekhez.

3.5.1.4 Redux Saga

Mivel a React és a Redux is a funkcionális paradigmákat követi, ezért a teljes alkalmazást egy tiszta függvényként képzelik el. Nekünk azonban bizonyos esetekben szükségünk van mellékhatásokra, ilyen lehet például egy AJAX kérés egy Web API-hoz, vagy valamilyen adat elmentése a localStorage-ba. A Redux Saga a meglévő Redux alkalmazásunkat egészíti ki egy redux middleware-rel, ahol deklaratívan definiálhatjuk, hogy egy bizonyos Action-re milyen mellékhatást szeretnénk kiváltani.

3.5.2 Komponens hierarchia

Az alkalmazásunkat felépítő komponenseket funkcionalitásuk alapján két csoportra bonthatjuk. Az állapotmentes, újrafelhasználható komponenseket prezentációs komponenseknek (presentational components [8]) nevezzük. Ezek

általánosított, az alkalmazásban többször előforduló komponensek, például listák, gombok. Nevezhetnénk az oldal építőkockáinak is, hiszen állapottól függetlenül bármilyen felületet összerakhatunk velük. Az állapottal rendelkező, az adott felhasználási hely által kötött komponenseket konténereknek (containers [8]) nevezzük. A konténerek általában egy adott nézet komponenseit és metódusait fogják össze. Ezek a komponensek kapják paraméterül a Redux Store egy állapotterét, definiálják az Action-öket, illetve az állapottéren elérhető Reducereket. Ilyen konténer lehet például a bejelentkezési felület vagy a valós idejű értesítéseket kezelő komponens. Ezzel a megközelítéssel növelhetjük a komponensek újrafelhasználhatóságát, a kód olvashatóságát, illetve tesztelhetőségét, mivel a felhasználás módja és nem a fájlkiterjesztés alapján választottuk szét a komponenseket (Separation of Concerns).

3.5.3 Navigálás

Mivel egy Single Page Application esetében az alkalmazáson belüli navigálás nem töltheti újra az egész weboldalt, ezért ezt kliens oldali útvonalválasztással kell megoldani (client side routing). A gyakorlatban ez azt jelenti, hogy amikor egy új oldalt szeretnénk meglátogatni, akkor az oldal címét dinamikusan hozzáadjuk a böngésző előzményeihez. Erre a modern böngészőkben már implementált HTML5 History API-t lehet használni. Az alkalmazás fel van iratkozva a History változásaira és az új cím alapján az aktuális nézetet kicseréli egy új nézetre. Az alkalmazás nézeteit és a hozzájuk kapcsolódó elérési útvonalakat deklaratív komponensekkel tudjuk leírni, amihez a react-router nevű NPM csomagot használom. Ez a legelterjedtebb navigációs könyvtár a React ökoszisztémában. A fenti módszerrel szimulálhatjuk egy új oldal betöltését anélkül, hogy feleslegesen újra kellene tölteni a teljes weboldalt és még a böngésző előre/vissza gombjainak funkcionalitását is megőriztük.

3.5.4 Kommunikáció

A szerver és a kliens oldal közötti kommunikáció AJAX kérésekkel történik, amihez a HTML5 Fetch API-t használom. Ezt még nem minden böngésző támogatja, illetve a jelenlegi implementációk is pár helyen különböznek, ezért ennek

kiküszöbölésére a whatwg-fetch csomagot használom, ami a GitHub saját implementációja.

Az API hívások elfedésére egy úgynevezett API konnektor objektumot hoztam létre, ami egy vékony absztrakciós réteg az egyszerű HTTP kérések felett, így a Redux Sagaknak nem kell tudniuk az API végpontjairól, csak egy metódus meghívása a feladatuk. Ez a megoldás elkülöníti a komponensek feladatait és megkönnyíti az API hívások tesztelhetőségét is.

