Jaettujen vapaa-ajan asuntojen lokijärjestelmä

Eetu Lehto

Haaga-Helia ammattikorkeakoulu

Amk-opinnäytetyö

2021

Tietojenkäsittely - ohjelmistotuotanto

Tiivistelmä

Tekijä(t)
Eetu Lehto

Tutkinto
Tietojenkäsittely - ohjelmistotuotanto

Raportin/Opinnäytetyön nimi
Jaettujen vapaa-ajan asuntojen lokijärjestelmä

Sivu- ja liitesivumäärä
48

Tässä opinnäytetyössä tehdään full stack sovellus jaettujen vapaa-ajan asuntojen tiedon keskittämiseen. Tutkitaan full stack kehityksen eri osa-alueita ja suunnitellaan näiden pohjalta sovelluksen sisältöä. Projektille pyritään asettamaan heti selkeä tavoite, mihin projektin tulisi päätyä. Tämän tavoitteen tueksi käydään läpi projektiin kuuluvat asiat, ja selkeyden vuoksi rajataan pois asiat, jotka eivät kuulu projektiin.

Projektin suunnittelu aloitettiin arkkitehtuurista, jotta olisi helpompaa lähteä pohtimaan projektissa käytettäviä resursseja ja teknologioita. Tutkittiin erilaisia vaihtoehtoja sovelluksen eri vaiheiden tekemiseen, sekä näiden vaihtoehtoisten tapojen hyötyjä ja haittoja. Tarkoituksena oli löytää mahdollisimman hyvä toteutustapa kuitenkin vain yhden ennalta päätetyn asian puitteissa; ohjelma tulisi kirjoittaa käyttämällä TypeScript -ohjelmointikieltä.

Projektin jokaisesta työvaiheesta käytiin läpi oleellisimmat osat, kuitenkaan avaamatta asioita liian syvällisesti, jotta opinnäytetyö pysyisi sopivan mittaisena. Full stack kehitys on niin laaja-alainen asia, että tässä opinnäytetyössä oli pakko karsia siitä pois liian syväluotaavaa tutkimustapaa. Pääpainona tällä projektilla on TypeScriptin käyttö, sen ominaisuudet sekä sallimat mahdollisuudet. TypeScriptin käytöstä etsittiin paljon tietoa ja siitä löydettiin useita julkaisuja.

Projektin päätyövaiheet olivat projektin suunnittelu, toteutus sekä julkaisu. Ohjelma siis julkaistiin Linux -palvelimelle avoimeksi internettiin. Tälle ohjelmalle ei ole odotettavissa suurta käyttäjäkuntaa, mutta se on vapaa kenelle tahansa käyttää.

Koska sovelluskehitys on jatkuvaa kehittymistä, tämän projektin lopuksi on myös huomioitu asiat, jotka jäävät jatkokehitettäväksi. Tämän projektin aikana tehtyä full stack -sovellusta tullaan siis jatkokehittämään vielä paremmaksi, kun minkälaiseksi se tämän opinnäytetyön lopuksi jäi.

Asiasanat
Ohjelmistokehitys, Fullstack, Web -sovellus, TypeScript

Sisällys

1. Johdanto

Full stack -ohjelmointi web -kehityksessä tarkoittaa web -sovelluksen kaikkien osa-alueiden kanssa työskentelyä. Nämä kaikki osa-alueet käsittävät pääosin tietokannan, palvelinpään (backend) ja käyttöliittymän (frontend). OpenJS Foundation on luonut Node.js -nimisen JavaScript -ajoympäristön, joka perustuu Chrome -selaimen V8 JavaScript moottoriin (OpenJS Foundation – Projects). Tämä mahdollistaa sen, että myös palvelinpään koodia voidaan kirjoittaa käyttämällä JavaScript -ohjelmointikieltä. Tämä ei ole ennen ollut mahdollista, sillä JavaScript on ollut perinteisesti vain internetselaimen ymmärtämä ohjelmointikieli dynaamisen sisällön näyttämiseen web -sivulla.

Nykypäivänä web -sovelluksia on todella paljon, arviolta lähes 1.2 miljardia (Huss 8.10.2021). Koska tällaisia sovelluksia tehdään näin paljon, on niiden tekijöitäkin useita. Web -sovelluksia voi tehdä lukuisilla erilaisilla arkkitehtuureilla yhdistelemällä useita erilaisia teknologioita. Tässä projektissa käytetään kuitenkin ohjelmointiin pelkästään TypeScript -ohjelmointikieltä, joka on JavaScriptin ylijoukko (Microsoft 2021). Tämä tarkoittaa sitä, että TypeScript sisältää kaiken saman syntaksin kuin JavaScript, sekä hieman enemmän, kuten vahvan tyypityksen. Koodin vahva tyypittäminen tarkoittaa sitä, että kaikelle ohjelmassa liikkuvalle tiedolle on tietotyyppi. Nämä tietotyypit voivat olla valmiita tyyppejä, kuten merkkijono(string) tai totuusarvo(boolean). Myös omia tietotyyppejä voi antaa esimerkiksi objektille, joka sisältää erilaisia avain-arvo-pareja, joilla on myös oma tyyppinsä. Tämä mahdollistaa sen, että ohjelmassa ei voida kutsua jonkin objektin sellaista arvoa, jota ei ole sinne alun perin määritelty. Näin vältytään turhilta ongelmilta koodissa. Huonoa logiikka TypeScript ei tunnista, sen suhteen täytyy ohjelmoijan itse olla tarkkana.

Koska web -sovelluksia voi tehdä niin monenlaisia ja erilaisiin tarkoituksiin, sen mahdolliset käyttökohteet ovat lähes rajattomat. Web -sovellukset ovat todella hyviä tiedon keskittämiseen suurellekin yleisölle, tai vain rajatulle joukolle. Tässä projektissa tehdään juuri sellainen sovellus, jonka avulla on helppo keskittää jaetun vapaa-ajan asunnon tiedonvälitys yhteen paikkaan rajatulle ryhmälle, joka käsittää henkilöt, jotka käyttävät tätä kyseistä vapaa-ajan asuntoa. Projektissa käydään läpi kaikki full stack -ohjelman kehitysvaiheet, ilman että sukelletaan todella syvälle mihinkään vaiheeseen. Tämä ohjelma kehitetään melko pienelle käyttäjäkunnalle, eikä siellä tule olemaan juurikaan arkaluontoista tietoa.

2. Tutkimuskysymykset

2.1 Projektin tavoite

Projektin tavoitteena on kehittää full stack -ohjelmisto TypeScript -ohjelmointikielellä, sekä perehtyä TypeScriptin käyttämiseen SQL (Structured Query Language)-tietokannan kanssa. Tarkoituksena on luoda mahdollisimman eheä tietokanta, sekä tehokkaasti toimiva sovellus. Projektiin kuuluu myös sovelluksen julkaiseminen alusta loppuun Linux -palvelimelle.

2.2 Projektiin kuuluvat asiat

Projektiin kuuluu palvelinpään logiikan rakentaminen, kuten pyyntöihin vastaaminen ja tiedon, sekä käyttäjien, validointi. On tärkeää, että pidetään huolta siitä, että käyttäjät pääsevät tekemään vain sellaisia asioita, joihin heillä on oikeudet. Palvelinpää, eli backend, hallitsee myös tietokantapyyntöjä. Pyrkimys on saada backend toimimaan mahdollisimman tehokkaasti, kuitenkaan liiallista optimointia tekemättä.

Sovellukselle luodaan käyttöliittymä React -kirjastolla. Käyttöliittymä kirjoitetaan myös TypeScript -ohjelmointikielellä. Käyttöliittymän tarkoitus on mahdollistaa käyttäjälle varausten luominen, tarkastelu sekä niiden muokkaus. Tämän lisäksi käyttäjän tulee pystyä lisäämään vapaa-ajan asuntoja, muokata niitä, lisätä viestejä sekä puutteita ja muokata näitäkin. On tärkeää, että käyttäjien oikeudet myös pätevät käyttöliittymässä, että käyttäjä ei voi muokata tai poistaa julkaisua, jota tämä ei ole itse luonut. Koska käyttöliittymä luodaan alusta loppuun, tulee se suunnitella oikein ja siitä on luotava mockup -kuvat, jotta käyttöliittymän ulkoasu saadaan hahmoteltua ennen kuin itse käyttöliittymää aletaan luomaan. Nämä mockup -kuvat sekä kokonainen suunnittelu myös toimii hyvänä vertailukohtana sille, että miten hyvin tässä projektissa onnistuttiin toteuttamaan alun perin suunniteltu käyttöliittymä. Käyttöliittymä pyritään saamaan myös skaalautuvaksi mobiililaitteille.

Tietokanta on projektin kannalta tarpeellinen. Tämä tullaan asentamaan samalle palvelimelle, ohjelman kanssa. Tähän tietokantaan ohjelma suorittaa kyselyitä sekä tallettaakseen tietoja, että saadakseen tietoa, jota tarjota käyttöliittymälle.

Projektissa tullaan ottamaan huomioon myös perustason tietoturva. Tämä käsittää lähinnä tiedon validoinnin, sekä pyritään poistamaan mahdollisuus syöttää haitallista koodia käyttöliittymän kautta. Myös julkaisualustana toimivan Linux -palvelimen tietoturva tullaan ottamaan huomioon.

2.3 Projektiin kuulumattomat asiat

Projektista tullaan jättämään pois hirveä määrä erilaisia ominaisuuksia, joita usein erilaiset nettisovellukset tarjoavat. Tällaisiin asioihin lukeutuvat muun muassa kaikki sähköpostiin liittyvät asiat. Tämä sovellus ei lähetä käyttäjälle ilmoituksia sähköpostiin, eikä silloin käyttäjällä ole mahdollisuuttakaan palauttaa mahdollisesti unohdettua salasanaa.

Sovelluksesta jätetään pois myös sellaisia kehitysvaiheen asioita, joita usein isommissa projekteissa tehtäisiin. Tässä projektissa ei tulla tekemään kattavaa koodin testausta eikä oteta huomioon julkaisupalvelimen resursseja. Julkaisupalvelimen resursseja ei tarvitse ottaa huomioon, sillä tähän sovellukseen ei ole odotettavissa suurta liikennemäärää.

Koska tämän sovelluksen käyttäjäkunta oletetaan olevan melko pieni, ei sille myöskään tehdä mobiiliapplikaatiota. Mobiiliapplikaation tekeminen olisi niin työläs prosessi jo selaimessa olevan käyttöliittymän lisäksi, että se ei ole tätä projektia ajateltuna kannattavaa. Isommassa projektissa se tietenkin olisi hyödyllistä käyttäjää ajatellen, sillä silloin käyttäjä voisi helposti käyttää tätä sovellusta myös mobiiliin optimoidulla käyttöliittymällä.

3. Projektin suunnittelu

Hyvä suunnittelu on ohjelmistokehityksen kannalta todella tärkeää. Kun sovellus on hyvin suunniteltu, on helpompaa lähteä kehittämään juuri niitä asioita, joita sovelluksessa tarvitaan. Chris Northwoodin lausahdus hänen kirjassaan *The Full Stack Developer,* kuvaa suunnittelun tärkeyttä ohjelmistokehityksessä oivallisesti sanoin "…building the right thing is more important than building the thing right." (Northwood 2018, 11.), joka tarkoittaa vapaasti suomennettuna, että on tärkeämpää rakentaa oikeaa asiaa, kun rakentaa asiaa oikein.

Kun projekti on hyvin suunniteltu, on sitä helppo myös jatkokehittää. Jatkokehittäminen on ohjelmistotuotannossa yleistä, sillä useita sovelluksia muokataan niiden ensimmäisen julkaistun version jälkeen. Nämä muokkaukset parantavat sovelluksen ulkoasua, sekä myös käytettävyyttä.

Jotta arkkitehtuurin suunnittelu olisi helpompaa, on ensin päätettävä mitä kaikkea käyttäjän on pystyttävä sovelluksella tekemään. Koska kyseessä on jaettujen vapaa-ajan asuntojen tiedon hallintaan keskittyvä sovellus, on ensiarvoisen tärkeää, että sovellukseen voidaan lisätä vapaa-ajan asuntoja. Jotta sovellusta on mielekästä käyttää, on sillä oltava käyttöliittymä, jolla muun muassa onnistuu uuden vapaa-ajan asunnon lisääminen sovellukseen. Tähän vapaa-ajan asuntoon on lisättävä myös käyttäjiä. Jotta tämä on mahdollista, on käyttäjän pystyttävä luomaan itselleen käyttäjätunnus sovellukseen. Käyttäjätunnuksella tulee myös olla salasana, jotta kuka tahansa ei voi mennä toisen henkilön nimissä tekemään sovellukseen mitään. Jotta suunnittelu helpottuisi, listataan ylätasolla kaikki asiat, joita sovelluksella käyttäjä voi tehdä:

- Luoda käyttäjätunnuksen
- Kirjautua käyttäjätunnuksella sovellukseen
- Luoda uuden profiilin vapaa-ajan asunnolle
- Lisätä muita käyttäjiä vapaa-ajan asuntoon
- Vapaa-ajan asunnon profiilin luonut käyttäjä tulee pystyä muokkaamaan asunnon tietoja
- Asuntoon linkitetyt käyttäjät voivat tehdä asuntoon varauksia, lisätä puutelistaan puutteita, sekä käydä keskustelua viestialueella
- Käyttäjän tulee pystyä muokkaamaan profiilinsa tietoja
- Käyttäjän tulee pystyä kirjautumaan ulos sovelluksesta

Koska käyttäjän tulee pystyä käyttämään sovellusta mielekkäästi, on sille luotava käyttöliittymä. Käyttöliittymä on se näkymä, jonka käyttäjä selaimessaan näkee, eli frontend. Jotta käyttöliittymässä tehtävät toimet oikeasti toimisivat, on frontendin tueksi tehtävä palvelinpään logiikkaa, eli backend. Backend tulee vastaanottamaan tietoa käyttöliittymältä,

ja tarjoilee sitä REST (REpresentional State Transfer)-rajapintana takaisin käyttöliittymälle.

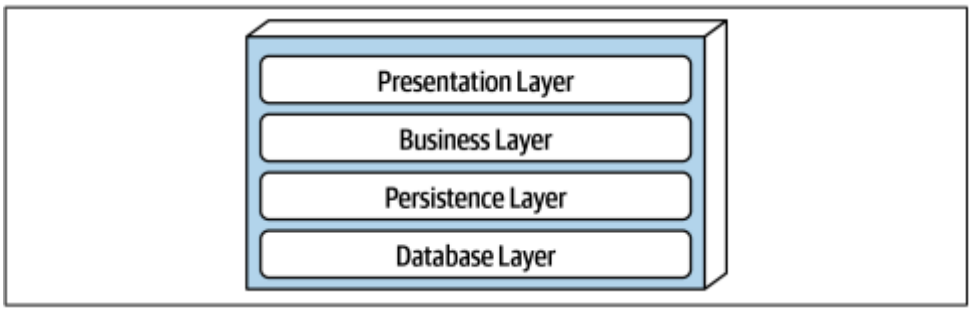
REST-rajapintaa käytetään HTTP (Hypertext Transfer Protocol) protokollan kanssa välittämään tietoa, eikä sen toiminta riipu ohjelmointikielestä tai käyttöjärjestelmästä. Se ei ole suoranaisesti oma teknologiansa, vaan nimetty tapa välittää tietoa tietyllä tavalla (Zammetti 2020, 170.). Tämä mahdollistaa sen, että tietoa on helppo välittää eri teknologioiden välillä HTTP protokollan avulla. Tässä ohjelmassa REST-rajapinta tulee tarjoamaan tietoa JSON (JavaScript Object Notation)-muodossa, joka on paljon käytetty tietomuoto REST-rajapinnoissa.

Koska ohjelman käyttämä tieto on saatava tallennettua jonnekin, niin ettei se katoa, otetaan avuksi tietokanta. Tietokannalle on useita eri vaihtoehtoja. Oikean tietokannan valinta on tärkeää, koska se on suuri tekijä siinä, kuinka ohjelman tiedon kulku saadaan rakennettua.

Otetaan suunnitteluvaiheessa huomioon myös ohjelmiston ihmiskielet. Itse ohjelmointikieli on jo päätetty olevaksi TypeScript, mutta on hyvä ottaa myös huomioon muut eri kielet, joita sovelluksen kehitykseen liittyy. Koodin ihmiskieli tulee olemaan englanti, tämä tarkoittaa siis sitä, että koodissa käytetyt muuttujien, funktioiden, loki-ilmoitusten, sekä muun koodista ulospäin näkymättömän sisällön kieli on englanti. Itse sovelluksen käyttöliittymän kieleksi tulee suomi, jatkokehityksen kannalta on huomioitava, että jatkossa sovellus tulee tukemaan useampaa eri kieltä.

