jamk

Tilanhallinta React-sovelluksissa

Severi Natunen

Opinnäytetyö, AMK
Toukokuu 2023
Tietojenkäsittelyn tutkinto-ohjelma (AMK)

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Natunen, Severi

Tilanhallinta React-sovelluksissa

Jyväskylä: Jyväskylän ammattikorkeakoulu. Toukokuu 2023, 34 sivua.

Tietojenkäsittelyn tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: kyllä

Tiivistelmä

Tilanhallinta mahdollistaa sovelluksen datan yksisuuntaisen kulun, nopeuttaa sen kehitystä ja testausta ja varmistaa, että dataan päästään käsiksi missä vaan sitä tarvitaan. React-sovelluksen tilanhallintaan on kehitetty useita kirjastoja. Opinnäytetyössä selvitettiin kolmen valitun tilanhallintakirjaston vahvuudet ja heikkoudet ja esitettiin niiden hyvät käytänteet. Näitä kolmea kirjastoa vertailtiin ja autettiin kehittäjiä valitsemaan oikea tilanhallintakirjasto, joka vastaa oman projektin tarpeisiin.

Opinnäytetyössä toteutettiin Reactilla pieni esimerkkisovellus, jonka teemana oli tehtävälista. Sen jälkeen tilanhallinta toteutettiin esimerkkisovellukseen opinnäytetyöhön valituilla tilanhallintakirjastoilla. Sovelluksen tilaan haluttiin viedä tehtävät ja niitä manipuloivat actionit. Lopuksi tilanhallintakirjastoja vertailtiin asetettujen aihealueiden mukaan.

Opinnäytetyön tilanhallintakirjastojen vertailusta voitiin todeta, että Zustand on kirjastoista nopein oppia, koska se muistuttaa käytänteiltään paljon Reactin komponentin tilanhallintaa. Zustand oli kompaktein kirjastoista ja soveltuu parhaiten yksinkertaisen sovelluksen tilanhallintaan. Redux oli käytänteiltään samankaltainen kuin Zustand, mutta tarjosi systemaattisemman arkkitehtuurin tilanhallinnan toteutukseen ja enemmän ominaisuuksia. Tämän takia Reduxin todettiin olevan hyvä valinta monimutkaisempaan sovellukseen. MobX:n todettiin tarjoavan vaihtoehto muille opinnäytetyön kirjastoille, jotka perustuvat funktionaaliseen ohjelmointiin. MobX perustuu olio-ohjelmointiin ja luokkapohjaisuuteen ja sen koettiin antavan kehittäjälle eniten vapautta tilanhallinnan toteutuksen ja muokattavuuden suhteen.

Avainsanat (asiasanat)

Tilanhallinta, React, Web-sovelluskehitys, JavaScript

Muut tiedot (salassa pidettävät liitteet)

jamk

Description

Natunen, Severi

State management in React-applications

Jyväskylä: JAMK University of Applied Sciences, May 2023, 34 pages.

Degree Programme in Web & Data. Bachelor´s thesis.

Permission for open access publication: Yes

Language of publication: Finnish

Abstract

State management enables data to flow in one direction, which makes development and testing faster and makes sure that data can be accessed anywhere it is needed. Several libraries have been developed for accomplishing state management in React-applications. In the thesis three selected state management libraries were researched to find out their strengths and weaknesses and show their best practices. These three libraries were compared to help developers choose the right state management library for their project's needs.

In the thesis an example application was made using React. Theme of the application was a task list. After that state management was executed with the three selected state management libraries. Tasks and the actions that manipulate them were taken to the application's state. Afterwards the three libraries were compared in the set areas.

The thesis found out that out of the three state management libraries Zustand was the quickest to learn. This was because its practices remind the state management inside a React component. It was the most compact library of the three and it was concluded that it fits best for state management in simple applications. Redux was familiar in its practices to Zustand but offered a more systematic way to execute state management and offered more features. Because of this it was concluded that Redux is best choice for a more complicated application. MobX offered an option for the other two libraries that based on functional programming. It uses object-oriented programming, and the state is based on the use of classes. It was concluded that MobX offered most freedom to the developer on how you want to carry out your state management.

Keywords/tags (subjects)

State management, React, Web development, JavaScript

Miscellaneous (Confidential information)

Sisältö

1 Johdanto ... 6
2 Tutkimuksen tavoitteet ... 6
2.1 Tutkimusmenetelmä ... 7
2.2 Tutkimustehtävä ... 8
3 React ... 8
3.1 React-sovelluksen rakenne ... 8
3.2 JavaScript ... 9
3.3 DOM ... 10
3.4 JSX ... 10
4 React ja tilanhallinta ... 11
4.1 Redux ... 12
4.2 Zustand ... 13
4.3 MobX ... 14
5 Tutkimuksen toteutus ... 15
5.1 React-projekti ... 16
5.2 Esimerkkisovellus ... 17
5.3 Tilanhallinta Reduxilla ... 18
5.4 Tilanhallinta Zustandilla ... 23
5.5 Tilanhallinta MobX:llä ... 25
6 Tulosten vertailu ... 28
6.1 Dokumentaatio ... 28
6.2 Ajankäyttö ... 29
6.3 Käyttäjäkokemus ... 30
7 Pohdinta ... 31
7.1 Luotettavuus ja eettisyys ... 31
7.2 Tulosten tarkastelu suhteessa teoreettiseen viitekehykseen ... 32
7.3 Johtopäätökset ja kehittämisehdotukset ... 33
Lähteet ... 35

Kuviot

Kuvio 1. JSX havainnollistettuna 11
Kuvio 2. Flux-arkkitehtuuri havainnollistettuna 12
Kuvio 3. Sivuvaikutusmalli havainnollistettuna 14
Kuvio 4. React-projektin kansiorakenne 17
Kuvio 5. Esimerkkisovelluksen käyttöliittymä 18
Kuvio 6. Redux storen tarjoaminen React-sovellukselle 19
Kuvio 7. Redux state slice havainnollistettuna 20
Kuvio 8. Redux store havainnollistettuna 21
Kuvio 9. useSelector-hook havainnollistettuna 21
Kuvio 10. useDispatch-hook ja addTodo-action 22
Kuvio 11. Zustand store havainnollistettuna 23
Kuvio 12. Form.js tiedosto 24
Kuvio 13. Mobx store havainnollistettuna 26
Kuvio 14. Observer-funktion ja storen käyttö komponentissa 27
Kuvio 15. Mobx actionin käyttö komponentissa 27

1 Johdanto

Nykypäivänä lähes kaikkien web-sivujen ja -sovellusten kehitykseen käytetään JavaScript-viitekehyksiä. Uusia viitekehyksiä kehitetään jatkuvasti, mutta viime vuosien yksi halutuimmista on React. Reactilla voidaan kehittää dynaamisia web-sovelluksia komponenttipohjaisesti ja niiden uudelleenkäytettävyys helpottaa sovelluksen kehitystä ja ylläpitoa. React kehittyy jatkuvasti, kun sen päälle tehdään uusia viitekehyksiä, jotta sen tapa toteuttaa tehokkaasti käyttöliittymiä saadaan yhdistettyä nykyaikaiseen full-stack sovelluskehitykseen. Siksi on tärkeää, että kehittäjillä on riittävät taidot ylläpitää sovelluksen käyttöliittymäpuolen ominaisuuksia ja sen logiikkaa.

Tässä opinnäytetyössä pyritään syventymään tilanhallintaan React-sovelluksissa. Tilanhallinta on tärkeä aihe nykyaikaisissa web-sovelluksissa, joissa käsitellään muuttuvaa dataa eri komponenttien välillä. Tilanhallinnalla pyritään varmistamaan se, että sovelluksen tila on yhdenmukainen käyttöliittymän kanssa, jolloin voidaan varmistaa sovelluksen halutunlainen toiminta niin ulkonäöllisesti kuin myös eri toiminnallisuuksien kannalta. Myös tilanhallintaan on kehitetty lukuisia kirjastoja, mutta tässä opinnäytetyössä keskitytään tutkimusta tehdessä eniten latauskertoja keränneisiin, jotka ovat Redux, Zustand ja Mobx. Opinnäytetyössä halutaan tuoda esiin näiden kirjastojen hyvien käytänteiden lisäksi niiden vahvuuksia ja heikkouksia toteuttaa tilanhallinta React-sovelluksessa ja näin tarjota apua tilanhallintakirjastoa valitsevalle kehittäjälle.

Opinnäytetyössä käsitellään ensin Reactia ja sen toimintaan liittyviä keskeisiä ominaisuuksia. Samassa yhteydessä tilanhallintaan, sen arkkitehtuuriin ja perusteisiin syvennytään tarkoituksena luoda kuva tilanhallinnan tärkeydestä. Opinnäytetyön toteutuksen vaiheessa tilanhallintakirjastojen käyttöönotto ja tilanhallinnan toteutus esitetään esimerkkisovelluksen avulla, josta johdetaan tulokset. Tuloksissa tilanhallintakirjastoja vertaillaan asetettujen ominaisuuksien valossa ja pyritään tarjoamaan kiteytys kunkin kirjaston vahvuuksista ja mahdollisista heikkouksista. Lopuksi tehdään johtopäätökset siitä, milloin kukin tilanhallintakirjasto olisi hyvä valinta kehittäjän projektin kannalta sekä pohditaan tutkimuksen tuloksia.

2 Tutkimuksen tavoitteet

Stack Overflow -sivuston tutkimuksen (2022) mukaan, React on toiseksi suosituin web-viitekehys ammattilaisten ja koodaamaan opettelevien keskuudessa. Ammattilaisista 22.54 % osoitti myös

halunsa aloittaa tai opetella kehittämään käyttäen Reactia, joka tekee siitä tutkimuksen halutuimman web-viitekehyksen. Koska React on niin suosittu ja haluttu teknologia, tutkimustieto siihen liittyen on hyvin ajankohtaista. Tilanhallinta aiheena on hyvin tärkeä, koska sen avulla monimutkaisissa sovelluksissa niiden tarvitsema data voidaan säilyttää saatavilla yhdessä ja tietyssä paikassa, joka itsessään auttaa vähentämään koodin toistoa ja näin pitää koodin saavutettavana. Tutkimuksen tavoite on löytää kunkin tilanhallintakirjaston vahvuudet ja heikkoudet, auttaa kehittäjiä valitsemaan omaan tarpeeseen sopiva tilanhallintakirjasto ja ymmärtämään sen hyvät käytänteet. Lopputulos on selvitys näistä edellä mainituista asioista ja esimerkkisovellus, johon on toteutettu tilanhallinta näillä tilanhallintakirjastoilla.

