Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Aku Kangas

Web-sovelluksen työkalut, kirjastot ja menetelmät React.js-projektissa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

27.5.2020

Tekijä Otsikko Sivumäärä Aika	Aku Kangas Web-sovelluksen työkalut, kirjastot ja menetelmät React.js-projektissa. 31 sivua 27.5.2020
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Lehtori Simo Silander Projektipäällikkö Kalle Sonninen

Insinöörityö muodostui työnantajan tarpeesta kehittää web-sovellukselle uusi käyttöliittymä. Opinnäytetyö käsittelee geneerisellä tasolla web-sovelluksen kehitys- ja ohjelmointiympäristöä. Työn lähtökohtana oli React-pohjaisen web-sovelluksen toteutus, jolle tarvittiin kehitys- ja ohjelmointiympäristö. Kehitysprojektille ei ollut määritelty rajoitteita, vaan sain itse valita ja tarkastella sopivia teknologioita.

Työn tavoitteena oli luoda ympäristöstä riippumaton ja useissa tuotteissa uudelleen käytettävä projektipohja. Ympäristöstä haluttiin tehdä mahdollisimman yksinkertainen ottaa käyttöön missä tahansa kehitysvaiheessa. Tavoitteeksi asetettiin ns. oneliner-komento, eli koko ympäristö tulisi asentaa ja alustaa yhdellä komennolla. Projektille etsittiin myös kehitystyötä nopeuttavia avoimen lähdekoodin kirjastoja. Kirjastoja tarvittiin niin käyttöliittymän toteutukseen, kuten myös sovelluksen tilan ja lomakkeiden logiikan hallintaan sekä projektin automatisointiin. Lisäksi selvitin myös muutamia Reactille ominaisia suunnittelumalleja.

Projekti perustettiin alustariippumattoman Node.js- JavaScript-tulkin päälle, ja se noudattaa npm-paketin mallia. Npm mahdollistaa koko projektin ”oneliner”-asennuksen. Avoimen lähdekoodin kirjastoista päädyin muun muassa käyttöliittymän osalta Material-ui-kirjastoon, joka sisältää useita valmiita käyttöliittymä komponentteja ja työkaluja oman ulkoasun luomiseen. Lomakkeiden hallintaan valitsin Formik-kirjaston, jossa on sisäänrakennettuna kaikki tarpeellinen lomakkeiden käsittelyyn. Sovelluksen tilanhallintaan valikoitui Reduxin päälle rakennettu easy-peasy-kirjasto. Insinöörityön aiheena käytetty projekti saavutti sille toivotut tavoitteet.

Avainsanat	Web-sovellus, avoin lähdekoodi, Node.js, npm, React

Author	Aku Kangas
Title	Web Application Tools, Libraries and Methods in React.js Project
Number of Pages	31 pages
Date	27th of May 2020
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Software Engineering
Instructors	Simo Silander, Senior Lecturer Kalle Sonninen, Project Manager

This Bachelor's Thesis derived from the employer's need to develop a new user interface for a web application. The thesis deals with the web application development and programming environment at a generic level. The starting point of the study was the implementation of a React-based web application, which required a development and programming environment. No restrictions had been set for the development project, so it was possible to choose and look at suitable technologies without limitations.

The aim of the thesis was to create a project base that is independent of the environment and reusable in several products. The aim was to make the environment as simple to use as possible at any stage of product development. The goal was to set the so-called one-liner command, meaning the entire environment should be installed and formatted with a single command. The study also discusses open source libraries to speed up development work. Libraries were needed to implement the user interface, as well as to manage the logic of the application status and to automate the project tasks. In addition, a few design models specific to React are also introduced.

The project was created on a platform independent Node.js JavaScript interpreter and it follows the model of the npm package. Npm enables the "oneliner" installation of the entire project. From the open source libraries, the Material-ui library was chosen, in terms of the user interface, containing several ready-made user interface components and tools for creating a design system. To manage the forms, the Formik library was chosen including everything needed to process forms. An easy-peasy library built on top of Redux was selected for the state management of the application.

Keywords	Web application, open source, Node.js, npm, React

Sisällys

Lyhenteet

1 Johdanto 1

2 Kehitys- ja ohjelmointiympäristö 2

3 Create-react-app 3

3.1.1 Projektin perustaminen 3
3.1.2 react-scripts 4
3.1.3 Kritiikki 5

4 Avoimen lähdekoodin kirjastot 6

4.1 Koodin ulkoasun formatointi 6
4.2 Käyttöliittymä & käyttäjäkokemus 7
4.2.1 React 7
4.2.2 Material-ui 8
4.2.3 notistack 10
4.2.4 Formik 11
4.3 Tilanhallinta 13
4.4 Kommunikointi (http) 15
4.5 Lokalisointi 16

5 Suunnittelumallit 18

5.1 Higher-Order Component 18
5.2 Render prop 20
5.3 Container componet 21
5.4 Hooks (react >= 16.8.0) 23

6 Testaus 25

7 Tuotantoversio 26

7.1 Koonti (build) 26
7.2 Julkaisu 27

8 Yhteenveto 29

Lähteet 30

Lyhenteet

CRA	Create React App. Työkalu React-projektin perustamiseen yhdellä komennolla.
ES	ECMAScript-määritys. JavaScriptin standardisoitu määritelmä.
SPA	Single Page Application. Yhden sivun sovellus on verkkosovellus tai verkkosivusto, jossa koko sivuston rakenne ladataan yhdellä kertaa.
CLI	Command-line interface. Tietokoneohjelman komentoliittymä tai komentorivi.
CSS	Cascading Style Sheets. WWW-dokumenttien kanssa käytettävä tyyliohje.
SCSS	Sass, CSS-tyyliohjeen laajennus syntaksi.
VDOM	VirtualDOM. Virtuaali DOM on ohjelmointikonsepti, jossa käyttöliittymän ideaali tai ”virtuaali” esitys pidetään muistissa ja synkronoidaan ”todellisen” DOM: n kanssa.
HTTP	Hypertext Transfer Protocol. Tiedonsiirtoprotokolla jonka avulla www-selaimet ja -palvelimet kommunikoivat.
IIFE	Immediately-invoked Function Expression. Tilanne jolloin funktiota kutsutaan välittömästi sen luomisen jälkeen.

1 Johdanto

Tässä insinöörityössä perehdyn web-sovelluksen kehityksessä käytettävien aputyökalujen ja kirjastojen käyttöön. Insinöörityön tilaaja Celebris Technology on osa Celebris-konsernia, ja sen tehtävä on kehittää ja ylläpitää palveluita muille konserniin kuuluville yrityksille. Insinöörityön aihe syntyi tilaajan tarpeesta kehittää web-sovellukselle uusi käyttöliittymä, jossa käytettiin Facebookin kehittämää React-kirjastoa. Työkalujen ja kirjastojen lisäksi selvitin muutamia Reactin kanssa hyväksi todettuja suunnittelumalleja ja tekniikoita. Näiden pohjalta sovittiin myös kehitystiimin kanssa yhteiset pelisäännöt.

Modernissa web-sovellusprojektissa käytetään nykypäivänä lukuisia erilaisia aputyökaluja, joilla pyritään helpottamaan ja automatisoimaan kehittäjän työvaiheita, parantamaan projektin laatua tai analysoimaan sen lähdekoodia. Työvaiheiden helpottamisella voidaan tarkoittaa esimerkiksi sitä, että useisiin triviaaleihin ongelmiin löytyy usein jonkin valmis avoimen lähdekoodin ratkaisu eli niin kutsuttu kirjasto. Työvaiheiden automatisoinnilla pyritään yleensä parantamaan projektin laatua. Olipa työvaihe kuinka helppo tai yksinkertainen tahansa, kannattaa se automatisoida, jos se vain on mahdollista. Esimerkiksi on täysin inhimillistä unohtaa manuaalinen testien suoritus.