3.1 Arkkitehtuuri

Ohjelmistojen arkkitehtuuriin on erilaisia tyylejä, niistä yleisin on kerrostettu arkkitehtuurityyli. Tämä arkkitehtuurityyli tarkoittaa käytännössä sitä, että ohjelmalla on 3–4 kerrosta, jotka vapaasti suomennettuna ovat: esityskerros, toimintakerros, säilyvyyskerros ja tietokantakerros. Pienemmissä ohjelmissa, kuten tässä, voidaan yhdistää toimintakerros, sekä säilyvyyskerros yhdeksi toimintakerrokseksi (Richards & Ford 2020, 133.).



Kuva 1. Kerrosarkkitehtuurin standardi tasojärjestys (Richards & Ford 2020, 134.)

Tämä sovellus on tarkoitus luoda käyttäen TypeScript -ohjelmointikieltä, joten se on otettava huomioon ohjelmistoarkkitehtuuria valittaessa.

Aluksi tulee selvittää minkälaisia asioita kokonaisessa web -sovelluksessa pääpuolin on. Full stack sovellus koostuu pääosin kahdesta osasta, käyttöliittymästä sekä palvelinpään logiikasta (Zammetti 2020, XXI.). Usein web -sovellukset käyttävät myös tietokantaa, johon sovelluksen käyttämä data tallennetaan. Tässä sovelluksessa siis käyttöliittymä tulee olemaan esityskerros, palvelinpään logiikka tulee olemaan toimintakerros ja tietokanta tulee olemaan tietokantakerros.

3.1.1 Backend

Backend, eli palvelinpään logiikka, tullaan tässä tapauksessa rakentamaan Node.js ajoympäristöön, jolloin voidaan käyttää TypeScript -ohjelmointikieltä. Palvelinpään logiikan rakentamiseen voitaisiin käyttää lukuisia muitakin erilaisia ohjelmointikieliä, kuten Pythonia tai Javaa. Pythonille on erilaisia ohjelmoinnin tukiympäristöjä palvelinpään logiikan rakentamiseen, kuten Django. Django on ylätason webohjelmoinnin tukiympäristö Python-ohjelmointikielelle (Django Software Foundation, 2020.). Javalle taas on luotu muun muassa Spring Framework niminen webohjelmoinnin tukiympäristö. Jos tämä projekti ei olisi sidottu TypeScriptiin, niin jompikumpi edellä mainituista teknologioista voisi olla hyviä vaihtoehtoja tämän projektin palvelinpäänlogiikan toteutukseen.

Koska kyseessä on projekti, joka tehdään TypeScript -ohjelmointikieltä käyttäen, on otettava käyttöön Node.js -ajoympäristö. Node.js sisältää paljon valmiita moduuleja webkehitykseen. Esimerkiksi pyyntöihin vastaaminen onnistuu http -moduulilla (OpenJs Foundation. 2021.). Tässä projektissa ei kuitenkaan tulla käyttämään Node.js ajoympäristön natiiveja moduuleita webpyyntöihin vastaamiseen, vaan siihen käytetään Express -nimistä webohjelmoinnin tukiympäristöä.

Node.js sisältää oman paketinhallintajärjestelmänsä, nimeltä npm. Npm on Node.js ajoympäristöä varten suunniteltu paketinhallintajärjestelmä, jonka avulla voidaan asentaa projektiin riippuvuuksia, eli muun muassa erialisia kirjastoja ja tukiympäristöjä, kuten Express. Pakettien asentaminen on tehty todella helpoksi komentoriviä käyttämällä.

Niin kuin usein tietotekniikassa, Node.js:än tarjoama paketinhallintajärjestelmä ei tule ilman ongelmia. On todella helppoa ja mielekästä käyttää näitä paketteja useaan erilaiseen tarkoitukseen, aina koodin tarkistuksesta valmiisiin toiminnallisuuksiin, kuten tiedon kryptaamiseen. Mutta näissä paketeissa on haittapuolena se, että jos joku ulkopuolinen taho

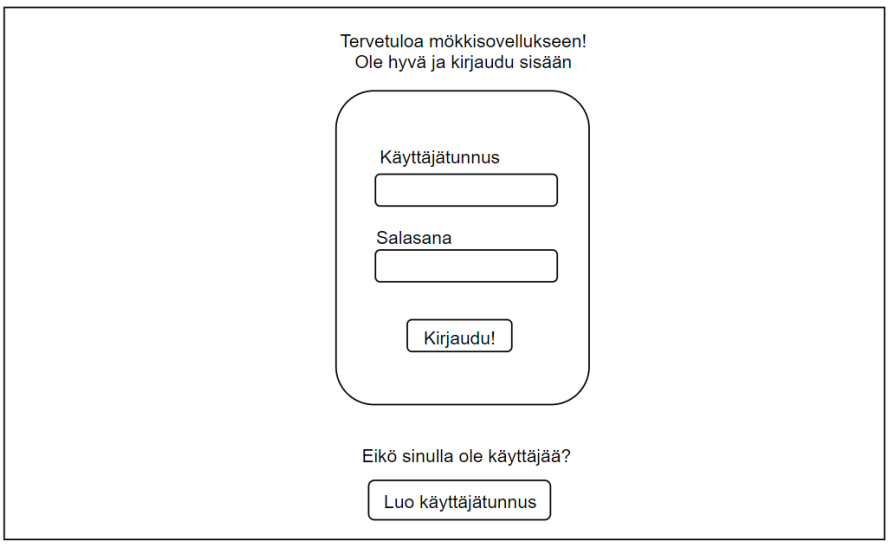
pääsee paketintarjoajan tietojen kautta itse pakettiin käsiksi, voi paketin turvallisuus vaarantua melkoisesti. Pakettiin saatetaan syöttää haitallista koodia levittämään haittaohjelmia tai vakoilemaan käyttäjiä. Yksi esimerkki löytyy CISA:n (Cybersecurity & Infrastrucure Security Agency) 22.10.2021 julkaisemassa tiedotteessa ua-parser-js -nimisen paketin sisältäneestä haittaohjelmasta. Tässä julkaisussa kerrotaan, että kolme tämän paketin versiota sisälsi haitallista koodia, joka mahdollistaisi etänä toimivan hyökkääjän saavan haltuunsa tietoja, tai jopa etähallinnan koneesta, johon tämä paketti on asennettu.

3.1.2 Frontend

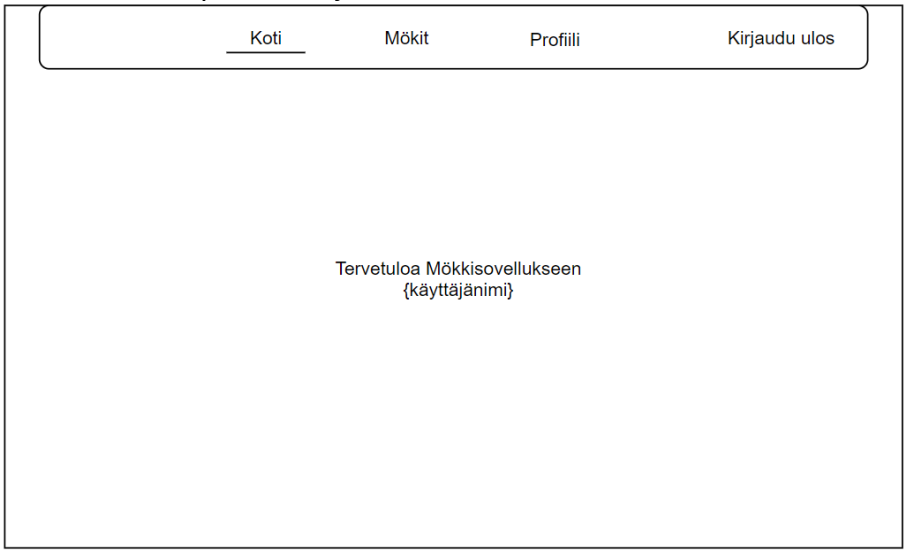
Frontend, eli tämän sovelluksen käyttöliittymä, tehdään käyttämällä JavaScript-kirjastoa nimeltä React. React on Facebookin luoma JavaScript-kirjasto nettisivujen näkymien tekoon, toimii oivallisesti myös TypeScriptin kanssa, sillä se tarjoaa jo projektin luontivaiheessa mahdollisuuden käyttää TypeScript pohjaa. React -kirjasto koodataan käyttämällä lähinnä JavaScriptiä, sekä hieman HTML-koodia ja CSS-tyylejä (Facebook, 2021). Kun koodi ajetaan selaimessa, näkyy se siellä HTML-muodossa.

Käyttöliittymänäkymän olisi voinut myös tehdä käyttäen vain HTML-koodia ja hieman JavaScript-koodia, sekä tyylit olisi voitu lisätä CSS-tyyleillä, mutta se ei olisi tukenut tämän projektin TypeScript-painotteisuutta.

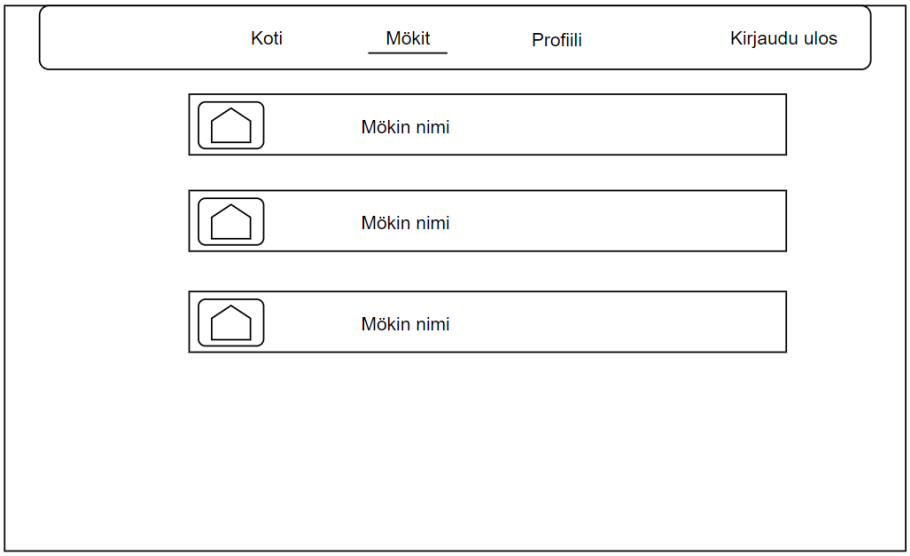
Käyttöliittymässä tullaan panostamaan ensisijaisesti käytännöllisyyteen. Tämä tarkoittaa siis sitä, että käyttöliittymän ulkomuoto ja koristelu tullaan ottamaan huomioon sovelluksen jatkokehityksessä, eikä ensimmäisessä versiossa. Jotta käyttöliittymää tehtäessä on jokin aavistus siitä, mitä lähdetään tavoittelemaan on hyvä tehdä mallikuvia Mallikuvat tunnetaan myös termillä mockup -kuva, jotka toimivat suunnannäyttäjänä käyttöliittymää toteutettaessa.



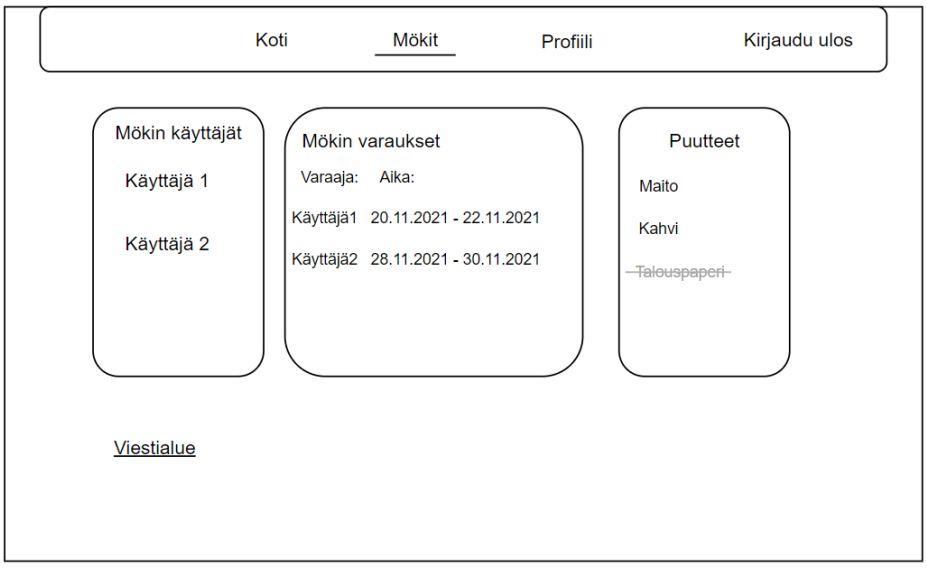
Kuva 2. Mockup -kuva kirjautumisikkunasta



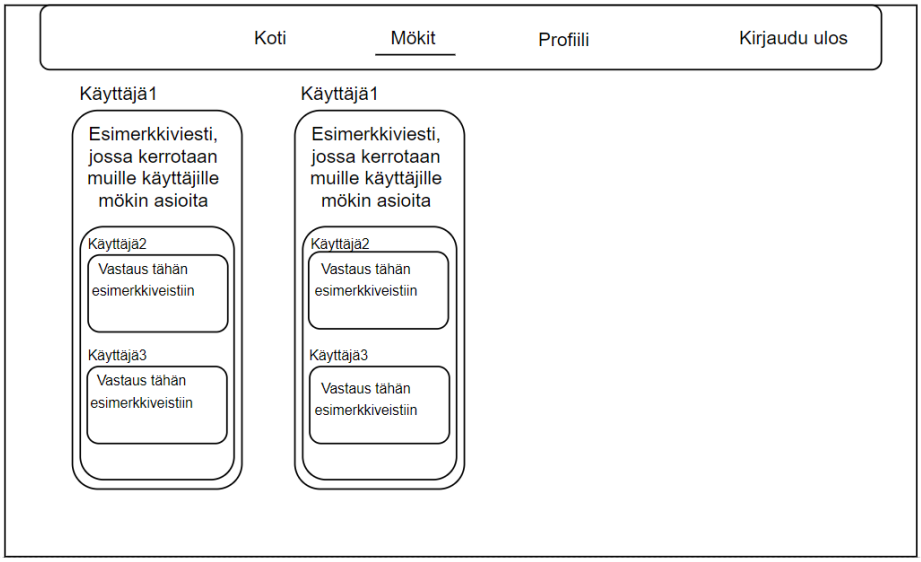
Kuva 3. Mockup -kuva kirjautuneen käyttäjän ensinäkymästä



Kuva 4. Mockup -kuva kirjautuneen käyttäjän Mökit -sivun ensinäkymästä



Kuva 5. Mockup -kuva kirjautuneen käyttäjän yhden mökin näkymästä, johon käyttäjällä on pääsy



Kuva 6. Mockup -kuva kirjautuneen käyttäjän yhden mökin viestialueesta

3.1.3 Tietokanta

Tietokantaa valittaessa tulee tarkkaan miettiä mikä on ohjelman käyttötarkoitus ja millaista tietoa ohjelma tulee sisältämään, sekä sitä, miten tieto on strukturoitu. Tässä halutaan valita SQL-tietokannan, sekä dokumenttitietokannan väliltä.

SQL-tietokanta toimii paremmin relaatioiden, eli taulujen välisten suhteiden, kanssa verrattuna dokumenttitietokantaan (Kleppmann, sivu 38). Relaatiotietokannassa jokainen tietorivi jokaisessa tietotaulussa saa itselleen uniikin ID tunnuksen. Tällöin taulujen tietoja voidaan helposti yhdistellä näiden tunnusten avulla (Oracle 2021.).

On tiedossa, että sovelluksen datassa tulee olemaan useampia erilaisia riippuvuuksia, joten halutaan valita tietokanta, joka tehokkaasti tukee näitä riippuvuuksia, eli SQL-tietokanta. Muutama esimerkki sovelluksessa käytettävän datan välisistä riippuvuuksista: Käyttäjällä voi olla pääsy useaan vapaa-ajan asuntoon ja vapaa-ajan asunnolla voi olla useita käyttäjiä. Viestillä voi olla vain yksi kirjoittaja, mutta yksi käyttäjä voi kirjoittaa monta viestiä.

Seuraavaksi haluamme päättää minkälaisen SQL-tietokannan haluamme. SQL-tietokantoja on tehty useita erilaisia, joilla on jokaisella hieman erilaisia ominaisuuksia. Esimerkiksi SQL-tietokanta nimeltä MySQL on tehty C ja C++ ohjelmointikieliä käyttäen, ja tarkoitettu tukemaan suuria datamääriä ja toimimaan nopeasti. Tätä ajatusta tukee myös se, että se on suunniteltu toimimaan niin, että jos on useita prosessoreita käytettävissä, se osaa jakaa tehtävänsä niiden välillä (Oracle 2021.). Toinen, hieman erilainen SQL-tietokanta, on nimeltään PostgreSQL. PostgreSQL on avoimen lähdekoodin projekti, jota myös

dokumentaatiossa kuvaillaan tehokkaana SQL-tietokantana, sekä tämä tietokanta palauttaa tietoa JSON muodossa (The PostgreSQL Global Development Group 2021.). JSON muotoisen datan saanti suoraan tietokannasta on hyvä lisä tähän projektiin, sillä se poistaa tarpeen suureen datan käsittelyyn, kun sitä pyydetään tietokannasta. Tämä helpottaa aiemmin mainittua REST-rajapinta kehitystä. Tähän projektiin valitaan siis tietokannaksi PostgreSQL -tietokanta.

