Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Simo Ojala

Hallintapaneeli tilauksien käsittelyyn

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tutkinto-ohjelma

Insinöörityö

04.03.2021

Tekijä Otsikko Sivumäärä Aika	Simo Ojala Hallintapaneeli tilauksien käsittelyyn 26 sivua 04.03.2021
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Yliopettaja Auvo Häkkinen

Insinöörityön tarkoituksena oli kehittää verkkosovellus mobiilisovelluksesta tulevien tilauksien hallintaan. Sovellus on tarkoitettu käytettäväksi ensisijaisesti tietokoneilla. Insinöörityössä perehdyttiin React-sovelluksiin ja niissä käytettäviin komponentteihin. Komponentit ovat React-ohjelmoinnin perusta, ja ne voidaan jakaa kahteen kategoriaan: luokkakomponentteihin ja funktionaalisiin komponentteihin. Työssä tutkittiin ja vertailtiin näitä kahta eri tapaa rakentaa ja käyttää komponentteja. Lisäksi työssä esiteltiin funktionaalisissa komponenteissa käytettäviä Hook-funktioita.

Työssä esiteltiin myös Googlen Firebase-palveluita, jotka toimivat sovelluksen palvelinpuolen ratkaisuna. Firebase on minimaalisella konfiguroinnilla käyttöön otettava web- ja mobiilisovelluksille tarkoitettu alusta. Firebasen avulla kehittäjien ei tarvitse käyttää aikaa back-end kehitykseen, vaan tarjolla on valmiita paketteja, joista valita. Tähän insinöörityöhön Firebasen palveluista on otettu käyttöön Cloud Firestore, Authentication, Storage ja Cloud Functions.

Insinöörityön lopputuloksena syntyi vaatimuksien täyttävä verkkosovellus, joka oli omasta sekä asiakkaan mielestä onnistunut. Verkkosovellus oli jo itsessään jatkokehitystä kokonaisuuteen kuuluvalle mobiilisovellukselle, eikä tiedossa ole enää lisää jatkokehitysideoita.

Avainsanat	ReactJS, React hook

Author Title Number of Pages Date	Simo Ojala Web Application for Order Management 26 pages 04 March 2021
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Auvo Häkkinen, Senior Lecturer

The purpose of the study was to create a web application for order management. The orders are created using a mobile application related to the present project. The web application is primarily intended for use on personal computers. The thesis introduces React applications and how they are built. React applications consist of components, which are independent, reusable pieces of code. Components can be separated into two categories: class and functional components. The two categories are presented and compared in the study, together with Hook-functions used in functional components.

The thesis also introduces Google Firebase, which was used as the back-end in the project. Firebase is a platform for mobile and web applications. With Firebase, programmers do not need to focus on back-end development. For this project, Cloud Firestore, Authentication, Storage and Cloud Function services were used from Firebase.

The result of the project was a working web application which fulfilled all the requirements set for it.

Keywords	ReactJS, React hook

Sisällys

Lyhenteet

1 Johdanto 1

2 React-hookit ja funktionaaliset komponentit 2

2.1 Funktionaaliset ja luokkakomponentit 3
2.2 React-hookit 4
2.3 State-hook 5
2.4 Effect-hook 7
2.5 Omat hookit 11

3 Kehitysympäristö 12

4 Palvelinpuoli ja tietokanta 13

4.1 Cloud Firestore -palvelu 13
4.2 Firebase Authentication -palvelu 17
4.3 Firebase Storage -palvelu 20
4.4 Firebase Cloud Functions -palvelu 20

5 Käyttöliittymä 20

6 Yhteenveto 24

Lähteet 26

Lyhenteet

DOM	Document Object Model. Malli, jonka avulla määritellään dokumentissa olevat elementit.
HTML	Hypertext Markup Language. Merkintäkieli, jolla kuvaillaan verkkosivujen sisällön rakenne.
JSX	JavaScript XML. JavaScriptin syntaktinen laajennus, jolla voi kuvata Reactin komponentteja.
SDK	Software development kit. Kokoelma ohjelmistokehityksen työkaluja yhdessä asennettavassa paketissa.

1 Johdanto

Verkkosovellusten historian aikana sovellusteknologiat ovat kehittyneet kovaa tahtia. Kehittämiseen on tarjolla niin paljon erilaisia vaihtoehtoa, että aloittaminen saattaa tuntua sekavalta aloittelevalle kehittäjälle. Näiden teknologioiden seasta yksi kasvattaa edelleen suosiotaan: ReactJS [1]. Reactin suosiosta kertookin myös se, että monet suuret ja tunnetut yritykset, kuten Facebook, Netflix, WhatsApp, Twitter, Instagram ja Dropbox käyttävät sitä [2].

React-sovellukset koostuvat useista komponenteista, jotka asetetaan puumaiseen rakenteeseen. Komponentit ovat React-ohjelmoinnin perusta ja yksinkertainen komponentti voi olla esimerkiksi pelkkä painike. Komponentit luodaan JSX-syntaksilla, joka on JavaScriptin syntaktinen laajennus. Reactin komponentteja voi kirjoittaa kahdella eri tavalla: luokkakomponentti tai funktionaalinen komponentti. Yksi insinöörityön tavoitteista on esitellä Reactin funktionaaliset komponentit ja niissä käytettävät hookit sekä vertailla niitä luokkakomponentteihin. Hookit mahdollistavat tilan ja muiden Reactin ominaisuuksien käytön ilman luokkakomponenttia.

Insinöörityön tarkoituksena on kehittää Next Peak Oy:n kanssa yhteistyössä suunnitelmien pohjalta verkkosovellus tilausten hallintaan Asfalttipartio Oy:lle. Next Peak on erikoistunut yritysten strategisen toiminnan ja myyvien tuotteiden kehittämiseen. Next Peak auttaa yrityksiä kohti digitaalista muutosta strategian, operatiivisen toiminnan ja asiakaskokemuksen näkökulmasta. Asfalttipartio Oy on asfalttivaurioiden asiantuntija ja tarjoavat nopeaa ratkaisua asfalttipintojen sekä pihojen vaurioihin. Asfalttipartio haluaa tuoda lisäarvoa asiakkailleen sovelluksen avulla. Ongelmana heillä on, että yhdestä kohteesta voi tulla monta ilmoitusta samanaikaisesti. Sovelluksen avulla tilauksia pystyisi suodattamaan eri parametreilla sekä lajittelemaan tulokset taulukon sarakkeiden perusteilla. Sovelluksella pystyisi myös muokkaamaan tilauksiin mahdollisesti muodostuneet virheet.

Kesällä yhteistyössä Next Peak Oy:n kanssa kehittämällämme Asfalttipartio sovelluksella voidaan tehdä ilmoituksia tienpintojen vaurioista. Sovelluksella Asfalttipartio ja mahdolliset alihankkijat voivat pitää asiakkaan ajan tasalla päivittämällä ilmoitukseen työn etenemistä.

Asfalttipartio mobiilisovellus koostuu kahdesta eri osasta: asiakkaan sovelluksesta, jolla asiakas voi tehdä edellä mainittuja ilmoituksia, ja Asfalttipartion sovelluksesta, jolla Asfalttipartion työntekijä tai alihankkija voi selata ja ottaa ilmoituksia työn alle. Asfalttipartion mobiilisovellus onkin tarkoitettu ensisijaisesti ilmoituksien käsittelyyn eikä hallinnointiin. Tämä ei kuitenkaan tarkoita sitä, etteikö mobiilisovelluksella voisi hallinnoida pieniä määriä ilmoituksia varsin vaivattomasti.

