AJAX-PYYNNÖT REACT-SOVELLUKSESSA

Vili Kinnunen

Opinnäytetyö
Syyskuu 2018
Tietojenkäsittely
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

KINNUNEN, VILI:
AJAX-pyynnöt React-sovelluksessa

Opinnäytetyö 38 sivua
Syyskuu 2018

Nykyaikaiset verkkosovellukset keskustelevat palvelimen kanssa AJAX-pyyntöjen avulla. Opinnäytetyön tavoitteena oli selvittää, kuinka AJAX-pyyntöjä tulisi tehdä React-sovelluksessa ja miten pyyntöjen tilanhallintaa voitaisiin helpottaa. Tarkoituksena oli tutustua olemassa oleviin menetelmiin ja kehittää avoimen lähdekoodin ohjelmointikirjasto vastaamaan nykyisten menetelmien ongelmiin.

Opinnäytetyön tuloksena npm-pakettirekisteriin julkaistiin React With Requests -ohjelmointikirjasto. Se tarjoaa työkalut AJAX-pyyntöjen määrittelyyn ja hyödyntämiseen React-komponenteissa. Ohjelmointikirjaston rakentamisen lisäksi opinnäytetyössä kerättiin tietoa myös muista menetelmistä AJAX-pyyntöjen hallintaan liittyen. Näistä menetelmistä merkittävimpänä esille nousi opinnäytetyön kirjoittamisen hetkellä vielä julkaisematon React Suspense -ominaisuus.

React With Requests -kirjaston voidaan katsoa onnistuneen hyvin. Se on asennettu npm-pakettirekisteristä yli 200 projektiin opinnäytetyön kirjoittamisen hetkellä. React Suspense -ominaisuuden odotetaan yhtenäistävän React-sovellusten AJAX-pyyntöihin liittyvää logiikkaa. Tämä on opinnäytetyön tekijän mielestä hyvä asia, vaikkakin se todennäköisesti vähentää samalla React With Requests -kirjaston merkitystä tulevaisuudessa.

Asiasanat: verkkosovellukset, react, ajax

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Business Information Systems
Software Production

KINNUNEN, VILI:
AJAX Requests in Web Based React Applications

Bachelor's thesis 38 pages
September 2018

The objective of the thesis was to find out how AJAX requests can be utilised in web based React applications, and how the process of handling the request state can be made easier. The purpose was to get a better understanding of the existing solutions, and to create a new open source library to address the issues of these existing methods.

As a result of the thesis, a package called React With Requests was published in the npm package registry. In addition to building the library, information was gathered on other methods of handling AJAX requests in React applications. The most prominent of these methods was React Suspense, a new feature of React that is yet to be published at the time of writing the thesis.

React With Requests can be considered a success. React Suspense, the upcoming feature in React, is expected to change how handling AJAX requests in React applications will happen in the future. It is most likely going to unify the way AJAX requests are utilised inside of React components. This can be seen as a positive thing even though it will likely make React With Requests a less relevant library in the future.

Key words: web applications, react, ajax

SISÄLLYS

1 JOHDANTO 6
2 TAUSTAA 7
2.1 SPA-sovellukset vs. perinteiset verkkosivut 7
2.2 React verkkosovelluskehityksen tukena 9
3 REACT KÄYTÄNNÖSSÄ 11
3.1 Yksinkertaiset, tilattomat komponentit 11
3.2 Komponenttien sisäinen tila 12
3.3 Komponenteista sovellukseksi 13
4 REACT JA AJAX 15
4.1 Fetch-rajapinta helpottamassa AJAX-pyyntöjä 15
4.2 Datan hakeminen komponentille 16
4.3 Datan hakeminen useissa komponenteissa 17
4.4 React Suspense 21
5 REACT WITH REQUESTS -KIRJASTO 23
5.1 Tausta ja tavoite 23
5.2 Kehitysympäristö, testaaminen ja julkaisu 23
5.3 Toteutus ja käytäntö 24
5.3.1 Request: AJAX pyyntöjen määrittely keskitetysti 24
5.3.2 RequestStateHandler: Yhteinen tilavarasto AJAX-pyynnöille 27
5.3.3 RequestStateProvider: Tilan jakaminen Context API:n avulla 29
5.3.4 RequestStateConsumer: Tilan yhdistäminen komponentteihin 31
5.4 Havaitut ongelmat 34
6 POHDINTA 36
LÄHTEET 38

LYHENTEET JA TERMIT

AJAX	Asynchronous JavaScript and XML. Tekniikka, jota käytetään palvelimen kanssa kommunikointiin verkkosivuilla.
Axios	AJAX-pyyntöjen tekemiseen suunniteltu JavaScript-ohjelmointikirjasto.
DOM	Document Object Model. Tapa, jolla esimerkiksi HTML-dokumentin rakenne voidaan esittää puuna.
ES5	EcmaScript 2013. EcmaScript-standardin viides versio.
ES6	EcmaScript 2015. EcmaScript-standardin kuudes versio.
ESLint	Työkalu, joka mahdollistaa tyylisääntöjen määrittelyn ja tarkastuksen JavaScript-projekteissa.
Fetch	Selaimen rajapinta, joka mahdollistaa AJAX-pyyntöjen tekemisen.
Git	Versionhallinnan työkalu.
GitHub	Verkkosivusto Git-versionhallintaa hyödyntäville projekteille.
HTML	Hypertext Markup Language. Kieli verkkosivujen rakenteen määrittelemiseen.
Java	Laitteistoriippumaton ohjelmointikieli.
JavaScript	EcmaScript-standardiin pohjautuva ohjelmointikieli, jota käytetään erityisesti verkkosivujen dynaamisen toiminnallisuuden toteuttamiseen.
jQuery	Avoimen lähdekoodin JavaScript-kirjasto.
JSON	JavaScript Object Notation. Tiedonsiirtoformaatti.
JSX	JavaScript XML. Lisäys JavaScriptiin, joka mahdollistaa HTML:n kaltaisen syntaksin kirjoittamisen JavaScript-koodissa.
React	Käyttöliittymien rakentamiseen suunniteltu ohjelmointikirjasto.
XMLHttpRequest	Selaimen rajapinta, joka mahdollistaa tiedon vaihtamisen verkkosivun ja palvelimen välillä.

1 JOHDANTO

Verkkosivut ovat nykyään paljon muutakin kuin staattisia yksityishenkiköiden ja yritysten kotisivuja. Verkosta löytyy täysillä ominaisuuksilla varustettuja sovelluksia, jotka toimivat suoraan käyttäjän selaimessa. Sosiaalisessa mediassa voidaan selata loputon määrä kuvia ja viestejä, ilman että sivulta tarvitsee koskaan siirtyä toiselle. Nämä ovat niin sanottuja dynaamisia verkkosovelluksia, jotka lataavat sisältönsä AJAX-pyyntöjen avulla palvelimelta.

Opinnäytetyössä keskitytään verkkosovelluskehitykseen React-ohjelmointikirjastoa hyödyntäen. React, toisin kuin monet ohjelmointikehykset, ei tarjoa omia työkaluja AJAX-pyyntöjen tekemiseen. Opinnäytetyön tavoitteena on selvittää, kuinka AJAX-pyyntöjä tulisi tehdä React-sovelluksissa, ja miten niiden tilanhallintaa voitaisiin helpottaa. Tarkoituksena on tutustua olemassa oleviin menetelmiin ja kehittää avoimen lähdekoodin ohjelmointikirjasto vastaamaan nykyisten menetelmien ongelmiin.

Raportissa tutustutaan ensin nykypäivän verkkosovellus-kehitykseen ja sen haasteisiin, joihin opinnäytetyöllä pyritään vastaamaan. Tätä seuraa lyhyt tutustuminen React-ohjelmointikirjaston perusteisiin. Perusteet luovat pohjaa menetelmille, joita AJAX-pyyntöjen tekemiseen voidaan hyödyntää. Raportissa kerrotaan perinteisistä menetelmistä ja tulevaisuuden näkymistä aiheeseen liittyen. Lopuksi esitellään React With Requests -ohjelmointikirjasto, joka kehitettiin osana opinnäytetyötä helpottamaan AJAX-pyyntöjen määrittelyä ja hyödyntämistä React-sovelluksissa.

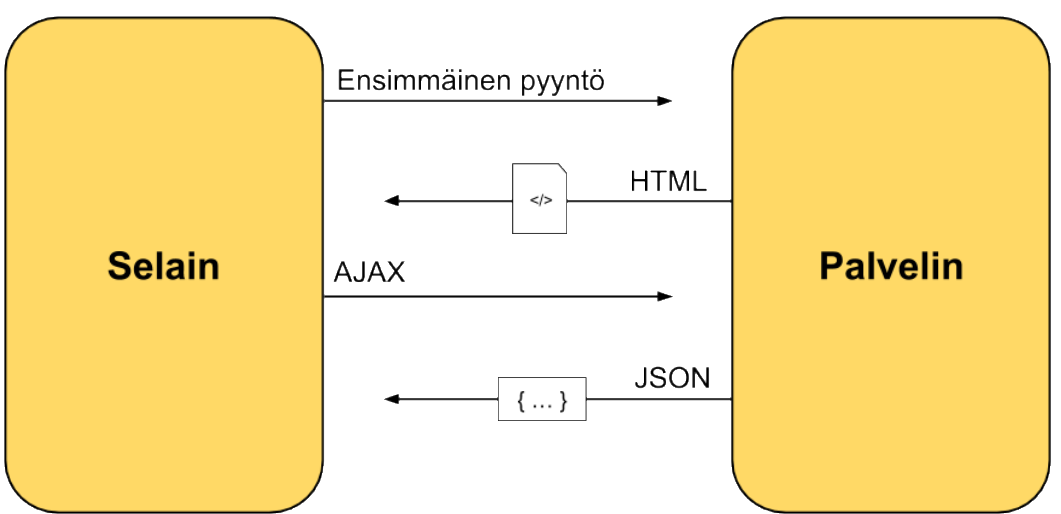
Opinnäytetyön toimeksiantajana toimii Futurice Oy. Futurice on Suomesta lähtöisin oleva kansainvälinen ohjelmistotalo, joka pyrkii ratkaisemaan asiakkaidensa ongelmia muun muassa ohjelmistokehityksen, designin ja palvelumuotoilun keinoin. Opinnäytetyössä rakennettu ohjelmointikirjasto on toteutettu osana Futuricen Spice Program -hanketta, joka sponsoroi avoimen lähdekoodin projekteja.

