jamk.fi

Redux-tilanhallinnan korvaaminen ReactJS:n uusilla ominaisuuksilla

Tiia Aarnio

Opinnäytetyö
Toukokuu 2019
Tekniikan ja liikenteen ala
Insinööri (AMK), tieto- ja viestintätekniikan tutkinto-ohjelma
Ohjelmistotekniikka

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Aarnio, Tiia	Opinnäytetyö, AMK	Toukokuu 2019
	Sivumäärä 40	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi

Redux-tilanhallinnan korvaaminen ReactJS:n uusilla ominaisuuksilla

Tutkinto-ohjelma

Insinööri (AMK), tieto- ja viestintätekniikka

Työn ohjaaja(t)

Jani Immonen, Petri Mutka

Toimeksiantaja(t)

Digia Finland Oy

Tiivistelmä

Opinnäytetyön tavoitteena oli toimittaa toimeksiantajalle selvitys Redux-tilanhallinnan korvaamisesta ReactJS:n uusilla ominaisuuksilla. Selvityksessä kartoitettiin ReactJS:n uusien ominaisuuksien hyödyt ja haitat sekä kypsyys tuotantokäyttöön. Pohdittiin myös, millä tavoin uusien ominaisuuksien mahdollinen käyttöönotto kannattaisi toteuttaa sovelluksessa, jossa Redux-tilanhallinta on tällä hetkellä jo käytössä. Lisäksi selvityksessä vertailtiin sovelluksen koodin luettavuutta sekä DOM-rakenteen selkeyttä Redux-tilanhallinnasta luopumisen jälkeen.

Selvitys toteutettiin referenssitoteutuksen avulla, joka vastasi pääpiirteiltään perinteistä React-sovellusta, jossa on käytössä yksinkertainen Redux-tilanhallinta. Redux-tilanhallinnan lisäksi referenssitoteutukseen toteutettiin muutamia komponentteja, joissa käytettiin elämänkaarifunktioita ja Reactin paikallista tilaa. Tällä tavoin ReactJS:n uusia ominaisuuksia saatiin vertailtua kattavammin.

ReactJS:n uudet ominaisuudet otettiin käyttöön referenssisovelluksesta luotuun kopioon. Kopioidusta sovelluksesta poistettiin Redux sekä kaikki luokkakomponentit muutettiin funktiokomponenteiksi. Globaali tilanhallinta, komponenttien paikallinen tila sekä elämänkaarifunktiot toteutettiin uudestaan käyttäen ReactJS:n uudistunutta kontekstia sekä Hooks-ominaisuuden tarjoamia toiminnallisuuksia.

Selvityksen tuloksena voitiin nähdä, että ReactJS:n uusia ominaisuuksia käyttämällä saatiin sovelluksen koodirivien määrää merkittävästi vähennettyä sekä DOM-rakenteen selkeyttä parannettua. Työssä tutkitut uudet ominaisuudet vaikuttivat käytössä stabiileilta, mutta todettiin kuitenkin, ettei niiden käyttöönotto tuotannossa olisi vielä täysin riskitöntä.

Avainsanat (asiasanat)

React, Redux, React Hooks, JavaScript

Muut tiedot (salassa pidettävät liitteet)

jamk.fi

Description

Author(s)	Type of publication	Date
Aarnio, Tiia	Bachelor's thesis	May 2019
		Language of publication: Finnish
	Number of pages 40	Permission for web publication: x

Title of publication

Replacing Redux with new features of React

Degree programme

Information and Communication Technology

Supervisor(s)

Immonen, Jani; Mutka, Petri

Assigned by

Digia Finland Oy

Abstract

The aim of the thesis was to deliver the assignor a report about the state management replacement of Redux with the new features of ReactJS. The study looked at the benefits and disadvantages of the new features of ReactJS and their matureness for use in production. It was also considered how the potential introduction of the new features should be implemented in an application where Redux is currently in use. In addition, the study compared the application code readability and the DOM structure clarity after Redux was abandoned.

The study was carried out by building a reference application including the main features of the traditional React application, which uses simple Redux state management. In addition to Redux state management, a few components using life cycle methods and a local state of React were implemented into the reference application. By this way, the new features of ReactJS could be more comprehensively compared.

The new features of ReactJS were implemented into the copy created from the reference application. Redux was removed from the copied application and then all class components were converted to function components. Global state management, local state of components and life cycle methods were handled with ReactJS's new context and Hook features.

As a result of the research, it was seen that using the new features of ReactJS, the application code readability and the DOM structure clarity were significantly improved. The new features studied in the project seemed to be stable in use; however, it was noted, that their introduction into production would not be completely risk-free.

Keywords/tags (subjects)

React, Redux, React Hooks, JavaScript

Miscellaneous (Confidential information)

Sisältö

Sanasto **4**

1 Työn lähtökohdat **6**

1.1 Toimeksiantajan esittely 6

1.2 Toimeksianto ja tavoitteet 6

2 Moderni web-kehitys **7**

3 ReactJS **8**

3.1 JSX 8

3.2 Komponentit 8

3.3 Paikallinen tila ja elämänkaari 10

3.4 DOM ja virtuaalinen DOM 11

3.5 Hooks 12

3.6 Context 14

4 Muut käytetyt tekniikat ja työkalut **15**

4.1 JavaScript 15

4.2 Redux 15

4.3 React Redux 16

4.4 React Router 17

4.5 Formik 17

5 Toteutus **18**

5.1 Yleistä 18

5.2 Kehitysympäristö 18

5.2.1 Yleistä 18

5.2.2 Webpack 20

5.2.3 Babel 20

5.2.4 ESLint ja Prettier ... 20

5.3 React-sovellus Redux-tilanhallinnalla ... 21

5.3.1 Yleistä ... 21

5.3.2 Redux-tilanhallinta ... 23

5.3.3 Listakomponentti ... 25

5.3.4 Lomakekomponentti ... 27

5.3.5 Notifikaatiokomponentti ... 27

5.4 React-sovellus ilman Reduxia ... 28

5.4.1 Yleistä ... 28

5.4.2 Tilanhallinta ilman Reduxia ... 29

5.4.3 Luokkakomponentista funktiokomponentiksi ... 31

5.4.4 Elämänkaarifunktiot funktiokomponentissa ... 32

6 Tulokset ... 33

6.1 Vertailu ... 34

6.2 Hyödyt ja haitat ... 35

7 Loppupohdinta ... 36

7.1 Tavoitteet ... 36

7.2 Tulokset ... 36

7.3 Työskentely ... 37

7.4 Johtopäätökset ... 38

Lähteet ... 39

Kuviot

Kuvio 1. Esimerkki JSX-syntaksista ... 8

Kuvio 2. Esimerkki funktiokomponentista ... 9

Kuvio 3. Esimerkki luokkakomponentista ... 9

Kuvio 4. Komponentin tilan alustus ja käyttö ... 10

Kuvio 5. Komponentin elämänkaarifunktiot 11
Kuvio 6. Virtuaalisen ja oikean DOM-rakenteen vertailu..... 12
Kuvio 7. Hooks-esimerkki 13
Kuvio 8 Reduxin datan kulku 16
Kuvio 9 Package.json-tiedosto..... 19
Kuvio 10. Sovelluksen juurikomponentti 22
Kuvio 11. Redux-tilasäilön luonti 23
Kuvio 12. Reducer-funktio 24
Kuvio 13. Reducer-funktiolle lähetettävä toiminto 24
Kuvio 14. Komponentin liittäminen Redux-tilasäilöön 25
Kuvio 15. GiraffeListComponent-luokkakomponentti 26
Kuvio 16. Paikallisen ja globaalin tilan muokkaaminen 27
Kuvio 17. Luokkakomponentin elämänkaarifunktiot..... 28
Kuvio 18. Provider-komponentti ilman Reduxia..... 29
Kuvio 19. RootReducer-funktio ilman Reduxia..... 30
Kuvio 20. Dispatch-metodi toiminnon sisällä 30
Kuvio 21. Kontekstin ja toimintojen käyttö komponentissa 31
Kuvio 22. Paikallinen tila funktiokomponentissa..... 32
Kuvio 23. useEffect-funktio komponentissa..... 33
Kuvio 24. Koodirivien määrät eri komponenteissa..... 34

Sanasto

API

Application programming interface eli ohjelmointirajapinta on ohjelmien keskinäiseen keskusteluun käytetty määritelmä.

CSS

Cascading Style Sheets on ohjelmointikieli dokumenttien tyyliohjeiden kirjoittamiseen.

DOM

Document Object Model eli dokumenttioliomalli kuvaa rakenteisen dokumentin puurakenteena.

HOC

Higher-Order Component on funktio, joka ottaa vastaan komponentin ja palauttaa uuden komponentin.

HTML

Hypertext Markup Language on hypertekstin standardoitu kuvauskieli.

IDE

Integrated development environment eli ohjelmisto, jota käytetään sovellusten kehittämiseen.

JavaScript

Erityisesti web-ohjelmointiin käytetty ohjelmointikieli.

JSON

JavaScript object notation on yksinkertainen tiedostomuoto tiedonvälitykseen.

JSX

JavaScriptin syntaksilaajennus

MVC-arkkitehtuuri

Model-View-Controller on usein käyttöliittymäkehityksessä käytetty arkkitehtuurimalli, joka sisältää kolme osaa: mallin, näkymän ja käsittelijän.

NPM

Node Package Manager eli paketinhallintatyökalu, jolla voi jakaa JavaScript paketteja sekä lisätä niitä omaan projektiin.

Package.json

JSON-muotoinen tiedosto, joka sisältää projektin metatiedot.

React

JavaScript-kirjasto käyttöliittymäkehitykseen

Redux

JavaScript-kirjasto sovelluksen tilanhallintaan

Renderöinti

Renderöinti tarkoittaa käyttöliittymäkehityksessä elementtien luomista näytölle.

SPA

Single-Page Application on web-sovellus, joka lataa yhden HTML-sivun ja päivittää sitä dynaamisesti käyttäjän toimintojen mukaan.

Syntaksi

Syntaksilla tarkoitetaan ohjelmointikielen sääntöjä, joilla määritetään ohjelmointikielen muotoilu.

URL

Uniform Resource Locator on merkkijono, jolla osoitetaan verkkosivujen sijainti.

1 Työn lähtökohdat

1.1 Toimeksiantajan esittely

Toimeksiantajana opinnäytetyölle toimi Digia Finland Oy. Digia on IT-alan palveluyritys, joka työllistää yli 1000 henkilöä usealla eri paikkakunnalla ja sen liikevaihto oli 112,1 miljoonaa euroa vuonna 2018. Erityisen vahvaa toimialaosaamista yritykseltä löytyy kaupan, logistiikan ja teollisuuden alalta sekä pankki- ja vakuutusalalta. (Digia yrityksenä n.d.)