Websovelluksen idea alkoi muodostua tarpeesta pystyä hallinnoimaan useita ilmoituksia, joita saattaa olla yhdessä kohteessa monta kerralla. Kun tilauksia kertyy yksittäiselle kohteelle monta, on niiden hallinnointi pelkällä mobiilisovelluksella hankalaa. Ennen web- tai mobiilisovellusta tilaukset käsiteltiin ja hallinnoitiin käsin.

Insinöörityön vaatimuksena oli kehittää websovellus, jolla voisi helposti hallinnoida suuria määriä asiakkaan mobiilisovelluksesta tulevia ilmoituksia. Asiakkaat voisivat itse myös halutessaan käyttää websovellusta omien ilmoituksien tilan seuraamiseen. Asfalttipartion kokemuksen perusteella jotkut asiakkaat tekevät suuria määriä ilmoituksia kerralla, jolloin he myös hyötyisivät websovelluksen tarjoamasta kokonaiskuvasta.

Tekniset vaatimukset olivat, että sovelluksen tulisi olla web-pohjainen ja siinä tulisi hyödyntää jo olemassa olevaa tietokantaa, joka on toteutettu Googlen Firebase-palvelulla. Käyttöliittymän tuli ensisijaisesti tukea käytettävyyttä tietokoneilla.

Aluksi insinöörityössä käydään läpi työn taustaa ja vaatimuksia, jossa kerrotaan idean alkuperästä ja tarpeesta. Sen jälkeen esitellään työssä käytettäviä teknologioita ja palveluita ja lopuksi vielä kerrotaan itse työn toteutuksesta.

2 React-hookit ja funktionaaliset komponentit

React-sovellukset koostuvat useista komponenteista, jotka asetetaan puumaiseen rakenteeseen. Komponentit mahdollistavat käyttöliittymän pilkkomisen itsenäisiin, uudelleenkäytettäviin palasiin. Komponentteja voi hyvin ajatella JavaScript-funktioina. Komponentit ottavat parametriksi mielivaltaisia syötteitä, joita kutsutaan ominaisuuksiksi (”properties”, lyhyemmin ”props”), ja palauttavat React-elementin, joka muunnetaan HTML-koodiksi.

Komponentit ovat React-ohjelmoinnin perusta ja yksinkertainen komponentti voi olla esimerkiksi pelkkä painike. Komponentit luodaan JSX-syntaksilla, joka on JavaScriptin syntaktinen laajennus. Reactin komponentteja voi kirjoittaa kahdella eri tavalla: luokkakomponentilla tai funktionaalisella komponentilla.

2.1 Funktionaaliset ja luokkakomponentit

Yksinkertaisin tapa tehdä komponentti on koodata JavaScript-funktio. Esimerkkikoodissa 1 on yksinkertainen komponentti, joka tervehtii käyttäjää.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Esimerkkikoodi 1. Yksinkertainen komponentti, joka tervehtii käyttäjää.

Tämä JavaScript-funktio on toimiva komponentti, koska se ottaa parametriksi ominaisuus-olion ja palauttaa React-elementin. Tällaisia komponentteja kutsutaan funktionaalisiksi komponenteiksi, koska ne ovat kirjaimellisesti vain JavaScript-funktioita.

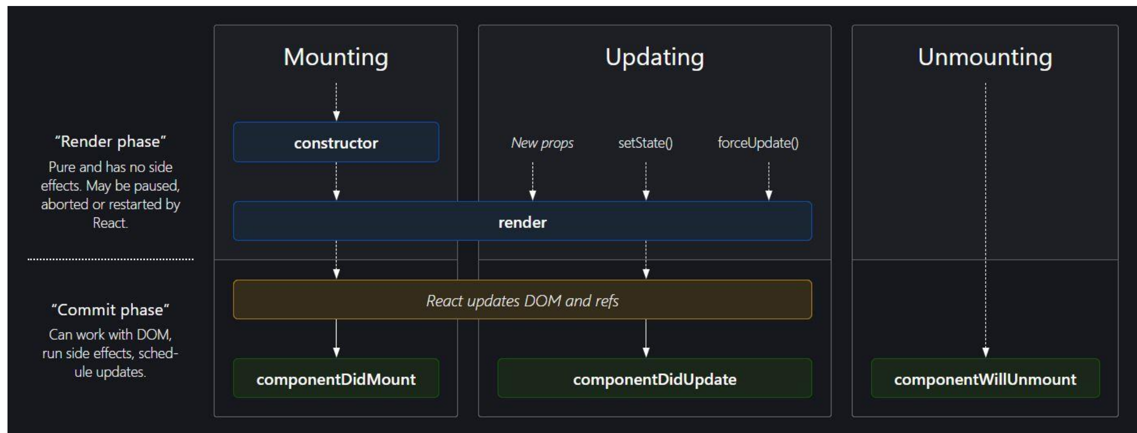
Täysin toiminnallisuuksiltaan samanlaisen komponentin voi myös tehdä käyttäen JavaScriptin luokkia. Esimerkkikoodissa 2 on käyttäjää tervehtivä komponentti, joka on tehty käyttäen luokkaa.

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Esimerkkikoodi 2. Käyttäjää tervehtivä komponentti, toteutettu JavaScript-luokalla.

Reactin näkökulmasta nämä molemmat komponentit ovat samanlaiset. Luokkakomponentilla on lisäksi muita ominaisuuksia, jotka eivät ole sisäänrakennettuna funktionaalisissa komponenteissa. Näitä ominaisuuksia pystyy kuitenkin käyttämään React-hookkien avulla funktionaalisissa komponenteissa, jotka esitellään seuraavassa luvussa. Näistä ominaisuuksista hyvänä esimerkkinä toimivat tila ja elinkaarimetodit. Reactissa tila on JavaScript olio, jossa voi säilyttää avain-arvo-pareja. React-komponenteilla on elinkaari, joka koostuu kolmesta eri vaiheesta: komponentti voi olla kiinnittymässä

(mounting), päivittymässä (updating) tai irtautumassa (unmounting). Jokaisessa elinkaaren vaiheessa komponentti voi suorittaa koodaajan ylikirjoittamia elinkaarimetodien toteutuksia. Kuvassa 1 on kaavio React-komponentin elinkaaresta.



Kuva 1. React-komponentin elinkaari [10].

Harmaissa laatikoissa on eriteltynä aiemmin mainitut elinkaaren vaiheet. Jokaisesta elinkaaren vaiheesta löytyy vihreä laatikko, joka edustaa elinkaarimetodia. Elinkaaren kiinnitysvaiheessa suoritetaan componentDidMount-funktio, päivitysvaiheessa suoritetaan componentDidUpdate-funktio ja irtautumisvaiheessa suoritetaan componentWillUnmount-funktio.