3.5.5 Felhasználói élmény

A felhasználói élmény (user experience) pont olyan fontos szempont egy webes alkalmazás fejlesztésénél, mint az alkalmazás funkcionalitása, hiszen, ha a felhasználók nem tudnak eligazodni és csak nehézkesen tudják használni, akkor az alkalmazásunk nem fog tudni jelentős felhasználó bázisra szert tenni. Esetünkben az alkalmazásnak átlátható, egyszerű felülettel kell rendelkeznie, mert várhatóan idősebb, számítógépes ismeretekkel alig rendelkező személyek is használni fogják. Az alkalmazásnak készítenünk kell mobil nézetet is, így érdemes a felületek tervezésénél a „mobile-first" szemléletet követni, azaz először mobil készülékre tervezünk, majd onnan kiindulva a nagyobb kijelzőkre.

A Material Design [9] a Google tervezési elveinek gyűjteménye, amit webes és natív alkalmazások felületeinek és felhasználói élményeinek kialakításához hoztak létre. Implementálásához a material-ui NPM csomagot használtam, amiben alapvető prezentációs komponensek, mint például beviteli mezők, vagy dialógus ablakok találhatóak.

3.6 Authentikáció

Egy modern webes alkalmazás tervezésének elengedhetetlen része a biztonságos és megbízható authentikációs stratégia kidolgozása. Ebben az alkalmazásban kifejezetten fontos, mivel két különböző szintű felhasználóval van dolgunk. A közös képviselők csak a saját csoportjaik, illetve azok lakóinak adataihoz férhetnek hozzá, számlákat állíthatnak ki, de egy lakó nevében óraállást nem jelenthetnek be. A lakók pedig csak a saját adataikhoz férhetnek hozzá, adhatnak be óraállást, de számlát nem állíthatnak ki. Feladatunkat még inkább

megnehezíti, hogy backend felületünk egy RESTful API. A REST egyik alapelve, hogy állapotmentes, a session kezelését a kliensre bízza, így a legtöbb klasszikus süti alapú authentikációs módszert el is felejthetjük.

3.6.1 Felhasznált technológiák

3.6.1.1 JWT

A JSON Web Token [10] (röviden JWT) egy JSON alapú nyílt sztenderd hozzáférési kulcsok generálására és validálására. A JWT kulcs három részből épül fel:

- Header
- Payload
- Signature

A Header tartalmazza, hogy melyik algoritmust használtuk a Signature generálására (ez általában az SHA256). A Payload tartalmazza a JSON objektumot. Ebben bármilyen adatot tehetünk, például egy felhasználó nevét, azonosítószámát, típusát, bármit, ami az authentikált felhasználó azonosításához szükséges. A Payload tartalmaz még egy időbélyeget (timestamp), ami a kulcs generálásának időpontja. Ennek később fontos szerepe lesz a kulcs validálásánál. A Signature egy Base64url enkódolt string, ami tartalmazza a Header-t és a Payload-ot. A Signature generálásánál szükségünk lesz egy biztonsági kulcsra, amivel „megsózzuk" az SHA256 hash-t. Fokozottan ügyeljünk, hogy ez a biztonsági kulcs ne kerüljön illetéktelenek kezébe. Végezetül a három komponenst külön-külön Base64url enkódoljuk. Az így kapott három stringet pontokkal elválasztva konkatenáljuk össze, így a végeredmény sokkal kompaktabb az átlagos XML alapú sztenderdeknél és könnyen utaztathatjuk őket HTML fájlban, vagy HTTP Header-ben.

3.6.2 Authentikáció folyamata

A bejelentkezési felületen a felhasználó beírja a felhasználónevet és a jelszót. Ez egy POST kérésben, biztonságos HTTPS kapcsolaton keresztül jut el a szerverhez. Mivel a jelszavakat titkosítva tároljuk, ezért először a kapott jelszót is hash-elni kell, majd összevetni az adatbázisban tárolt adatokkal. Ha megtaláljuk a