Digian historia on saanut alkunsa vuonna 1990 perustetusta SysOpen Oyj:stä sekä vuonna 1997 perustetusta Digia Oy:stä. Yritykset yhdistyivät SysOpen Digia Oyj:ksi vuonna 2005 ja vaihtoivat nimensä lopulta Digiaksi vuonna 2008. Lisäksi vuosien saatossa on tehty useita yrityskauppoja. (Historia ja yrityskaupat n.d.)

1.2 Toimeksianto ja tavoitteet

Opinnäytetyön tarkoituksena oli toimittaa toimeksiantajalle selvitys Reactin uusista ominaisuuksista ja niiden hyödyistä sekä kypsyydestä tuotantokäyttöön. Tavoitteena oli kartoittaa, onko jo kehityksessä olevan keskikokoisen sovelluksen käyttöliittymäkerrosta mahdollista yksinkertaistaa pitkässä juoksussa korvaamalla Redux-tilanhallinta Reactin uusilla ominaisuuksilla.

Yksinkertaistamisella haettiin sovelluksen käyttöliittymäkehitykseen helpompaa lähestyttävyyttä. Tarkasteltiin, ratkaiseeko Hooks-ominaisuus Reactin tunnistetut tilanhallintaongelmat ja vertailtiin, vähentääkö Redux-kirjastosta luopuminen kirjoitettujen koodirivien määrää ja selkeytyykö sovelluksen DOM-rakenne.

Työssä käytetyt teknologiat ovat toimeksiantajan valitsemia ja ne on esitelty työn taustatutkimuksessa. Tietoperustana on käytetty pääasiassa teknologioiden omia dokumentaatioita sekä aiheeseen liittyviä kirjoja ja artikkeleita.

2 Moderni web-kehitys

Vuonna 1980 fyysikko Tim Berners-Lee loi yksinkertaisen hypertekstiohjelman nimeltä ENQUIRE henkilökohtaiseen käyttöönsä. Myöhemmin hän sai tehtäväkseen keksiä keino, jolla voidaan jakaa helposti dataa sekä dokumentteja tiedemiesten välillä ja vuonna 1990 hän loi tätä tarkoitusta varten ensimmäisen version HTML:stä. Päästäkseen lopulliseen tavoitteeseen, Tim loi lisäksi HTTP-protokollan, ensimmäisen internetselaimen sekä ensimmäisen web-serverin. Omissa muistiinpanoissaan Tim oli myös listannut tietosanakirjan yhdeksi mahdolliseksi käyttökohteeksi HTML:lle. (Nutter 2017.)

Nykyään web-kehitys on kuitenkin paljon muutakin kuin pelkkää HTML:ää ja se on edennyt vielä paljon Wikipediaakin pidemmälle. Yksinkertaisempien vuosien jälkeen on koittanut aika, jolloin web-kehitys elää jatkuvassa muutoksessa, koska web-sovelluksilta vaaditaan koko ajan enemmän ja ne muistuttavat vuosi vuodelta enemmän perinteisiä tietokoneohjelmia. Jatkuvat muutokset vaatimuksissa sekä teknologioissa luovat kuitenkin epävarmuutta tulevasta ja hankaloittavat teknologiavalintojen tekemistä. (Nutter 2017.)

Suurin osa web-kehityksestä tehdään nykyään erilaisilla JavaScript- kehyksillä ja -kirjastoilla kuten AngularJS tai ReactJS. Siinä missä kehykset antavat kehittäjälle tarkemmat rajat toteutusmahdollisuuksille, ne saattavat piilottaa kehittäjiltä paljon asioita abstraktien kerrosten taakse. Kirjastot sen sijaan ovat helppoja ymmärtää, mutta toteutusmahdollisuuksien häilyvien rajojen vuoksi kehittäjille voi olla epäselvää, miten asioita kannattaisi oikeasti tehdä. JavaScript-kehykset ja -kirjastot helpottavat kuitenkin yhtä lailla monimutkaisten web-sovellusten rakentamista, joten pelkän HTML:n, CSS:n ja JavaScriptin käyttäminen ei välttämättä olisi enää tänä päivänä järkevää. (Nutter 2017.)

Stack Overflow-sivuston teettämästä tutkimuksesta selviää, että vuonna 2019 yleisin ohjelmointikieli on seitsemättä vuotta putkeen JavaScript. Tästä voidaankin päätellä, että JavaScript-kirjastot ja -kehykset menestyvät hyvin. Vuoden 2019 tammi-helmikuussa toteutetusta tutkimuksesta selviää myös, että ReactJS on vuonna 2019 pidetyin ja halutuin web-kehys. Tutkimukseen vastasi lähes 90 tuhatta ohjelmoijaa 179 eri maasta. (Developer Survey Results 2019.)

3 ReactJS

React eli ReactJS tai React.js on alun perin Facebookin insinöörien luoma avoimen lähdekoodin JavaScript-kirjasto, ja sen ensimmäinen versio julkaistiin vuonna 2013. React kehitettiin ratkaisemaan ongelmia liittyen monimutkaisten ja paljon muuttuvaa dataa sisältävien käyttöliittymien kehitykseen. Paljon käytetystä MVC-arkkitehtuurista React mahdollistaa näkymäosuuden(view) rakentamisen. (Gackenheimer 2015, luku 1.)

3.1 JSX

JSX on JavaScriptin syntaksilaajennus, joka muistuttaa HTML-syntaksia sisältäen kuitenkin kaikki JavaScriptin ominaisuudet. Kuviossa 1 nähdään, kuinka JSX mahdollistaa minkä tahansa validin JavaScript-lausekkeen sisällyttämisen elementteihin aaltosulkuja käyttämällä. JSX-lausekkeet kääntyvät puhtaaksi JavaScript-koodiksi, joka mahdollistaa JSX-elementtien käytön myös esimerkiksi silmukoiden sisällä tai muuttujina. (Introducing JSX n.d.)

```
const giraffeName = "Kaapo"
const jsxElement = <h1>Hello, {giraffeName}</h1>
```

Kuvio 1. Esimerkki JSX-syntaksista

JSX-syntaksi helpottaa käyttöliittymän hahmottamista JavaScript-koodin sisällä sen visuaalisemman syntaksin vuoksi. Tämän lisäksi sillä voidaan tuottaa puhtaita React-elementtejä ja sitä suositellaankin käytettävän yhdessä Reactin kanssa, vaikka se ei olekaan pakollista. JSX mahdollistaa myös Reactille hyödyllisempien virheilmoitusten näyttämisen. (Introducing JSX n.d.)

3.2 Komponentit

React on komponenttipohjainen kirjasto, jolla mahdollistetaan käyttöliittymän pilkkominen itsenäisiin uusiokäyttöisiin palasiin eli komponentteihin. Komponentit ovat

yksinkertaisuudessaan JavaScript funktiota, jotka palauttavat React-elementtejä. React-elementin tarkoitus on kuvata, mitä näytöllä pitäisi näkyä. (Components and props n.d.)

Komponentteja on kahdenlaisia: funktiokomponentteja ja luokkakomponentteja. Yksinkertaisin tapa luoda komponentti on funktiokomponentti. Funktiokomponentit ovat normaaleja JavaScript-funktioita, jotka ottavat vastaan ominaisuusobjektin eli properties-objektin(props) ja palauttavat React-elementin (ks. kuvio 2). (Components and props n.d.)

```
function GiraffeInfo(props) {
  return <p>Name: {props.giraffeName}</p>
}
```

Kuvio 2. Esimerkki funktiokomponentista

Luokkakomponentit ovat nimensä mukaan JavaScript-luokkia ja niillä voi olla myös muita ominaisuuksia props-objektin lisäksi (ks. kuvio 3). Kuvioissa 2 ja 3 esitellyt komponentit ovat identtisiä Reactin näkökulmasta. (Components and props n.d.)

```
class GiraffeInfo extends React.Component {
  render() {
    return <p>Name: {props.giraffeName}</p>
  }
}
```

Kuvio 3. Esimerkki luokkakomponentista

Props-objekti, joka komponentille voidaan antaa, voi sisältää mitä tahansa dataa. Ominaisuudet voidaan antaa komponentille esimerkiksi merkkijonona tai JavaScript-lausekkeena, mikäli käytössä on JSX-syntaksi. Props-objektin arvoja ei tulisi ikinä

muuttaa komponentin sisällä riippumatta siitä, onko komponentti funktio- vai luokkakomponentti. (Bagnardi, Fieldman, Hall & Højberg 2016, luku 4.)

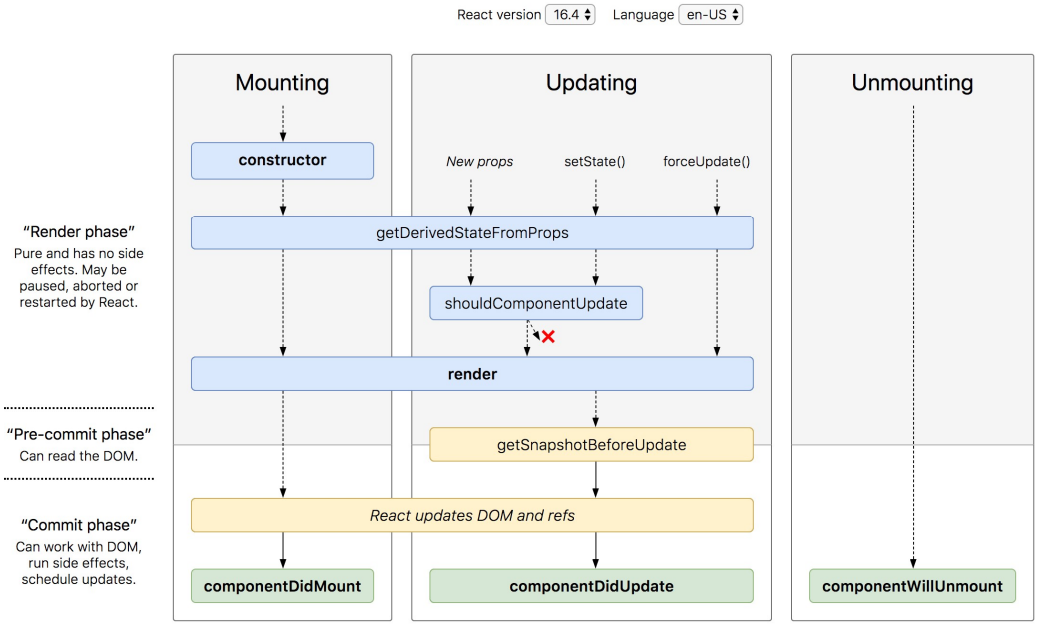
3.3 Paikallinen tila ja elämänkaari

React-komponentilla voi olla paikallinen tila(state), joka eroaa ominaisuuksista siten, että paikallinen tila on komponentin sisäinen ja sen arvoja voidaan manipuloida komponentin sisällä. Ilman sisäistä tilaa komponentti luokitellaan staattiseksi, eikä se tällöin ole interaktiivinen. Kuviossa 4 nähdään, kuinka komponentin tila alustetaan ensin luokkakomponentin constructor-funktiossa ja otetaan sen jälkeen käyttöön komponentin sisällä. Paikallisen tilan päivittäminen laukaisee reaktiivisen renderöinnin, jolloin komponentti ja sen lapsikomponentit renderöidään uudelleen. (de Sousa Antonio 2015, luku 1.)

```
export class StateExample extends React.Component {
    constructor(props) {
        super(props)
        this.state = {giraffeName: 'Kaapo'};
    }
    render() {
        return (
            <div>
                <p>{this.state.giraffeName}</p>
                <button onClick={() => this.setState({giraffeName: 'Kosti'})}>
                    Change name
                </button>
            </div>
        )
      }
}
```

Kuvio 4. Komponentin tilan alustus ja käyttö

Komponentille voidaan luoda elämänkaarifunktioita. Nämä ovat funktioita, joita ajetaan komponentin niin kutsutuissa elämänvaiheissa. Eri elämänvaiheita ovat kiinnittyminen(mounting), päivittäminen(updating) ja irtautuminen(unmounting) (ks. kuvio 5). (Hamedani 2018a.)

