Joona Enbuska

Tilanhallinta ja yksisuuntainen dataflow ReactJS-sovelluksessa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Opinnäytetyö

19.12.2017

Tekijä(t) Otsikko Sivumäärä Aika	Joona Enbuska Tilanhallinta ja yksisuuntainen dataflow ReactJS-sovelluksessa 61 sivua + 3 liitettä
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Simo Silander

Insinöörityö pyrkii aukaisemaan, mitä yksisuuntainen dafa flow tarkoittaa ja kuinka sitä hyödynnetään ReactJS-sovelluksessa tilan hallintaan.
Vaikka tämän opinnäytetyön tarkastelupohjana käytetäänkin ReactJS:ää, yksisuuntaisen dataflow'n periaatteita voidaan hyödyntää muissakin kuin ReactJS-sovelluksissa.

Aluksi käydään läpi joitakin vaihtoehtoisia tilanhallinnan ratkaisuja ja syitä, miksi yksisuuntaisen dataflow on noussut suosituksi tavaksi hoitaa sovelluksen tilanhallintaa.

Seuraavaksi tarkastellaan, minkälaisia rajapintoja ja käsitteitä ReactJS, ja Reactin ekosysteemi tarjoaa, jotta yksisuuntaisen dataflow'n periaatteita on helppo soveltaa tehdessä ReactJS-sovellusta.

Tämän jälkeen tarkastellaan hieman Reactin de facto -tilanhallintakirjastoa Redux.

Tämän opinnäytetyön tuloksena syntyi prototyyppikirjasto, jonka lähtökohtana on vähentää toisteista koodia React-sovelluksen tilanhallinnassa.

Avainsanat	React, Redux, sovelluskehitys, tilanhallinta, yksisuuntainen dataflow

Author(s) Title Number of Pages Date	Joona Enbuska State management and unidirectional dataflow in ReactJS application 61 pages + 3 appendices
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Simo Silander, Master

This thesis tries to explain what unidirectional dataflow means, and how ReactJS uses it for state management.

Even though most of the code examples are based on ReactJS, the core principles are not React specific, but can be adapted to be used in application development in general.

First, I'll go through some alternative solutions for state managements, and the reasons why unidirectional dataflow has grown in popularity among different patterns.

Next I'll discover what kind of concepts and patterns React ecosystem presents, for implementing unidirectional dataflow in the applications.

After that I'll go through a library called Redux, that probably the most popular library used for state handling with React.

The result of this thesis was a prototype JavaScript state management library for React, that tries to reduce the boilerplate code associated with Reacts state management.

Keywords	React, Redux, software development, state management, unidirectional dataflow

Sisällys

Sisällys 4

1 Johdanto 1

2 Yksisuuntanen dataflow 3

2.1 Yksisuuntaisen dataflow’n määritelmä 3

2.2 Yksisuuntainen dataflow sovelluksessa 5

3 ReactJS ja yksisuuntainen dataflow 10

3.1 JSX-syntaksi 10

3.2 Komponentit 12

3.3 Komponenttimuuttujat 14

3.4 Komponentin tila (state) 15

3.5 Komponentin parametrit (properties) 21

3.6 Komponenttien konteksti (context) 24

3.6.1 Konteksti yleisesti 24

3.6.2 Kontekstidatan määrittäminen 25

3.6.3 Kontekstidatan hyödyntäminen 28

4 Tilanhallinta käyttäen Redux-tilanhallintakirjastoa 31

4.1 Yleisesti 31

4.2 Redux store 31

4.3 Reduserit (Redux reducers) ja Actionit 33

4.4 Redux middlewaret 37

4.5 Yhteenveto Reduxista 40

5 Tilanhallinta hyödyntäen JavaScript Proxyjä (react-proxy-state) 41

5.1 Motivaatio 41

5.2 JavaScript Proxy käsite 41
5.3 Vaikutteet 42
5.4 Ratkaisut 43
5.4.1 Kontekstitilan asettaminen propseihin 43
5.4.2 Kontekstitilan muuttaminen 47
5.4.3 Proxy-solmujen toteutus 54
5.4.4 Rajoitteet 56
6 Pohdinta 57
7 Yhteenveto 59
Lähteet 60

Liite 1. React-Proxy-State Map Context State To Props API-kuvaus
Liite 2. React-Proxy-State ContextProvider API-kuvaus
Liite 3. React-Proxy-State Context eventhandlers API-kuvaus

1 Johdanto

Viime vuosina ovat puhelinsovellukset määrittäneet uudelleen, mitä sovellusten käytettävyydeltä, estetiikalta sekä käyttäjäystävällisyydeltä odotetaan. Odotukset ja hyvin toimivien sovellusten tuoma lisäarvo ovat nostaneet merkitystään myös web-sovellusten kehityksessä.

Single page -web-sovellusteknologia pyrkii osaltaan parantamaan sovellusten käytettävyyttä tuottamalla lähempänä natiivisovellusta olevan käyttökokemuksen. Koska sovelluksen käyttöliittymä ei ole riippuvainen sivustoa tarjoilevasta palvelimesta vaan ainoastaan käyttäjän selaimesta, voidaan suuri osa sovelluksen logiikasta suorittaa paikallisesti käyttäjän laitteessa, jonka seurauksena sovelluksen käyttökokemuksesta voidaan tehdä nopea ja saumaton.

Perinteiset web-sivut lataavat kulloisenkin näkymän erillisenä palvelimelta. Kukin näkymä sisältää oman HTML-tiedostonsa, ja jossain määrin CSS:ää ja JavaScriptiä. Näkymien responsiivisuutta, kuten animaatioita, formivalidointeja, sekä elementtien piilottamista ja näyttämistä hallitaan käyttäen apukirjastoja ja selaimen rajapintoja. Selaimessa suoritettava JavaScript ei ota paljoa kantaa sovelluksen bisneslogiikkaan vaan sen vastuuna on huolehtia pääasiassa näkymien visuaalisista yksityiskohdista.

Single page -sovellukset (SPA, Single page applications) ovat monella tapaa kompleksisempia ja enemmän työtä vaativia kokonaisuuksia verrattuna perinteisiin websivuihin. Kun ensimmäisiä Single page -sovelluksia toteutettiin, ei single page -sovellusten tekemiseen ollut selkeitä yleisiä parhaita käytäntöjä, sillä mallit, joilla JavaScriptin näkymälogiikkaa oli alkujaan ohjattu, ei skaalautunut laajaan bisneslogiikan hallintaan. Ensimmäisen sukupolven SPA-ohjelmistokehykset kuten Angular, Ember ja Backbone, antoivat paremman pohjan sille, kuinka single page -applikaatioista saatiin strukturoituja sekä helpommin hallittavia. Näiden sovelluskehyksien kanssa tilanhallinta ja suorituskyky olivat toistuvia ongelmia.

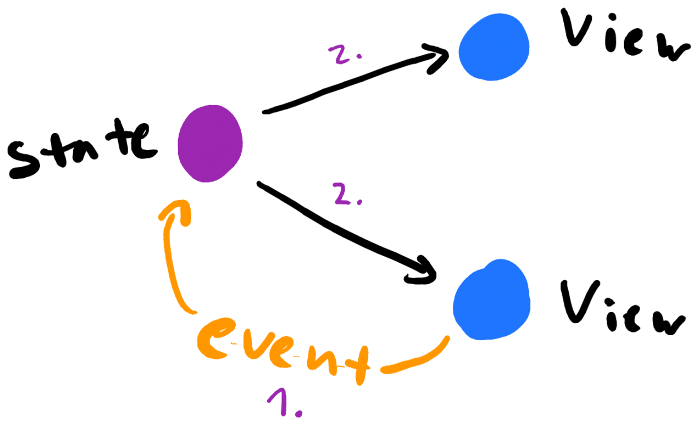
Toisen sukupolvien SPA-sovelluksien toteuttamiseen tehdyt kirjastot, kuten React ja Vue pyrkivät olemaan kirjastoja, jotka huolehtivat vain siitä, miltä käyttöliittymä näyttää. Nämä

kirjastot välttävät ottamasta kantaa, kuinka sovelluksen muuta logiikkaa hallitaan, mutta kummatkin näistä kirjastoista ohjaavat voimakkaasti noudattamaan yksisuuntaisen dataflow'n periaatteita, mikä on osoittautunut selkeäksi ja suorituskykyiseksi tavaksi hallita sovelluksen tilaa.

2 Yksisuuntanen dataflow

2.1 Yksisuuntaisen dataflow'n määritelmä

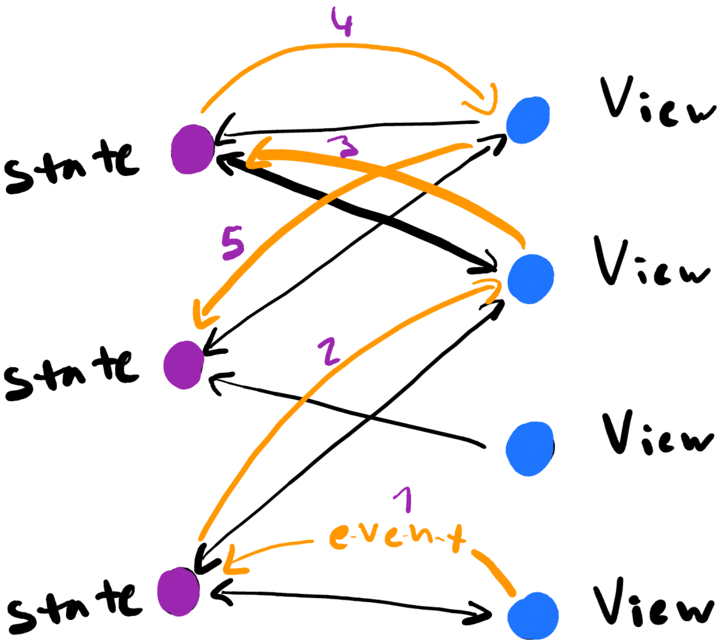
Yksisuuntainen dataflow on malli, joka rajoittaa, kuinka sovelluksen tilaa voidaan observoida ja muuttaa. Sovelluksessa tila määrittää näkymän sisällön. Muutokset näkymässä eivät ole mahdollisia, jos niitä määrittävä tila ei muutu. Täten näkymä on esitys sitä määrittävästä tilasta.



Kuva 1 Yksisuuntainen dataflow

Kuvassa 1 on kaksi yhteiseen tilaan (state) sidottua näkymää (view). Näkymien tapahtumat, kuten näppäinpainallukset voivat luoda tapahtumia, jotka muuttavat tilaa. Kun näkymiä määrittävä tila muuttuu, päivittyvät myös tilaa esittävät näkymät. Tämän ominaisuuden etuna on muun muassa, ettei näkymiä tarvitse deklaratiivisesti kontrolloida, sillä tilanmuutokset automaattisesti johtavat näkymien päivittymiseen.

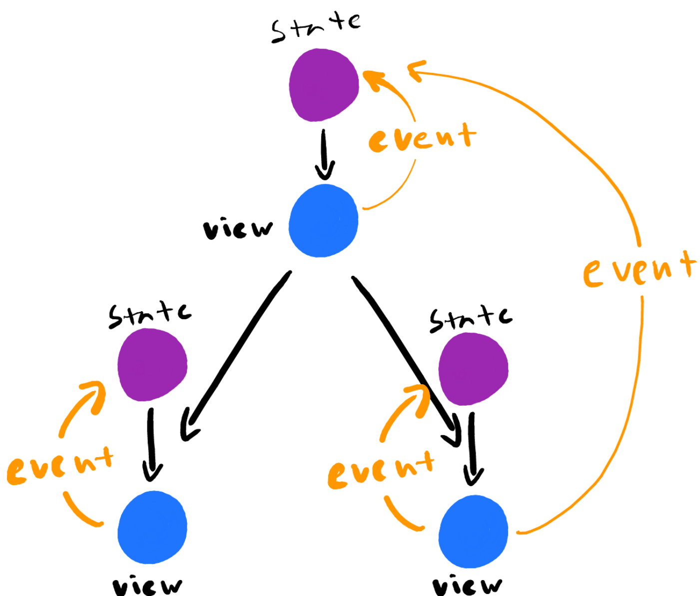
Toinen tapa ajatella yksisuuntaista dataflow'ta on, että näkymät ovat tilan lapsia, jotka tarkkailevat vanhempaansa. Vastaavasti tila on näkymien vanhempi, joka ei tarkkaile lapsiaan. Aina, kun tila päivittyy, päivittyvät myös tilaa observoivat lapsinäkymät. Koska tila ei observoi näkymiä, eivät tilamuutokset voi johtaa dominoefektin kaltaisiin sivuvaikutuksiin. [1.]



Kuva 2 Monisuuntainen observointi

Kuva 2 esittää tilannetta, jossa näkymät observoivat tiloja ja tilat observoivat näkymiä. Tämä menetelmä niin kutsuttu two-way-binding, jota on käytetty esim. Angular-kehityksessä. Tämän kyseisen toimintamallin on todettu aiheuttavan vaikeita tilanhallinnan ja suorituskyvyn ongelmia.

Kokonaisessa sovelluksessa näkymät ja tilat elävät puumaisessa yksisuuntaisessa hierarkiassa. Kun useampi hierarkkisesti rinnakkainen näkymä on riippuvainen samasta tilasta, nostetaan näiden yhteinen tila näiden näkymien lähimpään yhteiseen vanhempaan, joka voi olla näiden näkymien vanhempinäkymä, tai muulla tavalla nämä kummatkin näkymät kattava konteksti.

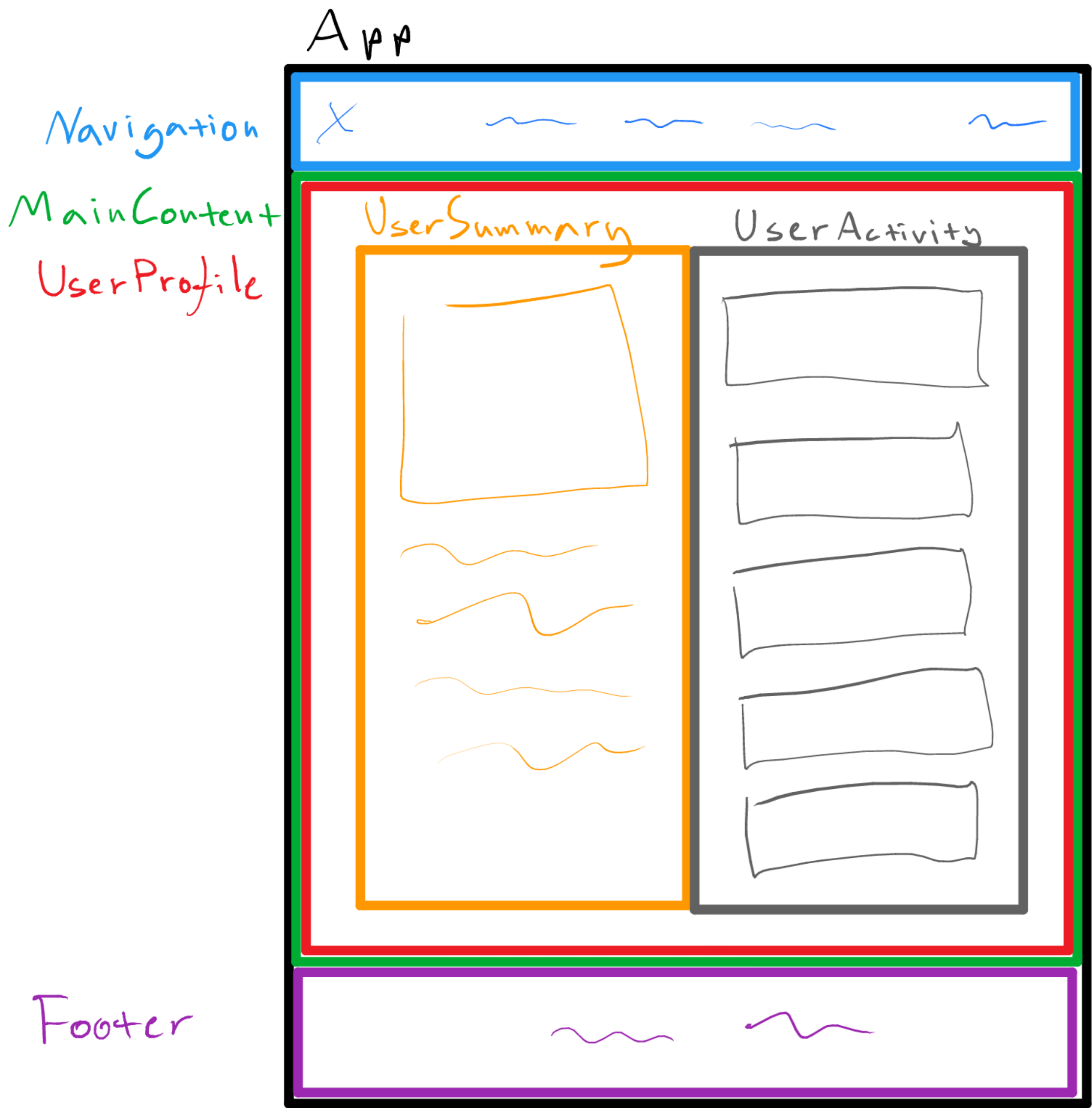


Kuva 3 Hierrakkinen tila

Kuvassa 3 on kolme näkymää. Jokaiseen näkymään on assosioitu näkymään liittyvä tila. Kuvassa alempana sijaitsevien hierarkkisesti rinnakkaisten näkymien yhteinen tila on nostettu näiden näkymien yhteiseen vanhempinäkymään, joten näiden näkymien kokonaistila koostuu kahdesta tilasta.

2.2 Yksisuuntainen dataflow sovelluksessa

Hyvänä lähtökohtana yksisuuntaisen dataflow'n soveltamiselle on, että se on selkeästi strukturoitavissa puumaiseen muotoon. Kuvassa 4 otetaan esimerkiksi kuvitteellisen käyttäjän profiilisivu, joka sisältää sisäkkäisiä näkymiä.