megadott felhasználói adatokat, akkor biztosak lehetünk benne, hogy egy létező felhasználó szeretne bejelentkezni. Mivel a szerverünk állapotmentes, ezért nem menthetjük el ezt a sessiont, és nem adhatunk cserébe egy sütit a kliensnek. Más megoldás kell. Valahogyan tudatnunk kell a kliens alkalmazással, hogy engedélyezzük a bejelentkezését, és biztosítékot kell adni, hogy ez a kijelentkezésig így is marad. Használjuk erre a JWT kulcsot. A kulcs payload-jába beletesszük az alapvető felhasználói adatokat, például név és azonosítószám, de ügyeljünk arra, hogy a jelszót sose. Aláírjuk a JWT kulcsot a szerver titkos kulcsával, majd a POST kérés válaszaként elküldjük az így kapott kulcsot. Innentől a kulcs megőrzése a kliens egyedüli feladata lesz. A tárolás módszere sokféle lehet, ajánlott valamelyik Web Storage technológiát használni, mondjuk localStorage, vagy sessionStorage. Tárolhatjuk sütiben is, de fontos átgondolni, hogy milyen veszélyekkel járhatnak a különféle megoldások.

A Web Storage elérhető JavaScript oldalról ugyanazon a domain-en belül. Ez azt jelenti, hogy minden JavaScript kód, ami az oldalon fut írhatja és olvashatja a Web Storage-ot, így az ott tárolt kulcs XSS támadás veszélyének van kitéve. Az XSS, azaz Cross Site Scripting támadás során a támadó JavaScript kódot próbál beilleszteni az oldalba. Ez történhet akár egy szövegbeviteli mezőn keresztül is. Ennek megakadályozására minden felhasználói interakcióból származó adatot validálni és escape-elni kell.

Alternatív megoldás lehet a sütik használata. Ha megadjuk a httpOnly flag-et, akkor a süti nem érhető el JavaScript oldalról, így megakadályozhatjuk az XSS támadásokat, és a Secure flag-gel biztosíthatjuk, hogy a sütik csak HTTPS kapcsolaton keresztül utazhassanak. Azonban a sütik ki vannak téve az úgynevezett Cross Site Request Forgery, vagy röviden CSRF támadásnak. Ennek során egy megbízhatatlan oldal megpróbálja rákényszeríteni a felhasználót, hogy meglátogasson egy olyan oldalt, ahol authentikálva van, majd a hamisított kérésen keresztül valamilyen műveletet hajt végre az oldalon. Például elküld egy vírusos linket egy chat alkalmazásban. Ennek a módszere lehet akár egy HTML img elem src-jébe beállított URL vagy elrejtett HTML form. Ennek megoldására általában valamilyen szerver oldali védelmet kell beépíteni, például CSRF-Token validálást,

de ezek a módszerek sajnos legtöbb esetben a szerver állapotmentességének feladásával járnak.

Szerencsére a React beépített XSS védelemmel van ellátva, így jelen alkalmazásunkban a JWT kulcs tárolására biztonságos választás lehet a localStorage. Innentől kezdve minden API hívásnál csatolni kell a kulcsot a HTTP kéréshez. Ennek módja is sokféle lehet, de a legelterjedtebb megoldás a HTTP Authorization Header használata. Ehhez a Bearer sémát fogjuk alkalmazni. A Header-be a Bearer kulcsszó után szóközzel elválasztva írjuk a JWT kulcsot.

Ha olyan végpontot hívunk a Web API-on, amihez authentikáció szükséges, akkor a szerver kiolvassa az Authorization Header-ből a kulcsot, majd validálja. Ellenőrzi, hogy a Payload-ban szereplő azonosító valódi-e, és megnézi a kulcs készítésének időpontját is, ugyanis a szerveren lehetőségünk van lejárati időt is beállítani. Ez bármekkora időintervallum lehet, 1 óra, 1 hét, vagy akár évek is. Nem ajánlott azonban túl hosszú lejárati időt használni, mivel, ha valaki megszerzi egy felhasználó JWT kulcsát, akkor szabadon használhatja az alkalmazást az eredeti tulajdonosnak álcázva magát. Ezt a veszélyt azzal lehet minimalizálni, ha a lejárati időt minél kisebbre vesszük, így a támadónak kevés ideje van az ellopott kulcs felhasználására.