Opinnäytetyö rajataan niin, että tilanhallintakirjastoista tutustutaan kolmeen viimeisen vuoden aikana npm-paketinhallintajärjestelmällä ladatuimpaan (npm trends n.d.). Nämä tilanhallintakirjastot ovat Redux, Zustand ja MobX. Rajauksessa on myös huomioitu kyseisten kirjastojen suosio GitHub-sivustolla ja kuinka niiden virallisessa dokumentaatiossa on huomioitu käyttöönotto Reactin kanssa.

2.1 Tutkimusmenetelmä

Tämä opinnäytetyö toteutetaan kvalitatiivisena vertailevana tutkimuksena. Opinnäytetyössä vertailtavia kohteita ovat rajauksen mukaiset tilanhallintakirjastot Redux, Zustand ja Mobx. Näille kirjastoille määritetään vertailtavat kohteet ja ominaisuudet ja niillä toteutetaan tilanhallinta esimerkkisovelluksessa.

Opinnäytetyö on kvalitatiivinen tutkimusprosessi koska sen ominaispiirteet tulevat vastaamaan toteutustapaa. Juhilan (n.d.) mukaan näitä piirteitä ovat muun muassa kvalitatiivisen aineiston suosiminen, sitoutuminen lähelle menevään tarkasteluun ja toimintaan keskittyminen, tutkijan tulkintojen korostaminen sekä mitä- ja miten-kysymyksien painottaminen. Saaranen-Kauppisen ja Puusniekan (2006) mukaan Töttö (2004) määrittelee kvalitatiivisen tutkimuksen muodostuvan tutkittavan aiheen teoriasta, aineistosta ja tutkijan omasta ajattelusta.

2.2 Tutkimustehtävä

Saaranen-Kauppisen ja Puusniekan (2006) mukaan, tutkimusongelman asettaminen on joustavaa laadullisessa tutkimuksessa ja joskus sen sijasta voidaan asettaa tutkimukselle tutkimustehtävä. Koska tässä opinnäytetyössä keskitytään rajattuun määrään tilanhallintakirjastoja, on järkevämpää asettaa tutkimustehtävä, jonka kautta näistä kirjastoista lähdetään selvittämään tietoa ja siten vertailemaan niitä keskenään. Tutkimustehtävänä on siis selvittää tilanhallintakirjastojen vahvuudet ja heikkoudet, esittää niiden hyvät käytänteet esimerkkisovelluksessa ja näin auttaa muita kehittäjiä valitsemaan omaan tarpeeseen sopiva tilanhallintakirjasto.

3 React

React on JavaScript-viitekehys. Sen kehitti alun perin Facebook tarkoituksenaan rakentaa viitekehys, jolla voidaan luoda monimutkaisia käyttöliittymiä komponenteista, jotka käsittelevät ajan myötä muuttuvaa dataa. (Gackenheimer 2015.) Myös Thomasin (2018) mukaan React on komponentteihin perustuva viitekehys, jonka vahvuuksia ovat selkeä käyttömalli ja suorituskykyinen tapa toteuttaa käyttöliittymiä, jotka toimivat useilla alustoilla.

Yksi Reactin keskeisistä ominaisuuksista on sen konsepti virtuaalisesta DOM-objektista. Kohdatessaan muutoksen datassa tai käyttäjän interaktion käyttöliittymässä, React käyttää virtuaalista DOMia laskeakseen pienimmän määrän muutoksia, jotka se asettaa sovelluksen varsinaiselle DOMille (Gackenheimer 2015). Toinen Reactin tärkeä ominaisuus on JSX-niminen JavaScript-syntaksin laajennus. Sen rakenne on samankaltainen HTML-merkintäkielen rakenteen kanssa, joka helpottaa kehittäjiä strukturoimaan React-koodia. Lisäksi se vähentää toiston määrää koodissa, koska tarvittava JavaScript-koodi kirjoitetaan yhdessä HTML-merkintäkielen kanssa (Gackenheimer 2015).

3.1 React-sovelluksen rakenne

React on suosittu toteuttaessa SPA-sovelluksia, joka on lyhenne sanoista single-page application. Nimensä mukaisesti tai kuten Freeman (2019) sen määrittelee, SPA-sovellukset toimivat yhdellä HTML-sivulla, joka lähetetään selaimeen ja jonka osia päivitetään käyttäjän toimintojen mukaan. Alkuperäistä HTML-tiedostoa ei koskaan korvata, vaan sen osia päivitetään ja käyttäjä voi jatkaa sovelluksen käyttöä yhtäaikaisesti päivityksen kanssa. React-sovelluksessa nämä päivitettävät osat

ovat sen komponentteja. Freemanin (2019) mielestä tämä on Reactin komponenttipohjaisuuteen perustaen suorituskykyisin tapa toteuttaa React-sovelluksia, verrattuna niin sanottuihin ”round-trip” -sovelluksiin, joiden toiminta perustuu koko sivun päivittämiseen jonkin käyttäjätoiminnon tapahtuessa.

Kuten mainittu, React-sovellukset perustuvat komponentteihin. Yleisesti React-sovelluksella on yksi juurikomponentti, johon sovellus rakennetaan käyttäen muita komponentteja. Nämä komponentit voivat kuvastaa esimerkiksi käyttöliittymän elementtejä, ja ne käyttävät apunaan sisäistä tilaansa ja ominaisuuksiaan määrittääkseen ulkonäkönsä. Komponenttien lisäksi React-sovelluksissa käytetään paljon kolmannen osapuolen kirjastoja hoitamaan yleisiä frontend-sovelluksen tehtäviä. (Thomas 2018.) Näitä kirjastoja ovat esimerkiksi opinnäytetyössä käytettävät tilanhallintakirjastot.

3.2 JavaScript

Alun perin JavaScript-ohjelmointikieli luotiin, jotta verkkosivuista saataisiin dynaamisia ja interaktiivisia. Vuosien varrella sen suosio on kasvanut ja useiden standardisointien kautta siitä on tullut web-kehittämisen suosituin ohjelmointikieli, koska sillä voidaan hallita sovelluksen tai verkkosivun sisältöä (HTML) ja ulkonäköä (CSS). Frontend-kehityksessä se siis määrittää sovelluksen tai verkkosivun käytöksen. (Ranjan, Sinha & Battewad 2020.)

Ranjan ja muut (2020) määrittelevät JavaScriptin monialustaiseksi ja olio-orientoituneeksi ohjelmointikieleksi. Ranjan ja muut (2020) yhdessä Shuten (2019) kanssa jakavat JavaScriptin rakennuspalat karkeasti neljään osaan. Niitä ovat muuttujat ja niiden tyypit, lausekkeet, funktiot ja näkyvyysalueet, joilla näiden toimintaa määritellään.

JavaScriptin suosiota lisää suuri valikoima viitekehyksiä ja kirjastoja, jotka auttavat ja nopeuttavat kehittäjää strukturoimaan koodia ja vähentämään monimutkaisen koodin määrää toiminnallisuuksia luodessa (Ranjan ym. 2020). React on tällainen viitekehys, kuten myös opinnäytetyössä käsiteltävät tilanhallintakirjastot.

Vaikka JavaScript nauttii suurta suosiota ohjelmointikielenä, moni React-kehittäjä valitsee projektiinsa nykyään TypeScriptin. TypeScript on Freemanin (2019) mukaan ennalta arvattava ja näin turvallisempi vaihtoehto JavaScriptille. Tämän TypeScript saavuttaa Elromin (2021) mukaan lisäämällä enemmän tyyppejä, joilla React-sovellukselle voidaan kuvata, mitä siltä odotetaan. TypeScriptin tyypitystä kutsutaan staattiseksi tyypitykseksi, eli tyypit tarkistetaan ennen sovelluksen ajoaikaa. Tämä tekee mahdollisten virheiden huomaamisesta helpompaa, helpottaa React-sovelluksen testaamista ja parantaa koodin laatua. (Elrom 2021.)

3.3 DOM

DOM eli Document Object Model (suom. dokumenttioliomalli) on keino kuvata HTML- tai XML-dokumentti hierarkkisten solmujen puuna (Elrom 2021). Thomasin (2018) mukaan DOM tarjoaa rajapinnan, jonka kautta JavaScriptin avulla näihin solmuihin voidaan päästä käsiksi ja näin muokata tai tallentaa verkkosivun tai sovelluksen sisältöä dynaamisesti.

React käyttää virtual DOMia, jonka tarkoitus on nopeuttaa suorituskykyä. React luo ja ylläpitää virtual DOMia muistissaan, jossa se laskee ainoastaan tarvittavat muutokset, jotka ovat syntyneet esimerkiksi käyttäjän painaessa nappia käyttöliittymässä ja sitten asettaa nämä muutokset sovelluksen todelliselle DOMille. Näin säästytään siltä, että koko DOM täytyisi päivittää muutoksen tapahtuessa. (Elrom 2021.) Myös Thomas (2018) toteaa, että erityisesti suurissa ja monimutkaisissa sovelluksissa virtual DOM kasvattaa suorituskykyä, koska esimerkiksi yksittäisen elementin muokkaaminen ei pakota sovelluksen tai verkkosivun oikean DOMin läpikäyntiä ja päivittämistä. Thomas (2018) kuitenkin muistuttaa, ettei virtual DOM yksinään tee React-sovelluksesta suorituskykyistä, vaan väittää sen olevan suunniteltu olemaan riittävän nopea Reactille uhraamatta sen helppokäyttöisyyttä.

