OULUN AMMATTIKORKEAKOULU

Niko Sotka

PROJEKTINHALLINTAJÄRJESTELMÄ

PROJEKTINHALLINTAJÄRJESTELMÄ

Niko Sotka
Opinnäytetyö
Syksy 2021
Tietojenkäsittely
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietojenkäsittelyn tutkinto-ohjelma

Tekijä(t): Niko Sotka
Opinnäytetyön nimi: Projektinhallintajärjestelmä
Työn ohjaaja(t): Pekka Ojala
Työn valmistumislukukausi ja -vuosi: Syksy 2021 Sivumäärä: esim. 49

Tämän opinnäytetyön tarkoituksena oli kehittää projektinhallintajärjestelmä, johon kuuluu web-pohjainen asiakasohjelma, ohjelmistorajapinta ja tietokanta. Työn tavoitteena oli oppia uutta ja kehittää omia taitoja full stack -kehittäjänä.

Opinnäytetyön tietoperusta sisältää yleistä tietoa toteutuksessa käytetyistä teknologioista ja menetelmistä sekä kirjastoista ja paketeista, jotka olivat toteutuksen kannalta tärkeässä roolissa. Käytännön osuudessa käydään läpi järjestelmän toteutusta ja varsinkin asiakasohjelman ulkoasua ja toiminnallisuutta.

Opinnäytetyön tuloksena oli toimiva järjestelmä, jossa oli lähes kaikki suunnitellut toiminnallisuudet ja jossa käyttäjä pystyi luomaan projektin sekä projektille tehtäviä, hallinnoimaan niitä sekä kutsumaan muita käyttäjiä liittymään projektiin. Järjestelmä vaatii vielä paljon kehitystä ennen kuin voidaan sanoa sen olevan valmis.

Asiasanat: React, Ruby on Rails, PostgreSQL, web-sovellus, ohjelmistorajapinta, projektinhallinta

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Business Information Systems

Author(s): Niko Sotka
Title of thesis: Project management system
Supervisor(s): Pekka Ojala
Term and year when the thesis was submitted: Autumn 2021
Number of pages: 49

The topic of this thesis was to develop a project management system that includes web-based client application, application programming interface and database. The aim was to learn more and develop as a full-stack developer.

The theoretical section of this thesis includes general information about technologies and methods used in the implementation as well as information about libraries and packages that were important from an implementation perspective. The practical section reviews the implementation of the system and in particular the layout of the client application and its functionality.

The result of this thesis was a functional system with almost all the planned functionalities and where a user can create project and tasks for the project, manage them, and invite other users to join the project. The system still needs a lot of development before it can be said to be ready.

Keywords: React, Ruby on Rails, PostgreSQL, web application, application programming interface, project management

SISÄLLYS

SANASTO

API	Application Programming Interface eli ohjelmointirajapinta
DOM	Lyhenne, jolla tarkoitetaan Document Object Model -mallia. Se on tapa kuvata HTML-dokumentti puurakenteena, jossa dokumentin osat kuvataan olio-solmuina
Renderointi	(myös renderöinti tai hahmonnus) tarkoittaa tiettyjen kappaleiden tai kuvien esittämistä näytöllä
JWT	Lyhenne sanoista JSON Web Token
Modaali	Komponentti, joka renderoi lapsisolmunsa taustakomponentin eteen
Props	React-kirjaston arvoja, joita välitetään React-komponenteille. Ominaisuuksia, jotka pitävät sisällään tietoa, jota komponentti voi käyttää.
Rails	Lyhenne, jolla tarkoitetaan Ruby on Rails-sovelluskehystä
String	Merkkijono, joka on ohjelmointikielen tietotyyppi, jossa on jono peräkkäisiä merkkejä eli tekstiä
UUID	Lyhenne sanoista universally unique identifier. 128-bittinen yksilöintitunniste

1 JOHDANTO

Tämän opinnäytetyön aiheena on projektinhallintajärjestelmän toteuttaminen, jossa asiakasohjelma on ReactJS-kirjastolla toteutettu responsiivinen web-sovellus, ohjelmointirajapintana toimii Ruby on Rails -ohjelmistokehyksellä toteutettu sovellus sekä tietokantajärjestelmänä PostgreSQL. Työllä ei ole toimeksiantajaa, vaan sen tavoitteena on syventää omaa osaamista ja kehittää ohjelmointitaitoja. Tarkoituksena on saada valmiiksi ensimmäinen versio järjestelmästä, jossa käyttäjä voi luoda projekteja, lisätä niihin jäseniä sekä tehtäviä ja hallinnoida niitä.

Asiakasohjelman käyttöliittymä koostuu useasta näkymästä ja niiden komponenteista, jotka pitävät sisällään sekä staattista että dynaamista sisältöä. Sovelluksessa on paljon interaktiivisuutta, jonka mukaan esimerkiksi toteutetaan rajapintakyselyitä ja sisältö muuttuu. Ohjelmointirajapinta taas ottaa vastaan asiakasohjelmasta tulevia rajapintakyselyitä, käsittelee saatua dataa, toteuttaa CRUD-operaatioita ja palauttaa käsitellyn vastauksen.

Tietoperustassa käydään läpi yleisellä tasolla järjestelmän toteutuksessa käytettyjä teknologioita ja menetelmiä, pohjalla toimivia ohjelmointikieliä ja niiden laajennuksia sekä toteutuksen kannalta suuressa merkityksessä olevia kirjastoja. Toiminnallisessa osiossa käydään pääsääntöisesti yleisellä tasolla läpi sovellusten sekä tietokannan toteutusta ja tuloksia. Joidenkin asiakasohjelman komponenttien toiminnallisuus on monimutkaisempaa kuin toisten, joten niiden toimintaa avataan syvemmin, mutta pelkästään asiakasohjelman pitäessä sisällään tuhansia rivejä kirjoitettua koodia, koodin syvällisempi tarkastelu päätettiin jättää tekemättä.

2 FRONTEND

2.1 ReactJS

React on alkujaan Facebookilla työskennelleen ohjelmistoinsinöörin Jordan Walken luoma JavaScript-kirjasto käyttöliittymien kehittämiseen. Vuonna 2013 React-kirjastosta tuli avointa lähdekoodia ja tuohon aikaan sen kuvailtiin olevan MVC-arkkitehtuurimallin näkymä eli View-osa. Tästä eteenpäin React-kirjasto kasvatti suosiotaan ja esimerkiksi Netflix ilmoitti vuonna 2015 käyttävänsä sitä käyttöliittymänsä kehityksessä. (Porcello & Banks 2020, luku 1.)

React-kirjastossa käyttöliittymän eri osat pilkotaan komponenteiksi ja nämä komponentit pitävät sisällään logiikan, miten näkymää käsitellään sekä itse näkymän. Koska DOM-manipulaatio on kallis toimenpide ja sitä pitäisi tehdä mahdollisimman vähän, React on kehitetty tällä ajatukselle tarjoamalla kehittäjälle virtuaalinen DOM renderointia varten varsinaisen DOM-rakenteen sijaan. React erottaa virtuaalisen DOM-rakenteen ja varsinaisen DOM-rakenteen erot ja suorittaa vain tarvittavan määrän DOM-operaatioita halutun lopputuloksen saavuttamiseksi. (A M & Sonpatki 2016, luku 1.)

Komponentit ja koukut

React-kirjastossa on kahdenlaista tapaa luoda komponentti – käyttämällä funktioita tai luokkaa. Suurin ero näiden välillä on niiden syntaksi. Funktionaaliset komponentit luodaan käyttämällä tavallisia JavaScript-funktioita, jotka ottavat vastaan ominaisuuksia eli props-arvoja parametrina ja palauttavat React-komponentin. Luokalliset komponentit luodaan käyttämällä JavaScriptin luokkasyntaksia ja niiden täytyy periä React.Component-luokan ominaisuudet.

Luokkakomponenteissa voidaan käyttää React.Component-luokan tarjoamaa state-oliota, joka ei ole mahdollista funktionaalisessa komponentissa. State eli tila pitää sisällään komponenttikohtaista dataa, joka voi muuttua ajan myötä ja on käyttäjän määrittelemä. Luokkakomponenttia käytettäessä on aina suotavaa välittää ominaisuudet constructor-metodille ja sen sisällä React.Component-luokalle käyttämällä super-metodia. Muita React.Component-luokalta perittäviä ominaisuuksia ovat

muun muassa LifeCycle-metodit ja näistä render-metodi on ainoa pakollinen metodi luokallisessa komponentissa. (Thakkar, M 2020, luku 2; React 2021a, viitattu 30.9.2021.)

Esimerkkikoodissa (kuvio 1) App-komponenttiin, joka on luokallinen komponentti ja toimii juurikomponenttina, alustetaan tila string-tyyppisellä arvolla. App-komponenttiin tuodaan luokallinen sekä funktionaalinen komponentti ja molemmille välitetään tilassa määritetty arvo textProp-nimisenä ominaisuutena. Lisäksi App-komponentissa on painike-elementti, jota painamalla tilassa määritetty arvo päivitetään uuteen arvoon.

```
1  import React, { Component } from "react";
2  import "./styles.css";
3
4  import CustomClass from "./CustomClass";
5  import CustomFunctional from "./CustomFunctional";
6
7  export default class App extends Component {
8    constructor(props) {
9      super(props);
10     // Alustetaan komponentin tila
11     this.state = {
12       rootStateValue: "Alustusarvo"
13     };
14   }
15   // Ainoa pakollinen LifeCycle-metodi luokka-komponentissa
16   // Palauttaa käyttäjän näkemän komponentin
17   render() {
18     // Destructuring assignment
19     // Puretaan tilasta tarvittava arvo
20     const { rootStateValue } = this.state;
21
22     return (
23       <div className="wrapper">
24         <CustomClass textProp={rootStateValue} />
25         <CustomFunctional textProp={rootStateValue} />
26         <button
27           onClick={() => {
28             this.setState({
29               rootStateValue: "click changed text"
30             });
31           }}
32         >
33           Change state
34         </button>
35       </div>
36     );
37   }
38 }
```

KUVIO 1. Esimerkkikoodi juurikomponenttina toimivasta App.js-tiedostosta

CustomClass-komponentin esimerkkikoodissa nähdään, kuinka komponentille välitettyä ominaisuutta voidaan hyödyntää (kuvio 2). Komponentti on hyvin yksinkertainen eikä siinä alusteta komponentin tilaa tai sidota metodeja (bind), joten constructor-metodia ei tarvita.

```
1  import React, { Component } from "react";
2
3  export default class CustomClass extends Component {
4    render() {
5      // Destructuring assignment
6      // Puretaan props-oliosta tarvittava arvo
7      const { textProp } = this.props;
8
9      return <p>{`Luokalliselle ominaisuutena välitetty arvo: ${textProp}`}</p>;
10   }
11 }
12
```

KUVIO 2. CustomClass.js-tiedoston esimerkkikoodi

CustomFunctional-komponentin esimerkkikoodissa komponentissa hyödynnetään sille välitettyä textProp-ominaisuutta renderoimalla se käyttäjälle näkyviin (kuvio 3). Tämän lisäksi komponentissa käytetään useState-koukkua tilan määrittämiseen ja se alustetaan kokonaisluvuksi nolla, jota käyttäjä voi kasvattaa painike-elementtiä painamalla.

```
1  import React, { useState } from "react";
2
3  export default function CustomFunctional(props) {
4    // useState-hook
5    // Tila alustetaan nollaksi
6    const [count, setCount] = useState(0);
7
8    // Destructuring assignment
9    // Puretaan props-oliosta tarvittava arvo
10   const { textProp } = props;
11
12   return (
13     <div>
14       <p>{`Funktionaaliselle ominaisuutena välitetty arvo: ${textProp}`}</p>
15       <button onClick={() => setCount(count + 1)}>
16         {`Painiketta painettu ${count} kertaa`}
17       </button>
18     </div>
19   );
20 }
```

KUVIO 3. CustomFunctional.js-tiedoston esimerkkikoodi

Hooks eli koukut mahdollistavat useiden luokallisista komponenteista tutun ominaisuuden käytön myös funktionaalisessa komponentissa. Aikaisemmin funktionaaliset komponentit tunnettiin myös

nimellä ”stateless components” eli tilattomat komponentit, koska niille ei voitu määrittää komponenttikohtaista tilaa.

Nykyään useState-koukku mahdollistaa React-tilan lisäämisen funktionaaliseen komponenttiin. Käytettäessä useState-koukkua, sille välitetään ainoastaan yksi argumentti, joka on sen alkutila. Toisin kuin luokallisessa komponentissa, tilan ei tarvitse olla JavaScript-olio, vaan se voi olla esimerkiksi numero tai merkkijono. useState palauttaa arvoparin; nykyisen tilan sekä funktion, jolla sitä voi päivittää. (React 2021b, viitattu 30.9.2021.)

Koukut ovat JavaScript-funktioita, mutta niitä käytettäessä on syytä noudattaa kahta sääntöä. Koukkuja käytetään vain ylätasolla; niitä ei tulisi kutsua toistorakenteiden, ehtojen tai sisäkkäisten toimintojen sisällä. Noudattamalla tätä sääntöä voidaan varmistua, että koukkuja kutsutaan aina samassa järjestyksessä jokaisella renderoinnilla. Koukkuja tulisi kutsua vain React-funktiosta, ei tavallisesta JavaScript-funktiosta. Koukkuja voidaan kutsua React-komponentista tai omista kustomoiduista koukuista. (React 2021c, viitattu 30.9.2021.)

2.2 JSX

JSX eli JavaScript XML on syntaksilaajennus JavaScriptiin, jolla voidaan määrittää React-elementtejä käyttäen tagipohjaista syntaksia suoraan JavaScript-koodissa. JSX muistuttaa paljon HTML-kieltä ja sen on toivottu helpottavan React-komponenttien luettavuutta. React-kirjastoa voidaan käyttää ilman JSX-syntaksia, mutta sen käyttö helpottaa komponenttien rakentamista. Sillä saadaan vähennettyä kirjoitetun koodin määrää, sen syntaksi on yksinkertainen sekä ytimekäs ja rakenteilla olevat komponentit ovat helppo visualisoida. (Porcello & Banks 2020, luku 5; A M & Sonpatki 2016, luku 2.)

JSX-syntaksissa elementin tyyppi määritetään tagilla ja tagin attribuutit edustavat ominaisuuksia. Jos elementillä on esimerkiksi lapsielementtejä, ne sijoitetaan elementin aloitus- ja lopetustagin väliin. JSX-syntaksia voidaan käyttää myös komponenttien kanssa. Esimerkiksi luokkakomponentti määritetään käyttämällä komponentin luokkanimeä ja sille voidaan välittää ominaisuuksia. Ominaisuudessa täytyy käyttää aaltosulkeita, kun komponentille välitetään JavaScript-arvoja ominaisuutena. Komponenteille voidaan välittää ominaisuuksia joko merkkijonona tai JavaScript-lausekkeena. Lauseke voi pitää sisällään taulukoita, olioita, tai jopa funktioita. Selaimet eivät tue JSX-

syntaksia, joten se täytyy ensin kääntää selaimen tulkitsemaan muotoon, jota varten on erillisiä työkaluja, kuten esimerkiksi Babel. (Porcello & Banks 2020, luku 5.)

Esimerkkikoodissa on toteutettu samanlainen elementti kahdella tavalla (kuvio 4). Ensimmäinen on toteutettu käyttäen JSX-syntaksia ja jälkimmäinen ilman. Näinkin yksinkertaisesta esimerkistä nähdään ero kirjoitetun koodin määrässä, vaikka renderoinnin tulos on identtinen.