2.2 React-hookit

Hookit ovat ominaisuus, joka lisättiin Reactiin versiossa 16.8. Ne mahdollistavat tilan ja muiden Reactin ominaisuuksien käytön ilman luokkakomponenttia. Hookkeja ei ole pakko käyttää, ja funktionaaliset komponentit toimivat ilman niitä ihan normaalisti. Hookeissa käytetään myös tuttuja Reactin konsepteja: ominaisuudet, tila ja elinkaari, Reactia ei siis tarvitse ”opetella uudestaan”. Ne eivät myöskään ole niin uusi ja rikkova ominaisuus, että niiden käyttöönotto hajottaisi vanhoja sovelluksia. Hookit onkin kehitetty uudelleenkäytettävyyttä miettien. Ennen hookeja ei ollut mitään tapaa kiinnittää tilallista logiikkaa komponenttiin. Hookit mahdollistavat tilallisen logiikan testauksen ja uudelleenkäytön helposti toisessa komponentissa.

Hookeja voi käyttää vain funktionaalisissa komponenteissa. Luokkakomponentit eivät kuitenkaan ole poistumassa Reactista, ja Reactin kehittäjät tukevat luokkakomponentteja vielä lähitulevaisuudessa [3].

Reactin kehittäjät ovat asettaneet hookien käytölle kaksi tärkeää sääntöä:

1. Älä kutsu hookkeja silmukoissa, ehdollisesti tai sisäkkäisissä funktiossa.

2. Älä kutsu hookkeja tavallisissa JavaScript-funktioissa.

Sääntöjen noudattamisen helpottamiseksi, React kehittäjät tekivät ESLint-säännön, joka huomauttaa ohjelmoijaa, jos hookeja käyttää väärin. ESLint on staattisen analyysin tarkistustyökalu, jonka avulla voidaan tarkistaa, että koodista ei löydy syntaksivirheitä ja että koodi täyttää tietyt ohjelmointisäännöt.

React-hookit alkavat aina ”use” -sanalla. Tämän ansiosta ESLint-sääntö pystyy tarkistamaan, että ohjelmoija käyttää hookeja oikein. Myös omien hookien luominen on mahdollista, ja Reactin kehittäjät kehottavatkin käyttämään ”use” -sanaa niiden nimeämisessä, että ESLint-sääntö osaa ottaa ne huomioon [4].

2.3 State-hook

State-hookilla voidaan ottaa käyttöön luokkakomponentista tuttu tila. Reactissa tila on yksinkertaisesti olio, jossa voidaan säilyttää avain-arvo -pareja. Tila mahdollistaa komponenttien dynaamisuuden ja vuorovaikutteisuuden. Esimerkkikoodissa 3 on yksinkertainen esimerkki, jossa tilaa käytetään painikkeen napsautusten määrän muistamiseen. Esimerkki on tehty funktionaalisella komponentilla State-hookkia käyttämällä.

```
const FLaskuri = () => {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You've clicked {count} times.</p>
      <button
        onClick={() => {
          setCount(count + 1);
        }}
      >
        Increment
```

```
      </button>
    </div>
  );
};
```

Esimerkkikoodi 3. Reactin funktionaalinen komponentti, joka näyttää, kuinka monta kertaa painiketta on napsautettu.

Komponentissa on käytössä useState-hookki. Hook palauttaa kaksi arvoa: tämän hetkisen tilan arvon (count) ja funktion, jolla arvoa voi päivittää (setCount). Hook ottaa myös vastaan yhden parametrin, jonka avulla voimme alustaa count-muuttujan arvon. Sama esimerkki toteutettaisiin esimerkkikoodin 4 mukaisesti luokkakomponenttia käyttäen.

```
class CLaskuri extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }

  render() {
    return (
      <div>
        <p>You've clicked {this.state.count} times.</p>
        <button onClick={() => this.setState({
                count: this.state.count + 1 })}>
          Increment
        </button>
      </div>
    );
  }
}
```

Esimerkkikoodi 4. Reactin luokkakomponentti, joka näyttää, kuinka monta kertaa painiketta on napsautettu.

Koska kyseessä on luokka, ”this” -termillä voidaan viitata itse komponenttiin. Luokkakomponentissa count-muuttuja on osa tila-oliota, ja saamme sen arvon this.state.count -ilmauksella. Arvoa voi myös päivittää setState-funktiolla, joka ottaa parametriksi olion, johon ohjelmoijan on määritettävä, mitä arvoja tilasta päivitetään. Esimerkkejä katsellessa voi huomata myös, kuinka paljon vähemmän koodia tulee funktionaalisella komponentilla.

2.4 Effect-hook

Effect-hookilla voi lisätä komponentteihin sivuvaikutuksia. Tietojen hakeminen, tilauksen asettaminen ja DOM-rakenteen manuaalinen muokkaus ovat kaikki yleisiä sivuvaikutuksia. Effect-hookkia voi ajatella componentDidMount-, componentDidUpdate- ja componentWillUnmount-elinkaarimetodien yhdistelmänä. Sivuvaikutukset voi myös karkeasti jakaa kahteen kategoriaan: sellaiset minkä jälkeen täytyy siivota ja sellaiset mitkä eivät vaadi siivoamista. Siivoaminen tarkoittaa esimerkiksi asetetun tilauksen poistoa. Siivoaminen estää muistivuotojen syntymisen.

Esimerkkikoodissa 5 on ensimmäisenä sivuvaikutus, jonka jälkeen ei tarvitse siivota. Se on yksinkertainen toteutus, jossa päivitetään dokumentin otsikko näyttämään painikkeen napsautuksien määrä. Koodi on tehty funktionaalisella komponentilla, hyödyntäen effect-hookkia.

```
const FTitle = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You've clicked ${count} times`;
  });

  return (
    <div>
      <p>You've clicked {count} times.</p>
      <button
        onClick={() => {
          setCount(count + 1);
        }}
      >
        Increment
      </button>
    </div>
  );
};
```

Esimerkkikoodi 5. Otsikon päivittäminen effect-hookin avulla. Funktionaalinen komponentti.

Effect-hook ottaa parametriksi funktion, joka suoritetaan. Oletusarvoisesti effect-hook ajetaan, kun komponentti piirretään näkyviin ensimmäistä kertaa, ja jokaisella päivityksellä sen jälkeen. Myöhemmin luvussa on esimerkki, miten effect-hookin ajoa voi rajoittaa riippuvuuksilla. Sama toteutus voidaan nähdä esimerkkikoodi 6:sta. Tällä kertaa käytössä on luokkakomponentti ja kaksi elinkaarimetodia.

```
class CTitle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }

  componentDidMount() {
    document.title = `You've clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You've clicked ${this.state.count} times`;
  }

  render() {
    return (
      <div>
        <p>You've clicked {this.state.count} times.</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Increment
         </button>
      </div>
    );
  }
}
```

Esimerkkikoodi 6. Otsikon päivittäminen elinkaarimetodien avulla. Luokkakomponentti.

Näitä kahta elinkaarimetodia ei kuitenkaan voi yhdistää, vaikka ne tekevät täysin saman asian. ComponentDidMount ajetaan vain yhden kerran, kun komponentti piirretään näkyviin ensimmäistä kertaa. ComponentDidUpdate ajetaan vasta, kun esimerkiksi painiketta napsautetaan ensimmäisen kerran.