3.4 JSX

JSX eli JavaScript XML on JavaScript-syntaksin laajennus (ks. kuvio 1), joka käy läpi XML-rakenteista koodia ja korvaa sen tagit JavaScript funktioilla, joista se generoi React komponentteja ja elementtejä (Gackenheimer 2015). Käytännössä, kuten myös Freeman (2019) toteaa, React luo HTML-elementtejä render-metodinsa avulla, joka luo objektin, joka saa määritelmänsä sille annettujen

ominaisuuksien kautta. Kehittäjä voi siis määrittää HTML-elementit normaalisti, josta Reactin kehittäjätyökalut kääntävät lopputuloksen JavaScriptiksi.

```
function App() {
  return (
    <>
      <div className="header">
        <p>Hello World!</p>
      </div>
      <ul className="todolist">
        <li>Thesis</li>
        <li>Thesis writing course</li>
        <li>Internship</li>
      </ul>
    </>
  );
}
```

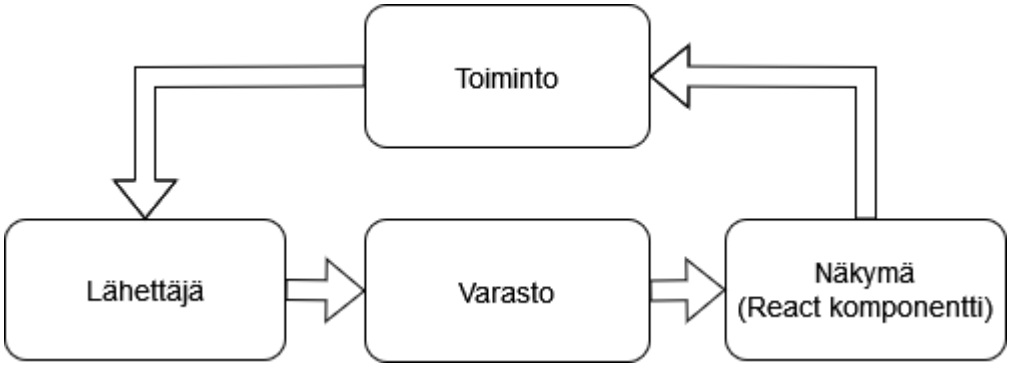
Kuvio 1. JSX havainnollistettuna

React ei vaadi JSX:n käyttöä, mutta sen käyttö on suositeltavaa useastakin syystä. JSX:n käyttö tekee koodista saavutettavampaa sen vähämerkkisyyden takia. Lisäksi sen HTML-merkintäkieltä muistuttava rakenne on tuttu useimmalle kehittäjälle, josta se automaattisesti kääntää Reactille vaadittavan JavaScriptin. Gackenheimerin (2015) mukaan tällöin voidaan saavuttaa sujuvampaa yhteistyötä esimerkiksi eri työryhmien välillä, koska sekä kehittäjät että suunnittelijat voivat ymmärtää React komponenttien ja elementtien rakenteen ja kuinka niitä luodaan. Freeman (2019) kuitenkin toteaa, että JSX jakaa mielipiteitä kehittäjien välillä, sillä kaikki eivät halua yhdistää HTML- ja JavaScript-kieliä vaan pitävät ne mieluummin erillään.

4 React ja tilanhallinta

Tilanhallinta Reactissa noudattaa vahvasti Facebookin aikoinaan luomaa Flux-arkkitehtuuria (ks. kuvio 2). Flux-arkkitehtuuri jakautuu karkeasti kolmeen osaan. On varasto (engl. store), joka tallettaa sovelluksen tilan (engl. state). Näkymät eli tässä tapauksessa React komponentit aloittavat toimintoja (engl. action), jotka vastaan ottaa lähettäjä (engl. dispatcher). Lähettäjä vie datan muutokset edelleen varastoon. (Elrom 2021.) Esimerkki tästä on, että käyttäjä lisää tuotteen ostoskoriin

näkymässä, jolloin toiminto vie tiedon uudesta tuotteesta lähettäjälle, joka taas lisää uuden tuotteen varastoon, jonka varasto tallennetaan ostoskoriin, joka on sovelluksen tila. Lopputuloksena päivittynyt ostoskori näkyy käyttäjälle näkymässä.



Kuvio 2. Flux-arkkitehtuuri havainnollistettuna

Reactin tilanhallinnan hyvä käytänne on siis viedä sovelluksen tila yhteen paikkaan eli varastoon (engl. data store). Tähän löytyy useitakin syitä. Tilan siirtäminen varastoon vie sen React komponenttihierarkian ulkopuolelle, jolloin säästytään siltä, että sovelluksen tilaa tulisi nostaa jatkuvasti korkeimmalle tasolle (Freeman 2019). Tällöin sovelluksen tilaa voidaan käyttää vain ja ainoastaan siellä missä sitä tarvitaan. Esimerkiksi kun käyttäjä vaihtaa sovelluksen teemaa, teeman arvo muuttuu varastossa, josta se jakaa tiedon eteenpäin komponenteille. Komponentit osaavat tämän tiedon perusteella muuttaa ulkoasuaan ilman, että niiden tarvitsee lähettää tietoa takaisinpäin. Erityisesti monimutkaisemmissa sovelluksissa kaikki tämä auttaa pitämään sovelluksen rakenteen siistimpänä ja helpommin testattavana.

Reactin tilanhallinnalle on kehitetty useita kirjastoja. Lähes kaikkia niitä yhdistää Fluxin kaltainen tai siitä inspiroitunut arkkitehtuuri. Arkkitehtuurin keskiössä on varasto, joka toimii yhtenä ja ainoana totuuden lähteenä eli globaalina sovelluksen tilana (Thomas 2018). Tässä opinnäytetyössä näistä tarkempaan tutkimukseen valitaan Redux, Zustand ja Mobx.

4.1 Redux

Redux on tilanhallintakirjasto JavaScript-sovelluksille. Reduxin tilanhallinnan voi jakaa kolmeen osaan. Sovelluksella on varasto (engl. store), johon se tallentaa kaikki sovelluksen tilat (engl. state)

yhteen objektiin. Varastoa voi päivittää vain objekteilla, jotka kuvaavat jotain tapahtumaa (engl. action). Funktiot, jotka Reduxissa tunnetaan nimellä reducer, kertovat kuinka tilaa muutetaan ottamalla nykyisen tilan ja actionin, jonka jälkeen se palauttaa uuden tilan muutoksineen. (Garreau & Faurot 2018.)

Reduxin tapaa tallentaa sovelluksen tila yhteen varastoon kutsutaan single source of truth -arkkitehtuuriksi. Kehittäjälle sen etu on vaivattomammin havainnollistaa sovelluksen kulku, uusien tapahtumien lopputulosten ennustaminen ja niiden mahdollisesti tuottamien ongelmien ratkominen. (Garreau ym. 2018.) Lisäksi tilan muuttaminen on mahdollista vain näiden tapahtumien kautta. Tärkeä käytänne Reduxissa on, että reducerit eivät muuta alkuperäistä dataa missään vaiheessa, vaan vanhasta varastosta otetaan aina kopio, johon muutokset päivitetään tarvittaessa.

Kuten Lee, Wei ja Mukhiya (2019) toteavat, Redux osoittautuu erityisen hyödylliseksi suurissa sovelluksissa, joissa ilman Reduxia sovelluksen tarvitsemaa dataan jouduttaisiin välittämään vanhempikomponentilta lapsikomponentille niiden ominaisuuksien kautta. Kuten myös Garreau ja muut (2018) sen sanovat, Redux mahdollistaa datan yksisuuntaisen kulun vain niille komponenteille, jotka sitä tarvitsevat. Lee ja muut (2019) sekä Garreau ja muut (2018) ovat yhdessä sitä mieltä, että tämä helpottaa sekä uusien toiminnallisuuksien lisäämistä että sovelluksen virheiden paikantamista ja poistamista. Garreau ja muut (2018) lisäävät, että Reduxin opetteluun vaadittava panos on myös pieni, koska se lisää vain muutaman ohjelmointirajapinnan opeteltavaksi Reactin jo hallitsevalle kehittäjälle.

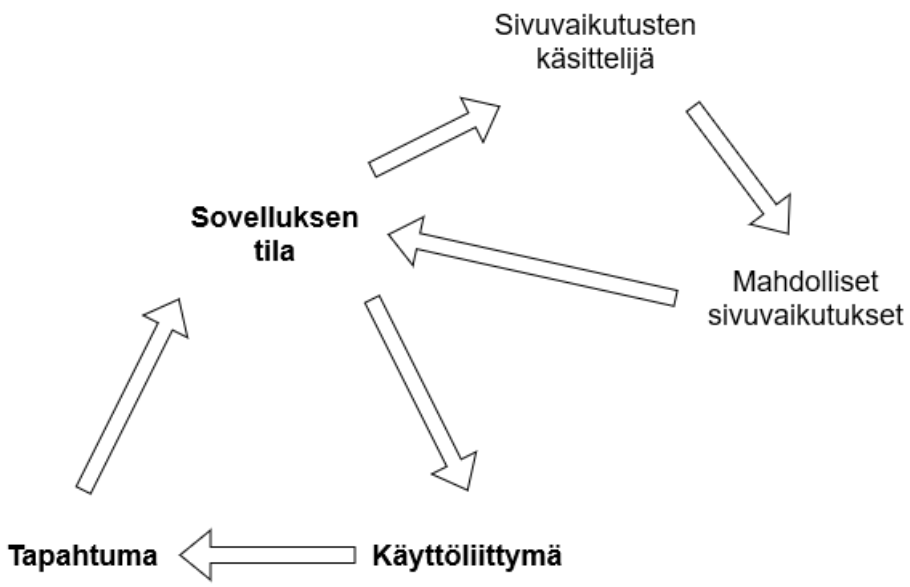
4.2 Zustand

Zustand on pieni mutta nopea tilanhallintakirjasto, joka on kehitetty erityisesti huomioiden Reactin tarpeita. Zustandin ohjelmointirajapinta perustuu koukkujen (engl. hooks) käyttöön. Zustandin store on itsessään hook, johon voi sijoittaa esimerkiksi funktioita tai objekteja. Storea voi sen jälkeen käyttää missä tahansa ilman tarjoajaa (engl. provider), joka on sen suurin eroavaisuus Reduxista. (Introduction n.d.) Muuten Zustand jakaa vahvasti käytänteensä Reduxin kanssa esimerkiksi tilan muuttumattomuudesta (immutable).

Zustand pyrkii yksinkertaiseen ja joustavaan tilanhallintaan. Sen yksi tärkeistä ominaisuuksista on tilapäiset päivitykset sovelluksen tilaan monesti toistuvissa päivityksissä. Zustand käyttää funktiota, jolla se pystyy yhdistämään sovelluksen tilaan ja käyttämään sitä ilman, että tilan muuttuminen pakottaa käyttöliittymän päivittämisen. Tällä tapaa Zustand pyrkii parantamaan sovelluksen suorituskykyä. Kaikkia näitä ominaisuuksiaan hyödyntäen Zustand kykenee tilanhallintaan ilman turhaa koodin toistamista ja väittää näin olevansa kehittäjälle juostavampi siinä, kuinka tilanhallinnan haluaa toteuttaa. (readme.md 2023.)

4.3 MobX

MobX on dokumentaationsa (README MobX n.d.) mukaan yksinkertainen ja skaalautuva tilanhallintakirjasto. Se noudattaa sivuvaikutusmallia, jossa käyttöliittymän aiheuttamat tapahtumat muuttavat sovelluksen tilaa, jossa käsitellään mahdolliset sivuvaikutukset, jotka edelleen muuttavat sovelluksen tilaa (ks. kuvio 3).



