Tampereen yliopisto

Vilma Salmikuukka

REACTIN TILANHALLINTAONGELMAT JA NIIDEN RATKOMINEN

Informaatioteknologian ja viestinnän tiedekunta
Kandidaattitutkielma
Joulukuu 2020

TIIVISTELMÄ

Vilma Salmikuukka: Reactin tilanhallintaongelmat ja niiden ratkominen
Kandidaattitutkielma
Tampereen yliopisto
Tietotekniikan tutkinto-ohjelma
Joulukuu 2020

Nykyajan verkkosivuilta vaaditaan näyttäviä ja monimutkaisia toiminnallisuuksia sekä interaktiivisia käyttöliittymiä. Sovellukset sisältävät kuitenkin suuria määriä tietoa, jonka käsitteleminen on kriittistä sovelluksen ylläpidon ja skaalautuvuuden kannalta. Kaikki sovelluksen sisäiset tiedot tiettynä ajanhetkenä muodostavat sovelluksen tilan. Tässä työssä tutkitaan React-sovellusten tilanhallinnassa ilmeneviä ongelmia. Työn tavoitteena on selvittää, kuinka näitä ongelmia voidaan ratkaista ja ennaltaehkäistä.

Tilanhallintaan liittyy tiedon välittäminen ohjelman sisällä sekä tilan varastointi ja päivittäminen. Kun sovelluksen eri osien välillä liikkuu paljon tietoa, ohjelmasta tulee nopeasti hyvin vaikeaselkoista. Työssä tutkitaan eri keinoja, joiden avulla tiedonvälitystä voidaan selkeyttää. Näistä merkittävimmiksi havaittiin JavaScriptin spread-operaattori, jonka avulla tietoa voitiin välittää yksinkertaisemmassa muodossa, sekä Reactin tarjoama context-työkalu, jonka avulla välitettävän tiedon määrä vähentyi.

Työssä havaittiin, että perinteinen tilan varastoiminen hajautetusti eri puolille ohjelmaa aiheutti liikaa riippuvuuksia ohjelman eri osien välille. Tästä seurasi vaikeasti seurattavia ja virhealttiita tapahtumaketjuja tilan päivittyessä. Työssä löydettiin yksittäisiin ongelmiin teknisiä ratkaisuja, mutta käytännön tasolla ohjelman hahmottamisen ja ylläpidon havaittiin muuttuvan lähes mahdottomaksi ongelmien kasaantuessa. Tähän parhaaksi ratkaisuksi havaittiin Redux-kirjasto, joka tarjoaa sovellukselle globaalin tietovaraston. Kun ohjelman tila säilötään keskitetysti yhdessä paikassa, ohjelman eri osat voivat käsitellä globaalia tilaa toisistaan riippumatta. Työssä päädyttiin johtopäätökseen, että pienissä sovelluksissa Reactin tarjoamat ominaisuudet ja työkalut ovat riittäviä, mutta suuremmissa sovelluksissa tilanhallinnan kirjastojen käyttö on lähes välttämätöntä.

Avainsanat: web-ohjelmointi, käyttöliittymäsovellus, React

SISÄLLYSLUETTELO

1 Johdanto

Monet nykyajan sovellukset tarjoavat interaktiivisen käyttöliittymän, joka mahdollistaa datan dynaamisen päivittämisen ja käsittelyn. Lisäksi useat sovellukset sisältävät suuria määriä monimutkaista dataa. Tämän datan esittäminen käyttäjälle siten, että se on helposti ymmärrettävissä ja intuitiivisesti käytettävissä onkin yksi käyttöliittymien rakentamisen suurimmista haasteista. [1]

Se, miten sovellus sisäisesti käsittelee tietoa, on kriittistä käyttöliittymän ylläpidon ja sovelluksen skaalautuvuuden kannalta. Kaikki käyttöliittymäsovelluksen sisäiset tiedot tiettynä ajanhetkenä muodostavat sovelluksen tilan [1]. Tilan hallintaan liittyy tilan varastointi ja tiedon välittäminen sovelluksen sisällä sekä tilan päivittäminen muutoksien tapahtuessa.

Tämän tutkielman tarkoitus on esitellä React-sovellusten tilanhallintaan liittyviä ongelmia ja perehtyä siihen, miten näitä ongelmia voidaan ratkaista. Tilanhallinnan ongelmia ilmenee etenkin silloin, kun sovelluksessa liikkuu suuria määriä tietoa ja kun samaa tietoa käsitellään useassa eri paikassa. Jokainen yksittäinen ongelma on teknisesti ratkaistavissa, mutta ongelmien kasaantuessa ohjelman hahmottaminen ja ylläpitäminen muuttuu käytännössä mahdottomaksi. Työn tavoitteena on myös tarkastella kolmannen osapuolen tarjoamia työkaluja tilanhallintaa varten.

Työn alku käsittelee Reactia yleisellä tasolla ja havainnollistaa React-sovelluksen rakennetta. Seuraavaksi perehdytään tiedon välittämiseen sovelluksen sisällä ja esitellään keinoja, joilla voidaan selkeyttää ohjelman toteutusta suurilla tietomäärillä. Tämän jälkeen käsitellään tilan päivittämistä ja sitä, kuinka muutokset näkyvät käyttöliittymässä. Tähän liittyen perehdytään myös tarvittaviin toimenpiteisiin, joilla tila ja käyttöliittymä saadaan pidettyä synkronoituina. Myöhemmin työssä tarkastellaan tilanhallinnan työkaluja, jotka ratkaisevat suurimman osan työssä esitellyistä ongelmista. Lopuksi yhteenvedossa esitellään työn keskeisimmät havainnot.

2 React

2.1 Reactin toimintaperiaate

React on avoimen lähdekoodin JavaScript-kirjasto verkkosovellusten käyttöliittymien rakentamiseen. Se julkaistiin vuonna 2013 ja on ollut suuressa suosiossa alusta asti. Yksi syy tähän on, että React käyttää deklaratiivista ohjelmointityyliä. Deklaratiivisella ohjelmoinnilla sovellus jäsennellään keskittymällä vain siihen, mitä halutaan saavuttaa, ottamatta kantaa siihen, miten halutut asiat saavutetaan. [2] Käytännössä tämä tarkoittaa sitä, että sama asia voidaan toteuttaa usealla eri tavalla. React ei siis aseta ohjelmoijalle mitään erityisiä vaatimuksia ohjelman toteuttamiseen.

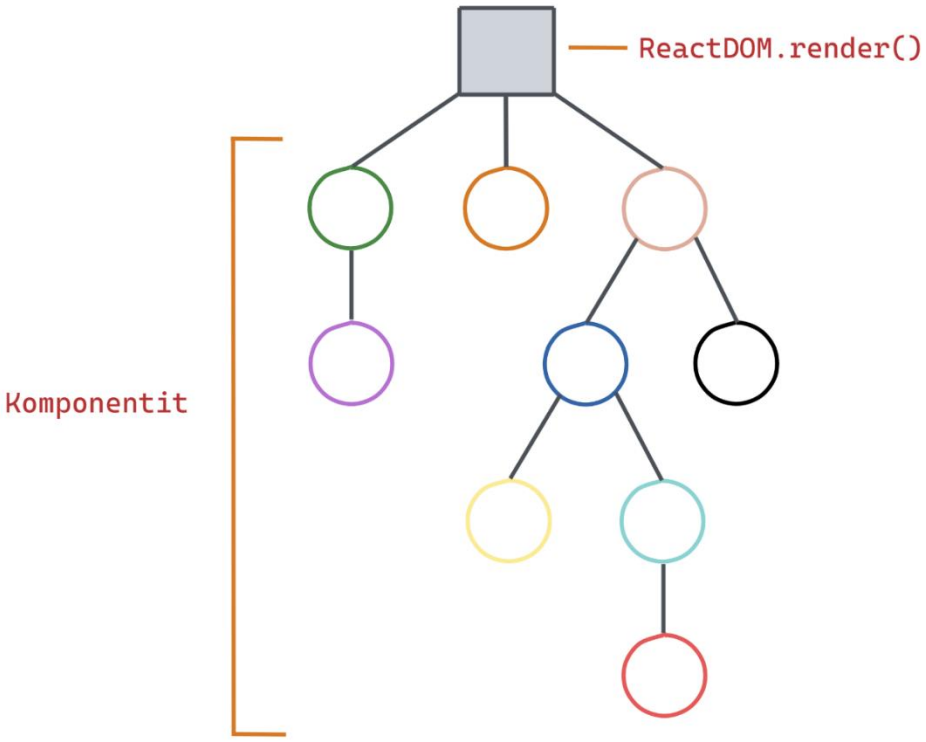
Reactia käytetään usein osana MV*-arkkitehtuuria, joka on yleinen ohjelmistojen suunnittelumalli. Kirjaimet M ja V tulevat englannin kielen sanoista Model ja View eli suomeksi malli ja näkymä. Tyypillisesti malli vastaa sovelluksen datan säilömisestä tietokantaan. Reactilla toteutettu osa toimii pääasiassa arkkitehtuurin näkymänä huolehtien sovelluksen visuaalisista elementeistä ja niiden pitämisestä ajan tasalla. Muut arkkitehtuurin osat ovat vapaasti valittavissa, mikä tekee Reactin käytöstä joustavaa. [3]