3.7 Értesítési rendszer

Az alkalmazásnak értesítéseket kell küldeni a felhasználóknak új számla kiállításakor. Két különböző értesítési módszert is támogatni kell, email, illetve alkalmazáson belüli értesítéseket.

Az email értesítések kiküldéséhez szükség lesz egy SMTP szerverre. Ha rendelkezünk Google fiókkal és email címmel, akkor szabadon használhatjuk a Google saját SMTP szolgáltatását. Ehhez szükségünk lesz a Google fiókunk email címére és jelszavára. Ha nem szeretnénk a saját jelszavunkat használni, akkor igényelhetünk alkalmazás specifikus jelszavakat. Ezzel a jelszóval csak az SMTP szolgáltatáshoz férhetünk hozzá, így akkor is biztonságban marad a Google fiókunk, ha illetéktelenek kezébe kerül a jelszó.

Az alkalmazáson belüli (in-app) értesítésekhez valós idejű, kétirányú kommunikáció szükséges a szerverrel. Ehhez a WebSocket technológiát fogjuk használni.

3.7.1 Felhasznált technológiák

3.7.1.1 Nodemailer

A Nodemailer egy NPM csomag, amivel pár kódsor megírásával küldhetünk email-eket Node.js környezetből. Külső SMTP hozzáférést és Amazon SES (Simple Email Service) integrációt is támogat. Küldhetünk szöveges és HTML alapú üzeneteket is, az alkalmazásban az utóbbit fogjuk használni.

3.7.1.2 WebSocket

A WebSocket [11] egy internetes kommunikációs protokoll, ami teljes, duplex kommunikációs csatornát valósít meg kliens és szerver között egyetlen TCP kapcsolaton keresztül. Az aktív WebSocket kapcsolatok szerver oldali tárolására egy kulcs-érték objektumot használunk, ahol a kulcs a felhasználó azonosítója, az érték pedig WebSocket kapcsolatok tömbje. Azért van szükség tömb adatszerkezetre, mert párhuzamosan egy felhasználó akár több eszközön is bejelentkezhetett, mondjuk egy személyi számítógépen és egy mobiltelefonon, így az új értesítéseket mindkét készülékre továbbítani kell. Amikor egy új értesítést kell küldeni egy adott felhasználónak, akkor végig iterálunk a tömb elemein, és minden kapcsolatnak elküldjük az értesítést. Ha a kapcsolat közben megszakadt, mondjuk a felhasználó bezárta a böngésző ablakot, akkor ezt a kapcsolatot kivesszük a tömb elemei közül.

3.7.2 Valós idejű értesítések kezelése

A valós idejű értesítéseket kezelő komponenst az Observer design pattern [12] alapján készítettem el. A kliens alkalmazás betöltésekor a kliens kiépíti a WebSocket kapcsolatot a szerverrel. A kapcsolat létrejöttekor a szerver kikeresi az adatbázisból azokat az értesítéseket, amiket még nem tekintett meg a felhasználó, majd ezeket továbbítja a kliensnek. Új számla kiállításakor a szerver először készít egy új értesítést az adatbázisban. Ha az adott felhasználó éppen be van jelentkezve, akkor elküldi ezt az értesítést a kliensnek. Ha a felhasználó megnyitja az értesítési

felületet, akkor a kliens automatikusan küld egy üzenetet a szervernek, hogy az eddig kézbesített értesítéseket láttamozza, így újabb kapcsolódás esetén a szerver nem fogja elküldeni még egyszer a már megtekintett értesítéseket.

3.8 Fejlesztőkörnyezet

3.8.1 Rendszerkövetelmények

- macOS (Sierra v10.12.6 és újabb verziók)
- MongoDB (v3.4.2)
- Node.js (v8.4.0 és újabb verziók)
- NPM (v5.3.0 és újabb verziók)
- Yarn.pkg (v1.0.1 és újabb verziók)

A fejlesztés természetesen történhet más operációs rendszeren is, de a követelményekben feltüntetett eszközök elérhetőségét és kompatibilitását az adott rendszeren előzetesen meg kell vizsgálni.