2 TAUSTAA

2.1 SPA-sovellukset vs. perinteiset verkkosivut

Verkkosovellukset korvaavat väistämättä vanhoja työpöytäsovelluksia. Niitä on helpompi käyttää, niiden päivittäminen on yksinkertaisempaa ja ne eivät ole sidottuja yhteen laitteeseen. Verkkosovellusten kehitykseen on kaksi johtavaa suunnittelumallia: yhden sivun verkkosovellukset (Single Page Applications, tästä eteenpäin SPA-sovellus) sekä monen sivun verkkosovellukset. (Skólski 2016.)

SPA-sovelluskehitys on mahdollista AJAX:n ansiosta. AJAX on tekniikka, joka antaa verkkosovelluksen keskustella palvelimen kanssa ja ladata sivulle sisältöä ilman, että selaimen tarvitsee ladata sivua kokonaan uudelleen. AJAX ei siis lataa raskaita HTML, CSS ja JavaScript tiedostoja, vaan sen välityksellä siirretään useimmiten dataa erilaisten tiedonsiirtoformaattien avulla. (Ezell 2018.) Kuviossa 1 esitellään SPA-sovelluksen elinkaari.



KUVIO 1. SPA-sovelluksen elinkaari (Wasson 2013, muokattu)

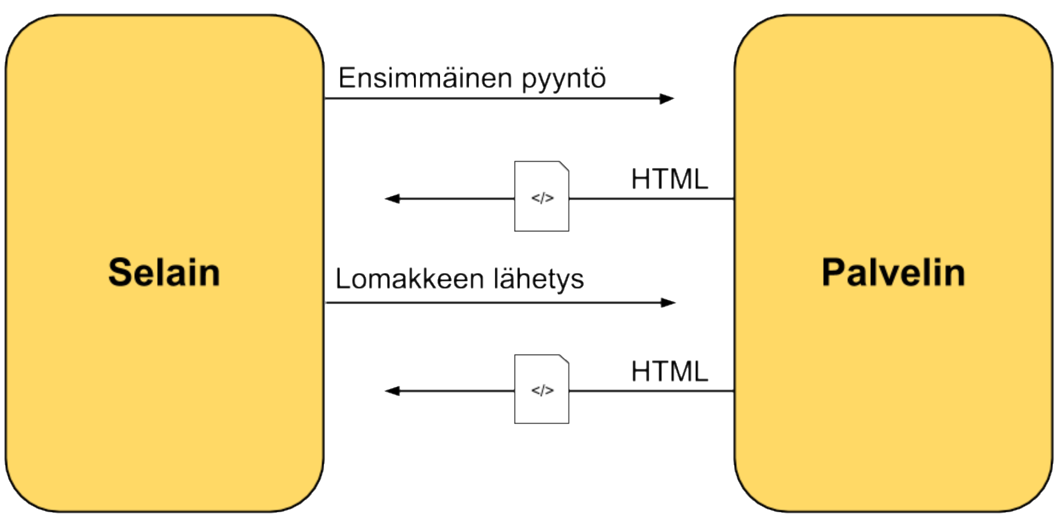
Teknisesti katsottuna SPA-sovellus siirtää suurimman osan prosessoinnista selaimen puolelle, sillä palvelimen vastuulla on ainoastaan sovelluksen datan hallinta. Kuviosta 1 nähdään, että selain lataa SPA-sovelluksen HTML-sivun vain kerran. Sitä mukaa kun käyttäjä käyttää sovellusta, palvelimelle lähetetään AJAX-pyyntöjä JavaScriptin avulla.

Uudelle sivulle navigoiminen tai lomakkeen lähettäminen tarkoittaa useimmiten uutta AJAX-pyyntöä. Pyyntöjen seurauksena käyttöliittymää päivitetään dynaamisesti JavaScriptin avulla. (Ezell 2018.)

SPA-sovellukset ovat nopeampia kuin monen sivun sovellukset, sillä raskaat HTML, CSS ja JavaScript tiedostot ladataan vain kerran. Alkulataus on tyypillisesti pitkä, mutta navigointi sivustolla on nopeaa ja sujuvan tuntuista. Nopeus johtuu siitä, että sivua ei tarvitse piirtää kokonaan uudelleen uuden datan saapuessa. Pelkän datan vaihtaminen palvelimen kanssa on yleensä nopeaa. Tunnettuja esimerkkejä SPA-sovelluksista ovat muun muassa Gmail, Google Maps ja Facebook. (Skólski 2016.)

Monen sivun sovellukset ovat perinteisempi tapa rakentaa dynaamisia verkkosovelluksia. Lähes kaikki sovelluslogiikka on palvelimen puolella, toisin kuin SPA-sovelluksissa. Tästä johtuen selaimen rooli on hyvin kevyt: sen tarvitsee ainoastaan piirtää palvelimen lähettämä HTML-dokumentti verkkosivuksi. (Ezell 2018.)

Samoin kuin SPA-sovelluksissa, ensimmäinen pyyntö palvelimelle palauttaa myös monen sivun sovelluksissa HTML-sivun. Kuviosta 2 nähdään, että monen sivun sovelluksissa seuraavat pyynnöt palvelimelle palauttavat HTML-sivuja, toisin kuin SPA-sovelluksissa. Palvelimen kanssa ei siis vaihdeta pelkkää dataa, vaan palvelimen vastuulla on myös näkymän rakentaminen datan ympärille. Tämä tulee myös selaimelle raskaaksi, sillä se joutuu piirtämään uudelleen koko näkymän, kun sivua joudutaan päivittämään käyttäjän toiminnan seurauksena. (Ezell 2018.)



KUVIO 2. Monen sivun sovelluksen elinkaari (Wasson 2013, muokattu)

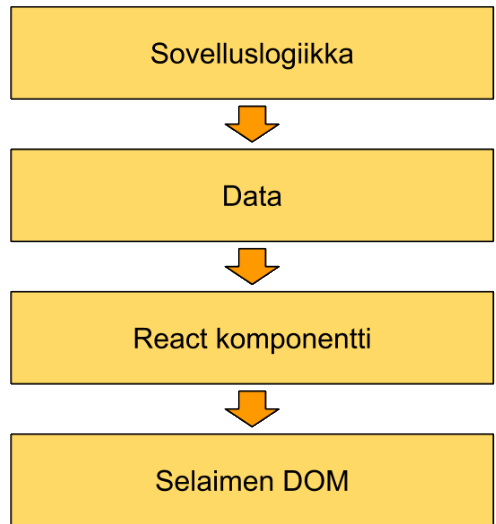
SPA-sovellusten suosio perustuu erinomaisen käytettävyyden lisäksi myös hyvin pitkälti niiden helppouteen kehittäjän näkökulmasta. Kun palvelin ja verkkosovellus eivät ole yksi ja sama asia, samaa taustapalvelua voidaan hyödyntää esimerkiksi mobiilisovelluksen rakentamiseen. Koska SPA-sovellus ei ole niin sanotusti kiinni palvelimessa, sen kehitystyö voidaan aloittaa tarvittaessa myös ilman palvelinta. Näin käyttöliittymää päästään kokeilemaan, vaikka varsinaista dataa ei olisikaan vielä olemassa. (Skólski 2016.)

2.2 React verkkosovelluskehityksen tukena

Laajoja verkkosovelluksia on tänä päivänä työlästä ja jopa vaikeaa rakentaa ilman apuvälineitä. Pelkkä HTML, CSS ja JavaScript osaaminen eivät riitä enää pitkälle. Kehityksen tukena hyödynnetäänkin usein erilaisia ohjelmointikehyksiä ja kirjastoja, jotka tekevät monipuolisten sovellusten kirjoittamisesta nopeampaa.

React on Facebookin vuonna 2013 kehittämä sovelluskirjasto, jonka on tarkoitus helpottaa data-lähtöisten verkkosovellusten rakentamista. Perinteisesti verkkosovelluskehityksen tukena käytetyt ohjelmointikehykset ovat olleet todella monipuolisia, ja tarjonneet työkalut AJAX-pyyntöjen tekemisestä erilaisiin navigaatio-toimintoihin. Toisin kuin nämä sovelluskehykset, React on sovelluskirjasto. Sen vastuulla on ainoastaan yksi teh-

tävä: näkymien piirtäminen. Kehittäjän on itse valittava työkalut sovelluksen muihin tarpeisiin, kuten AJAX-pyyntöjen tekemiseen. (Banks & Porcello 2017.) Kuvio 3 selventää, mikä on Reactin rooli verkkosovelluksessa.