React soveltuu erityisesti yksisivuisten sovellusten (engl. Single-page application, lyhennettynä SPA) luomiseen. Yksisivuinen sovellus tarkoittaa sovellusta, jossa pysytään yhdellä www-sivulla. Tämän ansiosta selaimen ei tarvitse päivittää sivua aina uudelleen näkymän vaihtuessa, toisin kuin perinteisissä, monisivuisissa sovelluksissa. [4] DOM eli Document Object Model on selaimen tapa käsitteellistää dokumentin sisältöä. Se siis muuttaa verkkosivun koodin visuaalisiksi objekteiksi. [5] SPA-sovelluksessa sivun näkymä päivittyy dynaamisesti, mutta DOMiin tehtävät muutokset tapahtuvat erittäin hitaasti. React ratkaisee tämän ongelman käyttämällä omaa virtuaalista DOMia [3].

Virtuaalinen DOM ylläpitää jäljitelmää oikeasta DOMista ja tekee muutoksia siihen. Muutoksia vertaillaan oikeaan DOMiin ja kun tarvittavat muutokset on tehty, React pystyy päivittämään oikean DOMin yksinkertaisesti. Virtuaalisen DOMin käsitteleminen on paljon nopeampaa kuin oikean DOMin. [3] Se mahdollistaa nopeiden peräkkäisten muutosten päivittämisen yhdellä kertaa. Tällöin oikeaa DOMia ei tarvitse päivittää yhtä usein ja sovelluksen tehokkuus parantuu.

2.2 Ohjelman rakenne

React on komponenttipohjainen kirjasto. Sen perusideana on ohjelman jakaminen pieniin paloihin eli komponentteihin. Komponentit voivat olla toteutukseltaan luokkia tai funktioita. Ne koostuvat usein alikomponenteista, mikä luo ohjelmalle hierarkkisen rakenteen. Komponentti ja sen alikomponentti muodostavat vanhempi–lapsi-suhteen, jossa yläkomponentti on vanhempi ja alikomponentti on sen lapsi. [6] Ohjelman rakennetta on havainnollistettu kuvassa 1.



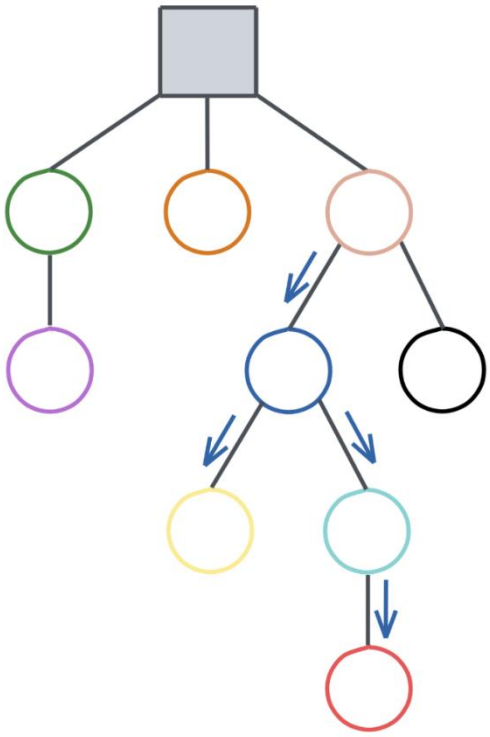
Kuva 1. React-ohjelman rakenne [3].

Ohjelma jakautuu komponentteihin yleensä toiminnallisuuksien perusteella. Ideaalisti jokaisella komponentilla tulisi olla vain yksi tehtävä tai vastuualue. Komponenttien tulisi myös olla toisistaan irrallisia ja riippumattomia ja täysin uudelleenkäytettäviä. [7] Tämän toteuttaminen on kuitenkin hankalaa, kun ohjelmassa on paljon komponenttien välillä liikkuvaa dataa.

3 Tiedon välittäminen

3.1 Komponenttien välinen kommunikointi

Tiedon välittäminen on olennainen osa ohjelman tilanhallintaa. Reactin tapauksessa tietoa voidaan välittää vain vanhemmalta tämän välittömälle lapselle. Jos tietoa halutaan välittää kaukaisemmalle alikomponentille, tiedon on kuljettava jokaisen reitillä olevan komponentin kautta. [3] Tällöin data valuu hierarkiassa alaspäin kuvan 2 mukaisesti.



Kuva 2. Datan valuminen. [3]

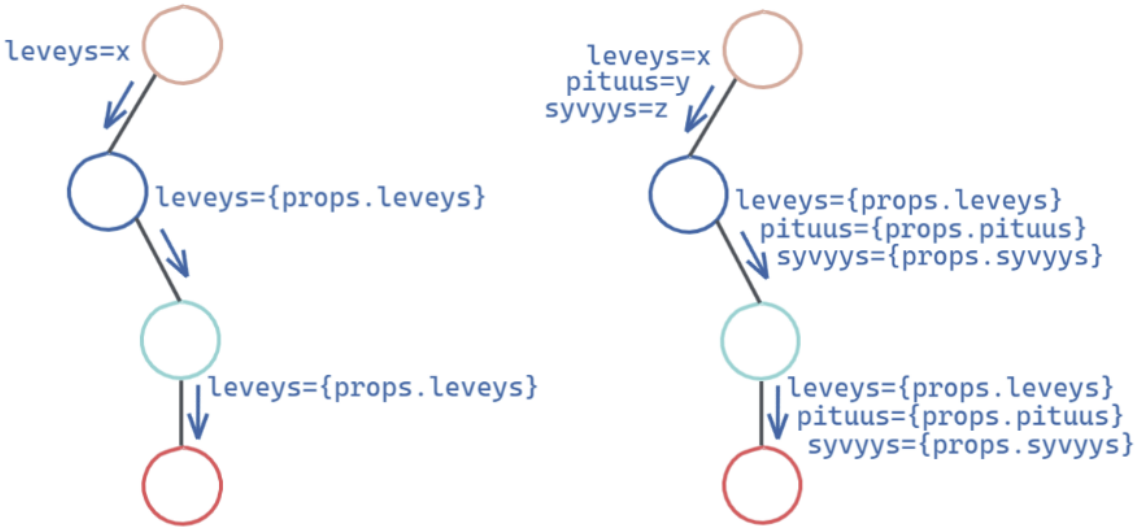
Reitillä olevat komponentit eivät välttämättä itse tarvitse tietoa, joka kulkee niiden läpi. Etenkin isommissa sovelluksissa datan lähteen ja sitä tarvitsevan komponentin välillä voi olla todella monta komponenttikerrosta. Jokaisella kerroksella komponentit vastaanottavat tietoa, jota ne vain välittävät edelleen eteenpäin. Tämä paisuttaa koodia ja tekee siitä vaikeasti skaalautuvaa. [2] Lisäksi voi olla vaikeaa hahmottaa, mikä vastaanotettavasta tiedosta on komponentille oikeasti relevanttia ja mikä vain välitetään eteenpäin.