KUVIO 4. Samannäköinen elementti toteutettu sekä JSX-syntaksilla että ilman

2.3 Frontend kirjastot ja pakkaukset

Material-UI

Material-UI on UI-kirjasto, joka tarjoaa laajan valikoiman valmiita komponentteja, jotka ovat helppokäyttöisiä ja muokattavissa (MUI 2021, viitattu 30.9.2021). Tämä helpottaa ja nopeuttaa React-sovelluksen kehitystä, koska kehittäjän ei tarvitse ohjelmoida kaikkea aivan alusta. MUI sai alkunsa vuonna 2014 ja siitä on kasvanut yksi suosituimmista React UI-kirjastoista.

Material-UI-kirjastossa on selkeät dokumentaatiot muun muassa:

- kirjaston käyttöönotosta

- yleistä tietoa komponenteista esimerkkikoodeineen sekä niiden rajapinnoista, joista selviää komponenttien ominaisuuksia ja tyylittelyjä
- ohjeita tyylittelyyn sekä teeman muokkaukseen

React Router

React Router on kokoelma navigointikomponentteja, joilla voidaan toteuttaa web- ja natiivisovelluksen sisällä tapahtuva navigointi. Siinä on kolme ensisijaista komponenttikategoriaa:

- Router-komponentit, <BrowserRouter> ja <HashRouter>
- reitin sovittaja -komponentit, <Route> ja <Switch>
- reitin muuttaja -komponentit, <Link> <NavLink> <Redirect>

Router-komponenttien tehtävä on varastoida URL-osoite sekä kommunikoida web-serverin kanssa ja sen tulee olla ylimmällä tasolla kaikkia navigointikomponentteja. Kun Switch-komponentti renderoidaan, se etsii sille lapsikomponenteiksi määritetyistä Route-komponenteista sen, jonka URL-osoite vastaa sen hetkistä Router-komponenttiin varastoitua URL-osoitetta. Kun vastaava Route-komponentti on löytynyt, sen sisältö renderoidaan. Link-komponentti luo anchor-elementin sovellukseen ja NavLink-komponentti on muuten samanlainen, mutta se voi tyylitellä itsensä aktiiviseksi, jos sille välitetty to-ominaisuus vastaa nykyistä URL-osoitetta. Jos sovelluksessa on tarve pakottaa navigoimaan tiettyyn osaan sovellusta, voidaan käyttää Redirect-komponenttia ja antaa sille to-ominaisuudeksi haluttu URL-osoite. (React Router 2021, viitattu 6.10.2021.)

MobX

MobX on kirjasto, jota käytetään sovelluksen sisäisten tilojen hallintaan. MobX erottaa sovelluksessa kolme eri käsitettä:

- state eli tila
- actions eli toiminnot
- derivations eli johtautumiset

Tila on dataa, johon sovelluksen toiminta perustuu. Tilaan voidaan tallentaa minkälaista tietorakennetta vain, esimerkiksi olioita, taulukoita ja luokkia. Näiden ominaisuudet, jotka voivat ajan myötä muuttua, pitää merkata observable-muuttujaksi eli tarkkailtavaksi, jotta MobX voi seurata niiden arvoja. Toiminnot ovat osa koodia, jotka muuttavat tilaa. Tällaisia voivat olla esimerkiksi käyttäjän tekemät toiminnot, palvelimelta tulevan datan käsittely ja ajoitetut tehtävät. On suositeltavaa, että jokainen koodin osa, joka muuttaa tilaa, merkataan toiminnoksi. Tällä tavalla MobX voi toimia automaattisesti mahdollisimman optimaalisen suorityskyvyn saavuttamiseksi. Johtautumiset voivat olla mitä vain, joka on jollain tavalla johdettu tilasta. (MobX 2021, viitattu 6.10.2021.)

3 BACKEND

3.1 Ruby

Ruby on ohjelmointikieli, jonka Yukihiro Matsumoto kehitti ja sittemmin julkaisi vuonna 1995. Matsumoto otti siihen vaikutteita suosikkikielistään ja tasapainotti funktionaalista ohjelmointia imperatiivisella ohjelmoinnilla.

Ruby-ohjelmointikielessä kaikki ovat olioita ja kaikelle pienimmällekin tiedolle ja koodille voidaan antaa ominaisuuksia ja toimintoja. Olio-ohjelmointi kutsuu ominaisuuksia instanssimuuttujiksi ja toimintoja metodeiksi. Monissa ohjelmointikielissä numerot ja muut primitiiviset tyypit eivät ole olioita, mutta Ruby seuraa Smalltalk-ohjelmointikielen vaikutteita ja antaa metodit ja instanssimuuttujat kaikille sen tyypeille. (Ruby 2021, viitattu 13.10.2021.)

3.2 Ruby on Rails

Rails on avoimen lähdekoodin web-sovelluskehys, joka pohjautuu Ruby-ohjelmointikieleen. Se on suunniteltu tekemään web-sovellusten ohjelmoinnista helpompaa ja vähällä määrällä koodia saadaan suoritettua paljon toiminnallisuutta. Rails noudattaa MVC-arkkitehtuurimallin tapaa ja jakaa sovelluksen toiminnot malleihin, näkymiin ja käsittelijöihin. Rails-sovelluskehyksen routes eli reitit ovat sääntöjä, jotka on kirjoitettu Ruby-täsmäkielellä. Tietty reitti kartoittaa kyselyn tietyn käsittelijän toimintaan, käsittelijä valmistelee datan näkymälle ja lopuksi näkymä esittää datan halutulla tavalla. (Ruby on Rails 2021b, viitattu 9.10.2021.)

Malli on Ruby-luokka, jota käytetään datan esittämiseen. Lisäksi käyttämällä Rails-ominaisuutta nimeltä Active Records, mallit voivat olla vuorovaikutuksessa sovelluksen tietokannan kanssa. Active record on järjestelmän kerros, joka vastaa business-datan ja -logiikan esittämisestä. Mallin nimeämiskäytäntö on yksikkö, koska luotu malli edustaa yhtä tietuetta. (Ruby on Rails 2021b, viitattu 9.10.2021.)

Kun mallit generoidaan Rails-sovelluskehyksen generate model -komentoa käyttäen, Rails luo muutaman tiedoston, joita ovat:

1. migrate-tiedosto; kuvaa tietokantaan tehtävät muutokset ja päivittää tietokannan skeeman
2. malli-tiedosto; Ruby-luokka, jonka avulla voidaan suorittaa CRUD-operaatioita
3. kaksi testaukseen liittyvää tiedostoa

Migrate-tiedostoa voidaan muokata luonnin jälkeen ja esimerkiksi lisätä generoinnin yhteydessä unohtuneet tiedot tai poistaa tarpeettomat. Tietokantamuutokset voidaan toteuttaa rails db:migrate -komennolla, joka tekee kaikki migrate-tiedostoissa, joita ei ole aikaisemmin suoritettu, määritetyt muutokset.

Näkymät (template) ovat myös tietynlaisia malleja, jotka ovat useimmiten kirjoitettu HTML- sekä Ruby-ohjelmointikielillä. Opinnäytetyön Rails-sovellus toimii rajapintana, joten näkymät eivät palauta sivurakennetta vaan yhdessä Jbuilder-kirjaston kanssa määrittävät kyselylle palautettavan vastauksen JSON-muotoon.

Käsittelijä on Ruby-luokka, joka perii ApplicationController-luokan ominaisuudet. Kun sovellus saa kyselyn, reititys määrittelee mikä käsittelijä sekä toiminto suoritetaan. Tämän jälkeen käsittelijästä tehdään ilmentymä ja siitä suoritetaan metodi, joka vastaa nimeltään toimintoa. Käsittelijälle voidaan välittää parametreja kahdella tavalla. Niin sanotut kyselymerkkijono-parametrit ovat osa URL-osoitetta ja ne tulevat osoitteen kysymysmerkin jälkeen. Toinen tapa on lähettää parametrit osana kyselyn runkoa niin sanottuina POST data -parametreina. Molemmat parametrit ovat käytettävissä Rails-sovelluskehyksen params-hajautuksessa. (Ruby on Rails 2021a, viitattu 6.10.2021.)

3.3 PostgreSQL

PostgreSQL on avoimen lähdekoodin olio-relaatiotietokantajärjestelmä, jota on kehitetty aktiivisesti yli 30 vuotta. Se käyttää ja laajentaa SQL-kieltä yhdistettynä moniin ominaisuuksiin, jotka tallentavat ja skaalaavat kaikkein monimutkaisimmat tietokuormat.

PostgreSQL tukee monenlaisia tietotyyppejä:

- Primitiivit: kokonaisluku, numeerinen, merkkijono, totuusarvo
- Strukturoitu: päivämäärä / aika, taulukko, arvoväli, UUID
- Asiakirja: JSON/JSONB, XML, avain-arvo -pari (Hstore)
- Geometria: piste, suora, ympyrä, monikulmio

- Mukautetut: komposiitti, mukautetut tyypit

(PostgreSQL 2021, viitattu 9.10.2021.)

PostgreSQL valikoitui opinnäytetyön tietokannaksi, koska sen käyttö on helppoa ja se oli jo ennestään tuttu töiden sekä erilaisten harrasteprojektien kautta.

3.4 JSON Web Token

JWT on avoin standardi (RFC 7519), joka määrittelee kompaktin ja itsenäisen tavan siirtää tietoja JSON-objektina osapuolten välillä turvallisesti. Tieto voidaan vahvistaa ja siihen voidaan luottaa, koska se on digitaalisesti allekirjoitettu. JWT voidaan allekirjoittaa käyttämällä tietynlaista salausavainta tai julkista/yksityistä avainparia käyttäen ja ne voidaan myös tarvittaessa salata enkryptaamalla. (JWT 2021, viitattu 12.10.2021.)

Skenaarioita, joissa JWT on hyödyllinen:

- Valtuutus: Yleisin skenaario JWT-standardin käyttöön. Onnistuneen kirjautumisen jälkeen jokainen seuraava kysely sisältää annetun tunnuksen, jolla käyttäjä voi käyttää reittejä, palveluita ja resursseja
- Tietojenvaihto: JWT on hyvä tapa välittää tietoja turvallisesti osapuolten välillä. Lähettäjästä voidaan varmistua, koska JWT voidaan allekirjoittaa

Opinnäytetyössä päätetiin käyttää JWT-standardia kyselyiden valtuutuksissa, koska jatkokehityksen kannalta sen käyttö ja prosessointi asiakasohjelman puolella on helppoa alustasta riippumatta toisin kuin esimerkiksi evästepohjainen valtuutus.

3.5 Backend-kirjastot

Devise

Devise on Rails Gem eli kirjasto, joka perustuu Warden-väliohjelmistoon ja tarjoaa joustavan todennusratkaisun Rails-sovellukseen.

Se koostuu kymmenestä moduulista:

- Database authenticatable: tekee salasanasta tiivisteen ja tallentaa sen tietokantaan, jolla todennetaan kirjautuva käyttäjä
- Omniauthable: lisää OmniAuth-tuen
- Confirmable: lähettää vahvistusohjeet sähköpostiin ja tarkistaa onko tili jo vahvistettu
- Recoverable: palauttaa käyttäjän salasanan ja lähettää nollausohjeet
- Registerable: hoitaa käyttäjien rekisteröinnit rekisteröintiprosessin kautta ja mahdollistaa heidän tilinsä muokkauksen ja poistamisen
- Rememberable: luo ja tyhjentää token-tunnisteet, joilla muistetaan käyttäjät tallennetuista evästeistä
- Trackable: seuraa kirjautumisten määrää, aikaleimoja ja IP-osoitteita
- Timeoutable: vanhentaa istuntoja, jotka eivät ole olleet aktiivisia tiettyyn aikajaksoon
- Validatable: tarjoaa sähköpostin ja salasanan tarkistukset. Valinnainen ja muokattavissa, joten voidaan määrittää omat tarkistukset
- Lockable: lukitsee tunnuksen, kun tietty määrä epäonnistuneita kirjautumisyrityksiä täyttyy. Tunnus voidaan avata sähköpostin kautta tai tietyn aikajakson jälkeen

Devise luo joitakin apumetodeja, joita voi käyttää hyödyksi Rails-sovelluskehyksen käsittelijöissä ja näkymissä. Näitä ovat muun muassa authenticate_user!-, user_signed_in?- ja current_user-metodit. (Devise 2021, viitattu 11.10.2021.)

Jbuilder

Jbuilder tarjoaa yksinkertaisen täsmäkielen JSON-rakenteiden toteuttamiseen. Se on hyödyllinen erityisesti silloin kun luontiprosessi pitää sisällään paljon ehtolauseita ja toistorakenteita. (Jbuilder 2021, viittattu 11.10.2021.) Jbuilder-kirjastolla voidaan toteuttaa rajapintakyselyiden vastaukset ja ne ovat kuin omia näkymiänsä, joita kirjasto hakee oletuksena sovelluksen views-kansion alta. Jbuilder-kirjasto pystyy hyödyntämään useita näkymämallin ominaisuuksia esimerkiksi current_user-apumetodia, josta on valtavasti hyötyä esimerkiksi tietojen hakemisessa ja oikeuksien tarkistamisessa.

4 SUUNNITTELU

Suunnitteluun ei käytetty kovin pitkää aikaa, vaikka syytä olisi ollut. Kehittämistehtävä oli tietoisesti laaja ja sitä haluttiin päästä nopeasti toteuttamaan vision pohjalta. Suunnittelussa mietittiin tarvittavia toiminnallisuuksia ja sen pohjalta tietokantarakennetta sekä käyttöliittymää. Ideoita haettiin myös samankaltaisista palveluista, esimerkiksi HacknPlan ja Trello.

Toiminnallisuus

Toiminnallisuuden suunnittelussa lähdettiin miettimään kuvitteellisen käyttäjän tapaa navigoida ja käyttää järjestelmää. Frontend-sovelluksen täytyi olla mahdollisimman helppokäyttöinen ja selkeä, jotta käyttäjä ymmärtää heti sen toiminnallisuuden ja miten sitä käytetään.