KUVIO 3. Reactin rooli verkkosovelluksessa (Bocuch 2018, muokattu)

React-sovelluksessa varsinainen sovelluslogiikka voi olla täysin erillään käyttöliittymästä. Sovelluslogiikan on tarkoitus tuottaa ainoastaan dataa, jonka perusteella käyttöliittymä voidaan piirtää. React-komponenttien tehtävänä on vastaanottaa sovelluslogiikan tuottama data ja piirtää sen pohjalta HTML-koodia sivulle. (Bocuch 2018.) Seuraavassa luvussa tutustutaan React-komponenttien toteuttamiseen käytännössä.

3 REACT KÄYTÄNNÖSSÄ

Tässä luvussa esitellään hyvin yksinkertaisella tasolla, miten React toimii käytännössä. Luvun koodiesimerkeissä rakennetaan hyvin yksinkertainen lamppu-komponentti Reactin avulla. Koodiesimerkkien ymmärtäminen vaatii JavaScriptin ES6-standardin osaamista.

3.1 Yksinkertaiset, tilattomat komponentit

Reactin kaksi tärkeintä rajapintaa ovat Component ja ReactDOM. Component-rajapinta tarjoaa työkalut uudelleenkäytettävien komponenttien rakentamiseen. Komponentit ovat ikään kuin rakennuspalikoita, joista React-sovellus koostuu. ReactDOM-rajapinnan vastuulla on sen sijaan React-komponenttien piirtäminen selaimen DOM:iin. (Bocuch 2018.)

Kuvassa 1 esitellään yksinkertaisen React-komponentin toteutus. React-komponentti voi yksinkertaisimmillaan sisältää pelkän render-metodin, joka määrittelee, miten komponentti piirretään ruudulle. Komponentin render-metodin palautusarvona on niin sanottua JSX-koodia. JSX muistuttaa hyvin pitkälti HTML-koodia. JSX muunnetaan todellisuudessa JavaScript objektiksi, joka kertoo Reactille, miten komponentti tulee piirtää (Boduch 2017).

```
class Lamp extends React.Component {
  render() {
    return <img src="lamp.png"/>
  }
}
```

KUVA 1. Esimerkki yksinkertaisesta React-komponentista

Jotta komponentit voisivat piirtää näkymän sovelluksen tilan mukaisesti, niille annetaan dataa, jonka perusteella render-metodin palautusarvo määräytyy. Näitä parametreja kutsutaan Reactissa propseiksi, englannin kielisen sanan properties mukaan. Annettuihin parametreihin voi viitata komponentin sisällä muuttujan this.props avulla. (React: Components and Props n.d.) Kuvassa 2 piirrettävän kuvan osoite valitaan color-parametrin perusteella.

```
class Lamp extends React.Component {
  render() {
    const { color } = this.props;
    return <img src={`${color}_lamp.png`} />
  }
}
```

KUVA 2. Esimerkki parametrien (props) hyödyntämisestä

Mikäli komponentti ei hyödynnä sisäistä tilaa tai elinkaarimetodeja, sitä voidaan kutsua tilattomaksi komponentiksi. Tällaisia komponentteja voidaan esittää Reactissa yksinkertaisemmin perinteisinä JavaScript funktioina. Tilattomat komponentit ottavat vastaan parametreja (props) ja palauttavat näkymän niiden pohjalta. Kuvassa 3 aiemmin määritelty lamppu-komponentti on esitetty JavaScript funktiona.

```
const Lamp = ({ color }) => (
  <img src={`${color}_lamp.png`}/>
);
```

KUVA 3. Lamppu-komponentti funktiona

3.2 Komponenttien sisäinen tila

Parametreista (props) tulevan datan lisäksi komponenteilla voi olla myös oma sisäinen tilansa, joka vaikuttaa komponentin piirtämään näkymään. Sisäisessä tilassa voi olla esimerkiksi tekstikentän sisältö, tai muu tieto, jota muut osat sovellusta eivät tarvitse. Komponentti vastaa itse tämän tilan ylläpitämisestä. Kuvassa 4 esitellään, kuinka komponentin sisäistä tilaa voidaan hyödyntää.

```
class Lamp extends React.Component {
  state = {
    on: true
  }

  toggleState = () => {
    this.setState({
      on: !this.state.on
    });
  }

  render() {
    return <img src={this.state.on ? "lamp_on.png" : "lamp_off.png"}
                onClick={this.toggleState}
           />
  }
}
```

KUVA 4. Esimerkki komponentin sisäisen tilan hyödyntämisestä

Kuvan 4 esimerkissä lampulla on sisäinen tila, joka kertoo, onko lamppu päällä vai pois päältä. Tilaa päivitetään toggleState-metodilla, joka kutsuu sisäisesti React-komponentin setState-metodia. Render-metodissa tilaa hyödynnetään piirrettävän kuvan valitsemiseen. Kuvaan kytketään myös onClick-tapahtumakäsittelijä, joka kutsuu toggleState-metodia. Kaikki logiikka on komponentin sisällä, eikä se näin aiheuta sivuvaikutuksia muihin osiin sovellusta.

3.3 Komponenteista sovellukseksi

Tähän mennessä raportissa ollaan tutustuttu yksittäisen komponentin rakentamiseen. Komponentteja voi ajatella sovelluksen rakennuspalikoina. Yksinään ne ovat lähes hyödyttömiä, mutta yhdessä niistä saa kasattua monipuolisia sovelluksia.

React komponentit voivat piirtää perinteisten HTML-elementtien lisäksi toisia React komponentteja. (React: Components and Props n.d.) Komponenteille voidaan antaa parametreja (props) samalla tavalla kuin HTM-koodissa annetaan parametreja elementeille. Kuvassa 5 määritellään ylimmän tason App-komponentti, joka piirtää kolme aiemmin määriteltyä lamppu-komponenttia.

```
class App extends React.Component {
  render() {
    <div>
      <Lamp color="red"/>
      <Lamp color="green"/>
      <Lamp color="blue"/>
    </div>
  }
}
```

KUVA 5. Esimerkki sisäkkäisistä React-komponenteista

Verkkosovelluksista selaimen DOM:iin piirretään useimmiten pelkkä ylimmän tason App-komponentti, joka kokoaa alleen kaikki muut komponentit. (React: Components and Props n.d.) Kuvassa 6 App-komponentti piirretään id:llä ”root” merkityn HTML-elementin sisään. Piirtämiseen hyödynnetään ReactDOM-rajapintaa.

```
ReactDOM.render(
  <App/>,
  document.getElementById("root")
);
```

KUVA 6. Esimerkki React-komponentin piirtämisestä DOM:iin

4 REACT JA AJAX

4.1 Fetch-rajapinta helpottamassa AJAX-pyyntöjä

Koska React keskittyy ainoastaan sovelluksen näkymä-kerrokseen, se ei varsinaisesti ota kantaa siihen, miten HTTP-pyynnöt toteutetaan. Tämä antaa kehittäjälle mahdollisuuden valita sopivimmat työkalut pyyntöjen tekemiseen. Vaihtoehtoina on useita erilaisia AJAX-kirjastoja ja selaimen Fetch-rajapinta. (Ceddia 2018.)

Ennen Fetch-rajapinnan tuloa HTTP-pyynnöt tehtiin selaimen XMLHttpRequest-rajapinnan avulla. Vaikeasti käytettävä rajapinta sai kilpailijakseen nopeasti erilaisia kirjastoja, jotka pyrkivät piilottamaan monimutkaisen logiikan helppokäyttöisten rajapintojen taakse. Esimerkiksi jQueryn ajax-metodi oli pitkään suosittu tapa ladata dynaamista sisältöä. (Vieira 2018.)

Kuvan 7 esimerkistä nähdään, kuinka GET pyyntö voidaan toteuttaa fetch-metodin avulla, antamalla parametriksi haettavan resurssin osoite. Fetch palauttaa Promise-objektin, jonka arvoa odotetaan esimerkissä async-await syntaksin avulla. Arvona on Response-objekti, jolta voidaan kutsua json-metodia, joka muuntaa palvelimen vastauksen JavaScript-objektiksi.

```
async function fetchThings() {
  const response = await fetch('http://example.com/things');
  return await response.json();
}
```

KUVA 7. Esimerkki Fetch-rajapinnan käytöstä

Kaikki uusimmat selaimet tukevat Fetch-rajapintaa, joten sen käyttäminen ei välttämättä vaadi erillisten kirjastojen asentamista. Taaksepäin yhteensopivuutta varten Fetch-rajapinta on saatavilla myös kirjastona, joka hyödyntää taustalla XMLHttpRequest-rajapintaa (Vieira 2018). Tämän opinnäytetyön esimerkeissä hyödynnetään Fetch-rajapintaa.

4.2 Datan hakeminen komponentille

Monet komponentit tarvitsevat palvelimelta tulevaa dataa näyttääkseen sisältöä käyttäjälle. Tässä luvussa selvitetään, miten yksittäinen komponentti voi hakea ja tallentaa dataa omaan käyttöönsä, ja piirtää sen pohjalta elementtejä käyttöliittymään. Tämä tapa on hyödyllinen silloin, kun kyseistä dataa tarvitsee ainoastaan yksi komponentti, sillä haettu data säilötään komponentin sisäiseen tilaan (Bertoli 2017).