Vanhempi voi myös välittää lapselleen takaisinkutsufunktion (engl. callback) eli funktion, joka on määritelty vanhemmassa ja annetaan kutsuttavaksi eteenpäin. Tällöin lapsi voi kutsua funktiota omilla argumenteillaan ja siten vaikuttaa yläpuo-

lella olevien komponenttien toimintaan. Tällainen funktio saattaa esimerkiksi päivittää yläkomponentissa olevaa tietoa, minkä jälkeen yläkomponentti välittää päivittyneen tiedon taas alaspäin. [1]

3.2 Propit

Tieto kulkee ohjelman sisällä attribuutteina eli proppeina (engl. property). Propit välitetään alikomponentille yhtenä props-oliona. [6] Mitä enemmän tietoa on välitettävänä, sitä enemmän tarvitaan proppeja. Otetaan esimerkiksi tilanne, jossa kuvan 2 vaaleanpunaisesta komponentista halutaan välittää tietoa punaiselle komponentille. Tiedon kulkua havainnollistetaan kuvassa 3.



Kuva 3. Proppien välittäminen [3].

Kun tietoa välitetään vain vähän, ohjelma pysyy yksinkertaisena. Ongelmaksi muodostuu kuitenkin proppien määrän nopea kasvu. Suuri määrä proppeja tekee koodista sotkuista ja vaikeasti ylläpidettävää. Kuvan 3 oikealla puolella proppeja on vain muutama enemmän kuin vasemmalla, mutta jo niinkin pieni määrä saa aikaan merkittävän vaikutuksen. Kuvan oikea puoli on huomattavasti vaikeaselkoisempi vasempaan verrattuna. Suurempien ohjelmien tapauksessa proppeja voi helposti olla useita kymmeniä, jolloin niitä on vaikea hallita ja seurata. Tällöin tarvitaan keinoja, joilla voidaan helpottaa proppien käyttöä.

Proppeja voidaan yhdistellä muodostamalla niistä aliolioita. Alioliot vähentävät proppien määrää ja ryhmittelevät propit ymmärrettäviksi kokonaisuuksiksi. Edellisen esimerkin propit voitaisiin siis yhdistää yhdeksi koko-olioksi, joka sisältäisi mitat avain-arvo-pareina. Propit eivät aina kuitenkaan liity samaan aihealueeseen, jolloin niiden ryhmittely ei välttämättä onnistu.

Toinen tapa siistiä proppien määrää on JavaScriptin spread-operaattori. Se hajauttaa olion alkiot yksittäisiksi elementeiksi tai pakkaa yksittäiset elementit takaisin yhdeksi olioksi [8]. Tällöin yläkomponentti voi välittää kaikki propit kerralla eteenpäin, eikä niitä tarvitse erikseen määritellä. Alikomponentti puolestaan voi poimia oliosta vain haluamansa propit ja pakata loput välitettäväksi eteenpäin.

```
1  function Komponentti(props) {
2    const { omaProp, ...loputPropit } = props;
3    if (omaProp) {
4      // tee jotain
5    }
6    return <AliKomponentti {...loputPropit} />;
7  }
```

Ohjelma 1. Spread-operaattorin käyttö.

Ohjelman 1 rivillä 2 props-oliosta poimitaan `omaProp` omaan muuttujaansa ja loput pakataan spread operaattorilla `loputPropit`-olioksi. Rivillä 6 pakatut propit puretaan eteenpäin alikomponentille. Spread-operaattori yksinkertaistaa tiedonvälitystä ja auttaa erottamaa komponentille relevantit propit niistä, jotka vain välitetään eteenpäin. Tällöin suurikin määrä proppeja voidaan välittää eteenpäin siististi.

3.3 Context

Context tarjoaa tavan välittää tietoa ilman proppeja. Se on suunniteltu erityisesti globaalin datan jakamiseen. Globaalia dataa voi olla esimerkiksi käytössä oleva teema, tämänhetkinen sisään kirjautunut käyttäjä tai valittu kieli. [9] Reactiin sisältyy `createContext()`-funktio, joka luo uuden `context`-olion. `Context`-olio sisältää `Context.Provider` -komponentin, joka välittää datan komponenteille. [2] Ohjelmassa 2 on toteutettu esimerkki teeman jakamisesta contextin avulla.

```
1  const TeemaContext = React.createContext('vaalea');
2
3  function Sovellus() {
4    const valittuTeema = 'tumma';
5    return <div>
6      <TeemaContext.Provider value={valittuTeema}>
7        <Komponentti />
8      </TeemaContext.Provider>
9    </div>;
10 }
11
12 function Komponentti() {
13   return <Alikomponentti />;
14 }
15
16 function Alikomponentti() {
17   const teema = useContext(TeemaContext);
18   return <button style={{ backgroundcolor: teema.value}} />;
19 }
```

Ohjelma 2. Context.

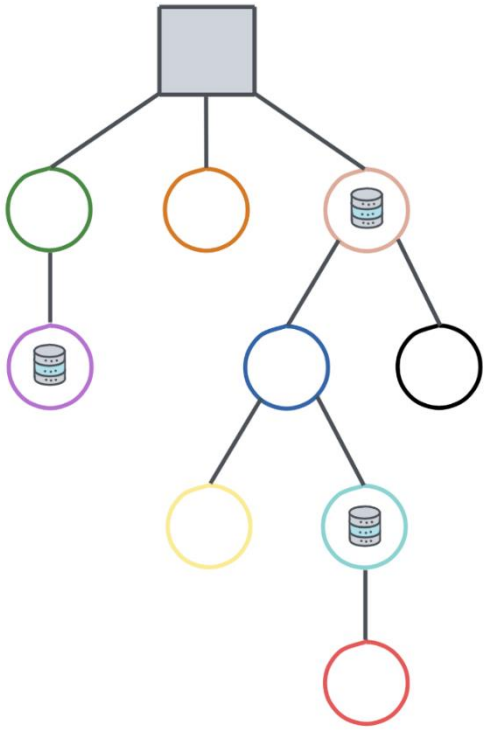
Ohjelman 2 ensimmäisellä rivillä luodaan context-olio sovelluksen teeman ylläpitoa varten. Sille annetaan argumenttina oletusarvo, joka on tässä esimerkissä vaalea. Rivillä 6 luodaan `TeemaContext.Provider`-komponentti, jolle annetaan proppina contextin nykyinen arvo eli tällä hetkellä valittu teema. Mikäli tätä arvoa ei annettaisi, käytettäisiin teeman oletusarvoa. Provider-komponentti välittää teeman alikomponenteilleen, jotka voivat lukea teeman arvon huolimatta siitä, kuinka syvällä komponenttipuussa ne ovat. Komponentin ei siis tarvitse välittää arvoa eteenpäin Alikomponentille, vaan Alikomponentti voi suoraan lukea teeman arvon `useContext()`-funktiolla, kuten rivillä 17.

Context tarjoaa siis tavan vähentää proppien määrää, jolloin koodia on helpompi seurata. Alikomponenttia ei kuitenkaan voida uudelleen käyttää sellaisenaan Sovellus-komponentin ulkopuolella, koska se vaatii `TeemaContext.Provider`-komponentin yläkomponentikseen. Contextin käyttö tekee siis komponenttien uudelleenkäyttämisestä hankalampaa [9]. Tämä voi olla ongelmallista suuremmissa ohjelmissa, joissa komponenttien uudelleenkäyttö on tärkeää ohjelman ylläpidettävyyden kannalta. Pienissä ohjelmissa tällä ei ole yhtä suurta merkitystä, sillä komponentteja on vain vähän ja ohjelmaa on helpompi hallita. Globaalia dataa voidaan kuitenkin käsitellä muullakin tavalla, mihin perehdytään luvussa 5.

4 Tilan hallinta

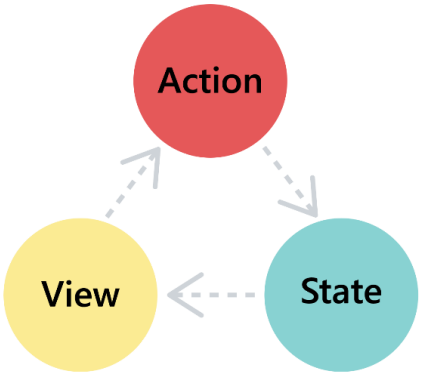
4.1 Komponenttien sisäinen tila

Propit välittävät dataa komponenteille, mutta komponenttien ei tulisi muokata vastaanottamaansa dataa [1]. Proppien lisäksi tarvitaan siis tapa, jolla voidaan säilöä muokattavaa tietoa. Tila (engl. state) mahdollistaa tämän. Tila säilötään komponenttien sisällä, eri puolilla ohjelmaa, mutta kaikki komponentit eivät tarvitse omaa tilaa. Tästä syystä komponentit voidaan luokitella joko tilallisiksi tai tilattomiksi komponenteiksi. [10] Kuva 4 havainnollistaa tätä.