Kuvio 3. Sivuvaikutusmalli havainnollistettuna

MobX sovelluksen tila on observable, joka tekee siitä reaktiivisen. Tätä tilaa voidaan muokata määrittelemällä sille sitä muokkaavia toimintoja (engl. actions) ja observaattoreita (engl. observer). Observaattoreille voidaan määritellä myös sivuvaikutuksia. Observer-ohjelmointirajapinta pitää

huolen, että komponentti päivitetään ainoastaan, jos sovelluksen tila muuttuu. (Podila & Weststrate 2018.)

Kuten aiemmin on mainittu, Redux ja Zustand noudattavat sovelluksen tilan muuttumattomuutta, eli sovelluksen tila päivitetään ainoastaan ottamalla kopio vanhasta tilasta, johon muutokset päivitetään, joka taas asetetaan sovelluksen tilaksi. Podilan ja Weststraten (2018) mukaan MobX suurin eroavaisuus tämänkaltaisista tilanhallintakirjastoista on sen tapa muuttaa sovelluksen tilaa suoraan ja näin väitetysti saavuttaa sama lopputulos pienemmällä määrällä koodia ja selkeämmin. Podilan ja Weststraten (2018) mukaan tämä saavutetaan esimerkiksi niin, että siinä missä Reduxia käytettäessä kehittäjä joutuu miettimään reducer-funktioita tilan muuttamiseksi, MobX-sovelluksessa kehittäjä voi keskittyä pelkästään sovelluksen observable-tilaan ja siihen, kuinka sitä vastaava React-komponentti saa sen käyttöönsä.

5 Tutkimuksen toteutus

Ennen tutkimukseen valittujen tilanhallintakirjastojen käyttöönottoa esimerkkisovelluksessa ja niillä tilanhallinnan toteutusta, kirjastoille määritetään vertailtavat kohteet ja ominaisuudet. Arvioinnille valittiin kolme pääaihealuetta, jotka ovat:

- dokumentaatio
- ajankäyttö ja
- käyttäjäkokemus.

Dokumentaatiossa kiinnitetään huomiota dokumentaation laajuuteen, sen informatiivisuuteen ja kuinka siinä on huomioitu käyttöönotto Reactin kanssa. Tähän aihealueeseen lukeutuu myös yhteisön tuki, eli kuinka paljon tietoa tilanhallintakirjastosta on saatavilla esimerkiksi GitHubissa tai muilla alustoilla kuten Stack Overflow-sivustolla.

Ajankäytössä huomioidaan, kuinka tehokkaasti tilanhallintakirjasto pystytään implementoimaan React-sovellukseen alkaen sen asennuksesta. Ajankäytössä huomioidaan myös boilerplate-koodin määrä eli kuinka paljon koodia joudutaan toistamaan, että tilanhallinta saadaan toteutettua.

Käyttäjäkokemuksen osalta keskitytään siihen, kuinka helposti tilanhallintakirjaston käytänteet ovat omaksuttavissa. Lisäksi huomioidaan, kuinka muokattavissa tilanhallintakirjasto on, eli onko kirjasto muokattavissa käyttäjän mieltymyksien mukaan vai pakottaako tilanhallintakirjasto käyttämään ennalta määrättyä mallia.

5.1 React-projekti

Tutkimuksen alussa luotiin uusi React-projekti esimerkkisovellusta varten. Tämä onnistuu helpoiten ajamalla terminaalissa komento:

```
npx create-react-app my-app
```

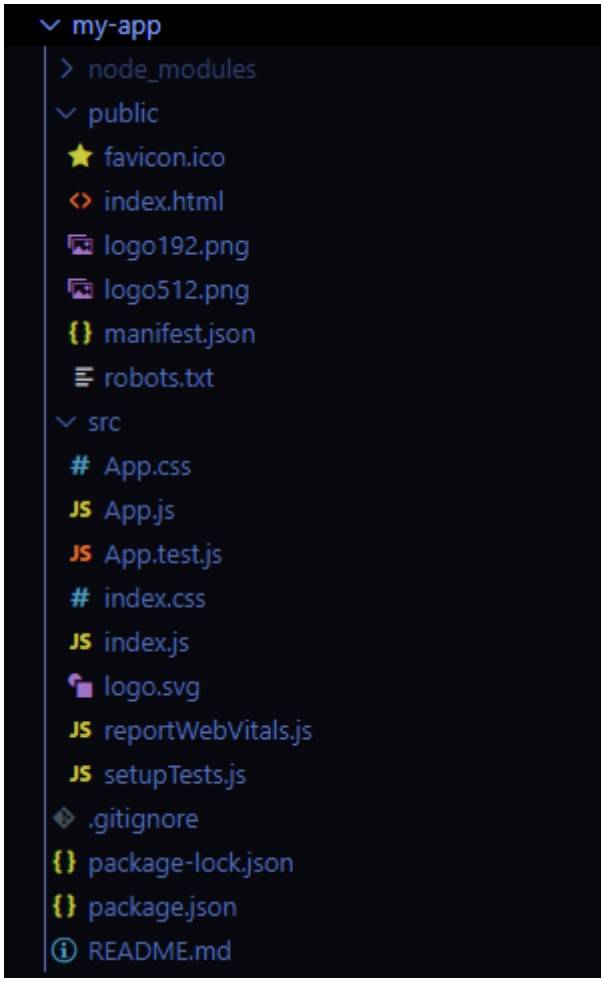
Komennossa my-app osan voi korvata haluamallaan nimellä ja se määrittyy projektikansion nimeksi. Komento luo ympäristön, jolla voidaan aloittaa kehittämään SPA-sovelluksen frontendiä (ks. kuvio 4). Projektin public-kansio sisältää index.html -tiedoston, johon skriptit sijoitetaan, kun sovellus rakennetaan käyttövalmiiksi. Sinne voidaan tehdä myös muutoksia fontteihin, metatunnisteisiin ja sovelluksen logoihin.

Projektin src-kansio sisältää kaikki sovelluksen toimintaan ja ulkonäköön liittyvät tiedostot. Esimerkiksi App.js-tiedostoon voidaan kirjoittaa sovellukselle funktioita tai lisätä käyttöliittymäelementtejä ja App.css-tiedostossa voidaan määrittää niiden ulkonäkö. Src-kansioon luodaan myös kaikki sovelluksen tarvitsemat komponentit. Projekti sisältää automaattisesti sovelluksen testaamiseen liittyviä tiedostoja kuten setupTests.js.

React-projektin voi ajaa selainympäristössä kehitysaikana siirtymällä projektikansioon ja antamalla terminaalissa komento:

```
npm start
```

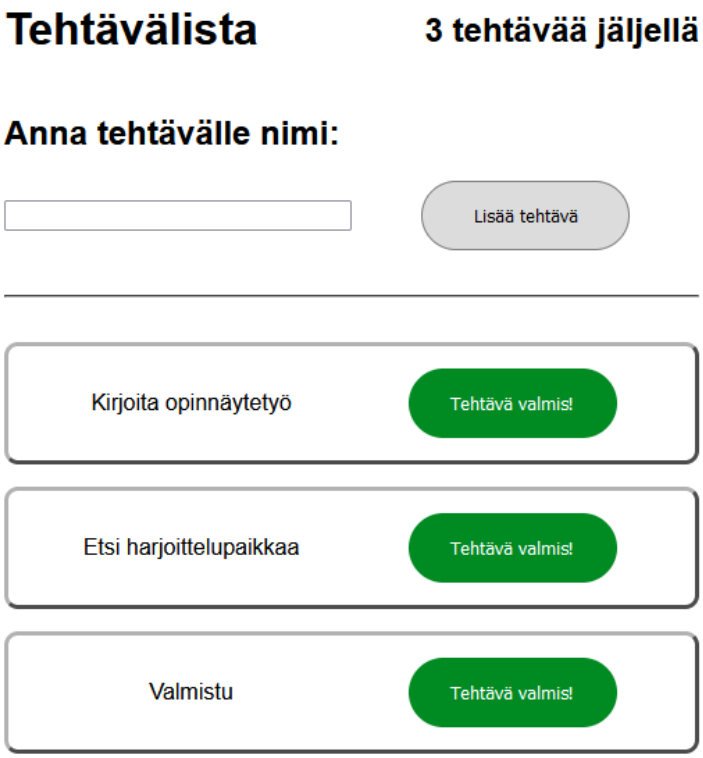
Projekti aukeaa selaimeen osoitteeseen https://localhost:3000. Projektin tiedostoja voi muokata ajon aikana ja muutokset päivittyvät selaimeen tallentaessa.



Kuvio 4. React-projektin kansiorakenne

5.2 Esimerkkisovellus

Tutkimusta varten toteutettiin yksinkertainen React SPA-sovellus. Sovellus on teemaltaan tehtävälista, johon käyttäjä voi tekstikenttään syöttämällä ja painiketta painamalla lisätä tehtäviä. Tehtävät päivittyvät käyttöliittymään uusina komponentteina (ks. kuvio 5). Tehtävän voi merkata tehdyksi painamalla painiketta, jolloin tehtävä ja sen komponentti poistuvat käyttöliittymästä. Lisäksi käyttöliittymässä on laskuri, joka kertoo jäljellä olevien tehtävien määrän.



Kuvio 5. Esimerkkisovelluksen käyttöliittymä

Sovelluksen tilassa säilytetään tehtävät objekteina, joista löytyy tehtävän id ja nimi. Tehtävät tulostetaan käyttöliittymään map-funktion avulla, joka palauttaa uusia komponentteja. Sovelluksella on Form-niminen komponentti, joka käsittelee tehtävän lisäämisen. Sen tilassa säilytetään käyttäjän syöte eli tehtävän nimi ja painettaessa Lisää tehtävä -nappia, se lisää sen sovelluksen tilaan ja antaa sille uniikin id:n. Jäljellä olevien tehtävien määrä lasketaan muuttujaan tehtävälistan pituudesta ja muuttuja näytetään käyttöliittymässä. Opinnäytetyöhön valituilla tilanhallintakirjastoilla sovelluksen tilaan eli data storeen halutaan viedä tehtävät.