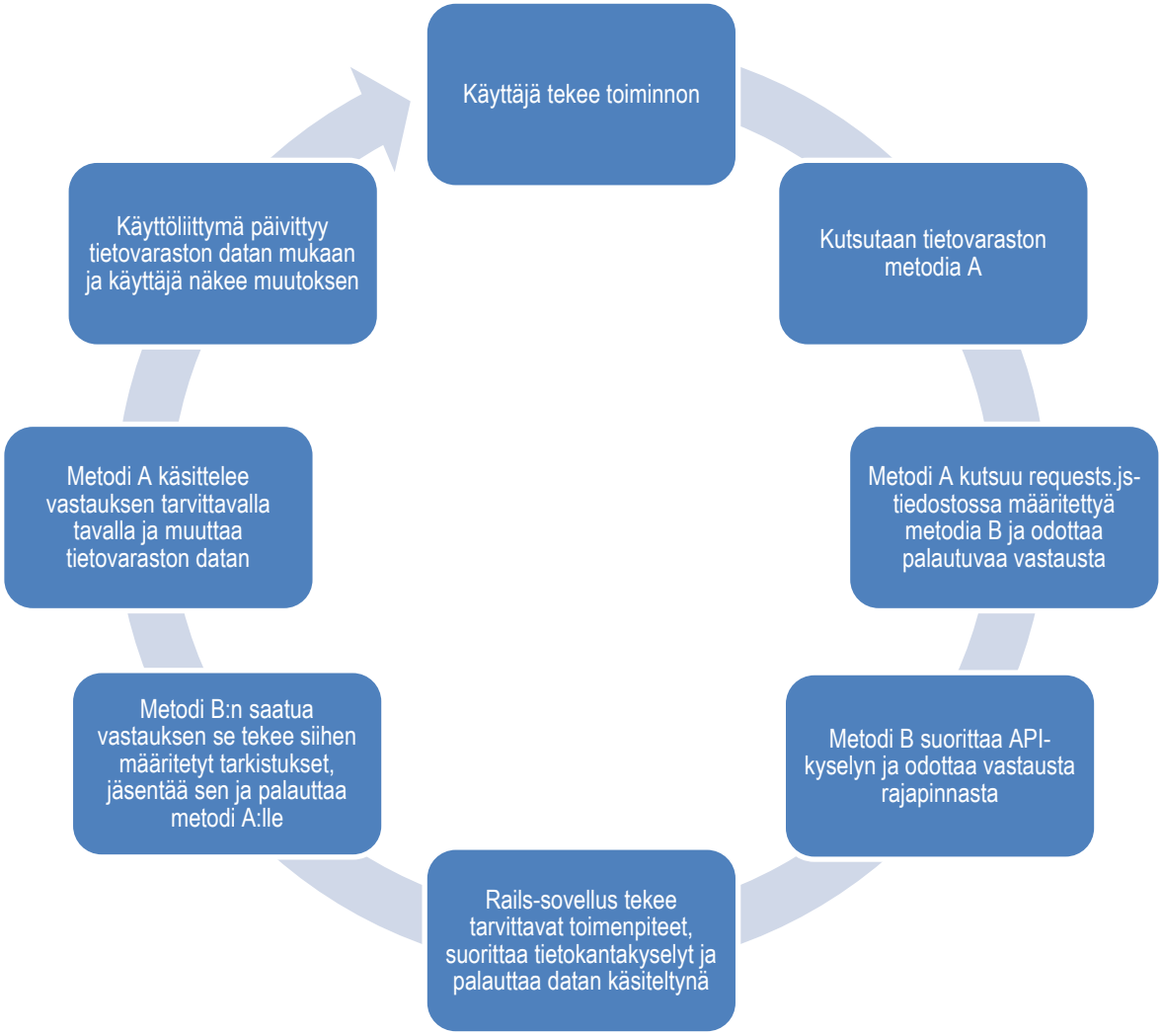
Sovelluksessa pitää olla kirjautuminen, jolla jokainen käyttäjä identifioidaan ja pystytään tuomaan käyttäjään liittyvä data tietokannasta. Käyttäjä A:n onnistuneen kirjautumisen yhteydessä noudetaan tietokannasta mahdollisimman paljon häneen liittyvää dataa ja tallennetaan se myöhempää käyttöä varten. A:n tehdessä muutoksia dataan, tiedot päivitetään tietokantaan ja sovelluksen sisäisiin tietovarastoihin. Reaaliaikaisuutta tarvitaan, jos käyttäjä B tekee muutoksia dataan, joka liittyy myös jollain tavalla käyttäjään A ja hänen pitäisi saada tieto muutoksista. Tämä toiminnallisuus voidaan toteuttaa käyttämällä Action Cable -kirjastoa, jonka avulla muutostiedot voidaan välittää reaaliaikaisesti kaikille käyttäjille, joille se on tarpeellista. Muutostiedon saavuttua frontend-sovellukseen sen sisältö päivitetään datan mukaan.

Perustoimintoina käyttäjän pitää pystyä luomaan, muokkaamaan ja poistamaan:

- Projekteja
- Sprinttejä
- Tehtäväkategorioita
- Työtehtäviä

Lisäksi käyttäjän pitää pystyä kutsumaan toisia käyttäjiä projektiin mukaan ja määrittämään käyttäjän projektikohtainen rooli. Työtehtäville voidaan merkata käyttäjiä, jotka vastaavat kyseisen tehtävän toteutuksesta. Käyttäjän tulee voida muuttaa tehtävällä olevaa tilaa vetämällä ja pudottamalla

komponentti haluamaansa tilaa kuvaavan otsikon alle. Kuviossa 5 on havainnollistettu yksinkertaistettuna prosessin kulku käyttäjän suorittamasta toiminnosta käyttöliittymän päivitykseen.



KUVIO 5. Prosessin kulku käyttäjän suorittamasta toiminnosta käyttöliittymän päivitykseen

Tietokanta

Tietokantaa ei syvällisemmin suunniteltu alussa, vaan siitä hahmoteltiin hyvin nopea kaavio perustoimintojen pohjalta. Tiedossa oli, että rakenne tulee muuttumaan ja kasvamaan kehityksen aikana, joten tauluja ja niissä olevia sarakkeita luotiin ja lisättiin sen mukaan mitä toiminnallisuuksien toteutukset vaativat.

Dynaamiset projektikohtaiset tehtäväkategoriat olivat yksi haaste tietokannan suunnittelussa. Nopeimpana ja helpoimpana ratkaisuna katsottiin, että jokaiselle projektille luodaan oletuskategoriat, joita käyttäjä voi halutessaan lisätä, muokata tai poistaa. Toinen haaste oli tehtäviin merkattavat käyttäjät, jotka vastaavat tehtävän toteutuksesta. Ensimmäinen ratkaisuvaihtoehto oli välitaulu, joka kyllä olisi ollut toimiva ja joissain tapauksissa parempikin ratkaisu, mutta lopulta päädyttiin käyttämään PosgreSQL-tietokantajärjestelmän mahdollistamaa taulukkotyyppistä saraketta, johon tallennetaan käyttäjien id-numerot, joiden mukaan käyttäjätiedot haetaan tietokannasta.

5 TOTEUTUS

Järjestelmä toteutettiin paikallisena versiona ja se aloitettiin tarvittavien työkalujen läpikäynnillä. Koska aikaisemmin oli jo luotu React- ja Ruby on Rails -projekteja, tarvittavat esivaatimukset niiden suhteen oli jo täytetty ja pystyttiin suoraan aloittamaan projektien luonnista. Ensimmäiseksi luotiin Rails-projekti, joka toimii ohjelmointirajapintana ja projektin alikansioksi luotiin client-kansio, johon luotiin React-projekti. Tämä tiedostorakenne helpotti sovellusten kehittämistä ja niiden testaamista. Projektien luonnin jälkeen molempien sovellusten kehitysympäristöt käynnistettiin ja varmistettiin niiden toimivuus. React-sovellus käynnistettiin oletusporttiin, joka on 3000 ja Rails-sovellus määritetään käynnistymään porttiin 3001. Näitä portteja käytettiin myös myöhemmissä kehitysvaiheissa. Heti alussa asennettiin mahdollisimman paljon tarvittavia kirjastoja ja pakkauksien sekä frontendin että backendin puolelle.

Alla on yhteenvetoa toiminnallisuuksista, joita järjestelmään toteutettiin kokonaan tai osittain sekä ideoita, jotka jäivät lopulta jatkokehitykseen. Listauksista voi puuttua, joitakin toiminnallisuuksia, jotka kehitettiin tai ideoita toteutettavista toiminnallisuuksista, koska niitä oli paljon eikä kehityksen aikana käytetty projektinhallintajärjestelmää.

Toteutetut toiminnallisuudet:

- Projektin lisääminen
- Projektikutsun hyväksyminen ja hylkääminen
- Aktiivisen projektin vaihto
- Monikielisyys ja kielen vaihto
- Sprintin luonti, muokkaus ja poisto
- Aktiivisen sprintin muuttaminen
- Sprintin sulkeminen ja avaaminen
- Tehtävän luonti, muokkaus ja poisto
- Käyttäjien lisääminen tehtävään sekä niiden poisto
- Tehtävän siirto sprintteihin ja kehitysjonoon
- Rekisteröityneen käyttäjän lisääminen projektin jäseneksi
- Avoimen projektikutsun poisto projektista

Osittain toteutetut:

- Projektin jäsenen roolin muuttaminen
- Reaaliaikaisuus
- Virheenkäsittely ja käännökset
- Sprintin määräajan esitys

Jatkokehitykseen jääneet toiminnallisuudet:

- Palveluun rekisteröityminen
- Projektin muokkaus ja poisto
- Projektikohtainen keskustelualue
- Tehtäväkategorian lisääminen, muokkaus ja poisto
- Tehtävän valmiustilojen kustomointi ja dynaamisuuden varmistus
- Tehtävien järjestely sprintissä
- Roolin ja oikeuksien tarkistuksen eri toiminnoissa
- Rekisteröitymättömän käyttäjän kutsuminen projektin jäseneksi ja sähköpostikutsun lähetys
- Projektista lähteminen/poistuminen
- Projektin jäsenen poisto projektista

5.1 React-sovellus

Kehitys alkoi frontendistä ja vaikka web-sovellus suunniteltiin käytettäväksi isolla näytöllä, pyrittiin pitämään mielessä myös responsiivisuus. Tämän toteutukseen käytettiin Material-UI-kirjaston tarjoamaa ruudukkojärjestelmää, joka mukautuu näytön koon ja suunnan mukaan.

Ensimmäisenä mietittiin pääkomponentit, jotka ovat pääsääntöisesti aina näkyvillä sivusta riippumatta. Tällaisia komponentteja olivat sovelluspalkki, johon dynaamisesti vaihtui sivun nimi ja jonka oikeasta laidasta käyttäjä pystyi valitsemaan sovelluksen kielen, sekä sovelluksen vasempaan laitaan sijoittuva vetolaatikko, joka piti sisällään sekä dynaamista että staattista sisältöä. Näiden lisäksi luotiin div-elementti, johon React Routerin toistorakenne vaihtaa sivun näkyvän sisällön käyttäjän navigoidessa sovelluksen sisällä.

Alussa React-sovellusta tehtiin melko pitkälle ennen kuin Rails-sovellus astui mukaan ja frontendin sisältö kovakoodattiin keksityllä datalla ja tyylit määriteltiin käyttäen Material-UI -kirjaston Hook-rajapintaa. Myöhemmin data korvattiin tietokannasta tulevalla oikealla datalla, joka aiheutti jonkin verran koodin refaktorointia, koska sovellus ei toiminutkaan oikean datan kanssa niin kuin oli suunniteltu. Samoihin aikoihin päätettiin ottaa n18next-kirjasto käyttöön ja toteuttaa sovellukseen käännökset suomeksi ja englanniksi.

5.1.1 Tietovarastot

Frontend sovelluksessa tarvittavat tietovarastot luotiin heti varhaisessa vaiheessa. Määritettiin kolme erillistä luokkatiedostoa, jotka toimivat varastoina. Niissä käytettiin MobX-tilanhallintaa varastoimaan tarvittavaa dataa sekä määritettiin metodeja, joilla tilan eli tallennetun datan muokkaaminen onnistui. Nämä varastot olivat:

1. UiStore, johon tallennettiin käyttöliittymään, kirjautumiseen ja kirjautuneeseen käyttäjään liittyvää tietoa
2. ProjectStore, johon tallennettiin kaikki projekteihin liittyvä tieto
3. RootStore, jonka tehtävänä oli koota kaikki tietovarastot yhdeksi varastoksi

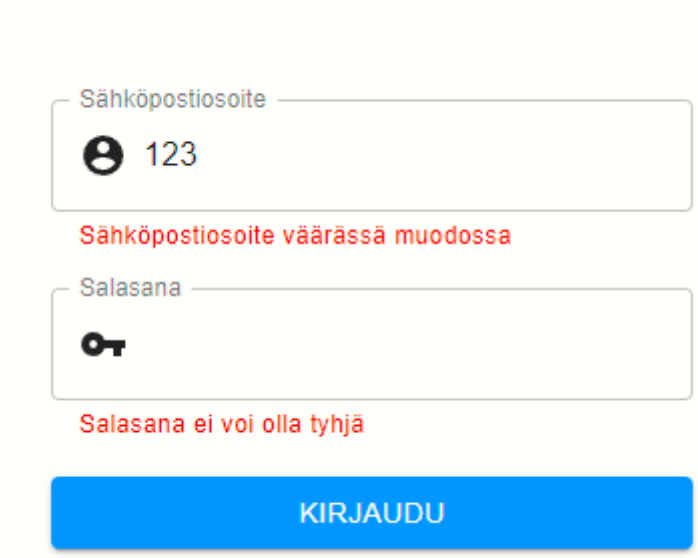
RootStore-tietovarastoon tuotiin myös Request.js-tiedosto, joka pitää sisällään kaikki sovelluksen sisällä tarvittavat API-kyselyt ja sen lopussa määritettiin React Hook, jota tuomalla muihin sovelluksen osiin päästiin käsiksi varastojen tiloihin ja metodeihin.

5.1.2 Kirjautumisnäkymä

Kirjautumisnäkymä oli ensimmäinen näkymä, jonka kirjautumaton käyttäjä näki sovellusta käytettäessä. Näkymä oli itsenäinen Login-komponentti, jonka juuressa oli niin sanottu wrapper-elementti. Tämä elementti oli tyylitelty sijoittumaan näytön keskelle ja se piti sisällään kaikki näkymän muut komponentit. Käyttäjä näki kaksi input-elementtiä, joista ylempään käyttäjän oli tarkoitus syöttää sähköpostiosoite, joka toimi käyttäjätunnuksena ja alempaan salasanansa. Kumpaankin input-elementtiin oli yhdistetty myös niin kutsuttu aputeksti, joka ilmoitti mahdollisesta syötevirheestä. Näiden elementtien alle oli sijoitettu painike, josta käyttäjä pystyi suorittamaan kirjautumisyrityksen.

Virheellisestä kirjautumisyrityksestä aiheutuva virhe näytettiin käyttäjälle sähköpostikentän yläpuolella. Ensimmäisessä versiossa ei mahdollistettu rekisteröitymistä vaan testitunnukset oli luotu manuaalisesti.

Kirjautumiseen oli yhdistetty kahdenlaista syötteen tarkistusta. Käyttäjän poistaessa tekstikentästä aktiivisuuden tapahtui onBlur-tapahtuma, jolloin tarkistettiin tekstikentän syöte ja ilmoitettiin mahdollisesta virheestä. Myös itse kirjautumisyritykseen oli yhdistetty tarkistukset siltä varalta, että käyttäjä yritti suorittaa kirjautumista virheellisillä syötteillä (kuvio 6).