Kuva 4. Tilalliset ja tilattomat komponentit. [3]

Komponentin sisäinen tila on lokaali, eivätkä muut komponentit pääse siihen käsiksi, mutta sen voi kuitenkin välittää lapsikomponentille proppeina. [10] Lapsikomponentti voi myös aiheuttaa yläkomponentin tilan muutoksen. Tämä tapahtuu antamalla lapsikomponentille proppina takaisinkutsufunktio, joka muuttaa tilaa. [2]

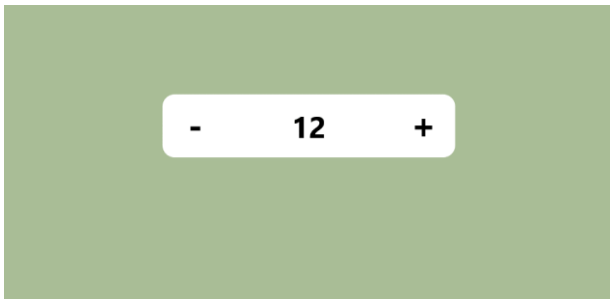


Kuva 5. Tilan päivittäminen. [11]

Tila kuvaa siis sovelluksen asemaa tietyllä ajanhetkellä. Näkymä (engl. View) on deklaratiivinen kuvaus käyttöliittymästä ja se esitetään eli *renderöidään* aina tilan perusteella. Käyttäjä luo tapahtumia näkymässä, joiden perusteella tilaa päivitetään. Kun tila päivittyy, näkymä renderöidään jälleen uuden tilan perusteella. [11] Kuva 5 havainnollistaa tätä sykliä.

4.1.1 Luokallisten komponenttien tila

Ennen Reactin versiota 16.8.0, ainoa tapa luoda tilallisia komponentteja oli käyttää luokkia [2]. Luokkakomponenttien syntaksi on hieman monimutkaisempi kuin funktionaalisten komponenttien. Syitä tähän ovat esimerkiksi luokkakomponenteissa käytettävä `constructor()`-metodi, jossa luokkainstanssi alustetaan ja `this`-avainsana, jolla voidaan käsitellä kyseistä instanssia.



Kuva 6. Laskuri.

Otetaan esimerkiksi kuvan 6 mukainen yksinkertainen laskuri. Laskurin plus- ja miinusnappuloilla voidaan lisätä ja vähentää keskellä näkyvää arvoa. Laskuri on toteutettu luokkakomponentilla ohjelmassa 3.

```
1  import React from 'react';
2
3  export class Counter {
4    constructor() {
5      this.state = { count: 0 };
6    }
7
8    changeCount = (amount) => {
9      this.setState((state) => ({
10       count: state.count + amount,
11     }));
12   }
13
14   render() {
15     return <div>
16       <button onClick={() => this.changeCount(-1)}>-</button>
17       <span>{this.state.count}</span>
18       <button onClick={() => this.changeCount(1)}>+</button>
19     </div>;
20   }
21 }
```

Ohjelma 3. Laskurin toteutus luokkakomponentilla.

Ohjelman 3 luokkakomponentin rakentajassa, rivillä 5, tila tallennetaan olioon ja tilamuuttujan `count` arvo alustetaan nollaksi. Tilaa voidaan muuttaa `changeCount()`-metodilla, joka ottaa argumentiksi arvon, joka lisätään `count`-muuttujan arvoon. Tilan muuttaminen tapahtuu rivillä 9 `this.setState()`-kutsulla. Arvon muutos lisätään aiempaan tilaan, jonka `setState()` ottaa argumenttina. Riveillä 16 ja 18 `Button`-alikomponenteille annetaan proppina `onClick`, joka on napin painalluksen tapahtumakäsittelijä. Tässä tapauksessa se kutsuu `changeCount()`-metodia annetulla arvolla aina, kun nappia painetaan.

Tässä esimerkissä tilaan säilötään vain yksi avain-arvo-pari. Todellisuudessa monet komponentit tarvitsevat useampia avain-arvo-pareja. Jos pareja on enemmän, ne tallennetaan kaikki `this.state`-olioon. Pareja voidaan myös päivittää yhtä aikaa, yhdellä `this.setState()`-kutsulla.

4.1.2 Funktionaalisten komponenttien tila

Funktionaaliset komponentit hyödyntävät Reactin tarjoamia koukkuja eli *hookkeja*. Niiden avulla funktionaalisiin komponentteihin voidaan sisällyttää Reactin toiminnallisuutta, kuten tilan käyttämistä. [12] Ohjelmassa 4 tarkastellaan vielä samaa laskuriesimerkkiä funktionaalisilla komponenteilla.

```
1  import React, { useState } from 'react';
2
3  export function Counter() {
4    const [count, setCount] = useState(0);
5
6    const changeCount = (amount) => {
7      setCount(count + amount);
8    }
9
10   return <div>
11     <button onClick={() => changeCount(-1)}>-</button>
12     <span>{count}</span>
13     <button onClick={() => changeCount(1)}>+</button>
14   </div>;
15 }
```

Ohjelma 4. Laskurin toteutus funktionaalisella komponentilla.

Funktionaalisten komponenttien tapauksessa tila ei löydy yhdestä oliosta, vaan se tallennetaan erillisiin muuttujiin `useState()`-hookin avulla, kuten ohjelman 4 rivillä 4. Alustettava arvo annetaan argumenttina. Muuttujan lisäksi `useState()` palauttaa funktion, jolla tilan arvoa voidaan vaihtaa. Tällaista funktiota kutsutaan setteriksi. Jos tilaan säilöttäisiin useampia arvoja, ne päivitettäisiin yksi kerrallaan, kukin omilla setter-funktioillaan.

Ohjelmasta 4 huomataan, että `Counter`-komponentti saatiin toteutettua huomattavasti pienemmällä rivimäärällä ja paljon suoraviivaisemmin kuin luokkakomponentin tapauksessa. Vertailun vuoksi `changeCount()` on jätetty omaksi funktiokseen, mutta koska sen toteutus on niin lyhyt, olisi mahdollista kutsua `setCount()`-setteriä suoraan `onClick`-käsittelijässä. Tämä yksinkertaistaisi komponentin toteutusta entisestään.

4.2 Tilan päivittäminen

Reactissa tilan päivittäminen vaatii erityisiä toimenpiteitä, sillä komponenttien tila päivittyy asynkronisesti. Tämä tarkoittaa sitä, että tilan päivityskäskyn jälkeen ohjelma jatkaa suorittamista eikä jää odottamaan, että tila on päivittynyt. Tilaan tehdyt muutokset eivät siis välttämättä tule heti näkyviin, jolloin tämänhetkiseen tilaan ei saisi viitata uutta tilaa laskiessa. [13] Tästä syystä esimerkiksi `setState()` ottaa argumenttina aiemman tilan.

4.2.1 Renderöityminen

Komponentin tilamuuttujille olisi hyvä antaa jokin alustava arvo, vaikka sitä ei vaaditakaan. Tämä johtuu siitä, että komponentti olettaa tilan muuttujilla olevan jotkin arvot. Jos odotettua arvoa ei ole annettu, komponentti saattaa renderöidä jotakin odottamatonta tai renderöityminen voi epäonnistua kokonaan. [12]

Aina kun komponentin tila muuttuu tai sille annettava proppi muuttuu, kaikki vaikutuksessa olevat komponentit renderöidään uudestaan. Kun komponentti renderöidään, myös sen lapsikomponentit renderöidään rekursiivisesti. Tämä johtaa usein komponenttien turhaan renderöitymiseen. Lapsikomponentteja renderöidään uudestaan, vaikka tapahtuneet muutokset eivät koskettaisi niitä mitenkään. Ylimääräinen renderöinti vie suoritustehoa turhaan. [3] Tämä ongelma voidaan ratkaista esimerkiksi riippuvuuslistojen avulla, joihin perehdytään luvussa 4.2.2.

