Tampereen ammattikorkeakoulu

React-sovelluksen optimointi

Antti Peltola

OPINNÄYTETYÖ
Marraskuu 2021

Tietojenkäsittelyn tutkinto-ohjelma
Web-palvelut

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Web-palvelut

PELTOLA, ANTTI
React-sovelluksen optimointi

Opinnäytetyö 27 sivua, joista liitteitä 0 sivua
Marraskuu 2021

Opinnäytetyön tarkoituksena oli tutkia erilaisia tapoja optimoida React-sovellusta ja soveltaa opittua eSend Oy:n olemassa oleviin sovelluksiin. Sovelluksissa pyritään ensisijaisesti optimoimaan suorituskykyä, jotta asiakaskokemus olisi mahdollisimman hyvä.

eSendin sovellukset on tehty ennen React hookien saapumista ja sisältävät paljon legacy-koodia, joten työn tavoitteena olisi päivittää sovelluksia nykyaikaan. Ensisijaisesti päivitettyjen sovellusten tulisi olla suorituskyvyltään tehokkaampia. Optimointiin valikoitui pääsääntöisesti hookit ja Redux. Hookit ovat Reactin uusia ominaisuuksia ja Redux kirjasto, jotka on tuotu juuri optimoimisen helpottamista varten.

Itse sovelluksen optimoimisen tuloksena voitiin huomata, että koodin rivien määrä ei niinkään vähentynyt merkittävästi, mutta loppukäyttäjän kokemus parani huomattavasti. Myös sovelluksen suorituskyvyssä oli havaittavissa kehitystä. Sovelluksen optimointia varten tehtyjä muutoksia otettiin suurissa määrin käyttöön heti ja osaa jatkokehitetään tulevaisuudessa.

Asiasanat: React, Redux, hook, optimointi

ABSTRACT

Tampere University of Applied Sciences
Business Information Systems
Web Services

Antti Peltola
Optimizing a react application

Bachelor's thesis 27 pages, appendices 0 pages
November 2021

The purpose of the bachelor’s thesis was to search different ways to optimise a React application and use the findings to improve existing applications of eSend Oy. Regarding the applications the primary objective was to optimize performance to ensure as good customer experience as possible.

The Applications of eSend have been made before the release of React hooks and contain lots of so-called legacy code, so the object of the work was to update the applications to this day. Primarily the updated applications should be more efficient regarding performance. The reason for using React hooks and Redux is that they are new features and created for optimising React applications.

As the result of the optimizing work, it was possible to notice that the amount of code did not significantly decrease but the end users experience improved noticeably. Also, it was possible to notice progress regarding performance of the application. Plenty of changes made for the application were brought to use right away, while other changes still need some further development.

Key words: React, Redux, hook, optimizing

SISÄLLYS

1 JOHDANTO 6
2 OPTIMOINTI 7
3 REACT HOOKIT 8
3.1 Mitä React hookit ovat 8
3.2 Hookit ja niiden käyttö 8
4 REDUX 11
4.1 Reduxin esittely 11
4.2 Reduxin käyttö 11
5 MUITA OPTIMOINNIN TAPOJA 15
5.1 Throttling 15
5.2 Debouncing 16
5.3 React Fragment 18
5.4 React lazy loading 19
6 OLEMASSA OLEVAN SOVELLUKSEN OPTIMOIMINEN 21
6.1 Työn taustaa 21
6.2 Työn vaiheet 21
6.3 Optimointi 22
6.3.1 React Fragmentien hyödyntäminen 22
6.3.2 Reduxin hyödyntäminen subscriptionilla 22
6.3.3 Throttlingin hyödyntäminen hakukentässä 23
6.3.4 Lazy loading käyttäjäkokemuksen parantamiseksi 24
6.4 Lopputulos 24
7 POHDINTA 26
LÄHTEET 27

ERITYISSANASTO

Debounce	Funktio, joka viivästyttää funktion kutsumista määritetyllä ajalla sen jälkeen, kun sitä on viimeksi kutsuttu.
Div-elementti	Määrittää tietynlaisen sektion, jonka sisälle voidaan laittaa muutakin.
DOM	Document Object Model kuvastaa dokumentin puurakenteena.
JavaScript	Pääasiassa web-ohjelmointiin käytetty ohjelmointikieli.
React	Facebookin kehittämä avoimen lähdekoodin JavaScript-kirjasto käyttöliittymien tekemiseen.
React hook	Hookit mahdollistavat tilanhallinnan ja muita Reactin ominaisuuksia, kirjoittamatta luokkia.
Redux	Avoimen lähdekoodin JavaScript-kirjasto, joka mahdollistaa sovellustilan hallintaa ja keskittämistä.
Renderöinti	Renderöinti tarkoittaa elementtien luomista näytölle niin, että loppukäyttäjä näkee ne.
Tilamuuttuja	Normaalisti muuttujan arvo ”katoaa” kun päästään funktion ulkopuolelle, mutta tilamuuttuja säilyttää sen hetkisen arvon ja sitä on helpompi muokata.
Throttling	Funktio, joka rajoittaa funktion kutsumisen aikaväliä.

1 JOHDANTO

Tämän opinnäytetyön tarkoituksena on käydä läpi erilaisia tapoja optimoida React-sovellusta, hyödyntäen mm. React hookeja ja Reduxia. Optimoitaessa sovellusta tarkoituksena ei ole tehdä koko sovellusta uudestaan vaan muokata jo olemassa olevia komponentteja tehden niistä suorituskykyisempiä. Olemassa olevaa sovellusta optimoitaessa tämä on suuri etu, sillä näin sovellus pysyy koko prosessin aikana toiminnallisena.