Esimerkkikoodissa 7 on sellainen sivuvaikutus, joka vaatii siivoamista, kun komponentti irrotetaan. Se on kuvitteellinen toteutus viestintäsovelluksesta, jossa asetetaan tilaus ystävän statuksen muuttumisen kuunteluun. Siivoaminen effect-hookilla tapahtuu yksinkertaisesti: palautetaan funktio hookin sisällä, joka tekee siivoamisen.

```
const ChatStatus = (props) => {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // clean up
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleSta-
tusChange);
    };
  });

  if (isOnline === null) {
```

```
    return "Loading...";
  }
  return isOnline ? "Online" : "Offline";
};
```

Esimerkkikoodi 7. Kuvitteellinen viestintäsovellus, jossa effect-hookissa siivotaan, kun komponentti irrotetaan.

Siivoamisfunktio ajetaan jokaisella päivityksellä vasta componentWillUnmount-elinkaaren kohdalla, jotta emme ensin aseta tilausta ja sitten poista sitä välittömästi. Tästä voimme päätellä, että jos sivuvaikutus on raskas funktio, voi siitä seurata haittaa suorituskyvylle. Myöhemmin luvussa on esimerkki, kuinka effect-hookkien ajoa voi rajoittaa.

Esimerkkikoodissa 8 on nähtävissä sama toteutus luokkakomponenttia ja kahta elinkaarimetodia hyödyntäen. Siivoaminen tapahtuu elinkaarimetodilla componentWillUnmount.

```
class CChatStatus extends React.Component {
    constructor(props) {
        super(props);
        this.state = { isOnline: null };
        this.handleStatusChange = this.handleStatusChange.bind(this);
    }

    componentDidMount() {
        ChatAPI.subscribeToFriendStatus(
            this.props.friend.id,
            this.handleStatusChange
        );
    }

    componentWillUnmount() {
        ChatAPI.unsubscribeFromFriendStatus(
            this.props.friend.id,
            this.handleStatusChange
        );
    }

    handleStatusChange(status) {
        this.setState({
            isOnline: status.isOnline;
        });
    }

    render() {
        if (this.state.isOnline === null) {
            return "Loading...";
        }
        return this.state.isOnline ? "Online" : "Offline";
    }
}
```

Esimerkkikoodi 8. Kuvitteellinen viestintäsovellus, jossa siivoaminen tapahtuu elinakaarimetodilla.

Hookin sijasta käytössä on tila ja kaksi elinkaarimetodia. handleStatusChange-funktio täytyy kiinnittää komponentin instanssiin bind-funktion avulla, että se pystyisi lukemaan ja päivittämään komponentin arvoja.

Jos tekisimme jotain raskasta effect-hookissamme, sovelluksen suorituskyky voi kärsiä. Effect-hookille voi asettaa riippuvuuksia, jotka ovat effect-hookiin määriteltäviä muuttujia. Riippuvuudet voivat estää effect-hookin ajon jokaisella päivityksellä. Riippuvuuksia voisi myös verrata componentDidUpdate-elinkaarimetodiin, jossa voidaan vertailla esimerkiksi edellistä tilaa nykyiseen tilaan. Esimerkkikoodissa 9 on tuttuun otsikon päivitys -komponenttiin lisätty riippuvuus effect-hookkiin, jonka avulla voisimme estää sen ajon, jos count-muuttuja ei olisi muuttunut.

```
const FTitle = () => {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You've clicked ${count} times`;
    // only run if count changes
  }, [count]);

  return (
    <div>
      <p>You've clicked {count} times.</p>
      <button
        onClick={() => {
          setCount(count + 1);
        }}
      >
        Increment
      </button>
    </div>
  );
};
```

Esimerkkikoodi 9. Otsikon päivitys -komponentti, mutta effect-hookiin on lisätty riippuvuus.

Riippuvuudet määritellään taulukkoon Effect-hookin parametriksi. Tässä toteutuksessa riippuvuus eli count-muuttuja on yksin taulukossa. Riippuvuuksia voi olla enemmän kuin yksi.

Viimeiseksi, effect-hookkia voi käyttää myös kuin componentDidMount-elinkaarimetodia. Toteutus on lähes sama kuin esimerkkikoodi 9:ssä, mutta riippuvuustaulukko määritetään vain tyhjäksi. Tyhjän riippuvuustaulukon kanssa kannattaa olla varovainen. Jos esimerkiksi hakisimme jostain rajapinnasta dataa, on tila saattanut taustalla muuttua jo sillä

välin. Jos käyttäisimme tilan arvoa effect-hookissa tällaisessa tilanteessa, arvo tulisi olemaan väärä. Tästä syystä riippuvuustaulukko on nimenomaan riippuvuuksia varten.

2.5 Omat hookit

Hookeja voi myös kehittää itse. Omien hookien tarkoitus on yleensä poimia komponenttilogiikkaa ja tehdä niistä uudelleenkäytettäviä funktioita. Tavallisesti komponenttilogiikan jakaminen tapahtui esimerkiksi komponentille annettavien ominaisuuksien kautta. Hookeja käyttämällä voimme ratkaista saman ongelman ilman, että meidän pitää lisätä komponenttipuuhun lisää komponentteja.

Reactin kehittäjät ovat laatineet hookeille sääntöjä, joita tulisi noudattaa niitä käytettäessä ja luotaessa. Sääntöjen noudattamisen helpottamiseksi kehittäjät tekivät ESLint-säännön, joka varmistaa, että kehittäjä noudattavat hookkien sääntöjä [5].

Omat hookit voivat tehdä kutsuja toisiin hookkeihin. Esimerkkikoodissa 10 olemme poimineet aiemmassa luvussa käytetystä viestintäsovelluksesta tilalogiikan, ja siirtäneet sen omaan useFriendStatus-hookkiimme. Esimerkkikoodissa 11 olemme uudelleenkirjoittaneet kuvitteellisen viestintä sovelluksen käyttämään useFriendStatus-hookkia.

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

Esimerkkikoodi 10. Viestintäsovelluksen logiikka siirrettynä mukautettuun hookkiin.

```
const FChatStatus = (props) => {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return "Loading...";
  }
  return isOnline ? "Online" : "Offline";
};
```

Esimerkkikoodi 11. Viestintäsovellus uudelleenkirjoitettuna, käyttäen useFriendStatus-hookkia.

Kuten aiemmin mainittiin, hookien tulisi aina alkaa ”use” -sanalla, joten annoin hookin nimeksi useFriendStatus. Hookimme koostuu kahdesta kahdesta Reactin omasta hookista. useState-hookki pitää online-tilan muistissa ja useEffect-hooki asettaa tai poistaa tilauksen kaverin statuksen kuunteluun.

3 Kehitysympäristö

Verkkosovelluksia voi kehittää monessa eri ympäristössä. Valitsin tähän insinöörityöhön alustakseni Windows 10 -käyttöjärjestelmän. Ohjelmointia varten olen valinnut suositun Visual Studio Code -editorin, joka on tarkoitettu debuggausta, versionhallintaa ja koodin editointia varten [6]. Visual Studio Code -editoriin on olemassa myös loistavia lisäosia, jotka helpottavat esimerkiksi koodin jäsennystä ja automaattista täydennystä. Editoriin on myös integroitu komentotulkki. Windows 10:n versiossa komentotulkki on powershell.