Kuva 4 Hierarkkinen näkymä

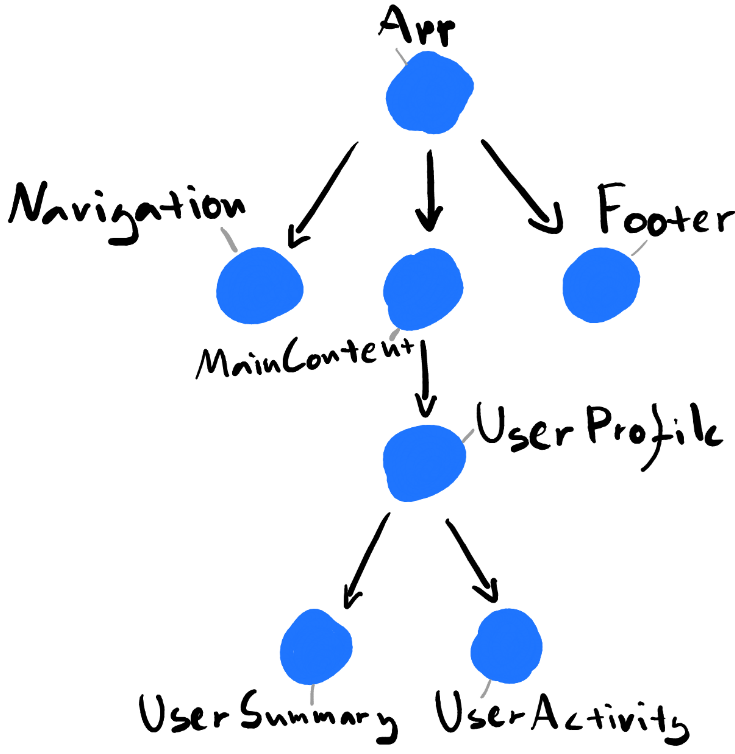
Kuvan 4 esimerkissä hierarkian ylin näkymän on 'App', joka sisältää näkymät 'Navigation', 'MainContent' ja 'Footer'. 'MainContent' sisältää 'UserProfile'-näkymän, joka sisältää vuorostaan näkymät 'UserSummary' ja 'UserActivity'.

Tällaista näkymien välistä hierarkiaa voidaan kuvata vanhempi-lapsi-suhteena. Kuvan esimerissä 'App' on 'Navigation'-, 'MainContent'- ja 'Footer'-näkymien vanhempi, ja vastaavasti nämä näkymät ovat 'App'-näkymän lapsinäkymiä.

Jotta näkymähierarkiat olisi mahdollista esittää puuhierarkiana, näkymien välisissä viittauksissa ei tulisi olla kehäviittauksia. Näkymän vanhempi voi viitata lapsinäkymiinsä,

joita voi olla useita, mutta lapsinäkymän ei tulisi viitata vanhempaansa. Tämän lisäksi jokaisella näkymällä tulisi olla enintään yksi vanhempi. Rinnakkain sijaitsevien näkymien kuten Navigation ja MainContentin ei tulisi viitata toisiinsa.

Näkymävanhemman tehtävä on antaa lapsinäkymilleen parametreja sekä ilmoittaa lapsinäkymilleen, kun näiden täytyy uudelleen päivittyä. Jos näkymähierarkiat sisältäisivät kehäviittauksia, johtaisi näkymien päivittyminen ikuiseen rekursioon.



Kuva 5 Näkymän hierarkiaesitys puuhiearkiana

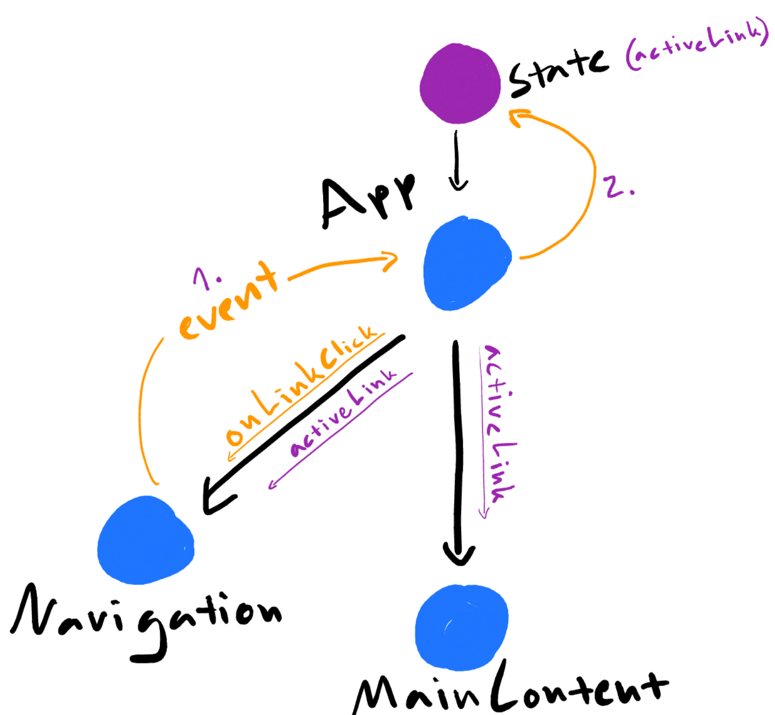
Kuvassa 5 on kuvan 4 käyttöliittymänäkymien hierarkia määritelty puurakenteena. Tästä puurakenteesta voi päätellä, että ’Navigation’-näkymällä ei ole viittausta ’MainContent’- tai ’Footer’-näkymiin. Jotta ’Navigation’-näkymän tapahtumat voisivat heijastua myös tämän rinnakkaisiin ’MainContent’- sekä Footer-näkymiin, täytyy näiden näkymien yhteinen tila olla peräisin niiden yhteisestä vanhempinäkymästä, joka kuvan esimerkissä on ’App’-näkymä.

Tilan muuttuessa uusi tila valuu järjestyksessä ylemmästä näkymistä alempiin näkymiin rekursiivisesti. Joten kun tila muuttuu, päivittyvät kaikki tilamuutoksen alapuolella olevat näkymät.

Sovellettaessa yksisuuntaista dataflow`ta tilan tietorakenteet tulisi pitää mielellään muuttumattomina. Jos tilaa halutaan muuttaa, täytyy tilamuutos tehdä luomalla uusi tila siinä hierarkiasolmussa, josta se on lähtöisin. Tämä johtuu siitä, että usein samaa tilaa hyödynnetään monella eri tasolla sovelluksen näkymiä, tai tila voi olla johdettu muualla hyödynnetystä tilasta.

Kun tilan muutos suoritetaan tilan lähteessä, varmistutaan, että kaikki muutoksista kiinnostuneet osapuolet saavat tiedon muuttuneesta tilasta. Jos tilaa muutetaan muualla kuin kyseisen tilan lähteessä, seuraa tästä, että toiset näkymät, jotka hyödyntävät samaa tilaa, päätyvät epäsymmetriaan todellisen tilan kanssa.

Usein lapsinäkymät tarvitsevat kyvyn muuttaa itseään hierarkkisesti korkeammalla määrittyä tilaa. Tätä varten näiden näkymien vanhemmat voivat antaa parametrina lapsinäkymilleen callback-funktioita (*Callback function* [2.]), joita kutsuttaessa vanhempinäkymä päivittää tilaansa.



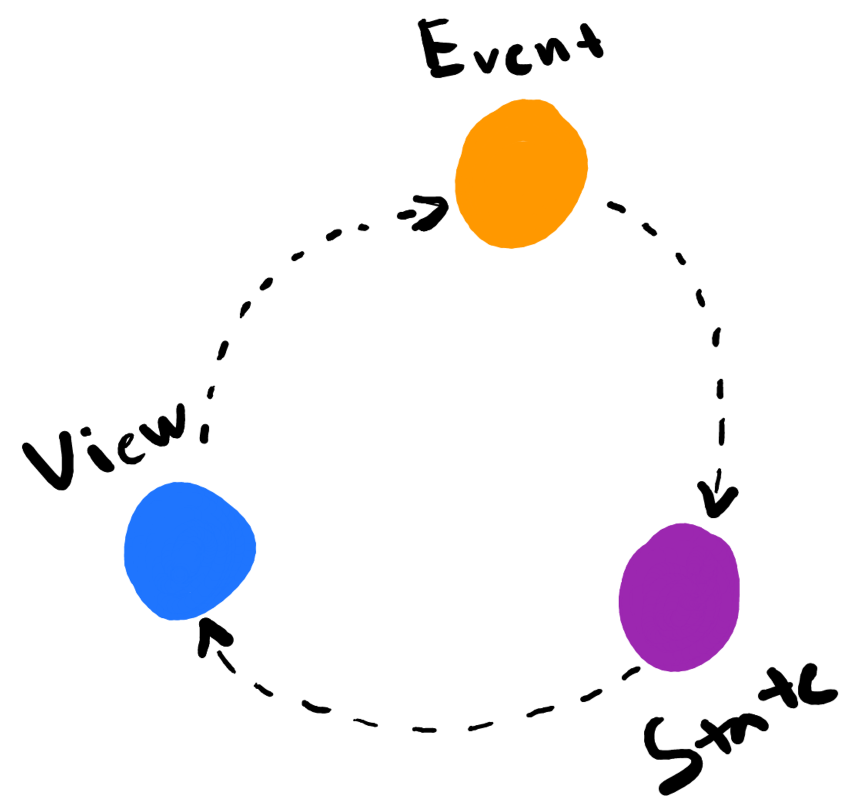
Kuva 7 Tilanmuuttaminen käyttäen callback-funktiota

Kuvan 7 esimerkissä 'App'-näkymän tila sisältää tiedon siitä, mikä on tämänhetkinen aktiivinen linkki. 'App' antaa 'Navigation'-näkymälle parametrina 'onLinkClick' callback-

funktion, jota kutsuttaessa ’App’ muuttaa ’activeLink’-tilansa vastaamaan funktiolle annettua parametria. Tästä seuraa, että kaikki ’App’-näkymän lapsinäkymät ’MainContent’ mukaan lukien päivittyvät uusine parametreineen, kun ’App’-näkymän tila päivittyy.

Sovelluksen tila voi elää monella tasolla puuhierarkiaa, mutta usein suurin osa sovelluksen tilasta elää sovelluksen hierarkkisesti korkeimmassa moduulissa.

Yleisesti kuvattuna yksisuuntaisessa dataflow’ssa tapahtumat muuttavat tilaa, josta seuraa tilan päivittyminen, josta vuorostaan seuraa tapahtuman tulos, joka käyttöliittymän kontekstissa tarkoittaa päivittynyttä näkymää.



.

Kuva 8 Yleinen kuvaus yksisuuntaisesta dataflow’sta

Kuva 8 havainnollistaa yleisesti yksisuuntaista dataflow’n elinkaarta. Yksisuuntaista dataflow’ta on mahdollista soveltaa käyttöliittymien rakentamisessa monella tapaa, mutta ne kaikki noudattavat tätä samaa tilan elinkaarta. Seuraavat luvut käsittelevät yksisuuntaisen dataflow’n soveltamista React-sovelluksissa, jossa näkymät ja tila määritetään synkronisesti toimivana puuhierarkiana. Kaikki yksisuuntaisen dataflow’n ratkaisut eivät

suinkaan muistuta paljoa toisiaan, esim. käyttäessä ReactiveX:n kirjastoja RxJava tai RxJS tilat määritetään asynkronisina datavirtoina, eikä näkymien ja tilojen hierarkioiden suinkaan tarvitse olla selkeästi strukturoituja. Rx loistaa käyttöliittymäkirjastojen ja ohjelmistokehyksien kanssa, joita käyttäessä näkymien toiminnallisuudet ja kuvaukset määritetään imperatiivisesti. Kirjastot kuten React ja Vue rakentuvat yksisuuntaisen dataflow'n periaatteen päälle, joten Rx-kirjastojen hyödyntäminen näiden kirjastojen rinnalla voi turhaan monimutkaistaa sovelluksen tilan hallintaa. [3.]

3 ReactJS ja yksisuuntainen dataflow

3.1 JSX-syntaksi

JSX-syntaksi on JavaScript-syntaksin laajennus, joka mahdollistaa XML:n syntaksin lisäämisen JavaScriptin sekaan. React-sovelluksissa JSX toimii korkean abstraktion kuvauskielenä, joka buildi-vaiheessa käännetään perinteiseksi JavaScriptiksi.
Seuraavissa luvuissa pyritään kuvaamaan, kuinka JSX-syntaksi itsessään ohjaa noudattamaan yksisuuntaista dataflow'ta. [4.]

```
const element = <h1 className='header'>Hello JSX</h1>;
```

Kuva 9 JSX-elementti

Kuvassa 9 on luotu h1-elementti käyttäen JSX-syntaksia. React-sovelluksen näkymien pienimpiä yksiköitä ovat elementit. Jos sovellusta tehdään selainta tai Chromiumia varten, näkymän elementit ovat html:n yksiköitä kuten *div*, *h1* jne.
React Native -sovelluksissa periaatteet ovat hyvin samanlaiset, mutta elementit, joita käytetään, ovat *Button*, *View* jne.

Web-kehityksessä JSX-html-elementtien syntaksi muistuttaa hyvin pitkälle html:lää, mutta se sisältää joitakin eroavaisuuksia kuten sen, että kaikki JSX:ssä määritetyt tagit täytyy sulkea, sekä elementtien attribuutit eli Reactin kielellä propertyt kuvataan pääasiassa camelCase-syntaksilla.

```
const element = React.createElement(
  'h1',
  {className: 'header'},
  ['Hello JSX']
);
```

Kuva 10 JavaScriptiksi käännetty JSX-koodi

Kuvassa 10 on esitetty, miltä kuvassa 9 määritetty JSX-h1-elementti näyttää, kun se käännetään JavaScriptiksi.

```
{
 "type": "h1",
 "key": null,
 "ref": null,
 "props": {
  "className": "header",
  "children": [
   "Hello JSX"
  ]
 },
 "_owner": null,
 "_store": {}
}
```

Kuva 11 createElement-funktion yksinkertaistettu paluuarvo

Kuvassa 11 on yksinkertaistettu esitys createElement-funktion palauttamasta objektista. Sovellusnäkymät, jotka sisältävät useita elementtejä, rakentavat objektipuun, jotka koostuvat tämän kaltaisista objekteista. Tätä objektipuuta joskus kutsutaan nimellä virtuaalinenDOM.

JSX-syntaksin sekaan on myös mahdollista kirjoittaa perinteistä JavaScriptiä. Kaikki perinteinen JavaScripti JSX-syntaksin seassa tulee kirjoittaa aaltosulkeiden sisään esimerkin 12 kuvaamalla syntaksilla.

```
<div className="button-container">
  <button onClick={() => console.log('clicked')}>Click Me</button>
</div>
```

Kuva 12 Perinteinen JavaScript JSX-syntaksin sisällä

Kuvan 12 esimerkissä määritetään div, jolle annetaan css class propertyksi 'button-container'-luokka. Tämän div:i sisältää napin, jolle on annettu anonyymi onClick callback-

funktio, joka tulostaa konsoliin tekstin ’clicked’, kun nappia painetaan. Koska tämä onClick-kuuntelija on määritetty aaltosulkeiden sisällä, on sen sisältö voitu määrittää normaalina JavaScriptinä.

3.2 Komponentit

Käyttämällä JSX-elementtejä rakennetaan näkymäkokonaisuuksia, joita kutsutaan komponenteiksi. Komponentit ovat joko React **Component** -luokan instansseja tai funktioita, jotka palauttavat JSX-elementtejä [5.].
Reactin *elementit* kuten h1 ja div ovat ikään kuin Reactin natiivikomponentteja.

React-sovelluskehityksessä käsite näkymä tarkoittaa samaa kuin komponentti, sillä kaikki React-sovelluksen näkymät ovat komponentteja. Sen sijaan kaikki komponentit eivät sisällä näkymää.

Komponentit pystyvät käyttämään myös muita komponentteja määrityksessään. Näitä komponentin käyttämiä muita komponentteja kutsutaan komponentin lapsiksi (*children*).

```
const Header = () => <h1>Navigate to</h1>;

const Navigation = () => (
  <div>
    <Header />
    <a href='/home'>Home</a>
    <a href='/profile'>Profile</a>
  </div>
);
```

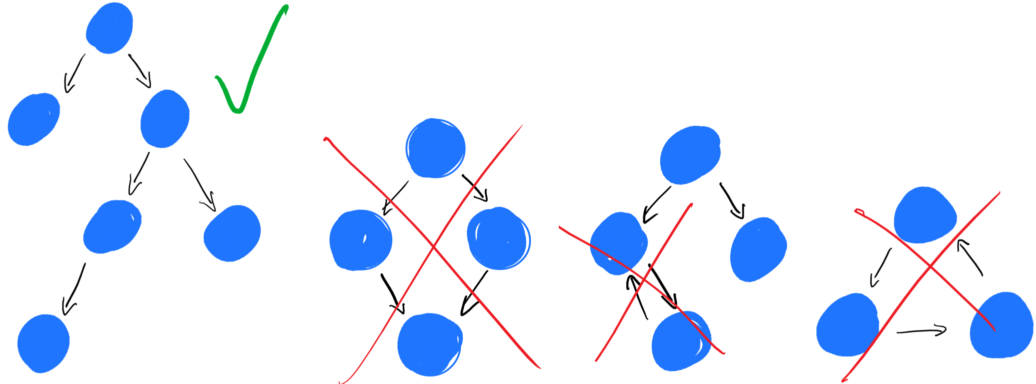
NAVIGATE TO

Home
Profile

Kuva 13 JSX-komponentit

Kuvan 13 Navigation-komponentti palauttaa div-elementin, joka sisältää Header-komponentin, sekä kaksi linkkiä. Header- ja a-tagit ovat Navigation-komponentin lapsia, ja vastaavasti Navigation on näiden vanhempi.

JSX-syntaksi itsessään ohjaa xml-syntaksin lailla noudattamaan yksisuuntaisen dataflow’n periaatteita pakottamalla komponettikokonaisuudet hierarkiseen puumuotoon.



Kuva 14 Komponenttirelaatiot

Komponenttien relaatiot ovat yksisuuntaisia vanhempi/lapsi-suhteita, joissa lapsi ei tunne vanhempaansa mutta vanhempi hallinnoi lapsiaan. Komponentin vanhempi ei suoranaisesti tunne, eikä instanssioi lapsikomponenttejaan, mutta se voi tietää näiden tyypit, ja on kykenevä antamaan näille parametrejä, joita kutsutaan React-kehityksessä propertyiksi.

Komponentin näkymägenerointiin selaimessa käytetään ReactDOM-kirjaston render-funktioita, joka ottaa vastaan sovelluksen juurikomponentin sekä html-elementin, johon komponentin tuottama näkymä renderöidään. Tämä render-kutsu on usein React-sovelluksen ensimmäinen sekä viimeinen suora viittaus selaimen html-dokumenttiin. Kun React-sovelluksen komponentit päivittyvät, ReactDOM-kirjasto huolehtii html-dokumentin päivittämisestä ohjelmoijan puolesta.