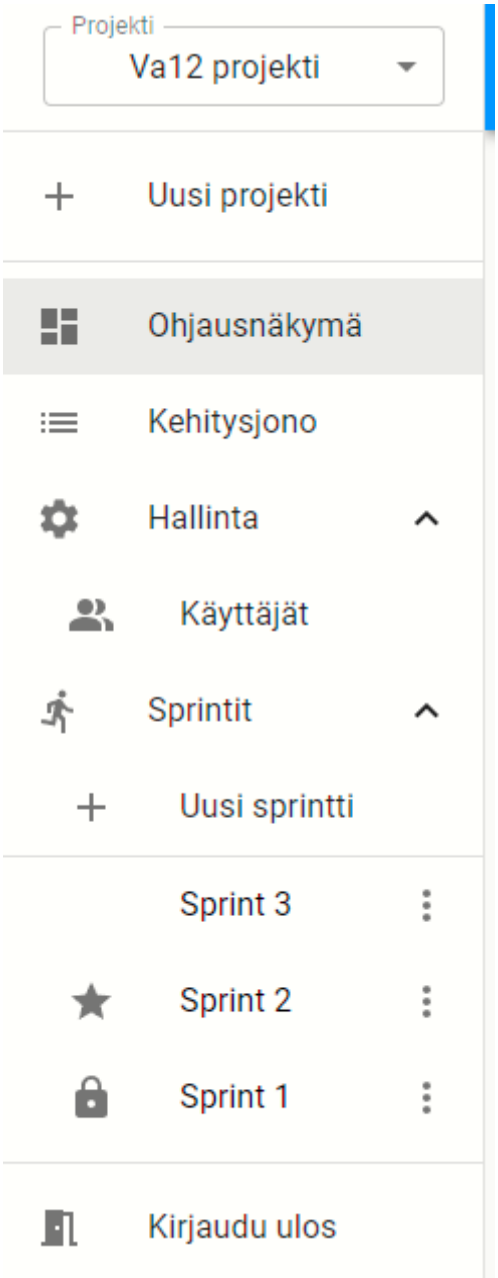


KUVIO 6. Käyttäjän kirjautumisyritys virheellisillä syötteillä

Onnistuneessa kirjautumisessa rajapinnasta palautuvan vastauksen otsikkotietoihin oli lisätty Authorization-tieto, joka sisälsi JWT-tunnisteen myöhempiä API-kyselyitä varten. Tämä tunniste tallennettiin selaimen paikalliseen varastoon, josta se luettiin ja lisättiin myöhempien kyselyiden otsikkotietoihin. Onnistuneen kirjautumisen yhteydessä haettiin myös käyttäjädata sekä projektidatat, joissa käyttäjä oli mukana. Käyttäjädata tallennettiin suoraan UiStore-tietovaraston currentUser-muuttujaan, mutta projektidata vaati pientä prosessointia ennen tallennusta. Projektidatasta etsittiin ensin projekti, joka oli merkattu aktiiviseksi ja tallennettiin se ProjectStore-tietovaraston currentProject-muuttujaan ja lopuksi koko projektidata tallennettiin tietovaraston userProjects-muuttujaan. Lopuksi UiStore-tietovaraston loggedIn-muuttujan totuusarvoksi muutettiin tosi, joka aiheutti käyttäjän ohjautumisen aktiivisen projektin ohjausnäkymään.

5.1.3 Responsiivinen vetolaatikko

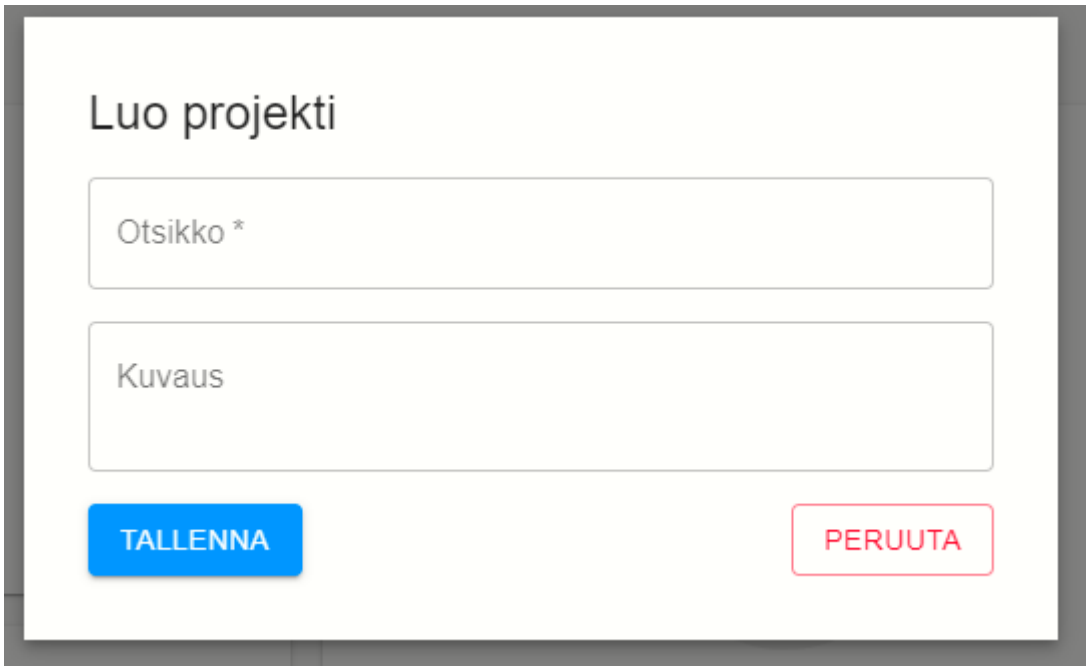
Kirjautumisen jälkeen sovelluksen rakenne muuttui olennaisesti ja yksi tärkeimmistä komponenteista oli vetolaatikko, joka oli sisään kirjautuneella käyttäjällä koko ajan käytettävissä. Vetolaatikko (kuvio 7) toteutettiin käyttämällä Material-UI-kirjaston tarjoamaa Drawer-komponenttia. Jokaisessa komponentissa, joka ohjasi käyttäjän eri näkymään, oli myös React Router -kirjaston komponentti.



KUVIO 7. Sovelluksen vetolaatikko-komponentti

Vetolaatikon yläosaan sijoittuva projekti-pudotusvalikko toteutettiin Material-UI-kirjaston Select-komponentilla, jossa oletusarvona oli projekti, joka on käyttäjälle merkattu aktiiviseksi projektiksi, jos sellainen löytyi. Projektin vaihtaminen pudotusvalikosta kutsui ProjectStore-tietovarastossa määritettyä changeCurrentProject-funktiota, joka suoritti API-kyselyn aktiivisen projektin muuttamiseksi sekä muutti currentProject-muuttujan arvon valituksi projektiksi. Projektin vaihtaminen ohjasi käyttäjän automaattisesti projektin ohjausnäkymään ja kaikki projektista näytettävä data muuttui valitun projektin mukaan.

Uusi projekti -komponenttia painettaessa avautui ProjectModal-komponentti, jota käytettiin sekä uuden projektin luontiin että projektin tietojen muokkaukseen. Modaalille välittyi tieto uuden projektin luonnista, joten se noudatti projektin luonnin logiikkaa ja esitti sisällön sen mukaan (kuvio 8KUVIO 8). Projektille ei ensimmäisessä versiossa ollut mahdollista antaa kuin otsikko, joka toimi projektin nimenä sekä mahdollinen kuvaus projektista. Näistä vain otsikko oli pakollinen kenttä ja sen syötteen tarkistus toimi samalla periaatteella kuin kirjautumisen syötteissä.

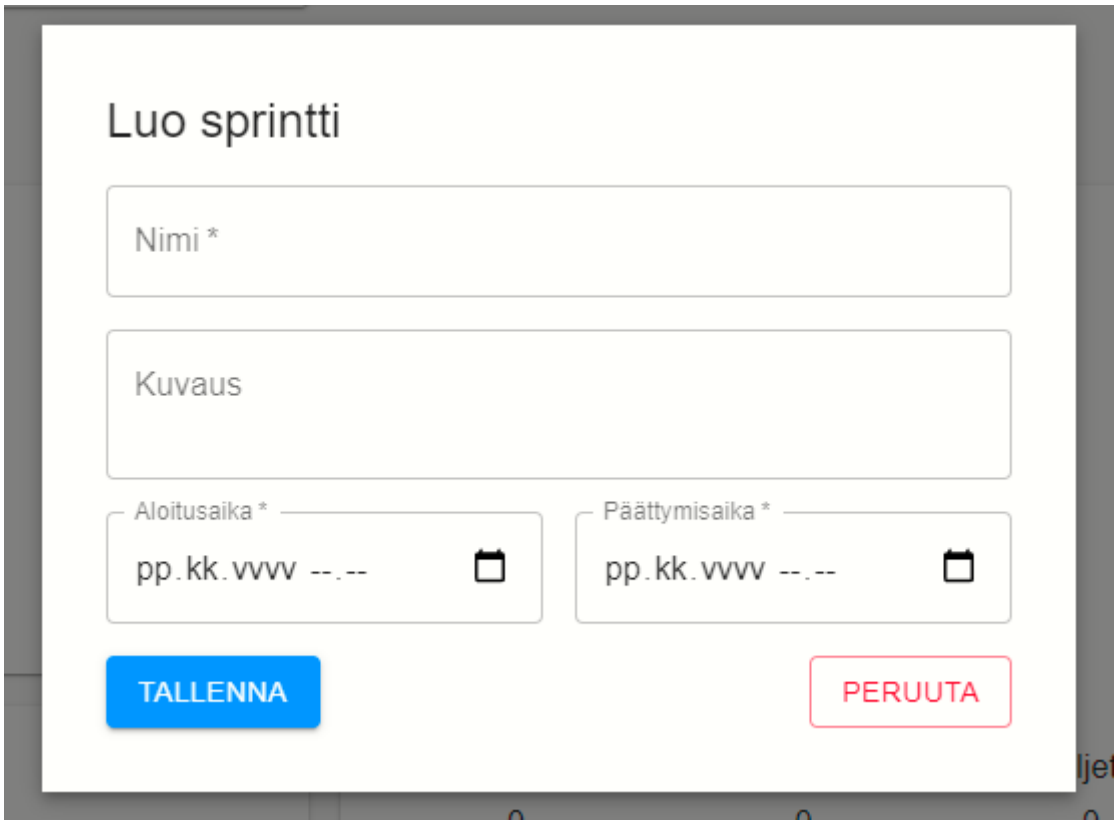


KUVIO 8. ProjectModal-komponentti avattuna projektin luontia varten

Ohjausnäkymä- ja kehitysjono-komponenttien toiminta olivat samanlaiset. Niiden lapsiksi oli määritetty React Router -kirjaston Link-komponentti, jolloin komponenttia painettaessa URL-osoite muuttui ja käyttäjä ohjautui eri näkymään.

Hallinta-komponenttia painettaessa sen lapseksi määritetty Collapse-komponentti avautui ja näkyviin tuli listaus projektin hallintaan liittyvistä komponenteista. Tässä vaiheessa sen alle tuotiin vain käyttäjät-komponentti, jota painamalla voitiin siirtyä projektin käyttäjähallinta-näkymään.

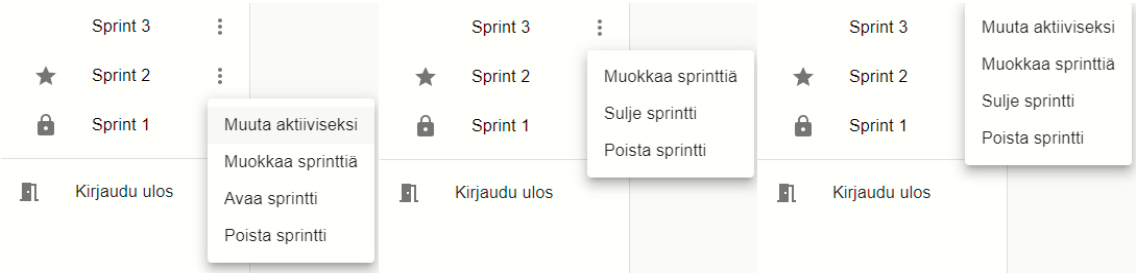
Sprintit-komponentissa käytettiin myös Collapse-komponenttia ja sen alta paljastui uuden sprintin luontiin tarkoitettu komponentti sekä jokaista projektin sprinttiä kohden renderoitiin oma komponentti. Uusi sprintti -komponenttia painettaessa avautui SprintModal-komponentti, jonka logiikka noudatti samaa periaatetta kuin ProjectModal-komponentti. Sprintin vaadittuja kenttiä olivat nimi sekä aloitusaika ja päättymisaika, jotka olivat datetime-tyyppisiä. Halutessaan sprintille pystyi antamaan myös kuvauksen (kuvio 9).



KUVIO 9. SprintModal-komponentti avattuna sprintin luontia varten

Sprinttilistaus oli toteutettu dynaamisesti, koska projektikohtaisten sprinttien lukumäärä vaihteli ja käyttäjän luodessa uusi sprintti, sen piti ilmestyä heti listauksen ylimmäksi. Jokaiselle sprinttilistauksen komponentille oli logiikka renderoituiko tekstin eteen ikoni. Jos sprintti oli aktiivinen, renderoitui tähti kuvaamaan aktiivista sprinttiä ja jos sprintti oli suljettu, renderoitui lukko. Lukko-ikoni toimi myös dominoivana ikonina ja korvasi tähti-ikonin, jos sprintti oli sekä aktiivinen että lukittu.

Sprintin tekstin perässä oleva ikoni kuvasti siihen liittyvän valikko ja ikonia painettaessa avautui DropdownMenu-komponentti, josta pyrittiin tekemään mahdollisimman uudelleenkäytettävä. Valikon vaihtoehdot vaihtuivat sille välitettyjen ominaisuuksien mukaan ja tässä tapauksessa komponentille välitettiin sprinttiin liittyvät vaihtoehdot ja nekin vaihtelivat dynaamisesti sprintin tilan mukaan (kuvio 10).



KUVIO 10. DropdownMenu-komponentti dynaamisilla vaihtoehdoilla

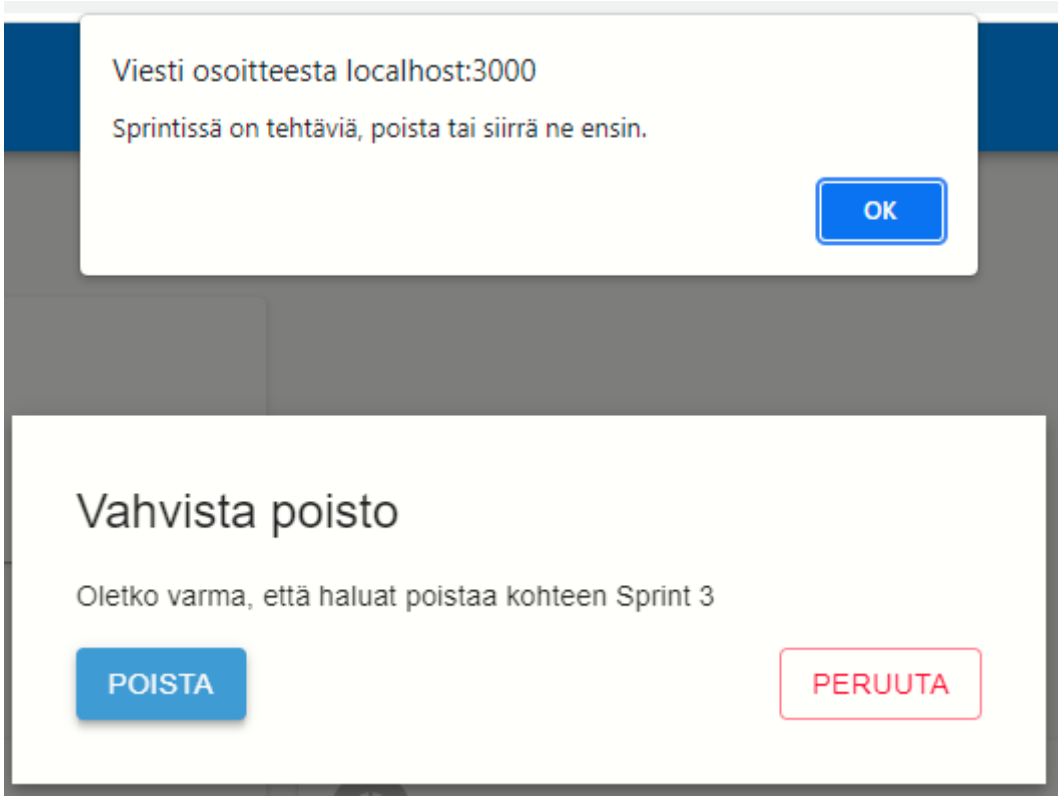
Muuta aktiiviseksi -toimintoa painettaessa suoritettiin API-kysely tietokantamuutoksia varten ja onnistuneen vastauksen jälkeen ProjectStore-tietovaraston currentProject-muuttujasta haettiin find()-metodilla sen hetkinen aktiivinen sprintti sekä muutoksen kohteena oleva sprintti ja niiden active-totuusarvot päivitettiin muutoksen mukaiseksi. Samalla userProjects-muuttujasta haettiin find()-metodilla projekti, joka vastasi aktiivista projektia ja sen sprintit päivitettiin ajan tasalle.