Lähdekoodin analysoinnilla voidaan tarkoittaa staattista analyysiä eli lähdekoodista etsitään mahdollisia virheitä tai antimalleja, jotka paljastuvat jo pelkästään koodia lukemalla. Tällaiset työkalut ovat yleensä käytössä aina, kun varsinaista kehitystyötä tehdään, jotta mahdolliset virheet paljastuisivat heti.

Minkään web-sovelluksen kehitys ei kuitenkaan vaadi kehittäjää käyttämään mitään edellä mainitun kaltaisia aputyökaluja, mutta niiden tarjoamat edut ovat niin merkittäviä, että muutamista työkaluista on muodostunut alalle ns. de facto.

Insinöörityö käsittelee projektia anonyymisti geneerisellä tasolla, eli projektin varsinaista tuotetta tai sen lähdekoodia en tässä työssä käsittele. Kaikki esimerkkikoodit jäljittelevät varsinaista toteutusta. Projekti toteutettiin tavanomaisilla web-tekniikoilla. Suurimmilta osin lähdekoodi on JavaScriptiä. Sen versionhallinnassa käytössä oli Git, ja kehitys- ja tuotantoympäristö vaativat toimiakseen Node.js:n sekä npm-paketinhallinnan.

2 Kehitys- ja ohjelmointiympäristö

Meillä jokaisella on omat mieltymykset ja mielipiteet työkalujen suhteen, joten ei ole lainkaan poikkeuksellista, etteivät kaikki pidä samoista työkaluista. Projektin ohjelmointiympäristön vaatimusten ei tulisi asettaa kovinkaan suuria rajoitteita kehittäjälle esimerkiksi käyttöjärjestelmän ja teksti- tai koodieditorin valinnassa. Ohjelmointiympäristöllä tarkoitan kehittäjän fyysistä työkalua eli tietokonetta ja sille asennettavia pakollisia sovelluksia. Ohjelmointiympäristö määrittää hyvin pitkälti, miten kehitysympäristö on mahdollista toteuttaa.

Yksinkertaisimmillaan ohjelmointiympäristö voi koostua tekstieditorista ja ohjelmointikielen kääntäjästä tai tulkista. Tällaisessa ympäristössä kaikki työvaiheet ovat manuaalisia ja vievät aikaa varsinaiselta kehitystyöltä. Web-sovelluksen kehittämisessä yksi esimerkki tällaisesta manuaalisesta työvaiheesta on selainikkunan virkistäminen, jotta lähdekoodin muutokset voidaan nähdä selaimessa. Lisäksi projektin lähdekoodi on käännettävä ennen kuin se voidaan antaa selaimen suoritettavaksi, koska Reactin kanssa käytetty JSX-syntaksilaajennos ei ole validia JavaScriptiä.

Projektin ohjelmointiympäristön vaatimusten tulisi olla suppeat, mutta samalla kehitysympäristön pitäisi pystyä suorittamaan automatisoituja tehtäviä, jotka nopeuttavat, selkeyttävät ja helpottavat kehittäjän työtä.

Kehitysympäristö perustettiin alustariippumattoman Node.js JavaScript -tulkin päälle. Node.js:n asennuksen mukana toimitetaan npm-cli (1), joka mahdollistaa npm-paketinhallinnan käytön projektissa. Kaikki projektin käyttämät lisätyökalut ja kirjastot ladattiin ja asennettiin npm-rekisteristä. Kehitysympäristö noudattaa npm-paketin mukaista mallia, eli sen toiminnot suoritetaan npm-cli:n kautta. Npm-perspektiivistä puhuttaessa kaikkia kirjastoja, työkaluja tai itse projektia kutsutaan paketiksi. Projektin ainoa vaatimus ohjelmointiympäristölle on Node.js, ja se on saatavilla Windows-, macOS- ja Linux-alustoille.

3 Create-react-app

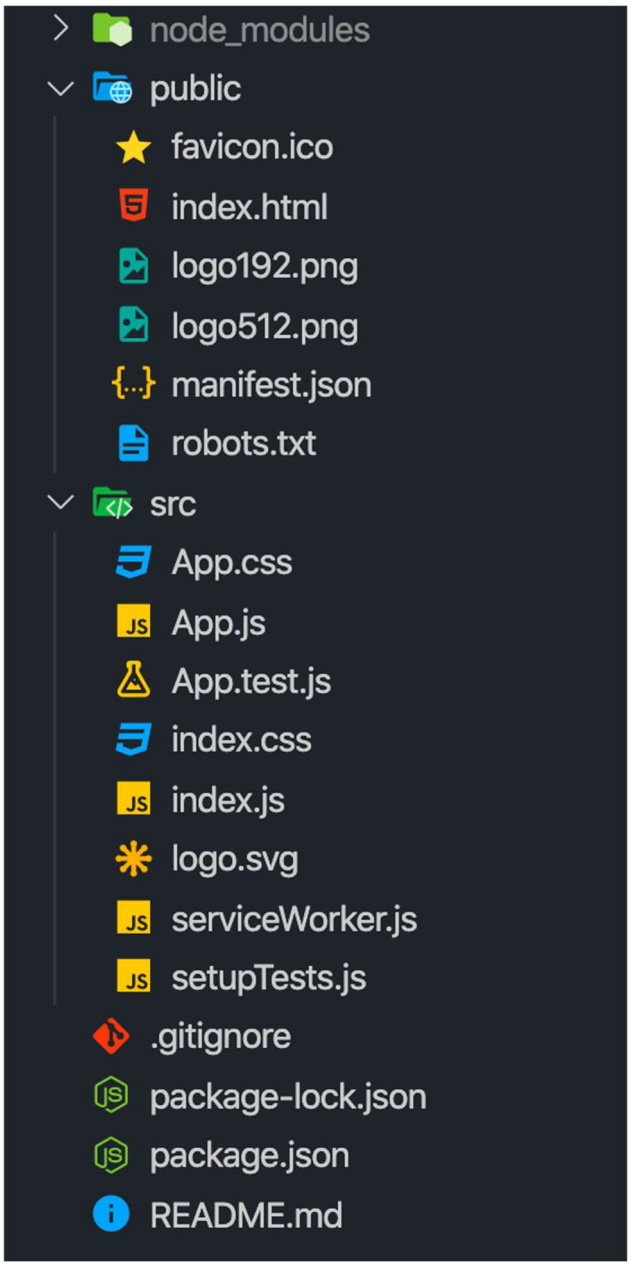
Kehittäjän kirjoittamaa React-koodia ei voida suorittaa sellaisenaan, koska Reactin kanssa on usein käytössä JSX-syntaksilaajennos, joka ei ole validia JavaScriptiä (2). Tämän vuoksi React-koodi täytyy kääntää Bable-kääntäjällä. Mikäli koodia ei haluta kääntää komentoriviltä manuaalisesti, tarvitaan tämän lisäksi jonkinlainen automatisoitu prosessi.

Webpack tarjoaa valmiit toteutukset paikalliselle kehityspalvelimelle, koodin jakamiselle sekä tuotantoversion pakkaamiseen. Lisäksi se tarjoa rajapinnan liitännäisille, jotka mahdollistavat useiden työkalujen automatisoinnin, kuten Babelin, jota Reactin kanssa tarvitaan. Webpack vaatii toimiakseen konfiguraation, ja useiden liitännäisten päivittäminen pitkällä aikavälillä voi muodostua haastavaksi, koska liitännäiset ovat usein hyvin riippuvaisia toisistaan.

Reactille on myös kokonaan valmis työkalu projektin perustamiseen. Create-react-appissä (CRA) kaikki tarpeellinen on pakattu yhteen komentoriviltä ajettavaan työkaluun. Se luo projektille pohjan eli muodostaa kansiorakenteen ja lataa kaikki tarvittavat paketit asennuksen yhteydessä, ilman erillistä konfiguraatiota. Luotu projekti käyttää react-scripts-kirjastoa, jonka sisään on pakattu koko Webpack-konfiguraatio ja sen käyttämät liitännäiset. Create-react-app on suositeltu ja ideaali ratkaisu SPA-tyyppisen web-sovelluksen pohjaksi. (3.)