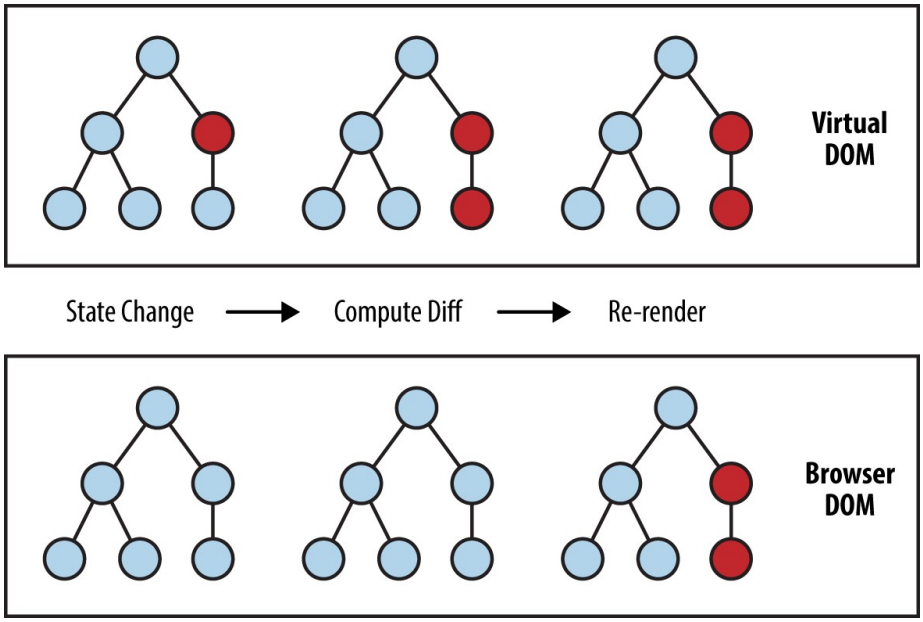


Kuvio 5. Komponentin elämänkaarifunktiot (Hamedani 2018a)

Kuviossa 5 näkyvistä elämänkaarifunktioista yleisimmät on lihavoitu. Renderöintifunktio(render) on näistä yleisin, sillä se on pakollinen kaikissa luokkakomponenteissa. Renderöintifunktio ajetaan aina komponentin kiinnittyessä ensimmäistä kertaa tai sen päivittyessä tapahtuneiden muutosten myötä. Heti kiinnittymisen jälkeen ajetaan componentDidMount-funktio, jossa voidaan esimerkiksi tehdä API-kutsuja. Elämänkaarifunktioista componentDidUpdate ajetaan heti, kun muutoksia havaitaan DOM-rakenteessa. Yleisin syy muutoksiin ovat ominaisuuksien tai tilan muutokset. Kun komponentti lopulta irtautuu, on componentWillUnmount -funktiossa hyvä hoitaa mahdolliset datan siistimiset. (Hamedani 2018a.)

3.4 DOM ja virtuaalinen DOM

DOM esittää sovelluksen käyttöliittymän puurakenteena. Puurakenteen ansiosta muutokset DOM-rakenteessa tapahtuvat nopeasti. Ongelmana kuitenkin on DOM-rakenteen muutoksien aiheuttama päivitetyn elementin sekä sen lapsielementtien uudelleenrenderöinti, joka hidastaa käyttöliittymän suorituskykyä merkittävästi. (Hamedani 2018b.)



Kuvio 6. Virtuaalisen ja oikean DOM-rakenteen vertailu (Hamedani 2018b)

React käyttää virtuaalista DOM-rakennetta korjatakseen tämän ongelman. Sen ideana on, että komponenttien muutokset päivitetään ensin virtuaaliseen DOM-rakenteeseen oikean DOM-rakenteen sijaan. Tämän jälkeen uutta virtuaalista DOM-rakennetta verrataan vanhaan virtuaaliseen DOM-rakenteeseen ja lasketaan niiden eroavaisuudet. Kun React tietää, mitkä komponentit ovat muuttuneet, se uudelleen renderöi vain ja ainoastaan nämä komponentit (ks. kuvio 6). Tämän ansiosta React-sovelluksen suorituskyky pysyy hyvänä myös monimutkaisissa ja laajoissa sovelluksissa. (Hamedani 2018b.)

3.5 Hooks

Hooks on Reactin uusin ominaisuus, joka julkaistiin Reactin versiossa 16.8. Uuden ominaisuuden tavoitteena on korjata monia erilaisia ongelmia, kuten esimerkiksi tilanhallintaan liittyviä haasteita. Hooks-ominaisuus on täysin taaksepäin yhteensopiva eikä sen käyttöönotto näin ollen ole pakollista. (Introducing Hooks n.d.)

React-sovelluksen komponentit muuttuvat nopeasti monimutkaisiksi tilallisiksi komponenteiksi, joiden uusiokäyttö on hankalaa. Esimerkiksi elämänkaarifunktiot, jotka

eivät ole aiemmin olleet käytettävissä funktiokomponenteissa, vaikeuttavat komponenttien uudelleenkäyttöä. Hooks-ominaisuus ratkaisee tämän ongelman tarjoamalla vaihtoehdon pilkkoa komponentit pienempiin funktioihin elämänkaarifunktioiden sijaan. (Hamedani 2019.)

Hooks-funktiot mahdollistavat sisäisen tilan ja elämänkaariominaisuuksien käyttämisen myös funktiokomponenteissa. Hooks-funktiot eivät toimi luokkakomponenteissa, mutta sen sijaan ne mahdollistavat Reactin käyttämisen ilman niitä. Kuviossa 7 nähdään, kuinka React-komponentin paikalliseen tilaan päästään käsiksi suoraan funktiokomponentin sisällä käyttäen tilakoukkua(useState). Tilan alustuksen lisäksi tilalle asetetaan toinen parametri (setGiraffeName), joka on funktio tilan uuden arvon asettamiselle. (Hamedani 2019.)

```
export function HookStateExample() {
    const [giraffeName, setGiraffeName] = useState('Kaapo')

    return (
        <div>
            <p>{giraffeName}</p>
            <button onClick={() => setGiraffeName('Kosti')}>
                Change name
            </button>
        </div>
    )
}
```

Kuvio 7. Hooks-esimerkki

Tilakoukun lisäksi Hooks tarjoaa myös esimerkiksi useEffect-funktion, jota voidaan käyttää perinteisten elämänkaarifunktioiden sijaan. Kuten elämänkaarifunktiot, useEffect ei estä selainta päivittämästä näyttöä, jonka ansiosta sovellus toimii saumattomammin. (Hamedani 2019.)

Tässä opinnäytetyössä erityisen oleellisia ovat myös Reactin tarjoamat valmiit useContext- ja useReducer-funktiot. Näistä ensimmäinen mahdollistaa kontekstin liittämisen funktiokomponenttiin ja toinen palauttaa reducer-funktiolla kuvatulle tilalle

muuttujan ja siihen yhdistetyn dispatch-metodin, jolla tilaa voidaan lähettää päivitystoimintoja. (Hooks API Reference n.d.)

Hooks-funktioita käytettäessä tulisi ottaa huomioon kaksi sääntöä. Ensinnäkin funktiota ei pitäisi kutsua silmukoissa, ehtolausekkeissa eikä sisäkkäisissä funktioissa. Tällä tavoin varmistetaan, että Hooks-funktiot kutsutaan aina samassa järjestyksessä, kun komponentti renderöidään. Toisena funktioita pitäisi aina kutsua pelkästään funktiokomponenteista tai muokatuista Hooks-funktioista, jotta tilalogiikka pysyy selkeästi näkyvillä. (Rules of Hooks n.d.)

3.6 Context

Context eli konteksti on suunniteltu sovelluksen globaalin datan jakamiseen. Tyypillisessä React-sovelluksessa data kulkee järjestelmällisesti vain yhteen suuntaan; ylhäältä alas. Se tekee esimerkiksi sovelluksen kieliasetusten jakamisen eri komponenteille hankalaksi. Konteksti mahdollistaa datan kulkemisen komponenttirakenteen läpi ilman, että dataa tarvitsee siirtää komponentin props-objektissa tasolta toiselle. (Context n.d.)

React tarjoaa valmiit rajapinnat kontekstin luomiselle sekä käyttämiselle. Uusi konteksti voidaan luoda helposti createContext-funktiolla ja sille voidaan asettaa tilan oletusarvo. Jokainen luotu konteksti sisältää valmiin Provider-komponentin, joka mahdollistaa kontekstin jakelun sen lapsikomponenteille. Provider-komponentille voidaan antaa props-objektissa arvoja, kuten tila tai toimintofunktioita, ja niitä voidaan käyttää kontekstiin liitetyissä komponenteissa. (Context n.d.)

Nykyinen tuotantokäyttöön suunniteltu konteksti julkaistiin Reactin versiossa 16.3. Sen edeltäjä (legacy context) oli vain kokeellinen ominaisuus ja se oli huomattavasti epävakaampi eikä sitä näin ollen ollut tarkoitettu käytettäväksi tuotannossa. (Leagacy Context n.d.)