Usein sovelluksen tekeminen on pitkä prosessi ja sidoksissa aikatauluun, kuten sprintteihin. Tällöin sorrutaan usein tekemään väliaikaisiksi kuviteltuja ratkaisuja, jotka myöhemmin kostautuvat, kun sovelluksen suorituskyky alkaa heikentyä. Reactissa pystytään hyödyntämään avoimen lähdekoodin kirjastoa Reduxia sekä tilanhallintaa ja ylläpitoa helpottavia React hookeja. React hookit ja Redux-kirjasto ovat suhteellisen tuoreita lisäyksiä, jotka mahdollistavat sovelluksen tekemisen helpommin skaalautuvaksi, nopeammaksi ja kaikin puolin suorituskykyisemmäksi.

2 OPTIMOINTI

Sovelluksen optimoimisella tarkoitetaan käyttäjän näkökulmasta koodin muokkaamista siten, että se toimii ihanteellisesti. Käyttäjäkokemus ei sisällä minkäänlaisia käyttökatkoksia, kaikki latautuu nopeasti ja moitteettomasti, eikä mitään niin sanotusti poikkeavaa tapahdu. Käyttäjäkokemus on kokonaisuudessaan eheä.

Optimoimisella tarkoitetaan ohjelmoijan näkökulmasta sitä, että sovelluksen suorittaminen vie mahdollisimman vähän prosessointitehoa. Tämä edellyttää, että dataa haetaan vain sen kokoisissa palasissa kuin on tarpeellista ja mielekästä menettämättä datan sujuvaa hallittavuutta. Lisäksi välimuistia käytetään tehokkaasti ja asioita prosessoidaan taustalla, jolloin käyttäjäkokemus ei kärsi.

Optimoinnista voi olla monenlaista hyötyä. Kaikessa lyhykäisyydessään se tarkoittaa, että käyttäjälle luodaan hyvä käyttökokemus. Mutta kun asiaa lähestytään ohjelmoijan näkökulmasta, se sisältää useammankin osa-alueen, jota loppukäyttäjä ei välttämättä edes tule ajatelleeksi. Loppujen lopuksi käyttäjä näkee vain mitä hänen kuuluukin nähdä, mutta ei tiedä mitä pinnan alla tapahtuu.

Sovellusta toteutettaessa voidaan saada hyvinkin nopeasti kasaan toimiva sovellus. Sovellus ei siitä huolimatta välttämättä toimi optimaalisesti. Optimaalisesti toimivan sovelluksen ei pitäisi aiheuttaa käyttäjälleen minkäänlaista negatiivista tuntemusta käyttäjäkokemuksen suhteen. Optimoimisella on usein myös ohjelmoijia hyödyttäviä vaikutuksia, sillä se tekee työstä usein helpompaa ja ylläpidettävämpää, kuten esimerkiksi Reduxia hyödyntäessä.

3 REACT HOOKIT

3.1 Mitä React hookit ovat

React hookit julkaistiin Reactin versiossa 16.8. Ne mahdollistavat Reactissa monipuolisempaa tilan ja ominaisuuksien käyttöä kuin aiemmin kirjoittamatta luokkia. Vaikka ne ovatkin suhteellisen tuore lisäys Reactin maailmaan, voi Reactia käyttää tismalleen samalla tavalla kuin aiemminkin, sillä hookit ovat taaksepäin yhteensopivia. React hookit kannustavatkin kirjoittamaan funktionaalisia komponentteja luokkakomponenttien sijasta. Funktionaalisen ja luokkakomponentin suurin ero on se, että luokkakomponentin tila on yksittäinen olio, kun taas funktionaalisessa komponentissa tila voi koostua useista muuttujista.

Aikaisemmin React-komponentit ovat muuttuneet helposti monimutkaisiksi, ja niiden uudelleenkäytöstä on tullut helposti haastavaa. Hookien käyttöä edelsivät niin sanotut elämänkaarifunktiot (lifecycle hooks) joita pystyi käyttämään vain luokkakomponenteissa. Hookit mahdollistavat samankaltaisten toiminnallisuuksien kirjoittamisen funktionaalisissa komponenteissa kuin elämänkaarifunktiot luokkakomponenteissa. Tämä mahdollistaa helpomman komponenttien uusiokäytön, sillä luokkapohjainen komponentti renderöityy tilanteissa, joissa komponentin tila päivittyy.

3.2 Hookit ja niiden käyttö

React hookit jaotellaan karkeasti kahteen osaan, joista ensimmäinen on perushookit. Perushookeihin lukeutuu useState, useEffect ja useContext. UseState on kaikista yksinkertaisin, mutta ehkä myös käytetyin ja samalla yksi hyödyllisimmistä React hookeista. Siinä on kyse tilallisesta muuttujasta, jonka arvoa voidaan päivittää saumattomasti. Ensimmäisen renderöinnin aikana muuttujan arvo on se, joka sille on sitä luotaessa annettu. Muuttujaa luotaessa luodaan myös funktio, jonka avulla muuttujan arvoa voidaan päivittää. UseState mahdollistaa sen, ettei uusia muuttujia tarvitse luoda joka kerta kun muuttujan arvoa halutaan päivittää.