Yksi mainitsemisen arvoinen lisäosa Visual Studio Code -editoriin on Prettier. Prettier mahdollistaa koodin yhtenäisen ulkonäön muotoilemalla koodin uudestaan käskystä, tai kun tallennetaan tiedosto, jos se on konfiguroinut sen tekemään niin. Versionhallintaan voidaan myös lisätä Prettierin konfigurointitiedosto, jolloin kaikki projektiin osallistuvat tuottavat yhtenäistä koodia, jos he vain käyttävät lisäosaa.

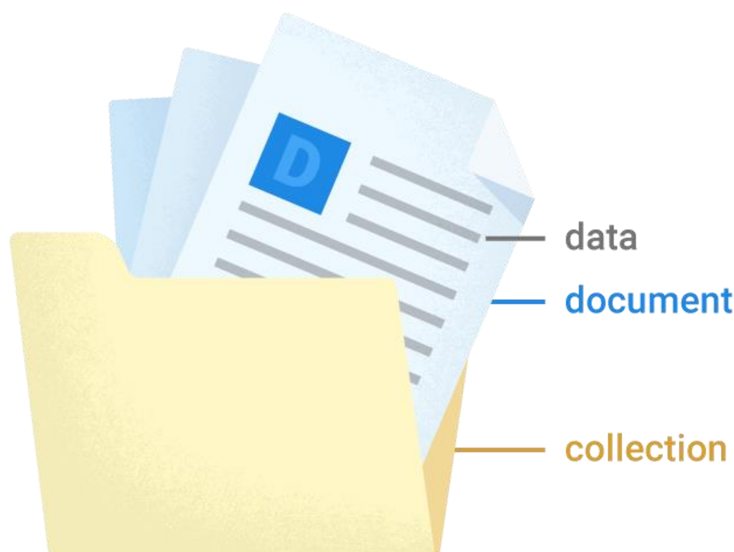
Sovelluksen pohja on luotu käyttäen Create React App -sovellusta. Create React App on Facebookin luoma sovellus, jolla React-projektin voi aloittaa vaivattomasti komentoriviltä antamalla sille vain nimen. Jotta sitä voidaan käyttää, Node.js täytyy asentaa kehitysympäristöön. Node.js voi ladata sen verkkosivuilta https://nodejs.org/en/download/. Omassa käytössä oli versio 14.5.0, uusin versio tällä hetkellä on 14.15.4.

4 Palvelinpuoli ja tietokanta

Sovelluksen palvelinpuolella käytössä on Googlen Firebase-pilvipalvelu. Firebase on alusta mobiili- ja web-sovelluksille. Firebasen avulla kehittäjien ei tarvitse käyttää aikaa back-end-kehitykseen, vaan tarjolla on valmiita paketteja, joista valita. Tähän insinöörityöhön Firebasen palveluista on otettu käyttöön Cloud Firestore, Authentication, Storage ja Cloud Functions.

4.1 Cloud Firestore -palvelu

Cloud Firestore on pilvessä toimiva NoSQL-tietokanta, johon sovellukset pääsevät käsiksi natiiveilla SDK:lla. NoSQL-tietokanta eroaa relaatiotietokannasta siten, että tietojenkäsittely perustuu olioihin. Cloud Firestoren tietomallissa data tallennetaan dokumentteihin, jotka koostuvat avain-arvo-pareista. Dokumentit tallennetaan kokoelmiin, joita voi käyttää datan organisoitiin ja kyselyjen rakentamiseen. Kuvassa 2 havainnollistetaan tietokannan rakennetta.



Kuva 2. Cloud Firestore -tietokannan rakenne [7].

Kokoelmia voi siis ajatella esimerkiksi Windows-käyttöjärjestelmän tapaan kansioina. Kansioihin voi tallentaa tiedostoja, jotka tunnettiin dokumentteina Cloud Firestoressa.

Dokumentit tukevat montaa eri datatyyppiä: normaalista merkkijonoista ja numeroista aina komplekseihin ja sisäkkäisiin objekteihin. Dokumentteihin voi myös tallentaa alikokoelmia. Alikokoelma on tavallinen kokoelma, joka voi puolestaan taas sisältää monta dokumenttia. Kuvan 3 mukaisesti kuvitteellisessa viestintäohjelmassa voitaisiin säilyttää keskustelun viestit alikokoelmassa keskusteluhuoneen alla.

Kuva 3. Alikokoelma viestintäohjelmassa, jossa keskustelun viestit säilytettäisiin keskusteluhuoneen alla [8].

Rooms edustaa kuvassa kokoelmaa. RoomA ja roomB ovat dokumentteja, joissa on messages-alikokoelma, jotka sisältävät viestidokumentteja.

Dokumenttien kysely on joustavaa. Halutessaan voi hakea kokonaisia kokoelmia, tai vain yksittäisiä dokumentteja määrittelemien parametrien mukaan. Kyselyihin voi myös liittää halutun tuloksien lajittelu tyylin, sekä yhden tai monta peräkkäistä suodattavaa käskyä. Kokoelmat on myös indeksoitu automaattisesti, jolloin hakujen suorituskyky kasvaa vain tulosjoukon koon mukaisesti, eikä tallennetun datan mukaisesti [7].

Yksittäisen dokumentin kysely muodostetaan kahdesta osasta: kokoelman tunnuksesta ja dokumentin tunnuksesta. Esimerkkikoodissa 12 haetaan kaupunki dokumenttia sen tunnuksella kokoelmasta, ja dokumenttiin liittyvä data.

```
const cityRef = db.collection('cities').doc('HEL');
const doc = await cityRef.get();
// check if document actually exists
if (!doc.exists) {
  console.log('No such document!');
} else {
  console.log('Document data:', doc.data());
}
```

Esimerkkikoodi 12. Kaupungin hakeminen kokoelmasta. JavaScript.

Koodiesimerkissä ensimmäisellä rivillä luodaan viite kaupunki-kokoelman sisällä olevalle ”HEL”-dokumentille viite. Toisella rivillä dokumentti haetaan tietokannasta, jonka jälkeen sitä käsitellään. Tässä toteutuksessa, jos dokumentti on olemassa, sen tiedot tulostetaan konsoliin.

Koko kokoelman hakeminen toimii samalla tavalla. Kysely muodostuu vain kokoelman tunnuksesta. Esimerkkikoodissa 13 haetaan kaikki dokumentit kaupunki-kokoelman sisältä ja tulostetaan dokumenttien tiedot.

```
const citiesRef = db.collection('cities');
const snapshot = await citiesRef.get();
// loop through results & print
snapshot.forEach(doc => {
  console.log(doc.id, '=>', doc.data());
});
```

Esimerkkikoodi 13. Kaupunki kokoelman hakeminen. JavaScript.

Esimerkissä ensimmäisellä rivillä kaupunki-kokoelmaan luodaan viite. Toisella rivillä kokoelman kaikki dokumentit haetaan tietokannasta, jonka jälkeen dokumenttien tiedot tulostetaan silmukan avulla konsoliin.

Usean dokumentin hakeminen kokoelmasta onnistuu myös helposti. Kysely muodostuu jälleen kerran kokoelman tunnuksesta, jonka jälkeen voidaan asettaa parametri, minkä haluamme täyttyvän. Esimerkkikoodissa 14 haetaan kaupunki-kokoelmasta vain ne kaupungit, jotka ovat pääkaupunkeja.

```
const citiesRef = db.collection('cities');
const snapshot = await citiesRef.where('capital', '==', true).get();
if (snapshot.empty) {
  console.log('No matching documents.');
  return;
}

snapshot.forEach(doc => {
  console.log(doc.id, '=>', doc.data());
});
```

Esimerkkikoodi 14. Pääkaupunkien haku kaupunki-kokoelmasta. JavaScript.