4.2.2 Riippuvuudet

Komponentit saattavat riippua toisista komponenteista tai niiden tilamuuttujista. Kun jotain komponenttia muokataan, täytyy yleensä muokata myös niitä komponentteja, jotka riippuvat kyseisestä komponentista [14]. Tämä tapahtuu rekursiivisesti aiheuttaen vaikeasti seurattavia tapahtumaketjuja. Useat tällaiset riippuvuudet hankaloittavat komponenttien uudelleenkäyttöä ja koodin ylläpidettävyyttä. Lisäksi tilaa säilötään yleensä eri puolilla ohjelmaa, mikä hankaloittaa tapahtumaketjujen seurattavuutta entisestään.

Funktionaalisissa komponenteissa riippuvuudet voidaan määritellä riippuvuuslistaan. Listan avulla komponentti kuuntelee vain tiettyjen arvojen muutoksia ja päivittyy vain silloin, kun nämä arvot muuttuvat. Tämä vähentävää turhaa uudelleenrenderöitymistä.

```
1  function Komponentti({ jokuProppi, toinenProppi }) {
2    return <div>
3      <p>{jokuProppi}</p>
4      <Alikomponentti proppi={toinenProppi} />
5    </div>;
6  }
```

Ohjelma 5. Riippuvuus esimerkki.

Jos ohjelman 5 Alikomponentti ei käytä riippuvuuslistaa, se päivittyy aina kun Komponentti päivittyy. Tällöin Alikomponentti päivittyy silloinkin, kun jokuProppi päivittyy, vaikka jokuProppi ei vaikuta Alikomponenttiin. Jos Alikomponentti on

määritellyt riippuvuuslistaan proppi-arvon, niin se päivittyy vain silloin, kun toinenProppi päivittyy eikä turhaan silloin, kun jokuProppi päivittyy.

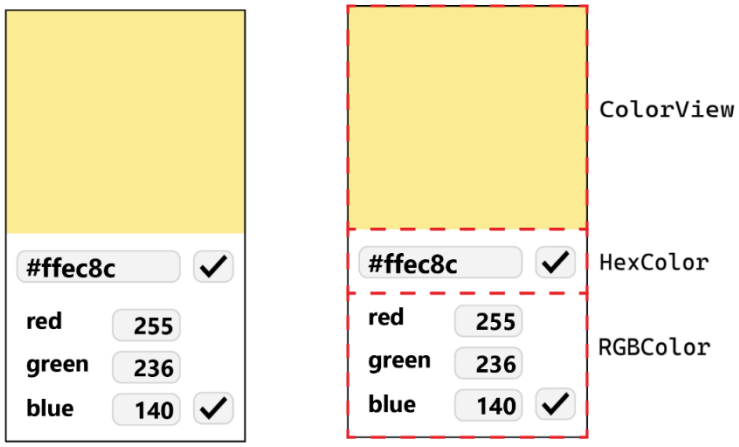
Riippuvuuslista annetaan tyypillisesti hookeille argumenttina. Yleisin tällaisista hookeista on useEffect(), jonka avulla määritellään suoritettava toiminto riippuvuusmuuttujan arvon muuttuessa. Vaikka riippuvuuslistoja käyttävät hookit ratkaisevatkin turhaan renderöitymiseen liittyviä ongelmia teknisellä tasolla, niiden käyttö muuttuu erittäin hankalaksi ohjelman koon kasvaessa. Riippuvuuksien määrä kasvaa nopeasti ja riippuvuuslistat voivat koostua pahimmillaan jopa kymmenistä riippuvuusmuuttujista. Tätä ongelmaa ei ilmene pienissä sovelluksissa, joissa riippuvuuksia on vain vähän ja riippuvuuslistat pysyvät hallittavina.

Luokkakomponenteilla ei ole riippuvuuslistoja. Niissä reaktiot tapahtumiin määritellään erityisien elinkaarimetodien avulla. Tällaisia tapahtumia ovat esimerkiksi komponentin kiinnittyminen DOMiin, tilan tai proppien muuttuminen, ja komponentin irrottautuminen DOMista. Jokaiselle näistä on oma metodi. [10] Luokkakomponenteissa samaan tilaan liittyvät eri tapahtumat joudutaan siis hajauttamaan eri elinkaarimetodeihin. Funktionaalisten komponenttien tapauksessa yhdessä hookissa voidaan määritellä reaktio useampaan samaan tilaan liittyvään tapahtumaan, jolloin komponentin toteutus pysyy ehyempänä.

4.2.3 Synkronointi

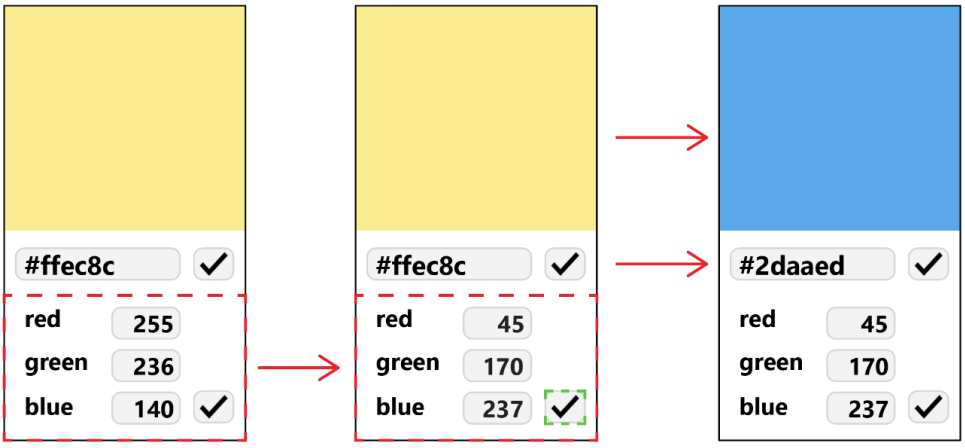
Monet komponentit voivat tarvita viittausta samaan tilaan. Tällainen jaettu tila tulisi olla tallennettu komponenttien lähimmälle esivanhemmalle. [15] Jos jaettua tilaa muokataan mistään komponentista, kaikki tilaan viittaavat alikomponentit saavat päivittyneen tiedon automaattisesti. Ongelma ilmenee, kun jokin alikomponenteista tarvitsee oman kopion tilasta voidakseen poiketa siitä väliaikaisesti. Tällöin tilapäisen tilan ja jaetun tilan synkronointi vaatii lisätoimenpiteitä, jotka monimutkaistavat ohjelman toteutusta ja hankaloittavat sen rakenteen hahmottamista.

Otetaan esimerkiksi kuvan 7 mukainen, yksinkertainen värinvalintaohjelma. Käyttäjä voi antaa ohjelmalle haluamansa värin joko heksadesimaali- tai RGB-muodossa. Ohjelma on jaettu kolmeksi komponentiksi, jotka on eroteltu punaisella katkoviivalla.



Kuva 7. Värinvalintaohjelma ja siihen liittyvät komponentit.

Ohjelman toimintaa on havainnollistettu kuvassa 8. Kun käyttäjä vaihtaa väriä, hän syöttää ensin haluamansa RGB-arvot ja painaa sitten ok-nappia. Napin painallus aktivoi muutokset vaihtamalla neliön värin ja päivittämällä heksadesimaaliarvon. Väriä ei siis haluta päivittää samalla, kun käyttäjä syöttää arvoja, vaan vasta sitten, kun käyttäjä hyväksyy tapahtuman painamalla nappia.



Kuva 8. Värinvalintaohjelman toiminta.

Ohjelman rakenne monimutkaistuu entisestään, kun arvojen synkronoinnissa on otettava huomioon molempien komponenttien tilapäiset värien arvot. Tästä esimerkkinä voisi olla tilanne, jossa väri päivitetään RGB-arvolla, kun heksadesimaalin arvoa on muutettu välissä painamatta nappia. Tarkastellaan ohjelman toimintaa vielä koodin tasolla. Ohjelmasta on toteutettu vain toiminnallisuus, joten se ei vastaa ulkonäöllisesti kuvien 7 ja 8 käyttöliittymää. Ohjelman toteutuksessa on käytetty funktionaalisia komponentteja yksinkertaisuuden vuoksi.

```
1  import React, { useState, useEffect } from 'react';
2
3  export function ColorPicker() {
4    const [color, setColor] = useState('#ffffff');
5
6    return <div>
7      <ColorView color={color} />
8      <HexColor
9        color={color}
10       changeColor={hexColor => setColor(hexColor)}
11     />
12     <RGBColor
13       color={hexToRgb(color)}
14       changeColor={(r, g, b) => setColor(rbgToHex(r, g, b))}
15     />
16   </div>;
17 }
```

Ohjelma 6. Komponentin ColorPicker toteutus.