Render tuottaa komponenttipuusta kuvan 11 kaltaisen JavaScript-objektin, jota ReactDOM hyödyntää, kun muutoksia injektoidaan render-funktiolle toisena parametrina annettuun html-elementtiin.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
<body>
  <div id="app"/>
  <script
    type="text/javascript"
    src="./index.js"/>
</body>
</html>
```

```
ReactDOM.render(
  Navigation,
  document.getElementById('app')
);
```

Kuva 15 ReactDOM render

Kuvan 15 esimerkissä html-dokumentista kutsutaan index.js tiedostoa, joka puolestaan viittaa takaisin dokumentin 'app'-id:llä löytyvään elementtiin ja renderöidään kuvan 13 esimerkin 'app'-elementin sisään. Perinteisesti React-sovelluksissa renderöidään vain yksi juuritason komponentti DOM:iin, joka sisältää koko sovellusnäkymän.

3.3 Komponenttimuuttujat

Kaikilla komponenteilla on kolme muuttujatyyppiä.

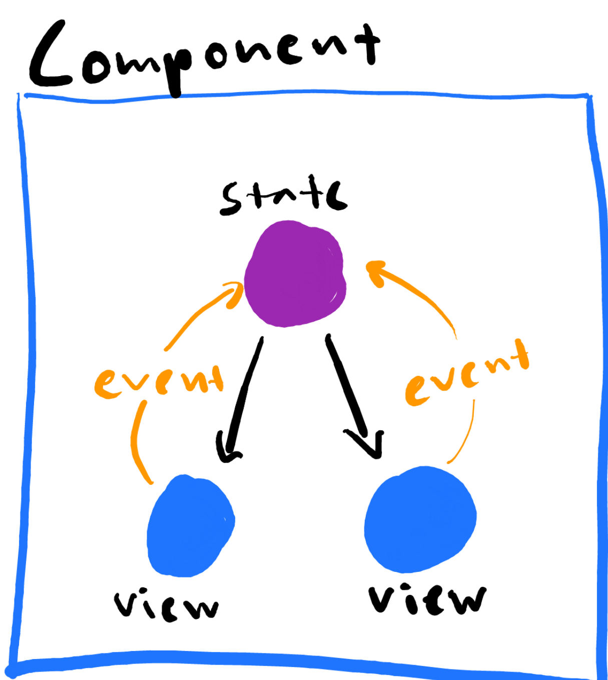
1. state-muuttuja on komponentin itsensä omistama tila. Puhuttaessa komponentin tilasta puhutaan yleensä komponentin state muuttujasta. Suurimmalla osalla komponenteista ei yleensä ole state-muuttujaa.
2. props-muuttuja (properties) on komponentille sen vanhemman välittämä muuttuja, joka voi päivittyä usean kerran komponentin elinkaaren aikana.
3. context-muuttuja on komponentin vanhemman tai esivanhemman välittämä muuttuja.

Yleisellä tasolla puhuttaessa React-sovelluksen tilasta viitataan tässä yhteydessä usein näiden kaikkien muuttujien muodostamaan kokonaistilaan.

Kaikki React-sovelluksen tilaan liittyvät tietorakenteet on suositeltavaa pitää muuttumattomina. Muuttumattomien tietorakenteiden eduista ja menetelmistä, kuinka tilaa muutetaan käyttämällä muuttumattomia tietorakenteita, esitellään seuraavassa luvussa.

3.4 Komponentin tila (state)

Aikaisemmassa esimerkissä 13 komponentit määritettiin funktioina. Jos komponentti määritetään 'React-Component'-luokkana, on sille mahdollista määrittää tila/**state**. State on komponentin itsensä hallinnoima tila, jota se pystyy myös jakamaan lapsikomponenteilleen. Tilalliset komponentit ovat kombinaatio tilaa ja näkymää. Komponentilla on valta muuttaa omaa tilaansa, ja tätä kautta mahdollisuus päivittää näkymää. [6.]



Kuva 16 Tilallinen komponentti

Myös yksittäinen komponentti noudattaa yksisuuntaista dataflow'ta sisäisessä tilanhallinnassaan. Yksinkertaistettuna voidaan ajatella, että myös komponentin tila on komponenttinäkymän vanhempi, jota komponentin näkymät observoivat.

Jotta komponentille on mahdollista määrittää tila, komponentti täytyy toteuttaa React Component-luokkana. Komponentti, joka on määritetty aikaisempien esimerkkien lailla funktiona, ei voi sisältää omaa tilaa.

Komponentin state on instanssimuuttuja jota ei saa suoraan muokata. Sen sijaan komponentti voi päivittää tilansa kutsumalla 'Component'-luokan metodia **setState**. Tilan muuttamiseksi tälle metodille annetaan parametriksi objekti, jota hyödyntämällä setState-toteutus luo komponentille uuden state-muuttujan, pintakopioimalla (shallow copy) komponentin nykyisen tilan ja yhdistämällä tämän kopion parametrina annetun objektin kanssa. Komponentin setState-metodille on myös mahdollista antaa callback-funktio parametriksi, mutta tämä opinnäytetyö ei käsittele kyseistä tapausta.

Kun komponentin tila on muuttunut, kutsuu React komponentin render-metodia, jonka seurauksena renderöidään myös komponentin lapsikomponentit rekursiivisesti, josta seuraa, että komponentin kattava näkymäosa päivittyy.

```
class LoginForm extends React.Component{

  state = {username: '', password: ''};

  render(){
    return (
      <form onSubmit={this.handleSubmit}>
        <input
          value={this.state.username}
          onChange={(e => this.setState({username: e.target.value}))}
          placeholder="username"/>
        <input
          type="password"
          value={this.state.password}
          onChange={(e => this.setState({password: e.target.value}))}
          placeholder="username"/>
      </form>
    )
  }

  handleSubmit(e){
    /*handle sending of request*/
  }

}
```

username

password

Kuva 17 Komponentin state-esimerkki

Kuvassa 17 on LoginForm-komponentti määritetty tilallisena komponenttina. Jokaisen luokkana määritetyn komponentin täytyy toteuttaa Component-rajapinta. Tämä tarkoittaa, että luokan täytyy toteuttaa render-metodi. Tätä funktiota React kutsuu, kun käyttöliittymä renderöidään uudelleen. Render-funktiota ei koskaan kutsuta itse omasta koodista. Esimerkissä 'username' ja 'password' input-kentät saavat arvokseen komponentin vastaavat tilat. Kun input-kenttiin kirjoitetaan, input-elementti kutsuu sille annettua onChange callback-funktiota, joka esimerkin tapauksessa päivittää komponentin tilaa. Tämän seurauksena näkymä tulee päivittymään käyttäen hyväksi komponentin muuttunutta tilaa.

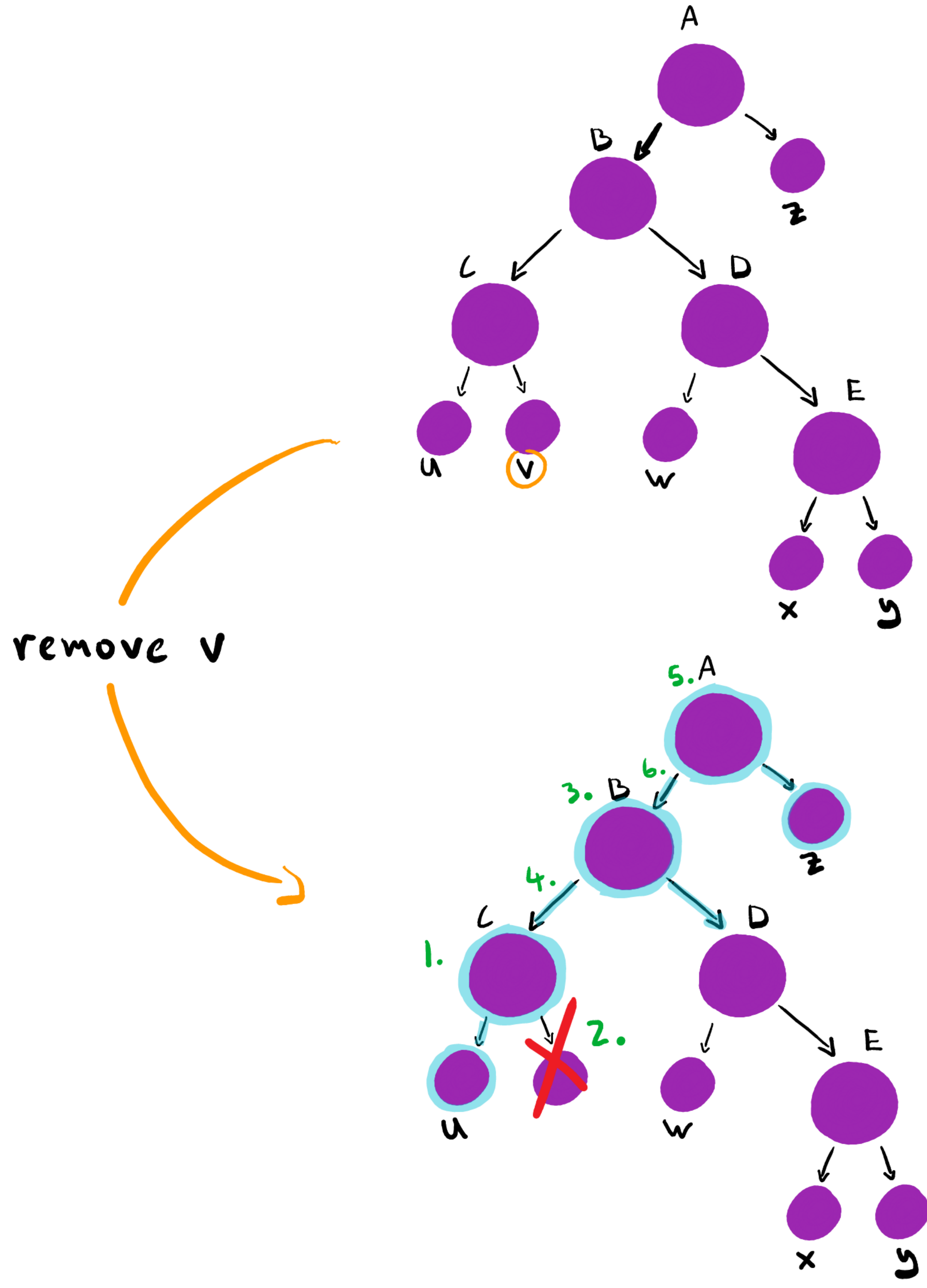
Tämän esimerkin input-elementtien arvot on siis sidottu vastaamaan komponentin tilaa. Tämän seurauksena, jos input-kenttiin kirjoitetaan, mutta komponentin tilaa ei muutettaisi, pysyisivät input-kenttien arvot muuttumattomina. Tämä toimintamalli mahdollistaa sen, että kun onChange-event tulee kutsutuksi, voidaan teksti formatoida ja validoida ennen kuin komponentin tila ja näkymä päivittyvät.

Näkymäkomponentin tilan hyödyntäminen tulisi rajoittaa vain synkronisiin tapahtumiin, sillä komponentin näkymä saattaa poistua ennen asynkronisen tapahtuman tulosta.

Usein komponentin tila voi olla johdettu tilasta, jota käytetään myös rinnakkaisissa komponenteissa. Jotta yksisuuntaista dataflow olisi mahdollista toteuttaa ilman ei-toivottuja sivuvaikutuksia, tulisi komponentin tila aina päivittää muuttamatta objektiviittauksia. Yleisimmin tämä tarkoittaa, että joka kerta, kun komponentin tilaa halutaan muuttaa, luodaan uusi tila polkukopioimalla alkuperäinen tila, muutoksen kohteesta tilan juureen saakka.

Polkukopiointi on yksi tavoista, joilla muuttumattomia tietorakenteita käsitellään, kun olemassa olevasta tietorakenteesta halutaan johtaa uusi tietorakenne / tila. Syvässä kopioinnissa jokaisesta tietorakenteessa sijaitsevasta viittauksesta luodaan rekursiivisesti uusi viittaus, jonka viittaukset asetetaan osoittamaan kopion tuottamiin uusiin muistiosoitteisiin ja arvoihin. Sen sijaan polkukopioidessa kopioidaan vain tietorakenteen osat, jotka ovat suorassa tai epäsuorassa relaatiossa muuttuvaan tietorakenteen solmuun. Polkukopioimalla luotu uusi tietorakenne on muilta osin täysin johdettu aikaisemmasta tietorakenteesta. Verrattuna syväkopioon tämä lähestymistapa voi olla eksponentiaalisesti tehokkaampi tapa varmistua siitä, että komponentin tila pysyy muuttumattomana. Jotta polkukopiointi olisi mahdollisimman suoraviivainen toteuttaa, täytyy tietorakenteella olla yksiselitteinen juuri, eivätkä tietorakenteen viittaukset saa sisältää kehäviittauksia.

Seuraavassa esimerkissä demonstroidaan, kuinka arvo, joka on syvällä olemassa olevassa tietorakenteessa, poistetaan tietorakenteesta käyttämällä polkukopiointia



Kuva 18 Polkukopionti

Kuvassa 18 demonstroidaan yleisellä tasolla, kuinka polkukopiointi suoritetaan hierakkisessa tietorakenteessa. Kuvassa nuolet kuvaavat viittauksia. Pallot **A-E** ovat objekteja, eli osoitteita, jotka viittaavat tietorakenteisiin. Pallot **u-z** ovat primitiiviarvoja kuten numeroita tai kirjaimia. Kuvassa demonstroidaan, kuinka primitiivi **u** poistetaan tietorakenteesta **C**, joka on tietorakenteen **B** alitietorakenne, joka vastaavasti on tietorakenteen **A** ali tietorakenne. Arvon **u** poistaminen on kuusivaiheinen prosessi. Käytännössä näiden askeleiden määrä on mahdollista puolittaa, sekä ne on mahdollista suorittaa vastakkaisessa järjestyksessä, mutta selkeyden vuoksi polkukopionti esitetään jäsennettynä yksinkertaisiin vaiheisiin.

1. Arvoon **u** viittaavasta objektista **C** luodaan pintakopio.
2. **C**:n kopiosta poistetaan arvo **u**.
3. Alkuperäiseen **C** objektiin viittaavasta objektista **B** tehdään pintakopio.
4. **B**-kopion viittaus (**B**-kopio -> **C**) korvataan viittauksella (**B**-kopio -> **C**-kopio).
5. Alkuperäiseen **B** objektiin viittaavasta objektista **A** tehdään pintakopio.
6. **A**-kopion viittaus (**A**-kopio -> **B**) korvataan viittauksella (**A**-kopio -> **B**-kopio).

Muuttumattomien tietorakenteiden hyödyntäminen sisältää useita etuja suhteessa muuttuviin tietorakenteisiin.

- Muuttumaton tila tekee koodista (kiistellysti) helposti pääteltävää ja testattavaa.
- Tilan muutoksilla ei ole sivuvaikutuksia.
- Monisäikeinen laskenta on turvallista.
- Aikaisemmat tilat eivät koskaan muutu, joten niitä voidaan uudelleen käyttää.
- Yksisuuntaisen dataflow'n noudattamisesta intuitiivista.
- Tila voidaan päätellä muuttuneeksi vertailemalla nykyisen ja aikaisemman tilan objekti viittauksia.

Viimeiseksi mainittu muuttumattomien tietorakenteiden etu helpottaa Reactin suorituskykyoptimointia.

Käytännössä muuttumattoman tilan toteuttaminen saattaa aluksi olla vaikea toteuttaa, jos tila on hyvin hierarkkinen, mutta JavaScript-kieli sisältää apuoperaatioita, joiden käyttäminen helpottaa polkukopioimisen toteuttamista. Seuraava esimerkki on hieman triviaali, mutta se pyrkii demonstroimaan joitakin vaikeuksia, joita muuttamaton tilan päivittäminen JavaScript-kielellä tuo mukanaan.

```
class Todos extends React.Component{
  state = {todos: {
    a: {id: 'a', done:false, description: 'Do Homework'},
    b: {id: 'b', done:true, description: 'Wash dishes'}
  }};
  render(){
    const todoViews = [];
    for(const todoItem of this.state.todos){
      todoViews.push(
        <div key={todoItem.id}>
          <p>{todoItem.describe}</p>
          <input
          type="checkbox"
          checked={todoItem.done}
          onChange={() => this.handleTodoToggle(todoItem.id)}
          />
        </div>
      )
    }
    return todoViews;
  }

  handleTodoToggle(todoId){
    const todos = {...this.state.todos}; //shallow copy
    const todo = {...todos[todoId]}; // shallow copy
    todo.done = !todo.done; // toggle
    todos[todoId] = todo; //assign copyed todoitem
    this.setState({todos});
  }
}
```

Do homework

Wash dishes

Kuva 18 Hierarkkisen tilan muuttaminen

Esimerkissä 18 'Todos'-komponenttitila sisältää avain-arvo-pareina 'todo'-alkioita, jotka löytyvät tilasta näiden id-arvolla. Komponentin render-funktio palauttaa listan 'todo'-näkymiä. Jokaiselle komponentin 'todo'-näkymälle on myös annettu avain (key), koska jos komponentti palauttaa listan, joka sisältää useita näkymiä, täytyy näkymälle antaa uniikki key property, jota React käyttää hyväkseen muun muassa renderöinnin suorituskykyoptimoinnissa.

Jokainen 'todo'-näkymä on div, joka sisältää tehtävän kuvauksen sekä valintaruudun. Kun 'todo'-näkymän valintaruutua klikataan, kutsutaan luokan 'handleTodoToggle'-me-

todia, jolle annetaan parametriksi klikatun ’todon’ id. Tämä metodi päivittää valitun ’todon’ ’done’-tilaa muuttamatta olemassa olevan tilan objektiviittauksia. Komponentin ’todos’-tila päivitetään polkukopioimalla komponentin tila ja muuttamalla tätä kopiota, jonka jälkeen komponentin ’todos’-tila vaihdetaan tähän kopioon.

Yksisuuntaisen dataflow’n merkitys hyvin hoidetussa tilanhallinnassa ei ole aivan itsestään selvää pienessä yhden komponentin kontekstissa. Seuraavissa luvuissa käsitellään, kuinka yksisuuntaista dataflow’ta sovelletaan näkymissä, jotka rakentuvat puumaisesti useammasta komponentista.

3.5 Komponentin parametrit (properties)

Properties (lyhyesti props) ovat komponenteille annettavia parametreja. Komponentit eivät omista omia propertyjään, joten komponentilla ei ole suoraa valtaa päivittää näitä muuttujia. Aikaisemmissa esimerkeissä input-elementeille annettiin muun muassa ’onChange’- ja ’value’-propertyt. Samalla tavoin mikä vain React-komponentti kykenee ottamaan vastaan propertyjä. Propertyjä ei voi ajatella konstruktoriparametreinä, sillä ne voivat vaihtua useasti komponentin elinkaaren aikana. Yleensä täysin toimiva ajattelumalli on, että komponentit ovat funktioita, jotka palauttavat näkymiä, ja propertyt ovat tämän funktion parametreja.

Kuvan 19 esimerkki on uudelleen määritys esimerkistä 17. Tässä esimerkissä input-elementti on korvattu MyInput-komponentilla.

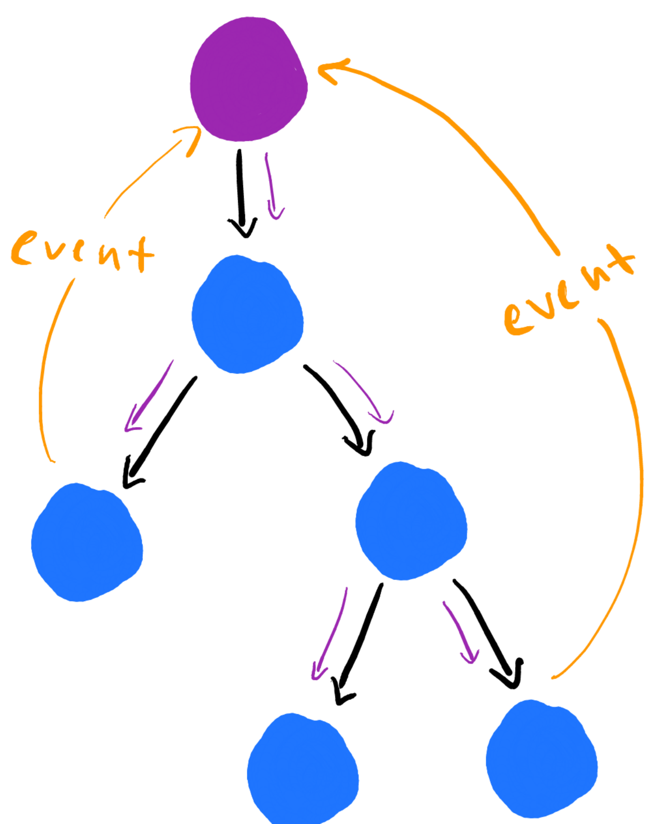
```
class LoginForm extends React.Component{
  state = { username: '', password: '' };
  render(){
    const {username, password} = this.state;
    return (
      <form onSubmit={this.handleSubmit}>
        <MyInput
          value={username}
          onChange={e => this.setState({username: e.target.value})}
          placeholder="username"
        />
        <MyInput
          type="password"
          value={password}
          onChange={e => this.setState({password: e.target.value})}
          placeholder="password"
        />
      </form>
    )
  }