Kun useStatea käytetään komponentissa, on kyseessä paikallinen tilamuuttuja, jonka arvoa voidaan muokata vain komponentin sisällä. Kuvassa 1 käydään läpi yksinkertainen esimerkki, jossa luodaan tilamuuttuja ”eläin”, jolle annetaan aloitusarvo ”Sika”. Kun nappia painetaan, kutsuu se funktiota, joka muuttaa eläimen arvon lampaaksi.

```
1  import "./styles.css";
2  import React, { useState } from 'react';
3
4  export default function App() {
5    const [eläin, vaihdaEläin] = useState('Sika');
6    function eläimenVaihto() {
7      vaihdaEläin('Lammas')
8    }
9    return (
10     <div className="App">
11       <h1>Maatilalla on</h1>
12       <h2>{eläin}</h2>
13       <button onClick={eläimenVaihto}>Vaihda eläintä</button>
14     </div>
15   );
16 }
17
```

KUVA 1. Komponentissa olevan paikallisen muuttujan käyttö ja arvon muuttaminen

UseEffectin sisälle voidaan sijoittaa funktioita, joilla voi olla jonkinlaisia sivuvaikutuksia. Lähtökohtaisesti useEffect aktivoituu jokaisella näkymän renderöinnillä. UseEffectille voidaan antaa kuitenkin tietynlaisia riippuvuuksia, jotka aiheuttavat useEffectin aktivoitumisen vain niissä tapauksissa, joissa riippuvuuksissa tapahtuu muutoksia.

Kuten useStaten kohdalla, on useEffectinkin käyttö komponenttikohtaista. Kuvassa 2 on jatkettu useStatesta edellä käyttämääni esimerkkiä. Kuten aiemminkin, on eläimen alkuarvoksi asetettu ”Sika”. Tilamuuttujan ”koira” aloitusarvo on ”Kultainennoutaja”, koska useEffectin sisältö renderöityy kerran sen riippuvuudesta huolimatta. Kuitenkin, kun riippuvuudeksi asetettu eläin-muuttuja muuttuu, päivittyy myös koiran arvo mopsiin.

```
import "./styles.css";
import React, { useState, useEffect } from "react";

export default function App() {
  const [eläin, vaihdaEläin] = useState("Sika");
  const [koira, asetaKoira] = useState("");

  function eläimenVaihto() {
    vaihdaEläin("Lammas");
  }

  useEffect(() => {
    asetaKoira("Kultainennoutaja");
    if (eläin === "Lammas") {
      asetaKoira("Mopsi");
    }
  }, [eläin]);

  return (
    <div className="App">
      <h1>Maatilalla on</h1>
      <h2>{eläin}</h2>
      <h2>{koira}</h2>
      <button onClick={eläimenVaihto}>Vaihda eläintä</button>
    </div>
  );
}
```

KUVA 2. useEffectin käyttö useState-muuttujan päivittämiseen

UseContext on kehitetty datan jakamiseen sovelluksen sisällä. Yleensä React-sovelluksessa tieto kulkee vain yläkomponentilta alikomponentille. Mikäli sovelluksessa olisi käytössä esimerkiksi lokalisaatiota tai tietynlainen käyttäjäprofiili, tulisi näiden tietojen jakamisesta sovelluksen eri komponenttien välillä erittäin vaikeaa. UseContext siis mahdollistaa datan kuljettamisen komponenttien välillä muutenkin kuin ylhäältä alaspäin propseina (Bugl 2019). Propsit ovat kuin funktiolle annettavia argumentteja.

4 REDUX

4.1 Reduxin esittely

Redux on avoimen lähdekoodin JavaScript-kirjasto, jota käytetään sovelluksen hallitsemiseen ja sen tilan keskittämiseen aivan kuten Reactin useContextia. Reduxia käytetään kuitenkin suuremmassa mittakaavassa. Kun Reactin toimintaa katsotaan laajemmasta näkökulmasta eikä tarkastella pelkästään käyttöliittymää, voidaan huomata, kuinka vaikeaa on ylläpitää sovelluksen tilaa (Chinnathambi 2018).

Reduxia käytettäessä voidaan säilöä mitä tahansa tilamuuttujia storeen eli eräänlaiseen komponenttien tilasäiliöobjektiin, josta niitä voidaan käyttää missä vain sovelluksen kerroksessa. Tämän ansiosta sovellus pysyy paremmin synkronoituna, kun ei tarvitse jatkuvasti huolehtia tilamuuttujien päivittymisestä. Reduxin käyttö parantaa täten myös sovelluksen suorituskykyä, kun voidaan päivittää yhdellä kertaa muuttujia, joita koko sovellus käyttää, eikä sitä tarvitse tehdä erikseen joka näkymässä.