Kuvan 8 esimerkissä rakennetaan React-komponentti, jonka tarkoituksena on näyttää lista viestejä, jotka ladataan palvelimelta. Viestien lataaminen suositellaan toteuttamaan componentDidMount-elinkaarimetodissa, jota komponentti kutsuu automaattisesti, kun se upotetaan sivulle (React: React.Component n.d.). Lataukseen käytetään tässä tapauksessa Fetch-rajapintaa, jolla parsitaan myös palvelimen JSON-muotoinen vastaus. JavaScript-taulukoksi parsittu vastaus tallennetaan komponentin sisäisen tilan messages-kenttään, joka alustettiin tyhjänä taulukkona komponentin luomisen aikana.

```
class MessageList extends React.Component {
  state = {
    messages: []
  };

  async componentDidMount() {
    const response = await fetch('http://example.com/messages');
    const messages = await response.json();

    this.setState({
      messages
    });
  }

  render() {
    return this.state.messages.map(message => (
      <div key={message.id}>
        <h2>{message.title}</h2>
        <p>{message.content}</p>
      </div>
    ));
  }
}
```
KUVA 8. Esimerkki datan lataamisesta React-komponentin sisällä

Komponentin render-metodi vastaa kuvan 8 esimerkissä saadun vastauksen muuntamisesta käyttöliittymän palasiksi. Vastauksena saadun taulukon oletetaan koostuvan objekteista, joilla on kentät id, title sekä content. Taulukon prototyypin sisäänrakennettua map-metodia hyödyntäen jokainen taulukon solu muunnetaan React-komponentiksi. Tässä tapauksessa render-metodin rakentaminen on vielä hyvin suoraviivaista, sillä komponentti ei ota huomioon AJAX-pyynnön tilaa. Kun pyyntö on kesken, render-metodi ei palauta mitään, sillä taulukko alustettiin tyhjänä.

Toisin kuin kuvassa 8, verkkosovelluksen käyttöliittymässä hyödynnetään usein myös AJAX-pyynnön tilaa. Tila voidaan esittää kolmella muuttujalla: loading, error, sekä result. Kuvan 9 esimerkissä tilan perusteella valitaan, mitä komponentin render-metodi palauttaa. Tilaa on tärkeä hyödyntää, jotta komponentti ei yritä käyttää hyödyksi resurssia, jota ei ole vielä olemassa.

```
render() {
  return this.state.loading ?
    <div>Loading...</div>
    :
    this.state.error ?
      <div>Something went wrong</div>
      :
      <div>{this.state.result}</div>
}
```

KUVA 9. Esimerkki AJAX-pyynnön tilan käsittelystä render-metodissa

Kuvassa 9 loading-muuttuja sisältää totuusarvon, joka indikoi nimensä mukaisesti, onko lataus kesken. Kyseisen muuttujan avulla on mahdollista näyttää esimerkiksi latausindikaattori varsinaisen komponentin sijasta, kun haluttu resurssi ei ole vielä saatavilla. Mikäli pyyntö epäonnistuu, asetetaan error-muuttujaan teksti, joka voidaan näyttää myös käyttöliittymässä. Pyynnön onnistuessa result-muuttujaan asetetaan palvelimen vastaus, jonka pohjalta varsinainen komponentti voidaan piirtää.

4.3 Datan hakeminen useissa komponenteissa

Kun useammat komponentit haluavat käyttää hyödyksi palvelimelta haettavaa dataa, on vältettävä saman logiikan monistamista komponenttien välillä. Vaikka haettavat resurssit olisivat erilaisia, kolme asiaa pysyvät usein samana komponentista riippumatta:

1. AJAX-pyyntö tehdään heti, kun komponentti upotetaan sivulle.
2. Pyynnön tilaa seurataan muuttujien loading, error ja result avulla.
3. Pyyntö toteutetaan samaa rajapintaa tai kirjastoa hyödyntäen.

React-sovelluskehityksessä hyödynnetään usein niin sanottuja ylemmän tason komponentteja (Higher Order Component, HoC). Niiden avulla logiikkaa, kuten edellä mainitut kolme AJAX-pyyntöihin liittyvää toiminnallisuutta, voidaan jakaa komponenttien välillä. Kaikessa yksinkertaisuudessaan ylemmän tason komponentti on funktio, joka ottaa vastaan komponentin, ja palauttaa sen ”rikastettuna”. Rikastamisella tarkoitetaan sitä, että ylemmän tason komponentti antaa alemman tason komponentille parametreja. (Bertoli 2017.)

Ylemmän tason komponentit voivat ratkaista ongelman AJAX-pyyntöihin liittyvästä logiikan monistamisesta. Datan lataaminen ja tilan hallinta voidaan siirtää ylemmälle tasolle, jolloin alemman tason komponentin vastuulle jää ainoastaan pyynnön tilan hyödyntäminen. (Bertoli 2017.)

Kuvassa 10 on Message-komponentti, jota rikastetaan vielä toteuttamattoman withResource-funktion avulla. Koska jokaista komponenttia ei haluta rikastaa samalla resurssilla, withResource-funktion on tarkoitus ottaa vastaan halutun resurssin URL-osoite ja palauttaa rikastukseen käytettävä funktio. Vasta palautettua funktiota kutsutaan rikastettavalla komponentilla Message.

```
const Message = (props) => (
  props.result ?
    <div className="message">
      <h2>{props.result.title}</h2>
      <p>{props.result.content}</p>
    </div>
    :
    <div>Message is not here yet</div>
);

export default withResource('http://example.com/message')(Message);
```

KUVA 10. Message-komponentin rikastaminen ylemmän tason komponentin avulla

Kuvassa 10 hyödynnettiin vielä toteuttamatonta withResource-funktiota. WithResource-funktio mahdollistaa AJAX-pyyntöihin liittyvän logiikan siirtämisen pois komponenteilta yhden tason ylemmäksi. Kuten termi ”ylemmän tason komponentti” kertoo, withResource-funktiossa on niin sanotusti monta tasoa. Kuva 11 esittelee withResource-funktion toteutuksen.

```
function withResource(url) {
  return (WrappedComponent) => {
    class EnhancedComponent extends React.Component {
      state = {
        loading: false,
        error: null,
        result: null
      };

      setRequestState = (error, loading, result) => {
        this.setState({
          error,
          loading,
          result
        });
      }

      componentDidMount() {
        this.setRequestState(null, true, null);

        try {
          const response = await fetch(url);
          const result = await response.json();
          this.setRequestState(null, false, result);
        } catch (err) {
          this.setRequestState(err.message, false, null);
        }
      }

      render() {
        return <WrappedComponent {...this.props} {...this.state} />
      }
    }

    return EnhancedComponent;
  }
}
```

KUVA 11. WithResource-funktion rakentaminen

Ylimmällä tasolla on funktio, joka ottaa vastaan URL-osoitteen, ja palauttaa uuden funktion. Palautettu funktio pystyy edelleen hyödyntämään ylimmälle funktiolle annettua URL-osoitetta. Ylimmän tason funktiosta palautettu nimetön funktio ottaa vastaan WrappedComponent muuttujan.

Nimettömän funktion tarkoituksena on luoda sisäisesti uusi React-komponentti, joka piirtää ainoastaan annetun komponentin (WrappedComponent), antaen sille omat parametrinsa (props) sekä sisäisen tilansa (state). Luodussa komponentissa toteutetaan AJAX-

pyyntöön liittyvä logiikka hyödyntäen sisäistä tilaa ja componentDidMount-elinkaarimetodia. Sisäinen tila, joka sisältää muuttujat loading, error ja state, siirtyy render-metodissa rikastettavalle komponentille sellaisenaan.

Kuvan 11 esimerkissä toteutettu withResource-funktio ei ole vielä kovin monipuolinen työkalu AJAX-pyyntöjen tilanhallintaa. Sovelluksissa on usein lukuisia AJAX-pyyntöihin liittyviä toiminnallisuuksia, joita esimerkin komponentti ei vielä pysty hoitamaan. Tähän tarkoitukseen on olemassa kirjastoja, kuten React Refetch, tai opinnäytetyön osana toteutettu React With Requests. Tulevaisuudessa vastauksen näihin ongelmiin voi tuoda myös React Suspense -ominaisuus, josta kerrotaan seuraavassa luvussa.

4.4 React Suspense

Dan Abramov kertoi puheessaan ”Beyond React 16” (2018), mihin suuntaan Reactia ollaan kehittämässä tulevaisuudessa. Hän esitteli ominaisuuden React Suspense, joka tulee helpottamaan asynkronisesti ladattavan sisällön käsittelyä React-sovelluksissa. Opinnäytetyön kirjoittamisen hetkellä näitä ominaisuuksia ei ole vielä julkaistu, joten tässä luvussa esiteltävät rajapinnat saattavat poiketa lopullisesta versiosta.

React ei edelleenkään tule ottamaan kantaa siihen, miten AJAX-pyynnöt toteutetaan. Sen sijaan se tarjoaa createFetcher-funktion, joka ottaa parametrina Promise-objekteja palauttavan funktion. CreateFetcher-funktio palauttaa Fetcher-objektin. Fetcher-objektin avulla pystytään hakemaan dataa render-metodissa, ja se toimii samalla välimuistina AJAX-pyynnöille. Sitä pystytään hyödyntämään myös kuvatiedostojen hakemiseen, sekä sivuston lähdekoodin lataamiseen osissa. (Abramov 2018.)