Tässä projektissa voitaisiin myös käyttää dokumenttitietokantaa, kuten esimerkiksi MongoDB -nimistä dokumenttitietokantaa. MongoDB myös tukee JSON-muotoista dataa, joka olisi etu tässä projektissa. MongoDB -tietokannan toiminta Node.js -ohjelman kanssa on, että kukin datataulu on niin sanottu ”Schema”, joka määrittelee tietokantaan tallennettavan tiedon ”muodon”. MongoDB:n avuksi on Node.js -projekteihin tarjolla myös kirjasto helpottamaan MongoDB:n käyttöä. Tämä kirjasto on nimeltään Mongoose. Mongoose helpottaa Schemojen tekoa ja MongoDB:n käyttöä muun muassa tarjoamalla valmiita funktioita tietokantakyselyihin (Mongoose 2021; MongoDB 2021.). Niin kuin aiemmin jo todettiin, relaatioiden käsittely ei ole dokumenttitietokannoilla niin mielekästä, kun se on SQL-tietokannalla, joten vaikka MongoDB on mielekäs käyttää koodin tasolla, ei se palvele oikeaa tarkoitusta tässä projektissa.

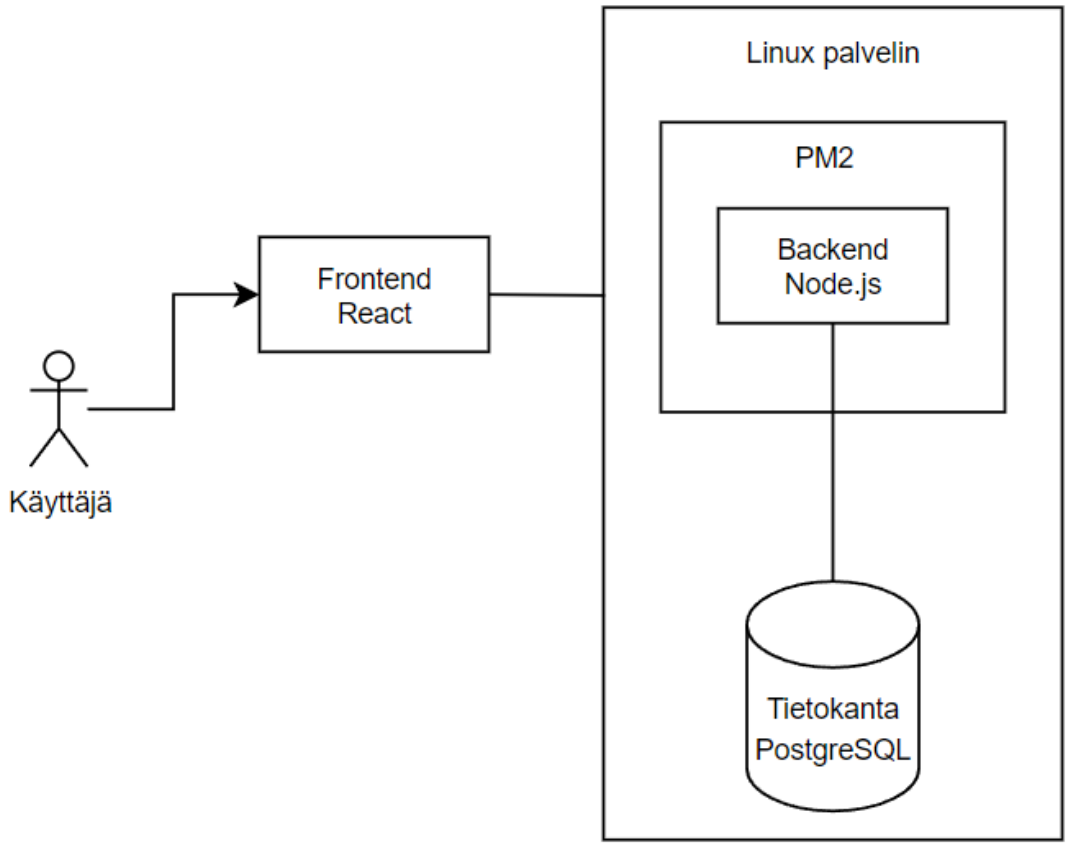
3.1.4 Julkaisualusta

Sovellus tullaan julkaisemaan Linux-palvelimella. Tietokanta tulee elämään samalla palvelimella sovelluksen kanssa. Tältä palvelimelta tarjoillaan käyttäjälle nettisivunäkymä, sekä vastaukset käyttöliittymästä lähetettyihin pyyntöihin.

Sovelluksen backend ajetaan PM2 nimisen Node.js prosessinhallintaohjelmaa käyttäen. PM2 mahdollistaa kaikki hallintatyöt mitä sovellukseen liittyy. Esimerkiksi automaattinen ohjelman käynnistys ja sovelluksen lokien tutkiminen onnistuvat tällä ohjelmalla helposti. Tämän ohjelman voi myös käynnistää palvelimen uudelleenkäynnistyksen yhteydessä automaattisesti, jolloin julkaisun jälkeen ei tarvitse huolehtia onko sovellus toiminnassa vai ei. (PM2 2021.).

3.1.5 Arkkitehtuurikaavio

Arkkitehtuuri voidaan kuvata myös seuraavalla kaaviolla:



Kuva 7. Arkkitehtuuria kuvaava kaavio

4. Projektin toteutus

Projektia toteutettaessa tutkitaan erilaisia tapoja, sekä käytäntöjä, joita fullstack -sovelluskehityksessä suositaan. Projektin aikana pyritään pysymään suunnitelmassa niin hyvin kuin mahdollista, mikä ei välttämättä aina onnistu niin hyvin kuin aluksi ajattelee. Projektia suunniteltaessa on mahdotonta ottaa kaikkia asioita huomioon, varsinkin silloin, kun projektin toteutukseen käytettävät teknologiat eivät ole ennestään tuttuja.

4.1 Kehitysympäristö

Koodin kirjoitusta varten käytämme Visual Studio Code – nimistä ohjelmaa, joka on Microsoftin luoma ja ylläpitämä koodinkäsittelyohjelma. Visual Studio Code, eli VS Code, tukee montaa erilaista ohjelmointikieltä ja ohjelmointiin liittyvää työkalua. (Microsoft 2021.).

Jotta ohjelmiston koodi on turvallisessa paikassa varmuuskopioituna, ja jotta ohjelmiston kehitystä voi suorittaa muuallakin, kun vain yhdellä laitteella, luodaan tälle projektille oma GitHub – repositorio. GitHub on hyvä paikka projektien taltiointiin. Sillä on paljon käyttäjiä, joten sen käytettävyyteen on panostettu ja se on turvallinen ja nopea, sillä se käyttää projektien datan siirtoon suoraa SSH (Secure Shell Protocol) -yhteyttä. (GitHub. 2021).

4.2 Backend & tietokanta

Koska backend toimii todella tiiviisti tietokannan kanssa, tullaan tietokantaan liittyvät asiat ja backend logiikka rakentamaan limittäin. Tämä tekee kehitysprosessista mielekkäämmän, sillä tietokantakyselyitä sekä -rakennetta ei tarvitse suunnitella aivan loppuun asti.

4.2.1 Ensiaskeleet

Niin kun edellä on jo mainittu, backend-ohjelma tullaan koodaamaan Node.js JavaScript ajoympäristöä hyväksi käyttäen. Node.js tarjoaa oman paketinhallintaohjelman npm, jota käytämme alustamaan projektin helposti terminaalista. Tämä projektin alustus luo automaattisesti tärkeimmät tarvittavat tiedostot joita Node.js tarvitsee ajaakseen projektin koodin. Nämä tiedostot sisältävät projektin dataa, sekä muun muassa listan projektin käyttämistä riippuvuuksista, joita voidaan asentaa npm -paketinhallintaohjelmalla.

TypeScript on vahvasti tyypitetty kieli, joka tarkastaa syntaksin ohjelman kokoamisvaiheessa. Eli TypeScript ei ole ajettavaa koodia, vaan muuttuu JavaScriptiksi ajettaessa. (Micorosft 2021). Tämän takia on asennettava kehityksen aikainen TypeScript -riippuvuus npm -paketinhallintaohjelmaa käyttäen. Kehityksen aikaiset riippuvuudet ovat siis sellaisia npm -paketinhallintaohjelmalla asennettuja riippuvuuksia, joita käytetään vain ohjelmiston

kehityksen aikana. Nämä riippuvuudet eivät siis ole mukana ohjelmassa, kun sitä ajetaan tuotantoympäristössä (Npm. 2021). Toinen tärkeä kehityksen aikainen riippuvuus on ESLint. ESLint on tarkoitettu JavaScript koodin tarkastukseen, joka auttaa ohjelmistokehittäjää huomaamaan virheitä hyvissä ajoin. ESLint käyttää hyödykseen konfiguraatiotiedostoa, jonka ohjelmistokehittäjä lisää projektiin. Tämä konfiguraatiotiedosto tulee sisältämään kaikki ESLint:in tarvitsemat tarkastussäännöt. (OpenJS Foundation 2021.) Tämä mahdollistaa sen, että saa vielä tiukempaa tarkastusta tyypityksen onnistumisesta kehitysvaiheessa. Tärkeää on esimerkiksi välttää ”any”-tyyppiä niin paljon kuin mahdollista, sillä se antaa tiedon olla mitä tyyppiä tahansa, jolloin ohjelma on alttiimpi virheille. (Microsoft. 2021).

Npm -paketinhallintaohjelmalla asennetuista riippuvuuksista tulee huomioida vielä eräs tärkeä seikka TypeScriptin kannalta: TypeScript projektiin asennetut paketit eivät sisällä tyypitystä. Jos näitä riippuvuuksien sisältämiä kirjastoja tai tukiympäristöjä käyttää ilman TypeScriptin vaatimaa tyypitystä, tulee TypeScript antamaan näistä virheitä, eikä ohjelmaa voida suorittaa. Tätä varten jokaisen tuotantoympäristöön liittyvän paketin asennuksen yhteydessä on asennettava erillisellä käskyllä tyypitys tälle riippuvuudelle kehityksenaikaiseksi riippuvuudeksi. (Microsoft 2021.).

Kun TypeScriptiin liittyvät riippuvuudet on asennettu, voidaan aloittaa varsinainen kehitystyö. Koska palvelinpäässä tapahtuvien reittien hallintaan on ollut tarkoitus käyttää Express – tukiympäristöä, asennetaan se npm-paketinhallintaohjelmalla. Tämä riippuvuus ei tule kehityksen aikaiseksi riippuvuudeksi, vaan tulee olla mukana myös tuotantovaiheessa. TypeScript -ohjelmoinnissa nämä ajonaikaiset paketit tulee kutsua siinä tiedostossa, missä tämän kyseisen paketin toiminnallisuuksia tarvitaan, jotta saa tämän paketin toiminnallisuudet käyttöön. Tässä projektissa tulee olemaan useita eri tiedostoja, sekä kansioita, joten aletaan alusta asti ottamaan tämä kehityskulma huomioon.

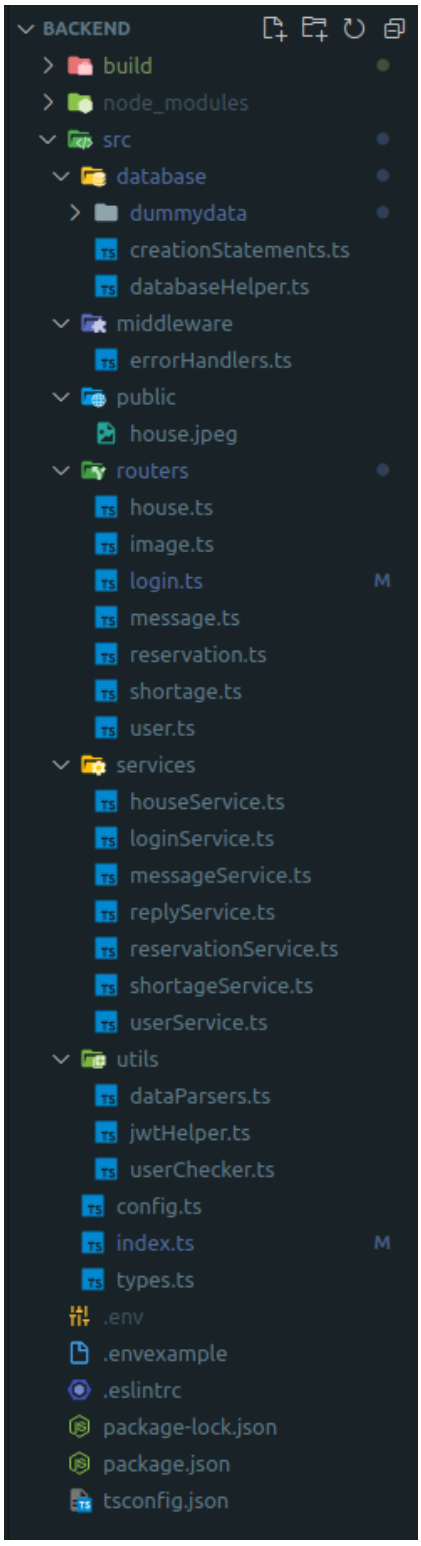
4.2.2 Projektin rakenne

“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” (Fowler & Beck 2019, 15.).

Ohjelmoinnissa on erityisen tärkeää pitää ohjelma selkeänä. Tämä helpottaa kehitysprosessia, sekä mahdollista tulevaisuuden jatkokehitystä huomattavasti. Koodin jakaminen omiin tiedostoihin, tai moduuleihin, tekee ohjelmasta selkeämmän kehittäjälle. Toiminnallisuuksien jakaminen moduuleihini myös mahdollistaa koodin uudelleenkäytettävyyden. (Flanagan 2011, 264.) Toiminnallisuuksien uudelleenkäytettävyys myös tekee projektista

mielekkäämmän, sillä ei tarvitse käyttää aikaa tehden samoja asioita uudelleen ja uudelleen.

Jotta backend -ohjelma pysyy selkeänä, erotellaan toiminnallisuudet heti eri moduuleihin. Tällöin jokainen moduuli on vastuussa vain omasta toimintatarkoituksestaan ja ongelmien etsiminen, sekä ominaisuuksien kehittäminen on yksinkertaisempaa.



Kuva 8. Projektin tiedostorakenne

Tarkastellaan hieman projektin kansioita, ja mitä ne sisältävät.

build

”build” -kansioon tulee TypeScriptin kokoama täysin JavaScript -kielinen tuotantoversio tästä ohjelmasta. Tämä build -kansion sisältä on se, joka lopuksi julkaistaan palvelimelle.

src

”src” -kansio sisältää kaiken ohjelman koodin. Tänne on varastoitu kaikki logiikka REST-rajapintaan kohdistuvien pyyntöjen hallinta, sekä niihin vastaaminen.

”database” -kansio sisältää kaikki tietokantaan liittyvät asiat. Siellä on testidataa kansiossa ”dummydata”, jota käytetään kehityksen alkuvaiheessa, ennen kuin tietokanta on käytettävissä. Tästä kansiosta myös löytyy tietokantaan liittyvä logiikka ja apufunktiot tietokantakyselyiden tekemiseen.

”middlewares” -kansio sisältää Express -tukiympäristölle ominaisena väliohjelmana palvelinpuolen virheenkäsittelijät. Express -tukiympäristön virallisen dokumentaation mukaan (Express 2017.) Express -tukiympäristöllä tehty sovellus on lähinnä useita väliohjelmia, joita kutsutaan vuoron perään. Tämä ohjelma pitää sisällään muitakin väliohjelmia, mutta ne ovat omissa kansioissaan, sillä ne on selkeyden takia järjestetty sekä nimetty tarkoituksensa mukaan, eikä ylätason kategoriansa mukaan.

”routers” -kansio on REST-rajapintaan tulevia pyyntöjä varten. Nämä tiedostot ”routers” -kansiossa ovat Express -tukiympäristön sovelluksille tyypillisiä väliohjelmia, jotka vastaanottavat palvelimelle tulevat pyynnöt ja suorittavat niihin liittyvät toiminnallisuudet, ja palauttavat tarvittavat asiat. (Express 2017.).

”services” -kansio pitää sisällään jokaiseen ”routers” -kansiossa olevaan väliohjelmaan liittyviä apufunktioita. Tämä on hyvä esimerkki projektin eri osien eriyttämisestä, niin että ohjelma on selkeämpi kokonaisuus.