4.2 Reduxin käyttö

Reduxissa erilaiset tilat tallennetaan storeen. Tämä mahdollistaa sen, että tärkeimpiä tilamuuttujia voidaan käyttää kaikkialla sovelluksessa ja niiden toiminnasta tulee paremmin ennustettavaa. Määritetyt tilat päivittyvät koko sovelluksen mittakaavalla, ja mikäli jokin menee pieleen, on syy paljon helpompi löytää.

```
1   import "./styles.css";
2   import { createStore } from "redux";
3
4   export default function App() {
5     function valitseKäyttäjä(state = { value: "" }, action) {
6       switch (action.type) {
7         case "Admin":
8           return { value: "Admin" };
9         case "Vierailija":
10          return { value: "Vierailija" };
11        default:
12          return state;
13      }
14    }
15
16    function admin() {
17      store.dispatch({ type: "Admin" });
18    }
19    function vierailija() {
20      store.dispatch({ type: "Vierailija" });
21    }
22
23    let store = createStore(valitseKäyttäjä);
24    store.subscribe(() => console.log(store.getState().value));
25    return (
26      <div className="App">
27        <h1>Tervetuloa</h1>
28        <h2>Valitse käyttäjä</h2>
29        <button onClick={admin}>Admin</button>
30        <button onClick={vierailija}>Vierailija</button>
31      </div>
32    );
33  }
```

KUVA 3. Esimerkki miten Reduxilla tallennetaan muuttujan tila

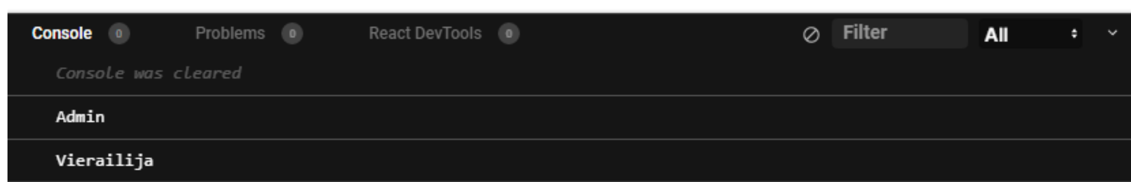
Kuvassa 3 käydään läpi yksinkertaisesti Reduxin käyttöä. Monesti verkkosovelluksissa on eritasoisia käyttäjiä. Helpoin tapa määrittää, mitä eri tason käyttäjille näyttää, on määrittää jonkinlainen tila kirjautumisen yhteydessä. Funktio valitseKäyttäjä on ns. reducer-funktio, joka ottaa argumentteinaan nykyisen tilan ja tapahtuman, sekä palauttaa uuden tilan tietyn tapahtuman jälkeen. Reducerissa ei ole pakko käyttää ehtoja niin kuin tässä esimerkissä käytän switch-lausetta ja sille määriteltyjä ehdollisia case-haaroja, mutta se helpottaa tilanhallintaa.

Kuvassa 3 näkyvään store nimiseen muuttujaan tallennetaan tämänhetkinen tila. Kuvassa 3 luon kaksi funktiota, joita kutsutaan riippuen kumpaa nappia painetaan. Nämä funktiot taas kutsuvat reducer-funktiota, joka päivittää storeen muuttujan tilan.

Tervetuloa

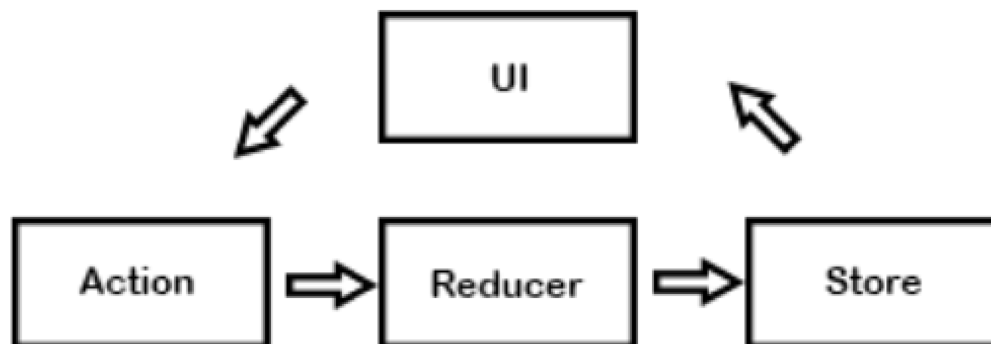
Valitse käyttäjä

Admin Vierailija



KUVA 4. Esimerkki siitä miten nappia painamalla storeen tallennettu tila muutetaan

Esimerkissäni (kuva 4) olen painanut molempia nappeja kerran ja näin koko sovelluksen käytössä oleva tila on muuttunut. Jos kyseessä olisi siis esimerkiksi kirjautumissivu voisi jo tässä vaiheessa määrittää, mitä sisältöä sovelluksen käyttäjälle näytettäisiin. Jos käyttäjä olisi vierailija, voitaisiin tietyt asiat jättää näyttämättä hyödyntämällä tätä tilaa ja renderöimällä elementtejä ehdollisesti. Lopulta kuvassa 5 käydään läpi tiivistetysti Reduxin toiminta.



KUVA 5. Havainnollistava kuva Reduxin toimintaa koskien

5 MUITA OPTIMOINNIN TAPOJA

5.1 Throttling

Pääasiallinen ero throttlingin ja debouncen välillä on se, että throttling takaa funktion suorittamisen säännöllisesti sille määritetyn ajan mukaan (Corbatcho 2018). Tämä tekeekin siitä hyvän vaihtoehdon funktioille, joiden suorittamista täytyy rajoittaa, muttei varsinaisesti estää tietyksi ajaksi. Kuten debouncea, voidaan myös throttlingia käyttää optimoinnin välineenä.