  handleSubmit = (e) => {
    /*Handle sending of request*/
  }
}

class MyInput extends React.Component{
  render(){
    const {placeholder, type, value, onChange} = this.props;
    return (
      <input
        type={type}
        value={value}
        placeholder={placeholder}
        onChange={onChange}
      />
    )
  }
}
```

Kuva 19 Komponentin property-esimerkki

Kuvassa 19 LoginForm antaa MyInput-komponenteille propertyt ’placeholder’, ’type’, ’value’ ja ’onChange’. MyInput-komponentti antaa nämä omat propertynsä edelleen propertynä input-elementille. Kun input-elementin onChange callback-funktio tulee kutsutuksi näppäinpainalluksen yhteydessä, kutsuu input-elementti MyInput-komponentin sille antamaa onChange callback-funktio propertyä. Parametrina tälle funktiolle input-elementti antaa tapahtuman. Kutsutussa callback-funktiossa luetaan tapahtuman kohteen arvo

(event.target.value), ja se asetetaan LoginForm-komponentin tilamuuttujaan ’password’ tai ’username’, riippuen kumman kentän tapahtumasta on kyse. Tämän seurauksena LoginFormin tila päivittyy, josta seuraa, että LoginForm-komponentin render-metodi tulee uudelleen kutsutuksi. Tämän seurauksena tapahtuman kohteen MyInput saa uuden ’value’-propertyn, jonka seurauksen MyInput-komponentin render tulee kutsutuksi, jonka seurauksena input-elementti saa uuden ’value’-propertyn.

Hierarkkisesti rinnakkaiset komponentit ovat usein riippuvaisia samasta tilasta. Yleensä paras tapa hallita usean näkymän jakamaa yhteistä tilaa React-sovelluksessa on siirtää suurin osa tilasta sovelluksen juureen niin, että kaikki sovelluksessa jaettu tila elää yhdessä ainoassa tilaobjektissa. Käytännössä tämä tarkoittaa sitä, että suurin osa komponenteista ei sisällä lainkaan omaa komponentin sisäistä tilaa, vaan kaikki komponentteihin liittyvä tila välitetään komponenteille propertyinä. Noudattaen tätä periaatetta varmistutaan siitä, että jokainen komponentti sisältää tiedon sovelluksen nykyhetkestä, eikä tapauksia, joissa komponenttien tilat olisivat ristiriidassa keskenään, pääse syntymään.



Kuva 20 Sovelluksen yhteinen tila

Kuva 20 kuvaa, kuinka sovelluksen tila valuu yhdestä lähteestä kaikkiin sovelluksen komponentteihin. Tässä mallissa myös ylempänä olevat komponentit, jotka eivät välttämättä edes hyödynnä propertynä saamaansa tilaa, antavat saamansa propertyt edelleen usean komponenttikerroksen läpi lopulta tilaa hyödyntäville ja muuttaville lapsikomponenteilleen.

Tämä tapa välittää tilaa sisältää joitakin seuraavaksi mainittavia ongelmia, mistä johtuen edellisessä esimerkissä esitelty tilan valuttaminen ei ole suositeltua suuressa mittakaavassa.

1. Kun näkymäpuun koko kasvaa syväksi, tila- ja callback-funktiot, jotka muuttavat tilaa, joudutaan valuttamaan usean komponenttikerroksen lävitse kyseistä tilaa hyödyntäville lapsikomponenteille. Tämä voi tehdä koodista vaikeammin pääteltävää, sekä alttiiksi virheille, joissa jokin ylemmistä komponenteista jättää välittämättä tietyn alempana käytettävän propertyn lapsilleen.
2. Asettaa joissakin reunatapauksissa turhaa rasitetta suorituskyvylle (voi johtaa käyttöliittymän viiveisiin ja animaatioiden pätkimiseen).
3. Komponenteista tulee riippuvaisia siitä, missä kohdin sovelluspuuta niitä hyödynnetään. Tämä lähestymistapa rajoittaa komponenttien uudelleenkäytettävyyttä, sillä kaikkea sovelluksen tilaa ei haluta välittää jokaiseen komponenttihaaraan.

Joitakin näistä yllä kuvatuista ongelmia React pyrkii paikkaamaan esittelemällä käsitteen context.

3.6 Komponenttien konteksti (context)

3.6.1 Konteksti yleisesti

Reactin komponenteilla on hierarkiaa kuvaava käsite nimeltä konteksti. Jos komponentilla on lapsikomponentteja, voidaan sanoa, että kaikki komponentin jälkeläiset ovat kyseisen komponentin kontekstissa. Komponenteilla on myös ’context’-niminen muuttuja, joka muistuttaa komponentti-propertynä, mutta tämän muuttujan arvot välitetään kontekstidataa tarjoajan komponentin toimesta epäsuorasti kaikille sen jälkeläisille. Komponentti voi Reactin konteksti-rajapintaa hyödyntämällä tarjota kaikille jälkeläisilleen kontekstidataa ilman, että välikomponenttien tulisi eksplisiittisesti kuljettaa nämä muuttujat

dataa hyödyntäville komponenteille. Vastaavasti kontekstidataa vastaanottavat komponentit pystyvät hyödyntämää kaikkien esivanhempiensa tarjoamia kontekstidatoja. [7.]

Yksi tapa ajatella kontekstidataa, on verrata sitä funktion näkyvyysalueella sijaitsevaan muuttujaan.

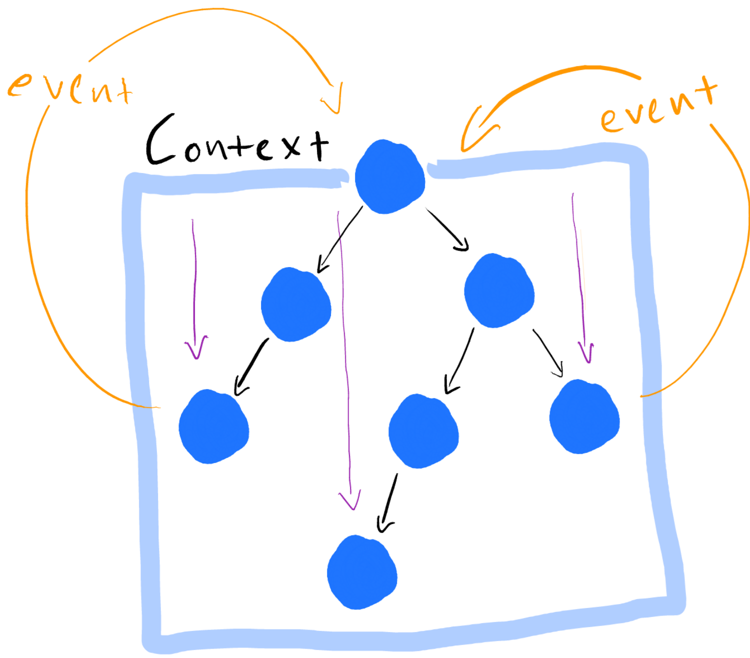
```
function App(){
  const context= {todos: {
    'a': {id: 'a', done: false, description: 'Do homework'},
    'b': {id: 'b', done: true, description: 'Wash dishes'}
  }};
  function Todos(){
    function TodoItem(props){
      const todo = context.todos[props.todoId];
      console.log(todo);
    }
    TodoItem({todoId: 'a'});
    TodoItem({todoId: 'b'});
  }
}
```

Kuva 21 React context-abstraktio

Esimerkissä 21 ei esitä oikeita komponentteja, vaan se pyrkii hahmottamaan konteksti-datan roolia komponenttihierarkiassa. Kuvassa App-funktio esittää konteksti-dataa tarjoavaa komponenttia, jolla on Todos-niminen lapsikomponentti. Todos esittää komponenttia, jolla on kaksi TodoItem-nimistä lapsikomponenttia, joita se kutsuu parametreilla, jotka esittävät propertyjä. TodoItemin saama 'props'-niminen parametri sisältää todoId-muuttujan. TodoItem hyödyntää saamansa 'todoId'-muuttujan arvoa viitatessaan näkyvyysalueensa 'context'-nimiseen muuttujaan, josta se valitsee todos-objektin, joka löytyy 'todoId'-muuttujan arvolla. Valittuaan 'todoId':llä löytyneen todon, tulostaa TodoItem kyseisen muuttujan arvon konsoliin.

3.6.2 Kontekstidatan määrittäminen

Kontekstidatan hyödyntäminen React-sovelluksissa tulee tarpeellisesi viimeistään siinä vaiheessa, kun sovellukseen luodaan useampia vaihtuvia kokonaisnäkymiä. Tilan nostamisella sovelluksen juurikontekstiin on se hyöty, ettei tilan tarvitse enää olla sidoksissa mihinkään näkymään, joka saattaisi poistua sovelluksen elinkaaren aikana.



Kuva 22 Komponenttien konteksti

Kuva 22 esittää, kuinka sovelluksen juurikontekstissa sijaitseva komponentti välittää kontekstidatana sovelluksen tilaa sekä tilaa muuttavia callback-funktioita kaikille lapsikomponenteilleen. Kun kontekstin tila päivittyy, päivittyvät myös kontekstissa sijaitsevat lapsikomponentit.

Jokainen luokkana määritetty komponentti pystyy tarjoamaan kontekstidataa jälkeläisilleen, mutta yleensä kontekstidataa tarjoava komponentti määritetään ainoastaan sovelluksen juuressa 'Provider'-komponentissa. Juuressa sijaitseva komponentti on myös kelvollinen käsittelemään asynkronisia pyyntöjä, koska tämän elinkaari kestää koko sovelluksen olemassaolon ajan.

Kontekstidataa tarjoavan komponentin tulee aina määrittää komponentti-kuvauksessaan, mitä dataa se tarjoaa jälkeläisilleen. Seuraavaksi esimerkissä esitellään, kuinka kontekstidataa tarjoava komponentti käytännössä määritetään.

```
class Provider extends React.Component {
  static childContextTypes = {
    posts: object,
    onAddPost: func,
  };

  state = {
    posts: {},
  };

  getChildContext() {
    const { posts, } = this.state;
    return {
      posts,
      onAddPost: this.onAddPost,
    };
  }

  render() {
    return this.props.children;
  }

  onAddPost = ({ title, text, created, }) => {
    const { posts, } = this.state;
    const id = uuid();
    this.setState({
      posts: { ...posts, [id]: { id, title, text, created, }, },
    });
  };
}
```

Kuva 23 Provider-esimerkki

Kuvan 23 Provider-komponentissa on määritetty luokkamuuttuja childContextTypes. Tässä muuttujassa määritetään, minkä nimisiä sekä tyyppisiä muuttujia komponentti tarjoilee lapsilleen. Tämän lisäksi Provider-komponentissa on määritetty instanssimetodi 'getChildContext', joka tulee kutsutuksi aina silloin, kun joku kontekstidataa hyödyntävistä lapsikomponenteista päivittyy.

Provider-komponentin render-funktio palauttaa komponentin erityisasemassa olevan propertyn 'children'. Tämä property sisältää kaikki komponentit, jotka on määritetty komponentin alku- ja lopputagien sisälle.
Seuraavassa kuvassa on käytännön esimerkki siitä, miten komponentin 'children' property määritetään.

```
const App = () => (
  <Provider>
    <Header/>
    <PostsPage/>
    <Footer/>
  </Provider>
);
```

Kuva 24 Alku- ja lopputageina määritetty komponentti

Kuvassa 24 on Provider-komponentti määritetty erillisinä alku- ja lopputageina. Tästä seuraa, että Provider saa propertyn, joka sisältää listana tämän kontekstuaaliset lapset: Header, PostsPage ja Footer. Koska Provider ei tunne lapsikomponenttejaan, ei se kykene välittämään näille propertyjä, mutta sen sijaan se on kykenevä toimimaan näiden komponenttien kontekstidatan tarjoajana.

Varoituksena liittyen kuvan 23 esimerkkiin täytyy mainita, että Reactin context-API:n hyödyntäminen sisältää joitakin sudenkuoppia, joista kerrotaan lisää myöhemmin.

3.6.3 Kontekstidatan hyödyntäminen

Kontekstidataa käyttävän komponentin tulee kuvauksessaan eksplisiittisesti määrittää, mistä kontekstin muuttujista se on kiinnostunut, jotta sen instanssit kykenevät hyödyntämään kyseisiä kontekstimuuttujia.

Seuraava esimerkki demonstroi, kuinka kontekstia on mahdollista hyödyntää komponentissa.

```
class PostsPage extends React.Component {
  static contextTypes = {
    onAddPost: func,
    posts: object
  };

  render() {
    const {title, text} = this.state;
    const {posts} = this.context;
    return (<div className="posts-page">
        <h1>Posts</h1>
        {Object.values(posts)
          .map(post => (
            <div>
              <h3>{post.title}</h3>
              <p>{post.text}</p>
              <p>{formatDate(post.created)}</p>
            </div>))}
        <div className="add-post">
          <h2>Add Post</h2>
          <input
            className="title"
            value={title}
            onChange={this.handleChange('title')}/>
          <textarea
            className="text"
            value={text}
            onChange={this.handleChange('text')}/>
          <button onClick={this.handleSubmit}> Submit</button>
        </div>
      </div>
    )
  }