Jatkossa React tukee asynkronista piirtämistä. Tämä tarkoittaa sitä, että render-metodissa määritellyn asynkronisen operaation tulee valmistua, ennen kuin näkymä piirretään uudelleen. Asynkronisella operaatiolla tarkoitetaan tässä tapauksessa mitä tahansa Fetcher-objektin read-metodin kutsua. Kuvassa 12 luodaan MoviePage-komponentti, joka piirtää movieFetcher-objektilta tulevaa dataa. Piirtäminen tapahtuu tällöin vasta, kun movieFetcher on ladannut pyydetyn datan. Mikäli movieFetcher-objektilla on haluttu resurssi jo välimuistissa, piirtäminen toimii viiveettä. (Abramov 2018.)

```
import { fetchMovies } from './api';
const movieFetcher = createFetcher(fetchMovies);

const MoviePage = () => {
  const movies = movieFetcher.read();

  return movies.map(movie => <MovieCard {...movie} />);
}
```

KUVA 12. Esimerkki asynkronisesti piirtämisestä

Käyttäjälle on hyvä antaa niin sanottuja vihjeitä siitä, että verkkosovellus lataa jotakin. Placeholder-komponentti mahdollistaa juuri tämän toiminnallisuuden. Ilman Placeholder-komponenttia näkymä pysyy ennallaan ja interaktiivisena sisältöä ladattaessa. Se ei kuitenkaan indikoi millään tavalla, että jotakin on tapahtumassa. Placeholder-komponentin avulla voidaan määritellä, mitä ruudulle piirretään, kun sen sisällä olevat komponentit eivät ole vielä valmiita piirrettäviksi. (Abramov 2018.)

Kuvan 13 esimerkissä MoviePage on komponentti, joka lataa asynkronisesti sisältönsä. Placeholder-komponentin avulla käyttäjälle näytetään Spinner-komponentti (fallback-parametri), mikäli lataus kestää pidempään kuin 1000 millisekuntia (delayMs-parametri). Kun MoviePage-komponentti on valmis näytettäväksi, Placeholder-komponentti ei vaikuta näkymään millään tavalla. (Abramov 2018.)

```
render() {
  return (
    <Placeholder
      delayMs={1000}
      fallback={<Spinner />}
    >
      <MoviePage />
    </Placeholder>
  );
}
```

KUVA 13. Esimerkki Placeholder-komponentin hyödyntämisestä

React Suspense tulee tarjoamaan työkalut sekä AJAX-pyyntöjen tekemiseen React-komponenteista käsin, että näkymän päivittämiseen pyyntöjen tilan mukaisesti. Tämä on hyödyllistä erityisesti hitaammilla yhteyksillä varustettujen laitteiden tukemisen kannalta. Uudet ominaisuudet tulevat todennäköisesti saataville Reactin seuraavassa versiossa vielä vuoden 2018 aikana. (Abramov 2018.)

5 REACT WITH REQUESTS -KIRJASTO

5.1 Tausta ja tavoite

Kuten luvussa 4.3 kerrottiin, ylemmän tason komponentteja voidaan hyödyntää sovelluksessa logiikan jakamiseen komponenttien välillä. Vaikka AJAX-pyyntöihin liittyvä logiikka olisikin vain yhdessä paikassa, voi sen miettimiseen mennä paljon aikaa sovellusta rakentaessa. React With Requests -kirjasto kehitettiin osana opinnäytetyötä tarjoamaan valmis ratkaisu AJAX-pyyntöjen käsittelyyn React-sovelluksessa.

Kirjaston tavoitteena on tarjota selkeä ja yhdenmukainen tapa määritellä sovelluksen käyttämät AJAX-pyynnöt yhdessä paikassa. Näin kaikki pyyntöihin liittyvä logiikka on keskitetty, ja siihen on helppo tehdä muutoksia käymättä koko sovelluksen lähdekoodia läpi. Yhdenmukaisesti määriteltyjä pyyntöjä on helppo hyödyntää kaikkialla sovelluksessa.

Pyyntöjen määrittelyn lisäksi kirjasto tarjoaa työkalut pyyntöjen tekemiseen komponenteista käsin. Kehittäjä voi määritellä, mistä AJAX-pyynnöistä komponentti on riippuvainen. Kirjasto hoitaa pyyntöjen lähettämisen palvelimelle. Komponentin vastuulle jää ainoastaan palvelimen vastauksen pohjalta käyttöliittymän päivittäminen.

5.2 Kehitysympäristö, testaaminen ja julkaisu

React With Requests kirjoitettiin kokonaan avoimen lähdekoodin Visual Studio Code -koodieditorilla, joka tarjoaa hyvät työkalut JavaScript-kehitykseen lukuisten lisäosiensa ansiosta. Kehitystä helpottivat muun muassa automaattinen testien ajo muutoksien yhteydessä, Git-integraatio sekä erinomainen tuki JavaScript-ohjelmointikielelle. Erityisen hyödyllinen projektissa oli myös koodieditorin sisäänrakennettu virheidenetsintä-tila, joka mahdollistaa ohjelman suorittamisen seurannan rivi riviltä reaaliajassa.

Kaikki ohjelmakoodi kirjoitettiin JavaScriptin ES6-versiolla, sillä se tarjoaa yksinkertaisemman syntaksin monien asioiden kirjoittamiseen, verrattuna esimerkiksi vanhempaan ES5-versioon. Lähdekoodin siistinä ja yhdenmukaisena pitämiseksi projektiin otettiin

käyttöön myös ESLint-työkalu, joka varmistaa, että sen määrittämiä tyylisääntöjä noudatetaan.

Kaikki ES6-version ominaisuudet eivät ole vielä tuettuna uusimmissa selaimissa. Varsinkaan vanhemmat selainversiot eivät ymmärrä uutta syntaksia. Tähän tarkoitukseen on kehitetty Babel, joka pystyy kääntämään uudella syntaksilla kirjoitettua JavaScriptiä vanhemmille selainversioille yhteensopivaksi. Kaikki kirjaston lähdekoodi on käännetty Babelin avulla JavaScriptin ES5-standardia vastaavaksi.

Kirjaston testaamiseen hyödynnettiin Jest-ohjelmointikehystä. Jest on Facebookin kehittämä työkalu, joka on suunniteltu erityisesti React-sovelluskehitystä silmällä pitäen. Jestin lisäksi testeihin käytettiin Enzymeä. Enzyme on AirBnB:n kehittämä kirjasto, joka sisältää lukuisia työkaluja React-komponenttien testaamiseen. Jest ja Enzyme toimivat hyvin yhteen, ja mahdollistavat kattavien yksikkö- ja integraatiotestien kirjoittamisen komponenteille. Testit tarjosivat varmuutta kehittämiseen, ja poistivat manuaalisen testaamisen tarpeen lähes kokonaan.

Ohjelmointikirjasto julkaistiin npm-pakettirekisteriin nimellä react-with-requests. Npm-pakettirekisteri on avoin julkaisualusta JavaScript-moduuleille, jota hyödynnetään monissa JavaScript-projekteissa. Kirjaston versionhallintaan hyödynnettiin GitHubia, josta ohjelmointikirjaston lähdekoodi löytyy MIT-lisensoituna. MIT-lisenssi tarkoittaa, että ohjelmointikirjastoa voidaan vapaasti hyödyntää missä tahansa projektissa.

5.3 Toteutus ja käytäntö

5.3.1 Request: AJAX pyyntöjen määrittely keskitetysti

React With Requests -kirjastoa käytettäessä kaikki sovelluksen AJAX-pyynnöt määritellään Request-luokan avulla. Request on tämän ohjelmointikirjaston tarpeisiin kehitetty apuluokka, joka mahdollistaa AJAX-pyyntöjen kuvailun siten, että ohjelmointikirjaston muut osat ymmärtävät niitä. Request-luokka ei kuitenkaan ota kantaa itse AJAX-pyyntöjen toteutukseen. Ohjelmointikirjaston kannalta ei siis ole väliä, tehdäänkö pyyntö Fetch-rajapinnalla tai esimerkiksi Axios-kirjastolla.

Kuvassa 14 on esimerkki Request-luokan käytöstä yksinkertaisimmillaan. Request-luokasta tehdään olio new-avainsanan avulla. Luokan rakentajalle annetaan parametrina konfiguraatio-objekti. Ainoana vaadittuna kenttänä konfiguraatio-objektilla on request, joka sisältää varsinaisen funktion, jolla AJAX-pyyntö on tarkoitus tehdä. Tässä tapauksessa pyyntö on toteutettu Fetch-rajapintaa hyödyntäen.

```
import { Request } from 'react-with-requests';

const fetchMovies = new Request({
  request: async () => {
    const response = fetch('http://example.com/movies');
    const movies = await response.json();
    return movies;
  }
});
```

KUVA 14. AJAX-pyynnön määrittely Request-luokan avulla

Pelkkä Request-olion luominen ei suorita AJAX-pyyntöä. Request-funktio säilytetään olion muistissa ja suoritetaan vasta kun objektin execute-metodia kutsutaan. Kehittäjän ei itse ole tarkoitus kutsua tätä metodia, vaan ohjelmointikehyksen muut osat pitävät huolen metodin kutsumisesta oikealla hetkellä.

Kun palvelimelta haetaan yksittäistä resurssia, AJAX-pyynnön yhteydessä on annettava usein jokin tunnistetieto. Kuvan 15 esimerkissä haetaan yksittäisen elokuvan tiedot palvelimelta, jolloin request-metodi ottaa parametrina haettavan elokuvan id:n. Teknisesti tällä Request-objektilla ei kuitenkaan ole vielä kaikkea vaadittua tietoa pyynnön suorittamiseen.

```
const fetchMovie = new Request({
  request: async (id) => {
    const response = fetch(`http://example.com/movies/${id}`);
    const movie = await response.json();
    return movie;
  }
});
```

KUVA 15. Parametrien hyödyntäminen Request-objektissa

Mikäli kuvan 15 Request-objektilta kutsuttaisiin execute-metodia, se ei vielä tietäisi, millä parametreilla request-funktiota tulee kutsua. Jotta pyyntöjä voitaisiin vertailla, Request-olio on rakennettu siten, että se säilyttää muistissaan parametreina annettavat arvot. Näin ollen fetchMovie ei itsessään ole vielä toimiva Request-olio. Sen sijaan siltä löytyy sisäänrakennettu withParams-metodi, joka palauttaa uuden, toimivan Request-olion. Palautettu olio on edellisen kopio, mutta sillä on muistissaan annetut parametrit.