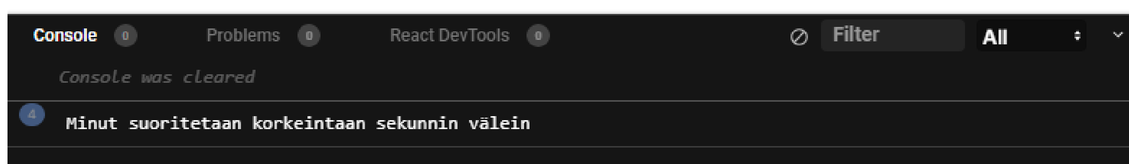
Kuten aiemmassa kappaleessa mainitsin throttling on eräänlainen säätely, joka määrittää kuinka usein jotakin toimintaa voidaan suorittaa. Olkoon nappi, jota painamalla kutsutaan funktiota, joka kasvattaa numeroa yhdellä. Jos tätä funktiota kutsutaan käyttäen throttlingia, voidaan samalla throttling-funktion sisällä määrittää aika, jonka välein tätä funktiota todellisuudessa kutsutaan. Kuvisa 6 ja 7 odotettavaksi ajaksi on määritetty sekunti, ja vaikka nappia painaisi useammin kuin kerran sekunnin aikana, tulostetaan konsoliin teksti vain korkeintaan joka sekunti. Kuvien 6 ja 7 esimerkeissä olen käyttänyt lodashin avoimen lähdekoodin kirjastoa. Kirjasto tarjoaa valmiin throttle-funktion ja säästää näin vaivaa.

```
1  import "./styles.css";
2
3  import throttling from "lodash.throttle";
4
5  export default function App() {
6
7    function log() {
8      console.log('Minut suoritetaan korkeintaan sekunnin välein')
9    }
10
11
12   const throttleFunc = () => throttling(log, 1000, 'leading')
13
14   return (
15     <div className="App">
16       <h1>Terveutuloa</h1>
17       <button onClick={throttleFunc()}>Paina minua</button>
18     </div>
19   );
20 }
21
```

KUVA 6. Esimerkki throttling-funktion käytöstä

Throttling

Paina minua



KUVA 7. Esimerkki throttling-funktion toiminnasta

5.2 Debouncing

Kuvitellaan, että olet hississä, jonka ovi on sulkeutumassa ja toinen ihminen tuleekin sisään. Hissi ei vaihda kerrosta vaan ovi aukeaa uudestaan ja tällä tapaa viivästyttää kerrosten vaihtamista (Corbatcho 2018). Tässä tapauksessa hissi ei siis liiku ennen kuin tietty aika on kulunut ja ovet sulkeutuneet.

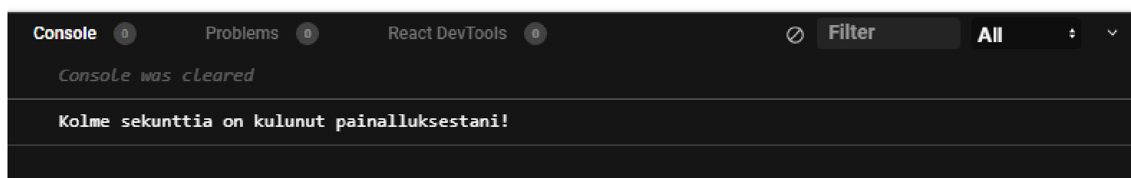
Kuten edellisessä kappaleessa kirjoitin, debounce on eräänlainen viivästys, joka määrittää, minkä ajan kuluttua tietty toiminto suoritetaan. Kuvitellaan, että on hakukenttä, johon kirjoitetaan nimeä. Sovelluksen suorituskyky rasittuu, jos listasta tehdään haku jokaisen kirjaimen jälkeen. Käytettäessä debouncea voidaan määrittää aika, joka täytyy olla kirjoittamatta ennen kuin varsinainen haku tai filtteröinti suoritetaan. Kuvien 8 ja 9 esimerkeissä on käytetty jo aiemmin mainitsemani lodash-kirjaston valmista debounce-funktiota.

```
import "./styles.css";

import debounce from "lodash.debounce";

export default function App() {
  function tapahtuma() {
    console.log("Kolme sekunttia on kulunut viimeisestä painalluksestani!");
  }

  const debounceFunc = () => debounce(tapahtuma, 3000);

  return (
    <div className="App">
      <h1>Tervetuloa</h1>
      <button onClick={debounceFunc()}>Paina minua</button>
    </div>
  );
}

```

KUVA 8. Esimerkki debounce-funktion käytöstä

Tervetuloa

Paina minua



KUVA 9. Funktiota suoritetaan vasta 3 sekuntia viimeisen kutsun jälkeen

5.3 React Fragment

React Fragment on elementti, joka on julkaistu versiossa 16.2.0. Vaikka ne ovatkin olleet olemassa jo jonkin aikaa, tämä ominaisuus ei ole kovin käytetty. Ominaisuutta käytetään turhien elementtien renderöinnin välttämiseksi DOMiin mikä säästää prosessoritehoa. Kuvassa 10 näytetään esimerkki, miten React Fragment toimii käytännössä.