  handleChange(key) {
    return e => this.setState({[key]: e.target.value})
  }
  handleSubmit() {
    this.setState({title: '', text: ''});
    this.context.onAddPost({title, text, created: new Date()})
  }
}
```

POSTS

SUNT SPECIESES REPERIRE TALIS, AUDAX TORQUISES.

When one discovers booda-hood and career, one is able to yearn stigma.

2017 09 12

ADD POST

SUBMIT

Kuva 25 Kontekstimuuttujien hyödyntäminen komponentissa

Kuvassa 25 PostsPage-komponentti tilaa kontekstidatasta muuttujat ’onAddPost’ ja ’posts’ esimerkissä 23 määritetystä Provider-komponentista.

Kontekstimuuttujat tilataan määrittämällä luokkamuuttuja 'contextTypes', joka mahdollistaa, että komponentilla on pääsy kyseisiin kontekstimuuttujiin. Komponentti viittaa kontekstimuuttujiinsa muuttujan context kautta. Kun komponentin konteksti dataa tarjoava vanhempi päivittyy, päivittyy myös PostsPage-komponentti uusine kontekstimuuttujineen.

Kontekstia hyödyntävän komponentin näkökulmasta ei kontekstin hyödyntäminen muilta osin merkittävästi poikkea propsien hyödyntämisestä.

Kontekstin hyödyntäminen sovelluksen juuritilana sisältää joitakin keskeisiä ongelmia, mistä johtuen kontekstia hyvin harvoin käytetään sellaisenaan.

Kontekstin ongelmia:

1. Jokin kontekstissa oleva komponentti voi estää oman uudelleen renderöintinsä, mikä johtaa siihen, etteivät tämän komponentin lapsikomponentit päivity kontekstimuuttujien päivittyessä. Yleinen syy tällaiselle käytökselle on, että komponentti pyrkii optimoimaan suorituskykyä. Ilman suorituskykyoptimointia kontekstin päivittyessä päivittyvät kaikki kontekstissa sijaitsevat näkyvät riippumatta siitä, riippuvatko näiden näkymät millään tavoin muuttuneesta kontekstin tilasta.
2. Sovelluksen kasvaessa kontekstissa elävä tilanhallinta voi käydä hyvin monimutkaiseksi ylläpitää.

Jokapäiväisessä kehityksessä kontekstin tilaa käytetään harvoin suoraan ilman apukirjastoja. Johtuen muun muassa näistä ylläolevista kontekstin ongelmista Reactin dokumentaatiossa varoitetaan, että konteksti-API saattaa muuttua merkittävästi tulevissa julkaisuissa. Sen sijaan hiemankaan tilaltaan kompleksisen sovelluksen kanssa suositellaan käytettäväksi erillisiä tilanhallintakirjastoja kuten Reduxia tai Mob-X:ää.

4 Tilanhallinta käyttäen Redux-tilanhallintakirjastoa

4.1 Yleisesti

Redux on yleinen funktionaalinen JavaScript-tilanhallintakirjasto, jota on mahdollista hyödyntää muidenkin kuin React-sovellusten kanssa. React ja Redux toimivat hyvin samassa sovelluksessa, sillä Redux-kirjastoa käytettäessä sovelletaan tilanhallinnassa yksisuuntaista dataflow'ta ja tarkkailijamallia.

Redux pohjautuu osittain aikaisempaan Facebookin kehittämään Flux-arkkitehtuuriin, jota hyödynnettiin vielä joitakin vuosia sitten yleisesti useimmissa React-sovelluksissa.

Kaikki Reduxin määrittämä tila sijaitsee objektissa, jota kutsutaan storeksi. React-komponentit voivat tilata storen tilamuutokset komponenttikohtaisesti, sekä muuttaa storen tilaa lähettämällä Redux storelle viestejä. [8.]

Redux ohjaa hyvin vahvasti, kuinka sovelluksen tilaa tulee hallita. Sen tilanhallintamenetelmät ovat jossain määrin päällekkäisiä Reactin sisäisten tilanhallintamekanismien kanssa. Suuri osa tällä hetkellä toteutettavista React-sovelluksissa käyttää hyväkseen Reduxia tilanhallinnassaan.

4.2 Redux store

Reduxia käytettäessä kaikkea yleistä sovelluksen tilaa hallinnoidaan Redux storen kautta. Sovellus voi sisältää vain yhden storen, jonka kolme yleisimmin käytettyä funktiota ovat:

1. getState-funktio palauttaa tämänhetkisen storen tilan.
2. subscribe-funktiolle annetaan parametriksi callback-funktio, jota Redux store kutsuu joka kerta, kun storen tila muuttuu. Subscribe-kutsu palauttaa callback-funktion, jota kutsumalla tilaus perutaan.
3. dispatch-funktio lähettää viestin, joka muuttaa storen tilaa. Se ottaa parametrinaan vastaan objektin, joka sisältää kuvauksen siitä, miten storen tilaa halutaan päivittää. Näitä storelle lähetettäviä objekteja kutsutaan actioneiksi

[9.]

Reduxia käyttäessä Reactin kanssa storen tilamuutokset tilataan komponenttikohtaisesti. Jos komponentti on kiinnostunut vain tietystä storen alitilasta, voi se välttää komponentin uudelleen päivittymisen tilanteissa, jossa sitä kiinnostava alitila ei ole muuttunut. Storen tilaaminen komponenttikohtaisesti tuo mukanaan sen edun, että komponentit eivät ole riippuvaisia vanhempansa, eivätkä kontekstinsa päivittymisestä saadakseen viimeisimmän sovelluksen tilan.

Koska komponentin observoivat itsenäisesti storen tilamuutoksia, mahdollistaa tämä turvallisen tavan suorituskykyoptimoinnille.

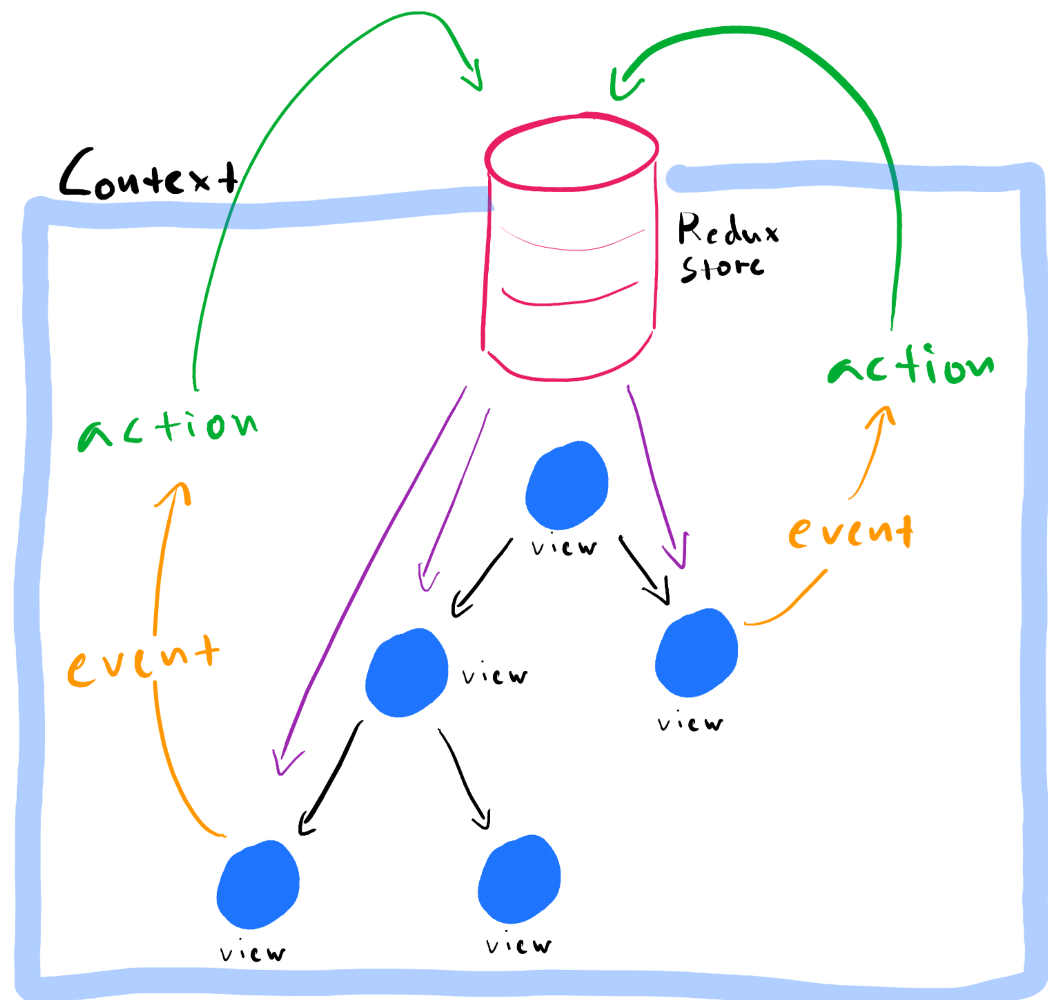
```
store.subscribe(() => {
  const todos = store.getState().todos;
  if(todos!==this.state.todos){
    this.setState({todos});
  }
});
```

Kuva 26 Storen tilamuutosten tilaus

Kuvassa 26 demonstroidaan, kuinka muutokset storen tilassa voidaan tilata. Tässä esimerkissä komponentti asettaa storen alitilan 'todos' omaan komponentin sisäiseen tilaansa aina, kun kyseinen storen alitila on muuttunut. Koska Redux storen tilaa muutetaan aina polkukopioimalla vanha tila, voidaan komponenttia kiinnostava alitila todeta muuttuneeksi vertailemalla edellisen ja nykyisen tilan objektiviittauksia, eikä syvää objektien vertailua tarvitse suorittaa.

Tämän kaltaista suorituskykyoptimointia harvoin tarvitsee toteuttaa itse, sillä React-sovelluksissa, joissa käytetään Reduxia, käytetään usein react-redux-lisäkirjastoa, joka antaa muutamia apufunktioita, jotka hoitavat nämä optimoinnit automaattisesti.

Käyttäessä storen tilaa sovelluksen tilana dataflow'n periaatteet toimivat jossain määrin samalla lailla kuin käyttäessä juurikomponentin kontekstintilaa.



Kuva 27 Reduxin rooli komponentti hierarkiassa

Kuvassa 27 kuvataan Redux storen roolia komponenttihierarkiassa. Vertailemalla tätä esimerkkiin 22 voidaan huomata, että React-sovelluksissa storen luonnollinen paikka komponenttihierarkiassa on toimia kontekstidatan tarjoajana. Tästä syystä storen toiminnallisuudet yleensä asetetaan komponenttihierarkian juureen context-Provider-komponenttiin.

4.3 Reduserit (Redux reducers) ja Actionit

Storen tilaa muutetaan käyttäen storen dispatch-funktiota, jolle annetaan parametriksi action-objekti. Actionin tulee sisältää vähintään tyyppi (type) ja mahdollisesti muita parametreja. Kun store ottaa vastaan uuden actionin, luo se uuden tilan, ja ilmoittaa observoijilleen, että tila on muuttunut. Storen tila sekä muutosrajapinnat määritetään luomalla

sovelluksen alustusvaiheessa tapahtumankäsittelijäfunktioita, joita kutsutaan redusereiksi. Reduserit voivat alustusvaiheessa määrittää näiden vastaamien alitilojen alkuarvot, sekä ne toimivat rajapintoina storen tilan muuttamiselle.

```
store.dispatch({type: 'TOGGLE_TODO', payload: {id: 'a'}});
```

Kuva 28 dispatch-esimerkki

Kuvan 28 esimerkissä storelle lähetetään action-tyyppiä ’TOGGLE_TODO’, joka saa lisäparametrinaan ’payload’-muuttujan, joka sisältää muuttujan ’id’ arvolla ’a’. Jotta actionit kykenevät muuttamaan storen tilaa, täytyy kyseisen action-tyypin käsittely määrittää storen redusereissa. [10.]

Reduserit ovat sivuvaikutuksettomia, puhtaita funktioita jotka palauttavat aina saman arvon samoilla parametreilla (pure function [11.]). Ne ottavat parametreina vastaan edellisen tilansa sekä action-objektin. Alustusvaiheessa reduserille täytyy määrittää myös alkutila. Reduserit sisältävät yleisesti switch case-lauseen, jonka perusteella reduser päättelee, kuinka tilaa muutetaan, kun sille annetaan tiettyä tyyppiä oleva action. Jos action-tyyppi liittyy kyseiseen reduseriin, luo se uuden tilan. Muussa tapauksessa reduser palauttaa edellisen tilansa. [12.]

Joskus usea reduser saattaa reagoida samaan action-tyyppiin luomalla uuden alitilan. Jokaisen lähetetyn actionin yhteydessä kaikki reduserit käydään läpi rekursiivisesti ja näin muodostetaan seuraava storen tila.

```
import {createStore, combineReducers} from 'redux';

const initialState = {};
function todos(state = initialState, action){
  switch (action.type){
    case 'ADD_TODO':{
      const {id, description} = action.payload;
      const todo = {id, description, done: false};
      const nextState = {...state}; // shallow copy
      nextState[id] = todo;
      return nextState;
    }
    case 'REMOVE_TODO': {
      const {id} = action.payload;
      const nextState = {...state}; //shallow copy
      delete nextState[id];
      return nextState;
    }
    case 'TOGGLE_TODO': {
      const {id} = action.payload;
      const nextState = {...state}; // shallow copy
      const todo = {...nextState[id]}; // shallow copy
      todo.done = !todo.done;
      nextState[id] = todo;
      return nextState;
    }
    default: return state;
  }
}
const rootReducer = combineReducers({todos});
const store = createStore(rootReducer);
console.log(store.getState()); // {todos: {}}
store.dispatch({type: 'ADD_TODO', payload: {id: 'a', description: 'Do homework'}})
console.log(store.getState()); // {todos: {a: {id: 'a', description: 'Do homework', done: false}}}
```

Kuva 29 Reduser-esimerkki

Kuvan 29 esimerkissä alustetaan Redux store yhdellä reduserilla, joka käsittelee kolmea action-tyyppiä ’ADD_TODO’, ’REMOVE_TODO’ sekä ’TOGGLE_TODO’. Koska reduseri ei saa alustusvaiheessaan palauttaa tyhjää arvoa, on sille annettu oletusarvoksi tyhjä objekti.

Funktio combineReducer ottaa parametrinaan vastaan objektin, joka sisältää redusereita. Sen tehtävä on yhdistää useita redudereita yhdeksi reduseriksi niin, että reduseiden tilat löytyvät niille määritetyillä objektin avaimia. Koska combineReducers luo yhden reduserin useasta reduserista, voidaan combineReducers-funktiolla määrittää hierarkkisia reduseri rakenteita. Yleensä hierarkkisten reduseri rakenteiden tekeminen ei silti ole tarpeellista.

Lopulta, kun kaikki reduserit on yhdistetty yhdeksi päälreduseriksi, luodaan store kutsumalla createStore funktiota, joka ottaa parametrinaan juuri reduserin. Esimerkki-storen alkutila on täten objekti, joka sisältää objektin ’todos’. Kun storen dispatch-funktiolle annetaan ’ADD_TODO’ action, luo store uuden tilan, joka sisältää yhden todon.

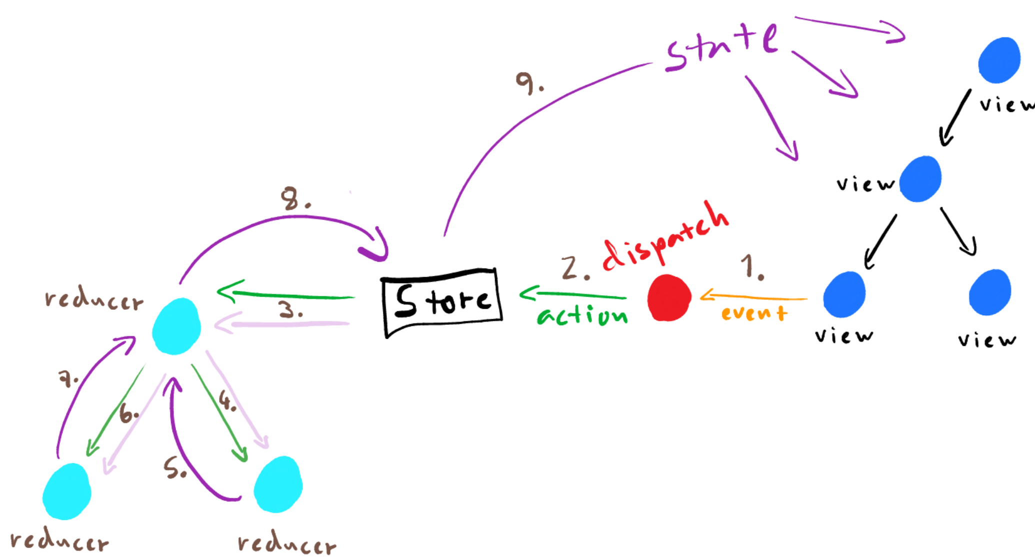
Redux-redusereiden tila pyritään aina pitämään normalisoituna. Ensimmäinen syy normalisoidun tilan suosimiselle on suorituskyky. Toinen syy on, että tila halutaan pystyä tallentamaan sekä lähettämään tekstimuodossa.

```
const todosState = {
    a: {id: 'a', description: 'Do homework', done: false},
    b: {id: 'b', description: 'Wash dishes', done: true},
    c: {id: 'c', description: 'Buy milk', done: true}
};
```

Kuva 30 Normalisoitu tila

Kuvassa 30 on demonstroitu muoto, jossa redusereiden tila pyritään pitämään. Normalisoidussa tilassa ei voi olla kehäviittauksia, muuttujat voivat olla vain yksinkertaisia objekteja, listoja, string-, boolean- tai numeroarvoja. Myös listojen käyttöä pyritään välttämään, jotta muutokset tilaan olisivat mahdollisimman nopeita.

Seuraavassa esimerkissä Reduxin dataflow on kuvattuna yleisellä tasolla. [13.]



Kuva 31 Reduxin dataflow

Kuvassa 31 on kuvattu askel askeleelta, mitä tapahtuu, kun storelle lähetetään action. Tapahtumat kuten input-kenttään kirjoittaminen tai hiiren painallukset saavat aikaan tapahtumia, joiden seurauksena lähetetään action.

Kun action lähetetään, saapuu se storen käsiteltäväksi. Kun store saa actionin käsiteltäväkseen, kutsuu se juuri reduser-funktiota antaen tälle parametriksi tämän hetkisen storen tilan sekä lähetetyn actionin. Tämän jälkeen juuri reduser kutsuu ali-redusereitaan näiden edellisillä kerroilla palauttamilla tiloilla sekä julkaistulla actionillä. Kun action on kulkenut kaikkien redusereiden läpi, juuri reduser palauttaa redusereiden rakentaman uuden tilan. Store asettaa juuri reduserin palauttaman tilan omaan tilaansa, jonka jälkeen se ilmoittaa kaikille observoijilleen, että tila on muuttunut.

4.4 Redux middlewaret

Redux middlewaret ovat pääosin vastuussa kaikista Redux storen sivuvaikutuksista. Middlewaret ovat funktioita, jotka voivat pysäyttää actionin, välittää actionin seuraavalle middlewarelle tai reagoida luomalla ja julkaisemalla uusia actioneita.
Jos storelle on määritetty middlewareja, päätyy action aina ensin middewareille ennen sen saapumista storen käsiteltäväksi.

Sopivia käyttötapauksia middlewareille ovat esimerkiksi api-kutsujen käsittely, ajastetut tapahtumat, actioneiden pysäyttäminen, poikkeusten käsittely ja actioneiden tallentaminen muistiin myöhempää käyttöä varten esimerkiksi undo/redo toiminnallisuuden toteuttamiseksi. [14.]

Store alustetaan middlewareilla antamalla middlewaret toisena parametrina createStore-funktiolle.