4 Muut käytetyt tekniikat ja työkalut

4.1 JavaScript

JavaScript on ohjelmointikieli, joka sai alkunsa jo vuonna 1995, kun Brendan Eich loi sen. Sitä kutsuttiin aluksi nimellä LiveScript, mutta se vaihdettiin JavaScriptiksi markkinointisyistä, vaikka sillä ei olekaan mitään tekemistä Java-ohjelmointikielen kanssa. (Nutter 2017.)

JavaScript oli aluksi vain keino lisätä ohjelmia verkkosivuille Netscape Navigator -selaimessa, mutta ajan myötä lähes kaikki selaimet ovat alkaneet tukea sitä, ja se onkin mahdollistanut modernien web-sovellusten syntymisen. Kun useat selaimet alkoivat tukea JavaScriptiä, luotiin ECMAScript-standardi kuvaamaan, miten JavaScriptin kuuluisi toimia. (Haverbeke 2014, luku 1.)

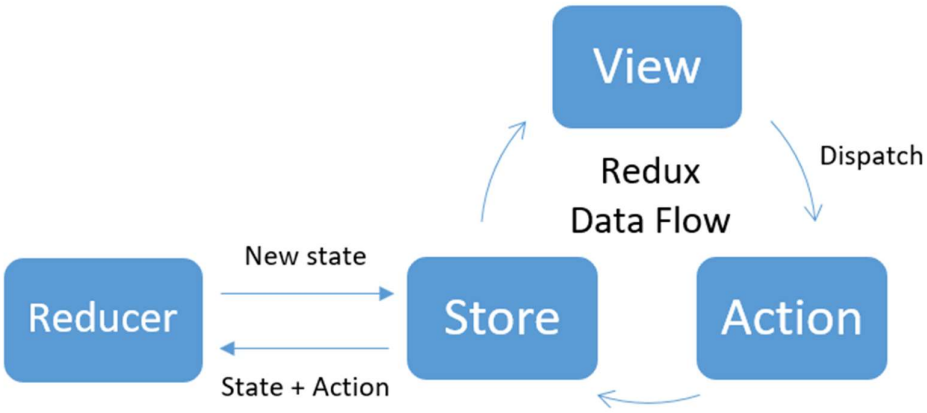
Joustavana ohjelmointikielenä JavaScript tuo mukanaan sekä hyötyjä että haittoja. Joustavuus mahdollistaa JavaScript-sovelluksissa sellaisten tekniikoiden käytön, jotka olisi mahdotonta toteuttaa tiukemmilla ohjelmointikielillä. Samalla kuitenkin virheiden löytämisestä tulee vaikeaa, koska järjestelmä ei usein pysty näyttämään niitä. (Haverbeke 2014, luku 1.)

4.2 Redux

Redux on Reactin tavoin avoimen lähdekoodin JavaScript-kirjasto, joka on suunniteltu sovelluksen globaalin datan jakamiseen kuten Reactin konteksti. Reduxia kutsutaan ennustavaksi tilanhallinnaksi, ja se on huomattavasti laajempi kokonaisuus kuin konteksti. Siitä on tullut erityisen suosittu React-sovelluksissa, mutta se toimii myös muiden JavaScript-sovellusten kanssa. (Paul & Nalwaya 2016, luku 7.)

Redux tarjoaa koko sovellukselle yhteisen tilasäilön(store), joka on kuvattu objektina. Komponentit voidaan liittää tilasäilöön riippumatta niiden sijainnista. Tilasäilön muuttamiseksi täytyy lähettää(dispatch) toiminto(action), joka on myös kuvattu tavallisena JavaScript objektina. Jokainen muutos on kuvattu toimintona, jonka vuoksi

sovelluksen tapahtumat pysyvät selkeinä. Jotta tilasäilö lopulta päivittyy, täytyy kutsua reducer-funktiota. Reducer-funktio ottaa sisään vanhan tilan sekä lähetetyn toiminnon ja palauttaa sovelluksen uuden tilan (ks. kuvio 8). (Core Concepts n.d.)



Kuvio 8 Reduxin datan kulku

Kolmeksi Reduxin pääperiaatteeksi voidaan nimetä seuraavat:

- Datalla vain yksi lähde
- Tilaa voi vain lukea
- Tilan muutokset tapahtuvat aina puhtailla funktioilla

Nämä tarkoittavat sitä, että ensinnäkin sovelluksen koko tilasäilö on kuvailtu yksittäisenä objektipuuna. Toisekseen ainoa tapa muokata sovelluksen tilaa ovat toiminnot, jotka kertovat, mitä tapahtui. Kolmanneksi tilan muutokset tapahtuvat lopulta aina puhtailla reducer-funktioilla. (Paul & Nalwaya 2016, luku 7.)

4.3 React Redux

React Redux on virallinen apukirjasto Reduxin käyttöön React-sovelluksessa. Sen tarkoituksena on niin sanotusti sitoa React- ja Redux-kirjastot yhteen, jolloin niiden käyttö on helpompaa. Lisäksi React Redux edistää hyvän React-arkkitehtuurin noudattamista tarjoamalla esimerkiksi valmiin connect-funktion, jolla komponentit voidaan yhdistää tilasäilöön. Tämä funktio huolehtii automaattisesti, mitä tilasäilön osaa käytetään, jolloin komponenttien ei itse tarvitse huolehtia siitä. Kirjasto hoitaa myös

sisäisesti suorituskyvyn optimointia, jolloin komponentit välttyvät turhilta renderöinneiltä. (Why Use React Redux? 2018.)

Provider-komponentti on React Redux-kirjaston tarjoama valmis komponentti, jolla sovelluksen muut komponentit kääritään. Näin Reduxilla luotu tilasäilö on muille sovelluksen komponenteille saatavilla ja ne voidaan liittää siihen connect-funktion avulla. (Quick Start 2019.)

4.4 React Router

Suosituin ratkaisu React-sovelluksen reititykselle on React Router. Se pitää käyttöliittymän ja URL:n synkronoituna liittämällä komponentit eri reitteihin. URL:n vaihtuessa käyttäjän toimesta reitteihin liitetyt komponentit kiinnittyvät ja irtautuvat automaattisesti. (de Sousa Antonio 2015, luku 5.)

React Router eroaa muista reitittimistä siten, että se on rakennettu JSX-elementeistä. Sovelluksen reitit määritellään kirjaston valmiilla Route-komponentilla, jolle annetaan vähintään reitti(path) ja komponentti(component) propsit. (Bagnardi, Fieldman, Hall & Højberg 2016, luku 15.)

4.5 Formik

Formik on pienehkö kirjasto Reactille. Sen tarkoituksena on helpottaa lomakkeiden rakentamista, joka voi olla työlästä ilman apukirjastoa. Formik seuraa automaattisesti lomakkeen tilan muutoksia ja tarjoaa valmiit metodit esimerkiksi lomakkeen lähettämisen käsittelylle. Lisäksi lomakkeen validoiminen ja virheviestien käsittely on helppoa Formikin avulla. (Overview n.d.)

Valmiit komponentit, joita Formik tarjoaa, käyttävät Reactin kontekstia päästäkseen käsiksi lomakkeen tilaan ja metodeihin. Näitä komponentteja ovat esimerkiksi lomakekentät (Field) ja lomakkeen virheviestit (ErrorMessage). (Overview n.d.)

5 Toteutus

5.1 Yleistä

Opinnäytetyön toteutus tehtiin rakentamalla ensin pohjalle referenssitoteutus toimeksiantajan sovelluksen käyttöliittymäkerroksesta. Referenssitoteutuksena toteutettiin yksinkertainen eläintenhallintasovellus, johon sisällytettiin vain oleellisimmat toiminnot opinnäytetyön aiheen kannalta. Tämän jälkeen pohjatoteutuksesta poistettiin Redux ja korvattiin se Reactin Hooks- ja konteksti -ominaisuuksilla. Lisäksi kaikki luokkakomponentit muutettiin funktiokomponenteiksi Hooks-ominaisuuden avulle. Lopuksi näitä sovelluksia verrattiin toisiinsa koodirivien määrän ja DOM-rakenteen selkeyden osalta.

5.2 Kehitysympäristö

5.2.1 Yleistä

Opinnäytetyön toteutus aloitettiin kehitysympäristön pystyttämisellä ja tarvittavien moduulien asentamisella. Sovelluksen kehittämiseen käytetty IDE oli Visual Studio Code, joka on Microsoftin kehittämä kevyt koodieditori. Se tukee monia eri ohjelmointikieliä ja teknologioita laajennusten avulla, joita voidaan lisätä editoriin helposti suoraan sen käyttöliittymältä. (Getting started n.d.)

Visual Studio Code tarjoaa myös sisäänrakennetun terminaalin, jota käytettiin projektin pakettien asentamiseen sekä projektin ajamiseen. Kuviossa 9 nähdään projektin package.json -tiedosto, jossa on listattu kaikki projektissa käytetyt moduulit sekä kaksi apuskriptiä. Apuskriptejä voidaan ajaa komennolla

```
npm run <skriptin_nimi>
```

```
{
  "scripts": {
    "build": "webpack --mode production",
    "start": "webpack-dev-server --mode development --hot"
  },
  "devDependencies": {
    "@babel/core": "7.4.0",
    "@babel/preset-env": "7.4.2",
    "@babel/preset-react": "7.0.0",
    "babel-loader": "8.0.5",
    "eslint": "5.15.3",
    "eslint-config-prettier": "4.1.0",
    "eslint-plugin-react": "7.12.4",
    "prettier": "1.16.4",
    "webpack": "4.29.6",
    "webpack-cli": "3.3.0",
    "webpack-dev-server": "3.2.1"
  },
  "dependencies": {
    "formik": "^1.5.2",
    "react": "16.8.5",
    "react-dom": "16.8.5",
    "react-hot-loader": "4.8.0",
    "react-redux": "7.0.2",
    "react-router-dom": "5.0.0",
    "redux": "4.0.1"
  }
}
```

Kuvio 9 Package.json-tiedosto

Moduulit on listattu kahteen eri luokkaan: dependencies ja devDependencies (ks. kuvio 9). Nämä eroavat toisistaan siten, että toiseksi mainitun luokan riippuvuudet asennetaan vain tietokoneelle, jolla sovelluksen kehitys tapahtuu eli niitä ei tarvitse asentaa tuotannossa ollenkaan. Paketit voidaan asentaa komennolla

```
npm install --save <paketin_nimi>
```

tai

```
npm install --save-dev <paketin_nimi>
```

riippuen siitä, kumpaan luokkaan paketti halutaan lisätä.