3.1.1 Projektin perustaminen

Create-react-app on saatavilla npm-rekisteristä, ja se voidaan suorittaa npx:n avulla ilman erillistä asennusta. Npx on työkalu npm-pakettien suorittamiseen ilman paikallista asennusta. Kuva 1 esittää Create-react-appillä luodun projektin pohjan.



Kuva 1. Create-react-app-projektin hakemistorakenne asennuksen jälkeen

3.1.2 react-scripts

Kun Create-react-app on ladannut ja luonut projektin pohjan, on koko projektin konfiguraatio pakattu react-scripts-kirjastoon. Tästä eteenpäin projektin käytöstä vastaa react-scripts-kirjasto. Sen etu verrattuna perinteiseen ja itse luotuun Webpack-konfiguraatioon on siinä, että react-scripts-kirjastoa kehittää ja ylläpitää yhteisö. Kirjasto voidaan tulevaisuudessa päivittää npm-cli:n kautta.

Kirjasto tarjoaa neljä eri komentoa

- Paikallinen kehityspalvelin käynnistyy komennolla start. Projekti on tämän jälkeen nähtävillä osoittessa http://localhost:3000/ ja se päivittyy automaattisesti, kun projektin lähdekoodia muokataan. Staattisen analyysin virheilmoitukset näkyvät konsolissa.
- Testaus käynnistyy komennolla test. Testauskomento on interaktiivinen ja testiajo suoritetaan aina, kun projektin lähdekoodia muokataan.
- Tuotantoversion pakkaaminen/koostaminen tapahtuu komennolla build. Komento luo projektin juureen kansion nimeltä build.
- Kirjasto voi myös purkaa itsensä komennolla eject. Komento on yksisuuntainen, ja sen peruuttaminen ei ole mahdollista myöhemmin.

3.1.3 Kritiikki

Create-react-app-ongelmat tulevat esille yleensä siinä vaiheessa, kun sillä luodun projektin Webpack-konfiguraatiota pitäisi muokata. Projektin koko konfiguraatio on pakattu yhteen kirjastoon, eikä se tarjoa mitään virallista tukea tai rajapintaa liitännäisille. React-scripts mahdollistaa niin kutsutun eject-toiminnon, eli kirjasto purkaa itsensä perinteisen Webpack-konfiguraation muotoon. Purkamisen jälkeinen konfiguraatio on todella laaja ja kompleksinen. Tällaisen konfiguraation ylläpito saattaa viedä paljon aikaa itse sovelluksen kehittämisestä.

Create-react-app-dokumentaatio suosittelee eject-toiminnon sijaan luomaan projektin lähdekoodista oman haaran (git fork). Tarvittavat kustomoinnit tehdään ”forkattuun” haaraan ja tulevaisuudessa react-scripts-päivitykset voidaan liittää (git merge) suoraan omaan haaraan.

4 Avoimen lähdekoodin kirjastot

Minkä tahansa sovelluksen kehittäminen vaatii aina aikaa, josta muodostuu poikkeuksetta kustannuksia. Näitä kustannuksia voidaan kaventaa muun muassa hyödyntämällä avoimen lähdekoodin toteutuksia. Avointa lähdekoodia saa kuka tahansa käyttää, muokata ja lisensoinnista riippuen myös jakaa ilmaiseksi.

Mikäli jonkin tietyn ongelman ratkaisemiseksi on jo olemassa valmis toteutus eikä sen käytöstä muodostu muita esteitä kuten yhteensopivuus- tai lisenssiongelmia. Tällaisen toteutuksen käyttöönottoa kannattaa harkita.

Avoin lähdekoodi tuo myös mukanaan muutamia riskejä. Huolimattomasti toteutettu koodi voi muodostaa suorituskykyongelmia tai vaikuttaa koko järjestelmän vakauteen. Lisäksi avoimen lähdekoodin projekteissa on tavattu haittaohjelmia (4). Näiden seikkojen vuoksi jokaista käyttöön otettavaa avoimen lähdekoodin kirjastoa tai työkalua tulee arvioida ja tutkia huolellisesti ennen sen käyttöönottoa.

4.1 Koodin ulkoasun formatointi

Versionhallinnan keskeinen ominaisuus on pitää kirjaa projektin lähdekoodin muutoksista. Turhien muutosten vieminen versionhallintaa vaikeuttaa muutosten seuraamista jälkikäteen. Turhilla muutoksilla tarkoitan tässä kohtaa syntaksin ulkoasullisia seikkoja. Rivinvaihto, sisennys tai puolipisteen käyttö lainausmerkin sijaan eivät vaikuta JavaScript-koodin lopputulokseen sitä suoritettaessa.

On täysin normaalia, että kehittäjillä on omat ”tyylinsä” kirjoittaa koodia, ja usein myös koodieditorin lisäosat vaikuttavat tähän. Tällaiset seikat tekevät usein versionhallintaan turhia merkintöjä.

Esimerkkitilanne henkilö A käyttää sisennykseen kahta välilyöntiä, kun vastaavasti Henkilö B käyttää neljää välilyöntiä. B:n täytyy käydä muokkaamassa A:n kirjoittamasta tiedostosta yhtä koodiriviä. B:n koodieditori formatoi koko tiedoston tallennuksen yhteydessä, jolloin myös kaikki A:n aikaisemmin kirjoittamat sisennykset muuttuvat kahdesta välilyönnistä neljään. Tämän jälkeen, kun B vie muutokset versionhallintaan, merkitsee

se kaikki muuttuneet kohdat uusina muutoksina, vaikka B:n oli tarkoitus muokata vain yhtä koodiriviä.

Formatoinnin integrointi ja automatisointi on suoraviivainen ja helppo tehtävä. Kaikki tarvittavat työkalut ladataan npm-rekisteristä ja tarvittavat konfiguraatiot tehdään package.json-tiedostoon.

Koodi formatoidaan käyttämällä Prettier-nimistä npm-pakettia. Sen tehtävä on huolehtia tiedostojen formatoinnista. Prettier lukee tiedoston ja palauttaa siitä formatoidun version. Prettierin tehtävä automatisoidaan Lint-staged npm -paketilla, ja se suorittaa määritellyn tehtävän kaikille tiedostoille, jotka ovat Git-versionhallinnassa staged-tilassa. Lint-staged-toiminto taas liitetään Gitin hook-rajapintaan Husky npm -paketilla. Esimerkkikoodi 1 esittää kaikki tarvittavat konfiguraatiot.

```
// package.json
{
...
  "prettier":{
    // formating rules here
  },
  "husky": {
    "hooks": {
      "pre-commit": "lint-staged"
    }
  },
  "lint-staged": {
    "src/**/*.{js,jsx,css,scss }": [
      "prettier --write"
    ]
  }
...
}
```

Esimerkkikoodi 1. Prettier-objektiin voidaan määrittää halutut formatointiasetukset, esimerkissä käytetään oletusasetuksia. Husky-objektiin määritettiin Gitin pre-commit-hook kutsumaan lint-staged-scriptiä. Lint-staged taas määritettiin suorittamaan prettier-scripti kaikille tiedostoille, joiden tiedostopääte on js, jsx, css tai scss ja ne sijaitsevat src-hakemistossa tai sen alihakemistoissa.