```
import {createStore, combineReducers, applyMiddleware} from 'redux';

/*
 * other definitions
 * */

const createTodosMiddleware = store => nextMiddleware => action => {
  nextMiddleware(action);
  if(action.type === 'FETCH_TODOS'){
    store.dispatch({type: 'DISPLAY_SPINNER'});
    api.fetchTodos()
      .then(data => store.dispatch({
        type: 'SET_TODOS', payload: data
      }))
      .catch(data => store.dispatch({
        type: 'DISPLAY_ALERT', payload: 'FETCH_TODOS_ERROR'
      }));
  }
};
const todosMiddleware = applyMiddleware(createTodosMiddleware);
const rootReducer = combineReducers({todos, spinner, notification});
const store = createStore(rootReducer, todosMiddleware);
```

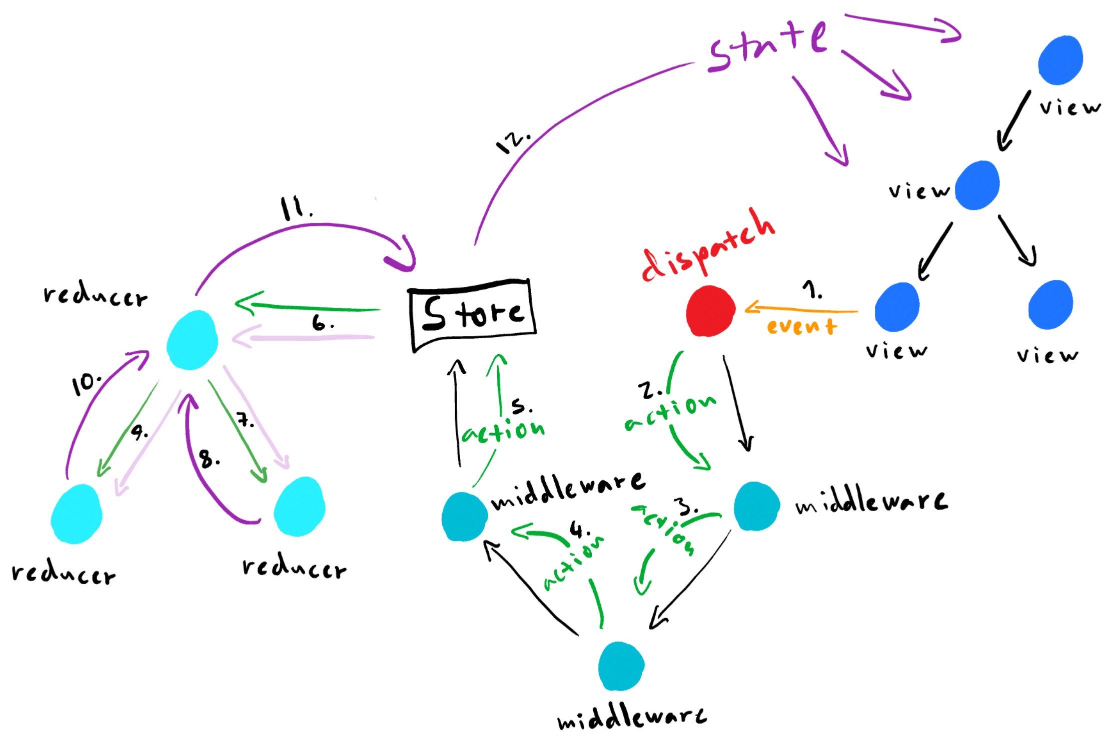
Kuva 32 Storen luonti middlewarella

Esimerkissä 32 todosMiddleware luodaan määrittämällä 'createTodosMiddleware'-funktio. Middleware alustetaan storen käytettäväksi antamalla middlewaren luojafunktio applyMiddleware-funktiolle. Middlewaren luova funktio on niin kutsuttu tehdasfunktio tai korkeamman tason funktio (*Higher-order function* [15.]). Se muodostetaan määrittämällä funktio, joka saa parametrinaan vastaan storen. Tämä funktio palauttaa funktion, joka ottaa vastaan seuraavan middlewaren. Tämä viimeinen funktio palauttaa middleware-funktion, joka ottaa vastaan lähetettyjä actioneita ja kutsuu seuraavaa middlewarea kyseisellä actionilla. Esimerkin todosMiddleware kuuntelee action tyyppiä 'FETCH_TODOS'. Kun 'FETCH_TODOS' action julkaistaan, reagoi middleware julkaisemalla 'DISPLAY_SPINNER'-actionin. Tämän jälkeen middleware lähettää pyynnön palvelimelle. Kun pyyntö saa vastauksen, tai se heittää poikkeuksen, lähettää middleware uuden actionin tilan päivittämiseksi.

Syy, miksi storen middlewaren luonti on suhteellisen kompleksinen, johtuu siitä, kuinka applyMiddleware valjastaa middewaret storen käytettäviksi.

Aina, kun koodista kutsutaan storen dispatch-funktiota, kutsutaan itseasiassa storen ensimmäistä middlewarea kyseisellä actionilla. Tässä middleware-ketjussa storen alkuperäinen dispatch-kutsu on viimeinen middleware. Jokainen uusi middleware siis ylikirjoittaa storeen liitetyn dispatch-funktion.

Seuraavassa esimerkissä kuvataan, kuinka middlewaret muuttavat Redux storen dataflow'ta.



Kuva 33 Reduxin dataflow middlewarejen kanssa

Esimerkki 33 on jalostettu versio esimerkistä 31, jossa esiteltiin Redux storen dataflow'ta. Tässä kuvauksessa storelle on annettu 3 middlewarea. Storen dispatch-kutsu välittää actionin ensimmäiselle storen middlewarelle, joka vuorostaan välittää sen seuraavalle, ja niin edelleen, kunnes lopulta viimeinen middleware antaa actionin viimeiselle middlewarelle, joka on storen alkuperäinen dispatch-funktio.

4.5 Yhteenveto Reduxista

React ja Redux yhdessä, lisäosineen, sisältävät paljon opeteltavaa. Lisäksi Redux lisää sovelluksen ylläpidettävää koodia merkittävästi. Pienemmissä sovelluksissa Reduxin hyödyt suhteessa sen vaatimaan työ- ja tietomäärään (*mental overhead*) eivät välttämättä ole aivan kiistattomia. Toisaalta se on erinomainen kirjasto ratkomaan vaikeita tilanhallinnollisia ongelmia.

Flux-arkkitehtuuriin pohjautuvat tilanhallintakirjastot ovat olleet *de facto*-tapa hoitaa tilanhallintaa React-sovelluksissa vuosien ajan. Ainut jossain määrin Flux-arkkitehtuurin kanssa kilpaileva tilanhallintatilanhallintakirjasto on kirjasto nimeltä MobX. MobX on reaktiivinen JavaScript-kirjasto, jonka etuna suhteessa Reduxiin on erityisesti se, että se on nopeasti omaksuttava eikä se vaadi yhtä paljon lisäkoodia kuin Redux.

MobX ei oman dokumentaationsa mukaan ole sovelluksen tilanhallintaan tarkoitettu kirjasto, mutta silti sitä käytetään ratkomaan samankaltaisia ongelmia kuin redux, react-redux, redux-thunk ohjelmistopinolla.

MobX:n pääpiirteiset erot suhteessa Reduxiin:

- ei tilan normalisointia
- vähemmän opeteltavaa
- vähemmän *boilerplate*-koodia
- antaa enemmän vapauksia tilanhallinnan suhteen (*less opinionaded*)
- paljon näkymätöntä automatiikkaa
- useita storeja
- objekti orientoitunut ohjelmointityyli.

MobX-tilanhallinta sopii erityisen hyvin tiimeihin, joilla ei ole aikaisempaa kokemusta Reduxin käytöstä, sekä kehittäjille, joiden tausta on objekti orientoituneessa ohjelmointityylissä, sekä sovelluksiin, joissa tilan kompleksisuus ei koskaan kasva hyvin suureksi. [16.]

5 Tilanhallinta hyödyntäen JavaScript Proxyjä (react-proxy-state)

5.1 Motivaatio

Tämän opinnäytetyön aikana tein tilanhallintakirjaston nimeltä react-proxy-state. Alkuperäinen pyrkimykseni oli löytää hyviä ratkaisuja, joita hyödyntämällä Redux-koodiin liittyvää koodia olisi mahdollista vähentää sekä selkeyttää. Näiden tavoitteiden saavuttaminen tuntui lopulta ristiriitaiselta, sillä Reduxin yksi suurimmista vahvuuksista on koodin eksplisiittisyys ja läpinäkyvyys. Koodin määrän vähentäminen olisi myös automaattisesti vähentänyt koodin läpinäkyvyyttä, joten päädyin toteuttamaan todistekonseptitilanhallintakirjaston, joka on Reduxista riippumaton React-spesifinen tilanhallintakirjasto. Kirjaston pääprioriteetit ovat koodimäärän vähentäminen, helppo omaksuttavuus, suorituskyky sekä helppo testattavuus. Näitä prioriteetteja kirjasto tavoittelee käyttämällä JavaScript kielen Proxyjä sekä Reactin itsessään määrittelemiä tilanhallinnan käsitteitä, kuten komponenttien kontekstia, jonka käyttöä Reactin virallisessa dokumentaatiossa varoitetaan käyttämästä. Ennen kaikkea kirjasto ottaa mallia aikaisemmista Redux-ekosysteemin kirjastoista.

5.2 JavaScript Proxy käsite

Proxyt ovat JavaScript-kielen ES6-standardissa esitelty uusi käsite, joka mahdollista uudenlaisia reflektio [17.] toimenpiteitä. Proxyja käyttämällä objektien ja funktioiden tiettyjä metaoperaatioita, sekä luku- että kirjoitustapahtumia on mahdollista tarkkailla sekä kontrolloida.

Proxyjen määrittäminen jokapäiväisessä koodissa ei ole mielekästä, ja niitä käyttämällä on mahdollista tehdä hyvin monimutkaisia ja huonoja ratkaisuja. Proxyjen etuna on, että ne mahdollistavat JavaScript-kirjastojen toteuttamisen, joiden toteuttaminen ei aikaisemmin JavaScript-kielellä olisi ollut mahdollista.

Seuraavassa esimerkissä nähdään, kuinka kirjoitusoperaatio on mahdollista ehdollistaa käärimällä objekti Proxyyn.

```
function rangeObject(min, max) {
  return new Proxy({}, {
    set(target, property, value) {
      if (Number.isInteger(value)) {
        if (value >= min && value <= max) {
          target[property] = value;
        } else {
          console.warn(value, 'is out of range');
        }
        return true;
      }
      return false;
    }
  });
}

const zeroToFive = rangeObject(0, 5);
zeroToFive.a = 1;
zeroToFive.b = 6;
zeroToFive.c = 5;
console.log(zeroToFive); // { a: 1, c: 5 }
```

Kuva 34 Proxy-esimerkki

Esimerkissä 34 luodaan funktiolla 'rangeObject' uusi Proxy. Tämä Proxy saa konstruktoriparametriksi objektin, jota Proxy tulee edustamaan sekä objektin, joka sisältää callback-funktioita, joita kutsutaan nimellä 'Proxy **traps'** [18.].
Esimerkissä Proxylle määritetään ainoastaan 'set' trap.

Proxyn 'set' trap kontrolloi kaikkia sijoitusoperaatioita, joita Proxyyn itseensä kohdistetaan. Kun sijoitettava arvo on esimerkki funktiossa määritettyjen min- ja max-arvojen sisäpuolella, kohdistaa Proxy sijoituksen tämän edustamaan objektiin. Kun sijoitettava arvo on min- ja max-arvojen ulkopuolella, jättää Proxy sijoitusoperaation suorittamatta, ja tulostaa varoituksen konsoliin. Jos sijoitettava arvo on väärän tyyppinen, palautetaan boolean false, jonka seurauksena syntyy ajonaikainen poikkeus.

5.3 Vaikutteet

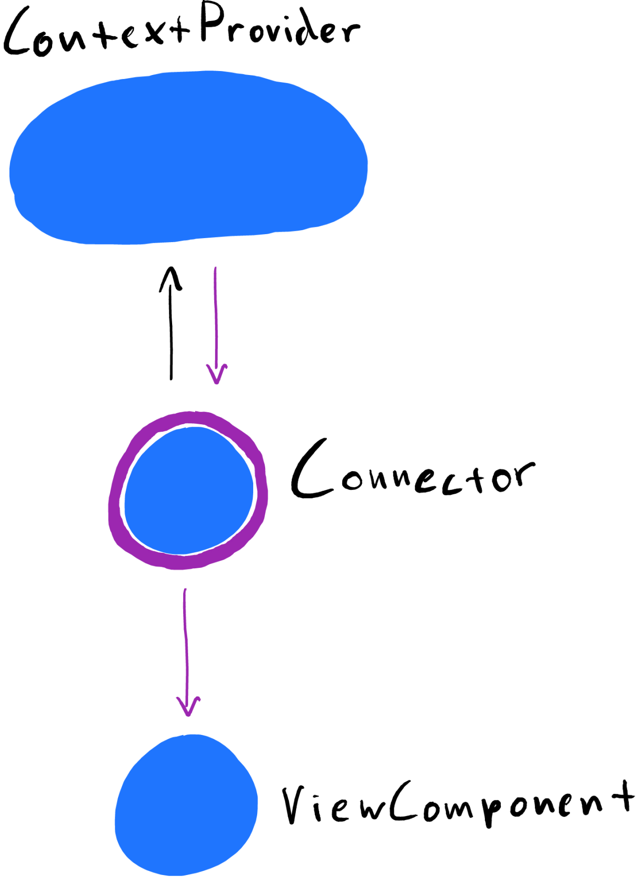
Opinnäytetyössä tehty react-proxy-state kirjasto on ottanut paljon vaikutteita redux, react-redux [19.] sekä redux-thunk-kirjastoista [20.]. React-proxy-state-kirjaston käyttö näyttää hyvin tutulta ohjelmistokehittäjille, jotka tuntevat nämä kirjastot.

Nämä kolme Redux-kirjastoa ovat usein React-sovelluksissa tilanhallinnan peruspilareita. Yhdessä Reactin kanssa nämä kirjastot ovat erillisistä paloista koottu ekosysteemi, joiden opettelu voi olla pitkä prosessi ohjelmistokehittäjälle. Verrattuna redux-ekosysteemiin react-proxy-state on pragmaattinen, helposti omaksuttava React-spesifinen tilanhallintakirjasto.

5.4 Ratkaisut

5.4.1 Kontekstitilan asettaminen propseihin

Suuri osa React-komponenttien tilasta on sellaista, joka halutaan olevan käytettävissä myös näkymävaihdosten välillä sekä jaettavissa useiden komponenttien välillä. Yleinen tapa on, että tila säilytetään kaikkien komponenttien kontekstissa. Naiivit kontekstitilariippuvaiset näkymät vaativat koko sovellusnäkymän päivittymisen jokaisen kontekstitilavaihdoksen välillä, joten näiden komponenttien suorituskykyoptimointi ei ole mahdollista. Tätä varten komponenttien tulee tilata kontekstin tilan muutokset komponenttikohtaisesti, joka taas tuo tarpeen manuaaliselle suorituskykyoptimoinnille. Samalla tavoin kuin react-redux, myös react-proxy-state mahdollistaa kontekstin tilan tilaamisen automatisoinnin luomalla niin kutsutun korkeamman abstraktion komponentin (*Higher-Order Component* [21.]), joka tilaa kontekstin tilan komponentin puolesta, sekä välittää tämän tilan tilaa käyttävälle komponentille propertynä.



Kuva 35 Higher-Order Connector Component

Kuvassa 35 ContextProvider mahdollistaa kontekstin tilanmuutosten tilaamisen. Connector-komponentti tilaa kontekstin tilanmuutokset ja asettaa kontekstin tilan omaan komponenttitilaansa. Tämän jälkeen Connector-komponentti päivittyy ja antaa ViewComponent-komponentille oman tilansa propertynä.

Yleensä näkymäkomponentit ovat kiinnostuneita vain tietystä kontekstitilan aliosasta. Tätä varten Connector-komponentti määritetään 'mapContextToProps'-funktiolla, joka luo komponenttia varten kustomoidun Connector-komponentin, joka antaa näkymäkomponentille vain tätä kiinnostavat kontekstin tilat propertynä.

```
import {mapContextToProps} from 'react-proxy-state';

class TodosApp extends Component {
  render() {
    return (
      <div>
        {Object.values(this.props.todos).map(todo => (
          <div key={todo.id}>
            <p>{todo.description}</p>
            <input type="checkbox" value={todo.done}/>
          </div>
        ))}
      </div>
    );
  }
}
const selector (contextState) => {
  return {todos: contextState.todos};
}
const createConnected = mapContextToProps(selector);
const ConnectedTodosApp = createConnected(TodosApp);
```

Kuva 36 mapContextToProps

Kuvassa 36 luodaan komponentti TodosApp, joka saa propseina 'todos'-tilan, ja näyttää 'todo'-rivien kuvaukset ja statukset. TodosApp-komponentin määrityksen jälkeen luodaan 'createConnected'-funktio kutsumalla mapContextToProps-funktiota. mapContextToProps ottaa parametrinaan vastaan 'selector'-funktion, joka valitsee kontekstin tilasta komponentille propertyna annettavat alitilat. Esimerkissä kontekstin tilasta valitaan 'todos'-alitila. Kun createConnected-funktiota kutsutaan, luo se komposition sille parametrina annetusta komponentista sekä Connected-nimisestä komponentista.

Muiden komponenttien ei siis tule käyttää TodosApp-komponenttia suoraan, vaan sen sijaan niiden tulee hyödyntää ConnectedTodosApp-komponenttia. ConnectedTodosApp Connector-komponentti pitää huolen myös siitä, että TodosApp-komponentti ei turhaan päivity, jos sitä kiinnostava alitila ei muutu. Tämän lisäksi Connector-komponentti välittää aina saamansa propertyt edelleen TodosApp-komponentille, joten ConnectedTodosApp-komponenttia hyödyntävien komponenttien ei tarvitse huomioida sitä, että kyseinen komponentti on kompositio kahdesta komponentista.

Usein komponentti tarvitsee vain pienen osan kontekstin tilaa propertikseen, mutta se, minkä kontekstin tilan komponentti tarvitsee, voi riippua siitä, mitä se saa vanhemmaltaan propertynä. Tätä varten mapContextToPropsin vastaanottama funktio saa toisena parametrina komponentille tulevat propertyt.

Seuraava esimerkki on siistitty versio kuvasta 36, ja se demonstroi samanaikaisesti sitä, kuinka propertyjen hyödyntämistä kontekstin valinnassa on mahdollista toteuttaa komponentin mapContextToProps ’selector’-funktiossa.

```
import {mapContextToProps} from 'react-proxy-state';
import TodoItem from './TodoItem';

class TodosApp extends Component {
  render() {
    const {ids} = this.props;
    return (
      <div>
        {ids.map(id => (<TodoItem key={id} id={id}/>))}
      </div>
    );
  }
}

const {keys} = Object;
export default mapContextToProps(({todos}) => ({ids: keys(todos)}))(TodosApp);
```

```
const TodoItem = ({description, done}) => (
  <div>
    <p>{description}</p>
    <input type='checkbox' checked={done} />
  </div>
);

export default mapContextToProps(({todos}, {id}) => todos[id])(TodoItem);
```

Kuva 37 Kontekstitilan valinta propsien perusteella

Kuvassa 37 on kuvan 36 toiminnallisuus uudelleen määritetty käyttäen TodoItem-komponenttia. Tällä kertaa TodosApp valitsee kontekstista propertykseen ainoastaan ’todos’ avaimet, eli id:t. Funktion mapContextToProps vastaanottama funktio ottaa toisena parametrinaan komponentin saamat propertyt. TodoItem käyttää mapContextToProps-kutsussaan propertyään ’id’ löytääkseen kontekstitilasta sille määrätyn alitilan.

Tässä yhteydessä tämä esimerkki ei suorituskyvyn kannalta ole paras valinta, mutta se pyrkii demonstroimaan sitä, kuinka saatuja propertyjä voidaan hyväksikäyttää kontekstitilan valinnassa. Lisätietoja mapContextToProps funktion käytöstä liitteestä 1.

5.4.2 Kontekstitilan muuttaminen

Jotta kontekstitilan muuttaminen on kerrottavissa ymmärrettävästi, täytyy ensin käydä läpi, kuinka ContextProvider-komponentti määritetään tapahtumankäsittelijöineen.

ContextProvider-komponentti luodaan funktiolla, joka ottaa parametrinaan vastaan sovelluksen alkutilan sekä objektin, joka sisältää tapahtumakäsittelijöiden funktiot. Tapahtumankäsittelijäfunktiot ovat tehdasfunktioita, joiden yksinkertaistetun version ContextProvider tarjoilee kontekstistaan kaikille komponenteille.