5.3 Tilanhallinta Reduxilla

Vaikka Redux on suosittu tilanhallintakirjasto Reactilla kehittäessä, tulee muistaa, että nämä kaksi kirjastoa ovat itsenäisiä. Reduxia voi siis käyttää monen muunkin JavaScript-viitekehyksen kanssa. Siksi lisätessä Reduxia React-projektiin tulee ottaa huomioon, että kehittäjä asentaa tarvittavat paketit siihen, että Redux ja käyttöliittymä saadaan sidottua yhteen. Reactin tapauksessa käyttöliittymän sitova kirjasto on nimeltään React Redux, joka sisältää ennalta asetettuja sääntöjä koskien sitä, miten Redux store ja sen logiikka keskustelee käyttöliittymän kanssa. Huomioitavaa on myös

se, että Redux suosittelee nykyään Redux Toolkitin käyttöä. Redux Toolkit sisältää Reduxin pääpaketit ja apuvälineet, joita tarvitaan Redux-sovelluksen rakentamiseen hyvien käytänteiden mukaisesti. Redux Toolkit yksinkertaistaa Redux ominaisuuksien kirjoittamista ja vähentää virheiden määrää. React Redux ja Redux Toolkit saadaan lisättyä helposti React-projektiin antamalla terminaalissa komento:

```
npm install @reduxjs/toolkit react-redux
```

Redux storea varten tulee luoda oma tiedostonsa projektin juurikansioon, jotta se saadaan Reactin komponenttihierarkian ulkopuolelle tilanhallinnan hyvien käytänteiden mukaisesti. Tässä projektissa juurikansioon luotiin store.js niminen tiedosto. Redux store tulee tarjota React-sovellukselle, ja tämä onnistuu importoimalla luotu store projektin index.html tiedostoon. Index.html tiedostoon importoidaan myös Redux Provider, jota voidaan käyttää HTML-tunnisteen kaltaisesti ja se asetetaan sovelluksen ympärille. Provider-tunnisteelle syötetään ominaisuutena luotu store (ks. kuvio 6).

```
import App from './App';
import store from './store.js';
import { Provider } from 'react-redux';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```

Kuvio 6. Redux storen tarjoaminen React-sovellukselle

Seuraavaksi projektiin luodaan Redux state slice. Slice tulee ensin nimetä ja sille tulee antaa alustava tilan arvo. Sen jälkeen sliceen voidaan luoda yksi tai useampi reducer-funktio, joiden sisään luodaan actioneita, joilla kuvataan, kuinka sovelluksen tilaa halutaan muuttaa. Tässä kohtaa tulee huomioida, että Redux noudattaa käytännettä muuttumattomasta tilasta (engl. immutable), eli kaikki päivitykset tilaan tulee tehdä ottamalla kopio vanhasta tilasta, johon muutokset tehdään ja

sen jälkeen kopio asetetaan uudeksi tilaksi. Koska käytämme Redux Toolkit-kirjastoa ja luomme slicen käyttämällä createSlice-metodia, tämä tapahtuu automaattisesti. Tämä ohjelmointirajapinta käyttää taustalla Immer-nimistä kirjastoa, joka antaa tekijän kirjoittaa koodia, joka mutatoi sovelluksen tilaa. Immer kuitenkin laskee kaikki muutokset, joita sovelluksen tilaan on tehty, ja palauttaa Reduxin käytänteiden mukaisen muuttumattoman tilan. Esimerkkisovelluksessa kirjoitettiin kaksi actionia, jotka hoitavat tehtävän lisäämisen ja poistamisen. Lopuksi actionin ja koko reducer viedään muun projektin käytettäväksi export-deklaraatiolla (ks. kuvio 7).

```
export const todoSlice = createSlice({
  name: 'todoList',
  initialState: {
    value: [
      { id: 1, name: 'Kirjoita opinnäytetyö' },
      { id: 2, name: 'Etsi harjoittelupaikkaa' },
      { id: 3, name: 'Valmistu' },
    ],
  },
  reducers: {
    addTodo: (state, action) => {
      state.value = [...state.value, { id: nanoid(), name: action.payload }];
    },
    deleteTodo: (state, action) => {
      state.value = state.value.filter((value) => value.id !== action.payload);
    },
  },
});

export const { addTodo, deleteTodo } = todoSlice.actions;
export default todoSlice.reducer;
```

Kuvio 7. Redux state slice havainnollistettuna

Lisäksi reducer tulee importoida sovelluksen tilalle, eli store.js tiedostoon. Antamalla luodun reducerin nimi parametrinä storelle, se tietää käyttää slicen reducer-funktiota sovelluksen tilan päivityksiin (ks. kuvio 8).

```
import { configureStore } from '@reduxjs/toolkit';
import todoReducer from './features/todoSlice.js';

export default configureStore({
  reducer: {
    todoList: todoReducer,
  },
});
```

Kuvio 8. Redux store havainnollistettuna

Viimeisenä Redux state slicen actioneita voidaan käyttää komponenteissa React Redux hookeina, jotka muistuttavat käytöltään Reactin omia komponentin tilanhallintaan käytettäviä hookeja. Käyttämällä useDispatch-hookia komponentissa voidaan lähettää luotuja actioneita, joilla muutetaan sovelluksen tilaa. Lisäksi voidaan käyttää useSelector-hookia, jolla sovelluksen tilan dataa voidaan lukea. Esimerkkisovelluksessa App.js tiedostoon useSelector-hookilla tuodaan sovelluksen storesta tehtävät, jotka map-funktion avulla tulostetaan omiin Todo-komponentteihinsa. Lisäksi storesta tuodaan tehtävätaulukon pituus, joka sijoitetaan muuttujaan, joka näkyy käyttöliittymässä jäljellä olevien tehtävien laskurina (ks. kuvio 9).

```
const todoList = useSelector((state) =>
  state.todoList.value.map((todo) => (
    <Todo id={todo.id} name={todo.name} key={todo.id} />
  ))
);

const todoCounter = useSelector(
  (state) => state.todoList.value.length + ' tehtävää jäljellä'
);
```

Kuvio 9. useSelector-hook havainnollistettuna

Todo-komponenttiin, jota käytetään tehtävien tulostamiseen, tuodaan deleteTodo-action, jolla sovelluksen storeen lähetetään poistettavan tehtävän id useDispatch-hookilla. Vastaavasti Form-komponenttiin tuodaan addTodo-action, joka lähettää sovelluksen storeen lisättävän tehtävän ni-

men useDispatch-hookilla. Form-komponentin omassa tilassa säilytetään käyttäjän syöte eli tehtävän nimi, jota addTodo-action käyttää, kun lomake toimitetaan eli käyttöliittymässä painetaan Lisää tehtävä” -painiketta (ks. kuvio 10).

```
export default function Form() {
  const [name, setName] = useState('');
  const dispatch = useDispatch();

  function handleChange(e) {
    setName(e.target.value);
  }

  function handleSubmit(e) {
    e.preventDefault();
    dispatch(addTodo(name));
    setName('');
  }
```

Kuvio 10. useDispatch-hook ja addTodo-action

Redux on pitkälle kehittynyt tilanhallintakirjasto, ja sen takia sen virallisella dokumentaatiolla selviää hyvin pitkälle, kun aloittaa toteuttamaan storea, reducereita ja actioneja. Redux Toolkit suoraviivaistaa tätä prosessia entisestään ja actioneiden yhdistäminen käyttöliittymään on tuttua, koska ne käyttäytyvät samalla tapaa kuin Reactin hookit. Muuttumattoman tilan hyvää käytännettä on helppo ylläpitää, koska createSlice-ohjelmointirajapinta ja sen taustalla toimiva Immer tekevät sen automaattisesti. Opinnäytetyön esimerkkisovellus on yksinkertainen ja siihen Reduxin implementointi oli melko helppoa. Monimutkaisemmissa sovelluksissa, joissa slicejen ja reducerien määrä moninkertaistuu, myös käyttöönottoon menevä aika kasvaa, koska Redux tuottaa huomattavan määrän boilerplate-koodia eli koodin toistoa. Redux on myös hyvin tarkka käytänteistään esimerkiksi slicen rakenteessa, joten se on rajoitettu muokattavuutensa osalta. Vaikka Redux vaatii opetteluunsa paljon aikaa, kehittäjä voi kokea sen systemaattisen tavan toteuttaa tilanhallinta olevan paras ratkaisu monimutkaisemmissa sovelluksissa. Koska Redux noudattaa tiukkaa arkkitehtuuria, sen tilan muutokset ovat ennustettavissa, joka taas helpottaa koodin ylläpitoa ja mahdollisten ongelmien ratkaisua.

5.4 Tilanhallinta Zustandilla

Ensin Zustand tulee asentaa React-projektiin ja se onnistuu antamalla terminaalissa komento:

```
npm install zustand
```

Zustand on opinnäytetyöhön valituista tilanhallintakirjastoista kooltaan pienin, joten sen lataamiseen ja asentamiseen kuluu erittäin lyhyt aika. Seuraavaksi Zustandin storea varten tulee luoda oma tiedostonsa projektin juurikansioon. Tämä perustuu Reactin tilanhallinnan hyvään käytänteeseen siitä, että store viedään komponenttihierarkian ulkopuolelle, jolloin siihen on helppo päästä käsiksi sitä tarvittavissa komponenteissa. Tiedostolle annetaan kuvaava nimi, tässä tapauksessa todoStore.js. Kuten mainittu, Zustandin store on hook, johon voi sijoittaa esimerkiksi objekteja ja funktioita (ks. kuvio 6). Esimerkkisovelluksen tehtävät sijoitettiin storeen sellaisenaan, eli taulukkona, joka sisältää tehtävän objektina, joka sisältää tehtävän id:n ja nimen. Lisäksi storeen sijoitettiin funktiot, jotka käsittelevät tehtävän lisäämisen ja poistamisen. Vastaavasti käyttäjän syöte eli tehtävän nimi säilytetään sovelluksen tilassa ja sen asettamiselle tarvittava funktio.

```
export const useTodoStore = create((set) => ({
  name: '',
  setName: (name) => set({ name }),
  todos: [
    { id: 1, name: 'Kirjoita opinnäytetyö' },
    { id: 2, name: 'Etsi harjoittelupaikkaa' },
    { id: 3, name: 'Valmistu' },
  ],
  addTodo: (name) =>
    set((state) => ({ todos: [...state.todos, { id: nanoid(), name }] })),
  deleteTodo: (id) =>
    set((state) => ({ todos: state.todos.filter((todo) => todo.id !== id) })),
}));
```

Kuvio 11. Zustand store havainnollistettuna

Seuraavaksi Zustand store tuodaan haluttuun komponenttiin import-deklaraatiolla. Sen jälkeen storen arvoja ja funktiota voidaan käyttää React hookin kaltaisesti. Esimerkiksi sovelluksen Form-

komponenttiin tuotiin storesta tehtävän nimi, sen asettamiseen tarvittava funktio ja tehtävän lisäämiseen tarvittava funktio (ks. kuvio 7).