Komponentin sisällä oleva renderöitävä JSX tarvitsee aina yhden parent-elementin, jonka sisälle kaikki renderöitävä sijoitetaan. Monessa tapauksessa kaiken renderöitävän sijoittaminen div-elementin sisään on kuitenkin turhaa, sillä div-elementti ei luo mitään näkyvää DOMiin, mutta joka kuitenkin renderöidään ja joka luo samalla ylimääräisen solmun. Tästä poikkeuksena on kuitenkin div-elementti, jolle luodaan jonkinlaista tyylittelyä. Pienessä sovelluksessa tämä ei haittaisi, mutta sovelluksen kasvaessa ja sen tullessa monimutkaisemmaksi luovat ylimääräiset elementit kuitenkin turhaa rasitusta sovelluksen prosessointia ajatellen.

```
1  import "./styles.css";
2
3  export default function App() {
4    return (
5      <>
6        <h1> Näin vältetään ylimääräisen elemntin renderöiminen</h1>
7      </>
8    );
9  }
10
```

KUVA 10. React Fragment:in käyttö div-elementin sijaan

React Fragment:in käyttö helpottaa myös ehdollista (engl. conditional) elementtien renderöintiä. Jos on monta erilaista elementtiä, joiden renderöinti riippuu

tietyistä ehdoista, ei kaikkia näitä tarvitse kietoa div-elementin tai jonkin muun elementin sisään.

Isossa sovelluksessa voidaan välttää helposti satojen turhien elementtien renderöiminen käyttämällä React Fragment -elementtiä. Näin pystytään välttämään helposti html-virheilmoituksia, tekemään sovelluksen toiminnasta nopeampaa ja säästämään muistin käyttöä.

5.4 React lazy loading

React lazy loading on julkaistu Reactin versiossa 16.6.0 ja se tuotiin kirjastoriippumattomana ominaisuutena. Se on merkittävä lisä React-sovelluksen suorituskykyä ajatellen. React lazy loadingin tarkoitus on jakaa komponenttien koodia osiin. Näin käyttäjä ei joudu katselemaan tyhjiä kohtia sivulla vaan näkee suspense-komponentin sisällä olevan väliaikaisen elementin. Suspense-komponentin sisään on mahdollista laittaa esimerkiksi latausikoni tai jokin yksinkertainen elementti kuten kuvassa 11.

```
1  import "./styles.css";
2  import React, { Suspense } from "react";
3
4  const Laiska = React.lazy(() => import('./laiska'));
5
6  export default function App() {
7    return (
8      <div className="App">
9        <Suspense fallback={<div>Ladataan...</div>}>
10         <Laiska></Laiska>
11       </Suspense>
12     </div>
13   );
14 }
15
```

KUVA 11. Esimerkki miten komponentti ladataan laiskasti ja miten näyttää väliaikainen elementti

Tämä ominaisuus on erityisen hyödyllinen, kun sovellus on suuri ja jokaisessa näkymässä on paljon ladattavaa. Lazy loading on yksi helpoimmista ja nopeiten arvoa tuottavista tavoista optimoida sovellusta ja sen tuoma arvo on nähtävissä nopeasti (Elrom 2021). Tämä tuo lisäarvoa myös käyttäjille, joilla on hidas

internet-yhteys, sillä silloin luodaan illuusio paremmin toimivasta sovelluksesta. Siispä käyttäjälle näytetään esimerkiksi nopeammin latautuvia elementtejä ennen kuin suuremmat on ladattu, kuten kuvassa 12 tapahtuu.

KUVA 12. Suspense-komponentin väliaikainen elementti ja lopulta laiskasti ladattu komponentti

6 OLEMASSA OLEVAN SOVELLUKSEN OPTIMOIMINEN

6.1 Työn taustaa

Työskentelen eSend Oy -nimisessä yrityksessä, jonka tuote on toiminnanohjaussovellus (ERP). Sovellus oli jo hyvin pitkällä, kun aloitin työskentelyn kyseisessä yrityksessä ja vei aikansa, että sain tutustuttua siihen kunnolla ja ymmärsin sovelluksen toimintaa. Sovellus toimii päällisin puolin erittäin hyvin, eikä käyttäjäkokemuksesta ole juurikaan mitään kielteistä sanottavaa, mutta sovelluksen kasvaessa olen huomannut, että jonkinlaiselle optimoimiselle olisi tarvetta. Loppukäyttäjä ei tätä niinkään välttämättä huomaa, mutta itse työskennellessäni ohjelmiston parissa koen, että joissain asioissa olisi parantamisen varaa.

Olemme töissä myös pyrkineet tuottamaan yhtenäistä koodia. Eri työn tekijöillä on usein erilaisia tapoja tehdä asioita ja se on alkanut hankaloittamaan koodin ymmärtämistä joissain kohdin. Tavoitteeni onkin, että saisin työlläni luotua jonkinlaista ohjenuoraa mitä kaikkien olisi hyvä noudattaa.

6.2 Työn vaiheet

Suunnitelmani työpaikkani sovelluksen optimointia varten oli seuraava. Ensin teen helpoimman ja kenties nopeimman työvaiheen alta pois, eli vaihdan kaikki turhat div- ja span-elementit React Fragment -elementteihin. Seuraavaksi siirryn hieman vaikeampaan vaiheeseen eli siirrän sovelluksen eniten käytetyn rajapinnan subscription-toiminnallisuuden Reduxia hyödyntäen yhteen tiedostoon, josta koko sovellus pystyy sitä käyttämään. Tässä vaiheessa en kuitenkaan uudista koko sovellusta niin, että jokainen näkymä käyttäisi jo tätä yhdestä tiedostosta. Viimeiseksi jätän debouncen hyödyntämisen karttapalvelumme osoitteenhakukentän optimoimisessa ja lazy loadingin hyödyntämisen kirjautumisen yhteydessä, sillä se on ylivoimaisesti pisimpään aikaa vievä siirtymä.