3.8.2 A fejlesztőkörnyezet kialakítása

3.8.2.1 MongoDB szerver elindítása

Másoljuk a DVD melléklet tartalmát a számítógépre. Készítsünk egy új mappát, ahol az adatbázis fájlokat fogjuk tárolni, majd indítsuk el a MongoDB szervert.

```
$ mkdir rezsi.io
$ mongod --dbpath rezsi.io
```

3.8.2.2 Resource szerver konfigurálása

A szervert környezetváltozókkal konfigurálhatjuk. A legegyszerűbb módja ennek, ha létrehozunk egy `.env` fájlt az `api.rezsi.io` mappa gyökerében, amelyben kulcs-érték párokban felsoroljuk a változókat.

- NODE_ENV: beállíthatjuk, hogy a szerver éles vagy fejlesztői módban induljon el
- PORT: szerver portja

- JWT_SECRET: JWT token generálásához szükséges titkosítási kulcs
- MONGO_HOST: MongoDB szerver kapcsolódási URI-ja
- MONGO_PORT: MongoDB szerver kapcsolódási portja
- GMAIL_USER: GMail SMTP felhasználó neve
- GMAIL_PASS: GMail SMTP felhasználó jelszava
- GMAIL_ADDRESS: GMail SMTP-n keresztül küldött email-ek feladója
- DEBUG: debug.js csomag konfigurációs string-je

A forrásfájlok között találhatunk egy `.env.example` névvel ellátott példafájlt, amely tartalmazza az alapvető konfigurációt. Ezt elég csak átnevezni és átírni az értékeket.

```
$ cp .env.example .env
```

Fokozottan figyeljünk arra, hogy ez a fájl jelszókat és titkosítási kulcsokat tartalmaz, ezért semmilyen körülmények között sem juthat illetéktelen kezekbe. Ha ez mégis megtörténik, azonnal cseréljünk jelszót és változtassuk meg a kulcsokat. Ezt a fájlt ne commitoljuk a verziókövető rendszerbe.

3.8.2.3 Resource szerver elindítása

Navigáljunk a DVD mellékletről származó forrásmappába, telepítsük a JavaScript dependenciákat és indítsuk el a szervert.

```
$ cd api.rezsi.io
$ yarn install
$ yarn start
```

A szervert watch üzemmódban is indíthatjuk, ekkor minden fájlmódosításkor a szerver újraindul. Ehhez az alábbi parancsot kell kiadni.

```
$ yarn start-dev
```

3.8.2.4 Static szerver elindítása

Navigáljunk a DVD mellékletről származó forrásmappába, telepítsük a JavaScript dependenciákat és indítsuk el a szervert.

```
$ cd app.rezsi.io
$ yarn install
$ yarn start
```

3.9 Fejlesztési munkafolyamat

3.9.1 Verziókövetés

A fejlesztéshez a Git verziókövető rendszert használtam. A resource szervert és a kliens alkalmazást (a static szerverrel együtt) külön repository-ban helyeztem el, mivel ezek a projektek akár párhuzamosan is fejleszthetők és csak egy közös interfészre van szükség a kommunikációjukhoz. A repositoryk elérhetőek a GitHub felületén is a https://github.com/balintsoos?tab=repositories címen.

3.9.2 Continuous delivery

Az alkalmazást a Heroku felhő platformon futtatjuk. Ez a szolgáltatás a legtöbb üzemeltetéssel kapcsolatos terhet leveszi a fejlesztő válláról. A rendszer összeköthető a GitHub repositoryval, így a master branch változáskor automatikusan kirakja az új verziót az éles környezetbe.

Mivel nem commitolhatjuk be a verziókövető rendszerbe a szenzitív adatokat tartalmazó `.env` fájlt, ezért a környezetváltozókat a Heroku erre a célra kialakított felületén kell megadni.