```
export default function Form() {
  const addTodo = useTodoStore((state) => state.addTodo);
  const name = useTodoStore((state) => state.name);
  const setName = useTodoStore((state) => state.setName);

  function handleChange(e) {
    setName(e.target.value);
  }

  function handleSubmit(e) {
    e.preventDefault();
    addTodo(name);
    setName('');
  }
```

Kuvio 12. Form.js tiedosto

Aiemmin Form-komponentilla oli oma tilansa, jossa se säilytti tehtävän nimen. Lisäksi se sai sovelluksen pääkomponentilta ominaisuutena tehtävän lisäämiseen tarvittavan funktion. Nyt Form-komponentti voi hoitaa tehtävän lisäämisen itsenäisesti, kun taas pääkomponenttiin tuodaan sovelluksen tilasta tehtävät, jotka siellä tulostetaan map-funktiota käyttäen uusiksi komponenteiksi. Tehtäväkomponentteihin storesta tuotiin poistamiseen tarvittava funktio. Näin koodi selkeytyi välittömästi, kun komponenttien ominaisuuksia ei enää tarvinnut välittää toisilleen, vaan jokainen niistä saa tarvittavan tiedon storesta ja kykenevät päivittämään sitä.

Zustandin käytön oppii nopeasti, koska kuten mainittu, se muistuttaa paljon Reactin omaa komponentin tilanhallintaan käytettävää hookia. Vaikka Zustand ei pakota kehittäjälle mitään käytänteitä, on hyvä ottaa huomioon Flux-arkkitehtuurista perustuvia käytänteitä. Zustandia käyttäessä tulee luoda vain yksi store, koska erityisesti suuremmissa ja monimutkaisemmissa sovelluksissa se helpottaa testausta, sillä mahdolliset ongelmat tilanhallinnassa voidaan paikantaa silloin yhteen paikkaan. Toinen tärkeä käytänne on käyttää storen ja sen toimintojen luomiseen set-metodia. Sen käyttäminen varmistaa, että päivitykset sovelluksen tilaan yhdistyvät vanhan tilan kanssa.

Tämä perustuu käytänteeseen siitä, että Zustand storen tila tulee olla muuttumaton (engl. immutable), eli vanhasta tilasta otetaan kopio, johon lisätään muutokset, joka taas asetetaan sovelluksen tilaksi.

5.5 Tilanhallinta MobX:llä

MobX saadaan asennettua React-projektiin ajamalla terminaalissa komento:

```
npm install mobx mobx-react –save
```

Komento asentaa MobX-kirjaston sekä Reactille tarvittavat paketit, joilla käyttöliittymä ja MobX saadaan sidottua yhteen. Esimerkkikomentoon lisäämällä -lite voidaan asentaa kirjaston lite-versio, joka tukee käyttöliittymän sidontaa ainoastaan Reactin funktiokomponenteille. Esimerkkikomento tukee sidontoja myös luokkatyyppisille komponenteille.

Kuten edeltävissäkin tilanhallintakirjastoissa, MobX storea varten luodaan oma tiedostonsa projektin juureen, jotta se saadaan komponenttihierarkian ulkopuolelle. Poiketen opinnäytetyön muista tilanhallintakirjastoista, MobX ei nimeä hyväksi käytänteeksi yhden storen arkkitehtuuria. Storeja voi olla useampi, mutta yleinen käytänne MobX-sovellukselle on sisältää vähintään kaksi storea. Toinen storeista toimii domain-storena, jonka tarkoitus on säilyttää sovelluksen toimintaan vaikuttavia tietoja. Esimerkkisovelluksessa nämä tiedot ovat tehtävät, joita käyttäjä tallentaa. Toinen store toimii UI-storena, johon tallennetaan kaikki istuntoon liittyvät tiedot. Näitä ovat esimerkiksi käyttöliittymään vaikuttavat tekijät, kuten valittu kieli tai teema. Esimerkkisovelluksessa UI-storelle ei ole käyttöä, joten projektin juureen luotiin yksi tiedosto, joka nimettiin kuvaavasti store.js. Store-tiedostoon importoidaan ensimmäiseksi makeAutoObservable-funktio. Funktiolla voidaan tehdä store-luokan tehtävälistasta ja siihen liittyvistä actioneista automaattisesti havaittavia (engl. observable). Storeen siis alustettiin lista, johon tehtävät tallennetaan objekteina ja lisäksi kirjoitettiin kaksi funktiota tehtävien lisäämiseen ja poistamiseen (ks. kuvio 13).

```
import { makeAutoObservable } from 'mobx';
import { nanoid } from 'nanoid';

class TodoStore {
  todos = [
    { id: 1, name: 'Kirjoita opinnäytetyö' },
    { id: 2, name: 'Etsi harjoittelupaikkaa' },
    { id: 3, name: 'Valmistu' },
  ];

  constructor() {
    makeAutoObservable(this);
  }

  addTodo(name) {
    this.todos = [...this.todos, { id: nanoid(), name: name }];
  }

  deleteTodo(id) {
    this.todos = this.todos.filter((todo) => id !== todo.id);
  }
}

const todoStore = new TodoStore();
export default todoStore;
```

Kuvio 13. Mobx store havainnollistettuna

Seuraavaksi MobX store saadaan käyttöön React-komponentissa importoimalla store-tiedosto ja mobx-react-kirjaston observer-funktio haluttuun komponenttiin. Paketoimalla komponentti observer-funktiolla React-komponentista saadaan reaktiivinen storen muutoksiin. Sen jälkeen storen arvoja ja actioneita saadaan käytettyä viittaamalla importoituun todoStoreen. App.js komponenttiin tuodaan storesta tehtävät, jotka map-funktiolla tulostetaan yhteen muuttujaan, joka näytetään käyttöliittymässä. Lisäksi jäljellä olevat tehtävät saadaan laskettua storen tehtävälistan pituudesta (ks. kuvio 14).

```
import { observer } from 'mobx-react';
import todoStore from './store';

const App = observer(() => {
  const todoList = todoStore.todos.map((todo) => (
    <Todo id={todo.id} name={todo.name} key={todo.id} />
  ));

  const todoCounter = todoStore.todos.length + ' tehtävää jäljellä';
```

Kuvio 14. Observer-funktion ja storen käyttö komponentissa

Tehtävän lisäämiseen tarvittavaa funktiota kutsutaan Form-komponentissa lomakkeen lähettämisen käsittelevässä funktiossa (ks. kuvio 15). Koska App.js tiedosto on paketoitu observable-funktioon, joka tekee siitä reaktiivisen, se päivittää uuden tehtävän käyttöliittymään, kun Form-komponentti lisää uuden tehtävän storen tehtävälistaan. Vastaavasti tehtäväkomponenteissa kutsutaan tehtävän poistamiseen tarvittavaa funktiota.

```
import todoStore from '../store';

const Form = observer(() => {

  const [name, setName] = useState('');