Ohjelman 6 rivillä 4, `ColorPicker`-komponentille määritellään tilamuuttuja, joka kuvaa tämänhetkistä väriä. Se alustetaan valkoiseksi. Riveillä 9 ja 13 tilamuuttuja välitetään alikomponenteille proppeina. Lisäksi riveillä 10 ja 14 alikomponenteille annetaan proppina takaisinkutsufunktio, joka suoritetaan, kun käyttäjä painaa ok-nappia. Tämä funktio muuttaa tilan arvon käyttäjän asettamaksi väriksi. Riveillä 13 ja 14 käytetään apufunktioita, jotka muuttavat heksadesimaalimuodossa olevan värin RGB-muotoon ja toisinpäin. Niiden toiminta ei ole tämän esimerkin kannalta olennaista.

```
1  function ColorView(props) {
2    return <div style={{ backgroundColor: props.color }} />;
3  }
```

Ohjelma 7. Komponentin ColorView toteutus.

`ColorView`-komponentti vastaa ainoastaan värin näyttämisestä. Ohjelman 7 rivillä 2 asetetaan palautettavan elementin taustavärin, `style`-argumentilla, proppina saaduksi arvoksi. Propin arvon muuttuessa taustaväri päivittyy automaattisesti.

```
1  function HexColor(props) {
2    const [hexColor, setHexColor] = useState('#ffffff');
3
4    useEffect(() => {
5      setHexColor(props.color);
6    }, [props.color]);
7
8    return <div>
9      <input
10       value={hexColor}
11       onChange={e => setHexColor(e.target.value)}
12     />
13     <button onClick={() => props.changeColor(hexColor)} />
14   </div>;
15 }
```

Ohjelma 8. Komponentin HexColor toteutus.

Ohjelman 8 rivillä 2, `HexColor`-komponentti tallentaa värin omaan tilaansa. Tämä johtuu siitä, että `ColorPicker`-komponentin tilassa olevasta väristä on poiketta va väliaikaisesti, kunnes käyttäjä painaa nappia. Jotta `hexColor`-muuttujan arvo pysyy synkronoituna proppina saatavan arvon kanssa, rivillä 4 käytetään `useEffect()`-hookkia. Hookille annetaan argumentteina funktio, joka halutaan suorittaa aina muutoksen tapahtuessa, ja riippuvuuslista, joka tässä tapauksessa sisältää vain propin. Tällöin aina propin arvon muuttuessa `hexColor`-muuttujan arvo vaihdetaan propin arvoksi. RGBColor-komponentti on toiminnaltaan täysin identtinen HexColor-komponentin kanssa, joten sen toteutus on jätetty tästä luvusta pois. Toteutuksen koodi löytyy kuitenkin liitteestä 1.

Koska värinvalintaohjelma on pieni ja yksinkertainen, ongelma oli suhteellisen helppo ratkaista teknisesti. Yksi useEffect()-kutsu ei myöskään hankaloittanut ohjelman ymmärrettävyyttä liikaa. Suuremmissa ohjelmissa komponenttien toteutus täyttyy kuitenkin nopeasti näistä kutsuista ja pitkistä riippuvuuslistoista. Usein useEffect() saattaa aiheuttaa rekursiivisesti tapahtumia myös komponentin ulkopuolella, jolloin tapahtumien kulkua on vaikea hahmottaa ja virheiden löytyminen hankaloituu. Tällaisten tapahtumaketjujen lisääntyessä koodin ylläpidettävyys muuttuu lähes mahdottomaksi ja muutosten tekeminen koodiin saattaa aiheuttaa yllättäviä seuraamuksia muualla ohjelmassa. Ratkaisuna tähän toimii globaali tietovarasto, jossa sovelluksen tila säilötään. Tähän perehdytään tarkemmin luvussa 5.

5 Tilanhallinnan työkalut

Reactin tilanhallintaa varten löytyy useita kolmannen osapuolen tarjoamia työkaluja, jotka ratkaisevat ainakin suurimman osan Reactin ongelmista. Tässä työssä esitellään lyhyesti MobX-kirjasto ja GraphQL. Näiden lisäksi Redux esitellään hieman perusteellisemmin.

5.1 MobX

MobX on tilanhallinnan kirjasto, joka mahdollistaa side effect modelin eli sivuvaikutusmallin. Sivuvaikutuksella tarkoitetaan reaktiota, jonka halutaan tapahtuvan tilan muuttuessa eli esimerkiksi renderöityminen tai datan päivittäminen palvelimelle. MobX tarjoaa ohjelmalle ydinrakenteen, jossa sovelluksen tila on käyttöliittymässä tapahtuvien asioiden keskipisteessä. Tätä kutsutaan observoitavaksi tilaksi. [16] Tila tulisi pitää mahdollisimman minimalistisena säilömällä sinne vain välttämättömät arvot ja laskemalla muut halutut arvot niiden pohjalta [17].

Havainnoitavaa tilaa muokataan toiminnoilla. Kun tila päivittyy, MobX huolehtii siitä, että kaikki tilaan liittyvät lasketut arvot päivittyvät ja sivuvaikutukset tapahtuvat automaattisesti. [17] Tällöin asynkronisuudesta ei tarvitse huolehtia ja käyttöliittymä on helppo pitää synkronoituna tilan kanssa.

5.2 GraphQL

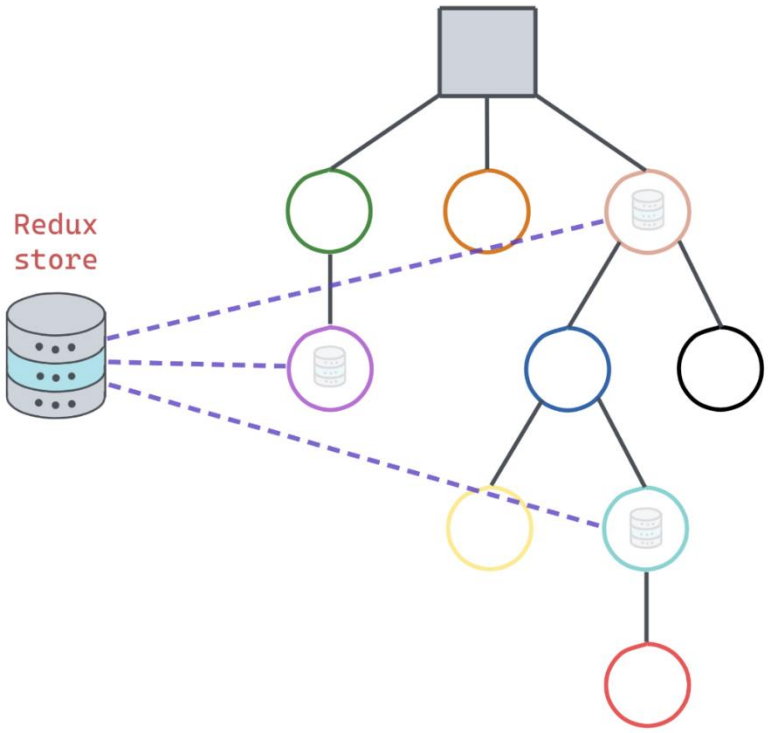
GraphQL on kieli, jolla voidaan deklaratiivisesti kommunikoida tietovaatimuksista palvelimelle. Tämän lisäksi GraphQL on suoritustaso, jota palvelin käyttää ymmärtääkseen sille tehtyjä pyyntöjä ja vastatakseen niihin. Suoritustaso toimii käytännössä GraphQL-kielen kääntäjänä. [18] GraphQL:ää käytetään yleensä clienttien kanssa. Näistä suosituimpia ovat Relay ja Apollo Client. [14]