3.9.3 Tesztelés

TODO

3.10 Fejlesztési lehetőségek

- Többnyelvűség támogatása: Az alkalmazás felépítése során kezdettől fogva szem előtt tartottam a többnyelvűség támogatását, így csak a felhasználói felület feliratainak lefordítása és egy nyelvválasztó beállítás elhelyezése van hátra.
- Értesítési rendszer kiszervezése külön web szolgáltatásba: A rendszert saját adatbázissal kell ellátni, ami tárolja az értesítéseket. A resource szerver HTTP

kéréseken keresztül kommunikálhat az értesítési szolgáltatással, ami kapcsolódik az SMTP szerverhez és WebSocket kapcsolaton keresztül küld értesítéseket a kliens számára.

- Felhasználó email címének és jelszavának módosítása: A jelszó cseréjét úgy célszerű megvalósítani, hogy egy email-ben elküldött linkre kattintva a felhasználót egy olyan oldalra navigáljuk, ahova beírhatja az új jelszavát. Ehhez szerver oldalon tárolni kell memóriában egy azonosítót, ami az új jelszó igénylésének valódiságát biztosítja. Ne fordulhasson elő olyan, hogy valaki a felhasználó tudta nélkül cserélte ki a jelszót, vagy az email címet.
- Felhasználói adatok megadása: A felhasználók megadhatják a lakcímüket, telefonszámukat és profil képet tölthetnek fel. Ezeket az adatokat csak a csoport tagjai tekinthetik meg.

Irodalomjegyzék

[1] Roy Thomas Fielding: Representational State Transfer (2017.12.03.) https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[2] Node.js Foundation: About Node.js (2017.12.03.) https://nodejs.org/en/about/

[3] StrongLoop: Express.js (2017.12.03.) https://expressjs.com/

[4] MongoDB Inc.: What is MongoDB? (2017.12.03.) https://www.mongodb.com/what-is-mongodb

[5] Mongoose,js: Documentation (2017.12.03.) http://mongoosejs.com/docs/index.html

[6] Facebook Inc. - React.js (2017.12.03.) https://reactjs.org/

[7] Cabot Solutions: A Detailed Study of Flux: the React.js Application Architecture (2017.11.29.) https://www.cabotsolutions.com/2017/01/detailed-study-flux-react-js-application-architecture

[8] Dan Abramov: Presentational and Container Components (2017.12.02.) https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

[9] Google Inc.: Material Design (2017.12.03.) https://material.io/

[10] Auth0: Introduction to JSON Web Tokens (2017.12.02.) https://jwt.io/introduction/

[11] WebSocket.org: About HTML5 WebSocket (2017.12.03.) https://www.websocket.org/aboutwebsocket.html

[12] Addy Osmani: Learning JavaScript Design Patterns (1st Edition), O'Reilly Media, 2012, [254], 978-1449331818 (ISBN-13)

Mellékletek

DVD melléklet tartalma

A mellékelt DVD két mappát tartalmaz. Az `api.rezsi.io` nevű mappában a resource szerver forráskódját, az `app.rezsi.io` mappában a static szerver és a kliens alkalmazás forráskódját találjuk.

```
api.rezsi.io
├── src
│   ├── config
│   ├── controllers
│   ├── models
│   ├── routes
│   ├── utils
│   └── index.js
├── .env.example
├── .editorconfig
├── .eslintignore
├── .eslintrc
├── package.json
└── yarn.lock

app.rezsi.io
├── app
│   ├── components
│   ├── containers
│   ├── images
│   ├── tests
│   ├── translations
│   ├── utils
│   ├── app.js
│   ├── configureStore.js
│   ├── global-styles.js
│   ├── i18n.js
│   ├── index.html
│   ├── manifest.json
│   └── reducers.js
├── internals
├── server
├── .editorconfig
├── package.json
├── Procfile
└── yarn.lock
```

A mappák további konfigurációs fájlokat is tartalmazhatnak, de ezek a dokumentáció szempontjából nem lényegesek.