Ensimmäisellä rivillä luodaan viite kaupunki-kokoelmaan. Toisella rivillä viitteelle asetetaan ehto, jonka perusteella kaikki ehdon toteuttavat dokumentit noudetaan tietokannasta. Ehtona tässä toteutuksessa on, että kaupunki-dokumentilla on oltava capital-kenttä, jonka arvo on true.

Kokoelmiin ja dokumentteihin voi myös liittää kuuntelijoita, jotka ilmoittavat, kun data vaihtuu niiden sisällä. Kuuntelijan voi liittää dokumenttiin tai kokoelmaan samalla tavalla, kun niitä haettaisiin vain kerran. Ensin määritellään referenssi kokoelmaan tai dokumenttiin, ja sitten get()-funktion sijasta käytetään onSnapshot()-funktiota. Esimerkkikoodissa 15 asetetaan kuuntelija kaupunki-kokoelmaan, joka antaa tiedon, kun uusi kaupunki lisättäisiin kokoelmaan. Kun kuuntelija asetetaan kokoelmaan, kaikki dokumentit sen sisältä palautuvat heti, ja sen jälkeen yksittäiset dokumentit, jos niitä lisätään. Sama pätee myös yksittäiseen dokumenttiin: kun kuuntelija asetetaan, palautuu kaikki dokumenttiin liittyvä data heti, ja sen jälkeen uudestaan, kun muutos tapahtuu dokumentissa [9].

```
const col = db.collection("cities");
const observer = col.onSnapshot((snapshot) => {
  if (snapshot) {
      snapshot.forEach((doc) => {
          console.log(doc.id, "=>", doc.data());
      });
  }
});
```

Esimerkkikoodi 15. Kaupunki-kokoelmaan asetettu kuuntelija, joka tulostaa, kun uusi kaupunki lisätään kokoelmaan.

Esimerkissä ensimmäisellä rivillä luodaan taas viite kaupunki-kokoelmaan. Seuraavalla rivillä viitteeseen liitetään kuuntelija. Heti kuuntelijan asetettua kaikki dokumentit palautuvat kokoelmasta, ja niiden tiedot tulostetaan konsoliin. Kun uusi dokumentti lisättäisiin kokoelmaan, sen tiedot tulostettaisiin konsoliin reaaliajassa.

4.2 Firebase Authentication -palvelu

Firebasen Authentication -palvelu vastaa käyttäjän todennuksesta. Authenticationilla kehittäjät voivat valita useita eri vaihtoehtoja kirjautumiselle. Tässä projektissa käytössä on perinteinen sähköposti ja salasana sekä anonyymi kirjautuminen. Anonyymillä kirjautumisella mahdollistimme tilauksen tilan katsomisen helposti ilman oikeata kirjautumista.

Käyttämällä Firebasen Authentication-palvelua, voimme myös käyttää Firebasen tarjoamaa pääsyoikeusjärjestelmää. Tällä pääsyoikeusjärjestelmällä on helppo luoda oikeuksia esimerkiksi tietokannan lukemiseen. Esimerkkikoodissa 16 on esimerkki siitä, millä tavalla oikeuksia rakennetaan.

```
service cloud.firestore {
  match /databases/{database}/documents {
    match /<some_path>/ {
      allow read, write: if <some_condition>;
    }
  }
}
```

Esimerkkikoodi 16. Pääsyoikeussäännön havainnollistaminen Cloud Firestoreen.

Sääntöön määritellään, mihin palveluun, tietokantaan, kokoelmaan ja/tai dokumenttiin sääntö pätee. Tietokannan voi määritellä villikortiksi ympäröimällä sille itse keksimän nimen aaltosulkuihin, jolloin se pätee kaikkiin tietokantoihin.

Käyttäen aikaisempaa kaupunki-kokoelmaa esimerkkinä, voimme rakentaa yksinkertaisen säännön dokumenttien lukemisen ja kirjoittamisen sallimiseen. Esimerkkikoodissa 17 on pääsyoikeussääntö kaupunki-kokoelmaan.

```
service cloud.firestore {
  match /databases/{database}/documents {

    // Match any document in the 'cities' collection
    match /cities/{city} {
      // allow anyone to read
      allow read: if true;
      // require authentication for writing
      allow write: if request.auth != null;
    }
  }
}
```

Esimerkkikoodi 17. Kaupunki-kokoelman pääsyoikeussäännöt lukemiselle ja kirjoittamiselle.

Sääntö sallii kaupunkien lukemisen ilman kirjautumista, mutta kaupungin lisäämiseen on pakko kirjautua sisään.

Toinen helppo ja hyödyllinen sääntö olisi varmistaa, että käyttäjät voivat päivittää ja lukea vain omaa dataansa. Esimerkkikoodissa 18 kuvataan käyttäjät-kokoelman sääntöä, johon luotaisiin käyttäjälle oma dokumentti rekisteröinnin ohella.

```
service cloud.firestore {
  match /databases/{database}/documents {
    // Make sure the uid of the requesting user matches name of the user
    // document. The wildcard expression {userId} makes the userId variable
    // available in rules.
    match /users/{userId} {
      allow read, update, delete: if request.auth != null && request.auth.uid
== userId;
      allow create: if request.auth != null;
    }
  }
}
```

Esimerkkikoodi 18. Käyttäjä-kokoelman pääsyoikeussäännöt lukemiseen, päivittämiseen, poistoon ja luontiin.

Dokumentin voisi päivittää, lukea tai poistaa vain, jos käyttäjän tunnus on sama kuin dokumenttiin tallennettu käyttäjän tunnus.

Sääntöjä pystyy luomaan myös datan validointia varten. Kaupunki-esimerkkiä käyttäen, esimerkkikoodissa 19 on sääntö, jolla käyttäjän syötettä voidaan validoida.

```
service cloud.firestore {
  match /databases/{database}/documents {
    // Make sure all cities have a positive population and
    // the name is not changed
    match /cities/{city} {
      allow update: if request.resource.data.population > 0
                    && request.resource.data.name == resource.data.name;
    }
  }
}
```

Esimerkkikoodi 19. Kaupungin päivittämiseen luotu sääntö, joka validoi lähetettyä syötettä.

Säännön mukaan kaupunkia voi päivittää vain, jos lähetetyssä dokumentissa väestön määrä on yli 0, eikä kaupungin nimi ole muuttunut.