Relay on tarkoitettu erityisesti suurien, datakeskeisten React-sovellusten tilanhallintaan. Sen avulla komponentit voivat lokaalisti spesifioida, mitä ulkoista dataa ne tarvitsevat huolehtimatta siitä, miten dataa haetaan. Tällöin voidaan nopeasti muokata komponenttien datatarpeita eikä muihin ohjelman osiin tarvitse tehdä muutoksia. Relay pitää myös komponentit automaattisesti ajan tasalla datan muuttuessa. [19] Relay tekee suurimman osan ohjelman suunnittelupäätöksistä ohjelmoijan puolesta asettamalla tiettyjä vaatimuksia sovelluksen rakenteelle. [18]

Apollo Client ajaa pääpiirteittäin saman asian, mutta on vapaamuotoisempi kuin Relay. Se ei ota kantaa ohjelman rakenteeseen, eikä rajoitu pelkästään React-sovelluksiin. Vapaamuotoisuuden takia Apollon käyttö vaatii kuitenkin enemmän manuaalista työtä kuin Relay. [20]

5.3 Redux

Redux on kirjasto, joka on luotu Reactin tilanhallintaan. Sen tavoitteena on lisätä sovellusten datan johdonmukaisuutta ja hallita dataa ennalta-arvattavasti. [14] Redux tarjoaa varaston sovelluksen globaalille tilalle. Tämä varasto on Redux Store. Varastoon pääsee käsiksi kaikkialta sovelluksesta kuvan 9 mukaisesti, mutta sitä voidaan päivittää vain tiettyjen sääntöjen mukaisesti. [11]



Kuva 9. Redux store. [3]

Reduxilla on kolme pääperiaatetta:

1. Tila säilötään yhdessä oliossa yhden varaston sisällä. Tämä helpottaa sovelluksen kokonaisuuden hahmottamista, toimintojen vaikutusten ennustamista ja virheiden löytämistä.
2. Tila voidaan vain lukea. Tällöin tilaan tehtävät muutokset eivät suoraan muuta tilassa olevaa dataa, vaan tilasta luodaan täysin uusi instanssi.
3. Tilaan voidaan tehdä muutoksia ainoastaan puhtailla funktioilla. Puhtaat funktiot tuottavat aina saman lopputuloksen samoilla annetuilla arvoilla, eivätkä muuta dataa prosessissa.

Näiden periaatteiden seuraaminen tekee tilan hallinnasta ennustettavaa. [14]

Redux mahdollistaa datan jakamisen ja päivittämisen suoraan sitä tarvitseville komponenteille. Tällöin komponenttien välillä liikkuvan tiedon määrä vähentyy.

Lisäksi ei tarvitse huolehtia siitä, kuinka data ja siihen tehtävät muutokset päätyvät oikeaan paikkaan aiheuttamatta turhaa renderöimistä. [3] Redux vaatii kuitenkin suhteellisen paljon koodia ja monimutkaistaa ohjelmaa huomattavasti lokaaliin tilaan nähden. Tästä syystä se ei sovellu pienen mittakaavan ohjelmille. [14]

5.3.1 Tilan ylläpito

Varasto ylläpitää tilaa yhdessä oliossa, jota kutsutaan tilapuuksi. Tilan säilömisen lisäksi sen tehtäviin kuuluu tarjota tapa päästä tilaa käsiksi tilaan ja määrittää tilan päivitykset. [14] Varastolla on `getState()`-metodi, joka palauttaa tilan tämänhetkisen arvon. Tämän lisäksi sille määritetään reducer, joka kertoo miten ja milloin tilaa päivitetään eli se sisältää tilan päivittämislogiikan. [11] Ohjelmassa 9 toteutetaan mahdollinen tilapuun konfigurointi luvussa 4 esiintyneelle laskurille. Reducerin toteutukseen perehdytään myöhemmin.

```
1  import { configureStore } from '@reduxjs/toolkit';
2
3  const store = configureStore({ reducer: counterReducer });
4  const state = store.getState();
```

Ohjelma 9. Tilapuun konfigurointi.

Komponentit voivat poimia vain tarvitsemansa tilan arvot valitsimien avulla. Valitsin (engl. Selector) on puhdas funktio, joka ottaa argumenttina tilapuun ja palauttaa halutun tilamuuttujan. Ilman valitsimia komponentit joutuisivat parsimaan haluamansa tilamuuttujan arvon manuaalisesti ja olisivat siten riippuvaisia tilapuun rakenteesta. Valitsimien ansiosta tilapuun rakennetta voidaan muuttaa joustavasti, ilman että komponentteihin tarvitsee tehdä muutoksia. [14]

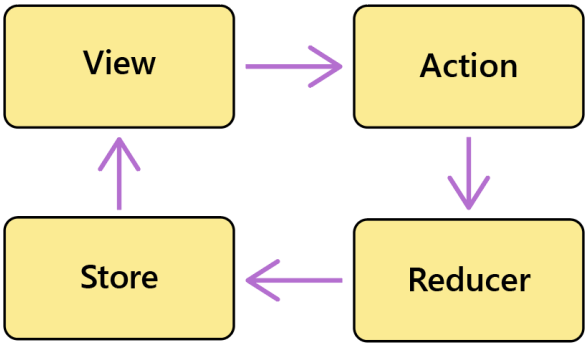
```
1  const count = useSelector(state => state.count);
```

Ohjelma 10. Valitsimen käyttö.

Ohjelmassa 10 on esitetty, miten tilapuusta saadaan count-muuttujan arvo. Kun tilamuuttujan arvo muuttuu varastossa, paikallisen muuttujan arvo päivitetään automaattisesti. Mikäli tilapuusta halutaan hakea useita muuttujia, jokainen niistä tulee hakea omalla valitsimellaan.

5.3.2 Tilan päivittäminen

Kuva 10 havainnollistaa, miten tilaa päivitetään Reduxissa. Yleensä käyttäjän toimesta näkymässä syntyy tarve muuttaa jotakin tilan arvoa [14]. Tätä tarvetta kuvataan toiminnoilla (engl. Action). Haluttu muutos toteutetaan tilan päivittämislogiikan mukaisesti reducerissa ja varasto päivittyy. Näkymä saa automaattisesti tietoonsa päivittyneen tilan.



Kuva 10. Tilan päivittäminen Reduxissa. [14]

Toiminnot kuvaavat siis, mitä halutaan muuttaa [3]. Toiminto on JavaScript-olio, jonka `type`-kenttä kertoo, mihin kategoriaan toiminto kuuluu ja mikä toiminto on kyseessä. [11] Toiminnot tuotetaan näkymässä tapahtuvien interaktioiden perusteella [14]. Niille on usein myös annettava lisätietoa siitä, mitä sovelluksessa on tapahtunut. Lisätiedot annetaan tyypillisesti `payload`-kentässä. [11]

```
1  const increaseCountAction = {
2    type: 'counter/increase',
3    payload: 1
4  }
```

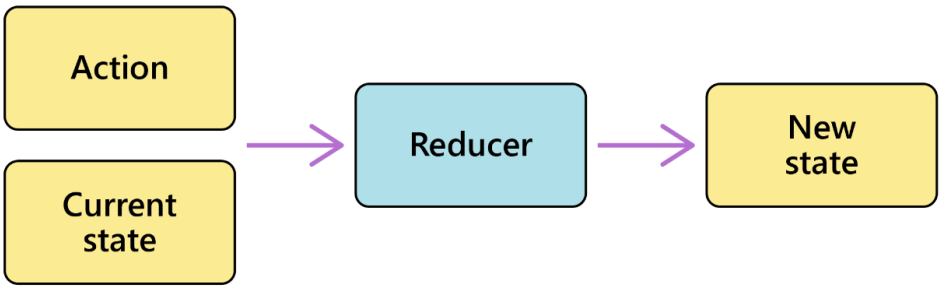
Ohjelma 11. Toiminto.

Toimintoja voidaan luoda manuaalisesti yksi kerrallaan, kuten ohjelmassa 11, tai sitten voidaan käyttää toimintojen luojia (engl. Action creator). Toimintojen luojat ovat funktioita, jotka ottavat lisätiedot argumenttina ja palauttavat toiminnon. [11] Tällöin jokaista toimintoa ei tarvitse määritellä erikseen. Toiminnon luoja on toteutettu ohjelmassa 12.

```
const increaseCount = amount => {
  return {
    type: 'counter/increase',
    payload: amount
  }
}
```

Ohjelma 12. Toiminnon luoja.