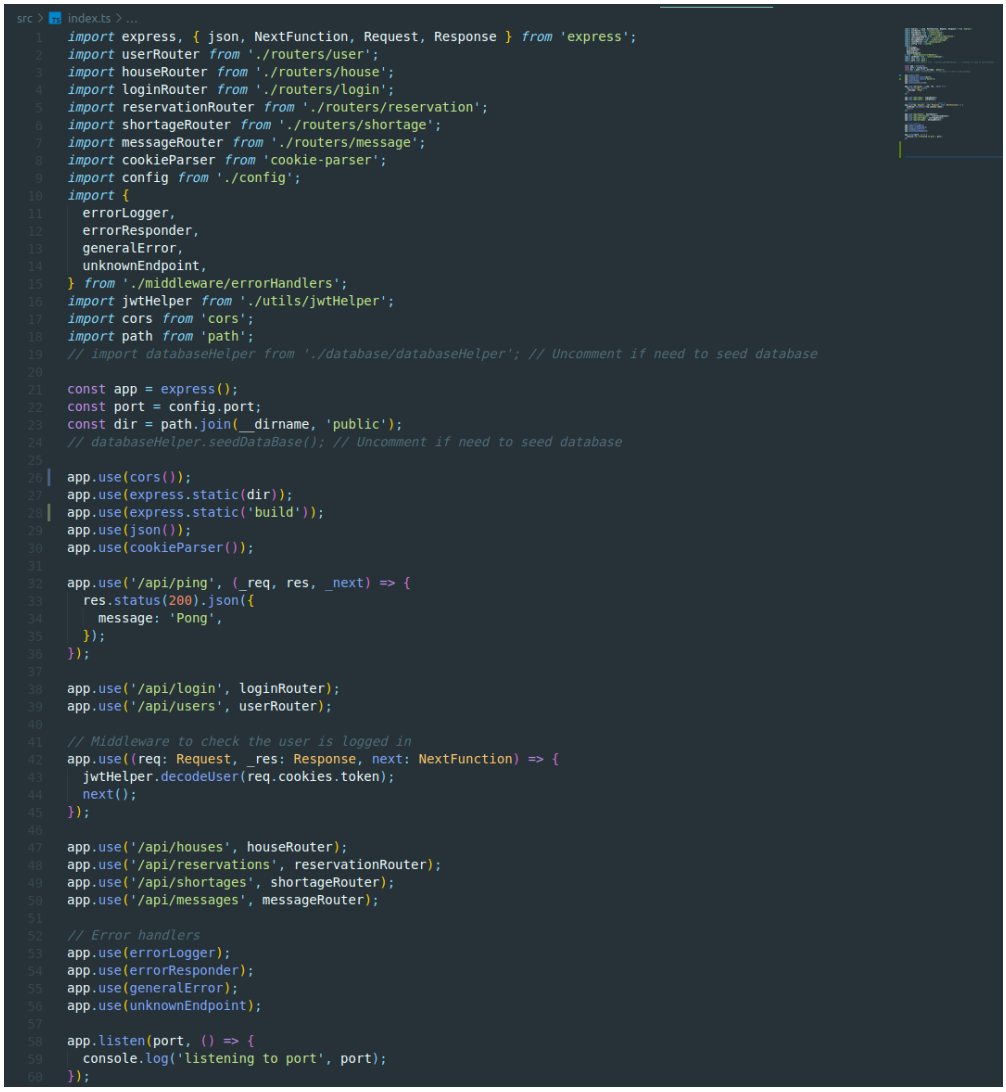
”utils” -kansiossa on tallennettu lisää apufunktioita, jotka ovat yleiskäytöllisimpiä kun ”services” -kansiossa olevat reittikohtaiset apufunktiot.

Loput tiedostot **src** -kansiossa ovat config.ts, index.ts, sekä types.ts. Config.ts on tarkoitettu hakemaan salattuja tietoja salaisesta .env -tiedostosta, joka on tarkoitettu arkaluontoisen tiedon tallentamiseen. Nämä tiedot ovat muun muassa tietokannan osoite, käyttäjätunnus ja salasana. Tämä .env -tiedosto tulee aina pitää vain ohjelmistokehittäjän omalla laitteella, eikä sitä tulisi koskaan julkaista internettiin. Index.ts -tiedosto on ohjelman

”päätiedosto”. Se on se tiedosto, jota kutsutaan kun ohjelma ajetaan Node.js -ympäristössä. Tämä tiedosto kutsuu kaikki muut ohjelman tiedostot, jolloin ohjelma on käytettävissä. Types.ts -tiedosto on TypeScriptille tyypillisten tietojen tyypitysten varastointia varten.

4.2.3 Index.ts

Backend -ohjelman päätiedosto index.ts sisältää suurimmilta osin väliohjelmien kutsuja. Tässä päätiedostossa luodaan Express -tukiympäristön tarjoama web-palvelimen käynnistyksen. Index.ts -tiedoston rivillä 61 alkava funktio suorittaa Express -tukiympäristön web palvelimen käynnistyksen. Tämä Expressin tarjoama funktio ottaa sisäänsä parametreina portin, sekä ”callback” -funktion. Express kuuntelee argumenttina annettua porttia, johon käyttöliittymältä tulevat kyselyt lähetetään. Tässä tapauksessa callback -funktio suoritetaan, kun portin kuuntelu on onnistuneesti aloitettu. Callback -funktio on toiselle funktiolle argumenttina annettu funktio, jonka alkuperäinen funktio suorittaa. (Mozilla 2021.).



```
src > index.ts > ...
import express, { json, NextFunction, Request, Response } from 'express';
import userRouter from './routers/user';
import houseRouter from './routers/house';
import loginRouter from './routers/login';
import reservationRouter from './routers/reservation';
import shortageRouter from './routers/shortage';
import messageRouter from './routers/message';
import cookieParser from 'cookie-parser';
import config from './config';
import {
  errorLogger,
  errorResponder,
  generalError,
  unknownEndpoint,
} from './middleware/errorHandlers';
import jwtHelper from './utils/jwtHelper';
import cors from 'cors';
import path from 'path';
// import databaseHelper from './database/databaseHelper'; // Uncomment if need to seed database

const app = express();
const port = config.port;
const dir = path.join(__dirname, 'public');
// databaseHelper.seedDataBase(); // Uncomment if need to seed database

app.use(cors());
app.use(express.static(dir));
app.use(express.static('build'));
app.use(json());
app.use(cookieParser());

app.use('/api/ping', (_req, res, _next) => {
  res.status(200).json({
    message: 'Pong',
  });
});

app.use('/api/login', loginRouter);
app.use('/api/users', userRouter);

// Middleware to check the user is logged in
app.use((req: Request, _res: Response, next: NextFunction) => {
  jwtHelper.decodeUser(req.cookies.token);
  next();
});

app.use('/api/houses', houseRouter);
app.use('/api/reservations', reservationRouter);
app.use('/api/shortages', shortageRouter);
app.use('/api/messages', messageRouter);

// Error handlers
app.use(errorLogger);
app.use(errorResponder);
app.use(generalError);
app.use(unknownEndpoint);

app.listen(port, () => {
  console.log('listening to port', port);
});
```

Kuva 9. index.ts -tiedoston sisältö

4.2.4 Väliohjelmat

Express -tukiympäristön väliohjelmien toimintaperiaate on simppeli. Jokaisen palvelimelle saapuvan pyynnön yhteydessä näiden väliohjelmien suorittaminen alkaa. Väliohjelmia käydään läpi yski kerrallaan siinä järjestyksessä, jossa ne on otettu pääohjelmassa, eli index.ts -tiedostossa, käyttöön. Kun väliohjelma suoritetaan, se joko palauttaa jotain vastauksen muodossa palvelimelle tehtyyn pyyntöön, tai sitten suorittaa jonkin toiminnon, ja menee seuraavaan väliohjelmaan next -funktiota hyödyntämällä. Jos next -funktiolle annetaan argumenttina jotain, sen oletetaan olevan virheen sisältävä argumentti, jolloin express siirtyy suoraan virheenkäsittelyyn liittyvään väliohjelmaan tämän argumentin kanssa. (Express 2017.).

Pääohjelmassa kutsutaan kaikki ohjelman käyttämät väliohjelmat. Osa väliohjelmista toiminnallisuuksia, joita kutsutaan npm -paketinhallintaohjelmalla ladatuista kirjastoista, mutta suurin osa väliohjelmista on itse kehitettyjä.

Backend ja frontend tässä projektissa kuuntelevat palvelimen eri portteja. Tämä käytännössä tarkoittaa sitä, että käyttöliittymän tehdessä pyyntöjä palvelimelle, tämä pyyntö ei tule siitä osoitteesta, jossa palvelimen backend ohjelma elää. Tämä liittyy Cross-Origin Resource Sharing, eli CORS, mekanismiin (Mozilla 2021.). CORS mekanismi on asia, joka tulee ottaa huomioon tätä ohjelmaa tehdessä, jotta palvelimelta saadaan dataa käyttöliittymään. Tätä varten index.ts -tiedostossa ensimmäisenä väliohjelmana on npm -paketinhallintaohjelmalla ladattu cors -väliohjelma, joka sallii palvelimen konfiguroinnin siten, että CORS mekanismin aiheuttamia ongelmia ei tule.

Koska tässä ohjelmassa tullaan käsittelemään dataa JSON -muodossa, on expressin saatava käyttöön myös JSON jäsennin. Tämä annetaan myös väliohjelmana, jonka express -tukiympäristö tarjoaa natiivina käyttäjälle. Tällöin saadaan käyttöön koko ohjelmassa JSON jäsennin, joka auttaa JSON datan käsittelyssä. (Express 2017).

Koska väliohjelmat suoritetaan järjestyksessä ja niiden tarjoamat työkalut otetaan myös käyttöön tässä järjestyksessä, on ohjelmaa kehitettäessä erittäin tärkeää pitää huolta, että väliohjelmien järjestys on looginen. Esimerkiksi väliohjelma joka tarkastaa pyynnön yhteydessä tulevan evästeen ja katsoo sen perusteella, onko käyttäjä kirjautuneena sisään ja palauttaa virheen, jos näin ei ole. Tämä väliohjelma ei toimisi oikein, jos pääohjelmassa ei ensin otettaisi käyttöön cookieParser -npm kirjaston tarjoamaa väliohjelmaa, joka on tarkoitettu siihen, että saadaan evästetiedot pyynnön yhteydessä käyttöliittymältä. Jos cookieParser -npm kirjaston väliohjelma otettaisiin myöhemmin käyttöön, kirjautumisen tarkastavalla väliohjelmalla ei olisi pääsyä tähän evästeiden tarkastukseen tarvittavaan

väliohjelmaan. Myöskin, jos itse kirjautumiseen tarvittava reitti ei olisi otettu väliohjelmana käyttöön ennen kirjautumisen tarkastamista, ei käyttäjä voisi ikinä kirjautua sisään, sillä ohjelma sallii pyyntöjen käsittelyn vain kirjautuneilta käyttäjiltä kaikille reiteille, jotka otetaan väliohjelmina käyttöön kirjautumistarkistuksen jälkeen.

Väliohjelmat, joiden vastuulla on virheiden käsittely, on pyritty tekemään niin että ne lähettävät mahdollisimman informatiivisia virheilmoituksia selaimeen, jos virhe sattuu. Tähän on tuonut helpotusta se, että luo ohjelman antamaan tietynlaisen virheen kussakin virhetilanteessa, jonka virheenkäsittelijä hoitaa. Tässä ohjelmassa asia on tehty niin, että tietyt oletetut virhetilanteet antavat virheen, jossa on omanlainen virheviesti. Tämän jälkeen virheen ottaa kiinni virheenkäsittelijä, joka toimii vaadittavalla tavalla riippuen virheviestistä. Tämä virheenkäsittelijän jakaminen omiin väliohjelmiin tukee ajattelutapaa siitä, että samaa asiaa ei tehdä moneen kertaan, jolloin ohjelman kehitys on mielekkäämpää. Kun kukin virheenkäsittelijä on oma väliohjelmansa, niin saadaan kullekin virheenkäsittelijälle relevantti virhe oikeaan paikkaan. Koska virheenkäsittelijät ovat väliohjelmia, niin väliohjelmille tuttuun tapaan virheenkäsittelijään argumenttina tullut virhe välitetään väliohjelmalta toiselle, jotta se saa sille relevantin kohtelun.

```
src > middleware > errorHandlers.ts > ...
import { NextFunction, Request, Response } from 'express';

export const errorLogger = (
  error: Error,
  _req: Request,
  _res: Response,
  next: NextFunction
): void => {
  console.error('ErrorLogger:', error);
  next(error);
};

export const errorResponder = (
  error: Error,
  req: Request,
  res: Response,
  next: NextFunction
): void => {
  if (error.message === 'parse-fail') {
    console.log('Data parser failed');
    res.status(400).json({
      error: 'Problem with input',
    });
  }
  if (error.message === 'no-user' || error.message === 'no-password') {
    console.log('Failed login attempt from:', req.socket.remoteAddress);
    res.status(200).json({
      error: 'Incorrect credentials',
    });
  }
  if (error.message.includes('duplicate')) {
    res.status(400).json({
      error: 'The username already exists',
    });
  }
  if (error.name === 'InvalidTokenError' || error.message === 'no-token') {
    res.status(401).json({
      error: 'Please login',
    });
  }
  if (error.message === 'db-error') {
    console.log('There was an error with the database');
    res.status(400).json({
      error: 'Database error',
    });
  }
  if (error.message === 'admin-error') {
    res.status(401).json({
      error: 'User is not admin for this house',
    });
  }
  if (error.message === 'no-userToAdd') {
    res.status(400).json({
      error: 'No such user',
    });
  }
  if (error.message === 'no-house') {
    res.status(400).json({
      error: 'No such house',
    });
  } else {
    next(error);
  }
};

export const generalError = (
  error: Error,
  _req: Request,
  res: Response,
  next: NextFunction
) => {
  res.status(500).json({
    message: 'Unknown error',
  });
  next(error);
};

export const unknownEndpoint = (_req: Request, res: Response) => {
  res.status(404).json({
    message: 'Unkwon endpoint',
  });
```

Kuva 10. Virheenkäsittelijät tiedostossa errorHandlers.ts

Yleisessä käytössä olevan web ohjelman on tärkeää pysyä toiminnassa koko ajan. Ohjelman ylläpitäjän on tärkeä tietää mahdollisista virhetilanteista ja ymmärtää mistä ne johtuvat. Tähän auttaa se, että ohjelma antaa ulos edes jonkinlaisia lokitietoja. Eri tyyppiset lokitiedot on hyvä eriyttää, niin ettei ohjelmasta tule ulos vain kaiken toiminnan lokitietojen sekamelska (Northwood 2018, 327.). Tässä ohjelmassa ei ole toimintoja niin vahvasti liitetty lokitietoihin, jolloin ohjelman käyttöä on hankala seurata. Sen sijaan on keskitytty siihen, että virhetilanteet antavat lokitietoja, joka ainakin auttaa ohjelman ylläpidossa, sillä voi tutkia jos jokin menee pieleen. Tätä varten virheiden käsittelyyn tarkoitetut väliohjelmat sisältävät virheisiin liittyvien lokitietojen antamisen, jonka läpi jokainen virhetapaus menee,

ja sitten se välitetään myöhemmille virheenkäsittelijöille, jotka tekevät virheelle sille sopivan ratkaisun.

4.2.5 Reitit

Koska express tukee reittien käyttöä väliohjelmina, on senkin kannalta jo intuitiivista ajatella reittien eriyttäminen ja nimeämiset alusta saakka. Tämä mahdollistaa myös sen, mitä koko projektissa paljon haetaan, eli asioiden selkeää eriyttämistä omiin osioihinsa. Tällöin saadaan keskitettyä jokaiseen reittiin liittyvät toiminnot helposti. Jokaisen reitin juuripolku määritellään pääohjelmassa, eli index.ts -tiedostossa. Tämän lisäksi voi määritellä tarkempia polkuja reitin sisällä.

Reittien hallintaan käytetään express tukiympäristön tarjoamaa Router -ominaisuutta. Tämä ominaisuus mahdollistaa juuri sen, että voidaan eriyttää eri reittien hallinta helposti toisiin tiedostoihin, jolloin tiedostorakenne saadaan pidettyä siistinä. Yksittäisessä reitinhallintatiedostossa voidaan ottaa käyttöön tämä Router -ominaisuus sitomalla se muuttujaan, ja sitten eksportoida tämä muuttuja, jonka kautta saadaan pääsy tähän Router -ominaisuuteen pääohjelmassa (Express 2017.).

Express tukiympäristön Router -ominaisuus sisältää valmiiksi jo eri tavat käsitellä erilaiset http -protokollan pyyntömetodit. Tämän takia samaan reittiin voi kohdistua useampaa erilaista kyselyä. Se, miten kysely käsitellään, riippuu siitä millä HTTP-pyyntömetodilla tämä pyyntö tulee. Näistä metodeista tässä ohjelmassa käytössä ovat GET, POST, PUT, DELETE. GET pyytää palvelinta lähettämään vastauksena jotain tietoa, POST lähettää tietoa selaimelta palvelimelle, joka palvelimen tulee käsitellä. PUT on tarkoitettu muokkaamaan tietoa, esimerkiksi tietokannassa, ja tämän pyynnön mukana tulee selaimelta jotain tietoa palvelimelle, joka sisältää sen tiedon, joka tulee asettaa korvattavan tiedon tilalle. DELETE on tehty poistamaan jotain tietoa, johon palvelin pääsee käsiksi. (Mozilla 2021.).