Työn vaiheet on pyritty jakamaan niin, että ne eivät ole varsinaisesti liitoksissa toisiinsa. Jokainen vaihe on erillinen prosessinsa. Vaiheet toteutetaan niiden tärkeysjärjestyksessä.

6.3 Optimointi

6.3.1 React Fragmentien hyödyntäminen

Aloitin optimoimiseni helpoimmasta, eli vaihdoin kaikki sovelluksen tyhjät div- ja span-elementit React Fragmenteihin. Huomasin että sovelluksessamme on käytetty todella paljon tyhjiä elementtejä turhaan. Turhaan voi olla hieman väärä ilmaisu sillä kaikki palautettava JSX on käärittävä jonkin elementin sisään. Tämä kuitenkin aiheuttaa aiemmin mainitsemieni turhien elementtien renderöimisen, joten korvasin ne React Fragmenteilla.

Työmäärä ei ollut suuri, sillä käytin koodieditoriani korvaamaan kaikki turhat elementit. Yllätykseksi muutoksia tuli kuitenkin 22 tiedostoon. Todellisuudessa korvasin 33 turhaa elementtiä React fragmenteilla. Jo tämän jälkeen oli havaittavissa jonkinasteista latauksen nopeutumista vaihdettaessa näkymää.

6.3.2 Reduxin hyödyntäminen subscriptionilla

Sovelluksemme, joka on logistiikan toiminnanohjausta varten, sisältää paljon erilaisten tilausten käsittelyä muodossa tai toisessa. Lista tilauksista saadaan back endiltä, ja mikäli niihin tulee muutoksia käyttäjän ollessa kirjautunut, laukaisee se rajapinnan subscriptionin, joka päivittää tilauslistaa. On kuitenkin vaivalloista, että jokaisessa näkymässä, jossa tilauksia käsitellään, joudutaan rajapintaan tekemään subscription ja lopuksi päättämään se.

Päätin ratkaista ongelman siirtämällä tilauslistan ja sen päivittämisen Reduxin storeen. Eli kuten aiemmin kävin luvussa 4 läpi, tallennetaan storeen koko sovelluksen käytössä oleva muuttuja. Näin säästetään paljon aikaa ja koodirivejä, kun aina ei tarvitse tehdä samaa koodin toistoa, mikäli kyseisiä tilauksia käsitellään. Myös mikäli ongelmia ilmenee, on ne helpompi jäljittää, jos koko sovellus käyttää samaa tilamuuttujaa.

Tämä työn vaihe oli ehkä monimutkaisin, mutta sekin sujui silti melko mutkattomasti. Koska niin moni näkymä sovelluksessamme käyttää tilausten listaamista, implementoin ratkaisuni vasta yhteen näkymään, jotta sitä voidaan testata ennen ratkaisuni kaikkialle siirtämistä. Toistaiseksi en kuitenkaan huomannut minkäänlaisia virheilmoituksia tai kaatumisia toimintani johdosta. Positiivista on myös, että sain poistettua jonkin verran koodia jo olemassa olevasta sovelluksesta sen jälkeen, kun käytin vain Reduxin storeen tallentamaani tilamuuttujaa.

6.3.3 Throttlingin hyödyntäminen hakukentässä

Sovelluksemme käyttää Mapbox-nimistä karttapalvelua, jonka hakukenttä on toteutettu ennakoivasti ehdotuksia antavalla ominaisuudella. Ongelma on ollut kuitenkin se, että hakukentän ehdotukset ovat vaihtuneet todella nopealla tahdilla. Koin tässä tilanteessa järkevämmäksi käyttää debounce-menetelmän sijasta throttlingia.

Itse toteutus oli myös melko yksin kertainen. Käytin avoimen lähdekoodin JavaScript-kirjastoa lodashia. Aiemmin sovelluksessamme oli funktio, jonka sisällä rajapintapyyntö hakukentän ehdotuksia varten tehtiin. Kutsuja tehtiin kuitenkin joka näppäimen painalluksella, kun tekstisyötteen pituus ylitti kolme merkkiä. Tämä loi tarpeettoman paljon rajapintakutsuja, ja ehdotukset vaihtuivat jatkuvasti. Ratkaisin ongelman kutsumalla tätä rajapintakutsun sisältävää funktiota throttle-funktiolla, korkeintaan kerran 1,5 sekunnissa. Tämä piti ehdotusten vaihtumisen maltillisempana, ja keskivertokirjoittajan nopeudella myös rajapintakutsujen määrä jää varsin pieneksi.

Muutos voi kuulostaa varsin pienelle, mutta se keventää sovelluksen suoritusta kyseisessä näkymässä. Myös virheilmoitusten mahdollisuuden määrä pieneni, sillä rajapintakutsuilla oli välillä vaikeuksia pysyä kirjoittamisen perässä. Vaikka debounce ja throttle ovatkin samankaltaisia toiminnallisuudeltaan on niiden käyttötarkoitus erilainen (Corbatcho 2018). Debouncen käyttäminen tässä tilanteessa ei olisi tuottanut arvoa hakukenttäehdotusten takia.

6.3.4 Lazy loading käyttäjäkokemuksen parantamiseksi

En ole törmännyt suuriin käyttäjäkokemukseen kielteisesti vaikuttaviin asioihin sovelluksemme parissa. Kuitenkin yksi asia, johon olen kiinnittänyt huomiota, on jo aiemmin mainitsemani Mapbox-kartan lataaminen sovelluksen eri näkymiin. Tila, johon kartan pitäisi latautua, on usein tyhjänä hetken.