Muokkaa sprinttiä -komponentista avautui tuttu SprintModal-komponentti, jolle tällä kertaa välitettiin tieto muokkauksesta sekä sprintti-olio, josta saatiin modaaliin tarvittava data. Modaalin logiikka muilta osin noudatti pitkälti samaa kuin uuden sprintin kohdalla, mutta API-kyselyn päätepiste oli eri.

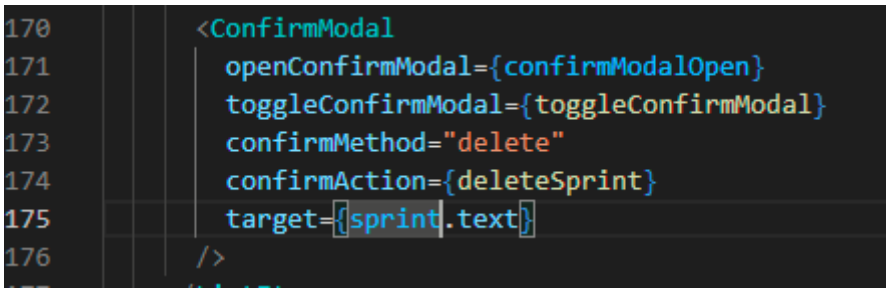
Komponentti, josta avattiin tai suljettiin sprintti, noudatti samaa logiikka kummassakin tapauksessa. Komponenttia painettaessa kutsuttiin ProjectStore-tietovaraston changeClosedStatus()-metodia, jolle välitettiin sprintin id argumenttina. Metodissa tarkistettiin onko sprintillä avoimia tehtäviä käyttämällä filter()-metodia ja jos sellaisia löytyi, näytettiin yksinkertainen alert-viesti avoimista tehtävistä eikä sprintille tehty toimenpiteitä. Jos taas sprintillä ei ollut avoimia tehtäviä, suoritettiin API-kysely tilan päivittämiseksi ja sprintin tila päivitettiin rajapinnasta palautuvan vastauksen mukaiseksi sekä currentProject- että userProjects-muuttujien alle.

Poista sprintti -komponentista avautui ConfirmModal-komponentti, jolla vahvistettiin kyseisen sprintin poisto. Jos yritettiin poistaa sprintti, jolle oli laitettu tehtäviä, näytettiin yksinkertainen alert-viesti eikä poistoa suoritettu (kuvio 11). Komponentille välitettiin seuraavat ominaisuudet (kuvio 12):

- openConfirmModal; totuusarvo, joka kertoi komponentin lapsikomponentiksi määritetylle modaalille, oliko se avoinna vai ei.
- toggleConfirmModal; metodi, jolla modaali avattiin ja suljettiin
- confirmMethod; string-tyyppinen arvo, jota käytettiin modaalin otsikon käännöksen hakuun sekä switch-toistorakenteessa selvittämään millainen käännös sisältötekstiin renderoitiin
- confirmAction; metodi, joka suoritettiin vahvistuksen jälkeen
- target; kohteen teksti esimerkiksi nimi tai otsikko, joka renderoitiin modaalin sisältötekstissä



KUVIO 11. ConfirmModal-komponentti avoinna ja yritetään poistaa sprintti, jolla on tehtäviä

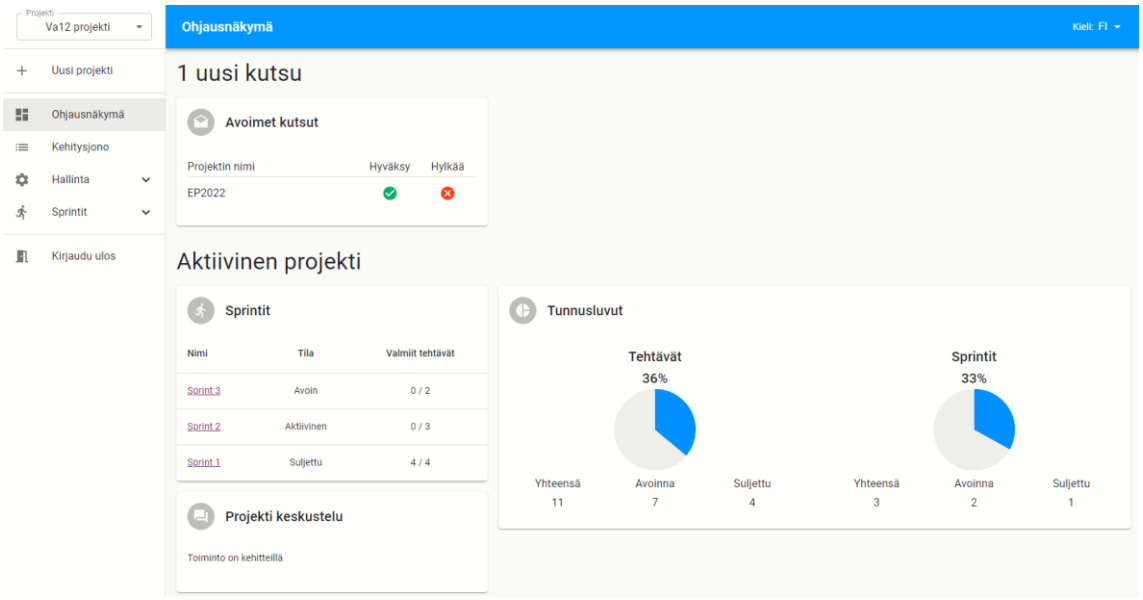


KUVIO 12. ConfirmModal-komponentille välitetyt ominaisuudet

Vetolaatikon viimeiseksi sijoitettua Kirjaudu ulos -komponenttia painettaessa, suoritettiin ulos kirjautumisen API-kysely. Samalla poistettiin selaimen paikallisesta varastosta JWT-tunniste sekä muutettiin sovelluksen kirjautumiseen liittyvä totuusarvo epätodeksi, joka aiheutti käyttäjän ohjautumisen kirjautumisnäkymään.

5.1.4 Ohjausnäkymä

Ohjausnäkymä jakautui alussa kolmeen pääosioon; sprintit, tunnusluvut ja projektin keskustelu, mutta myöhemmin myös kutsut eri projekteihin päätettiin tuoda tähän näkymään (kuvio 13). Kutsut-komponentilla oli ehdollinen renderointi eli jos käyttäjällä ei ollut yhtään avointa kutsua, komponentti ei renderoitunut. Jos taas kutsuja oli avoinna, se oli näkyvillä projektista riippumatta. Sijoitusta pohdittiin myös vetolaatikon hallinta-komponentin alle, jolloin uusista kutsuista olisi voinut indikoida paremmin, mutta se päätettiin jättää jatkokehitykseksi. Kutsujen otsikon luku muuttui dynaamisesti kutsujen lukumäärän mukaan sekä tekstiosa yksikön ja monikon välillä. Kutsuista voitiin nähdä kutsun kohteena olevan projektin nimi ja kutsu voitiin hyväksyä tai hylätä IconButton-komponenttia painamalla.



KUVIO 13. Ohjaus-näkymä, jossa uusi projektikutsu

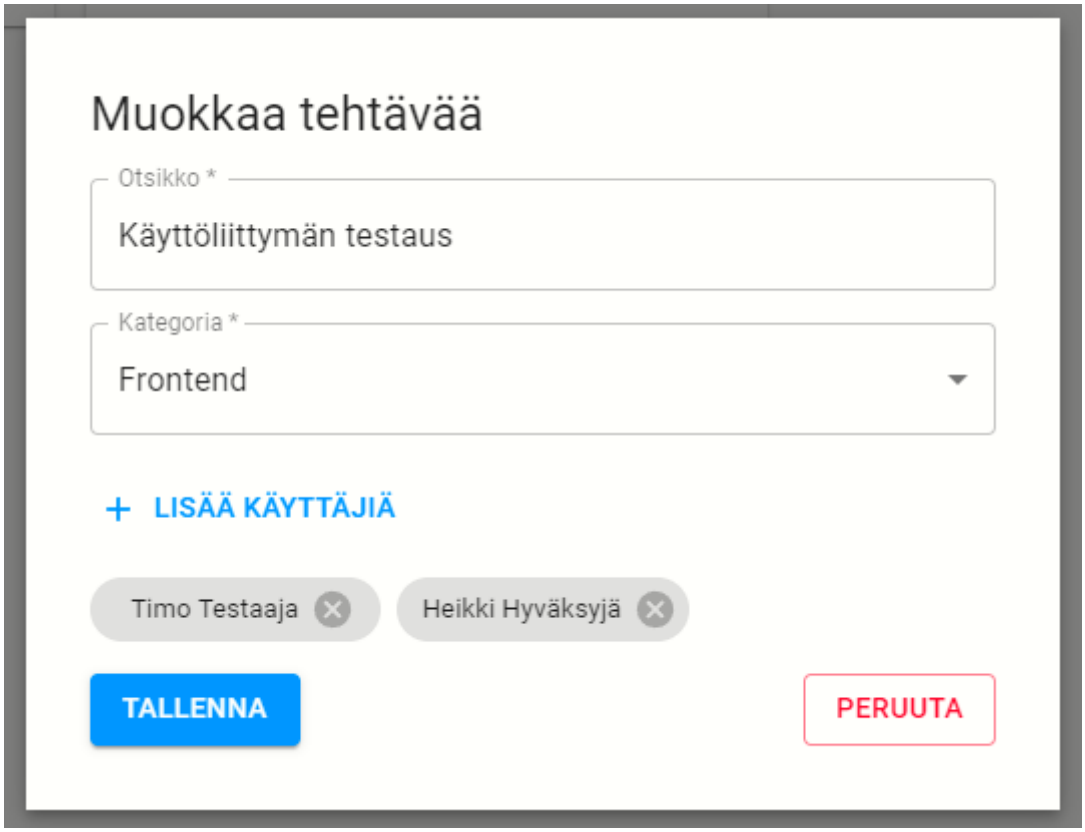
Sprintit-komponentti oli aluksi vain listaus projektiin kuuluvista sprinteistä linkkeineen, mutta loppuvaiheessa sen ei todettu tuovan lisäarvoa tai tiedollista hyötyä, joten se refaktoroitiin käyttämään Material-UI-kirjaston Table-komponenttia ja komponentille välitettiin enemmän dataa, josta voisi olla hyötyä käyttäjälle. Lopulta sprinteistä esitettiin niiden nimi linkin kanssa, joka ohjasi käyttäjän sprintin näkymään, sprintin tila sekä siihen kuuluvien tehtävien tila.

Tunnusluvut-komponentissa oli statistiikkaa projektin yleisestä tilanteesta jakautuen tehtäviin ja sprintteihin. Kumpaankin osioon kuului oma ympyrädiagrammi, joka kuvasi prosentuaalista valmiusastetta. Nämä diagrammit toteutettiin käyttämällä Material-UI-kirjaston CircularProgress-komponenttia, jota yleensä käytetään kuvaamaan esimerkiksi asynkronisen toiminnon edistymistä, mutta se soveltui myös tähän käyttötarkoitukseen. Osioiden alla näkyi myös lukumäärät, kuinka monta tehtävää tai sprinttiä projektiin oli luotu, kuinka monta niistä oli avoinna ja kuinka monta suljettu.

Projektin keskusteluun suunniteltiin reaaliaikaista projektikohtaista keskustelualuetta. Käyttäjälle oli tarkoitus näkyä esimerkiksi WhatsApp-sovelluksesta tuttu status-viesti, jos joku muu projektiin osallistuja kirjoittaa viestiä. Reaaliaikaisen toiminnallisuuden olisi hoitanut Action Cable -kirjasto ja ensimmäisessä versiossa olisi mahdollistettu vain String-tyyppiset viestit, mutta jatkokehityksessä olisi lisätty myös kuvien lähetys.

5.1.5 Kehitysjono-näkymä

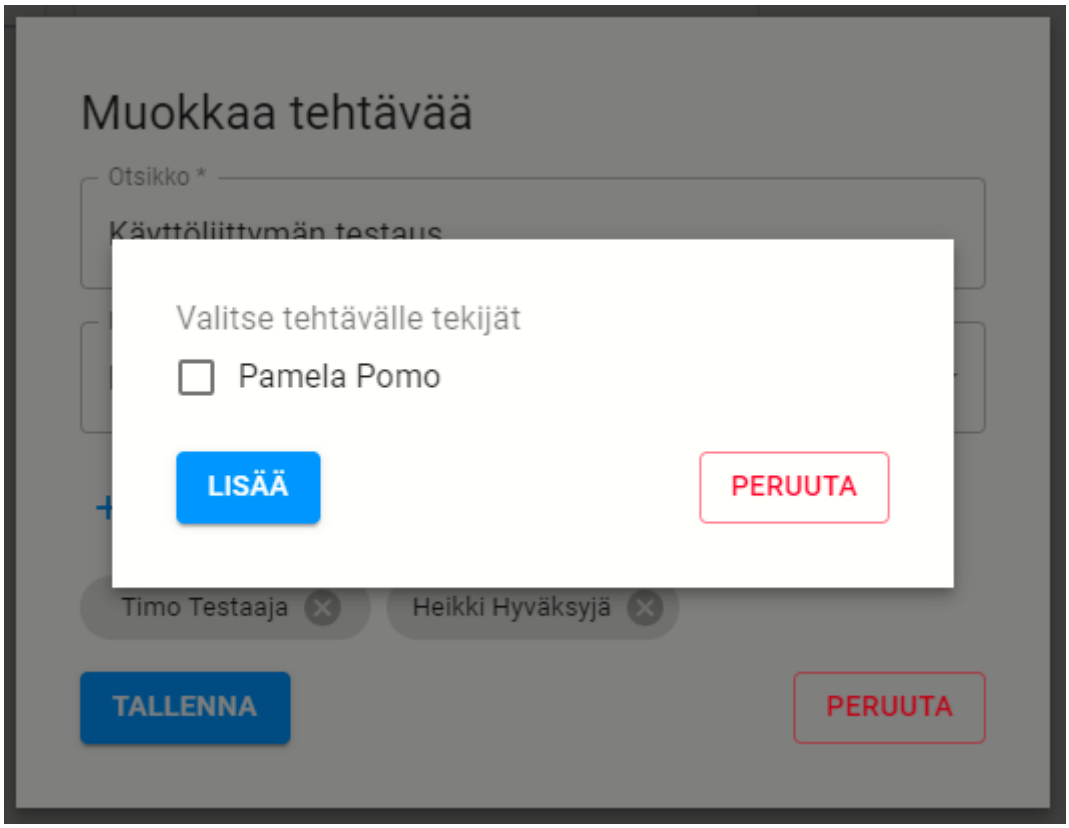
Kehitysjono-näkymässä luotiin kaikki projektiin kuuluvat tehtävät ja myöhemmin ne sijoiteltiin halutuille sprinteille. Näkymän yläosaan sijoitettiin CustomTopBar-komponentti, jota käytettiin useassa sovelluksen näkymässä ja jonka sisältö muuttui dynaamisesti sille välitettyjen ominaisuuksien mukaan. Tässä näkymässä komponentti piti sisällään vain Uusi tehtävä -painikkeen, jota painamalla avautui TaskModal-komponentti, jota käytettiin sekä uuden tehtävän luontiin että tehtävän muokkaukseen (kuvio 14).