Kun sovellukselle määritellään Request-oliot, withParams-metodia ei ole vielä tarkoitus hyödyntää. WithParams-metodia käytetään vasta, kun komponenteille määritellään riippuvuuksia Request-olioihin. Näin ollen parametrit voivat olla riippuvaisia komponentin omista parametreista (props).

Koska Request-oliot säilyttävät muistissaan annetut parametrit, niiden tekemiä AJAX-pyyntöjä voidaan vertailla luotettavasti. Vertailua varten Request-olio laskee aina alustuksensa yhteydessä itselleen tiivisteen, joka on konfiguraatio-objektiin ja annettuihin parametreihin perustuva merkkijono. Kaikilta Request-olioilta löytyy equals-metodi, joka vertaa olion omaa tiivistettä annetun Request-olion tiivisteeseen.

Request-luokan konfiguraatio-objekti rakennettiin mahdollisimman helposti laajennettavaksi ja kehittäjäystävälliseksi. Objektin sisältö validoidaan aina Request-olion alustuksen yhteydessä, joten kehittäjä ei voi huomaamattaan syöttää virheellistä konfiguraatiota oliolle. Kuvan 16 konfiguraatio-objektissa esitellään kaikki opinnäytetyön kirjoittamisen hetkellä toteutetut asetukset.

```
const configuration = {
  request: () => {
    …
  },
  defaultMappings: {
    statusProp: 'movie',
    requestProp: 'fetchMovie',
    executeOnMount: true,
  },
  transformError: (originalError) => {
    if (originalError.statusCode === 404) {
      return 'This movie does not exist';
    } else {
      return 'Something went wrong';
    }
  }
};
```

KUVA 16. Request-luokan konfiguraatio-objekti

Kuvassa 16 konfiguraatioon on määritelty request-funktion lisäksi defaultMappings- ja transformError-kentät. DefaultMapping-kentän avulla määritellään oletusasetukset komponenteille, jotka tulevat määrittelemään riippuvuuden kyseiseen Request-olioon. Näistä oletusasetuksista kerrotaan lisää luvussa 5.3.4.

TransformError-kentällä voidaan määritellä Request-oliolle funktio, jonka läpi jokainen epäonnistunut AJAX-pyyntö ajetaan. Tämä ominaisuus mahdollistaa palvelimen antamien virheilmoitusten muuntamisen ihmisen luettavaan muotoon, jota komponentit voivat suoraan hyödyntää. Kuvassa 16 virheviesti muodostetaan palvelimen palauttaman tilakoodin perusteella.

5.3.2 RequestStateHandler: Yhteinen tilavarasto AJAX-pyynnöille

RequestStateHadler-luokka on React With Requests -kirjaston oma toteutus tilavarastosta, joka vastaa kaikkien sovelluksen AJAX-pyyntöjen tilanhallinnasta. Luokan toteutus on suhteellisen yksinkertainen. RequestStateHandler vastaa ainoastaan pyyntöjen tekemisestä, ja niiden tilan seuraamisen mahdollistamisesta kuuntelijoiden avulla.

Pyyntöjen tilavarasto on rakennettu muuttumattomaksi, eli sen sisältämiä objekteja ei koskaan teknisesti muokata. Tilan muuttuessa objektit luodaan aina uudestaan syvinä kopioina edellisistä. Myös luokan ulosviemät objektit ovat aina syviä kopioita, jotta tilaa ei

pystytä muokkaamaan tahattomasti luokan ulkopuolelta. Tämä suunnittelumalli teki tilanhallinnasta arvattavampaa, koska yllättäviä sivuvaikutuksia ei voinut aiheutua muutoksien seurauksena.

RequestStateHandler-olio alustetaan ilman parametreja. Alustuksen yhteydessä se luo tyhjän tilavaraston AJAX-pyyntöjen säilyttämistä varten. Pyyntöjä voidaan tehdä olion makeRequest-metodin avulla. Metodi ottaa parametrinaan Request-olion. Kuvassa 17 hyödynnetään fetchMovie-oliota, jonka withParams-metodia kutsutaan id:n määrittelemiseksi.

```
import { fetchMovie } from './requests';

const requestStateHandler = new RequestStateHandler();
const request = requestStateHandler.makeRequest(fetchMovie.withParams(1));

const response = await request.promise;
```

KUVA 17. AJAX-pyynnön suorittaminen RequestStateHandler-luokan avulla

Palautusarvona makeRequest-metodilta saadaan objekti, joka sisältää muun muassa pyynnölle generoidun uniikin id:n, Promise-objektin sekä pyynnön tilan. Varsinainen vastaus saadaan kuvan 17 tapauksessa odottamalla await-syntaksin avulla palautetun Promise-objektin valmistumista.

AJAX-pyyntöjen tilaa kuvataan tilavarastossa kolmella muuttujalla: loading (onko lataus kesken), error (virhe) ja result (palvelimen vastaus). Pyynnön tila muuttuu ainoastaan pyynnön onnistuessa tai päättyessä virheeseen. Tällöin muuttunut tila välitetään kaikille tilavaraston kuuntelijoille. Kuvan 18 esimerkissä näytetään, kuinka tilavarastoa voidaan kuunnella.

```
requestStateHandler.addStateChangeListener((newState) => {
  console.log('This is the new state', newState);
});

requestStateHandler.makeRequest(someRequest);
// -> This is the new state
// -> [{ id: 1, loading: true, error: null, result: null, ... }]
```

KUVA 18. Esimerkki tilavaraston kuuntelemisesta

Kuvassa 18 RequestStateHandler-oliolta kutsutaan addStateChangeListener-metodia, joka ottaa parametrina kuuntelijafunktion. RequestStateHandler pitää muistissaan kaikki kuuntelijafunktiot. Se kutsuu kuuntelijoita aina, kun tilavarastossa tapahtuu muutoksia. Kuuntelija saa parametrinaan aina tilavaraston uuden tilan, joka on tilaobjekteista muodostettu taulukko. Koodissa olevat kommentti-rivit kertovat, kuinka kuuntelija tulostaa konsoliin välittömästi pyynnön lähettämisen jälkeen tilavaraston sen hetkisen tilan.

Sekä Request- että RequestStateHandler-luokka ovat täysin riippumattomia Reactista, eli teoriassa niitä voisi hyödyntää myös muilla kirjastoilla toteutetuissa verkkosovelluksissa. React With Requests on rakennettu siten, että tilavaraston voisi myös suhteellisen helposti vaihtaa luokasta toiseen. Näiden luokkien tehtävänä on ainoastaan määritellä, suorittaa ja säilyttää AJAX-pyyntöjä. Seuraavassa luvussa tutustutaan näiden luokkien hyödyntämiseen React-sovelluksessa.

5.3.3 RequestStateProvider: Tilan jakaminen Context API:n avulla

Edellisissä luvuissa käsiteltiin AJAX-pyyntöjen määrittelyä ja tilanhallintaa sovelluksen käyttöliittymästä erillään, hyödyntäen Request- ja RequestStateHandler-luokkia. Toisin kuin edellä mainitut luokat, RequestStateProvider on React-komponentti. Sen ainoana tehtävänä on tarjota sovelluksen muille React-komponenteille mahdollisuus hyödyntää RequestStateHandler-luokan tarjoamaa tilavarastoa. Tämä tapahtuu Context-rajapinnan avulla.

Tyypillisesti React-sovelluksissa tila siirrettäisiin komponentilta komponentille parametrien (props) avulla. Tämä prosessi muuttuu kuitenkin haastavaksi, kun tilaa siirretään useita tasoja alaspäin. Reactin 16.3-versiossa julkaistu Context-rajapinta tarjoaa uuden

tavan tilan jakamiseen sovelluksessa. Rajapinnan avulla voidaan luoda uusi konteksti, jolla on Provider- ja Consumer-komponentit. Provider vastaa tilan tarjoamisesta alemmille tasoille komponentti-puuta, ja Consumer ottaa tarjotun tilan vastaan. (Nwamba, 2018.)

React With Requests -kirjasto toteuttaa oman metodinsa kontekstin luomiselle. Kuten kuvasta 19 nähdään, getRequestStateContext-metodi pitää huolen, että konteksti luodaan ainoastaan ensimmäisellä kutsulla. Seuraavat kutsut palauttavat aiemmin luodun kontekstin. Näin varmistetaan, että kaikki kirjaston Provider- ja Consumer-komponentit ovat kiinni samassa kontekstissa.

```
import React from 'react';

let requestStateContext;

export function getRequestStateContext() {
  if (requestStateContext) return requestStateContext;

  requestStateContext = React.createContext({});
  return requestStateContext;
}
```

KUVA 19. Kontekstin toteutus React With Requests -kirjastossa

RequestStateProvider-komponentin toteutus on todella yksinkertainen. Komponentin alustuksen yhteydessä luodaan uusi RequestStateHandler-olio tilavarastoksi AJAX-pyynnöillä. Tilavarastolle asetetaan kuuntelija, jonka vastuulla on pitää komponentin sisäinen tila synkronoituna tilavaraston kanssa. Komponentin render-metodissa piirretään ainoastaan kontekstin Provider-komponentti, jolle annetaan välitettäväksi arvoksi komponentin tila, joka on kuuntelijan ansiosta identtinen tilavaraston kanssa. Lisäksi kontekstin avulla välitetään referenssi itse RequestStateHandler-olioon, joka mahdollistaa pyyntöjen suorittamisen Consumer-komponentista käsin.