REST rajapintaa tehdessä on tärkeää kiinnittää huomiota nimeämiskäytäntöihin. Kun nimetään reittejä, joita ohjelma käyttää, on hyvä käyttää substantiiveja (Gupta 2.11.2021.). Tässä kannattaa myös kiinnittää huomiota siihen, mitä tietoa reitti palauttaa. Esimerkiksi reitti ’/api/houses on monikossa, sillä se liittyy kaikkien asuntojen käsittelyyn. Yksittäisen asunnon hakuun ei kuitenkaan ole omaa täysin erillistä reittiään, vaan yksittäinen viesti haetaan kaikkien asuntojen joukosta asunnon uniikilla ID tunnuksella.

src > routers > house.ts > ...

```ts
import express from 'express';
import houseService from '../services/houseService';
import { parseNewHouse, parseString } from '../utils/dataParsers';
import jwtHelper from '../utils/jwtHelper';

const router = express.Router();

// Get all houses for user
router.get('/', (req, res, next) => {
  const user = jwtHelper.decodeUser(req.cookies.token);
  houseService
    .getUsersHouses(user.id)
    .then((result) => {
      res.status(200).json(result);
    })
    .catch((err) => next(err));
});

// Get house by id
router.get('/:id', (req, res) => {
  const id = req.params.id;
  console.log('Hello from gethousebyid, with id:', id);
  houseService
    .getAllHouseDataById(id)
    .then((result) => {
      res.status(200).json(result);
    })
    .catch((err) => console.log(err));
});

// Add new house
router.post('/', (req, res, next) => {
  const token = <string>req.cookies.token;
  const user = jwtHelper.decodeUser(token);
  const houseToAdd = parseNewHouse({
    ...req.body,
    adminId: user.id,
  });
  houseToAdd.users.push({
    id: user.id,
    username: user.username,
  });
  houseService
    .addHouse(houseToAdd)
    .then((result) => {
      res.status(201).json({
        message: 'Added a new house!',
        houseAdded: result,
      });
    })
    .catch((err) => {
      next(err);
    });
});

// Grant user the access to a house
router.post('/:houseId/addUser', (req, res, next) => {
  const houseId = req.params.houseId;
  const admin = jwtHelper.decodeUser(req.cookies.token);
  const userToAddId = parseString(req.body.userId);
  houseService
    .addUserToHouse(admin.id, userToAddId, houseId)
    .then(() => {
      res.status(201).json({
        message: `Succesfully added user with id: ${userToAddId}`,
      });
    })
    .catch((err) => next(err));
});

// TODO: make it work in the DB end - gives error of foreign key constraint
// Remove house
router.delete('/:id', (req, res, next) => {
  const houseId = req.params.id;
  const user = jwtHelper.decodeUser(req.cookies.token);
  houseService
    .deleteHouse(houseId, user.id)
    .then(() => {
      res.end();
    })
    .catch((err) => next(err));
});

export default router;
```

Kuva 11. Reittien hallintaa house.ts -reittitiedostossa

Reittien hallintaa hoitava tiedosto on lähinnä vain todella ylätason logiikkaa. Täällä kutsutaan funktioita, joita on määritelty toisissa tiedostoissa, joiden avulla voidaan suorittaa eri toimintoja, joita eri kutsut eri reitteihin laukaisevat ja sen perusteella palauttavat tietoa käyttäjälle.

4.2.6 Tietokanta

Kaikki reitit reittienhallintatiedostoissa liittyvät toiminnallisuuksien kautta tietokantaan. Näillä reiteillä siis hallitaan kaikkea ohjelmassa liikkuvaa tietoa. Kaikki tieto mikä ohjelmassa liikkuu, on tallennettu tietokantaan. Ohjelman kehittämisen ensivaiheilla käytetään mallidataa, joka jäljittelee dataa mikä tulevaisuudessa tullaan tallentamaan tietokantaan. Tämä mallidata on tässä tapauksessa relevanttia ohjelmalle oikeastaan vain hyvin pienen hetken, kun testataan muutamat reitit ja todetaan ne toimiviksi.

Kun reitit on todettu toimiviksi, otetaan tietokanta käyttöön. Tässä ohjelmassa käytetään PostgreSQL -nimistä tietokantaa, niin kuin suunnitteluvaiheessa jo päätettiin. PostgreSQL asennetaan testiympäristöön ja määritetään sinne käyttäjä ja käyttäjälle salasana, niin kuin on suositeltavaa kaikkien tietokantojen kohdalla, jotta ne olisivat turvallisempia (Imperva 2021.).

Palvelinpään koodin tulee pystyä ottamaan yhteyttä tietokantaan, joten sen tulee tietää tietokannan osoite, portti, tunnukset ja tietokannan nimi, jota tässä ohjelmassa käytetään. Nämä tiedot ovat arkaluontoisia, ja ne tulee pitää salassa turvallisuussyistä, joten ne tallennetaan .env -tiedostoon. Tätä varten ohjelmassa on config.ts tiedosto, joka jakaa .env -tiedoston sisällön muulle ohjelmalle siellä missä näitä tietoja tarvitaan.

Jotta tietokannan käyttö olisi kehityksen aikana mielekästä, on käytännöllistä että ohjelmassa, joka tietokantaa käyttää, on valmiiksi tehty tietokannan luontilauseet. Nämä luontilauseet tallennetaan omaan tiedostoonsa ja niitä kutsutaan tarvittaessa. Tietokanta usein pysyy samana koko ohjelman eliniän, sillä sen struktuurin muuttaminen on hankalaa ja altistaa virheisiin (Northwood 2018, 260.). Kun luodaan tietokantatauluja, on silloin viimeistään suunniteltava tietokannan rakenne oikein, jotta vältytään mahdolliselta datastruktuurin muuttamiselta tulevaisuudessa.

Kun luo tietokantatauluja tietokantaan, on tärkeää ottaa huomioon niiden nimeämiskäytännöt. On suositeltavaa välttää lainausmerkkejä, isoja kirjaimia, sekä lyhenteitä. Taulut ja attribuutit tulisi nimetä informatiivisesti ja jos ne koostuvat useammasta sanasta, niin käyttää alaviivaa yhdistämään nämä sanat. Tietokannoissa on myös joitakin tietokannan omaan käyttöön varattuja sanoja, kuten ”user”, joten tässä ohjelmassa esimerkiksi käyttäjätiedot on tallennettu tauluun nimeltä ”app_user”. Kun luodaan tauluja, tulee myös muistaa eri taulujen väliset yhteydet. Esimerkiksi käyttäjällä sekä asunnolla on tässä sovelluksessa yhteys. Käyttäjällä voi olla pääsy useaan taloon, ja talolla voi olla useampia käyttäjiä. Tällaista yhteyttä varten on luotava aputaulu, joka auttaa pitämään yllä asuntojen ja käyttäjien välisen yhteyden. (Sarkuni 16.2.2014.).

Tietokannassa olevien taulujen yhteyksien vuoksi on jotkin kyselyt hieman monimutkaisempia. Näissä kyselyissä tulee ottaa huomioon molemmat taulut, sekä niiden yhdistävä taulu. Esimerkiksi kysely funktiossa houseUsersFromDb, joka palauttaa taloon liittyvien käyttäjien käyttäjänimen, sekä käyttäjän ID-tunnuksen näyttää seuraavalta:

```
const houseUsersFromDb = await pool.query(`
SELECT app_user_id, username from app_user AS a_user
FULL JOIN house_users AS h_users
ON a_user.app_user_id = h_users.user_id
FULL JOIN house AS house
ON house.house_id = h_users.house_id WHERE h_users.house_id = '${id}';
`);
```

Kuva 12. houseUsersFromDb -funktio

Tämän funktion sisältävä kysely ensin kertoo, että mitä valitaan, tämän jälkeen kerrotaan, että mistä tämä tieto saadaan ja annetaan tämän tiedon sisältävälle taululle alias, jotta säästytään ylimääräiseltä kirjoittamiselta. Tämän jälkeen käytetään ”FULL JOIN” liitostyyppiä, joka ottaa huomioon molemmista toisiinsa liitetyistä tauluista kaikki tapaukset, jotka liittyvät tähän kyselyyn, vaikka liitostaulun arvoissa olisi puutteita. Kyseessä olevalle liitostaululle annetaan myös alias. Tämän jälkeen kerrotaan, että miten kyselyn tulee yhdistää tietoja liitostaulussa ON parametrilla. Tämän jälkeen tehdään toinen liitos asuntoja sisältävään tauluun, ja lopuksi kerrotaan minkä perusteella tietoa palautetaan. Tässä tapauksessa perusteena on liitostaulusta löytyvä talon ID-tunnus, jolloin palautetaan kaikki käyttäjät, joiden talon ID tunnus on tämä haluttu talon ID-tunnus. (PostgreSQL 2021.).

Tässä ohjelmassa oleva tietokanta ei tule olemaan hirveän tehokas, jos sovellukseen liittyy valtava määrä käyttäjiä. Eräs tapa tehostaa tietokannan toimintaa on sen indeksoiminen. Tietokannan indeksointi tarkoittaa sitä, että tietokantaan lisätään sen sisältämään dataan riippumatonta rakennetta, jolloin alkuperäinen data ei siis muutu mihinkään. Indeksointi perustuu siihen, että tietokannassa on oma taulu, joka sisältää avain-arvo-pareina arvon, ja sen sijainnin tietokannassa. Tietokannan indeksointi nopeuttaa datan lukunopeutta, mutta hidastaa datan kirjoitusnopeutta, sillä kirjoitettaessa dataa, tulee indeksoinnin luomaa avain-arvo-pari taulua myöskin päivittää. (Kleppmann 2017, 71-72.).

4.2.7 Typescript ohjelmointikielenä backendissä

Koska Typescript on niin vahvasti tyypitetty kieli, mahdollistaa se sen, että kaikki data mikä liikkuu palvelinpäässä, saadaan pysymään oikeassa muodossa. Ohjelmoinnissa tärkeää on tietotyypit, sillä eri tietotyypit saavat erilaista kohtelua. Ja jotta tämä tietotyyppien

kohtelu ja ohjelman toiminen olisi johdonmukaista, on edullista että ohjelma on kokoajan tietoinen kunkin liikkuvan tiedon tyypistä.

Otetaan tarkasteluun funktio joka on vastuussa uuden asunnon lisäämisestä sovellukseen, nimeltä addHouse. Tämä funktio on asynkroninen, joka on JavaScriptille ominaista, ja siksi toimii myös hyvin TypeScriptin kanssa. Asynkronisia funktioita käytetään JavaScriptissä sellaisiin toimintoihin, jotka eivät ole heti valmiita, kuten vaikka tietokantakyselyihin. Asynkroniset funktiot JavaScriptissa palauttavat Promisen. Promise on funktion palauttama data, joka ei ole vielä valmis, vaan odottaa lisää tietoa, ja lopulta onnistuu, ja palauttaa tiedon, tai epäonnistuu ja palauttaa virheen (Mozilla 2021.). Kyseessä oleva funktio addHouse, sisältää asynkronisen funktion, joka suorittaa datan talletuksen tietokantaan, ja on tästä syystä julistettava itsekin asynkroniseksi. Ilmaisu ”await” saa asynkronisen funktion pysähtymään tähän kohdalle ja odottamaan datan lopullista ratkaisua, jolloin funktio käyttäytyy ikään kuin se olisi synkroninen. Luodessa funktioita TypeScript vaatii, että sille sisään annettava parametri tyypitetään, sekä funktiosta palautettava arvo tyypitetään. Tässä tapauksessa kerrotaan TypeScriptille, että parametrina saatu muuttuja ”house” on tyyppiä ”NewHouse”, ja funktio palauttaa Promisen, sillä se on asynkroninen, joka lopulta on tyyppiä ”House”. Jotta TypeScript tietäisi, minkälainen tietotyyppi on ”NewHouse”, se tulee määritellä erikseen.

```
export type NewHouse = Omit<House, 'id'>;
```

Kuva 13. NewHouse -tyyppi

Tyyppien teko TypeScriptissä on suositeltavaa tehdä käyttäen interfacea mielummin kun typea (Microsoft 2021.). Tässä tapauksessa on kuitenkin mielekästä sitoa tämä tyypin arvo type -tyyppialiakseen, koska ”NewHouse” tyyppi juonnetaan alkuperäisestä ”House” tyypistä siten, että sillä on kaikki samat arvot kuin ”House” tyypillä, lukuun ottamatta kenttää ”id”. Tämä on mahdollista TypeScriptin tarjoamalla työkalulla ”Omit” joka ottaa argumenttina sisäänsä sen tyypin, josta otetaan tietoja ja sen attribuutin, joka halutaan jättää tästä uudesta tyypistä pois.

```
export interface House {
  id: string;
  adminId: string;
  name: string;
  address?: string;
  maxResidents: number;
  imageUrl?: string;
  timestamp: string;
  users: UserForHouse[];
}
```

Kuva 14. House -tyyppi

”House” tyyppi on luotu käyttämällä TypeScriptin interface ominaisuutta ja sisältää attribuutit ”id”, joka on jokaiselle talolle uniikki tunnus, ”adminId”, joka on jokaisen talon ylläpitävän käyttäjän id -tunnus, ”address” on talon osoite, ”name” talon nimi, ”maxResidents” on maksimi määrä henkilöitä joita taloon kerralla mahtuu, ”imageUrl” on talon kuvan osoite, ”timestamp” on talon luontiajankohta ja ”users” on taloon liittyvät käyttäjät. Suurin osa näistä ”House” tyypin sisältävistä attribuuteista on tyyppiä ”string”, joka tarkoittaa merkkijonoa. Huomioitavaa on myös, että attribuutteihin ”address” ja ”imageUrl” kuuluu perään kysymysmerkki. Kysymysmerkki tässä tapauksessa tarkoittaa sitä, että tämä ei ole pakollinen tieto tälle tyypille. Käytännössä siis, jos luodaan objekti joka on tyyppiä ”House”, TypeScript ei anna virhettä jos tästä objektista puuttuu arvot ”address” ja ”imageUrl”, kun taas virhe annetaan jos mikä tahansa muista tälle tyypille määritellyistä arvoista puuttuu, tai on väärää muotoa. Esimerkiksi attribuutti ”maxResidents” on tietotyypiltään ”number”, joka tarkoittaa numeroa. Kun luodaan ”House” tyyppiä oleva objekti, ei ”maxResidents” attribuutin arvoksi voi silloin tulla merkkijonoa, tai mitään muutakaan tietotyyppiä kun vain se, joka sille on määritelty, eli numero.

```
const addHouse = async (house: NewHouse): Promise<House> => {
  try {
    const id = uuidv4();
    let houseToAdd: House = {
      id,
      ...house,
    };
    await databaseHelper.addHouseToDb(houseToAdd);
    const usersToAdd: UserForHouse[] = [];
    houseToAdd.users.forEach((item) => {
      if (usersToAdd.filter((user) => user.id === item.id).length === 0) {
        usersToAdd.push(item);
      }
    });
    usersToAdd.forEach((user) => {
      databaseHelper
        .addUserToHouse(user.id, houseToAdd.id)
        // eslint-disable-next-line @typescript-eslint/no-explicit-any
        .catch((err: any) => {
          console.log(err);
          throw new Error(err.message);
        });
    });
    houseToAdd = {
      ...houseToAdd,
      users: usersToAdd,
    };
    console.log('houseToadd:', houseToAdd);
    return houseToAdd;
  } catch (error) {
    throw new Error('db-error');
  }
};
```

Kuva 15. addHouse -funktio

Typescript tukee useita tietotyyppejä, kuten edellä mainitut ”number”, numero, sekä ”string”, merkkijono. Sekä näiden lisäksi muunmuassa ”boolean”, joka on totuusarvo, ”Array”, joka on lista, ”any”, joka voi olla mitä vaan tietotyyppiä ja ”undefined”, joka on määrittämätön arvo. Määrittäessä kustomoituja tyyppejä TypeScriptillä voidaan myös määrittää jollekin attribuutille vain rajattu määrä mahdollisia arvoja. Näitä kutsutaan nimellä ”union type” ja käytännössä ne on kovakoodattuja arvoja. Esimerkiksi mahdollisella kustomoidulla tyypillä nimeltä ”User” voisi olla attribuuttina ”role”, joka määrittäisin käyttäjän roolin ohjelmassa. Tälle ”role” attribuutille voisi antaa mahdollisiksi arvoiksi ”regularUser”, ”admin” tai ”superAdmin”. (Micorosft 2021.).

```
interface User {
  name: string;
  role: 'regularUser' | 'admin' | 'superAdmin';
}
```