5.2.2 Webpack

Perinteisesti JavaScript-koodi on pilkottu useisiin eri tiedostoihin, jotka on sisällytetty yksitellen niitä käyttäviin HTML-tiedostoihin. Tämä on kuitenkin työlästä ja laajemmissa projekteissa lähes mahdotonta ylläpitää. (Subramanian 2017, luku 7.)

Webpack on tarkoitettu tämän ongelman korjaamiseen. Sen tarkoitus on kerätä kaikki sovelluksessa käytetyt JavaScript-moduulit automaattisesti yhteen oikeassa järjestyksessä ja sen jälkeen koota ne yhdeksi isoksi tiedostoksi, joka voidaan sitten sisällyttää HTML-tiedostoon. (Subramanian 2017, luku 7.)

Myös Webpack käyttää konfiguraatiotiedostoa, jossa voidaan määritellä sille säännöt. Konfiguraatiotiedosto ei ole pakollinen Webpackin versiosta 4.0.0 lähtien, mutta sillä voidaan varmistaa, että se toimii projektille sopivalla tavalla. (Concepts n.d.)

Tässä projektissa käytettiin myös Webpackin tarjoamaa DevServeriä, jonka avulla sovelluksen kehittäminen on nopeampaa. Webpack DevServer tarjoaa yksinkertaisen web serverin, joka kääntää kirjoitetun koodin automaattisesti havaittuaan muutoksia koodissa ja päivittää sen livenä selaimeen. (Development n.d.)

5.2.3 Babel

Koska Webpack pystyy kokoamaan pelkästään JavaScript-tiedostoja, tarvitaan sen tueksi tässä projektissa moduuli, joka kääntää JSX-tiedostot JavaScriptiksi. Babel-loader-moduuli ratkaisee ongelman. (Subramanian 2017, luku 7.)

Babel on JavaScript-kääntäjä. Sitä käytetään pääasiassa uudemman JavaScriptin kääntämisessä muotoon, joka on yhteensopiva vanhempien JavaScriptin versioiden kanssa, jotta kaikki selaimet ymmärtävät sitä. Lisäksi Babel pystyy myös kääntämään JSX-syntaksin puhtaaksi JavaScript koodiksi, joka onkin tässä projektissa oleellisempi ominaisuus. (What is Babel? n.d.)

5.2.4 ESLint ja Prettier

ESLint on yksinkertainen työkalu koodin syntaksin tarkistamiseen. Sen tehtävänä on ilmoittaa mahdollisista virheistä koodissa tai jos sille asetettuja sääntöjä rikotaan. ESLint on joustava syntaksin tarkistaja, sillä sille voi asettaa itse säännöt, joita haluaa

projektissa noudattaa. Säännöt asetetaan erillisessä konfiguraatiotiedostossa. (Subramanian 2017, luku 7.)

Prettier on suosittu työkalu koodin automaattiseen muotoiluun. Se muotoilee koodin editorissa siten, ettei se vaikuta sen syntaksiin. Myös Prettier ottaa vastaan konfiguraatiotiedoston, johon voidaan asettaa projektille valitut säännöt koodin muotoilulle. (What is Prettier? n.d.)

Yhteiset säännöt syntaksin tarkistamiselle ja koodin muotoilulle esimerkiksi tiimin sisällä ovat tärkeitä, jotta koodin yhdenmukaisuus säilyy projektin sisällä. ESLint ja Prettier yhdessä tarjoavat automatisoinnin näiden sääntöjen noudattamisen seuraamiselle. (Subramanian 2017, luku 7; Why Prettier? n.d). Tässä työssä ESLint oli käytössä kuitenkin lähinnä virheiden tarkistuksen osalta ja Prettier selkeämmän editorinäkymän osalta.

5.3 React-sovellus Redux-tilanhallinnalla

5.3.1 Yleistä

Referenssitoteutuksen rakentaminen aloitettiin tunnistamalla komponentit ja ominaisuudet, jotka ovat oleellisia opinnäytetyön tavoitteen kannalta. Globaalin Redux-tilanhallinnan lisäksi sovellukseen otettiin mukaan myös paikallista tilaa sekä elämänkaarifunktiota käyttäviä komponentteja, jotta Reactin uutta Hooks-ominaisuutta pystyttiin esittelemään ja vertailemaan laajemmin.

Tarvittavien komponenttien ja ominaisuuksien kartoituksen jälkeen tehtiin sovellukseen ensimmäisenä juurikomponentti. Juurikomponentti voitiin toteuttaa yksinkertaisena funktiokomponenttina, sillä sen sisällä ei tarvita paikallista tilaa tai elämänkaarifunktioita (ks. kuvio 10).

```
export const Root = () => (
  <Provider store={store}>
    <BrowserRouter>
      <Header />
      <Grid>
        <Route exact path="/" component={LandingPage} />
        <Route path="/giraffes" component={GiraffeList} />
        <Route path="/create" component={CreateGiraffe} />
      </Grid>
    </BrowserRouter>
  </Provider>
)
```

Kuvio 10. Sovelluksen juurikomponentti

Tässä tapauksessa juurikomponentista selviää koko sovelluksen päärakenne. Kuviosta 10 nähdään, että uloimpana komponenttina on React Redux-kirjaston tarjoama Provider-komponentti, joka mahdollistaa kaikkien sen sisällä olevien komponenttien liittymisen globaaliin Redux-tilaan ilman props-objektin kuljettamista läpi sovelluksen. Toiseksi uloin komponentti on React Router-kirjastosta löytyvä BrowserRouter, jota käytettiin sovelluksen sivujen väliseen reititykseen. Sen sisältä löytyy kaikkialla näkyvissä oleva Header-komponentti sekä kaikki sovelluksen sisältämät reitit. Jokaiselle reitille määriteltiin polun lisäksi komponentti, joka renderöitiin käyttäjän saapuessa kyseiseen polkuun. Juurikomponentin rakentamisen jälkeen lähdettiin toteuttamaan globaalia tilanhallintaa sekä juurikomponentissa määriteltyjen reittien tarvitsemia komponentteja.

Referenssitoteutukseen toteutettiin yksinkertaistettu realistinen komponenttirakenne. Sovellukseen ei sisällytetty tyyliteltyä ulkoasua, mutta pelkistetyt Grid-, Row- ja Col-komponentit luotiin pitämään elementtien keskinäistä järjestystä yllä. Lisäksi aina näkyvillä olevaan Header-komponenttiin luotiin navigaatio sovelluksessa liikkumista varten. Etusivuksi tehtiin LandingPage-komponentti, joka toimi tässä tapauksessa vain niin sanottuna päävalikkona. Navigaation ja päävalikon linkitykset sovelluksen sivujen välillä hoidettiin React Router-kirjaston tarjoamalla Link-komponentilla tai withRouter-ominaisuudella. Kaikki edellä mainitut sovellukseen luodut komponentit pystyttiin toteuttamaan funktiokomponentteina niiden yksinkertaisuuden vuoksi.

5.3.2 Redux-tilanhallinta

Edellisessä luvussa esitelty Provider-komponentti tarvitsee store-objektin, joka kuvailee globaalin tilasäilön muodon. Objekti luotiin Redux-kirjaston tarjoamalla valmiilla createStore-funktiolla, jolle annettiin parametriksi sovelluksen RootReducer-funktio (ks. kuvio 11).

```
export const RootReducer = Redux.combineReducers({
  giraffes: GiraffesReducer,
  notifications: NotificationsReducer
})

export const store = Redux.createStore(RootReducer)
```

Kuvio 11. Redux-tilasäilön luonti

Kuviossa 11 nähdään, että sovelluksen tilasäilölle annettiin reducer-funktio (RootReducer), jolla koottiin yhteen kaksi pienempää reducer-funktiota Reduxin combineReducers-funktion avulla. Tilasäilö jaettiin kahteen pienempään osioon, koska kumpikin osio hallitsee vain omaa osuuttaan tilasta ja näin ollen niiden jakaminen selkeytti niiden hallintaa sekä koodin luettavuutta.

Pienempiin reducer-funktioihin määriteltiin tilasäilön eri osioiden päivittämiseen tarvitut toimenpiteet. Kuviossa 12 näkyvät kirahvien tilasäilön osuuden päivitykseen toteutetut toimenpiteet, jotka suoritettiin aina toiminnon tyypin mukaan. Mikäli reducer-funktiolle lähetettiin tuntematon toiminto, palautettiin sieltä aina muokkaamaton tila takaisin, jotta virheitä ei pääsisi syntymään.

```
export function GiraffesReducer(state = giraffesInitialState, action) {
  switch (action.type) {
    case ACTIONS.INSERT_GIRAFFE:
      return state.concat(action.payload)
    case ACTIONS.DELETE_GIRAFFE:
      return state.filter(giraffe => giraffe.id !== action.payload.id)
    default:
      return state
  }
}
```

Kuvio 12. Reducer-funktio

Toiminnot(actions), jotka johtavat reducer-funktiossa kuvattuihin tilan päivitystoimenpiteisiin, luotiin erilliseen tiedostoon. Kuviossa 13 nähdään, miltä yksittäisen kirahvin poistotoiminto näyttää. Toimintofunktiosta palautettiin toiminnon tyyppi, joka ohjasi oikeaan päivitystoimintoon sekä payload-objekti, jossa vietiin poistettavan kirahvin tunniste(id) päivitystoiminnolle. Tässä tilanteessa toiminnossa ei tehty mitään sille annetulle arvolle vaan se vietiin sellaisenaan päivitystoimintoon. Reaalimaailman sovelluksessa arvolla voitaisiin kuitenkin hakea esimerkiksi dataa rajapinnan kautta tietokannasta.

```
export function deleteGiraffe(id) {
  return { type: ACTIONS.DELETE_GIRAFFE, payload: { id } }
}
```

Kuvio 13. Reducer-funktiolle lähetettävä toiminto

Jotta Redux-tilanhallinta saatiin lopulta käyttöön sovelluksessa, luotiin komponentti, joka liitettiin tilasäilön sisältöön sekä tarvittaviin päivitystoimintoihin. Kuviossa 14 näkyy, kuinka GiraffeListComponent-komponenttin props-objektiin on liitetty Redux-tilasäilössä sijaitsevat giraffes- ja notifications-objektit sekä onDelete-funktio, joka lähettää eteenpäin deleteGiraffe- ja createNotification-toiminnot.

```
const mapDispatchToProps = dispatch => {
  return {
    onDelete: (id, name) => {
      dispatch(deleteGiraffe(id)),
        dispatch(
          createNotification(id, "warning", `${name}-kirahvi poistettu!`)
        )
    }
  }
}
const mapStateToProps = state => {
  return {
    giraffes: state.giraffes,
    notifications: state.notifications
  }
}
export const GiraffeList = connect(
  mapStateToProps,
  mapDispatchToProps
)(GiraffeListComponent)
```

Kuvio 14. Komponentin liittäminen Redux-tilasäilöön

React Redux-kirjaston tarjoamalla connect-funktiolla luotiin GiraffeList-komponentti, joka on niin sanottu HOC-komponentti. Connect-funktiolle annettiin mapStateToProps- ja mapDispatchToProps-funktiot, joilla saatiin kiinnitettyä tilasäilön sisältö sekä lähetettävät toiminnot GiraffeListComponent-komponentin props-objektiin (ks. kuvio 14).