4.2 Käyttöliittymä & käyttäjäkokemus

4.2.1 React

Projektin käyttöliittymä rakennettiin Facebookin kehittämällä ja ylläpitämällä React-kirjastolla. Se on paradigmaltaan deklaratiivinen, ja sen pääpiirre on uudelleen käytettävät

komponentit. React-komponentti on yksinkertaisimmillaan JavaScript-funktio, joka palauttaa osan käyttöliittymää tai muita React-komponentteja. Esimerkkikoodissa 2 nähdään uudelleenkäytettävä painike, jonka tekstiä tai toimintoa voidaan muuttaa, mutta tyyli on ennalta määritelty ja aina sama.

```
// määritys
const StyledButton = ({ label, onClick }) => (
  <button onClick={onClick} style={{ color: "red" }}>{label}</button>
);

// käyttö
<StyledButton label="alert " onClick={() => alert("hello")} />
<StyledButton label="log" onClick={() => console.log("hello")} />
```

Esimerkkikoodi 2. React-komponentin määrittely ja käyttö

4.2.2 Material-ui

Web-sovelluksen ulkoasun määrittely ja suunnittelu on keskeinen ja tärkeä osa käyttöliittymän kehitystä. Kokonaisen suunnittelujärjestelmän (Design System) määrittely on usein aikaa vaativa työvaihe. Tämän lisäksi suunnittelujärjestelmän pohjalta tulisi luoda uudelleenkäytettävät komponentit eli kokonainen kirjasto käyttöliittymäkomponentteja. Projektin toteutuksessa ei kuitenkaan haluttu luoda omaa käyttöliittymäkirjastoa.

Päädyin etsimään valmiita avoimen lähdekoodin ratkaisuja. Alkuvaiheessa projektissa käytettiin Bootstrap-käyttöliittymäkirjastoa, mutta hyvin nopeasti selvisi, ettei Bootstrap ole ideaali ratkaisu React-sovelluksessa (5). Bootstrapin käyttämät JavaScript-toteutukset (jQuery) eivät täysin sovellu käytettäväksi VDOM-ympäristöissä

Lopulta päädyin Material-ui-kirjastoon, joka tarjoaa laajan valikoiman valmiita käyttöliittymäkomponentteja, ja niiden ulkoasu on kokonaan kustomoitavissa. Oletuksena Material-ui pohjautuu Googlen kehittämään material desing -malliin. Material-ui-valintaan vaikuttivat selkeä dokumentaatio ja aktiivinen kehittäjäyhteisö. Valintaa helpotti myös lista nimekkäistä teknologia-alan yrityksistä (mm Netflix, Amazon ja Nasa), jotka käyttävät Material-ui-kirjastoa palveluissaan. (6.)

Material-ui käyttää komponenttien ulkoasuun teemaa, jonka se saa teemakontekstista. Kontekstiin voidaan antaa kustomoitu teema tai käyttää oletusteemaa. Esimerkkikoo-

dissa 3 näkyy Material-ui-teemaan luominen, createMuiTheme-funktio ottaa parametrina teeman määrittelevän objektin, määrittelemättömät kohdat tulkitaan oletusarvoina. Funktio palauttaa teemaobjektin, jossa on määritelty kaikki tyylit kuten värivariantit, fonttikoot, varjostukset sekä paljon muita tyyliseikkoja.

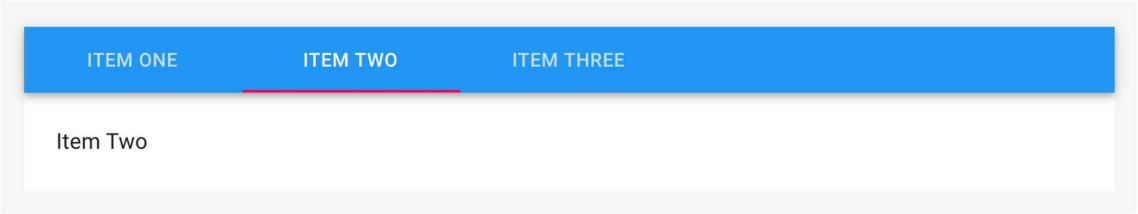
```
const theme = createMuiTheme({
  overrides: {…},
  palette: {…},
  typography: {…},
});
```

Esimerkkikoodi 3. Material-ui-teema luodaan createMuiTheme-funktiolla. Se ottaa parametrikseen objektin teeman asetuksista. Overrides määrittää yksittäisten komponenttien css-muutokset. Palette pitää sisällään kaikki teeman väriasetukset. Typography-osio määrittelee kaikki tekstiin liittyvät seikat.

Material-uissa määritellään kaksi teemaväriä, pääväri ja toissijainen korostusväri. Näille väreille lasketaan automaattisesti sopivat kontrastivärit tekstejä ja ikoneita varten, jotta niiden lukeminen olisi miellyttävämpää. Kuvissa 2, 3 ja 4 nähdään muutamia yleisimmin käytettyjä Material-ui-komponentteja oletusteemalla.

Kuva 2. Material-ui-painike eli Button-komponentti. Kuvasta näkyy myös teeman oletusvärit primary ja secondary.

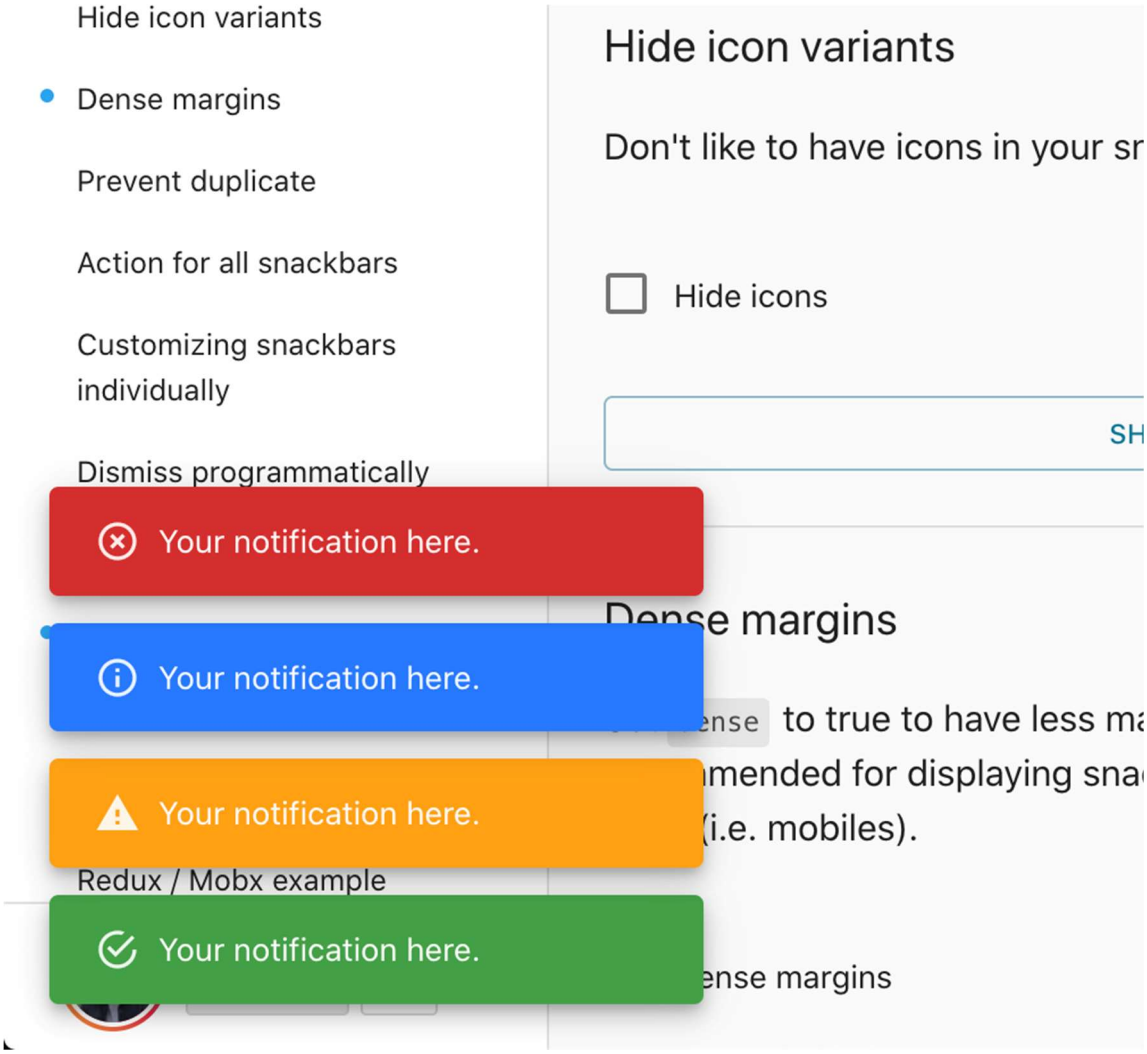
Kuva 3. Material-ui-syöttökenttä eli TextField-komponenti ja sen kaikki kolme eri tyylivarianttia.