Reducer on puhdas funktio, joka kuvaa, miten tilaa muutetaan annetun toiminnon perusteella [3]. Se toimii siis tapahtumien kuuntelijana. Tapahtuman kuullessaan se ottaa tapahtumasta tuotetun toiminnon ja nykyisen tilan argumentteina, tekee tarvittavat toiminnot ja palauttaa uuden tilan. [11] Kuva 11 havainnollistaa reducerin toimintaa.



Kuva 11. Reducerin toiminta. [14]

Koska reducer on puhdas funktio, se ei saa muokata nykyisen tilan arvoa suoraan. Tästä syystä reducer kopioi nykyisen tilan, tekee tarvittavat muutokset kopioon ja palauttaa sen. [11] Ohjelmassa 13 on toteutettu laskurin reducer.

```
function counterReducer(state, action) {
  if (action.type === 'counter/increase') {
    const newState = state.copy();
    newState.count += action.payload;
    return newState;
  }
  return state;
}
```

Ohjelma 13. Reducer.

Varastolla on myös dispatch-metodi, jonka avulla tila voidaan päivittää varastossa. Sille annetaan argumenttina haluttu toiminto, jolla se suorittaa varaston reducer-funktion. [11] Kun reducer on prosessoinut actionin ja palauttanut uuden

tilan, varasto päivittää itsensä ja lähettää uuden tilan päivityksiä kuunteleville näkymille [14].

Koska tilan päivittämislogiikka on koottu reducer-funktioon, sitä voidaan muokata ilman, että tarvitsee muuttaa muita ohjelman osia. Tällöin ohjelman ylläpito ja uusien toiminnallisuuksien lisääminen helpottuu, mikä tekee ohjelmasta skaalautuvamman. Tämän lisäksi reducer-funktioiden puhtaus edesauttaa virheiden löytämistä, sillä koodia on helpompi seurata.

6 Yhteenveto

Tässä tutkielmassa käsiteltiin React-sovellusten tilanhallinnassa ilmeneviä ongelmia. Ongelmia havaittiin ohjelman sisäisessä tiedonvälityksessä sekä tilan varastoinnissa ja päivittämisessä. Työn tavoitteena oli selvittää, miten näitä ongelmia voidaan lieventää ja ratkaista.

Tiedonvälityksessä ongelmaksi muodostui välitettävän tiedon määrä. Suuri määrä liikutettavaa tietoa tekee koodista vaikeaselkoista. Merkittävimmäksi ratkaisuksi havaittiin JavaScriptin spread-operaattori, jonka avulla tietoa voidaan välittää yksinkertaisemmassa muodossa, ja Reactin context-työkalu, jonka avulla voidaan vähentää välitettävän tiedon määrää.

Toiseksi merkittäväksi ongelmaksi havaittiin alikomponenttien turha uudelleenrenderöiminen yläkomponentin tilan päivittyessä. Tämä voidaan ratkaista määrittelemällä komponentille riippuvuuslista eli lista muuttujista, joiden arvojen muutoksiin halutaan reagoida. Kyseinen ratkaisu toimii, kun ohjelman koko pysyy pienenä, mutta riippuvuuksien määrän kasvaessa ohjelman ylläpito hankaloituu. Kun tilaa varastoidaan hajautetusti eri puolilla ohjelmaa, riippuvuudet aiheuttavat vaikeasti seurattavia ja virhealttiita tapahtumaketjuja.

Työssä tutkittiin lopuksi kolmannen osapuolen tarjoamia JavaScript-kirjastoja, jotka helpottavat tilanhallintaa suuremmissa ohjelmissa. Näistä tarkemmin esiteltiin Redux, joka tarjoaa ohjelmalle globaalin tietovaraston tilan säilöntään. Komponentit voivat käyttää ja päivittää tilaa toisistaan ja sijainnistaan riippumatta. Tällöin komponenttien välillä liikutettavan tiedon määrä ja riippuvuudet vähentyvät. Reduxissa tilan päivittämislogiikka on koottu yhteen funktioon, jolloin virheiden löytyminen ja ohjelman skaalautuminen helpottuu. Redux kuitenkin monimutkaistaa ohjelmaa huomattavasti, joten se ei välttämättä sovellu pieniin sovelluksiin kovin hyvin.

Työssä havaittiin, että pienen mittakaavan sovelluksissa Reactin tarjoamat ominaisuudet ja työkalut ovat riittäviä sovelluksen ylläpidettävyyden kannalta. Ohjelman koon kasvaessa ja ongelmien kasaantuessa, ulkoisten kirjastojen käyttö on kuitenkin lähes välttämätöntä.

Lähdeluettelo

[1] Thomas, Mark Tielens, Eastmond, Tomz, Jakovcevic, Toni, Yockey, Megan, and Robb, Dan. React in Action . 1st edition. Shelter Island, NY: Manning Publications Co., 2018. Print.

[2] Banks, Alex, and Porcello, Eve. Learning React, 2nd Edition. 2nd ed. O'Reilly Media, Inc, 2020. Print.

[3] Chinnathambi, Kirupa. Learning React: A Hands-On Guide to Building Web Applications Using React and Redux, Second Edition. 1st ed. Addison-Wesley Professional, 2018. Print.

[4] Scott, Emmit A. SPA Design and Architecture : Understanding Single-Page Web Applications. Place of publication not identified: Manning, 2016. Print.

[5] Lindley, Cody. Dom Enlightenment. First edition. Sebastopol, California: O'Reilly Media, 2013. Print.

[6] React. Components and Props. [Viitattu 7.10.2020]. Saatavissa: *https://reactjs.org/docs/components-andprops.html*

[7] React. Thinking in React. [Viitattu 29.10.2020]. Saatavissa: *https://reactjs.org/docs/thinking-in-react.html*

[8] Horstmann, Cay S. Modern JavaScript for the Impatient. 1st ed. Addison-Wesley Professional, 2020. Print.

[9] React. Context. [Viitattu 30.10.2020]. Saatavissa: *https://reactjs.org/docs/context.html*

[10] React. State and Lifecycle. [Viitattu 7.10.2020]. Saatavissa: *https://reactjs.org/docs/state-and-lifecycle.html*

[11] Redux. Redux Essentials, Part 1: Redux Overview and Concepts. [Viitattu 15.10.2020]. Saatavissa: *https://redux.js.org/tutorials/essentials/part-1-overview-concepts*

[12] Boduch, Adam, and Derks, Roy. React and React Native - Third Edition. 3rd ed. Packt Publishing, 2020. Print.

[13] Bugl, Daniel. Learning Redux. 1st ed. Packt Publishing, 2017. Print.

[14] Faurot, Marc. Redux in Action. 1st ed. Manning Publications, 2018. Print.

[15] React. Lifting State Up. [Viitattu 29.10.2020]. Saatavissa: *https://reactjs.org/docs/lifting-state-up.html*

[16] Podila, Pavan, and Weststrate, Michel. MobX Quick Start Guide: Superc harge the Client State in Your React Apps with MobX . Birmingham;: Packt Publishing, 2018. Print.

[17] MobX. About MobX. [Viitattu 14.11.2020]. Saatavissa: *https://mobx.js.org/README.html*

[18] Buna, Samer. Learning Graphql and Relay: Build Data-Driven React Applications with Ease Using GraphQL and Relay . Birmingham, England;: Packt Publishing, 2017. Print.

[19] Relay. Built for scale. [Viitattu 14.11.2020]. Saatavissa: *https://relay.dev/*

[20] Apollo. Introduction to Apollo Client. [Viitattu 16.10.2020]. Saatavissa: *https://www.apollographql.com/docs/react/*

Liite 1: RGBColor-komponentin toteutus

```
function RGBColor(props) {
  const [r, setR] = useState(255);
  const [g, setG] = useState(255);
  const [b, setB] = useState(255);

  useEffect(() => {
    setR(props.color.r);
    setG(props.color.g);
    setB(props.color.b);
  }, [props.color]);

  return <div>
    red <input value={r} onChange={e => setR(e.target.value)} />
    green <input value={g} onChange={e => setG(e.target.value)} />
    blue <input value={b} onChange={e => setB(e.target.value)} />
    <button onClick={() => props.changeColor(r, g, b)} />
  </div>;
}
```