KUVIO 14. TaskModal-komponentti avattuna muokkausta varten

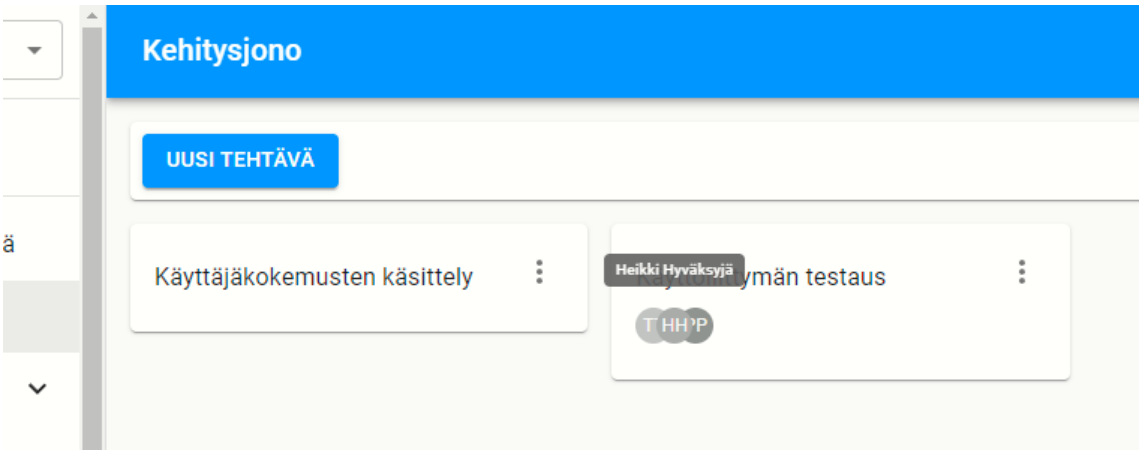
Modaalin peruslogiikka noudatti pitkälti samaa kaavaa kuin aikaisemmatkin modaalit ja sen otsikko sekä alustus muuttui metodin perusteella. Lomakkeen vaadittuja kenttiä olivat otsikko ja tehtävän kategoria, mutta painamalla lisää käyttäjiä -painiketta tehtävälle voitiin merkata myös projektiin kuuluvia jäseniä käyttäjiksi, jotka vastaavat tehtävän suorituksesta. Käyttäjien lisääminen päädyttiin toteuttamaan toisella modaalilla, joka avautui TaskModal-komponentin päälle (kuvio 15). Ajan puutteen takia modaalin ulkoasu kärsi tyylittelyjen puutteesta, koska prioriteettina oli toiminnallisuuden toteuttaminen ja tiedossa oli mahdollinen refaktorointi myöhemmässä vaiheessa.

Modaalin toiminnallisuus poikkesi aikaisemmista modaaleista jonkin verran eikä se esimerkiksi käyttänyt dataa sovelluksen sisäisistä tietovarastoista vaan sille välitettiin ominaisuutena muun muassa sen tarvitsema data sekä submit-tapahtuman metodi. Äitikomponentissa (TaskModal) projektiin kuuluvista jäsenistä suodatettiin addableMembers-muuttujaan javascriptin filter()-metodilla kaikki jäsenet, jotka eivät jo olleet merkattu tehtävän tekijöiksi. Lisäksi äitikomponentissa määritettiin handleUserSubmit-metodi, joka otti parametreina vastaan tapahtuman (event) sekä users-olion, josta tarkistettiin käyttäjät, jotka oli valittu tehtävälle. Modaalin ominaisuutena välitettiin addableMembers-muuttujan arvo sekä handleUserSubmit-metodi ja niiden perusteella modaaliin renderoitiin vaihtoehdot tekijöistä sekä lisää-painikeesta tapahtuva logiikka.



KUVIO 15. Käyttäjän lisäämiseen tarkoitettu modaali avattuna

Tehtävän luonnin jälkeen näkymään renderoitui uusi TaskItem-komponentti, joka sai ominaisuutena luodun tehtävän datan. Datasta renderoitiin otsikko sekä mahdolliset tehtävän suorittajat Avatar-komponentteina, joissa on suorittajien etukirjaimet. Liikuttamalla kursori jonkun avatar-komponentin päälle tai mobiilissa painamalla sitä 100 millisekunnin ajan, se nostettiin päällimmäiseksi ja henkilön koko nimi näytettiin yläpuolella (kuvio 16).

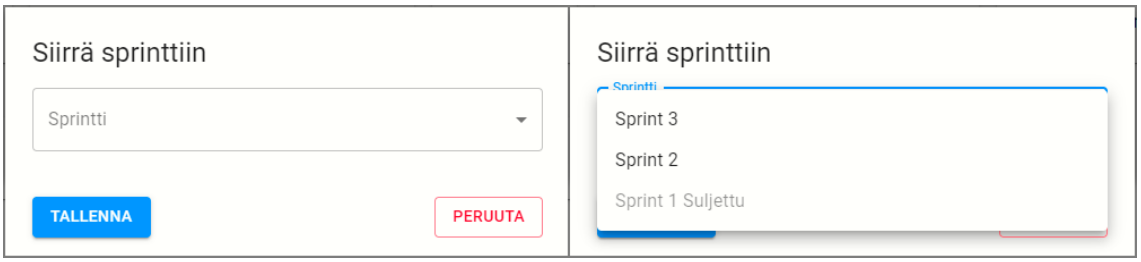


KUVIO 16. Kursori käyttäjän avatarin päällä

TaskItem-komponenttia käytettiin myös sprint-näkymässä ja sen lapsikomponenttina uudelleen käytettiin DropDownMenu-komponenttia, jolle välitettiin tähän käyttötarkoitukseen soveltuvat vaihtoehdot. Kehitysjonossa olevalla tehtävällä vaihtoehdot olivat:

- Siirrä sprinttiin
- Muokkaa tehtävää
- Poista tehtävä

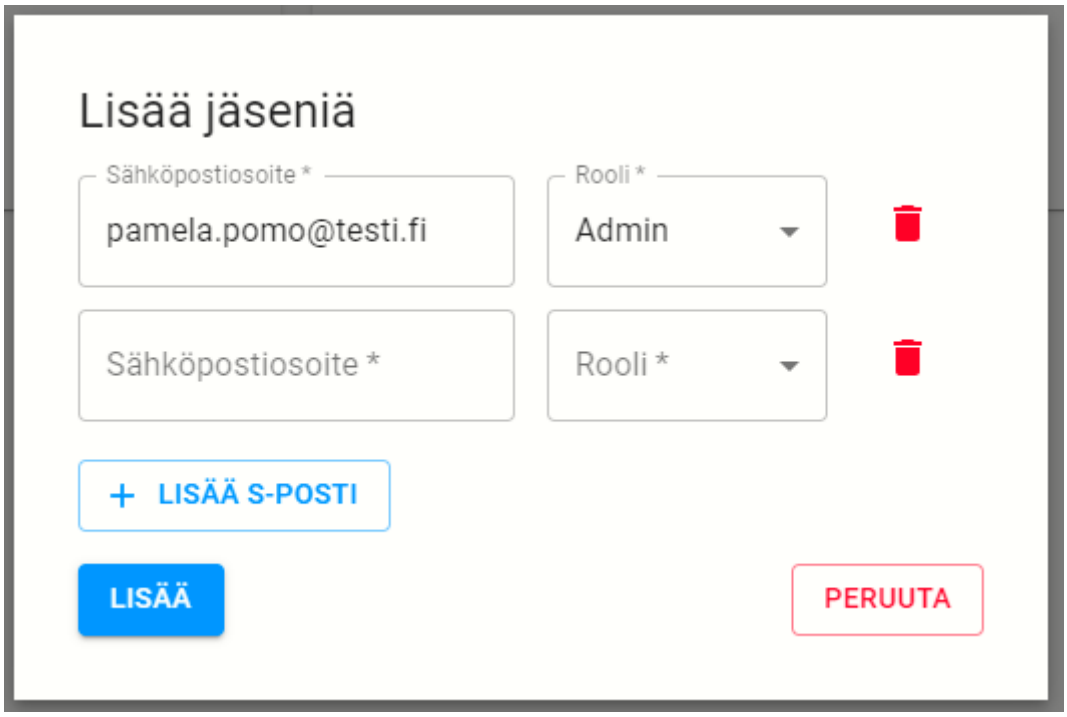
Siirrä sprinttiin -vaihtoehdosta avautui yksinkertainen modaali, jossa Select-komponentilla voitiin valita sprintti, johon tehtävä siirretään. Jos jokin sprintti oli suljettu, valinta oli poistettu käytöstä ja sprintin perässä oli vielä teksti kuvaamassa syytä (kuvio 17). Muokkaa tehtävää -valinnasta avautui TaskModal-komponentti muokkaustilassa ja poista tehtävä -valinnasta ConfirmModal-komponentti poiston vahvistamiseksi. Jos tehtävä oli jo sijoitettu sprintille, yllä mainittujen vaihtoehtojen lisäksi valikossa oli myös ”siirrä kehitysjonoon” vaihtoehto.



KUVIO 17.Vasemmalla puolella modaali avattuna kehitysjonossa olevasta tehtävästä. Oikealla puolella select-komponentin vaihtoehdot avattuna

5.1.6 Käyttäjät-näkymä

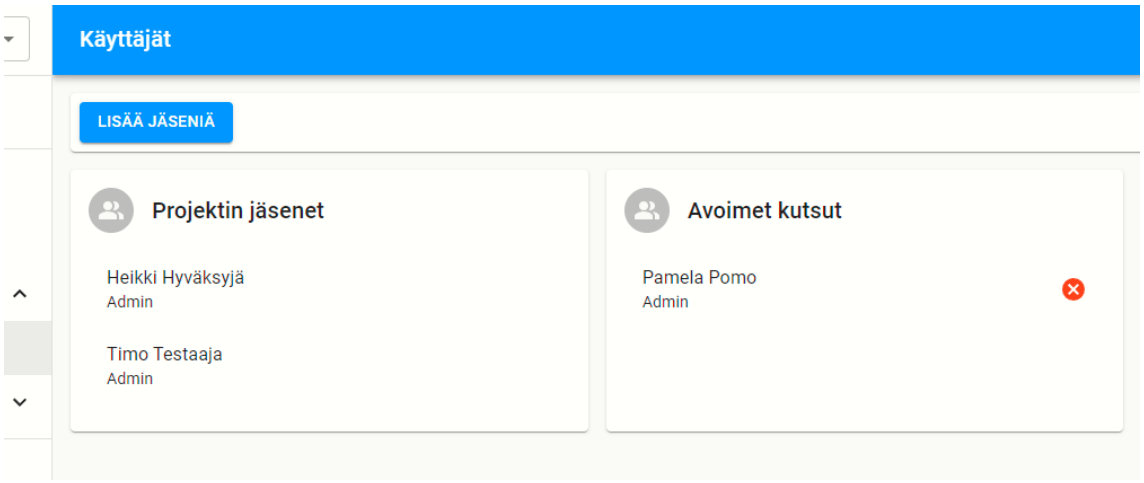
Näkymän yläosassa käytettiin aikaisemmin luotua CustomTopBar-komponenttia ja sen lapsena oli painike, josta projektiin voitiin kutsua uusia jäseniä. Painikkeesta avautui addMembersModal-komponetti, johon renderoitiin jokaista members-taulukon sisällä olevaa oliota kohden yksi rivi sähköpostin ja roolin syöttämistä varten. Komponentin avautuessa members-taulukko alustetaan pitämään sisällään vain yksi olio tyhjillä arvoilla, jolloin käyttäjä voi täyttää annettavat tiedot. Käyttäjiä voitiin kutsua myös useampi samalla kertaa painamalla lisää s-posti -painiketta, jolloin members-taulukkoon lisättiin uusi olio ja tästä johtuen myös käyttöliittymään renderoitui uusi rivi tietojen syöttämistä varten (kuvio 18).



KUVIO 18. Modaaliin lisätty toinen rivi tietojen syöttöä varten

Tietojen tarkistuksen toteutuksessa oli pientä haastetta, mutta lopulta sekin tuntui toimivan halutulla tavalla. Jos lisättiin uusia rivejä, mutta niille ei syötetty tietoja, pyydettiin täyttämään puuttuvat tiedot tai vaihtoehtoisesti ylimääräiset rivit tuli poistaa roskakoripainikkeesta, jotta kutsut voitiin lähettää. Ensimmäisessä versiossa kutsut menivät perille vain niille käyttäjille, jotka olivat rekisteröityneet palveluun.

Projektissa jo mukana olleet jäsenet sekä käyttäjät, joille oli lähetetty kutsu, näytettiin näkymässä omissa osioissaan (kuvio 19). Lähetetyn kutsun pystyi tarvittaessa vielä perumaan rastipainikkeesta, jolloin ConfirmModal-komponentti avautui ja käyttäjää pyydettiin vahvistamaan poisto.

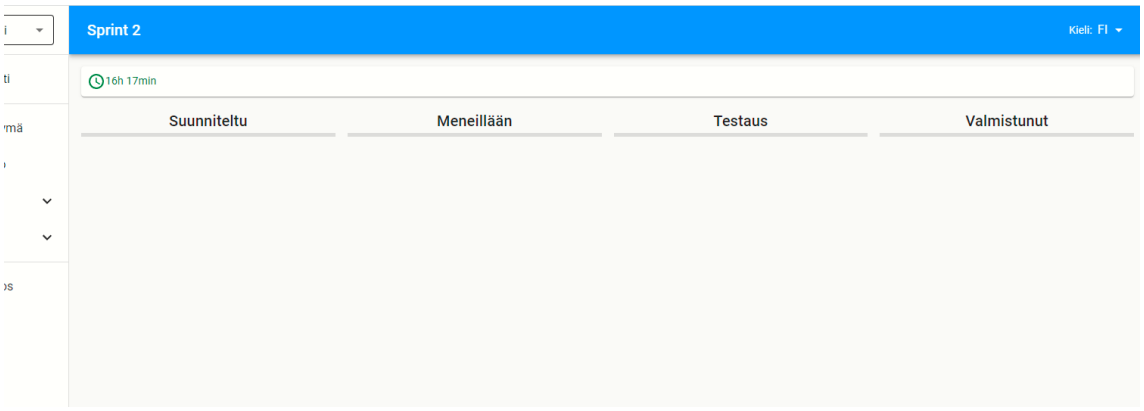


KUVIO 19. Projektiin kuuluvat jäsenet ja avoimet kutsut eroteltu omiin osioihin

5.1.7 Sprint-näkymä

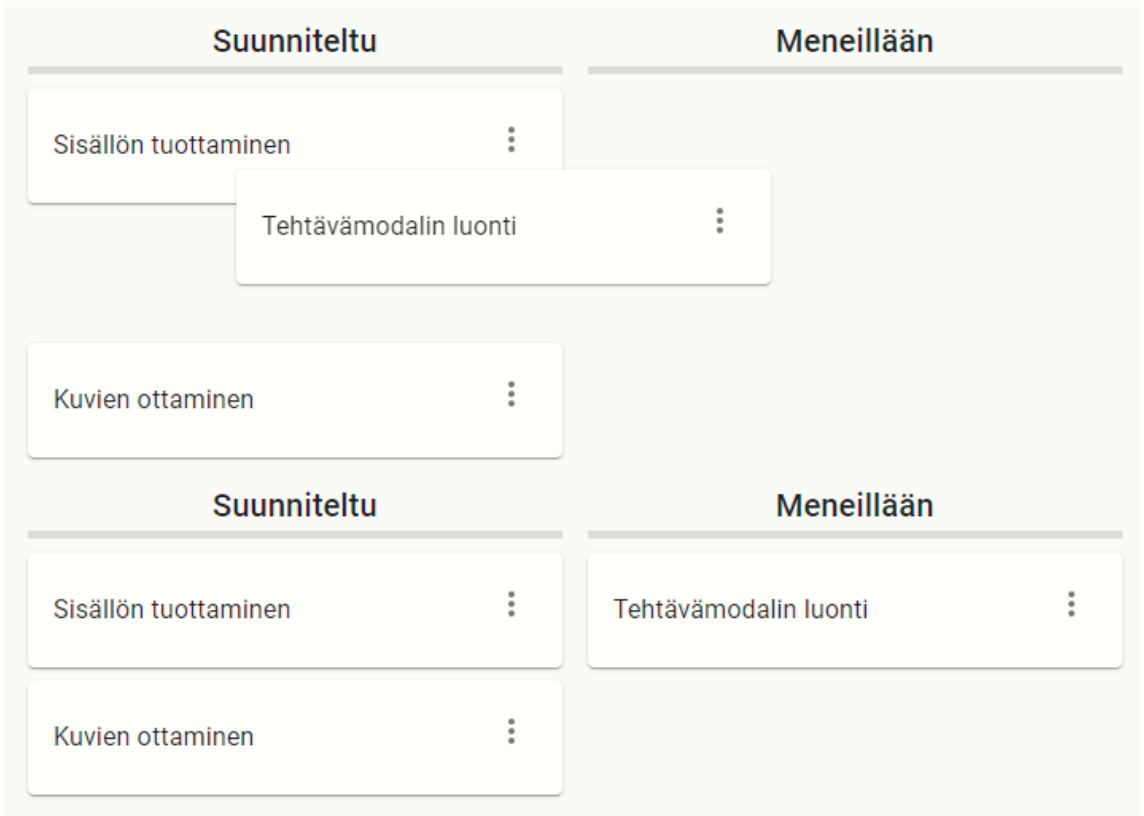
Näkymä oli aina yhden valitun sprintin näkymä, jonka yläosaan sijoitettiin jälleen CustomTopBar-komponentti. Komponentti piti sisällään kelloikonin ja sen perässä esitettiin merkkijonona aika, miten kauan sprinttiä oli jäljellä tai kuinka kauan se oli myöhässä eli ylittänyt suunnitellun päättymisajan. Fontin väri indikoi, onko sprintti myöhässä vai ei ja esitettävästä ajasta riippuen se esitettiin joko päivinä tai tunteina ja minuutteina.