React lazy loading on suunniteltu tämänkaltaisia tilanteita varten. Latasin karttakomponentin sitä käyttäneisiin näkymiin ”laiskasti”. Nämä laiskasti ladatut komponentit taas sijoitin Suspense-komponentin sisälle. Suspense-komponentille annoin propsina elementin, joka näytetään, mikäli laiskasti ladatun komponentin lataamisessa kestää. Käytin hyödykseni avoimen lähdekoodin Material-UI-komponenttikirjaston valmista latausikonia.

Itse latausikoni ei näkynyt kauan karttanäkymiäni testattaessa, mutta se parantaa silti käyttäjän kokemusta, että tämän ei tarvitse tuijottaa tyhjää kohtaa, johon yhtäkkiä ilmestyy kartta. Tämän sijasta hän näkee latausikonin, eikä käyttäjäkokemus rikkoudu missään vaiheessa. Testasin ominaisuutta vielä hitaammalla nettiyhteydellä, ja tässä tapauksessa latausikoni todella pyöri pidempään. Onkin hyvä varautua mahdollisiin häiriötilanteisiin niin, että sivu ei näytä rikkonaiselle.

6.4 Lopputulos

Olen itse varsin tyytyväinen optimoimisen lopputulokseen. Tiedossa olikin, että työmäärä ei tule olemaan suuri, mutta sitäkin opettavaisempi. Erityisen hyvin koen onnistuneeni käytetyimmän ja eniten sovellusta rasittavan toimitus-subscriptionin säilömisessä React storeen. Tätä ei ole kuitenkaan vielä hyödynnetty kaikkialla sovelluksessa, sillä ensin halutaan olla varmoja, että ominaisuus toimii niin kuin pitää yhdessä näkymässä, ennen kuin se voidaan ottaa käyttöön kaikkialla sovelluksessa. Toteutus herätti kuitenkin mielenkiintoa työpaikallani ja se tullaan todennäköisesti ottamaan koko sovelluksessa käyttöön.

Pienemmät ominaisuudet onnistuivat myös hyvin, eikä mitään kielteistä niidenkään suhteen ole ilmennyt. Vaikka näiden pienempien ominaisuuksien välitön hyöty oli pienempi kuin edellä mainitun, niistäkin irtosi sitä enemmän oppia. Tästä lähtien osaan kiinnittää huomiota siihen, miten sovelluksen turhaa rasittamista voidaan välttää eri tavoin, ja pyrin siihen, että myös muut työpaikallani oppisivat toimimaan näin.

7 POHDINTA

eSendille toimeksiantona tehty optimointityö saatiin tehtyä aikataulussa ja jopa nopeammin kuin aluksi oli suunniteltu. Toistaiseksi en ole kuullut, että tekemäni työ olisi aiheuttanut minkäänlaista toiminnallisuuden rikkoutumista, sillä loppujen lopuksi se oli vain olemassa olevan koodin parantelua ja pienien ominaisuuksien lisäämistä. Ennen varsinaista toteutusta olin miettinyt valmiiksi sovelluksen toiminnallisuuksia, jotka tarvitsisivat suorituksen kannalta parantelua. Tämä mahdollisti sen, että itse käytännön toteutus sujui nopeasti. Sovelluksen käyttö ei katkennut missään välissä toimieni takia.

Optimoimisen eri tapojen opiskelu on auttanut itseäni kehittymään ohjelmoijana, ja osaan ottaa erilaisia asioita huomioon jo kehitysvaiheessa, eikä uusien ominaisuuksien pariin tarvitse ehkä palata enää niin useasti vaan osaan tehdä asiat kerralla oikein. Toivon myös, että pystyn opastamaan työkavereitani oppimieni asioiden tiimoilta ja auttamaan myös heitä kehittymään. Loppujen lopuksi optimoimisesta jäi paljon käteen – tärkeimpänä tieto siitä, miten tehdä asioita paremmin.

Jälkeenpäin ajateltuna en tekisi montaakaan asiaa eri tavalla. Kaikki sujui melko lailla suunnitelmieni mukaan, eikä suurempia vastoinkäymisiä ilmaantunut. Varmasti olisi ollut paljon enemmänkin tehtävissä sovelluksen suorituskyvyn hyväksi, mutta olen varsin tyytyväinen lopputulokseeni. Lisäksi olisin ainakin teoriassa voinut perehtyä erilaisiin React hookeihin vielä laajemmin. Hookeja on olemassa suhteellisen paljon, mutta opinnäytetyön laajuutta ja resurssien järkevää kohdentamista silmällä pitäen päädyin käyttämään hyväkseni vain tunnetuimpia hookeja.

LÄHTEET

Bugl, D. 2019. Learn React Hooks: build and refactor modern React.js applications using Hooks. 1. painos. Birmingham: Packt Publishing, limited. (Viitattu 13.11.2021)

Chinnathambi, K. 2018. Learning React: a hands-on guide to building web applications using React and Redux. 2. painos. Addison-Wesley Professional. (Viitattu 15.10.2021)

Corbatcho David. 2018. Debouncing and Throttling Explained Through Examples. Julkaistu 6.4.2016. Päivitetty 3.1.2018. Luettu 20.11.2021. https://css-tricks.com/debouncing-throttling-explained-examples/

Elrom, E. 2021. React and Libraries: Your Complete Guide to the React Ecosystem. 1. painos. Berkeley, CA: Apress L. P. (Viitattu 15.11.2021)