Sovellus pääsee hyödyntämään tilavarastoa, kun RequestStateProvider-komponentti on ylimmällä tasolla sovelluksen komponentti-puussa. Tilaa voidaan hyödyntää kontekstin tarjoamien Consumer-komponenttien avulla. Seuraavassa luvussa kerrotaan RequestStateConsumer-komponentista, joka perustuu kontekstin Consumer-komponenttiin.

5.3.4 RequestStateConsumer: Tilan yhdistäminen komponentteihin

Kuten edellisessä luvussa kerrottiin, kontekstilla on sekä Provider- että Consumer-komponentti. RequestStateConsumer on React With Requests -kirjaston oma toteutus Consumer-komponentin ympärille. Se antaa komponenteille mahdollisuuden määritellä riippuvuuksia AJAX-pyyntöihin.

RequestStateConsumer ottaa parametrinaan funktion, joka määrittelee, mitä AJAX-pyyntöjä komponentti haluaa hyödyntää. Funktio saa argumenttina komponentin parametrit (props) ja palauttaa niiden pohjalta määrittely-taulukon. RequestStateConsumer pitää huolen, että pyyntö tehdään oikeaan aikaan, ja sen tila on saatavilla render prop -arvona. Render prop -malli tarkoittaa sitä, että komponentin lapseksi (tai render-parametriksi) annetaan funktio, joka saa argumentteinaan komponentin tarjoaman tilan (Jackson 2017). Kuvassa 20 RequestStateConsumer-komponenttia hyödynnetään datan hakemiseen MovieList-komponentille.

```
import { RequestStateConsumer } from 'react-with-requests';
import { fetchMovies } from './api';

const requestMapping = (props) => [
  {
    request: fetchMovies,
    executeOnMount: true,
    statusProp: 'movies',
    requestProp: 'fetchMovies'
  }
];

const MovieList = () => (
  <RequestStateConsumer requests={requestMapping}>
    {({ movies, fetchMovies }) =>
      // render list of movies here
    }
  </RequestStateConsumer>
);
```

KUVA 20. Request-olioiden määrittely RequestStateConsumer-komponentille

Kuvassa 20 määritellään requestMapping-funktion avulla riippuvuus fetchMovies-pyyntöön. ExecuteOnMount-kenttä kertoo, että pyyntö tulee suorittaa heti, kun komponentti upotetaan DOM:iin. Pyynnön tila tulee saataville parametrissa movies, ja se voidaan tarvittaessa lähettää uudelleen fetchMovies-parametrin sisältämällä funktiolla. Määritykset

annetaan RequestStateConsumer-komponentille requests-parametrina. Tämän jälkeen pyynnön tilaa voidaan hyödyntää render prop -mallin avulla.

Request-oliolla voi olla myös oletusmääritykset (konfiguraatio-objektin defaultMappings-kenttä). Tällöin taulukkoon voidaan lisätä määrittelyobjektin sijasta pelkkä Request-olio. Request-olion oletusmääritykset korvataan, mikäli sille on asetettu eriäviä määrityksiä taulukossa.

Kuvan 20 requestMapping-funktio ei hyödyntänyt props-argumenttia. Koska requestMapping on funktio, props-argumentin avulla riippuvuudet voitaisiin kustomoida komponentin parametrien (props) perusteella. Tämä on erityisen hyödyllistä esimerkiksi yksittäisen elokuvan sivulla, jonka sisältö vaihtuu elokuvan id:n perusteella. Kuva 21 havainnollistaa, kuinka komponentin parametreja voidaan hyödyntää pyyntöjen määrittelyssä.

```
import { fetchMovie } from './api';

const requestMapping = (props) => ([
  fetchMovie.withParams(props.match.params.id)
]);
```

KUVA 22. Pyynnön määrittely parametrien perusteella

Kuvassa 22 pyyntö määritellään suoraan Request-oliona, koska sen konfiguraatio sisältää jo oletusmääritykset. Tällä kertaa props-argumenttia hyödynnetään pyynnön määrittelyssä. React Router -kirjasto tarjoaa kaikille sovelluksen komponenteille parametreja liittyen sovelluksen reititykseen. Näistä parametreista saadaan elokuvan id, joka on niin sanottu URL-osoitteen hakuparametri (englanniksi query parameter). Hakuparametrien vaihtuessa RequestStateConsumer kutsuu automaattisesti requestMapping-funktiota uusilla parametreilla selvittääkseen, täytyykö sen tehdä pyyntö uuteen osoitteeseen.

Jacksonin (2017) mielestä render prop -malli voisi korvata ylemmän tason komponentit kokonaan. Ylemmän tason komponentit ovat kuitenkin olleet osa React-ekosysteemiä pidempään, ja ne ovat tutumpi vaihtoehto monille kehittäjille. Tästä syystä kirjastoon toteutettiin myös withRequests-funktio, joka mahdollistaa AJAX-pyyntöjen määrittelyn ylemmän tason komponentin avulla. Kuvassa 23 esitellään, kuinka withRequests-funktiota voidaan hyödyntää.

```
import { withRequests } from 'react-with-requests';
import { fetchMovies } from './api';

const MovieList = (props) => {
  const { loading, error, result } = props.movies;

  if (loading) return <Spinner />;
  if (error) return <Alert message={error} />

  return (
    <div>
      <ul>
        {
          result.map(movie => <li>{movie.name}</li>)
        }
      </ul>

      <button onClick={props.fetchMovies}>
        Refetch
      </button>
    </div>
  );
};

const requests = (props) => ([ fetchMovies ]);
export default withRequests(requests)(MovieList);
```

KUVA 23. WithRequests-funktion hyödyntäminen

Kun AJAX-pyyntöjen tekemiseen liittyvä logiikka siirretään ylemmälle tasolle, komponenteista voidaan usein tehdä tilattomia. Kuvassa 23 MovieList on toteutettu tilattomana komponenttina. MovieList-komponentti saa parametreinaan pyynnön tilan, ja piirtää sen perusteella näkymän. Näkymää piirrettäessä on aina tärkeää ottaa huomioon, että pyyntö voi olla vielä kesken, tai sitä tehdessä on saattanut tapahtua virhe. MovieList-komponentti tarkistaakin ensimmäiseksi, tarvitseeko sen piirtää latausindikaattori tai virheviesti. Vasta tämän jälkeen komponentti piirtää listan elokuvia. Pyynnön tilan lisäksi komponentti hyödyntää kirjaston tarjoamaa uudelleenlatausfunktiota fetchMovies, jota kutsutaan, jos listan alla olevaa painiketta klikataan.

Pyyntö-määritykset tehdään kuvan 23 kahdella viimeisellä rivillä. WithRequests-funktiolle annetaan requests-argumenttina funktio, joka määrittelee komponentin riippuvaiseksi fetchMovies nimisestä Request-oliosta. Tiedostosta ei viedä ulos itse MovieList komponenttia, vaan fetchMovies-funktiolla luotu ylemmän tason komponentti. Ylemmän

tason komponentti pitää huolen, että oikea AJAX-pyyntö suoritetaan aina, kun komponentti upotetaan DOM:iin.

RequestStateConsumer-komponentti ja withRequests-funktio mahdollistavat varsinaisten käyttöliittymäkomponenttien yksinkertaisemman toteutuksen. Kehittäjän ei tarvitse kiinnittää huomiota muuhun kuin pyyntöjen tilan pohjalta näkymien piirtämiseen. Tämä vähentää huomattavasti kirjoitettavan koodin määrää React-sovelluksia kehittäessä.

5.4 Havaitut ongelmat

Kirjaston kehittäminen ei ollut helppo tehtävä. JavaScript on dynaamisesti tyypitetty kieli, joka mahdollistaa monia asioita. Se kuitenkin poistaa samalla varmuuden kehittämisestä, joka on vahvasti tyypitettyjen kielien, kuten Javan, vahvuus. Suhteellisen monimutkaisen logiikan ja useista funktioista koostuvien komponenttien rakentaminen ilman tyypitystä osoittautui erittäin hankalaksi. Ilman kattavia yksikkö- ja integraatiotestejä kirjaston rakentaminen olisi ollut lähes mahdotonta.

Ongelman olisi voinut ratkaista TypeScript, joka on Microsoftin kehittämä tyypitetty versio JavaScriptistä. Tyypitys ei ole vahva, toisinkuin Javassa, sillä TypeScript käännetään suoraan JavaScriptiksi. Ennen käännöstä TypeScript-kääntäjä pystyy kuitenkin tarkistamaan, ettei ohjelmassa ole selkeitä tyyppivirheitä. Tyypityksien vahvuus riippuu hyvin pitkälti siitä, kuinka paljon tyypityksien tekoon käytetään aikaa. Oikein tyypitettynä kirjaston kehittäminen olisi varmasti ollut huomattavasti helpompaa ja nopeampaa.