  function handleSubmit(e) {
    e.preventDefault();
    todoStore.addTodo(name);
    setName('');
  }
```

Kuvio 15. Mobx actionin käyttö komponentissa

MobX:n implementointiin käytettiin eniten aikaa tutkimuksen tilanhallintakirjastoista ja suurin syy siihen on sen eroavaisuus muista tutkimukseen käytetyistä kirjastoista. Sen tapa käyttää luokkia storen datan säilytykseen ja manipulointiin suorien funktioiden sijasta on huomattava eroavaisuus muihin kirjastoihin verrattaessa. MobX-kirjastoa käyttäessä ei myöskään tarvitse miettiä tilan muuttumattomuutta (engl. immutable), vaan kehittäjä voi huoletta kirjoittaa tilaa suoraan muta-

toivaa koodia. Ensimmäistä kertaa käytettäessä, Mobx:n vapaus toteuttaa tilanhallintaa voi vaikuttaa ylivoimaiselta. MobX ei pakota käytänteitä ja virallisessa dokumentaatiossaan korostaa sitä vain yhdeksi tavaksi toteuttaa tilanhallintaa MobX:llä. Verkosta löytää loputtomasti tutoriaaleja, joissa käytänteet vaihtelevat suuresti. Jos kehittäjä etsii systemaattista tapaa toteuttaa tilanhallintaa, Mobx ei välttämättä ole helpoin ratkaisu. Tutkimuksen muut tilanhallintakirjastot, jotka ovat hieman itsepäisempiä käytänteissään, voivat vähentää virheiden määrää ja näin helpottaa esimerkiksi sovelluksen testausta. Toisaalta toteuttamisen vapaus näkyy koodin luettavuudessa, koska boilerplate-koodin määrä vähenee.

6 Tulosten vertailu

Tilanhallinta on toteutettu tutkimukseen valituilla tilanhallintakirjastoilla, ja jokaisen kirjaston toteutuksen yhteydessä hyviä käytänteitä on tuotu esiin, sekä alustavasti pohdittu niiden vahvuuksia ja mahdollisia heikkouksia. Seuraavaksi otetaan toteutuksen alussa nimetyt kolme pääaihealutta, dokumentaatio, ajankäyttö ja käyttäjäkokemus, ja verrataan tilanhallintakirjastoja toisiinsa näillä kolmella alueella ja tehdään niistä alustavia johtopäätöksiä.

6.1 Dokumentaatio

Redux on tutkimuksen kolmesta tilanhallintakirjastosta latauskertoihin verrattuna suosituin, ja se näkyy myös dokumentaation tasossa. Redux on pitkälle kehittynyt tilanhallintakirjasto ja sen dokumentaatiota ylläpidetään aktiivisesti sen kehityksen tasalla. Reduxin käyttö Reactin kanssa huomioidaan erityisen hyvin, koska React Redux kirjastolle on oma erillinen dokumentaationsa. Dokumentaatiossa otetaan huomioon sekä uuden projektin aloittaminen Reduxilla sekä sen lisääminen olemassa olevaan projektiin. Itse tilanhallinnan implementointi on helppoa, sillä esimerkiksi dokumentaatiosta löytyvät tutoriaalit ohjaavat kehittäjää suoraan luomaan oikeanlaisen rakenteen storelle ja sliceille, ja auttaa niiden yhdistämisessä ja käyttöönotossa komponenteissa. Kehittäjälle jää jäljelle omien funktioiden kirjoittaminen sliceihin ja niiden konfigurointi käyttöliittymään. Suosionsa takia Reduxin käyttöön löytyy valtavasti apua myös esimerkiksi Stack Overflow-sivustolta, jos vastausta ei virallisesta dokumentaatiosta löydy.

Zustand on latauskertoja katsoessa ohittanut MobX:n suosiossa, mutta sen dokumentaatio on kapein näistä kolmesta tilanhallintakirjastosta. Osasyy tähän on se, että Zustand on yksinkertaisin

näistä kolmesta kirjastosta, kun esimerkiksi storea ei tarvitse Redux Providerin tavoin tarjota sovellukselle tai MobX:n tapaan miettiä storen havaittavuutta. Näin välivaiheita mitä kahden muun tilanhallintakirjaston dokumentaatiosta löytyy jää Zustandin tapauksessa pois. Zustandin käyttö Reactin kanssa huomioidaan hyvin, koska Zustand on dokumentaationsa mukaan kehitetty huomioiden ongelmat, joita muiden kirjastojen kanssa on esiintynyt. Dokumentaatio on suoraviivainen ja yksinkertainen, jossa Reduxin dokumentaation tapaan kehittäjälle jää pohdittavaksi ainoastaan omien funktioiden luominen ja niiden käyttäminen hookina käyttöliittymässä. Tutkimusta tehdessä huomattiin, että yksityiskohtaisten virheiden sattuessa yhteisön puolesta apua sai etsiä tarkkaan ja yleensä vastauksen sai jonkin isomman kokonaisuuden yhteydessä. Zustand on kompakti tilanhallintakirjasto, mutta yhteisö on kehittänyt siihen paljon kolmannen osapuolen kirjastoja, joilla sen toiminnallisuuksia voi lisätä. Esimerkiksi halutessaan kehittäjä voi ladata Mobz-nimisen kirjaston, joka yhdistää Zustandin tyylin luoda store MobX:n ohjelmointirajapintaan.

Vaikka MobX dokumentaatiossa tuodaan esille, että se on itsenäinen tilanhallintakirjasto, joka toimii ilman Reactia, sen esimerkit storen ja sen actioneiden luomisesta toimivat Reactin kanssa sellaisenaan. React on huomioitu dokumentaatiossa ja esimerkiksi MobX:n integraatiolle Reactiin on oma sivunsa dokumentaatiossa. Tutkimuksen muiden tilanhallintakirjastojen tapaan MobX dokumentaation avulla pärjää storen luomisessa, mutta tutkimusta tehdessä todettiin sen käyttöönoton komponenteissa olevan ongelmallista. Dokumentaation viimeisimmästä päivityksestä ei löytynyt mainintaa, ja yhteisön puolesta toteutukseen löytyi paljon erilaisia vaihtoehtoja ja mielipiteitä. Tämä johtuu todennäköisesti siitä, että MobX ei noudata niin vahvoja käytänteitä verrattuna esimerkiksi Reduxiin. Lisäksi MobX:n tapa toteuttaa tilanhallintaa poikkeaa vahvasti muista tutkimuksen tilanhallintakirjastoista, joten sekin vaikutti dokumentaation ymmärrettävyyteen kehittäjän näkökulmasta.

6.2 Ajankäyttö

Reduxin opetteluun, käytänteiden sisäistämiseen ja tilanhallinnan toteutukseen kului aikaa noin työpäivän verran. Kuten todettu, informatiivinen dokumentaatio helpotti asiaa huomattavasti. Storen luonti, slicen ja sen reducereiden tekeminen ja tarjoaminen storelle ja lopuksi storen tarjoaminen sovellukselle käytettäväksi tuottivat välivaiheita, jotka lisäsivät ajankäyttöä verrattuna muihin kirjastoihin. Actioneiden käyttäminen ja lähettäminen käyttöliittymästä koettiin tehok-

kaimmaksi toteuttaa tutkimuksen kolmesta tilanhallintakirjastosta. Redux Tookitin käyttöä suositellaan kehittäjien toimesta, koska se yksinkertaistaa yleisimpiä toimenpiteitä, ja tutkimuksessa todettiin koodin toiston olevan minimaalista.

Zustandin opiskeluun ja sillä tilanhallinnan toteutukseen kului aikaa suunnilleen saman verran kuin Reduxilla toteuttaessa. Itse storen luonti ja sen käyttö sovelluksen käyttöliittymässä oli ajankäytöltään tehokkainta tutkimuksen tilanhallintakirjastoista. Vaikka Zustand hookien käyttäminen oli helppoa, koska se muistuttaa paljon Reactin omaa komponentin tilanhallintaan käytettävää hookia, sen koettiin tutkimusta tehdessä tuottavan näistä kolmesta kirjastosta eniten koodin toistoa.

MobX:n opiskeluun kului eniten aikaa tutkimuksen tilanhallintakirjastoista. Kehittäjän näkökulmasta, joka valitsee ensimmäistä tilanhallintakirjastoaan käytettäväksi, MobX:n monipuolisuus toteuttaa tilanhallinta vaatii eniten sisäistämistä. Luokkatyyppisen storen luominen oli kuitenkin kolmesta kirjastosta tehokkainta, jos luokkien käyttäminen on kehittäjälle ennestään tuttua. Actioneiden käyttäminen sovelluksen käyttöliittymässä tuotti tutkimuksen tilanhallintakirjastoista vähiten koodin toistoa.

6.3 Käyttäjäkokemus

Redux pitää kiinni vahvasti käytänteistään, eikä jätä kehittäjälle muokattavuuteen tilaa siinä, kuinka tilanhallinnan haluaa toteuttaa. Tutkimuksessa tämä osoittautui kuitenkin vahvuudeksi, jos kehittäjän lähtökohta on se, että Redux on ensimmäinen tilanhallintakirjasto, johon kehittäjä tutustuu. Sen systemaattinen tapa toteuttaa tilanhallinta on helppo ymmärtää ja slicen ja sen reducer-funktioiden luominen sekä niiden kutsuminen käyttöliittymässä useDispatch- ja useSelector-hookeja käyttäen on hyvin itsensä selittävää. Kehittäjä pystyy myös lataamaan selaimeensa Redux DevTools lisäosan, jonka avulla pystyy tarkkailemaan actioneiden lähettämiä tietosisältöjä ja niin sanotusti aikamatkustamaan perumalla actioneita. Tämä helpottaa omien reducer-funktioiden koodaamista ja mahdollisten virheiden selvittämistä.

Zustand on tutkimuksen tilanhallintakirjastoista yksinkertaisin ymmärtää ja se koettiin tutkimuksen aikana luontevana siirtymänä Reactin omasta useState-hookista kohti koko sovelluksen tilanhallintaa. Kuten mainittu, storen käyttöönotto komponenteissa ei tarvitse muita välivaiheita, kuin

actioneiden kutsumisen useStore-hookia käyttäen. Zustand jättää yhden storen käytännettä lukuun ottamatta kehittäjälle paljon tilaa muokattavuudelle ja storen voi muodostaa täysin mieleisekseen. Erityisesti pienemmissä sovelluksissa, kuten tutkimuksen esimerkkisovelluksessa, Zustandin kaltainen pieni kirjasto on nopea implementoida. Vastaavasti monimutkaisemmissa sovelluksissa, joissa toiminnallisuutta vaaditaan myös enemmän tilanhallintakirjastola, Reduxin kaltainen vahvaa arkkitehtuuria noudattava ja sitä myötä helpommin ennustettava tilanhallintakirjasto voi olla parempi valinta.

MobX ei pakota kehittäjälle niin vahvoja käytänteitä, ja jos kehittäjä on entuudestaan pätevä olio-ohjelmoinnissa ja ymmärtää erityisesti luokkapohjaisuudesta, MobX:n tapa toteuttaa store on oikea valinta. Jos kehittäjän preferenssi on käyttää sovelluksessa useampaa storea, silloin kannattaa valita MobX. Tutkimusta tehdessä siirryttiin single source of truth-arkkitehtuuria noudattavista Reduxista ja Zustandista viimeisenä MobX:n, jonka monipuolisuus tilanhallinnassa ja storejen määrässä vaikutti ylivoimaiselta sisäistää. Tämä on kuitenkin täysin subjektiivista ja riippuu kehittäjän mielipiteestä. MobX käyttäessä tilan muuttumattomuutta (engl. immutable) ei tarvitse miettiä, joka voi helpottaa actioneiden toteutusta. Vastapainona tilanhallinnan ennustettavuus kärsii.

7 Pohdinta

Tutkimuksen tilanhallintakirjastoja on nyt vertailtu kolmella aihealueella ja alustavia johtopäätöksiä on tehty. Seuraavaksi opinnäytetyössä pohditaan tutkimuksen luotettavuutta ja eettisyyttä, tarkastellaan tuloksia suhteessa teoreettiseen viitekehykseen ja tehdään lopulliset johtopäätökset sekä pohditaan kehittämismahdollisuuksia.

7.1 Luotettavuus ja eettisyys