```
import {createProvider} from 'react-proxy-state';

const eventHandlers = {/*none*/}
const initialState = {todos: {}};
const ContextProvider = createProvider(initialState, eventHandlers);

const Root = () => (
  <ContextProvider>
    <App/>
  </ContextProvider>
);
```

Kuva 38 ContextProviderin alustus

Kuvassa 38 määritetään ContextProvider-komponentti createProvider-funktiolla. Ensimmäisenä parametrina createProviderille annetaan alkutila, joka on objekti, joka sisältää tyhjän todos-objektin. Toisena parametrina funktiolle annetaan tapahtumankäsittelijöiksi tyhjä objekti. Tämä tarkoittaa sitä, ettei yhtään tapahtumankäsittelijää määritetä. Lisätietoja ContextProvider-komponentin rajapinnoista löytyy liitteestä 2.

Tapahtumankäsittelijät ovat korkeamman tason funktioita, jotka muistuttavat hieman Reduxin middlewareja. Tapahtumankäsittelijät palauttavat funktion, jolle annetaan konteksti tilan proxy-tietorakenne silloin, kun komponentti kutsuu kyseistä tapahtumankäsittelijää.

```
const changeTodo = (id, property, value) => (proxy) => {
  proxy.todos[id].assign({[property]: value});
};
```

Kuva 39 Yleinen todo-tapahtumankäsittelijä

Kuvassa 39 on määritetty yleinen todo-tapahtumankäsittelijä, joka ottaa parametrinaan todon id:n, muutettavan propertyn nimen, ja uuden arvon. Kun tapahtumankäsittelijää kutsutaan, kutsutaan funktion palauttamaa funktiota tilan Proxylla.

Tapahtumankäsittelijöille annettava Proxy on ikään kuin varjotietorakenne todellisesta sovelluksen tilasta. Tämä varjotietorakenne rakentuu solmuista, joista kukin vastaa tiettyä osaa sovelluksen tilaa. Tapahtumankäsittelijöille annettava Proxy on tämän tietorakenteen juurisolmu. Täten solmujen kautta on mahdollista viitata kaikkialle sovelluksen alitiloihin.

Tapahtumankäsittelijöissä on mahdollista viitata sovelluksen tilaan viittaamalla solmujen state-muuttujaan, joka on dynaamisesti päätelty arvo kyseisen solmun vastaamasta sovelluksen alitilasta sillä hetkellä, kun state-muuttujaan viitataan.

Solmujen kautta tilaa on mahdollista muuttaa, neljällä eri funktiolla.

- clear-funktio ottaa parametrinaan vastaan minkä vain arvon. Kun tätä metodia kutsutaan, muutetaan solmun sijainnissa oleva alitila vastaamaan parametrina annettua arvoa.
- assign-funktio ottaa parametrinaan vastaan objektin, joka pinnallisesti yhdistetään olemassa olevan alitilan kanssa.
- remove-funktio ottaa parametrinaan vastaan useita String-muuttujia ja poistaa kyseisillä avaimella löytyvät alitilat.
- toggle-funktio muuttaa solmun vastaaman alitilan nykyisen tilan negaatioksi. Funktion tuloksena kyseisen tilan arvo muuttaa joko Boolean **true-** tai **false-**arvoksi.

Kutsumalla näitä solmujen metodeita kaikki muutokset, joita tilaan kohdistetaan, suoritetaan automaattisesti polkukopioimalla. Näin tila pysyy muuttumattomana, joten Reactin suorituskykyoptimointi on helppo automatisoida Connected-komponenttien toimesta.

Muuttumaton tila tuo myös mukanaan helpommin ennakoitavan kokonaistilan hallinnan, koska tilan muutoksilla (vaihdoksilla) ei ole sivuvaikutuksia.

Seuraavassa esimerkissä tarkastellaan, kuinka sovelluksen tilaa yleisesti päivitetään, jos käytössä ei ole mitään tilanhallinnan kirjastoa. Usein nämä ratkaisut sisältävät paljon toisteista koodia, jonka tarkoitus on ainoastaan vastata siitä, ettei olemassa olevan tilan objektiviittauksia muuteta, vaan tilan muutos suoritetaan polkukopioimalla.

```
const addTodo = (description) => (currentState) => {
    let {todos} = currentState;
    const id = uuid();
    const todo = {id, description, done: false};
    todos = {...todos, [id]: todo};
    return {...currentState, todos}
};
```

Kuva 40 addTodon manuaalinen tilan kopiointi

Kuvan 40 esimerkki demonstroi perinteisillä JavaScriptin mahdollistamilla polkukopiointimenetelmillä toteutettua 'addTodo'-tapahtumankäsittelijää. Tapahtumankäsittelijä saa tapahtumalta parametrinaan vastaan 'description'-tekstin. Kun tapahtumankäsittelijää kutsutaan, antaa Provider sille kontekstin tilan. Funktion ensimmäisellä rivillä tilasta luetaan alitila todos vastaavan nimiseen muuttujaan. Toisella rivillä luodaan uusi todo. Kolmannella rivillä vanha todos-tila kopioidaan ja siihen lisätään juuri luotu uusi todo. Tämän jälkeen kokonaistila kopioidaan, jonka yhteydessä vanha todos-arvo korvataan uudella todos-objektilla. Tämä uusi kokonaistila palautetaan kutsuvalle osapuolelle, joka asettaisi tämän tilan nykyiseksi sovelluksen tilaksi.

Suuremmissa sovelluksissa tila on usein monta kerrosta syvempi, ja imperatiivista tilan polkukopiointia joudutaan tekemään usean kerroksen lävitse. Tämän takia on oleellista, että tilan muuttaminen olisi helppoa ja nopeaa, sillä tilan muutoksia hoitava koodi saattaa muuten kattaa helposti merkittävän osan koko sovelluksen lähdekoodista.

react-proxy-staten tapahtumankäsittelijöiden solmut automatisoivat tämän saman polkukopionnin, ja näin vähentävät virheiden mahdollisuutta, sekä testattavan sekä ylläpidettävän koodin määrää. Seuraavassa esimerkissä suoritetaan vastaava operaatio käyttäen react-proxy-staten tapahtumankäsittelijöitä.

```
const addTodo = (description) => ({todos}) => {
    const id = uuid();
    todos.assign({[id]: {id, description, done: false}});
};
```

Kuva 41 addTodo-tilan päivittäminen käyttäen solmuja.

Kuvassa 41 on suoritettu kuvan 40 toimenpide käyttäen react-proxy-staten toiminnallisuuksia. Tapahtumankäsittelijä ’addTodo’ saa tapahtumalta parametrina ’description’-tekstin. Kun funktiota kutsutaan, ContextProvider antaa tapahtumankäsittelijän palauttamalle funktiolle tilan juuritilasta vastaavan solmun. Funktiokutsun parametrissa tilasta luetaan muuttuja ’todos’. Kun ’todos’-muuttuja luetaan, juurisolmusta palautetaan ’todos’-tilasta vastaava solmu. Jos kyseistä tilaa vastaavaa solmua ei ole vielä luotu, luodaan se tässä yhteydessä automaattisesti solmun Proxyn toimesta olettaen, että kyseinen tila on olemassa.

Ensimmäisellä funktion rivillä määritetään uuden todon id, ja toisella rivillä kutsutaan solmun ’assign’-funktiota, jolle annetaan parametrina ’todo’-objekti. Tämän seurauksena tila muutetaan polkukopioimalla, ja ContextProvider saa viestin siitä, että tila on muuttunut.

ContextProvider-komponentti välittää rikastetut tapahtumankäsittelijät kontekstimuuttujina kaikille komponenteille. Koska tapahtumankäsittelijät eivät koskaan muutu sovelluksen elinkaaren aikana, on tapahtumankäsittelijöiden tarjoaminen kontekstin kautta turvallista. Tämän lisäksi kontekstin hyödyntäminen poistaa tarpeen viittauksilta tiedostoihin, joissa tapahtumankäsittelijät sijaitsevat, ja täten voidaan vähentää koodin määrää sekä helpottaa testatessa komponentin tapahtumankäsittelijöiden jäljittelyä.

Seuraavassa esimerkissä esitellään, kuinka ContextProvider määritetään tapahtumankäsittelijöineen.

```
import {createProvider} from 'react-proxy-state';

const addTodo = description => ({todos}) => {
  const id = uuid();
  todos.assign({[id]: {id, description, done: false}});
};

const toggleTodo = id => ({todos}) => todos[id].done.toggle();
const removeTodo = id => ({todos}) => todos.remove(id);
const removeAllTodos = () => ({todos}) => todos.clear({});

const eventHandlers = {addTodo, toggleTodo, removeTodo, removeAllTodos};

const initialState = {
  todos: {a: {id: 'a', description: 'Finish thesis', done: false}}
};

const ContextProvider = createProvider(initialState, eventHandlers);

const Root = () => (
  <ContextProvider>
    <TodosApp/>
  </ContextProvider>
);
```

Kuva 42 ContextProviderin luonti

Kuvassa 42 rivillä 1 haetaan createProvider-funktio react-proxy-state-moduulista. Seuraavilla riveillä määritetään tapahtumankäsitteliät 'addTodo', 'toggleTodo', 'removeTodo' sekä 'removeAllTodos'. Tämän jälkeen määritetään sovelluksen alkutila 'initialState'-muuttuja, joka saa alkuarvokseen objektin todos, joka sisältää yhden todon. Alkutilan määrityksen jälkeen luodaan ContextProvider-komponentti antamalla createProvider-funktiolle alkutila sekä objekti, joka sisältää tapahtumankäsittelijät. Lopuksi ContextProvider määritetään JSX-syntaksilla sovelluksen juureen niin, että sovelluksen tila sekä tapahtumankäsittelijät ovat käytettävissä App-komponentissa sekä rekursiivisesti kaikissa App-komponentin lapsikomponenteissa.

Jotta komponentin olisi mahdollista hyödyntää kontekstin tapahtumankäsittelijöitä, tulee komponentin määrittää komponentin 'contextTypes'-muuttuja, joka määrittää, mitkä kontekstimuuttujat komponentille annetaan käytettäviksi.

```
import {mapContextToProps} from 'react-proxy-state';
import TodoItem from './TodoItem';

class TodosApp extends Component {

  static contextTypes = {
    addTodo: func,
  };

  state = {description: ''};

  render() {
    const {state: {description}, props: {ids}} = this;
    return (
      <div>
        <input value={description} onChange={e => this.setState({description: e.target.value})}/>
        <button onClick={this.onAddTodo}>Add todo</button>
        {ids.map(id => (<TodoItem key={id} id={id}/>))}
      </div>
    );
  }

  onAddTodo = () => {
    const {description} = this.state;
    this.setState({description: ''});
    this.context.addTodo(description);
  }
}

const {keys} = Object;
export default mapContextToProps(({todos}) => ({ids: keys(todos)}))(TodosApp);
```

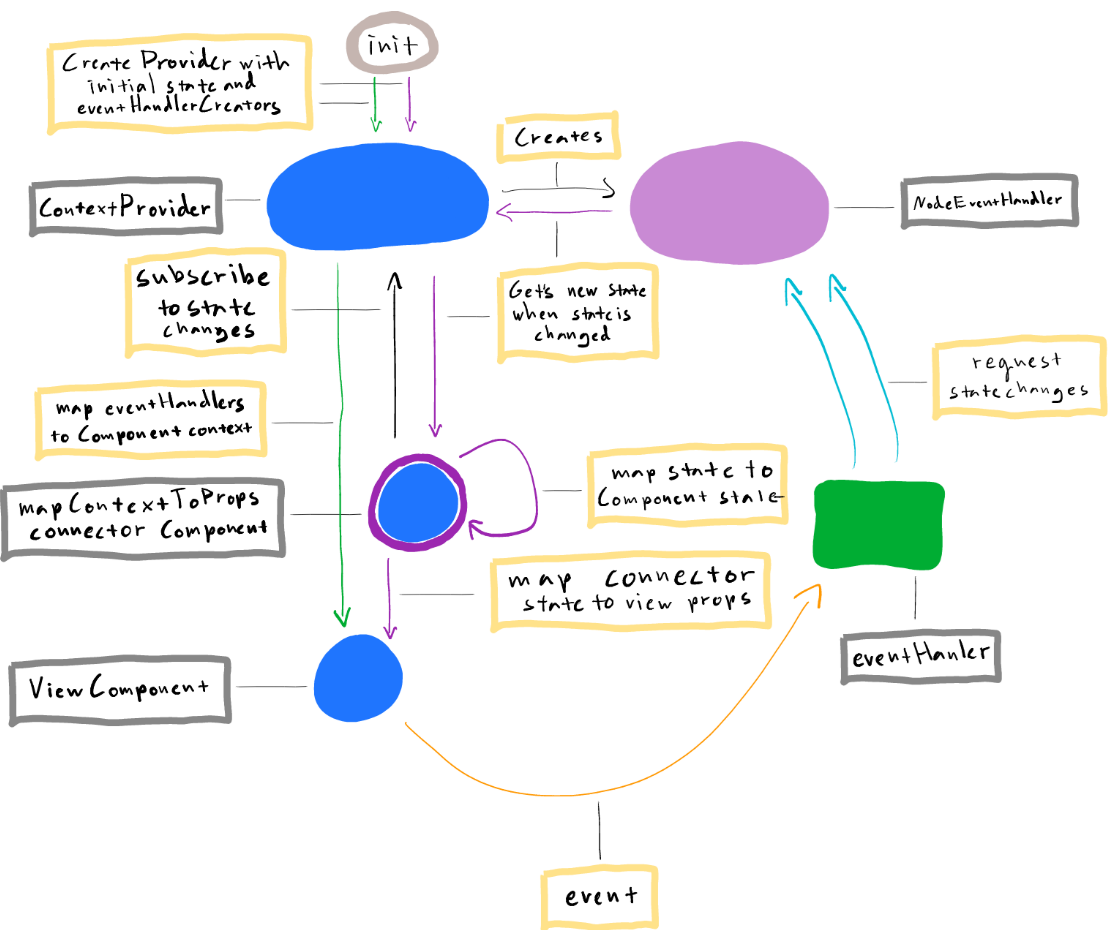
```
import {mapContextToProps} from 'react-proxy-state';

const TodoItem = ({id, description, done}, {toggleTodo, removeTodo}) => (
  <div>
    <p>{description}</p>
    <input type='checkbox' checked={done} onChange={() => toggleTodo(id)}/>
    <button onClick={() => removeTodo(id)}>remove</button>
  </div>
);

TodoItem.contextTypes = {
  removeTodo: func,
  toggleTodo: func
};

export default mapContextToProps(({todos}, {id}) => todos[id])(TodoItem);
```

Kuva 43 Kontekstin tapahtumankäsittelijöiden hyödyntäminen komponenteissa

Kuvassa 43 on kuvan 37 toteutukseen lisätty toiminnallisuus muuttaa kontekstin ’todos’-tilaa. Komponentit tilaavat tapahtumankäsittelijät määrittämällä komponenttikohtaisen contextType-muuttujan, joka sisältää tapahtumankäsittelijöiden nimet sekä muuttujatyypit. TodosApp-komponentti on määritetty luokkana, joten contextTypes määritetään staattisena luokkamuuttujana. TodosApp valitsee kontekstista tapahtumankäsittelijän ’addTodo’. Komponentissa on määritetty ’onAddTodo’-instanssifunktio, jota kutsutaan,

kun 'Add todo'-nappia painetaan. Kun 'Add todo'-nappi tulee painetuksi, tästä seuraa, että luokan 'onAddTodo' tulee kutsutuksi. Tämä funktio kutsuu kontekstin 'addTodo'-tapahtumankäsittelijää, joka määritettiin kuvassa 41. Tämä tapahtumankäsittelijä ottaa parametrinaan vastaan 'descriptionin'-tekstin ja luo kontekstin tilaan uuden 'todon', kyseisellä kuvauksella. 'TodoItem'-komponentti valitsee kontekstista tapahtumankäsittelijät 'toggleTodon' sekä 'removeTodon'. Koska 'TodoItem' on määritetty funktiona, sen contextTypes määritetään asettamalla contextTypes-muuttuja tähän funktioon. Funktiokomponentit saavat kontekstimuuttujansa toisena parametrinaan renderöinnin yhteydessä. Kun 'TodoItemin' 'done' valintaruutua painetaan, kutsuu tämä input-kenttä callback-funktiota, joka kutsuu kontekstin 'toggleTodo'-tapahtumankäsittelijää antaen tälle kyseisen todon 'id':n. 'TodoItemin' 'remove'-nappi on määritetty samalla tavoin kuin 'toggleTodo'-nappi, mutta se kutsuu painettaessa 'removeTodo'-tapahtumankäsittelijää. Lisä tietoa react-proxy-state tapahtumankäsittelijöistä ja tilasolmuista löytyy liitteestä 3.

Seuraavassa esimerkissä on jäsennetty react-proxy-staten kokonais-flow.



Kuva 44 react-proxy-state flow

Kohdassa init luodaan ContextProvider-funktiolla createProvider, jolle annetaan parametriksi sovelluksen alkutila sekä tapahtumankäsittelijät. ContextProvider luo NodeEventHandler-muuttujan (solmutilankäsittelijä) sekä tilaa siltä tältä tiedon jokaisesta tilan muutoksesta. Esimerkissä on vain yksi näkymä. Kun näkymä luodaan käyttäen mapContextToProps-funktiota, luodaan Connetor-komponentti, joka toimii näkymäkomponentin vanhempana. Tämä komponentti tilaa kontekstista tilanmuutokset ja päivittää oman tilansa vastaamaan selector-funktiossa määritettyä alitilaa.

Aina kun NodeEventHandler ilmoittaa ContextProviderille, että tila on muuttunut, välittää ContextProvider ilmoituksen kaikille Connector-komponenteille. Tämän seurauksena Connector-komponentti päivittää state-muutujansa, josta seuraa, että Connector-komponentti päivittyy. Kun Connetor päivittyy, saa ViewComponent tuoreimman kontekstitilan propertynä Connector-komponentilta.

ViewComponent määrittää contextTypes-määrityksessään, mistä tapahtumankäsittelijöistä se on kiinnostunut. Kun komponentti kutsuu kontekstin tapahtumankäsittelijöitä, annetaan tapahtumankäsittelijän palauttamalle funktiolle parametriksi sovelluksen juuritilasta vastaava solmu. Tämä solmu lähettää NodeEventHandlerille pyyntöjä muuttaa tilaa.

Kun tila on muuttunut, välittää NodeEventHandler ContextProviderille tiedon, että tila on muuttunut, jonka seurauksena ContextProvider kertoo Connector-komponenteille muuttuneesta tilasta, jonka seurauksena näkymät päivittyvät.

5.4.3 Proxy-solmujen toteutus

Toteutuksen prosessoinnin kannalta raskaimpia sekä monimutkaisimpia vaiheita toteuttaa oli solmujen logiikka. Tässä kappaleessa käydään hyvin lyhyesti sekä yksinkertaistetusti lävitse, kuinka solmut toimivat.

Tila säilytetään aina tietorakenteen juurisolmussa. Jokainen viittaus tilaan luetaan dynaamisesti tästä samasta muuttujasta. Kun solmun tilaan (state-muuttujaan) viitataan, solmu kysyy tietorakenteen juurelta nykyistä sovelluksen tilaa. Käyttäen tietona omaa

sijaintiaan kokonaistietorakenteessa solmu päättelee oman alitilansa, jonka se palauttaa state-viittauksen yhteydessä.