Kehitystyökalujen lisäksi React With Requests- kirjaston toteutuksessa on muutamia ongelmia. Alun perin suunnitelmana oli kehittää kirjastoon välimuisti AJAX-pyynnöille, jotta samaa pyyntöä ei tarvitsisi välttämättä koskaan tehdä uudestaan. Vastaus voitaisiin antaa välimuistista komponenteille ilman viivettä. Toteutuksen loppuvaiheilla välimuisti päätettiin kuitenkin jättää toteuttamatta. Tähän johti ajatus siitä, että välimuistin käsittely voitaisiin jättää selaimen vastuulle.

Koska keskitettyä välimuistia ei päädytty toteuttamaan, kirjasto olisi voitu toteuttaa kokonaan ilman Context-rajapintaa ja keskitettyä tilavarastoa. Kaikki logiikka olisi voitu

sisällyttää yhteen ylemmän tason komponenttiin, ja kirjaston käyttäminen olisi ollut asteen yksinkertaisempaa. Toisaalta sovelluksen arkkitehtuuri pysyi helposti laajennettavana useisiin komponentteihin jaetun logiikan ansiosta, ja keskitetty välimuisti voidaan vielä toteuttaa tulevaisuudessa.

Context-rajapinnan hyödyntämisen takia esille nousi myös toinen ongelma. Keskitetyn tilan hyödyntämiseen tarvitaan aina Consumer-komponentti, jonka päälle rakennettiin vielä kirjaston oma toteutus, RequestStateConsumer. Varsinaisen käyttöliittymäkomponentin ympärille tulee näin useita tasoja ympäröiviä komponentteja. Tämä voi osoittautua ongelmaksi, jos React With Requests -kirjastolla toteutetun sovelluksen komponentti-puuta halutaan tutkia virheiden löytämiseksi. Useiden ympäröivien komponenttien joukosta on usein vaikeaa löytää varsinaista käyttöliittymä-komponenttia, jota oikeasti haluttaisiin tarkastella.

6 POHDINTA

Opinnäytetyön tavoitteena oli selvittää, kuinka AJAX-pyyntöjä tulisi tehdä React-sovelluksessa, ja miten pyyntöjen tilanhallintaa voitaisiin helpottaa. Tarkoituksena oli tutustua olemassa oleviin ratkaisuihin, ja kehittää lisäksi avoimen lähdekoodin kirjasto vastaamaan nykyisten menetelmien ongelmiin.

AJAX-pyyntöjen tekemiseen ja tilanhallintaan löydettiin useita toimivia menetelmiä. Opinnäytetyössä esiteltiin sekä perinteinen tapa tehdä AJAX-pyyntöjä suoraan komponenteista käsin, että ylemmän tason komponenttien hyödyntäminen pyyntöjen tekemiseen. Lisäksi tutustuttiin React Suspense -ominaisuuteen, joka tulee jatkossa helpottamaan käyttöliittymän piirtämistä AJAX-pyyntöjen tilan pohjalta.

Ylemmän tason komponentit havaittiin hyväksi tavaksi keskittää pyyntöjen tekemiseen liittyvä logiikka yhteen komponenttiin. Jotta tätä logiikkaa ei tarvitsisi rakentaa jokaista sovellusta varten erikseen, osana opinnäytetyötä toteutettiin React With Requests -ohjelmointikirjasto. Se tarjoaa työkalut pyyntöjen määrittelyyn ja hyödyntämiseen React-sovelluksissa. Kirjasto julkaistiin npm-pakettirekisterissä, ja se on asennettu opinnäytetyön kirjoittamisen hetkellä jo yli kahteensataan projektiin. Voidaan siis olettaa, että kirjasto on vastannut oikeaan tarpeeseen.

React With Requests -kirjasto päätettiin toteuttaa kokonaan EcmaScriptin kuudetta versiota (ES6) hyödyntäen, vaikka se ei ole tuettuna lainkaan esimerkiksi Internet Explorer -selaimissa. Päätökseen vaikutti erityisesti ES6-version tarjoama luokka-syntaksi, jota kirjastossa hyödynnettiin. Ilman ES6-standardin tarjoamia ominaisuuksia lähdekoodia oltaisiin jouduttu kirjoittamaan enemmän, ja siitä olisi tullut huomattavasti vaikealukuisempaa.

Koska kirjaston haluttiin tukevan myös vanhempia selaimia, projektissa otettiin käyttöön Babel-työkalu. Babelin avulla ES6-standardin avulla kirjoitettu koodi saatiin muunnettua ES5-standardin mukaiseksi. Lähdekoodista oli siis olemassa yhtä aikaa sekä ES6- että ES5-yhteensopiva versio. Npm-pakettirekisteriin jaettiin ES5-yhteensopiva versio, jotta kirjastoa hyödyntävät sovellukset toimisivat suoraan useimmilla selaimilla.

Kirjaston yksikkö- ja integraatiotestit kehitettiin hyödyntäen Jest- ja Enzyme-kirjastoja. Integraatiotesteillä testattiin kirjaston todellisia käyttötarkoituksia. Virtuaalisille komponenteille määriteltiin riippuvuuksia erilaisiin resursseihin, ja testit tarkastivat, että komponentit saivat heti käyttöönsä oikeiden resurssien tilan. Mielenkiintoinen huomio oli, että testit sisälsivät lopulta jopa hieman enemmän koodia kuin itse kirjaston varsinaiset luokat.

Testien ansiosta varsinainen kehitystyö oli nopeampaa ja varmempaa, koska manuaalinen testaaminen selaimessa voitiin jättää kokonaan pois. Manuaalinen testaaminen olisi ollut tässä tapauksessa todella hidasta, johtuen kirjaston monimutkaisesta logiikasta ja erilaisten testattavien skenaarioiden suuresta määrästä. Testit ajettiin jokaisen koodimuutoksen jälkeen automaattisesti. Koodin kirjoittaminen oli edelleen haasteellista, mutta ainakin testit paljastivat heti mahdolliset virheet.

Kriittisesti ajateltuna React With Requests -kirjastosta ei tullut täydellinen. Sen dokumentaatio on vielä puutteellinen, joten ulkopuolisen on vaikea ottaa kirjaston tarjoamista ominaisuuksista kaikki hyöty irti. Myös kirjaston jatkokehitys on suhteellisen vaikeaa JavaScriptin dynaamisen luonteen vuoksi. Tähän olisi voinut auttaa TypeScriptin tarjoama staattinen tyypitys, mutta sitä ei ymmärretty ottaa mukaan projektia aloittaessa. Jatkokehitystä auttavat kuitenkin huomattavasti kirjaston laajat yksikkö- ja integraatiotestit.

React Suspense -ominaisuus tulee tarjoamaan täysin uuden tavan hyödyntää AJAX-pyyntöjä React-komponenteissa. Uuden React-version myötä se tulee ainakin osittain poistamaan tarpeen React With Requests -kirjastolle. Tämä on opinnäytetyön tekijän mielestä erittäin hyvä asia, sillä React ekosysteemi on jo pitkään kaivannut yhtenäistä tapaa hallita AJAX-pyyntöjä. Toisin kuin tarkoitukseen kehitetyt kolmansien osapuolien kirjastot, React Suspense ei tule olemaan niin sanotusti päälle liimattu ratkaisu, vaan luonnollisesti toimiva osa Reactia.

Kaiken kaikkiaan opinnäytetyön voidaan katsoa onnistuneen hyvin. Sen avulla saatiin paljon tietoa erilaisista menetelmistä AJAX-pyyntöjen tekemiseen ja tilanhallintaan, ja samalla valmistauduttiin Reactin seuraavan version tuomiin uudistuksiin. React With Requests vastasi oikeaan tarpeeseen, vaikka sen tuomat hyödyt vähenevätkin huomattavasti seuraavassa React-versiossa.

LÄHTEET

Banks, A. ja Porcello, E. 2017. Learning React. O'Reilly Media, Inc.

Bertoli, M. 2017. React Design Patterns and Best Practices. Packt Publishing.

Boduch, A. 2017. React and React Native. Packt Publishing.

Ceddia, D. 2018. AJAX Requests in React: How and Where to Fetch Data. Luettu 20.8.2018. https://daveceddia.com/ajax-requests-in-react/

Ezell, W. 2018. Everything You Need to Know About Single Page Applications. Luettu 2.7.2018 https://dotcms.com/blog/post/everything-you-need-to-know-about-single-page-applications

House, C. 2016. React Stateless Functional Components: Nine Wins You Might Have Overlooked. Luettu 9.7.2018. https://hackernoon.com/react-stateless-functional-components-nine-wins-you-might-have-overlooked-997b0d933dbc

Jackson, M. 2017. Use a Render Prop! Luettu 5.8.2018. https://cdb.reacttraining.com/use-a-render-prop-50de598f11ce

Skólski, P. 2016. Single-page application vs. multiple-page application. Luettu 2.7.2018 https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58

Nwamba, C. 2018. Working with the new React Context API. Luettu 3.8.2018. https://blog.pusher.com/new-react-context-api/

React. N.d. React.Component. Luettu 15.7.2018. https://reactjs.org/docs/react-component.html

React. N.d. Components and Props. Luettu 20.8.2018. https://reactjs.org/docs/components-and-props.html

Vieira, S. 2017. How to Use the JavaScript Fetch API to Get Data. Luettu 10.7.2018. https://scotch.io/tutorials/how-to-use-the-javascript-fetch-api-to-get-data

Wasson, M. 2013. ASP.NET - Single-Page Applications: Build Modern, Responsive Web Apps with ASP.NET. Luettu 23.7.2018. https://msdn.microsoft.com/en-us/magazine/dn463786.aspx