Kuva 4. Material-ui-välilehtivalikko koostuu yhdestä ympäröivästä Tabs-komponentista, joka määrittelee muun muassa valikon kontekstin. Yksittäiset välilehdet määritellään Tab-komponen tilla.

4.2.3 notistack

Material-ui tarjotaan toteutuksen Androidista tutulle snackbar-ilmoitukselle. Sen tarkoitus on parantaa käyttäjäkokemusta antamalla käyttäjälle palautetta toiminnon suorituksen jälkeen. Material-ui-toteutuksessa jokaisen snackbar-ilmoituksen tila täytyy määrittää erikseen. Notistack on laajennos, jolla kaikkien snackbar-ilmoitusten tila viedään omaan kontekstiin, ja notistack huolehtii jokaisen ilmoituksen sijoittamisesta, avaamisesta, väri-variantista, aktiviteetistä ja sulkemisesta. Kehittäjän tehtäväksi jää ainoastaan kertoa mil-loin ja minkälaisen ilmoituksen haluaa käyttäjälle näyttää. Käyttäjän toimintojen lisäksi snackbar-ilmoituksia voidaan käyttää myös muuhun käyttäjälle annettavaan tietoon, esi-merkiksi jos sovellus havaitsee internetyhteyden katkenneen, voidaan käyttäjälle kertoa, että sovellus toimii tällä hetkellä offline-tilassa. Kuvasta 5 nähdään snackbar-ilmoituspino eli notisctack.



Kuva 5. Snackbar-ilmoitukset on määritelty näytettäväksi sovelluksen oikeaan alakulmaan neljä ilmoitusta kerrallaan. Kuvassa näkyy neljä eri värivarianttia: Error punaisella, Info sinisenä, Warning keltaisella ja Success virhreällä. Näiden lisäksi on vielä Default-variantti, jonka väri on oletuksena tummanharmaa.

4.2.4 Formik

Web-sovelluksissa lomakkeet hoitavat yleisesti datan lähettämisen palvelimelle, ja yksi sovellus saattaa pitää sisällään kymmeniä erilaisia lomakkeita. Vaikka lomakkeet voivat olla hyvinkin erilaisia, on niiden logiikka lähes aina sama. Tyypillisesti lomake ottaa vastaan dataa ja tekee tällä datalla jotain. Useimmiten se lähetetään palvelimelle. Tätä ennen käyttäjän antama data usein myös validoidaan. Projektissa hyvin nopeasti huomattiin, että lomakkeiden hallintaan tarvitaan jokin jaettu logiikka, jotta lomakkeiden toiminta olisi aina sama. Reactin dokumentaation lomakkeita käsittelevässä luvussa suositellaan

Formik-kirjastoa lomakkeiden kokonaisvaltaiseen käsittelyyn (7). Formikissa on valmis toteutus muun muassa. validointiin, virheviesteihin, lomakkeen lähetykseen ja palautukseen. Esimerkkikoodissa 4 on esitelty yksi kolmesta tavasta käyttää Formikia.

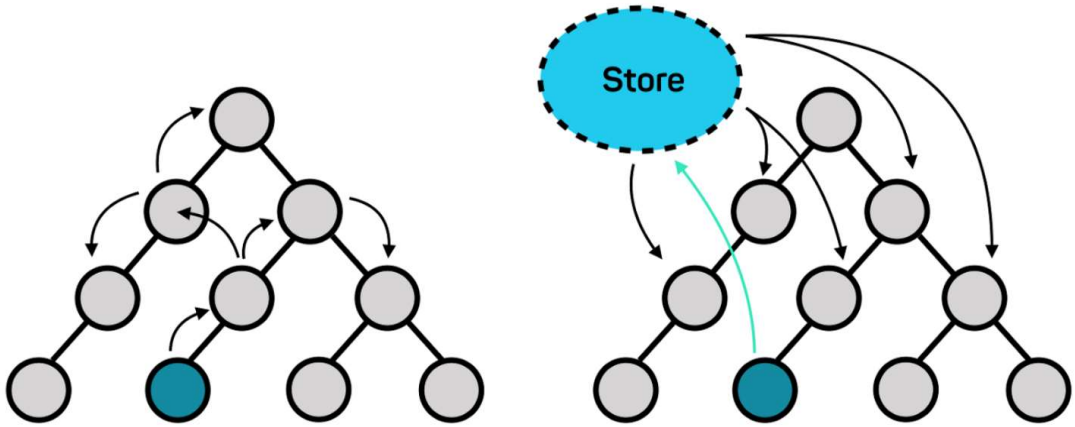
```
const Form = () => {
  const formik = useFormik({
    initialValues: {
      firstName: "",
      ...
    },
    validate: values => {
      const errors = {};
      if (!values.firstName) {
        errors.firstName = "required";
      }
      ...
      return errors;
    },
    onSubmit: values => {
      alert(JSON.stringify(values, null, 2));
    }
  });

  return (
    <Grid
      container
      direction="column"
      component="form"
      onSubmit={formik.handleSubmit}
      onReset={formik.handleReset}
      spacing={2}
    >
      <Grid item>
        <TextField
          fullWidth
          label="First name"
          name="firstName"
          onChange={formik.handleChange}
          onBlur={formik.handleBlur}
          value={formik.values.firstName}
          error={formik.touched.firstName && formik.errors.firstName}
          helperText={formik.touched.firstName && formik.errors.firstName}
        />
      </Grid>
      ...
      <Grid container justify="space-between">
        <Grid item>
          <Button type="reset" color="secondary">
            Reset
          </Button>
        </Grid>
        <Grid item>
          <Button type="submit" variant="contained" color="primary">
            Submit
          </Button>
        </Grid>
      </Grid>
    </Grid>
  );
};
```

Esimerkkikoodi 4. Funktio useFormik (react hook) ottaa parametrina objektin, joka määrittää kenttien oletusarvot sekä validointi- ja lähetyslogiikan. Funktio palauttaa lomakkeen logiikan sekä kenttien arvot ja tilan.

4.3 Tilanhallinta

Reactissa tai missä tahansa komponentteihin perustuvassa tekniikassa on tärkeä huolehtia datan luotettavuudesta. Kun useat komponentit käyttävät samaa dataa, tulisi niiden noudattaa ns. Single source of thruth -mallia, eli kaikki komponentit saavat ja muokkaavat dataa yhdestä ja samasta paikasta. Tätä paikkaa kutsutaan usein storeksi. Kuva 6 esittää kaksi mallia, joista toinen on keskitetyllä tilanhallinnalla ja toinen ilman.



Kuva 6. Vasemmalla nähdään kuinka dataa muokannut komponentti ilmoittaa muutoksestaan vain ylemmälle tasolle ja tämä toistuu rekursiivisesti niin kauan, kunnes data on saavuttanut kaikki sitä käyttävät komponentit. Oikealla nähdään keskitetty malli, jossa komponentti päivittää datan keskitetysti ja store huolehtii datan päivittämisestä komponenteille.

Reactissa keskitetty tilanhallinta vähentää myös propsien, eli React-komponentin argumenttien niin kutsuttua läpiporaamista (prop drilling) sekä sitä kautta turhaa uudelleenrenderöintiä. Prop drilling tarkoitaa sitä, että komponentti saa datan vain antaakseen sen eteenpäin eikä komponentti itsessään käytä annettua dataa mihinkään. React-komponentit renderöivät aina kun niille annettu data muuttuu eli tämä johtaa siihen, että myös ”läpi poratut” komponentit renderöivät uudelleen datan muuttuessa. Turhat renderöinnit heikentävät sovelluksen suorituskykyä. (8.)