```
get state() {
  const location = getLocation(this);
  const state = rootNode.state;
  const subState = findState(state, location);
  return subState;
}
```

Kuva 46 state getteri

Esimerkki 46 selittää korkealla tasolla, mitä tapahtuu, kun solmun tilaan viitataan. Ensin solmu selvittää oman sijaintinsa tietorakenteessa. Seuraavaksi luetaan sovelluksen juuritila, jota säilytetään aina juurisolmussa. Tämän jälkeen solmun itsensä edustama alitila etsitään kokonaistilasta.

Solmun tilamuutosmetodit, kuten assign, ovat funktioita, jotka lähettävät viestejä NodeEventHandlerille muuttaa tämä sovelluksen tilaa viestin sisällön mukaisesti, jonka jälkeen se antaa uuden tilan tietorakenteen juurelle.

```
assign(param) { // {d: 4}
  const location = getLocation(this); // ['b', 'c']
  sendRequest('ASSIGN', identity, param);
}
```

```
const eventHandler(request, location, param) => {
  const {state} = rootNode; // { a: 1, b: {c: {}} }
  const nextState = pathCopy(state, location);
  switch (request) {
    case 'ASSIGN:
      const subState = findState(nextState, location); // {}
      Object.assign(subState, param); // {d: 4}
      break;
    .... /*other event handlesr*/
  }
  rootNode.state = nextState; // { a: 1, b: {c: {d: 4}} }
};
```

Kuva 46 Tilaa muuttavat operaatiot

Esimerkissä 46 on yksinkertaistettu esimerkki siitä, kuinka solmujen assign-operaatio toimii. Esimerkin ylemmässä kuvassa on solmun metodi assign, joka saa parametrinaan objektin. Kun metodi suoritetaan, solmu selvittää ensin oman sijaintinsa, jonka jälkeen

se lähettää serialisoituvan pyynnön NodeEventHandlerille. Viesti sisältää tiedon parametrina annetusta muuttujasta, tapahtuman tyypistä sekä tapahtuman kohteesta.

Kun viesti tulee NodeEventHandlerille, polkukopioi tämä nykyisen tilan objektin juuresta muutoskohteeseen saakka samalla luoden vanhasta tilasta johdetun uuden tilan. Tämän jälkeen muutoskohteen alitila ja parametrin arvo yhdistetään nykyiseen alitilan kopioon. Lopuksi juurisolmulle annetaan uusi tila.

5.4.4 Rajoitteet

Redux- ja MobX-ekosysteemit tarjoavat paljon kehitystyökaluja, jotka helpottavat testaamista sekä virheiden löytämistä.

Kirjaston Proxy-toteutus sisältää joitakin epäjohdonmukaisuuksia, kun tilassa säilytetään taulukkoja tai tilaan kohdistetaan suoria sijoitusoperaatioita. Proxyn hallinnoimassa tilassa ei myöskään ole mahdollista säilyttää muuta kuin JavaScriptin yleisimpiä natiivityyppejä kuten Object, Array, Number, String jne. Todennäköisesi vastaan tulee reunatapauksia, joita ei vielä ole osattu ottaa huomioon.

JavaScript-kielen Proxy-toteutuksessa jää paljon toivomisen varaa. React-proxy-state-kirjaston näkökulmasta hyödylliset puuttuvat Proxy-ominaisuudet liittyvät operaattoreiden ylikirjoittamiseen (*operator overloading* [22.]) sekä roskien keruun observointiin. JavaScript-kielestä puuttuu myös määrite ’heikko viittaus’ (*weak reference* [23.]), jota hyödyntämällä kirjaston toteutuksesta voisi saada tehokkaamman sekä yksinkertaisemman. Kirjaston suorituskyky on silti yllättävän hyvä, eikä sen luomien abstraktioiden pitäisi näkyä pullonkaulana sovelluksen suorituskyvyssä.

6 Pohdinta

Reactin-tilanhallinta Reduxin kanssa tuntuu usein ylisuunnitellulta ja liian jaarittelevalta (**verbose**). Reactin komponenttitila taas on suoraviivainen sekä intuitiivisempi tapa lähteä määrittämään tilaa, ja lopulta tilaa on ripustettu usealla tasolle komponenttien hierarkiaa. Ongelmat komponenttipuussa elävän tilan kanssa tulevat usein vastaan, kun sovelluksen toiminallisuusvaatimukset vähänkin kasvavat. Tässä vaiheessa tulee yleensä tarve nostaa komponenttipuussa useassa kerroksessa sijaitseva tila keskitetysti yhteen tai kahteen komponenttiin, jotka sijaitsevat korkealla komponenttipuussa. Tällöin herää kysymys, olisiko tila kannattanut suoraan suunnitella ylläpidettäväksi kontekstista käsin, esimerkiksi Redux-kirjaston konteksti-tilassa. Viimeistään siinä vaiheessa, kun sovelluksen eri reiteissä sijaitsevat näkymät alkavat hyödyntämään samaa dataa, on tila järkevintä nostaa sovelluksen juuri kontekstiin, joka pysyy aktiivisena ja käytettävissä koko sovelluksen elinkaaren ajan. Tämän jälkeen komponenttien state-muuttujiin jää enää tiloja, jotka vastaavat yleensä vain pienistä tilapäisistä tiloista.

Reactin tilanhallinnasta tuntuu puuttuvan *sweet spot,* jossa tilan hallinta on luotettavaa, nopeaa ja helppoa. Jos tilan hallinta on hetken aikaa tosi helppoa ja suoraviivaista, niin viidentoista minuutin päästä koodi pitää refaktoroida kaavaksi, josta voisi kirjottaa romaanin. Keskitetystä kontekstista suoritettu tilanhallinta tuo yleensä mukanaan helpon testattavuuden sekä voimakkaan varmuuden siitä, että sovelluksen tila vastaa ennakoitua. Sivutuotteena keskitetysti hallinnoitu tila tuo yleensä mukanaan myös paljon lisää koodia, varsinkin jos käytetään Redux-kirjastoa. Reduxin kanssa sovelluksen pääasiallinen tila on helppo pitää ennakoitavana sekä oikean muotoisena. Sen sijaan ongelmakohtia Redux-tilanhallinnassa ovat metatilojen käsittelyt. Tilan metatilat ovat tiloja, jotka kuvaavat tilan statuksia kuten: Onko serveriltä haku kesken, onko tila tyhjä, onko tallennus kesken, onko tilan tallennuksessa tai haussa tapahtunut virheitä, onko tila muokattavissa, onko tila validissa muodossa. jne.

Myös pienet asynkroniset yksityiskohdat, kuten viiveiden määrittäminen, viivästettyjen tapahtumien peruminen, animaatioiden käynnistäminen ja sammuttaminen, ovat virhealttiita toimenpiteitä, joiden hallintaan Redux ei itsessään ole kovin kykenevä kirjasto.

Mitä käyttäjäystävällisempi sovellukseen halutaan tehdä, sitä enemmän tilaa joudutaan yleensä rikastamaan tällaisilla metatiloilla. Tehdessä react-proxy-state-kirjastoa pyrin toteuttamaan jotakin, joka sopisi hyvin tähän välimaastoon, jossa supersuunniteltua tilan

hallintaa ei tarvita, mutta tilan hallinnan halutaan selviävän 99 % vastaan tulevista ongelmista. Lopputuloksena ymmärsin, että MobX-tilanhallintakirjasto on tehty tätä tarveryhmää varten.

7 Yhteenveto

Tämän opinnäytetyön tavoitteena oli esitellä sekä tutustua React-kirjastolla toteutettujen sovelluksen tilanhallintaan sekä pyrkiä paremmin ymmärtämään yksisuuntaiseen dataflow'iin liittyviä tilanhallinnan ongelmia. Tämän lisäksi työssä tutustuttiin Redux-kirjastoon, joka tällä hetkellä varsinkin web-kehityksessä on Reactin de facto -tilanhallintakirjasto. Lopuksi käytiin läpi react-proxy-state -tilanhallintakirjastosta, joka toteutettiin tämän opinnäytetyön ohella.

Kokonaisuudessaan Reactin tilanhallintaan syvällinen tutustuminen oli hyvin mielenkiintoinen ja opettavainen prosessi. Syitä sille miksi, nykyisen parhaat käytännöt React-sovelluksen tilanhallinnassa ovat sellaisia kuin ne tällä hetkellä ovat, selkeytyivät monella tapaa.

React-proxy-state -tilanhallintakirjaston toteutukseen jäi vielä paljon parannettavan varaa, ja sen ylläpito ja jatkokehitys jatkuu vielä harrasteprojektina.

Lähteet

1. Neary, Adam. Unidirectional DataFlow? Yes. Flux? I Am Not So Sure. Verkkoaineisto: <https://medium.com/@AdamRNeary/unidirectional-data-flow-yes-flux-i-am-not-so-sure-b4acf988196c>. 17.5.2015. Luettu 1.8.2017.
2. Wikipedia. Callback (Computer programming). Verkkoaineisto: <https://en.wikipedia.org/wiki/Callback_(computer_programming)>. 9.1.2018. Luettu 1.9.2017.
3. ReactiveX. Documentation. Verkkoaineisto: <http://reactivex.io/documentation>. Luettu 1.6.2017.
4. React. Introducing JSX. Verkkoaineisto: <https://reactjs.org/docs/introducing-jsx.html>. Luettu 1.12.2017.
5. React. Components and Props. Verkkoaineisto: <https://reactjs.org/docs/components-and-props.html>. Luettu 1.12.2017.
6. React. State and Lifecycle. Verkkoaineosto: <https://reactjs.org/docs/state-and-lifecycle.html>. Luettu 1.12.2017.
7. React. Context. Verkkoaineisto: <https://reactjs.org/docs/context.html>. Luettu 1.12.2017.
8. Redux. Synopsis. Verkkoaineisto: <https://github.com/reactjs/redux/blob/master/README.md>. 18.12.2017. Luettu 1.7.2017.
9. Redux. Store. Verkkoaineisto: <https://github.com/reactjs/redux/blob/master/docs/basics/Store.md>. 24.8.2017. Luettu 1.7.2017.
10. Redux. Actions. Verkkoaineisto: <https://github.com/reactjs/redux/blob/master/docs/basics/Actions.md>. 15.10.2017. Luettu 1.7.2017.
11. Pure function. Verkkoaineisto: <https://en.wikipedia.org/wiki/Pure_function>. 28.12.2017. Luettu 1.6.2016.
12. Redux. Reducers. Verkkoaineisto: <https://github.com/reactjs/redux/blob/master/docs/basics/Reducers.md>. 3.1.2018. Luettu 1.6.2017.
13. Redux. Data Flow. Verkkoaineisto: <https://github.com/reactjs/redux/blob/master/docs/basics/DataFlow.md>. 25.10.2017. Luettu 1.6.2017.
14. Redux. Middlewares. Verkkoaineisto: <https://github.com/reactjs/redux/blob/master/docs/advanced/Middleware.md>. 16.11.2017. Luettu 1.6.2017.
15. Wikipedia. Higher-order function. Verkkoaineisto: <https://en.wikipedia.org/wiki/Higher-order_function>. 10.1.2018. Luettu 1.9.2017

16. Robin Wieruch. Redux or MobX: An attempt to dissolve the Confusion. Verkkoaineisto: <https://www.robinwieruch.de/redux-mobx-confusion>. 29.3.2017. Luettu 1.7.2017.
17. Wikipedia. Reflection. Verkkoaineisto: <https://en.wikipedia.org/wiki/Reflection_(computer_programming)>. 14.1.2018. Luettu 1.9.2018.
18. Mozilla. Proxy. Verkkoaineisto: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy>. Luettu 1.8.2017.
19. React. React-redux. Verkkoaineisto: <https://github.com/reactjs/react-redux>. 5.4.2017. Luettu 1.6.2017.
20. Daniel Lythin. Redux Thunk. Verkkoaineisto: <https://github.com/gaearon/redux-thunk>. 10.5.2016. Luettu 1.6.2017.
21. React. Higher-Order Components. Verkkoaineisto: <https://reactjs.org/docs/higher-order-components.html>. Luettu 1.12.2017.
22. Wikipedia. Operator overloading. Verkkoaineisto: <https://en.wikipedia.org/wiki/Operator_overloading>. 10.1.2018. Luettu 1.9.2017.
23. Wikipedia. Weak reference. Verkkoaineisto: <https://en.wikipedia.org/wiki/Weak_reference>. 21.6.2017. Luettu 1.7.2017.

Liite 1

React-proxy-state Map Context State To Props API-kuvaus

Map Context State To Props

Components using context state, should be defined by using ***mapContextToProps***. mapContextToProps helps creating a higher-order *Connected*-component that wraps the *actual component*. mapContextToProps *subscribes* the context state, on behave of the *actual component*. Every time the context state changes, *Connector* passes the context state as properties to the *actual component*.

```
import {mapContextToProps} from 'react-proxy-state';

const Todo = ({description, done}) => (
    <div>
      <p>{description}</p>
      <p>{done ? 'Done' : 'Not done'}</p>
    </div>
)

const selector = (contextState, ownProps) => {
  const todo = contextState.todos[ownProps.todoId];
  return todo;
}
const createConnected = mapContextToProps(selector);
export default createConnected(Todo);
```

mapContextToProps is a ***higher order function*** that takes a state **selector** as parameter. *mapContextToProps returns a ***createConnected*** function that takes the *actual component* as parameter.

selector is a function that takes contexts state and component own properties as parameters.

Everytime* Components properties or context state changes, this selector functions is re-run, and what ever it returns, gets passed as property by *Connector* to the component passed to ***createConnected***

(if the output changes compared to the previous output)*.

ks. https://github.com/jEnbuska/react-proxy-state

Liite 2
React-Proxy-State ContextProvider API-kuvaus

ContextProvider

Context state is served by ***ContextProvider***-component. ContextProvider-component is created by ***createProvider*** function, that takes the initialState as first argument and eventHandlers as second argument. Both initialState and eventHandlers should be objects.

```
import {createProvider} from 'react-proxy-state'
...

const ContextProvider = createProvider(initialState, eventHandlers);

const Root = () => (
    <ContextProvider>
        <App/>
    </ContextProvider>
);
```

Read more about ContextProviders eventHandlers in section Context eventhandlers

By default ContextProvider offers *subscribe* and *getState* context functions, to all of its contextual child components.

Subscribe

`subscribe` takes a callback function as argument, and it return a function for cancelling the subscription. Callback function provided as the argument will be called everytime context state changes.

Get State

`getState` returns the current context state

Though context state can be manually be subscribed from context, Components should not access it directly. Read more about how to access context state from the next section about mapContextToProps

ks. https://github.com/jEnbuska/react-proxy-state

Liite 3

React-Proxy-State Context eventhandlers API-kuvaus

Context eventhandlers

Event handlers are functions that are responsible for updating the context state. These function must be passed to ***createProvider*** as second argument durin initialization. ContextProvider server these eventHandlers to all of it's children, and components can access these eventHandlers throught context api.

```
const doSomething = () => proxy => {...};
const doSomethingElse = (parameter) => proxy => {...}
const eventHandlers = {doSomething, doSomethingElse}
const ContextProvider = createProvider(initialState, eventHandlers);
```

Eventhandlers must be defined as ***higher order functions***. When ever a Component invokes these eventHandlers, it's result is invoked with a ***context state proxy as 1st argument***.

ContextProviders simply transforms eventHandlers, into a reqular function, before they are server to components:

```
const before = (...params) => proxy => {...};
const after = (...params) => before(...param)(proxy);
```

For a component to be able to access eventhandlers server by ContextProvider, the component has to announce which context variables it is going to be using, by defining the components ***contextTypes***.

```
import {func} from 'prop-types';
...
const Todo = ({id, description, done}, {setTodoDone}) => (
    <div>
      <p>{description}</p>
      <p onClick={() => setTodoDone(id, !done)}>{done ? 'Done' : 'Not done'}</p>
    </div>
)
Todo.contextTypes = {
  setTodoDone: func
}
const selector = ...
export default mapContextToProps(selector)(Todos);
```

Eventhandler proxy nodes

Every eventhanlers output (the handle function) gets a context state Proxy as the first parameter.

```
const myEventHandler () => proxy => { ... };
```

This Proxy is a root node in a ***shadow datastructure of the actual state***. This node acts as an ***interface for making changes*** to the state, while keeping the states datastructure ***immutable***. A single location in the shadow state is called a ***node***.

The benefit of updating the state by using eventHandler nodes, is that all the update actions are mirrored back to the actual state, and they are applied by using ***pathcopying***. As a result, the context state stays allways ***immutable***.

State variable

Every node represents a particular location of the state. That state can be read by accessing nodes ***state*** variable

```
const logTodoStatus = (id) => (proxy) => {
    const status = proxy.todos[id].done.state;
    console.log(status); // --> true or false
}
```

state variable is a getter, so when accessed it always gets re-evaluated.

You should never be directly to changed or mutate nodes state property.

ks. https://github.com/jEnbuska/react-proxy-state

Updating state

There is four methods that are recommended to be used when ever the underlying state should be updated.

clear

Clear acts on behave of the assigment operation.

```
const setUserName = (userId, name) => {
  objective state.users[userId].name = name
  return function implementation({users}){
    users[userId].name.clear(name);
  }
  result { ...state, users: {...state.users, [userId]: {...state.users[userId], name}} }
}
```

assign

Use assign when ever you would use Object.assign

```
const updateUser = (userId, update) => {
   objective Object.assign(state.users[userId], update)
   return function implementation({users}){
     users[userId].assign(update);
   }
   result { ...state, users: {...state.users, [userId]: {...state.users[userId], ...update}} }
}
```

remove

Use remove when ever you would delete values

```
const removeUsers = (userIds) => {
  objektive usersIds.forEach(id => delete state.users[id])
  function implementation({users}){
    users.remove(...userIds);
  }
  result
    { ...state, users: Object.entries(state.users)
      .filter(([id]) => userIds.every(next => id!==next)
      .reduce((users, [id, user]) => Object.assign(users, {[id]: user}), {}) }
}
```

Toggle

Toggle simply negates the current substate

```
const toggleUserActive = (userId) => {
   objektive state.users[userId].active = !state.users[userId].active
   function implementation({users}){
      users[userId].active.toggle();
   }
   result { ...state, users: {...state.users, [userId]: {...state.users[userId], active: !state.users[userId].active}} }
}
```

ks. https://github.com/jEnbuska/react-proxy-state