Näkymän ydinajatus oli sprinttiin siirrettyjen tehtävien valmiustilan päivittäminen ja sitä kautta koko sprintin tilanteen ylläpito. Tietokannasta noudetut valmiustilat esitettiin dynaamisesti omina kategorioinaan otsikon kanssa siten, että käytettävänä oleva tila isolla näytöllä jakautui valmiustilojen lukumäärän mukaan (kuvio 20).



KUVIO 20. Yleiskuva näkymästä dynaamisilla valmiustiloilla

Kategorioiden alle renderoitiin sprintille siirretyt tehtävät sen hetkisen valmiustilan mukaan TaskItem-komponenttina ja niitä voitiin raahata ja pudottaa haluttuun kategoriaan, jolloin valmiustila päivitettiin siirron mukaan (kuvio 21). Toiminnallisuuden pohjalla oli react-beautiful-dnd-kirjasto, jonka avulla raahaa ja pudota -toiminnallisuus oli helppo toteuttaa.



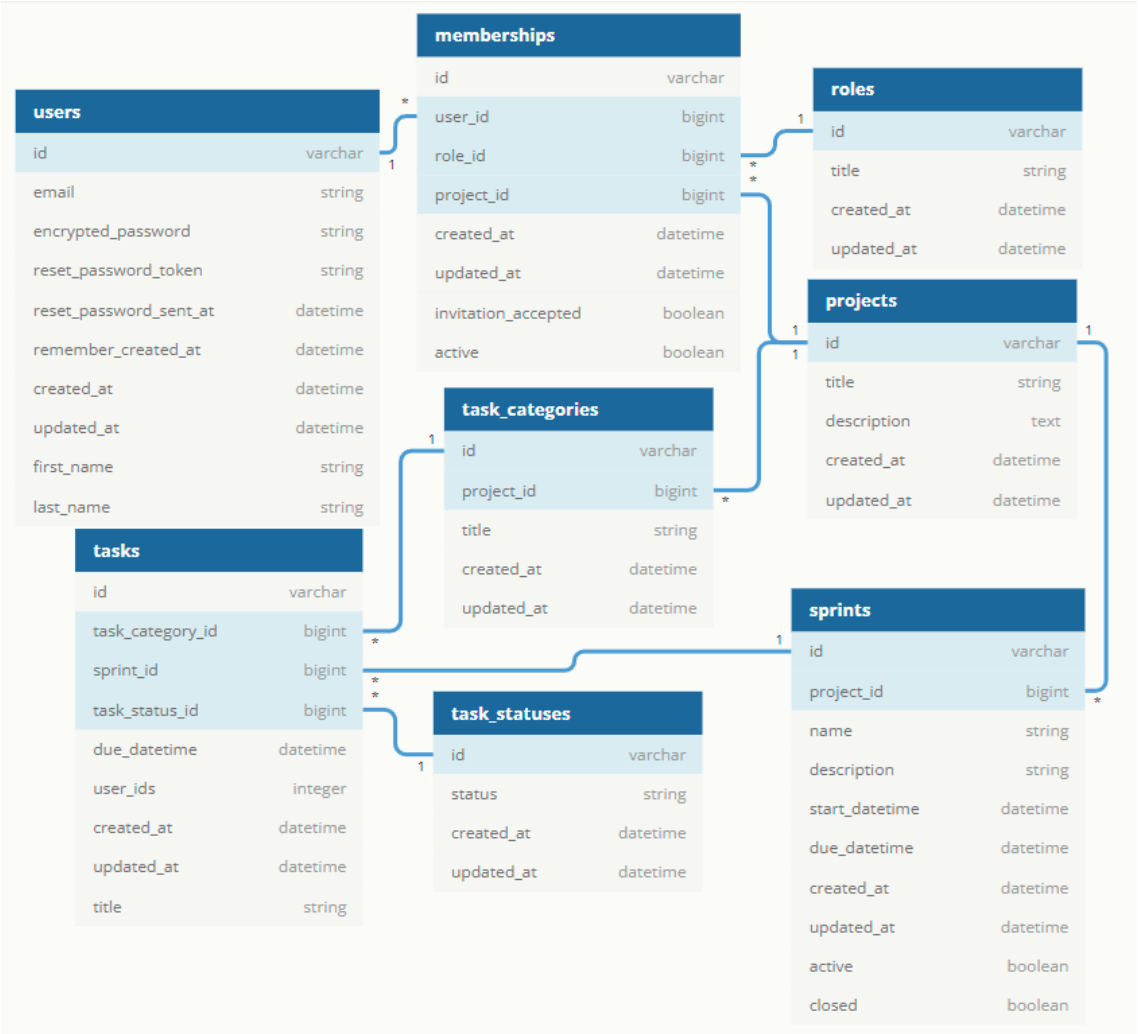
KUVIO 21. Ylemmässä raahaa jo pudota -toiminnallisuus aloitettu. Alemmassa tehtävän tila päivitetty

5.2 Tietokanta

Ennen varsinaisen backend-kehityksen aloittamista tietokantajärjestelmään ja Rails-sovellukseen joutui tekemään jonkin verran alustavia asennuksia ja asetuksien määrittelyjä. PostgreSQL-tietokantajärjestelmään luotiin Rails-sovellukselle oma käyttäjä, jolla oli oikeudet luoda ja muokata tietokantoja ja näitä käyttäjätietoja käytettiin sovelluksen kehitysympäristössä. Tietokannan hallinnointi tehtiin Rails-sovelluksen kautta kirjoittamalla komentoja Visual Studio Code -tekstieditorin komentoriville, joten sovellukseen asennettiin pg-kirjasto, joka on Ruby-käyttöliittymä PostgreSQL-tietokantajärjestelmään.

Kun Rails-sovelluksen database.yml-tiedostoon oli määritetty kehitysympäristössä käytettävän tietokannan asetukset ja muut tarvittavat tiedot, opinnäytetyön tietokanta luotiin kirjoittamalla komentoriville komento rails db:create. Tietokannan olemassaolo tarkistettiin vielä erikseen pgAdmin-työkalulla, joka on graafinen käyttöliittymä PostgreSQL-tietokannan hallinnointiin. Rails-sovelluksen ja tietokannan välisen yhteyden toimiessa, niiden kehitys ja rakentaminen kulkivat käsi kädessä malli (model) ja tietokantataulu kerrallaan.

Kehityksen aikana havaittiin, että joistakin tauluista puuttui tarvittavaa tietoa tai jokin sarake oli väärässä taulussa. Esimerkkinä voidaan ottaa projektin aktiivisuuden määrittämä active-sarake, joka aluksi kiireessä ja ajattelematta lisättiin projects-tauluun, kunnes ennen ongelmien ilmenemistä tajuttiin sen aiheuttavan ongelmia. Teoreettisena skenaariona käyttäjän muuttaessa aktiivista projektia, kaikille käyttäjille olisi vaihtunut valittu projekti aktiiviseksi. Tämä olisi ollut erityisen ongelmallista varsinkin käyttäjien kohdalla, jotka eivät kuuluneet aktiiviseksi muutettuun projektiin. Tämä saatiin onneksi helposti korjattua ja sarake laitettiin membership-tauluun, jolloin voitiin muuttaa tietyn käyttäjän aktiivista projektia. Lopullisessa tietokannan versiossa tauluja luotiin kahdeksan kappaletta ja sarakkeet löysivät oikean paikkansa (kuvio 22).



KUVIO 22. Tietokantakaavio ensimmäisestä versiosta

5.3 Ruby on Rails-sovellus

Kun Rails-sovelluksen ja tietokannan välinen yhteys oli saatu toimimaan, tiedossa oli vielä muutaman kirjaston asennus ennen kehityksen jatkamista. Koska asiakasohjelma ja ohjelmistorajapinta toimivat eri portin takana ja tulevaisuudessa voisi olla vielä natiivit sovellukset, tiedostettiin tulevat CORS- eli Cross-Origin Resource Sharing -ongelmat, joten Rails-sovellukseen asennettiin rack-cors -kirjasto, jolla voitiin määrittää CORS-asetukset niin että rajapintakyselyt menevät läpi, vaikka kyselyn lähettäjän ja rajapinnan verkkotunnukset poikkeavat toisistaan.

Kun ensisijaiset asennukset ja määrittelyt oli saatu suoritettua, asennettiin Devise-kirjasto, generoitiin devise user -malli oletusasetuksilla ja suoritettiin siitä syntynyt migrate-tiedosto, joka loi tietokantaan Users-taulun. Lisäksi asennettiin kirjaston laajennus devise-jwt, joka käyttää JWT-tunnistetta käyttäjän todentamiseen ja tunniste määritettiin välittymään onnistuneesta kirjautumisesta

palautuvan vastauksen otsikkotietoihin. Devise-kirjaston ja sen laajennuksen asetusten määrittelyt veivät jonkin aikaa, esimerkiksi user-malli muokattiin käyttämään laajennuksen todentamista sekä generoitiin secret_key-avain, jota käytettiin luotujen JWT-tunnisteiden allekirjoitukseen, mutta lopulta kuitenkin päästiin testaamaan käyttäjätunnuksen luontia sekä kirjautumista ja tätä varten routes.rb-tiedostoon määritettiin API-päätepisteet.

Testaus suoritettiin Postman-sovelluksen avulla, jolla pystyi nopeasti ja helposti suorittamaan rajapintakyselyitä sekä tarkastelemaan palautuvia vastauksia. Kun käyttäjäntunnuksen luonti ja onnistuneesta kirjautumisesta palautuva tunniste oli saatu todennettua, päästiin jatkamaan kehitystä.

5.3.1 Mallit ja käsittelijät

Rails-sovelluksen ja sitä mukaan tietokannan kehitys ja rakentaminen etenivät käsite kerrallaan. Vaikka pohjalla oli visio ja karkea hahmotelma tietokannasta, ne usein miten muuttuivat toteutuksen aikana. Esimerkiksi heti projektin alkuvaiheessa käyttäjiin ja projekteihin liittyviä malleja, käsittelijöitä ja tietokantatauluja luodessa alettiin pohtimaan näiden suhteita toisiinsa sekä tarvittavaa dataa ja päädyttiin muuttamaan alkuperäistä tietokantasuunnitelmaa. Ensin user_project-välitaulu muuttui membership-tauluksi ja tämän rinnalle tuli roles-taulu, johon tallennettiin erilaiset roolit ja jota käytettiin lopulta projektikohtaisen roolin määrityksessä.

Projektissa ilmenevä käsitteitä olivat:

- Jäsenyys (membership)
- Projekti (project)
- Rooli (role)
- Sprintti (Sprint)
- tejtäväkategoria (task_category)
- Tehtävä (task)
- Tehtävän tila (task_status)
- Käyttäjä (user)

Näistä jokaista kohden luotiin malli kirjoittamalla komentoriville rails generate model -komento, joka loi tarvittavat tiedostot automaattisesti. Esimerkiksi komento ”rails generate model project

title:string description:string” loi Rails-projektin models-kansioon project.rb-tiedoston eli mallin sekä migare-kansioon tiedoston, joka piti sisällään tietokantaan tehtävät muutokset. Joissain tapauksissa migrate-tiedostoa piti muokata generoinnin jälkeen ja lopulta se piti sisällään mallista riippuen esimerkiksi luotavan taulun, sen sarakkeet, avaimet, indeksoinnit ja oletusarvot. Mallin generointi loi myös kaksi testaukseen tarkoitettua tiedostoa, joita ei käytetty projektissa.

User-mallia lukuun ottamatta, johon tuli Devise-kirjaston määrityksiä, model-tiedostoihin määritettiin vain mallien yhteydet toisiinsa. Koska tietokannan taulujen väliset yhteydet olivat yhden suhde moneen (1-n), käytettiin model-tiedostoissa vain Rails-sovelluskehyksen belong_to- ja has_many-assosiaatioita.

Jokaista käsitettä kohden luotiin myös käsittelijä rails generate controller -komennolla. Toisin kuin model-tiedostojen kohdalla, käsittelijät nimettiin monikossa, koska ne eivät vastaa yhtä tiettyä tietokantatietuetta. Esimerkiksi komento ”rails generate controller api/projects” loi Rails-projektin controllers/api -polkuun projects_controller.rb-tiedoston.