5.3.3 Listakomponentti

Kuviossa 15 nähdään edellisessä luvussa Redux-tilasäilöön liitetty GiraffeListComponent-komponentti, joka toteutettiin luokkakomponenttina. Komponentin tarkoituksena oli listata globaalista tilasta löytyvät kirahvit, näyttää niiden tarkemmat tiedot yksittäin sekä mahdollistaa niiden poistaminen tilasäilöstä käyttäjän toimesta. Lisäksi komponentin sisälle renderöitiin notification-komponentteja, jos niitä oli globaalissa tilassa olemassa. Luokan constructor-metodissa alustettiin komponentin paikalliseen tilaobjektiin määrittelemätön(undefined) giraffeInfo-muuttuja, jota käytettiin yksittäisen kirahvin tietojen näyttämiseen listauksen vierellä.

```
class GiraffeListComponent extends React.Component {
  constructor(props) {
    super(props)
    this.state = { giraffeInfo: undefined }
  }
  render() {
    return (
      <Row>
        <Col>
          <table>
            <thead>···
            <tbody>
              {this.props.giraffes.map((giraffe, index) => (
                <tr key={giraffe.id}>···
              ))}
            </tbody>
          </table>
        </Col>
        {this.state.giraffeInfo && (
          <Col>
            <GiraffeInfo
              giraffe={this.state.giraffeInfo}
              onHide={() => this.setState({ giraffeInfo: undefined })}
            />
          </Col>
        )}
        {this.props.notifications.map(notification => (
          <Notification
            id={notification.id}···
            key={notification.id}
          />
        ))}
      </Row>
    )
  }
}
```

Kuvio 15. GiraffeListComponent-luokkakomponentti

Mikäli listakomponentin paikallisesta tilasta löytyi kirahvin tiedot, renderöitiin GiraffeInfo-komponentti, joka oli yksinkertainen funktiokomponentti props-objektin sisällön esittämiseen. Tälle komponentille annettiin props-objektissa tilasta löytyvän kirahvin tiedot sekä funktio, jolla tarkemmat tiedot saatiin taas pois näkyvistä asettamalla GiraffeList-komponentin paikallisen tilan giraffeInfo-muuttuja takaisin määrittelemättömäksi (ks. kuvio 15).

Komponentista löytyvään taulukkoelementtiin(table) listattiin JavaScriptin map-funktion avulla kaikki sovelluksen tilasäilöstä löytyvät kirahvit. Jokaisella taulukon rivillä näytettiin kirahvin nimi sekä rivinumero. Lisäksi riveille luotiin painikkeet kirahvien tietojen näyttämiselle ja poistamiselle. Kirahvin tiedot asetettiin nappulaa painetta-

essa komponentin paikalliseen tilaan, kun taas poistonappi lähetti kirahvin poistotoiminnon reducer-funktiolle aiheuttaen globaalin tilan päivittymisen ja komponentin uudelleen renderöitymisen. (ks. kuvio 16).

```
<td>
  <button onClick={() => { this.setState({ giraffeInfo: giraffe }) }}>
    Näytä
  </button>
</td>
<td>
  <button onClick={() => { this.props.onDelete(giraffe.id, giraffe.name) }}>
    Poista
  </button>
</td>
```

Kuvio 16. Paikallisen ja globaalin tilan muokkaaminen

5.3.4 Lomakekomponentti

Kirahvin lisäys globaaliin tilasäilöön toteutettiin pienellä lomakkeella, joka luotiin Formik-kirjaston tarjoamien komponenttien avulla. Lomakkeelle asetettiin erilliset kentät kirahvin tiedoille sekä nappula lomakkeen lähettämiselle, jota painettaessa lähetettiin lomakkeelle syötetyt tiedot reducer-funktiolle, joka taas päivitti tiedot globaaliin tilaan. Lisäyksen jälkeen käyttäjä ohjattiin takaisin kirahvien listaukseen.

CreateGiraffeComponent-komponentti liitettiin globaaliin tilaan lähes samoin kuin edellisessä luvussa esitelty listakomponentti. Ainoana erona oli, että tässä tilanteessa mapStateToProps-funktion arvo asetettiin määrittelemättömäksi. Tämä johtui siitä, ettei lomakkeella ollut tarpeellista näyttää tilasäilössä jo olevia tietoja vaan tarkoituksena oli ainoastaan lisätä niitä.

5.3.5 Notifikaatiokomponentti

Viimeinen oleellinen komponentti, joka referenssisovellukseen toteutettiin, oli NotificationComponent. Sen tarkoituksena oli tuoda käyttäjälle näkyviin mahdollisia ilmoituksia sovelluksen tilasta. Tässä sovelluksessa ilmoitukset koskivat joko kirahvin lisää-

mistä tai sen poistoa. Komponentti toteutettiin luokkakomponentin elämänkaarifunktioiden avulla. Ideana oli, että komponentti näkyy käyttäjälle vain muutaman sekunnin ajan, jonka jälkeen se poistuu (ks. kuvio 17).

```
class NotificationComponent extends React.Component {
  componentDidMount() {
    this.notificationTimer = setInterval(
      () => this.props.onHide(this.props.id),
      4000
    )
  }
  componentWillUnmount() {
    clearInterval(this.notificationTimer)
  }
  render() {
    return (
      <div ···
    )
  }
}
```

Kuvio 17. Luokkakomponentin elämänkaarifunktiot

Kuten lista- ja lomakekomponenteista, myös notifikaatiokomponentista täytyi tehdä React Redux-kirjaston connect-funktiolla HOC-komponentti, jotta se saatiin liitettyä globaaliin tilaan. Näin ilmoitukset saatiin poistettua tilasäilöstä sekä näkyvistä elämänkaarifunktiossa asetetun aikarajan tullessa täyteen.

5.4 React-sovellus ilman Reduxia

5.4.1 Yleistä

Uuden Hook-ominaisuuden mahdollistaessa paikallisen tilan sekä elämänkaarifunktioiden käytön myös funktiokomponenteissa ja kokonaan luokkakomponenteista luopumisen, voitiin kaikki sovelluksen komponentit toteuttaa funktiokomponentteina muuttamatta sovelluksen rakennetta. Jotta rakenne saatiin pysymään mahdollisimman muuttamattomana, otettiin Reactin uusien ominaisuuksien käyttöönottoa varten pohjalle kopio referenssitoteutuksesta.

Toteutus aloitettiin poistamalla sovelluksesta Redux- ja React Redux-kirjastot komennolla:

```
npm uninstall --save redux react-redux
```

Sovelluksen juurikomponentin rakenne säilytettiin entisellään. Ainoastaan React Redux-kirjaston myötä poistunut Provider-komponentti korvattiin uudella vastaavalla komponentilla, joka esitellään seuraavassa luvussa.

5.4.2 Tilanhallinta ilman Reduxia

Uutta tilanhallintaa lähdettiin rakentamaan siltä pohjalta, että sen rakenne ja datan kulku saataisiin pidettyä mahdollisimman samankaltaisena Redux-tilanhallintaan nähden. Siispä myös toiminnot ja reducer-funktiot pyrittiin pitämään mahdollisimman muuttumattomina.

Redux- ja React Redux-kirjastot tarjosivat monia valmiita funktioita ja komponentteja globaalin tilanhallinnan rakentamiseen. Niiden poistumisen myötä puuttuvat palaset täytyi toteuttaa muilla tavoilla. Referenssitoteutuksessa käytetty Provider-komponentti korvattiin uudella Provider-komponentilla, joka rakennettiin Reactin konteksstia sekä uutta Hooks-ominaisuutta käyttäen (ks. kuvio 18).

```
export const Context = createContext(initialState)
export const Provider = props => {
  const [state, dispatch] = useReducer(RootReducer, initialState)
  const actions = getActions(dispatch)
  return (
    <Context.Provider value={{ state, actions }}>
      {props.children}
    </Context.Provider>
  )
}
```

Kuvio 18. Provider-komponentti ilman Reduxia

Kuviossa 18 nähdään kuinka tilasäilönä käytetty konteksti luotiin Reactin createContext-funktiolla ja se alustettiin initialState-objektilla, jolla kuvattiin sovelluksen ensimmäinen tila. Globaali tila ja siihen paritettu dispatch-metodi saatiin käyttöön Reactin

uudella useReducer-metodilla, joka ottaa vastaan reducer-funktion sekä sovelluksen ensimmäisen tilan. Toiminnot saatiin käyttöön getActions-funktiolla, joka keräsi sovelluksen kaikki toiminnot yhteen ja palautti ne. Tila ja toiminnot annettiin kontekstin valmiille Provider-komponentille, jotta sovelluksen muut komponentit voivat käyttää niitä.

Sovelluksen reducer-funktiot sekä toiminnot pystyttiin pitämään pääpiirteiltään samoina. RootReducer-funktiossa käytetty Redux-kirjaston tarjoama valmis combineReducers-funktio, joka yhdisti referenssisovelluksen pienemmät reducer-funktiot, täytyi kuitenkin kirjoittaa auki. Se tarkoitti käytännössä sitä, että pienemmille reducer-funktioille määritettiin erikseen parametreissa oikea lohko tilasäilöstä toiminnon lisäksi (ks. kuvio 19).

```
export const RootReducer = (state, action) => ({
  giraffes: GiraffesReducer(state.giraffes, action),
  notifications: NotificationsReducer(state.notifications, action)
})
```

Kuvio 19. RootReducer-funktio ilman Reduxia

Toiminnot saatiin sovelluksen käyttöön palauttamalla ne funktiosta, jolle annettiin parametrina dispatch-metodi. Referenssisovelluksesta poiketen, dispatch-metodi sisällytettiin toimintoihin (ks. kuvio 20). Tällä tavoin komponentit pystyivät päivittämään tilasäilöä pelkän toimintofunktion avulla ja toiminnot pystyivät kutsumaan toisiaan suoraan.

```
function deleteGiraffe(id, name) {
  dispatch({ type: ACTIONS.DELETE_GIRAFFE, payload: { id } })
  createNotification(id, "warning", `${name}-kirahvi poistettu!`)
}
```

Kuvio 20. Dispatch-metodi toiminnon sisällä

Komponenteissa tilanhallintaan päästiin käsiksi Reactin uudella useContext-metodilla, jolle annettiin sovellukselle aiemmin luotu konteksti. Kuviossa 21 nähdään,

kuinka aiemmin Provider-komponentille annettu actions-arvo saadaan suoraan asetettua funktiokomponentin käyttöön.

```
export const CreateGiraffe = withRouter(props => {
  const { actions } = useContext(Context)
  return (
    <Formik
      initialValues={{ giraffeName: "", giraffeDescription: "" }}
      onSubmit={values => (
        actions.createGiraffe(
          createId(),
          values.giraffeName,
          values.giraffeDescription
        ),
        props.history.push("/giraffes")
      )}
      render={props => (
        <Form>···
      )}
    />
  )
})
```

Kuvio 21. Kontekstin ja toimintojen käyttö komponentissa

Kun tilanhallinnan toiminnot oli liitetty komponenttiin useContext-metodilla, voitiin niitä kutsua suoraan komponentin sisältä (ks. kuvio 21). Mikäli dispatch-funktiota ei olisi sisällytetty toimintoihin, olisi se pitänyt asettaa myös yhdeksi Provider-komponentin arvoksi sekä liittää erikseen komponenttiin. Myös tilasäilön sisältö saadaan liitettyä komponenttiin samalla tavalla kuin toiminnot.