Kuva 16. Union type -esimerkki tyypistä User

Kun jotain tietoa tulee backendiin, joko selaimelta uutena tietona, tai tietokannasta haettuna tietona, tulee TypeScriptille määrittä mitä tyyppiä tämä tieto on. TypeScript mahdollistaa tyypin määrittämisen suoralla tyypin julkistuksella. Tällä tavoin voidaan kertoa TypeScriptille suoraan, että tämä tieto on tätä tyyppiä, niin että sitä ei tarkasteta ollenkaan, vaan TypeScript olettaa, että ohjelmoija tietää tämän datan olevan juuri tätä tiettyä muotoa aina. Tämä ei kuitenkaan ole välttämättä hyvä tapa, sillä jos jotain meneekin jossain vaiheessa vikaan, tätä virhettä ei huomioida missään tapauksessa. Esimerkiksi selaimelta tuleva tieto saattaa tulla missä vain muodossa, jos frontendia tehdessä tätä tietoa ei validoida mitenkään. TypeScript kannustaa käyttämään Type Guard -nimistä tyypin tarkastusmenetelmää. Tässä on kyse siitä, että tarkastetaan että jokin tieto, esimerkiksi objekti, on juuri sitä tyyppiä, kun sen tuleekin olla. Jos Type Guard havaitsee tiedon olevan väärää muotoa, siitä annetaan virhe, ja mahdollisuuksien mukaan vastataan käyttäjälle, tai tehdään jokin muu asianmukainen virheenkäsittely. Näin saadaan turvallisesti hallittua tyyppejä ohjelmassa.

```
export const parseNewHouse = (obj: any): NewHouse => {
  console.log('Hello from parseNewHouse');
  const newHouse: NewHouse = {
    adminId: parseString(obj.adminId),
    name: parseString(obj.name),
    address: parseOptionalString(obj.address),
    maxResidents: parseNumber(obj.maxResidents),
    users: parseOptionalUserForHouseIdList(obj.users) || [],
    timestamp: parseDate(dayjs().locale('fi').format()),
    imageUrl: parseOptionalString(obj.imageUrl),
  };

  return newHouse;
};
```

Kuva 17. Type Guard -esimerkki tiedostosta dataParsers.ts Type Guard funktio parseNewHouse, joka määrittää selaimelta tulevan lisättävän talon attribuutit ja niiden arvot, että ne ovat oikeita ja tähän tyyppiin kuuluvia

Ohjelmaa tehdessä havaittiin että TypeScriptin käyttö tietokannan kanssa on erittäin mielekästä. Tässä käytetty PostgreSQL -tietokanta on tarkka sinne talletettavan tiedon tyypeistä. Tähän tietokantaan voi myös määritellä, että onko jokin tieto pakollinen, vai voiko tämän kentän jättää tyhjäksi, aivan niin kuin TypeScriptin kustomoitujen tyyppinen attribuuttien määrityksessäkin sai tiedon pakollisuuteen vaikuttaa. TypeScriptiä käytettäessä tietokantaan tallennettava data on jo valmiiksi tyypitetty, jolloin vältytään yllättäviltä tietokantavirheiltä.

4.2.8 Sovellukseen kirjautuminen, sekä muita avaintoimintoja

Sovelluksessa on isossa roolissa käyttäjät, niiden luomat resurssit, sekä käyttäjien pääsy kuhunkin resurssiin. Tällöin on erittäin olennaista, että käyttäjän tulee kirjautua sovellukseen, ja kaikki toimet sovelluksessa hänen tulee tehdä kirjautuneena. Sovelluksessa on tärkeää, että eri tietoja ja sisältöä, jota sovelluksessa on, ei voi muokata tai poistaa muut, kun ne lisännyt käyttäjä. Tämän takia sovelluksessa on tärkeää, että heti kirjautumisen yhteydessä generoidaan käyttäjälle jokin tapa tunnistautua jokaisen pyynnön yhteydessä, jotta sovellus on selvillä siitä, kuka palvelimelle pyynnön lähettää.

Käyttäjän kirjautumista varten on oma reitti backendissä, johon lähetetään käyttäjän tiedot selaimesta, kun käyttäjä kirjautuu sisään. Koska kaikkiin muihin reitteihin, paitsi käyttäjän kirjautumiseen ja uuden käyttäjän luontiin vaaditaan käyttäjän kirjautuneena olo, voidaan varmistua siitä, että ilman tunnuksia ei pääse tarkastelemaan tämän sovelluksen sisältämiä tietoja.

Kun sovellukseen luodaan uusi käyttäjä, selaimelta lähetetään käyttäjän syöttämät tiedot palvelimelle. Palvelimella käyttäjälle luodaan oma id käyttämällä uuid -kirjastoa. Tämä kirjasto on tarkoitettu uniikkien id -tunnusten generointiin ja se on asennettu npm -paketinhallintaohjelmalla. Käyttäjän syöttämä salasana on lähetetty palvelimelle pelkässä tekstimuodossa, ja ennen kun se tallennetaan tietokantaan, se tulee salata. Salasanan salaamiseen käytetään bcrypt -kirjastoa, joka on myös asennettu npm -paketinhallintaohjelmalla. Bcrypt tarjoaa funktiot salasanan salaamiseen, kuin myös salasanojen vertailuun sisäänkirjautumisvaiheessa (Npm 2021.). Bcrypt on tehokas salausalgoritmi, joka salaa tehokkaasti salasanat, joka hankaloittaa niiden arvaamista, jopa tietokoneiden toimesta (Hale 31.1.2010.). Kun nämä vaiheet on tehty, voidaan käyttäjän tiedot tallettaa tietokantaan.

Käyttäjän luotua käyttäjätunnuksen, hän voi kirjautua ohjelmaan, ja päästä käsiksi tietoihin, jotka ovat hänelle relevantteja.

```
import express from 'express';
import dayjs from 'dayjs';
import loginService from '../services/loginService';
import { parseLogin } from '../utils/dataParsers';
import { userIsLoggedIn } from '../utils/userChecker';

const router = express.Router();

// Login
router.post('/', (req, res, next) => {
  if (!userIsLoggedIn(req.cookies.token)) {
    const userFromInput = parseLogin(req.body);
    loginService
      .checkPassAndLogin(userFromInput)
      .then((jwt) => {
        console.log('jwt for login is!', jwt);
        const tokenExpirty = userFromInput.keepLoggedIn
          ? undefined
          : dayjs().add(1, 'hour').toDate(); // expirty in 1h

        res.cookie('token', jwt, {
          secure: false,    // HTTPS cerificate required
          httpOnly: false,  // this should later be changed
          expires: tokenExpirty,
        });

        res.status(200).send({
          message: 'Logged in!',
        });
      })
      .catch((err) => {
        next(err);
      });
  } else {
    res.status(200).json({
      message: 'User already logged in',
    });
  }
});

// Logout
router.post('/logout', (req, res, next) => {
  console.log(req.cookies.token);
  try {
    res
      .clearCookie('token')
      .json({
        message: 'Logged out!',
      })
      .end();
  } catch (error) {
    next(error);
  }
});

export default router;
```

Kuva 18. Käyttäjän kirjautumisen reittien hallintaa tekevä login.ts -reittitiedosto

Login.ts -reittitiedosto sisältää kaiken ylätason logiikan mitä kirjautumiseen liittyy. Ensin tarkastetaan, onko käyttäjä kirjautuneena jo sisään. Tämä perustuu siihen, että tarkastetaan onko käyttäjällä selaimessaan evästeinä ”token” niminen evästeen arvo. Jos tätä ei ole, niin siirrytään eteenpäin. Seuraavaksi tarkastetaan, onko käyttäjän syöttämät tiedot sen mukaiset, mitä niiden tulisi olla, eli suoritetaan validointi. Tässä käyttäjän syöttämät tiedot syötetään TypeScriptille ominaiseen Type Guardiin, joka tarkastaa tiedot, ja jos ne ovat oikein, niin tämä tieto sidotaan muuttujaan, jonka TypeScript nyt tietää olevan tyyppiä LoginUser. Jos kirjautumistietojen mukana ei tullut tietoa siitä, pidetäänkö käyttäjä kirjautuneena sisään, niin asetetaan oletukseksi arvo ”false”, joka on tietotyypiltään totuusarvo.

```
export interface LoginUser {
  username: string;
  password: string;
  keepLoggedIn: boolean;
}
```

Kuva 19. LoginUser -tyyppi

```
export const parseLogin = (obj: any): LoginUser => {
  console.log('Hello from parseLogin');
  const loginCredentials: LoginUser = {
    username: parseString(obj.username),
    password: parseString(obj.password),
    keepLoggedIn: parseOptionalBoolean(obj.keepLoggedIn) || false,
  };
  return loginCredentials;
};
```

Kuva 20. parseLogin -Type Guard

Tämän jälkeen tehdään salasanan tarkastus funktiolla checkPassAndLogin. Tämä funktio hakee ensin käyttäjän tietokannasta tämän käyttäjän käyttäjätunnuksella. Funktiossa suoritetaan vertailu käyttäjän syöttämän salasanan, ja tähän käyttäjätunnukseen sidotun salasanan välillä bcryptin tarjoamalla salasanojen vertailuun tarkoitetulla funktiolla. Jos käyttäjän salasana ei ole oikein, heittää funktio virheen, jolla on oma virheviesti. Tälle virheviestille on virheidenkäsittelyyn tarkoitetussa tiedostossa errorHandlers.ts (kuva 10) oma osio, joka ottaa virheviestin kiinni ja lähettää selaimeen virheviestin. Jos salasana on kuitenkin oikein, luodaan käyttäjätunnuksen perusteella tietokannasta palautuneesta käyttäjän tiedoista jwtHelper.ts -tiedoston funktiota ”encodeUser” apuna käyttäen JSON Web Token, joka palautetaan funktiosta. JSON Web Token on standardi tiedon siirtoon JSON objektina. Tämän tiedon voi salata algoritmilla, tai yleisellä tai yksityisellä avaimella. (Auth0 2016.). Kun JSON Web Token, eli JWT, palautetaan funktiosta, tulee se käytettäväksi reittienhallintatiedostossa sijaitsevaan sisäänkirjautumisen hoitavan reitin logiikkaan. Käyttäjä kirjautuessaan päättää, pidetäänkö tämä kirjautuneena sisään kunnes hän itse kirjautuu ulos, vai vanheneeko sisäänkirjautuminen. Tätä varten sidotaan mahdollinen sisäänkirjautumisen vanhenemisaika muuttujaan. Tämän muuttujan sisältämä arvo annetaan evästeelle, joka lähetetään selaimeen. Jos kirjautumisen vanhenemiselle ei annettu aikaa, ei evästeen sisältämä ”token” -arvon data vanhene, jolloin käyttäjää ei kirjata sovelluksesta ulos automaattisesti. Jotta JWT noudattaisi turvallisuuden suosituksia, JWT:n sisältämä eväste tulisi olla ”httpOnly”, eli ”httpOnly” attribuutin arvo evästeen määrittelevässä funktiossa tulisi olla ”true”, tällöin ohjelma on altis XSS hyökkäyksille (Chenkie 30.4.2020.). XSS, eli Cross-Site Scripting hyökkäykset ovat hyökkäyksiä, joissa hyökkääjä saa kaapattua käyttäjien sivulle syöttämiä tietoja oman haitallisen koodinsa avulla (PortSwigger 2021.). Tässä tapauksessa kuitenkaan ei käytetä ”httpOnly” evästeitä, sillä käyttöliittymän ensimmäinen versio tulee hyödyntämään suoraan evästeissä olevaa käyttäjän tietoa. Jos evästeet ovat ”httpOnly” ei niitä voida lähettää HTTP-protokollalla, vaan niihin vaadittaisiin suojattu versio, eli HTTPS-protokolla. Evästeisiin, jotka ovat ”httpOnly” ei myöskään

pääse käsiksi JavaScrpitillä, jota tässä sovelluksessa käyttöliittymä hyödyntää (Mozilla 2021.). Tämä tullaan korjaamaan myöhemmin jatkokehityksen aikana.

Evästeet ovat hyvä tapa välittää tietoa selaimelta lähetettyjen pyyntöjen yhteydessä, ilman, että käyttöliittymässä tarvitsee tehdä mitään toimia tämän vuoksi. Evästeet lähetetään pyyntöjen yhteydessä palvelimelle, jolloin palvelimella on pääsy näihin evästeisiin ja niitä voidaan palvelimen päässä helposti hyödyntää. (Mozilla 2021.).

Kun kirjautumisen hoitava logiikka on päästyt siihen pisteeseen, että sille on luotu JWT ja määritelty evästeen vanhenemisaika, tämä eväste lähetetään selaimelle viestin kera, joka kertoo onnistuneesta kirjautumisesta. Jos jokin kirjautumisen aikana menee pieleen, välitetään virhe virheenkäsittelijöille, jotka käsittelevät virheen asianmukaisella tavalla.

Jos käyttäjä haluaa kirjautua ulos, lähetetään selaimesta pyyntö reittiin ”/logout”, jolloin käyttäjän evästetiedoista poistetaan tähän kirjautumiseen liittyvä eväste, jolloin tulevien palvelimelle kohdistuvien pyyntöjen mukana ei tule evästeen sisältämää ”token” nimellä olevia arvoja, jolloin ohjelma ei palauta käyttäjälle mitään tietoja, vaan sen sijaan kehottaa käyttäjää kirjautumaan sisään.

```
import bcrypt from 'bcrypt';
import { LoginUser, User, UserForJwt } from '../types';
import databaseHelper from '../database/databaseHelper';
import jwtHelper from '../utils/jwtHelper';

const passWordIsCorrect = async (user: User, passwordFromInput: string): Promise<boolean> => {
  const passwordFromDb = user.password;
  return await bcrypt.compare(passwordFromInput, passwordFromDb);
};

const checkPassAndLogin = async (userFromInput: LoginUser): Promise<string> => {
  const userFromDb = await databaseHelper.getUserByUsername(userFromInput.username);
  if (!await passWordIsCorrect(userFromDb, userFromInput.password)) {
    throw new Error('no-password');
  }
  const userForJwt: UserForJwt = {
    id: userFromDb.id,
    fname: userFromDb.fname,
    lname: userFromDb.lname,
    username: userFromDb.username,
    email: userFromDb.email,
    role: userFromDb.role,
  };
  return jwtHelper.encodeUser(userForJwt);
};

export default {
  passWordIsCorrect,
  checkPassAndLogin,
};
```

Kuva 21. loginServices.ts -tiedosto

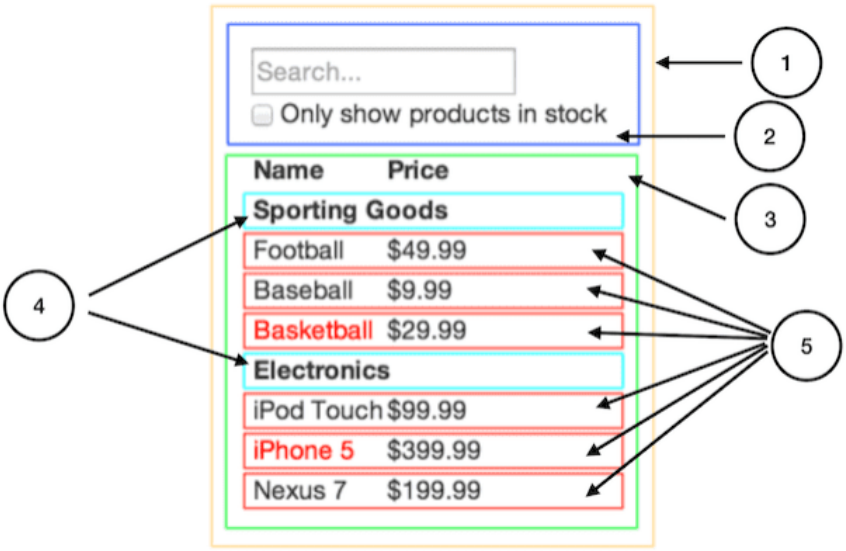
```
import jwt from 'jsonwebtoken';
import jwtDecode from 'jwt-decode';
import config from '../config';
import { UserForJwt } from '../types';

const secret = config.jwt_secret as string; // Casting the type to remove the possibility for 'undefined'

const encodeUser = (user: UserForJwt): string => {
  if (!user) {
    throw new Error('No user');
  }
  return jwt.sign(user, secret);
};

const decodeUser = (jwt: string): UserForJwt => {
  if (!jwt) {
    throw new Error('no-token');
  }
  return jwtDecode(jwt);
};

export default {
  encodeUser,
  decodeUser,
};
```

Kuva 22. jwtHelper.ts -tiedosto

4.3 Frontend

Sovelluksen käyttöliittymä tehdään käyttäen React -JavaScript kirjastoa. React helpottaa interaktiivisten käyttöliittymien tekoa, se perustuu erilaisten komponenttien tekoon, joita voidaan eriyttää ohjelman tiedostorakenteessa, jotta voidaan ylläpitää selkeää kansiorakennetta. Tämä helpottaa myös komponenttien uudelleenkäytettävyyttä. React komponentit ovat siis pieniä osia kokonaisesta näkymästä. Tällainen komponentti voi esimerkiksi olla käyttäjän luontiin tarkoitettu lomake. Tämä lomake on sivulla joka on vastuussa koko käyttäjän luontiin liittyvästä prosessista, mutta lomakkeen tehtävä on ainoastaan kerätä käyttäjän tiedot. Koska React komponentit voivat pitää sisällään erilaisia ”tiloja”, eli tässä tapauksessa muuttuja sisältää syöttökenttään syötettyä dataa. Kun ”tila” muuttuu, näkymä päivitetään uudelleen, jolloin mahdollistetaan se, että syöttökenttään syötetty tieto pysyy ajan tasalla, ja kun lomakkeen tiedot lähetetään palvelimelle selaimelta, ne lähtevät juuri oikeassa muodossa ja kokonaisina, niin kuin käyttäjä on ne syöttänyt. (React 2021.).