Redux on yksi parhaimmin tunnetuista tilanhallintaan käytettävistä työkaluista Reactin kanssa. Sen suurin ongelma on kuitenkin niin kutsutun boilerplate-koodin kirjoitus. Jopa Reduxin virallisessa dokumentaatiossa on osio boilerplaten vähentämiseksi. (9) Reduxin sijaan päädyin käyttämään sen päälle rakennettua easy-peasy-tilanhallintaa.

Easy-peasy on intuitivinen ja suoraviivainen työkalu tilanhallintaan. Se ei vaadi konfiguraatiota tai kehittäjää kirjoittamaan boilerplate-koodia. Sen parhaat sisäänrakennetut ominaisuudet ovat

- react hookeihin perustuva rajapinta
- asynkroniset toiminnot / sivuvaikutukset
- johdetut / lasketut tietueet
- tilanjako ja tallennus (selaimen local- ja sessionStorageen)
- globaali, jaettu ja komponenttikohtainen tila.

Easy-peasy store määritellään JavaScript-objektilla, jossa on määritelty oletustila ja toiminnot. Esimerkkikoodissa 5 luodaan tila, jolla on johdettu tietue sekä toiminnot tilan päivittämiseen.

```
import { createStore, computed, action } from 'easy-peasy';

const store = createStore({
  todos: {
    items: ['Create store', 'Wrap application', 'Use store'],
    itemsCount: computed(state => state. items.length),
    addItem: action((state, payload) => {
      state.items.push(payload)
    }),
    replaceItem: action((state, payload) => {
      state.items.filter(item=>items !== payload.old)
      state.items.push(payload.new)
    }),
  }
});
```

Esimerkkikoodi 5. Tila luodaan createStore-funktiolla ja sen parametriksi annetaan objekti tilan toiminnoista ja oletusarvoista. Tila saa nimekseen todos, ja sen data on items kentässä. itemsCount-tietue on johdettu (computed) arvo eli se muodostetaan tilan datasta. addItem ja replaceItem ovat tilan toimintoja (action) joilla tilan dataa voidaan muokata.

Johdetut arvot saattavat vaikuttaa hieman turhalta tällaisessa pienessä esimerkissä, mutta kun asiaa tarkastellaan Reactin kannalta, huomataan, että niistä voi olla paljonkin hyötyä sovelluksen suorituskyvylle. Kuten aiemmin mainitsin, komponentti renderöi aina, kun data muuttuu. Johdetuilla arvoilla tämä vältetään, jos esimerkki tilaan korvataan yksi

items-jonon elementti. Tällöin vain komponentit, jotka käyttävät items-dataa, renderöivät, kun taas komponentit, jotka käyttävät johdettua itemsCount-arvoa, eivät huomaa muutosta lainkaan.

4.4 Kommunikointi (http)

JavaScriptissä on tällä hetkellä kaksi eri tapaa toteuttaa http-pyyntö. Fetch API on näistä uudempi toteutus, mutta sille ei ole tukea Internet Explorer 11 -selaimessa. Tuki Internet Explorer 11:lle haluttiin pitää projektissa mukana, niin pitkään kuin se onnistuisi ilman suurempia uhrauksia. XMLHttpRequest (myöhemmin XHR) on vaihtoehdoista vanhempi ja sille on tuki käytännössä kaikissa selaimissa. XHR:n ongelma on kuitenkin sen yksinkertaisuus, sillä ei ole modernia rajapintaa ja useat triviaalit asiat on jätetty kehittäjän ratkaistavaksi. Yleisesti ottaen, jos web-sovellus käyttää XHR:ää useammin kuin yhden kerran, tulisi sen ylle rakentaa jonkinlainen funktio, jonka avulla sitä voidaan myös uudelleen käyttää.

Axios on avoimen lähdekoodin HTTP-Client, joka on toteutettu JavaScript Promisella XHR:n päälle. Se pitää sisällään monia toimintoja, kuten esimerkiksi URL-parametrien käsittelyn, http-pyynnön ja vastauksen manipuloinnin globaalisti sekä vastauksen automaattisen formatoinnin statuksen mukaan. Sillä on myös kutsun peruutuksen mahdollistava toiminto, joka on yksi tärkeä ominaisuus React-sovellukselle muistivuotojen ehkäisemiseksi (10). Esimerkkikoodeissa 6 ja 7 on vertailun vuoksi kuvattu saman http-kyselyn toteutus perinteisen XHR:n ja Axios-kirjaston avulla.

```
var xhr = new XMLHttpRequest();

xhr.onreadystatechange = function () {

// request is complete
  if (xhr.readyState !== 4) return;

  // Process our return data
  if (xhr.status >= 200 && xhr.status < 300) {
    // request is successful
    console.log(JSON.parse(xhr.responseText));
  } else {
    // request has failed
    console.log(xhr);
  }
};

xhr.open('GET', '/user?ID=12345');
xhr.send();
```

Esimerkkikoodi 6. XMLHttpRequestin käyttö on tapahtumapohjaista ja eikä siinä ole mitään valmista toteutusta esimerkiksi kyselyn parametreille tai vastauksen tilalle. Kehittäjän pitää huolehtia if- ja else-lohkoissa, mitä kussakin tilanteessa halutaan tehdä.

```
axios.get('/user', {
    params: {
      ID: 12345
    }
  })
  .then(response => {
    console.log(response);
  })
  .catch(error => {
    console.log(error);
  })
```

Esimerkkikoodi 7. Axioksen rajapinnan tarjoamat funktiot ovat lupauksia (promise). Parametri, headerit ja muut muuttujat voidaan antaa suoraan kutsuttavan funktion parametrina. Axios hylkää lupauksen, jos http-vastauksen status kuvaa virhettä (4xx ja 5xx).

4.5 Lokalisointi

Web-sovelluksen lokalisointi voi olla todella raskas prosessi, jos sitä ei oteta huomioon heti kehityksen alkuvaiheessa. Selvää on, että jokaisen staattisen arvon muuttaminen dynaamiseksi tarkoittaa koko projektin läpikäymistä tavalla tai toisella.

Projektin varsinainen sisältö on suunniteltu kotimaan markkinoille, joten lokalisointi rajattiin kattamaan ainoastaan käännökset.

Projektin käännökset toteutettiin i18next- ja i18next-react-kirjastojen avulla. Kirjastoista ensin mainittu on yleinen JavaScript-toteutus lokalisointityökaluista ja jälkimmäinen on laajennos React-projekteille. i18next valikoitui projektiin hyvän ja selkeän dokumentaation vuoksi. React-hookeihin perustuva rajapinta oli myös yksi sen merkittävimmistä eduista. Esimerkkikoodissa 8 esitetään i18next-konfiguraatio.

```
import i18n from "i18next";
import { initReactI18next } from "react-i18next";
import { en, fi } from "./translations.json";

i18n
  .use(initReactI18next) // passes i18n down to react-i18next
  .init({
    resources: {
      en,
      fi
    },

    lng: "en",

    interpolation: {
      escapeValue: false // react already safes from xss
    }
  });

  export default i18n;
```

Esimerkkikoodi 8. i18next määritellään käyttämään i18next-react-laajennosta. Resources-kenttään annetaan json-tiedostosta tuodot käännökset. Lng määrittelee oletuskielen.

React-hookeihin perustuva rajapinta mahdollistaa käännösten toteuttamisen ilman erillistä ympäröivää komponenttia ja samalla selkeyttää koodin luettavuutta. Esimerkkikoodissa 9 esitetään käännöksen käyttöä komponentissa.

```
import React from 'react';

// the hook
import { useTranslation } from 'react-i18next';

function MyComponent () {
  const { t } = useTranslation();
  return <h1>{t('Welcome to React',{count:1})}</h1>
}
```

Esimerkkikoodi 9. useTranslation hook palauttaa varsinaisen kääntäjä funktion t ja objektin i18n. Funktiolle annetaan parametriksi käännettävä merkkijono, toisena parametrina voidaan määritellä asetus-objekti, esimerkiksi monikko käännösten dynaamiseen luomiseen. Funktio palauttaa merkkijonoa vastaavan käännöksen mikäli sellainen lötyy. Muutoin se palauttaa saamansa merkkijonon. i18n-objekti esittää kääntäjän asetuksia mm. kulloinkin valitun kielen.