5.4.3 Luokkakomponentista funktiokomponentiksi

Alkuperäisessä referenssisovelluksessa paikallinen tila toteutettiin listakomponenttiin perinteisellä tavalla Reactin luokkakomponentissa. Hooks-ominaisuudella paikallinen tila saadaan kuitenkin käyttöön myös funktiokomponentissa.

Ensin GiraffeList-komponentista poistettiin constructor-funktio, koska funktiokomponentissa sitä ei tarvita. Lisäksi render-metodista jätettiin vain palautettava rakenne jäljelle. Lopuksi luokkakomponentti muutettiin funktioksi. Paikallinen tila saatiin

Reactin tarjoaman useState-metodin avulla komponentin käyttöön. Metodille annettiin parametrina määrittelemätön tilanalustus ja se palautti paikallisen tilan arvon muuttujana sekä funktion tilan arvon päivittämistä varten (ks. kuvio 22).

```
export const GiraffeList = () => {
  const { state, actions } = useContext(Context)
  const [giraffeInfo, setGiraffeInfo] = useState(undefined)
  return (
    <Row>···
  )
}
```

Kuvio 22. Paikallinen tila funktiokomponentissa

Paikallisen tilamuuttujan näyttäminen ja päivittäminen hoidettiin Hooks-ominaisuudella samoin kuin luokkakomponentissa. Ainoana erona oli funktiokomponentissa erikseen nimetty funktio uuden tilan asettamiseen luokkakomponentissa käytetyn perinteisen setState-funktion sijaan.

5.4.4 Elämänkaarifunktiot funktiokomponentissa

Myös elämänkaarifunktioita käyttänyt notifikaatiokomponentti saatiin muutettua funktiokomponentiksi Reactin uuden useEffects-funktion avulla. Komponentista poistettiin componentDidMount- ja componentWillUnmount-funktiot ja niiden toiminta siirrettiin kokonaisuudessaan useEffect-funktion sisälle (ks. kuvio 23). Lisäksi React Redux-kirjaston connect-funktio poistettiin, jolloin komponentti lakkasi olemasta HOC-komponentti.

```
export const Notification = props => {
  const { actions } = useContext(Context)
  useEffect(() => {
    const notificationTimer = setInterval(
      () => actions.dismissNotification(props.id),
      4000
    )
    return function cleanUp() {
      clearInterval(notificationTimer)
    }
  })
  return (
    <div className={`notification ${props.className ? props.className : ""}`}>
      {props.message}
    </div>
  )
}
```

Kuvio 23. useEffect-funktio komponentissa

Komponentin sisällä kutsuttava useEffect-funktio ajettiin aina, kun komponentti päivittyi eli sen sisällä voitiin hoitaa componentDidMount-funktion toiminnallisuus. Vastaavasti componentDidUnmount-funktiota vastaava toiminta saatiin useEffect-funktiosta palautettavaan cleanUp-funktioon, joka ajettiin aina komponentin poistuessa DOM-puusta.

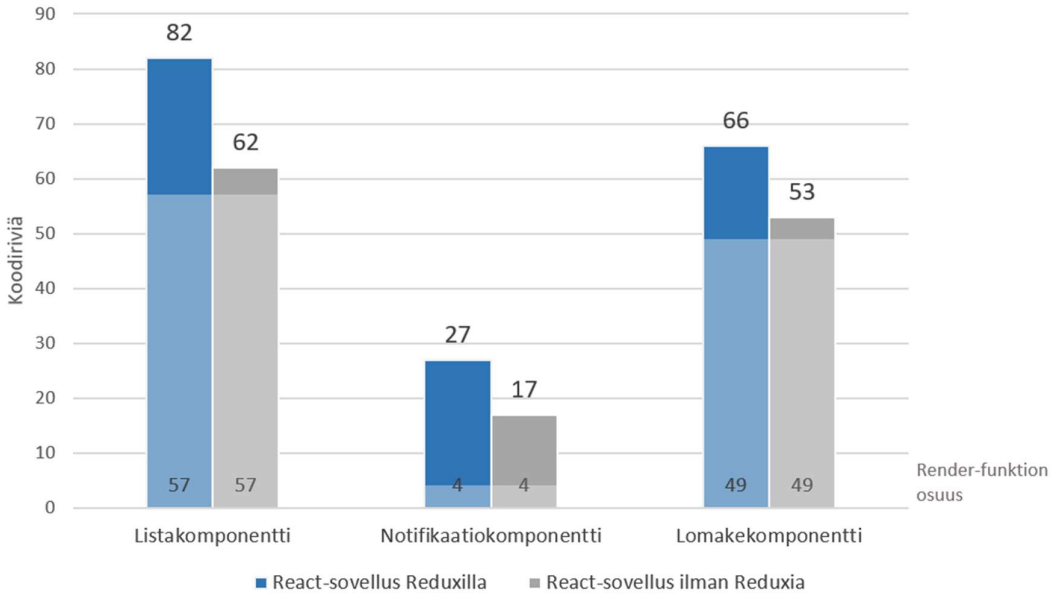
6 Tulokset

Opinnäytetyön aikana toteutettiin kaksi React-sovellusta, joiden ominaisuudet ja rakenne pyrittiin pitämään mahdollisimman samanlaisena. Toisessa sovelluksista käytettiin Redux-tilanhallintaa ja toisessa sovelluksessa todistettiin, että globaali tilanhallinta on teknisesti mahdollista rakentaa pelkällä Reactilla sen uusien ominaisuuksien avulla. Voitiin siis todeta, että uudet ominaisuudet ratkaisivat Reactin tunnistetut tilanhallintaongelmat. Toteutettujen sovellusten koodia sekä DOM-rakenteita vertailtiin keskenään, jotta nähtiin, selkeytyykö käyttöliittymäkerros Reduxista luopumisen myötä.

6.1 Vertailu

Sovelluksia vertailtiin erilaisten komponenttien koodirivien määrän sekä DOM-rakenteen syvyyden perusteella. Lisäksi tarkasteltiin, onko sovelluksien renderöintiajoissa eroavaisuuksia.

Koodirivien määrässä oli huomattavia eroja sovellusten välillä. Kuviosta 23 huomataan, että Reduxista luopuminen vähensi keskimäärin 14,3 koodiriviä komponenttia kohden. Mikäli renderöintifunktion sisältö otettiin huomioon, koodirivien määrä väheni keskimäärin noin 26,9 prosenttia. Vastaava luku oli 66,7 prosenttia, jos renderöintifunktion sisältöä ei otettu huomioon. Tästä voidaan päätellä, että mitä pienempi komponentti on kyseessä, sitä suurempi vaikutus koodirivien kannalta tilanhallinnan ratkaisulla on. Monimutkaisemmassa komponentissa vaikutus on prosentuaalisesti pienempi.



Kuvio 24. Koodirivien määrät eri komponenteissa

DOM-rakenteissa näkyvät eroavaisuudet olivat tässä tapauksessa melko pieniä. Jokainen Redux-tilanhallintaan liitetty komponentti oli niin sanottu HOC-komponentti,

jonka takia itse komponentti luotiin DOM-puussa yksi tai kaksi tasoa syvemmälle riippuen siitä käytettiinkö komponentissa pelkästään tilanhallinnan toimintoja vai oliko myös tilasäilön sisältö käytössä. Monimutkaisemmassa sovelluksessa, jossa tilanhallintaan liitettyjä komponentteja on paljon eri tasoilla, DOM-rakenne voikin muuttua nopeasti todella sekavaksi.

Sovellusten renderöintiaikaa testattiin selaimen suorituskykytyökaluilla eikä sovellusten välillä havaittu eroavaisuuksia. Molemmat sovellukset pystyttiin renderöimään noin kahdessakymmenessä millisekunnissa.

6.2 Hyödyt ja haitat

Reactin uudet ominaisuudet toivat mukanaan hyödyllisiä ratkaisuja Reactin tilanhallintaongelmiin. Esimerkiksi Hooks-ominaisuuden myötä tullut mahdollisuus lisätä myös funktiokomponentille paikallinen tila sekä elämänkaarifunktioita, tekee funktiokomponenteista yhä monikäyttöisempiä, jolloin tarve komponenttien uudelleenkirjoitukseen vähenee muutoksia tehdessä ja komponenteista voidaan tehdä monikäyttöisempiä.

Uudistunut konteksti mahdollistaa erilaisten tilanhallintaratkaisuiden toteuttamisen sovelluksen tarpeiden mukaan. Yhdessä Hooks-ominaisuuden kanssa kontekstilla voidaan rakentaa tarvittaessa myös Reduxin kaltainen tilanhallinta yksinkertaisemmassa muodossa, jolloin sovelluksen riippuvuuksia sekä komponenttien koodirivejä saadaan vähennettyä.

Haittapuolena Reduxista luopumisessa on sen tarjoamien apufunktioiden ja valmiiden komponenttien poistuminen käytöstä, jolloin tilanhallinnan rakentaminen on hieman vaikeampaa. Lisäksi esimerkiksi Reduxin tarjoama middleware-ominaisuus ei ole enää käytettävissä. Sen avulla voidaan luoda esimerkiksi asynkronisia API-kutsuja tai funktioita, jotka kirjoittavat lokiin automaattisesti lähetettyjen päivitystoimintojen tietoja tai virheilmoituksia.