Käsittelijät pitivät sisällään logiikan mitä tietyt rajapintakyselyt toteuttavat. Pääsääntöisesti jokaisen käsittelijän alussa tarkistetaan, että rajapintakyselyn takana on kirjautunut ja vahvistettu käyttäjä hyödyntäen Rails-sovelluskehyksen before_action-metodia ja Devise-kirjaston tarjoamaa authenticate_user-metodia. Osa käsittelijöistä olivat hyvinkin yksinkertaisia ja pitivät sisällään vähän kirjoitettua koodia (kuvio 23), kun taas toiset pitivät sisällään paljon metodeja ja logiikkaa. Yleisimpiä käsittelijöissä esiintyviä metodeja olivat:

- index
- create
- update
- destroy

Index-metodit hakivat pääsääntöisesti aina suuren määrän dataa tietokannasta taulukko-tyyppisenä datana ja palauttivat sen vastauksena. Joissain tapauksissa tietokannasta haettiin kaikki tietyn taulun tietueet ja toisissa taas määritettiin logiikkaa, jolla haettiin vain tarkasti määritetyt tietueet. Create-metodeita käytettiin, kun haluttiin tallentaa uusi tietue tietokantaa ja ne pitivät sisällään vaihtelevissa määrin logiikkaa tietueen tallentamiseksi, mutta palauttivat kuitenkin aina tallennetun tietueen datan. Update-metodeja käytettiin, kun haluttiin päivittää jonkin olemassa olevan tietueen

yhtä tai useampaa tietoa ja vastauksena palautettiin muutetun tietueen data. Destroy-metodeilla poistettiin haluttu tietue tietokannasta ja vastauksena palautettiin poistetun tietueen data.

```
1  class Api::TaskStatusesController < ApplicationController
2    before_action :authenticate_user!
3
4    def index
5      @task_statuses = TaskStatus.all
6
7      render :index
8    end
9  end
```

KUVIO 23. task_statuses_controller.rb-tiedoston koodi

Eniten logiikkaa oli uutta projektia luodessa, vaikka tietokannassa project-taulu piti itsessään sisällään vain vähän dataa. Uuden projektin luonnin yhteydessä tietokannan membership-taulusta haettiin jäsenyys, joka kuvaa käyttäjän aktiivista projektia eli tietuetta, jonka active-sarakkeen arvoksi oli määritetty tosi ja jos sellainen löytyi, sarakkeen arvo muutettiin epätodeksi. Samalla membership-tauluun luotiin uusi tietue määritetyillä active- ja role_id-sarakkeiden tiedoilla, jolloin käyttäjälle muodostui admin-tason jäsenyys luotuun projektiin ja projekti oli niin sanotusti aktiivinen projekti. Projektille luotiin myös ensimmäinen sprintti sprint-tauluun sekä kaksi oletuskategoriaa task_category-tauluun, joita käyttäjä voi halutessaan muokata jälkeenpäin. Kaikki tämä tapahtuu transaktionin sisällä eli jos joku näistä tapahtumista epäonnistuu, mitään ei tallennu tietokantaan ja käyttäjä saa ilmoituksen virheestä.

5.3.2 Näkymät (views)

Rails-sovelluksessa ei varsinaisesti käytetty näkymiä, vaikka käytettävät tiedostot sijaitsivat views-kansiossa. Tiedostot ovat malleja (template), jotka on kirjoitettu Jbuilder-täsmäkielellä kuvaamaan vastauksessa palautettavan datan JSON-rakennetta. Kun esimerkiksi luotiin uutta tehtävää käsittelijän create-metodissa ja sen datasta oli tehty ilmentymä @task-muuttujaan, if-else-ehtorakenteella tarkistettiin, onnistuiko tehtävän tallennus tietokantaan ja jos onnistui, kutsuttiin Rails-sovelluskehyksen render-metodia, jolle välitettiin projektin ilmentymä (kuvio 24).

```ruby
def create
  @task = Task.new(task_params)

  status = TaskStatus.find_by(status: "planned")
  @task.task_status = status

  if @task.save
    render @task
  else
    render json: {error: 'Unable to create task'}
  end
end
```

KUVIO 24. tasks_controller.rb-tiedoston create-metodi

Sovellus haki automaattisesti views/api/tasks-polussa sijaitsevan _task.json.jbuilder-tiedoston ja renderoi palautettavan vastauksen sen mukaan. Tehtävästä palautettava JSON-vastaus koostuu, tasks-, task_statuses- ja users-taulujen datasta ja koska tehtävälle merkatut käyttäjät päätettiin toteuttaa taulukkomuotoisena sarakkeena, haettiin ensin kaikkien tehtävään merkattujen käyttäjien data muuttujaan. Käyttäjien data käytiin toistorakenteessa läpi ja jokainen käyttäjä lisättiin tehtävän JSON-rakenteeseen _user.json.jbuilder-tiedostossa määritetyn JSON-rakenteen mukaan. Muu tehtävään liittyvä data saatiin mukaan parilla yksinkertaisella koodirivillä (kuvio 25). Vastauksen mallissa voitiin tarvittaessa käyttää muun muassa mallien assosiaatioita, Devise-kirjaston current_user-metodia ja erilaisia toistorakenteita JSON-rakenteen muodostamiseksi.

```ruby
users = User.where("id IN (?)", task.user_ids)

json.(task, :id, :title, :task_category_id, :sprint_id, :due_datetime, :created_at)

json.status task.task_status.status

json.users do
  json.partial! partial: "api/users/user", collection: users, as: :user
end
```

```
Task data ▼{id: 51, title: 'Käyttöliittymän testaus', task_category_id: 24, sprint_id: null, due_datetime: null, …}
    created_at: "2021-11-08T09:09:35.083Z"
    due_datetime: null
    id: 51
    sprint_id: null
    status: "planned"
    task_category_id: 24
    title: "Käyttöliittymän testaus"
  ▼users: Array(3)
    ▶0: {id: 1, email: 'testi@testi.fi', first_name: 'Timo', last_name: 'Testaaja', private: {…}}
    ▶1: {id: 2, email: 'testi1@testi.fi', first_name: 'Heikki', last_name: 'Hyväksyjä'}
    ▶2: {id: 3, email: 'testi2@testi.fi', first_name: 'Pamela', last_name: 'Pomo'}
      length: 3
    ▶[[Prototype]]: Array(0)
  ▶[[Prototype]]: Object
```

KUVIO 25. Yläpuolella JSON-rakenteen muodostaminen _task.json.jbuilder-tiedostossa. Alapuolella palautuva vastaus

5.3.3 Routes

Routes.rb-tiedostoon määritettiin sovelluksen reitit, jotka toimivat myös ohjelmistorajapinnan päätepisteinä. Tehtäessä rajapintakysely tietyllä URL-osoitteella, sovellus kävi läpi routes-tiedostossa määritetyt päätepisteet ja antoi virheilmoitusta, jos URL-osoite ei vastannut yhtäkään reittiä. Jos taas osoitteelle löytyi vastaavuus, määritetystä käsittelijästä suoritettiin tapahtumaa vastaava metodi.

Reitityksien määrittämisessä käytettiin niin sanottuja resourceful- ja non-resourceful-määrityksiä (kuvio 26). Devise-kirjaston reitit määritettiin erillään muista reiteistä ja sen käyttämät oletusreitit muutettiin halutuiksi. Lisäksi mukaan määritettiin itse määritetty me-reitti, josta käyttäjä hakee itseensä liittyvän datan onnistuneen kirjautumisen jälkeen.

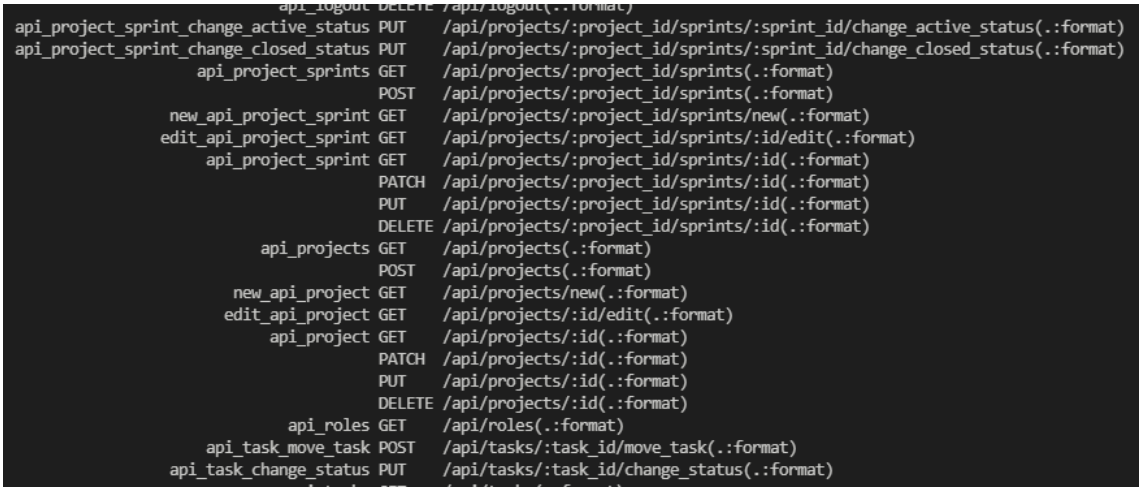
```
Rails.application.routes.draw do
  devise_for :users, defaults: { format: :json }
  devise_scope :user do
        get "api/me", to: "api/users#me"
        post "api/signup", to: "api/registrations#create"
        post "api/login", to: "api/sessions#create"
        delete "api/logout", to: "api/sessions#destroy"
  end

  namespace :api, defaults: {format: :json} do

    resources :projects do
      resources :sprints do
        put "/change_active_status", to: "sprints#change_active_status"
        put "/change_closed_status", to: "sprints#change_closed_status"
      end
    end

    resources :roles, only: [:index]

    resources :tasks do
      post "/move_task", to: "tasks#move_task"
      put "/change_status", to: "tasks#change_task_status"
    end

    resources :task_categories
    post "/task_categories/get_project_categories", to: "task_categories#get_project_categories"

    resources :task_statuses

    resources :memberships do
      put "/accept", to: "memberships#accept_invitation"
      put "/deny", to: "memberships#deny_invitation"
    end
    post "/memberships/change_active_project", to: "memberships#change_active_project"
    post "/memberships/add_members", to: "memberships#add_members"
  end
end
```

KUVIO 26. Sovellukseen toteutetut reititykset

Rails-sovelluskehyksen resourceful-määrityksessä käytettiin resource-metodia, jolloin sovellus loi automaattisesti reitit annetun käsittelijän index-, show-, new-, edit-, create-, update-, destroy-tapahtumille. Käyttämällä resourceful-määrityksessä only-valintaa, voitiin kertoa sovellukselle

mitkä tietyt reitit haluttiin sen luovan. Sovelluksessa käytettiin myös sisäkkäistä resourceful-määrittelyä, jossa projektien sisälle määritettiin sprinttien reitit. Tämä oli loogista, koska jokainen sprintti kuului aina johonkin projektiin ja näin ollen URL-osoitteeseen voitiin sijoittaa tarvittavia parametreja.

Non-resourceful-määrityksiä käytettiin tapauksissa, joissa resourceful-määritys ei sopinut. Tämän tapaisia tilanteita olivat niin sanotut kustomoidut metodit, esimerkiksi aktiivisen projektin muuttaminen. Kirjoittamalla komentoriville komento ”rails routes” saatiin esiin kaikki luodut reitit luettavammassa muodossa, joka helpotti ainakin päätepisteiden hahmotuksessa (kuvio 27).



KUVIO 27. Osa rails routes -komennon tulosteesta

6 POHDINTA

Järjestelmän toteutus onnistui mielestäni hyvin, koska lähes kaikki alussa suunnitellut perustoiminnallisuudet saatiin toteutettua, vaikka uusia ideoita tulvi koko ajan lisää järjestelmää kehitettäessä. Pelkässä asiakassovelluksessa olisi ollut riittävä laajuus opinnäytetyöksi, mutta mielestäni se olisi ollut järjestelmän toimivuuden kannalta vajavainen ja huomioiden oma kokemus, päätin toteuttaa koko järjestelmän. Koska kokemusta oli jo React-kirjastosta ja Rails-sovelluskehyksestä, lisähaastetta haettiin esimerkiksi toteuttamalla komponentit funktionaalisina luokkakomponenttien sijaan ja käyttämällä niissä koukkuja. Laajuudesta johtuen raporttiin jouduttiin valitsemaan tietyt aiheet, joihin perehdyttiin syvemmin kuin toisiin.

Jotkin toiminnot jätettiin tietoisesti pois, esimerkiksi projektin poiston kohdalla olisi pitänyt miettiä jäsenten rooleja pitemmälle sekä siihen että itse poistoon liittyvää logiikkaa eri tilanteissa. Moni toiminnallisuus tehtiin osittain valmiiksi siksi, että nähtiin, onko mielessä ollut idea mahdollisuus toteuttaa ajatellulla tavalla. Monet suunnitellut toiminnot kuitenkin jäivät toteuttamatta tai tehtiin osittain ihan vain ajan puutteen vuoksi, koska raportin kirjoittaminen vei huomattavasti enemmän aikaa ja energiaa mitä osasi odottaa. Raportin osioita kirjoitettiin uusiksi useampaan kertaan kokonaan tai osittain, jotta tekstistä saadaan mahdollisimman helppolukuista ja ymmärrettävää.

Jatkokehitysideoita on valtavasti, mutta ensimmäisessä versiossa käyttäjään ja hänen rooleihinsa perustuvia tarkistuksia toteutettiin vähän, joten tämä olisi ensimmäisiä toteutettavia asioita. Raportin kirjoittamisen yhteydessä toteutettujen testien yhteydessä ilmeni myös koodivirheitä muun muassa uudelleen renderoitumisen kanssa, joten nämä olisivat myös korkealla prioriteetilla jatkokehityksessä. Koska omiin vahvuuksiini ei kuulu käyttöliittymäsuunnittelu, olisi myös syytä hankkia ammattilaisen toteuttamat suunnitelmat, joiden pohjalta näyttävä ja toimiva käyttöliittymä on huomattavasti helpompi toteuttaa.

Olen lopputulokseen tyytyväinen. Järjestelmän toteutus antoi mukavasti haastetta oppia uutta ja kehittää omia taitoja, joka olikin yksi tärkeimmistä motivaattoreista opinnäytetyötä aloittaessa. Vaikka jatkokehitettävää on vielä, ei suunnitelmissa ollut saada valmista tuotetta jonka kaupallistamista voisi harkita, joten järjestelmän kehitys jää hyvin mahdollisesti tähän ja siirretään katse tulevaan.

LÄHTEET

A M, V & Sonpatki, P. 2016. ReactJS by Example – Building Modern Web Applications with React. E-kirja. Viitattu 30.9.2021, https://learning.oreilly.com/library/view/reactjs-by-example/9781785289644.

Devise 2021. Devise. Viitattu 11.10.2021, https://github.com/heartcombo/devise

Jbuilder 2021. Jbuilder. Viitattu 11.10.2021, https://github.com/rails/jbuilder

JWT 2021. Introduction to JSON Web Tokens. Viitattu 12.10.2021, https://jwt.io/introduction

MobX 2021. The gist of MobX. Viitattu 6.10.2021, https://mobx.js.org/the-gist-of-mobx.html

MUI 2021. The React UI Library you always wanted. Viitattu 30.9.2021, https://mui.com.

Porcello, E & Banks, A. 2020. Learning React, 2nd Edition. E-kirja. Viitattu 30.9.2021, https://learning.oreilly.com/library/view/learning-react-2nd/9781492051718.

PostgreSQL 2021. About. Viitattu 9.10.2021, https://www.postgresql.org/about/

React 2021a. React.Component. Viitattu 30.9.2021, https://reactjs.org/docs/react-component.html.

React 2021b. Using the State Hook. Viitattu 30.9.2021, https://reactjs.org/docs/hooks-state.html.

React 2021c. Rules of Hooks. Viitattu 30.9.2021, https://reactjs.org/docs/hooks-rules.html

React Router 2021. Primary Components. Viitattu 6.10.2021, https://reactrouter.com/web/guides/primary-components

Ruby 2021. About Ruby. Viitattu 13.10.2021, https://www.ruby-lang.org/en/about/

Ruby on Rails 2021a. Action Controller Overview. Viitattu 9.10.2021, https://guides.rubyonrails.org/action_controller_overview.html

Ruby on Rails 2021b. Getting Started with Rails. Viitattu 9.10.2021, https://guides.rubyonrails.org/getting_started.html

Thakkar, M. 2020. Building Apps with Server-Side Rendering: Use React, Redux, and Mext to Build Full Server-Side Rendering Applications. E-kirja. Viitattu 30.9.2021, https://learning.oreilly.com/library/view/building-react-apps/9781484258699