5 Suunnittelumallit

Suunnittelumallit ovat ohjelmistokehityksen rakennemalleja, jotka ratkaisevat jonkin triviaaliongelma abstraktilla tasolla. Reactin kanssa on myös syytä tutustua muutamiin yleisesti käytettyihin malleihin, jotta vältyttäisiin ”keksimästä pyörää uudelleen”. Suunnittelumallien avulla voidaan usean kehittäjän tiimissä kirjottaa helpommin yhtenäistä koodia. Lisäksi ne auttavat ymmärtämään muiden kehittäjien kirjoittamaa koodia helpommin. Yksi sovelluskehityksen haastavimmista tehtävistä on ymmärtää ja muokata jo kirjoitettua koodia. (11.)

5.1 Higher-Order Component

Higher-Order Component (HOC) jäljittelee ohjelmistokehityksessä yleisesti tunnettua Decorator pattern -mallia (12). Sen keskeinen piirre on laajentaa ja/tai lisätä komponentin tilaan perustuvia toiminnallisuuksia ja/tai ominaisuuksia. HOC on teknisesti ylemmän tason funktio (higher-order function), joka saa parametrikseen vähintään laajennettavan komponentin. Esimerkkikoodissa 10 nähdään yksinkertainen klikkauksia laskeva HOC ja malli luodun HOC:n käytöstä. Esimerkkikoodi 11 esittää laajennettavaa komponenttia.

```
// HOC wrapper definition
function withCounter(WrappedComponent) {
  return class extends Component {
    constructor(props) {
      super(props);
      this.state = { count: 0 };
      this.increment = this.increment.bind(this);
    }
    // increase
    increment = () => {
      const { count } = this.state;
      return this.setState({ count: count + 1 });
    };
    render() {
      const { count } = this.state;
      return (
        <WrappedComponent
          count={count}
          increment={this.increment}
          {...this.props}
        />
      );
    }
  };
}

// HOC usage
import ViewComponent from "./ViewComponent ";
```

```
import withCounter from "./HOC";

const ViewComponentWithCounter = withCounter(CounterView);

function App() {
  return < ViewComponentWithCounter />;
}
```

Esimerkkikoodi 10. HOC mallissa with-funktio palauttaa anonyymin komponentin, joka käärii sisäänsä laajennettavan komponentin, jotta HOC pystyy pitämään lukua (tilaa) klikkauksista. Anonyymin komponentin tulee olla Reactin luokkakomponentti.

```
function ViewComponent({ count, increment }) {
  return (
    <div>
      <p>{`Click coount: ${count}`}</p>
      <button onClick={increment}>Click me</button>
    </div>
  );
}
```

Esimerkkikoodi 11. ViewComponent ei ota kantaa siihen, miten toiminnot on toteutettu. Komponentin tarkoitus on tarjota vain rajapinnan toimintojen ja käyttäjän välille.

Lisäksi on myös hyvin tavanomaista, että funktiolle annetaan parametrina muitakin argumentteja. Näiden parametrien tavoitteena on yleensä laajentaa HOC:in käyttötapauksia tai konfiguroida sen toimintoja. HOC-funktioiden yleinen nimeämisetuliite on with. Esimerkkikoodissa 12 on käytetty Formik-kirjaston withFormik-funktiota, joka laajentaa sille annetun komponentin logiikkaa ja ominaisuuksia Formik-lomakkeen hallinnalla.

```
const MyEnhancedForm = withFormik({
  mapPropsToValues: (props) => ({ name: '' }),

  // Custom sync validation
  validate: values => {
    const errors = {};

    if (!values.name) {
      errors.name = 'Required';
    }

    return errors;
  },

  handleSubmit: (values, actions)=> {
    alert(JSON.stringify(values, null, 2));
    actions.setSubmitting(false);
  },

  displayName: 'MyFormWithFormik',
})(MyForm);
```

Esimerkkikoodi 12. WithFormik HOC saa ensin parametrikseen Formikin määritykseen tarvittavat argumentit. MapPropsToValues määritellään lomakkeen oletus arvot,

validation määrittää validointi menetelmät ja handleSubmit määrittelee lomakkeen lähetyksessä käytettävän logiikan. Lopuksi HOC:lle annetaan laajennettava komponentti. Kannattaa huomata, että laajennettavaa MyForm-komponenttia kutsutaan lopussa IIFE-menetelmällä.

Usein HOC luodaan yhden ylimääräisen sulkeuman (closure) kanssa. Ylimääräisen sulkeuman käyttöön ei ole mitään varsinaista sääntöä, ja sitä käytetäänkin melko vapaalla tyylillä. Esimerkiksi withFormikin lähdekoodissa sitä on kommentoitu seuraavasti.

> ”Meidän on käytettävä sulkeumaa, jotta käärittävän komponentin propsit voidaan toimittaa withFormikin konfigurointimetodeille.” (13)

Ylimääräinen sulkeuma ilmenee, kun with-funktiolle annetaan kaikki muut parametrit paitsi laajennettava komponentti. Funktio palauttaa uuden funktion (sulkeuman), jolle annetaan parametrina ainoastaan laajennettava komponentti. Esimerkkikoodi 13 ViewComponent havainnollistaa ylimääräisen sulkeuman olemassaoloa withFormik HOC:ssa.

```
// yleinen HOC kirjoitusasu (IIFE)
const MyEnhancedForm = withFormik({ args })(MyForm);

// osiin purettu kirjoitusasu
const closure = withFormik({ args });
const MyEnhancedForm = closure (MyForm);
```

Esimerkkikoodi 13. MyEnhancedForm-komponentti muodostetaan vasta withFormikin palauttamalla funktiolla.

5.2 Render prop

Kuten HOC, myös render prop -mallissa tavoite on laajentaa komponentin tilaan perustuvaa logiikkaa ja ominaisuuksia. Render prop saa nimensä siitä, että propsina annettua funktiota kutsutaan komponentin renderöintivaiheessa (render-funktiossa tai return-lauseessa). Se pystyy toteuttamaan kaikki samat toiminnallisuudet kuin HOC. Esimerkkikoodissa 14 nähdään yksinkertainen klikkauksia laskeva render prop -komponentti, ja se käyttää myös esimerkkikoodissa 11 esiteltyä ViewComponentia.

```
// RenderProp wrapper definition
class RenderProp extends Component {
  state = {
    count: 0
  };
  // increase
```

```
  increment = () => {
    const { count } = this.state;
    return this.setState({ count: count + 1 });
  };
  render() {
    const { count } = this.state;
    return this.props.render({
      increment: this.increment,
      count: count
    });
  }
}

// RenderProp usage
import ViewComponent from "./ ViewComponent ";
import RenderProp from "./RenderProp";

function App() {
  return (
    <RenderProp
      render={props => (
        < ViewComponent count={props.count} increment={props.increment} />
      )}
      // render={ ViewComponent }  <- short-handed version
    />
  );
}
```

Esimerkkikoodi 14. Komponentti kutsuu propsina annettua funktiota renderöintivaiheessa. Samoin kuin HOC:in, tulee myös render prop -komponentista määritellä Reactin luokkakomponentti, jotta se pystyy pitämään lukua (tilaa) klikkauksista.

Vertailemalla esimerkkikoodeja 10 ja 14 voidaan todeta, että valinta render propin ja HOC:n välillä perustuu yleensä kehittäjän omaan mieltymykseen.

HOC- ja render prop -komponentteja voidaan myös käyttää sisäkkäin aivan kuten mitä tahansa React-komponentteja. Useat sisäkkäiset render prop -komponentit saattavat alkaa muistuttaa JavaScriptistä yleisesti tuttua ”callback hell” -tilannetta, joka vaikeuttaa koodin luettavuutta. Tällainen tilanne syntyy, kun render prop -komponentin funktio palauttaa uuden render prop -komponentin, jonka funktio palauttaa jälleen render prop -komponentin ja niin edelleen.