7 Loppupohdinta

7.1 Tavoitteet

Opinnäytetyön päätavoitteena oli selvittää, onko Reactin uusilla ominaisuuksilla mahdollista rakentaa keskikokoiseen React-sovellukseen globaali tilanhallinta, joka olisi yksinkertaisempi kuin Redux-kirjaston tarjoama tilanhallinta. Tavoitteen tarkoituksena oli yksinkertaistaa pienempien ja keskikokoisten React-sovellusten käyttöliittymäkerrosta. Lisäksi tavoitteena oli kartoittaa uusien ominaisuuksien hyötyjä ja haittoja sekä vertailla niitä korvattuihin ominaisuuksiin.

Tavoitteet olivat alusta asti tekijän mielestä realistiset ja tarkoituksenmukaiset. Tavoitteet myös pystyttiin työn aikana saavuttamaan. Erityisesti päätavoitteen saavuttaminen ja todentaminen teknisesti onnistui hyvin.

7.2 Tulokset

Selvityksen myötä voidaan todeta, että yksinkertainen globaali tilanhallinta voitiin toteuttaa täysin Reactin uusilla ominaisuuksilla. Vaikka Reactin uusien ominaisuuksien avulla toteutettu tilanhallinta olikin monimutkaisempi rakentaa, oli komponenttien liittäminen siihen huomattavasti helpompaa ja vaivattomampaa. Näin ollen pienen tai keskikokoisen React-sovelluksen käyttöliittymäkerrosta voitaisiin yksinkertaistaa sovelluksen riippuvuuksien, koodirivien ja DOM-rakenteen osalta. Lisäksi selvityksessä voitiin todentaa, että luokkakomponenteista kokonaan luopuminen on Hooks-ominaisuuden myötä mahdollista, jolloin koodia voidaan myös yhtenäistää siirtymällä käyttämään pelkkiä funktiokomponentteja. Tämän myötä komponentit olisivat myös helpommin uudelleenkäytettäviä.

Työn tulokset olivat pääasiassa positiivisia, mutta niitä tarkastellessa täytyy ottaa huomioon, että työssä toteutetut sovellukset olivat erittäin yksinkertaisia. Tästä syystä tulisi myös huomioida, että Reactin uusien ominaisuuksien mahdollisia vaikutuksia sovelluksen suorituskykyyn ei pystytty realistisesti opinnäytetyössä testaamaan, joten suorituskykyä olisi kannattavaa testata erityisesti ennen mahdollista

käyttöönottoa esimerkiksi tuotannossa. Suorituskyvyn lisäksi työstä jätettiin tutkimuksen aihetta rajatessa pois komponenttien yksikkötestaaminen, joka on kuitenkin tärkeä osa sovellusten kehitystyötä ja olisi tuonut lisäarvoa työn tuloksille. Jatkokehityksen kannalta, olisikin oleellista selvittää miten Hook-ominaisuutta ja kontekstia käyttävien komponenttien yksikkötestausta käytännössä tehdään.

Mikäli Reactin uudet ominaisuudet haluttaisiin ottaa käyttöön jo kehityksessä olevaan sovellukseen, jossa Redux-tilanhallinta on käytössä, kannattaisi se toteuttaa asteittain esimerkiksi niin, että uusia ominaisuuksia otettaisiin käyttöön aluksi vain työn alla oleviin tai kokonaan uusiin komponentteihin. Vaikka sovelluksen riippuvuuksien ja koodirivien määrää saataisiin tällä tavoin vähennettyä, olisi kerralla kaikkien komponenttien uudelleen kirjoittamiseen käytetty työ verrattain suuri suhteessa sen tuomiin hyötyihin.

7.3 Työskentely

Kehitystyön eteneminen oli sujuvaa läpi toteutuksen ja Reactin uusien ominaisuuksien käyttöönotto oli yllättävän selkeää ja yksinkertaista, vaikka tietoa ominaisuuksista löytyi paikoittain vähän. Erityisesti kontekstin käytöstä Reduxia vastaavassa tilanhallinnassa löytyi tietoa todella vähän. Uudet ominaisuudet kuitenkin ylittivät odotukset koodin luettavuuden ja yhtenäisyyden osalta. Myös muut valitut teknologiat osoittautuivat toimiviksi eikä niiden kanssa esiintynyt ongelmia.

Vaikka opinnäytetyössä käytetyt teknologiat Reactin uusia ominaisuuksia lukuun ottamatta olivat työn tekijälle jo ennestään tuttuja, syventyi niiden tuntemus huomattavasti työn tekemisen aikana. Erityisesti Redux-tilanhallinnan datan kulku tuli tekijälle selvemmäksi työn myötä. Aihevalinnan tekijä koki myös erityisen onnistuneeksi, sillä se oli ajankohtainen Reactin juuri julkaistun Hooks-ominaisuuden myötä sekä tekijälle mielekäs. Lisäksi työn tuloksien avulla voidaan jatkossa perustella mahdollisia muutosehdotuksia toimeksiantajalla kehityksessä olevien sovellusten tilanhallintaratkaisuihin sekä komponenttien rakenteisiin.

7.4 Johtopäätökset

Opinnäytetyö oli kokonaisuutena onnistunut oppimiskokemus ja sille asetetut tavoitteet saavutettiin, vaikka myös testaamisen huomiointi olisikin tuonut työlle lisäarvoa. Tavoitteiden täyttymisen lisäksi työ oli tekijälle kuitenkin mielenkiintoinen ja antoisa projekti sekä sen tulokset olivat toivottuja ja johtavat jatkotutkimuksiin.

Selvityksen lopputulemana voidaan pitää, että Redux-tilanhallinta on teknisesti mahdollista korvata Reactin uusilla ominaisuuksilla, vaikka se ei silti täysin korvaa Reduxia. Riippuukin täysin sovelluksen koosta sekä tilanhallinnan tarpeen kompleksisuudesta, onko Redux-kirjastosta luopuminen kannattavaa. Sen sijaan pelkän Hooks-ominaisuuden käytöstä funktiokomponenteissa ei huomattu työn aikana haittapuolia ja se vaikutti täysin stabiililta.

Lähteet

Bagnardi, F., Feldman, R., Hall, J. & Højberg, S. 2016. Developing a React.js Edge: The JavaScript Library for User Interfaces, Second Edition. Bleeding Edge Press.

Components and props. N.d. Dokumentaatio Reactin sivustolla. Viitattu 23.3.2019. https://reactjs.org/docs/components-and-props.html.

Concepts. N.d. Dokumentaatio Webpackin sivustolla. Viitattu 30.3.2019. https://webpack.js.org/concepts.

Context. N.d. Dokumentaatio Reactin sivustolla. Viitattu 24.3.2019. https://reactjs.org/docs/context.html.

Core Concepts. N.d. Dokumentaatio Reduxin sivustolla. Viitattu 31.3.2019. https://redux.js.org/introduction/core-concepts.

de Sousa Antonio, C. 2015. Pro React. Apress.

Developer Survey Results. 2019. Tutkimus Stack Overflow-sivustolla. Viitattu 22.5.2019. https://insights.stackoverflow.com/survey/2019.

Development. N.d. Dokumentaatio Webpackin sivustolla. Viitattu 30.3.2019. https://webpack.js.org/guides/development/.

Digia yrityksenä. N.d. Yritysesittely Digia Oyj:n kotisivuilla. Viitattu 20.3.2019. https://digia.com/yritys/.

Gackenheimer, C. 2015. Introduction to React. New York: Apress.

Getting started. N.d. Dokumentaatio Visual Studio Coden sivustolla. Viitattu 30.3.2019. https://code.visualstudio.com/docs.

Hamedani, M. 2018a. React Lifecycle Methods – A Deep Dive. Ohjelmistokehittäjän ja verkkokouluttajan blogikirjoitus. Viitattu 23.3.2019. https://programmingwithmosh.com/javascript/react-lifecycle-methods/.

Hamedani, M. 2018b. React Virtual DOM Explained in Simple English. Ohjelmistokehittäjän ja verkkokouluttajan blogikirjoitus. Viitattu 24.3.2019. https://programmingwithmosh.com/react/react-virtual-dom-explained/.

Hamedani, M. 2019. What's new in React – React Hooks. Ohjelmistokehittäjän ja verkkokouluttajan blogikirjoitus. Viitattu 23.3.2019. https://programmingwithmosh.com/react/whats-new-in-react-react-hooks/.

Haverbeke, M. 2014. Eloquent JavaScript, 2nd Ed. No Starch Press.

Historia ja yrityskaupat. N.d. Yritysesittely Digia Oyj:n kotisivuilla. Viitattu 20.3.2019. https://digia.com/yritys/historia-ja-yrityskaupat/.

Hooks API Reference. N.d. Rajapintakuvaus Reactin sivustolla. Viitattu 17.5.2019. https://reactjs.org/docs/hooks-reference.html.

Introducing Hooks. N.d. Dokumentaatio Reactin sivustolla. Viitattu 19.3.2019. https://reactjs.org/docs/hooks-intro.html.

Introducing JSX. N.d. Dokumentaatio Reactin sivustolla. Viitattu 19.3.2019. https://reactjs.org/docs/introducing-jsx.html.

Leagacy Context. N.d. Dokumentaatio Reactin sivustolla. Viitattu 24.3.2019. https://reactjs.org/docs/legacy-context.html.

Nutter, M. 2017. Modern Web Development. Freelancer ohjelmistokehittäjän kirjoittama artikkeli. Viitattu 22.5.2019. https://hackernoon.com/modern-web-development-bf0b2ef0e22e.

Overview. N.d. Dokumentaatio Formikin sivustolla. Viitattu 31.3.2019. https://jaredpalmer.com/formik/docs/overview.

Paul, A. & Nalwaya, A. 2016. React Native for iOS Development. Apress.

Quick Start. 2019. Dokumentaatio React Reduxin sivustolla. Viitattu 22.5.2019. https://react-redux.js.org/introduction/quick-start.

Rules of Hooks. N.d. Dokumentaatio Reactin sivustolla. Viitattu 24.3.2019. https://reactjs.org/docs/hooks-rules.html.

Subramanian, V. 2017. Pro MERN stack: Full stack web app development with Mongo, Express, React, and Node. [Berkeley, California]: Apress.

What is Babel? N.d. Dokumentaatio Babelin sivustolla. Viitattu 30.3.2019. https://babeljs.io/docs/en/.

What is Prettier? N.d. Dokumentaatio Prettierin sivustolla. Viitattu 30.3.2019. https://prettier.io/docs/en/index.html.

Why Prettier? N.d. Dokumentaatio Prettierin sivustolla. Viitattu 30.3.2019. https://prettier.io/docs/en/why-prettier.html.

Why Use React Redux? 2018. Dokumentaatio React Reduxin sivustolla. Viitattu 22.5.2019. https://react-redux.js.org/introduction/why-use-react-redux.