Säännöt voivat joskus tulla niin monimutkaiseksi, että niiden luettavuus kärsii. Sääntöihin voi tästä syystä määritellä omia funktiota, jotka voivat parantaa luettavuutta huomattavasti. Esimerkkikoodissa 20 on esimerkki tähän insinöörityöhön liittyvästä tietokantasäännöstä, jossa on käytetty omia funktioita luettavuuden parantamiseksi.

```
service cloud.firestore {
  match /databases/{database}/documents {

    // allow updating if: you are the reporter or nobody has claimed it
    // (you're probably claiming it when updating) or you are the claimer
    function sameUserOrClaimed() {
      return request.auth.uid == resource.data.reporter
      || request.auth.uid == resource.data.claimedBy || re-
source.data.claimedBy == "" || reviewSubmit(request, resource);
    }

    // also allow if you're submitting a review as anonymous
    function reviewSubmit(request, resource) {
     return request.resource.data.review != null && resource.data.review ==
null;
    }

    // admin users can request data
    function isAdmin() {
     return request.auth != null && get(/databases/$(database)/documents/us-
ers/$(request.auth.uid)).data.isAdmin == true;
    }

    match /transactions/{transaction} {
      // to create, read or delete a transaction, you have to be authenticated
      allow read, delete: if request.auth != null;
      // to update a transaction, you have to be the "owner" or "claimer" or
"admin"
      allow update: if sameUserOrClaimed() || isAdmin();
      allow create: if request.auth != null;
    }
  }
}
```

Esimerkkikoodi 20. Insinöörityöhön käytetty pääsyoikeussääntö, jossa hyödynnettiin funktiota.

Säännön tarkoitus on antaa pääkäyttäjälle täydet oikeudet transaction-nimisiin dokumentteihin. Toiseksi sääntö mahdollistaa transaction-dokumentin päivityksen niille käyttäjille, jotka sen on luonut tai ottanut itselleen käsittelyyn. Kolmanneksi sääntö mahdollistaa arvostelukentän päivittämisen, jos olet kirjautunut sovellukseen anonyymisti.

4.3 Firebase Storage -palvelu

Firebase Storage -palvelu mahdollistaa tiedostojen tallentamisen ja lataamisen Googlen pilvessä. Käytimme Firebasen Storage -palvelua ilmoituksiin liitettävien kuvien säilyttämiseen. Yhteen ilmoitukseen voi tulla useita kuvia, ja kuville voi luoda viitteen suoraan kyseisen ilmoituksen dokumenttiin. Tila Firebase Storagessa ei kuitenkaan ole ilmaista, joten prosessoimme kuvat ennen niiden lähettämistä. Storageen voi myös luoda samanlaisia pääsyoikeussääntöjä kuin kokoelmiin.

4.4 Firebase Cloud Functions -palvelu

Firebasen Cloud Functions -palvelun avulla pystymme suorittamaan palvelinpuolen koodia ilman omaa palvelinta. Palvelu on kehys, johon luodaan JavaScript-funktiota, jotka voidaan suorittaa esimerkiksi, kun uusi dokumentti luodaan tiettyyn kokoelmaan. Palvelun avulla meidän ei tarvitse huolehtia omien palvelimien skaalauksesta tai ylläpidosta.

Käytimme Cloud Functions -palvelua uusien käyttäjien dokumenttien luomiseen, sekä automaattisten sähköpostien lähettämiseen. Uuden käyttäjätilin voi luoda web-sovelluksen kautta, joka kutsuu käyttäjätilin luontiin tarkoitettua pilvi-funktiota. Uuden käyttäjän luonti luo käyttäjälle automaattisesti dokumentin käyttäjät-kokoelmaan, sekä lähettää käyttäjätilin haltian sähköpostiin tiedot käyttäjätilistä.

Cloud Functions pystyy myös liittämään kuuntelijoita kokoelmiin. Yksi projektiin toteuttamamme funktio kuuntelee tilaus dokumenttien muutoksia. Jos funktio havaitsee, että tilaukseen on liitetty tarjousehdotus, lähetämme tilauksen tekijälle sähköpostiin tiedon tarjouksen saamisesta.

5 Käyttöliittymä

Sovelluksen käyttöliittymä toteutetaan ReactJS-kirjaston, Material-UI-kirjaston sekä Bootstrap CSS -sovelluskehyksen avulla. Material-UI on Googlen kehittämä Material Design -muotokieleen perustuva React-kirjasto. Bootstrap on alun perin Twitterin kehittämä CSS-sovelluskehys, joka koostuu valmiista CSS-säännöistä ja JavaScript-tiedostoista.

Valitsin Reactin ja Bootstrapin, koska minulla oli niistä jo aikaisempaa kokemusta toisista projekteista. Valitsin Material-UI-kirjaston projektiin, koska hakujen perusteella sen tarjoama taulukomponentti olisi juuri sopiva tähän työhön. Tauluun voi helposti määritellä sivut ja kuinka monta riviä on näkyvissä per sivu.

Sovellus koostuu kolmesta eri näkymästä: kirjautuminen, tilaukset ja käyttäjät. Kuvassa 4 on kirjautumisnäkymän toteutus.

Kuva 4. Kuva toteutetusta kirjautumisnäkymästä.

Kirjautumisnäkymässä on yksinkertaisesti vain kaksi kenttää, mihin käyttäjä voi syöttää sähköpostin ja salasanan. Virheen sattuessa sähköpostikentän yläpuolelle tulee punainen teksti, jossa käyttäjää opastetaan mahdollisesti korjaamaan virhe.

Tilaus-näkymä koostuu neljästä osasta. Kuvassa 5 on nähtävissä tilaus-näkymän toteutus. Kuvasta on sensuroitu osa datasta mustin laatikoin.

Tilaukset Käyttäjät

Kirjaudu ulos

Tilaukset

Näytetään: 9 tilausta -

Suodata

Tekstihaku: Kirjoita | Tilaaja: Valitse | Postinumero: 00000 - 99999

Status: Luotu, Tarjous tehty, Hylätty, Hyväksytty - Odottaa työtä, Työn alla, Valmis

Suodata

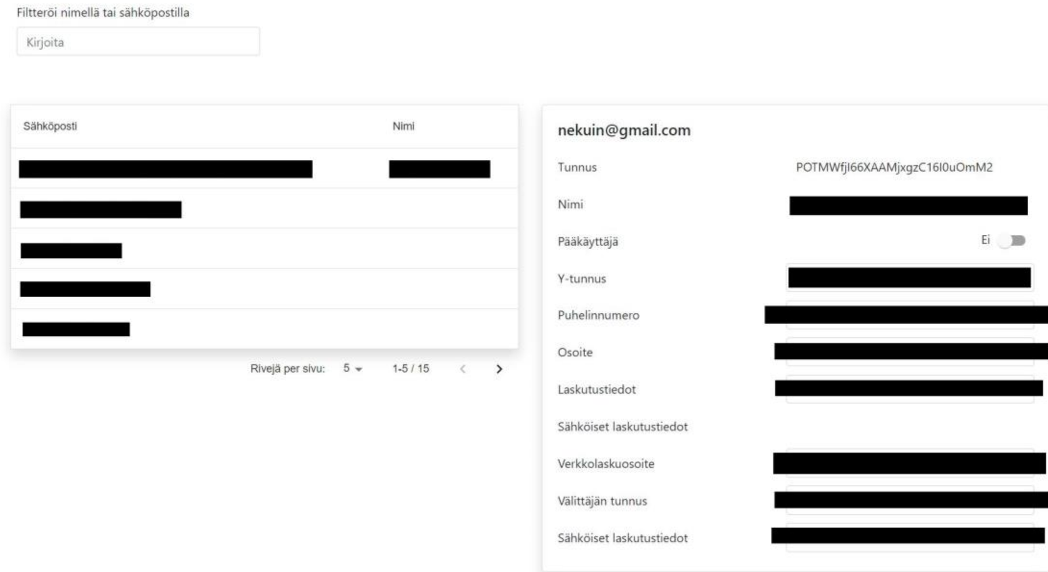
Tunnus	Yritys/Tilaaja	Kohteen nimi	Osoite	Postinumero	Tilausaika	Status	Hinta	Avaa
90100-QsGe				90100	25.11.2020	Valmis		Avaa
90120-skKP				90120	24.11.2020	Valmis		Avaa
90670-5J8m				90670	26.9.2020	Valmis		Avaa
15540-XBsL				15540	2.9.2020	Tarjous tehty		Avaa
90630-Xkar				90630	31.8.2020	Valmis		Avaa
90100-n7Uq				90100	1.10.2020	Luotu		Avaa
90630-b7d2				90630	31.8.2020	Valmis		Avaa
90630-25OH				90630	31.8.2020	Valmis		Avaa
13200-Wnsw				13200	7.10.2020	Työn alla		Avaa

Rivejä per sivu: 10 1-9 / 9

Kuva 5. Tilaus-näkymän toteutus. Osa datasta sensuroitu.