Opinnäytetyön luotettavuuteen vaikuttaa käytetyn esimerkkisovelluksen yksinkertaisuus. Koska yhden tilanhallintakirjaston kaikkia ominaisuuksia ei päästy yksinkertaisen esimerkkisovelluksen takia hyödyntämään, se vaikutti käyttäjäkokemukseen kyseisen tilanhallintakirjaston kohdalla. Sen takia tuloksia kyseisen kirjaston kohdalta ei voida pitää täysin pätevinä. Opinnäytetyöhön valituilla tilanhallintakirjastoilla toteutettu tilanhallinta perustuu vahvasti niiden omaan dokumentaatioon ja opinnäytetyön tietoperustassa omaksuttuihin hyviin käytänteisiin. Tämän vuoksi opinnäytetyön toteutuksen vaihe on nyt relevanttia tietoa, mutta tulee huomioida, että jokainen opinnäytetyön

tilanhallintakirjastoista on kehittyvää teknologiaa. Tämä johtaa siihen, että tulevaisuudessa opinnäytetyön toteutuksen osa ei välttämättä ole paikkansapitävää tietoa. Sama koskee Reactia viitekehyksenä ja tässä opinnäytetyössä käsiteltyjä teknologioita, joita React käyttää.

Opinnäytetyössä noudatettiin hyvää tieteellistä käytäntöä, eli opinnäytetyö on toteutettu rehellisesti, huolellisesti ja tarkkuudella. Tutkimuksen tulokset on raportoitu totuudenmukaisesti kehittäjän näkökulmasta, eli opinnäytetyön vertailevassa osassa tilanhallintakirjastoja on vertailtu yleisesti kehittäjän näkökulmasta eikä opinnäytetyön tekijän mielipiteen näkökulmasta.

7.2 Tulosten tarkastelu suhteessa teoreettiseen viitekehykseen

Redux noudattaa opinnäytetyön tietoperustassa alustettua arkkitehtuuria, jossa sen storessa säilytetään sovelluksen tila, jota muutetaan reducereilla, jotka ottavat sovelluksen tilan ja sitä muuttavan actionin ja jonka jälkeen se palauttaa uuden muuttuneen tilan. Redux Toolkitin käyttäminen nopeuttaa tämän arkkitehtuurin toteuttamista entisestään. Tulee huomioida, että Redux Toolkitin käyttämät ohjelmointirajapinnat toteuttavat paljon Reduxin toimintoja niin sanotusti taustalla. Kehittäjän tulee ottaa tämä huomioon opiskellessaan Reduxia ymmärtääkseen käytänteiden tarkoituksen, vaikka se ei suoraan sovelluksen koodista selviä.

Zustand, kuten myös Redux, noudattaa opinnäytetyön tietoperustassa läpikäytyä yhden storen periaatetta ja storen datan tulee aina olla muuttumatonta (engl. immutable). Vaikka Zustandin ja Redux Toolkitin ohjelmointirajapinnat antavat kirjoittaa tilaa suoraan muuttavaa koodia, on ymmärrettävä, miksi tilan muutokset tehdään muuttumattomasti taustalla. Jos tilan muutokset tehtäisiin muuttuvasti, se aiheuttaisi React komponenttien uudelleen päivityksen joka kerta, vaikka muutoksia itse tilaan ei syntyisikään. Tämä vaikuttaisi suoraan esimerkiksi sovelluksen suorituskykyyn. MobX:n tapauksessa observer-funktio pitää huolen komponentin päivittämisestä muutoksen tapahtuessa.

Teoriaosassa väitettiin MobX:n saavuttavan muita tilanhallintakirjastoja nopeammin ja vähemmällä koodilla saman lopputuloksen, koska se sallii tilaa suoraan muuttavan koodin kirjoittamisen. Mutta kuten todettu, Redux Toolkitin ja Zustandin ohjelmointirajapinta sallii sen myös. Siksi täytyy todeta, ettei sitä voi yksin nimetä MobX:n vahvuudeksi.

7.3 Johtopäätökset ja kehittämisehdotukset

Opinnäytetyön tavoite oli toteuttaa tilanhallinta esimerkkisovellukseen tutkimukseen valituilla kolmella tilanhallintakirjastolla ja löytää niiden vahvuudet ja mahdolliset heikkoudet. Lisäksi valittujen tilanhallintakirjastojen hyviä käytänteitä tuotiin esille tavoitteena auttaa kehittäjiä valitsemaan sopiva tilanhallintakirjasto omaan projektiinsa.

Tutkimuksen tuloksista saatiin tietoa, missä tilanteessa kukin tilanhallintakirjasto on hyvä valinta ja kuinka tilanhallinta niitä käyttäen tulisi toteuttaa. Zustand on tutkimuksen kirjastoista paras valinta kehittäjälle, joka tarvitsee tilanhallintaa yksinkertaiseen sovellukseen ja etsii ensimmäistä tilanhallintakirjastoa opeteltavaksi. Zustandin opettelu on pienin kynnys tutkimuksen kirjastoista, koska sen tilanhallinta muistuttaa paljon Reactin omaa komponentin tilanhallintaa ja storen luominen ja käyttöönotto vaatii vähiten välivaiheita. Redux on luonteva askel Zustandin jälkeen, sillä ne noudattavat paljon samoja käytänteitä. Redux tarjoaa säännöllisemmän tavan toteuttaa tilanhallintaa ja sen selvä arkkitehtuuri auttaa kirjoittamaan saavutettavaa koodia, joka on testattavaa ja mahdollisten virheiden paikantaminen helpottuu. Tämän takia monimutkaisemmissa sovelluksissa Redux on oikea valinta ja lisäksi se tarjoaa parhaan yhteisön tuen dokumentaationsa ja Redux DevToolien puolesta. MobX tarjoaa vaihtoehdon tutkimuksen muihin funktionaaliseen ohjelmointiin nojaaviin tilanhallintakirjastoihin. MobX:n opetteluun vaadittava kynnys pienenee, jos kehittäjä on jo ennestään harjaantunut olio-ohjelmoinnissa. Kuten opinnäytetyön tietoperustassa sekä tutkimusta tehdessä todettiin, MobX tuotti tutkimuksen kirjastoista vähiten boilerplate-koodia eli koodin toistoa. Tämä voi auttaa pitämään yksinkertaisen sovelluksen koodin saavutettavana, mutta MobX on riittävän monipuolinen monimutkaisemmankin sovelluksen tilanhallintaan.

Opinnäytetyön esimerkkisovellus oli hyvin yksinkertainen, joka toimi tilanhallinnan havainnollistamiseen Reduxin ja Zustandin kanssa. Yksinkertaisuus oli ongelma ainoastaan MobX:n kanssa. Kuten tietoperustassa todettiin, MobX sovelluksen tila on observable, jolle voidaan määrittää actioneiden lisäksi sivuvaikutuksia. Esimerkkisovelluksen yksinkertaisuuden vuoksi MobX:n koko potentiaalia ei päästy käyttämään. Sovelluksen tilasta olisi voitu esimerkiksi sen vaihtuessa laskea automaattisesti jokin arvo tai lähettää käyttäjälle automaattinen ilmoitus tilan vaihtuessa tiettyyn arvoon. Opinnäytetyössä käytettiin sovelluksen ja tilanhallinnan tekemiseen JavaScript-ohjelmointikieltä. Esimerkkisovelluksen yksinkertaisuus vaikuttaa myös muiden tilanhallintakirjastojen vertailun tarkkuuteen. Kuten opinnäytetyön tietoperustassa alustettiin, moni React-kehittäjä valitsee

nykyään JavaScriptin sijasta TypeScriptin. TypeScriptin käyttäminen olisi todennäköisesti vaikuttanut kirjastokohtaisesti niiden ajankäyttöön ja käyttäjäkokemukseen. Jos esimerkkisovelluksessa olisi käytetty tyypitystä, se olisi voinut vähentää käyttäjävirheitä tutkimusta tehdessä. Kehittäjä olisi esimerkiksi huomannut nopeammin, jos sovelluksen funktio palauttaa väärän arvon.

Tutkimusta voidaankin lähteä jatkokehittämään esimerkiksi vertailemalla kirjastokohtaista toteutusta käyttäen TypeScriptiä JavaScriptin sijasta. Vertailuun voi käyttää tämän tutkimuksen aihealueita tai vertailla esimerkiksi suorituskykyä. Toinen jatkokehittämismahdollisuus olisi näiden kirjastojen implementointi johonkin monimutkaisempaan sovellukseen tai tutkimus siitä, kuinka nämä tilanhallintakirjastot saadaan kommunikoimaan ja toimimaan jostain serveriltä saadun datan kanssa. Opinnäytetyössä käytettiin Reactia viitekehyksenä, mutta opinnäytetyöhön valittujen tilanhallintakirjastojen käyttöä voisi vertailla jonkun muun viitekehyksen kanssa. Tutkimuksen tuloksia voi hyödyntää kehittäjät, jotka pyrkivät valitsemaan omaan projektiinsa sopivan tilanhallintakirjaston.

Lähteet

Elrom, E. 2021. React and Libraries: Your Complete Guide to the React Ecosystem. Apress. Viitattu 22.2.2023. https://janet.finna.fi, Skillsoft Books ITPro (Skillport Platform).

Freeman, A. 2019. Essential TypeScript From Beginner to Pro. Apress. Viitattu 31.3.2023. https://janet.finna.fi, Skillsoft Books ITPro (Skillport Platform).

Freeman, A. 2019. Pro React 16. Apress. Viitattu 1.3.2023. https://janet.finna.fi, Skillsoft Books ITPro (Skillport Platform).

Gackenheimer, C. 2015. Introduction to React. Apress. Viitattu 21.2.2023. https://janet.finna.fi, Skillsoft Books ITPro (Skillport Platform).

Garreau, M. Faurot, W. 2018. Redux in Action. Manning Publications. Viitattu 2.3.2023. https://janet.finna.fi, Skillsoft Books ITPro (Skillport Platform).

Introduction. N.d. Zustand-tilanhallintakirjaston virallinen dokumentaatio kirjaston käyttöönotosta. Viitattu 2.3.2023. https://docs.pmnd.rs/zustand/getting-started/introduction.

Juhila, K. N.d. Laadullisen tutkimuksen verkkokäsikirja. Laadullisen tutkimuksen ominaispiirteet. Tietoarkisto. Viitattu 14.3.2023. https://www.fsd.tuni.fi/fi/palvelut/menetelmaopetus/kvali/mita-on-laadullinen-tutkimus/laadullisen-tutkimuksen-ominaispiirteet/.

Lee, J. Wei, T. Mukhiya, S. 2019. Redux Quick Start Guide: A Beginner's Guide to Managing App State with Redux. Packt Publishing. Viitattu 4.4.2023. https://janet.finna.fi, EBook Central Academic Complete International Edition.

Npm trends. N.d. Npm-paketinhallintajärjestelmällä ladattujen kirjastojen latausmäärien vertailuun tehty verkkosivu. Viitattu 14.3.2023. https://npmtrends.com/mobx-vs-redux-vs-zustand.

Podila, P. Weststrate, M. 2018. MobX Quick Start Guide: Supercharge the Client State in Your React Apps with MobX. Packt Publishing. Viitattu 2.3.2023. https://janet.finna.fi, EBook Central Academic Complete International Edition.

Ranjan, A. Sinha, A. Battewad, R. 2020. JavaScript for Modern Web Development: Building a Web Application Using HTML, CSS and JavaScript. BPB Publications. Viitattu 22.2.2023. https://janet.finna.fi, Skillsoft Books ITPro (Skillport Platform).

Readme.md. 2023. Zustand-tilanhallintakirjaston dokumentaatio GitHub-sivustolla. Viitattu 15.3.2023. https://github.com/pmndrs/zustand/blob/main/readme.md.

README MobX. N.d. MobX-tilanhallintakirjaston virallinen dokumentaatio verkkosivulla. Viitattu 2.3.2023. https://mobx.js.org/README.html.

Saaranen-Kauppinen, A. Puusniekka, S. 2006. KvaliMOTV – 1.2.2 Laadullisen tutkimuksen elementit. Tampere. Yhteiskuntatieteellinen tietoarkisto. Viitattu 28.3.2023. https://www.fsd.tuni.fi/menetelmaopetus/kvali/L1_2_2.html.

Saaranen-Kauppinen, A. Puusniekka, S. 2006. KvaliMOTV – 2.3.1 Tutkimusongelmat. Tampere. Yhteiskuntatieteellinen tietoarkisto. Viitattu 15.3.2023. https://www.fsd.tuni.fi/menetelmaopetus/kvali/L2_3_1.html.

Shute, Z. 2019. Advanced JavaScript. Packt Publishing. Viitattu 29.3.2023. https://janet.finna.fi, EBook Central Academic Complete International Edition.

Stack Overflow Developer Survey 2022. 2022. Stack Overflow -sivuston toteuttama kehittäjäkysely toukokuussa 2022. Viitattu 14.3.2023. https://survey.stackoverflow.co/2022/.

Thomas, M. 2018. React in Action. Manning Publications. Viitattu 1.3.2023. https://janet.finna.fi, Skillsoft Books ITPro (Skillport Platform).