React -sovellus suositellaan hajottamaan pienempiin komponentteihin, joista näkymä rakennetaan. Yhden komponentin tulisi olla vastuussa vain yhdestä asiasta. Näitä komponentteja yhdistelemällä saadaan luotua kokonainen näkymä. (React 2021.).



Kuva 23. React -komponentti ajattelumalli (React 2021.)

Komponentit välittävät dataa propsien tai tilan avulla. Propsit ovat tietoja, jotka välitetään suoraan komponentille ylemmiltä komponenteilta. Tila on jonkin komponentin sisältävä data kullakin hetkellä. Jokin komponentti voi olla siis tietyssä ”tilassa” ja välittää tästä ”tilasta” löytyvän datan alemmille komponenteille. React komponenttien välillä data liikkuu vain yhteen suuntaan: ylhäältä alas.

On myös mahdollista luoda ohjelmaan jollekin muuttujalle globaali tila, joka on saatavilla koko ohjelmassa. Tätä varten on rakennettava erillinen tilan hallitsija, joka käsittää oman tilansa ja jolla on omat funktiot tilan hallintaan. Tässä on kyse Flux arkkitehtuurimallista. Flux arkkitehtuurimalli tarkoittaa käytännössä sitä, että yksittäiset komponentit eivät hallitse tietoa, vaan tieto on eriytetty kokonaan omaan hallitsijaansa, jossa tieto säilytetään ja missä tietoa muokataan. Tietoa ei muokata suoraan muuttamalla jonkin muuttujan arvoa, vaan tähän tilankäsittelijään ”lähetetään” uusi tieto, jolla korvataan vanhan tilan tieto. (Thisdot 14.10.2021).

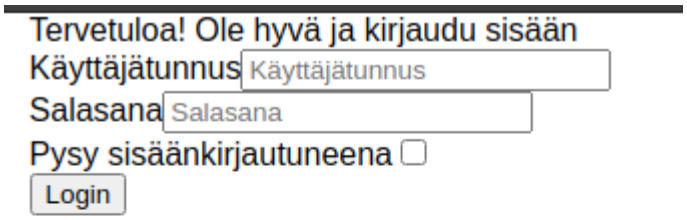
4.3.1 Käyttöliittymän toteutus

Käyttöliittymän tekoa aloittaessa on hyödyllistä käyttää npm -paketinhallinnan mukana asennettua npx -työkalua, joka on tehty automatisoimaan React -sovelluksen alustamisen. Koska React -sovellus on monimutkaisempi kokonaisuus, kun esimerkiksi aiemmin tehty lähes tyhjästä aloitettu backend -sovellus, on mielekästä että React -sovelluksen alustus on automatisoitu. Tälle alustuskäskylle voidaan antaa parametrina vielä jokin tietty pohja, minkä mukaan sovellus alustetaan, joten tässä tapauksessa kerrotaan npx -työkalulle, että tämä sovellus halutaan alustaa TypeScript pohjaan. Tällöin kaikki tarvittavat tiedostomuodot ovat jo valmiiksi oikeita, jotta ei tarvitse tehdä ylimääräistä työtä. (React 2021.).

Käyttöliittymä toteutetaan mockup -kuvia mukaillen. Jos käyttäjä ei ole kirjautuneena sisään, ensimmäinen näkymä on kirjautumisikkuna. Käyttäjä syöttää syöttökenttiin kirjautumistunnuksensa, jotka tallennetaan React -sovelluksen tilaan. Tästä kirjautuessa lähetetään POST-tyyppinen pyyntö palvelimelle ”/login” reittiin. Pyynnön mukana lähtee React -sovelluksen tilassa olevat kirjautumistiedot palvelimelle, jossa pyynnön mukana olevat kirjautumistiedot käsitellään ja annetaan käyttöliittymään vastaus tämän käsittelyn lopputuloksen perusteella.

```tsx
import React from 'react';
import { useStateValue } from '../state/state';
import loginHelper from '../utils/loginHelper';

// interface Credentials {
//   username: string;
//   password: string;
// }

const LoginView = () => {
  const [{ username, userPassword, stayLoggedIn }, dispatch] = useStateValue();
  // const [credentials, setCredentails] = React.useState<Credentials>({
  //   username: '',
  //   password: '',
  // });

  // console.log('here is cookies.get', Cookies.get('token'))

  const handleUsernameChange: React.ChangeEventHandler<HTMLInputElement> = (
    event
  ) => {
    dispatch({
      type: 'SET_USERNAME',
      payload: event.target.value,
    });
  };
  const handlePasswordChange: React.ChangeEventHandler<HTMLInputElement> = (
    event
  ) => {
    dispatch({
      type: 'SET_PASSWORD',
      payload: event.target.value,
    });
  };
  const handleStayLoggedInChange = () => {
    dispatch({
      type: 'STAY_LOGGEDIN',
      payload: !stayLoggedIn,
    });
  };

  const handleLogin = async (): Promise<void> => {
    try {
      const loginResponse = await loginHelper.login({
        username,
        password: userPassword,
        keepLoggedIn: stayLoggedIn,
      });
      if (loginResponse.message === 'Logged in!') {
        dispatch({
          type: 'LOGIN',
          payload: true,
        });
      } else {
        window.alert('Wrong credentials');
      }
    } catch (error) {
      console.log(error);
    }
  };

  return (
    <div>
      <div>Tervetuloa! Ole hyvä ja kirjaudu sisään</div>
      <div>
        <label htmlFor='username'>Käyttäjätunnus</label>
        <input
          onChange={handleUsernameChange}
          type='text'
          name='username'
          placeholder='Käyttäjätunnus'
        />
      </div>
      <div>
        <label htmlFor='password'>Salasana</label>
        <input
          onChange={handlePasswordChange}
          type='password'
          name='password'
          placeholder='Salasana'
        />
      </div>
      <div>
        <label htmlFor='stayLoggedIn'>Pysy sisäänkirjautuneena</label>
        <input
          checked={stayLoggedIn}
          onChange={handleStayLoggedInChange}
          type='checkbox'
          name='stayLoggedIn'
        />
      </div>
      <button onClick={handleLogin}>Login</button>
    </div>
  );
};

export default LoginView;
```

Kuva 24. LoginView.tsx -tiedosto, joka näyttää kirjautumisnäkymän ja on vastuussa käyttäjän sisäänkirjautumisesta



Kuva 25. Kirjautumisnäkymä, jota LoginView.tsx -tiedosto hoitaa

Tyylit tähän sovellukseen on suurin osa määritelty tyylitiedostoissa, joka on eriytetty selkeyden vuoksi omaan kansioonsa. Nämä tyylitiedostot helpottavat saman tyylin käyttämistä sovelluksen useassa tiedostossa, jotta samoja asioita ei tarvitse kirjoittaa useaan kertaan.

Käyttäjä näkee sovelluksesta asunnot, joihin tällä on pääsy ”Mökit” osiosta. Tässä osiossa näkyvät mökit on kutsuttu palvelimelta käyttäjän omien tietojen perusteella. Tämä osio ei näytä muita asuntoja, kun vain ne, johon käyttäjällä on pääsy. Valittaessa yhden mökin, käyttäjä näkee tähän valittuun mökkiin liittyvät tiedot. Näitä tietoja on mökin varaukset, siihen liittyvät käyttäjät, mökin puutteet, viestit sekä niiden vastaukset.

Kaikki tiedot mitä sovellukseen saadaan, haetaan backendistä. Nämä tiedot haetaan käyttöliittymälle käyttämällä Axios -nimistä kirjastoa, joka on tarkoitettu HTTP-kyselyiden tekoon. Axios saadaan asennettua projektiin käyttämällä npm -paketinhallintaohjelmaa. Kaikki kyselyt ovat asynkronisia, joten tämä tulee huomioida tehdessä funktioita, jotka suorittavat kyselyitä. Kun frontendistä tehdään kysely backendiin, backendin lähettämän vastauksen sisältämä data tallennetaan tälle datalle relevanttiin tilaan. Tästä tilasta komponentti pystyy tämän datan lukemaan ja asettamaan sen näkymään siihen kohtaan, jossa se tuleekin näyttää. Useat tiedot jota tässä sovelluksessa palvelin palauttaa ovat kokoelmia. Palvelin lähettää käyttöliittymälle dataa sisältävän listan. Tästä listasta tulee käyttöliittymän näyttää kaikki listan data. Jotta saadaan kaikki listan data näkymään sivulla samalla tavalla, on mielekästä käydä lista läpi indeksi kerrallaan ja tulostaa sieltä saman elementin sisällä indeksikohtainen tieto. Näin tehdään esimerkiksi kun tulostetaan lista taloista, joihin käyttäjällä on pääsy.

```
const HouseList = ({ houses }: Props) => {
  return (
    <div>
      {houses.map((item) => {
        return (
          <div style={{padding: '8px'}} key={item.id}>
            <Link style={{textDecoration: 'none'}} to={item.id}>
              <div className='HouseItem'>
                <img
                  src='http://localhost:3001/house.jpeg'
                  style={{ maxWidth: '64px', paddingRight: '16px' }}
                />
                <p>{item.name}</p>
              </div>
            </Link>
          </div>
        );
      })}
    </div>
  );
};
```

Kuva 26. HouseList.tsx -komponentin tapa tulostaa talolista

4.3.2 Tilan hallinta

Tässä käyttöliittymässä käytetään tilanhallintaan Flux -arkkitehtuurityypin mukaista globaalia tilanhallintaa. Tätä varten on ohjelmaan eriytetty oma ”state” -kansio, joka sisältää kaiken tilanhallintaan liittyvän logiikan sisältävät tiedostot. Tiedosto, joka hallitsee tilaa on nimeltään reducer.ts. Reducer.ts tiedosto sisältää tyypitykset jokaiselle ”Action” tyyppiselle käskylle, joiden avulla tilaa hallitaan. Tämän lisäksi on konkreettiset funktiot tilan muutoksille.

```
import {
  House,
  Message,
  MessageReply,
  Reservation,
  Shortage,
  State,
} from '../types';

export type Action =
  | {
      type: 'LOGIN';
      payload: true;
    }
  | {
      type: 'LOGOUT';
      payload: false;
    }
  | {
      type: 'SET_USERNAME';
      payload: string;
    }
  | {
      type: 'SET_PASSWORD';
      payload: string;
    }
  | {
      type: 'STAY_LOGGEDIN';
      payload: boolean;
    }
  | {
      type: 'HOUSES_FROM_DB';
      payload: House[];
    }
  | {
      type: 'SET_HOUSE_TO_EDIT';
      payload: House;
    }
  | {
      type: 'SET_RESERVATIONS';
      payload: Reservation[];
    }
  | {
      type: 'SET_SHORTAGES';
      payload: Shortage[];
    }
  | {
      type: 'SHORTAGEINPUT_OPEN';
      payload: boolean;
    }
  | {
      type: 'SET_NEW_SHORTAGE';
      payload: string;
    }
  | {
      type: 'SET_MESSAGES';
      payload: Message[];
    }
  | {
      type: 'SET_NEW_MESSAGE';
      payload: string;
    }
  | {
      type: 'SET_MESSAGE_REPLIES';
      payload: MessageReply[];
    }
  | {
      type: 'SET_REPLY_INPUT';
      payload: string;
    };
```

Kuva 27a. reducer.ts

```
export const reducer = (state: State, action: Action): State => {
  switch (action.type) {
    case 'LOGIN':
      return {
        ...state,
        isLoggedIn: true,
      };
    case 'LOGOUT':
      return {
        ...state,
        isLoggedIn: false,
      };
    case 'SET_USERNAME':
      return {
        ...state,
        username: action.payload,
      };
    case 'SET_PASSWORD':
      return {
        ...state,
        userPassword: action.payload,
      };
    case 'STAY_LOGGEDIN':
      return {
        ...state,
        stayLoggedIn: action.payload,
      };
    case 'HOUSES_FROM_DB':
      return {
        ...state,
        houses: action.payload,
      };
    case 'SET_HOUSE_TO_EDIT':
      return {
        ...state,
        houseToEdit: action.payload,
      };
    case 'SET_RESERVATIONS':
      return {
        ...state,
        reservations: action.payload,
      };
    case 'SET_SHORTAGES':
      return {
        ...state,
        shortages: action.payload,
      };
    case 'SHORTAGEINPUT_OPEN':
      return {
        ...state,
        shortageInputOpen: action.payload,
      };
    case 'SET_NEW_SHORTAGE':
      return {
        ...state,
        newShortage: action.payload,
      };
    case 'SET_MESSAGES':
      return {
        ...state,
        messages: action.payload,
      };
    case 'SET_NEW_MESSAGE':
      return {
        ...state,
        newMessageInput: action.payload,
      };
    case 'SET_MESSAGE_REPLIES':
      return {
        ...state,
        messageReplies: action.payload,
      };
    case 'SET_REPLY_INPUT':
      return {
        ...state,
        newMessageReplyInput: action.payload,
      };
    default:
      return state;
  }
};
```

Kuva 27b. reducer.ts

```
export const loginUser = (): Action => {
  return {
    type: 'LOGIN',
    payload: true,
  };
};

export const logoutUser = (): Action => {
  return {
    type: 'LOGOUT',
    payload: false,
  };
};

export const listHouses = (houses: House[]): Action => {
  return {
    type: 'HOUSES_FROM_DB',
    payload: houses,
  };
};

export const setHousetoEdit = (house: House): Action => {
  return {
    type: 'SET_HOUSE_TO_EDIT',
    payload: house,
  };
};

export const setReservations = (reservations: Reservation[]): Action => {
  return {
    type: 'SET_RESERVATIONS',
    payload: reservations,
  };
};

export const setShortages = (shortages: Shortage[]): Action => {
  return {
    type: 'SET_SHORTAGES',
    payload: shortages,
  };
};

export const setShortageInputOpen = (arg: boolean): Action => {
  return {
    type: 'SHORTAGEINPUT_OPEN',
    payload: arg,
  };
};

export const setNewshortage = (shortage: string): Action => {
  return {
    type: 'SET_NEW_SHORTAGE',
    payload: shortage,
  };
};

export const setMessages = (messages: Message[]): Action => {
  return {
    type: 'SET_MESSAGES',
    payload: messages,
  };
};

export const setNewMessageInput = (message: string): Action => {
  return {
    type: 'SET_NEW_MESSAGE',
    payload: message,
  };
};

export const setMessageReplies = (replies: MessageReply[]): Action => {
  return {
    type: 'SET_MESSAGE_REPLIES',
    payload: replies,
  };
};
```

Kuva 27c. reducer.ts

Jotta jokaisen komponentin tilalla olisi ohjelma käynnistyessä jokin alkutila, on sitä varten tehty tiedosto state.tsx, joka määrittelee alkutilat. Tässä tiedostossa myös luodaan ja otetaan käyttöön tilan Context. Context on Reactissa tapa välittää tietoa komponenttien välillä ilman propseja (React 2021). Jotta Context:illa luotu tilan globaalivälitystapa saadaan koko käyttöliittymälle käytettäväksi, on koko frontend sovellus ”käärittävä” tämän tilan tarjoavaan komponenttiin sisään index.tsx -tiedostossa.

```tsx
import React, { createContext, useContext, useReducer } from 'react';
import { State } from '../types';
import { Action } from './reducer';

const initialState: State = {
  isLoggedIn: false,
  username: '',
  userPassword: '',
  stayLoggedIn: false,
  houses: [],
  houseToEdit: {
    id: '',
    adminId: '',
    maxResidents: 0,
    name: '',
    timestamp: '',
    address: '',
    imageUrl: '',
    users: [],
  },
  reservations: [],
  shortages: [],
  shortageInputOpen: false,
  newShortage: '',
  messages: [],
  messageReplies: [],
  newMessageInput: '',
  newMessageReplyInput: '',
};

export const StateContext = createContext<[State, React.Dispatch<Action>]>([
  initialState,
  () => initialState,
]);

type StateProviderProps = {
  reducer: React.Reducer<State, Action>;
  children: React.ReactElement;
};

export const StateProvider = ({ reducer, children }: StateProviderProps) => {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <StateContext.Provider value={[state, dispatch]}>
      {children}
    </StateContext.Provider>
  );
};

export const useStateValue = () => useContext(StateContext);
```

Kuva 28. state.tsx

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import { BrowserRouter as Router } from 'react-router-dom';
import { StateProvider } from './state/state';
import { reducer } from './state/reducer';

ReactDOM.render(
  <StateProvider reducer={reducer}>
    <Router>
      <App />
    </Router>
  </StateProvider>,
  document.getElementById('root')
);
```

Kuva 29. index.tsx

4.4 Ohjelman julkaisu

Ohjelmaa kehitettäessä on se toiminut omassa kehitysympäristössään. Kun se julkaistaan palvelimelle, tulee siitä tehdä tuotantoversio. Tälle on valmiit työkalut sekä backendissä, että frontendissä. Ajamalla yksi komentosarja kummankin projektin juurikansiossa saadaan valmiit tuotantoversiot. React -sovelluksesta tulee yksi html-tiedosto, joka sisältää kaiken koodin mitä selain tarvitsee käyttöliittymän näyttämiseen. Tämä html-tiedosto tulee lisätä vielä backend -sovelluksen tiedostoksi, joka tarjoillaan palvelimen osoitteeseen mentäessä.