Ylhäällä on navigointivaihtoehdot ja uloskirjautumisen painike. Seuraavaksi on otsikko-osio, jossa näytetään taulussa olevien tilauksen lukumäärä, sekä niiden tarjouksista yhteenlaskettu euromäärä. Otsikon alapuolelta löytyvät suodatusmahdollisuudet. Tilauksia voi suodattaa kohteen nimen, tilaajan, postinumeron sekä tilauksen tilan mukaan. Suodattimen alapuolella näkyy hakutulokset Material-UI:n taulukomponentissa, jossa taulun otsikon mukaan tuloksia voi järjestellä uudelleen. Taulukomponentti mahdollistaa hakutuloksien asettamisen usealle eri sivulle, tulosrivien määrä per sivu on vaihdettavissa käyttäjän toimesta.

Käyttäjät-näkymässä pääkäyttäjä voi päivittää käyttäjätilien tietoja sekä luoda uusia käyttäjiä sovellukseen. Kuvassa 6 on olemassa olevien käyttäjien osuus.



Kuva 6. Käyttäjät-näkymän ylempi osio, jossa näytetään olemassa olevat käyttäjät ja valitun käyttäjän tiedot.

Näkymän yläosassa on navigointivaihtoehdot ja uloskirjautumisen painike. Seuraava osio koostuu suodatinkentästä, jolla olemassa olevia käyttäjiä voi suodattaa nimen tai sähköpostin perusteella. Käyttäjät näytetään Material-UI-taulukomponentissa, jossa oletusarvoisesti näkyy viisi käyttäjää jokaista sivua kohden. Sivukoot ovat käyttäjän vaihdettavissa. Napsauttamalla käyttäjää taulusta sen oikealla puolelle avautuu käyttäjästä tarkemmat tiedot, jotka on tallennettu tietokantaan. Käyttäjän tietoja voi myös päivittää samasta laatikosta. Olemassa olevien käyttäjien alapuolella on lomake, jolla pääkäyttäjä voi luoda uuden käyttäjätilin. Kuvassa 7 on uuden käyttäjän luomiseen tarkoitettu lomake.

Syötä uusi käyttäjä

Sähköposti

matti.meikalainen@gmail.com

Salasana luodaan käyttäjälle automaattisesti

Y-tunnus

1234567-8

Osoite

Haaparannantie 1 C 11

Puhelinnumero

050123123

Laskutustiedot

Laskutustiedot

Sähköiset laskutustiedot

Verkkolaskuosoite

Välittäjän tunnus

Välittäjä

Lähetä

Kuva 7. Käyttäjät-näkymän alempi osa, jossa on uuden käyttäjätilin luomiseen tarkoitettu lomake.

Käyttäjälle voi syöttää asiakkuuteen liittyviä tärkeitä tietoja, jotka pohdittiin yhdessä Asfalttipartion kanssa. Salasana muodostuu uudelle käyttäjälle automaattisesti. Käyttäjätilin haltija saa salasanan tietoonsa sähköpostin kautta.

6 Yhteenveto

Insinöörityön tavoitteena oli luoda websovellus, jolla käyttäjä pystyisi suodattamaan suuria määriä ilmoituksia, jotka muodostuvat kokonaisuuteen kuuluvasta mobiilisovelluksesta. Websovellus luotiin käyttämällä ReactJS-kirjastoa, joka on suunniteltu käyttöliittymien rakentamiseen. Insinöörityössä esiteltiin, kuinka React-sovellukset koostuvat komponenteista, jotka voidaan jakaa kahteen kategoriaan: luokkaan ja funktionaaliseen. Websovellus toteutettiin käyttäen funktionaalisia komponentteja, joihin Reactin kehittäjät ovat keksineet Hook-funktiot. Insinöörityössä käytiin läpi Hook-funktioita ja niiden toteutuksia verrattiin luokkakomponenttien vastaavaan toteutukseen.

Insinöörityön palvelinpuolella käytössä oli helposti käyttöönotettava Googlen Firebase -palvelu. Työssä esiteltiin niitä Firebasen ominaisuuksia, jotka oli otettu websovellusta varten käyttöön.

Insinöörityössä täysin uutena teknologiana minulle oli Material-UI-kirjasto, joka oli Googlen kehittämä Material Design -muotokieleen perustuva React-kirjasto. Omasta mielestäni kaikki Googlen kehittämä on niin monipuolisesti dokumentoitu, että niiden käyttöönotto on suhteellisen helppoa.

Työssä saavutettu lopputulos täyttää sille asetetut vaatimukset ja on omasta sekä asiakkaan mielestä onnistunut. Insinöörityössä luotu websovellus oli jo itsessään jatkokehitystä mobiilisovelluksen rinnalle, eikä tiedossa ole enää lisää jatkokehitysideoita.

Lähteet

1 2020 Developer Survey. 2020. Verkkoaineisto. <https://insights.stackoverflow.com/survey/2020>. Luettu 12.01.2021.

2 React: The Most Popular JavaScript Web Framework. 2020. Verkkoaineisto. < https://medium.com/@kparth2010/react-the-most-popular-javascript-web-framework-3d6c53ad9755>. Luettu 12.01.2021.

3 Introducing Hooks. Verkkoaineisto <https://reactjs.org/docs/hooks-intro.html>. Luettu 19.01.2021.

4 Building Your Own Hooks. Verkkoaineisto. <https://reactjs.org/docs/hooks-custom.html>. Luettu 29.01.2021.

5 Rules of Hooks. Verkkoaineisto. <https://reactjs.org/docs/hooks-rules.html>. Luettu 29.01.2021.

6 Developer Survey Results. 2019. Verkkoaineisto. <https://insights.stackoverflow.com/survey/2019>. Luettu 02.02.2021.

7 Cloud Firestore. 2020. Media. <https://firebase.google.com/docs/firestore>. Luettu 03.02.2021.

8 Cloud Firestore Data model. 2021. Media. <https://firebase.google.com/docs/firestore/data-model>. Luettu 03.02.2021.

9 Get Realtime Updates With Cloud Firestore. 2021. Verkkoaineisto. <https://firebase.google.com/docs/firestore/query-data/listen>. Luettu 03.02.2021.

10 React Lifeycle Methods Diagram. Verkkoaineisto. <https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>. Luettu 03.03.2021.