```
app.use(express.static('build'));
```

Kuva 30. Viittaus käyttöliittymän sisältävään ”build” kansioon backend sovelluksessa

Julkaisua varten luodaan uusi virtuaalikoneinstanssi Oraclen pilvipalveluun. Luontivaiheessa annetaan virtuaalikoneelle itse määrittelemäni ssh private key. Tämä avain mahdollistaa sen, että tälle virtuaalitietokoneelle voi kirjautua vain niiltä laitteilta, joilla on tämä sama salainen ssh avain. Tämä vahvistaa tietoturvaa, sillä tässä tapauksessa ei ole salasanaa jota mahdollinen hyökkääjä voisi arvata. Tämä myös helpottaa tietokoneelle SSH -yhteyden, ottamista, sillä kirjautuakseen ei tarvitse syöttää salasanaa. Käyttöjärjestelmäksi virtuaalikoneelle tulee Ubuntu 20.04, joka on Linux -käyttöjärjestelmä. Käytännössä tämä on virtuaalinen tietokone, joka on käynnissä Oraclen tarjoamalla palvelimella.

Kun virtuaalikoneinstanssi on luotu, voidaan siihen yhdistää SSH-yhteydellä. Heti luonnin jälkeen tietokoneessa ei ole mitään muuta, kun käyttöjärjestelmän toimintaa edeltävät

tiedostot. Jotta saadaan sovellus julkaistua palvelimella, tulee sinne asentaa PostgreSQL tietokantaohjelma, jota sovellus käyttää, sekä Node.js ajoympäristö, että PM2 Node.js prosessinhallintatyökalu. Jotta voidaan varmistua siitä, että sovellus tulee toimimaan tietokannan kanssa oikein, tulee sille konfiguroida käyttäjä, sekä itse tietokanta, jota tämä sovellus käyttää. Nämä tiedot kerrotaan sovellukselle salaisessa .env -tiedostossa joka elää sovelluksen kanssa palvelimella.

Ajoympäristöjen lisäksi tulee lisätä jokin ohjelma joka tarjoilee sivut palvelimelta. Tähän tarkoitukseen ladataan palvelimelle NGINX-niminen HTTP-palvelin. NGINX on avoimen lähdekoodin HTTP-palvelin, joka käyttää skaalautuvaa asynkronista arkkitehtuuria, joka on nopea ja tehokas. Se toimii myös välityspalvelimena, jollaisena sitä tässä tapauksessa käytetään. (NGINX 2021.)

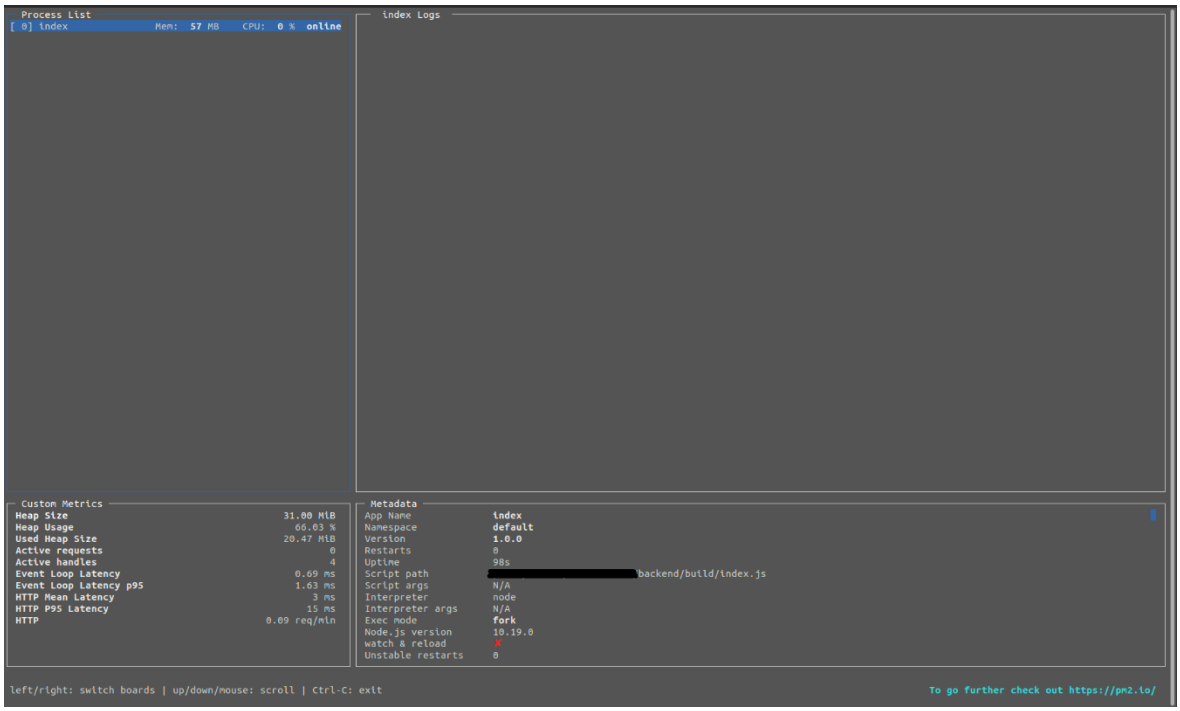
NGINX ei vaadi kovin paljon Linuxin sisäistä konfiguraatiota, jotta se palvelee Node.js sovelluksen mentäessä selaimella palvelimen osoitteeseen. Koska NGINX toimii myös välityspalvelimena, tässä tapauksessa on todella mielekästä ohjata palvelimeen tuleva liikenne palvelimen omassa verkossa toimivaan Node.js sovellukseen.

```
location / {
        proxy_pass http://localhost:3001;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
}
```

Kuva 31. NGINX konfiguroinnin hoitava osuus

Koska koko sovellus on määritetty kuuntelemaan pyyntöjä, jotka kohdistuvat porttiin 3001, määritetään NGINX ohjaamaan kaiken palvelimelle kohdistuvan liikenteen tähän porttiin.

Palvelimella on myös PM2 -ohjelma, joka pitää itse sovelluksesta huolta, jolloin sovellus on koko ajan toiminnassa ja näin NGINX voi huoletta ohjata liikennettä tälle sovellukselle.



Kuva 32. PM2 monitorointi näkymä sovelluksen ollessa toiminnassa

5. Huomioita ohjelman kehityksestä, sekä lopputulemasta

Tätä ohjelmaa tehdessä huomattiin paljon hyötyjä TypeScript-ohjelmointikielestä. Tyypitys aiheutti ajoittain lisätyötä ohjelmaa kehitettäessä, mutta se varmasti säästi aikaa virheiden etsimisessä. TypeScript oli tähän sovellukseen todella hyvin toimiva ja tuntui palvelevan tarkoitustaan mallikkaasti.

Relaatiotietokannan käyttö ohjelmassa osoittautui hyväksi vaihtoehdoksi, sillä tässä käytettiin jonkin verran taulujen välisiä suhteita, ja näin niitä oli helpompi hallinnoida. SQL-kieli aiheutti hieman lisätyötä, sillä kyselyt olivat niin uniikkeja, että ne tuli kaikki kirjoittaa alusta loppuun. Olisi ollut mielekästä kehittää tapa, jolla olisi voinut nopeuttaa kyselyiden kirjoittamista, ja asettaa niille tehokkaammin omia apufunktioita, mutta se ei ollut tämän sovelluksen kehittämisen pääpiste, joten sille ei annettu aikaa.

Itse sovellus toimii hyvin ja päästiin siihen lopputulokseen, mitä alussa lähdettiinkin tekemään. Kaikki ratkaisut mitä kehityksen aikana tehtiin, eivät kaikki palvele alkuperäistä tietoturvanäkemystä. Esimerkiksi JWT:n mahdollinen kaappaaminen ja lopullinen tietoturvan läpikäynti, jolla olisi havaittu suurimmat ongelmat, jäivät puuttumaan. Sovelluskehitykseen kuuluu kuitenkin ohjelman jatkuva kehittäminen, sekä parantaminen, joka on tämänkin ohjelman kannalta tulevaisuutta. Ohjelman kehitystä tullaan jatkamaan ja tiedettyjä ongelmia tullaan korjaamaan. Nyt jo tiedossa olevia ongelmia on muun muassa:

- JWT ”httpOnly” evästeeksi ja käyttäjän tieto otetaan käyttöliittymään suoraan palvelimelta
- React -sovelluksen komponenttipainotteisuutta tullaan parantamaan ennestään
- React -sovelluksen tilanhallintaan pieniä muutoksia, kuten osan tilanhallintaan liittyvistä arvoista otetaan vain komponentin paikalliseen käyttöön globaalin käytön sijaan
- Ohjelman tietoturvallisuuteen tullaan perehtymään tarkemmin
- Ohjelman protokolla siirretään salattuun HTTP-protokollaan, eli HTTPS-protokollaan

Tässä ohjelman versiossa käytetään ainoastaan HTTP-protokollaa, sillä suojatun HTTPS-protokollan käyttämiseen vaaditaan oma sertifikaatti, sekä palvelimen lisäkonfiguraatiota (SSL.com Support Team 12.10.2021).

Ohjelman lähdekoodi on nähtävissä GitHubissa osoitteessa https://github.com/olavle/summerhouse-log.

Lähteet

Auth0 2016. Get Started with JSON Web Tokens. Luettavissa: https://auth0.com/learn/json-web-tokens/. Luettu: 22.11.2021.

Chenkie, R. 30.4.2020. React Authentication: How to Store JWT in a Cookie. Luettavissa: https://medium.com/@ryanchenkie_40935/react-authentication-how-to-store-jwt-in-a-cookie-346519310e81. Luettu: 22.11.2021.

Cybersecurity & Infrastrucure Security Agency. 22.10.2021. Malware Discovered in Popular NPM Package, ua-parser-js. Luettavissa: https://us-cert.cisa.gov/ncas/current-activity/2021/10/22/malware-discovered-popular-npm-package-ua-parser-js. Luettu 27.10.2021.

Django Software Foundation, 2020. Luettavissa https://www.djangoproject.com/. Luettu: 26.10.2021.

Drkusic, E. 22.1.2020. Learn SQL: Types of relations. Luettavissa: https://www.sqlshack.com/learn-sql-types-of-relations. Luettu: 10.10.2021.

ESLint. 2021. Getting Started with ESLint. Luettavissa: https://eslint.org/docs/user-guide/getting-started. Luettu 17.11.2021.

Express. 2017. Guide. Luettavissa: https://expressjs.com/en/guide/routing.html. Luettu: 17.11.2021.

Facebook 2021. React. Luettavissa https://reactjs.org/. Luettu: 27.10.2021.

Facebook 2021. React Docs. Luettavissa https://reactjs.org/docs/getting-started.html/. Luettu: 22.11.2021

Flanagan, D. 2011. JavaScript: The Definitive Guide, Sixth Edition. O'Reilly Media. Sebastopol.

Fowler, M. & Beck, K. 2019. Refactoring, Improving the Design of Existing Code, Second Edition. Addison-Wesley. Boston.

GitHub 2021. Connect With SSH. Luettavissa: https://docs.github.com/en/authentication/connecting-to-github-with-ssh/about-ssh. Luettu: 17.11.2021.

Gupta, L. 2.11.2021. REST Resource Naming Guide. Luettavissa: https://restfulapi.net/resource-naming/. Luettu: 21.11.2021.

Hale, C. 31.1.2010. How To Safely Store A Password. Luettavissa: https://codahale.com/how-to-safely-store-a-password/. Luettu: 22.11.2021.

Huss, N. 8.10.2021. How Many Websites Are There in the World? [2021]. Luettavissa: https://siteefy.com/how-many-websites-are-there. Luettu: 21.10.2021.

Imperva 2021. Database Security. Luettavissa: https://www.imperva.com/learn/data-security/database-security/. Luettu: 21.11.2021.

Kleppmann, M. 2017. Designing Data-Intensive Applications. O'Reilly Media. Sebastopol.

Microsoft. 2021. TypeScript Documentation. Luettavissa: https://www.typescriptlang.org/docs/v. Luettu: 1.11.2021

Microsoft. 2021. Visual Studio Code Documentation. Luettavissa: https://code.visualstudio.com/docs. Luettu: 30.10.2021

Mozilla MDN Web Docs, 2021. Callback function. Luettavissa: https://developer.mozilla.org/en-US/docs/Glossary/Callback_function. Luettu 17.11.2021.

Mozilla MDN Web Docs, 2021. Cross-Oirigin Resource Sharing (CORS). Luettavissa: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS. Luettu: 20.11.2021.

MongoDB. 2021. Documentation. Luettavissa https://docs.mongodb.com/manual. Luettu: 26.10.2021.

Mongoose. 2021. Documentation. Luettavissa: https://mongoosejs.com/docs/guide.html. Luettu: 26.10.2021.

NGINX. 2021. Wiki. Luettavissa: https://www.nginx.com/resources/wiki/. Luettu: 22.11.2021.

Node.js. 2021. Introduction to Node.js. Luettavissa: https://nodejs.dev/learn. Luettu: 26.10.2021.

Node-Postgres Documentation. 2021. Luettavissa: https://node-postgres.com/. Luettu: 1.11.2021

Northwood, C. 2018. The Full Stack Developer Your Essential Guide to the Everyday Skills Expected of a Modern Full Stack Web Developer. Appress. Berkley.

Npm. 2021. Docs. Luettavissa: https://docs.npmjs.com/. Luettu: 17.11.2021.

Npm. 2021. node.bcrypt.js. Luettavissa: https://www.npmjs.com/package/bcrypt/. Luettu: 22.11.2021.

OpenJs Foundation. 2021. Luettavissa: https://openjsf.org/projects/. Luettu: 21.10.2021.

Oracle 2021. What is a Relational Database (RDBMS)?. Luettavissa: https://www.oracle.com/database/what-is-a-relational-database/. Luettu: 26.10.2021

Oracle 2021. MySQL 8.0 Reference Manual. Luettavissa: https://dev.mysql.com/doc/refman/8.0/. Luettu: 26.10.2021.

PM2 2021. Documentation. Luettavissa: https://pm2.keymetrics.io/. Luettu 17.11.2021

PortSwigger 2021. Cross-Site scripting. Luettavissa: https://portswigger.net/web-security/cross-site-scripting. Luettu: 22.11.2021.

The PostgreSQL Global Development Group 2021. About. Luettavissa: https://www.postgresql.org/about/. Luettu: 26.10.2021.

The PostgreSQL Global Development Group 2021. Documentation. Luettavissa: https://www.postgresql.org/docs/13/index.html. Luettu: 20.10.2021.

Richards, M. & Ford, N. 2020. Fundamentals of Software Architecture. O'Reilly Media. Sebastopol.

Sarkuni, S. 16.2.2014. How I write SQL, Part 1: Naming Conventions. Luettavissa: https://launchbylunch.com/posts/2014/Feb/16/sql-naming-conventions/. Luettu: 21.11.2021.

SSL.com Support Team. 12.10.2012. What is HTTPS?. Luettavissa: https://www.ssl.com/faqs/what-is-https/. Luettu 23.11.2021.

Thisdot. 14.10.2021. Creating a Global State with React Hooks. Luettavissa: https://www.thisdot.co/blog/creating-a-global-state-with-react-hooks Luettu: 22.11.2021.

Torppa, T. Peuraniemi, T. & Rapo, J. Full Stack Open -internetkurssi Osa 9. Helsingin Yliopisto, Houston. Luettavissa: https://fullstackopen.com/en/part9. Luettu: 3.11.2021

Zammetti, F. 2020. Modern Full-Stack Development. Appress. Berkley.

Ziolkowski, D. 5.1.2021. Top 11 Node.js security best practices. Sqreen. Luettavissa: https://blog.sqreen.com/nodejs-security-best-practices/. Luettu: 21.10.2021.