5.3 Container componet

Komponentin logiikan ja esityksen erottelu omiin komponentteihin on tehokas tapa rakentaa uudelleenkäytettäviä komponentteja. Container component -mallissa komponen-

tin logiikka erotetaan esittävästä osasta. Esimerkkikoodi 14 esittää monoliittisen komponentin ja esimerkkikoodi 15 esittää erotetun mallin Container- ja View- komponenteista.

```
import React from "react";

const UserList = () => {
  const users = getAllUsers();

  return (
    <li>
      {users.map((user) => (
        <ul>{user.name}</ul>
      ))}
    </li>
  );
};
```

Esimerkkikoodi 15. Monoliittisessa UserList-komponentissa kaikki toiminnot on rakennettu yhden komponentin sisään. Komponentti siis vastaa datan hakemista sekä näyttämisestä. Tällaisen komponentin uudelleenkäyttö on mahdollista vain silloin, kun data ja logiikka pysyy samana.

```
import React from "react";

const UserList = ({ users }) => (
  <li>
    {users.map((user) => (
      <ul>{user.name}</ul>
    ))}
  </li>
);

const AllUsersContainer = () => {
  const allUsers = getAllUsers();

  return <UserList users={allUsers} />;
};

const SomeUsersContainer = () => {
  const someUsers = getSomeUsers();

  return <UserList users={someUsers} />;
};
```

Esimerkkikoodi 16. Logiikatonta UserList-komponenttia voidaan käyttää, koska data ja logiikka määritellään sitä ympäröivässä Container-komponentissa.

Containet component -malli on mielestäni yksi tärkeimmistä Reactin kanssa käytettävistä suunnittelumalleista, koska se on tehokas ja helppo tapa rakentaa uudelleen käytettäviä komponentteja.

5.4 Hooks (react >= 16.8.0)

Reactin versiossa 16.8.0 esiteltiin uusi Hook-rajapinta, joka mahdollistaa tilan ja logiikan erotuksen itse komponentista (13). Sitä kautta tilasta ja logiikasta tulee myös helpommin uudelleenkäytettäviä. Vaikka HOC ja render prop on kehitetty ratkaisemaan tätä ongelmaa, ei niiden käyttö ole aina suoraviivaista, ja ne tekevät usein koodin lukemisesta hankalaa. (14.)

Päivityksessä esiteltiin Hookit muun muassa seuraaville toiminnoille

- useState yksinkertaisen tilan hallintaan (esim. yksi arvo)
- useReducer kompleksisen tilan hallintaan (esim. useita sisäkkäisiä objekteja)
- useEffect tilan muutoksen sivuvaikutuksille
- useContext kontekstin kuormittamiseen
- useCallback funktioiden muistamiseen renderöintien välillä
- useMemo johdettujen tietueiden muistamiseen renderöintien välillä
- useRef elementtien viittauksiin.

Hookeilla on mahdollista tuoda komponentille tilaan perustuvia toimintoja ilman niin kutsuttua käärekomponenttia (wrapper component), johon HOC ja render prop perustuvat. Esimerkkikoodissa 16 toteutetaan React-hookien avulla sama yksinkertainen klikkauksia laskeva toiminnallisuus kuin esimerkkikoodeissa 10 ja 13. Esimerkissä on käytetty esimerkkikoodissa 11 esiteltyä ViewComponentia.

```
// hook definition
import { useState, useCallback } from "react";

function useCounter() {
  const [count, setCount] = useState(0);
  const increment = useCallback(() => {
    setCount(state => state + 1);
  }, [setCount]);

  return { count, increment };
}
// hook usage
import ViewComponent from "./ViewComponent";
import useCounter from "./hook";

function App() {
  const { count, increment } = useCounter();
  return <ViewComponent count={count} increment={increment} />;
}
```

Esimerkkikoodi 17. Tila ja logiikka on määritelty useCounter-hookissa. Tila tallennetaan Reactin tarjoamalla useState-hookilla, ja increment-funktio tallennetaan renderöitien välillä useCallback-hookilla. UseCallback on tärkeää osa siksi että Reactin sisäänrakennettu optimointi pystyy tunnistamaan, onko funktio oikeasti muuttunut vai onko kyseessä vain uusi viittaus renderöinnin vuoksi. UseCounter-hookia voidaan kutsua nyt missä tahansa funktiokomponentissa ja se palauttaa objektin, joka sisältää count-tilan sekä increment-funktion tilan muuttamiseksi.

Hookien käytössä on muutamia sääntöjä

- Hookeja voidaan kuormittaa ainoastaan Reactin funktiokomponenteissa.
- Hookeja tulee kutsua aina funktiokomponentin ylimmällä tasolla.
- Hookien yleinen nimeämisetuliite on use.

Vaikka hookit eivät ole suunnittelumalli, on ne kehitetty ratkaisemaan samaa ongelmaa kuin render prop ja HOC. Reactin dokumentaatiossa ja monissa muissa lähteissä on puhuttu hookien korvaavan HOC- ja render prop -mallien käyttötapaukset lähes kokonaan. (15) Hook-rajapinnan käyttö edellyttää, että kehitysprojektissa on käytössä React 16.8.0 tai sitä uudempi versio.

6 Testaus

Sovelluksen testaaminen on oleellinen osa kehitysprosessin laadun varmistusta. Testaamalla voidaan havaita kehitysprosessissa syntyneitä ongelmia jo aikaisessa vaiheessa ja ennaltaehkäistä niiden päätymistä tuotantoversioon asti.

Projektille ei määritelty kovinkaan tiukkoja testaus kriteerejä, mutta projektin kehittäjien kesken sovittiin muutamista oleellisista testaukseen liittyvistä asioista. Yksikkötestit sopivat hyvin huonosti React-komponenttien testaamiseen, sillä ne jakavat usein dataa ja logiikkaa ylemmän tason komponentin (funktion) kanssa.

Yksi projektin tärkeimmistä testaustavoista on niin kutsuttu smoke-test. Testi yksinkertaisesti vain kokeilee, renderöikö komponentti ilman virheilmoituksia. Esimerkkikoodissa 17 nähdään, ettei tällaisen testin kirjoittaminen vaadi juurikaan aikaa tai vaivaa. Smoke-testin antama tieto on kuitenkin todella arvokasta silloin, kun jotain on pielessä.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
});
```

Esimerkkikoodi 18. Smoke-testi kokeilee yksittäisen komponentin renderöintiä ja ilmoittaa, mikäli renderöinnin aikana syntyy virheilmoituksia.

Testaus haluttiin myös integroida versionhallinnan yhteyteen. Integraatio poistaa inhimillisen unohduksen mahdollisuuden kokonaan, koska testi ajetaan aina ennen muutosten vientiä versionhallintaan. Tämä toteutettiin myös aiemmin mainitun Husky-kirjaston avulla gitin pre-push hookiin.

7 Tuotantoversio

Web-sovelluksen lähdekoodi harvoin päätyy sellaisenaan tuotantokäyttöön. React sisältää monia kehitysvaiheeseen tarkoitettuja varoituksia. ja nämä tekevät varsinaisesta sovelluskoonnista suuremman ja hitaamman. (16.)

7.1 Koonti (build)

React-sovellus on tarkoitus koota tuotantoversioksi (production build) ennen varsinaista julkaisua. Koontivaiheessa sovelluksen lähdekoodi käy läpi monia manipulointivaiheita. Lähdekoodista jätetään esimerkiksi pois kaikki kehitysvaiheessa käytettävät toiminnot. Lisäksi lähdekoodia pienennetään niin kutsutulla pienennys (minifying) toiminolla. Esimerkkikoodissa 18 nähdään erot samasta funktiosta kehitys- ja tuotantoversiossa.

```
// development
const multiply = (veryLongArgumentName, anotherEvenLongerArgumentName) => {
  // code comment
  return veryLongArgumentName * anotherEvenLongerArgumentName;
};

// production
const multiply=(l,t)=>l*t;
```

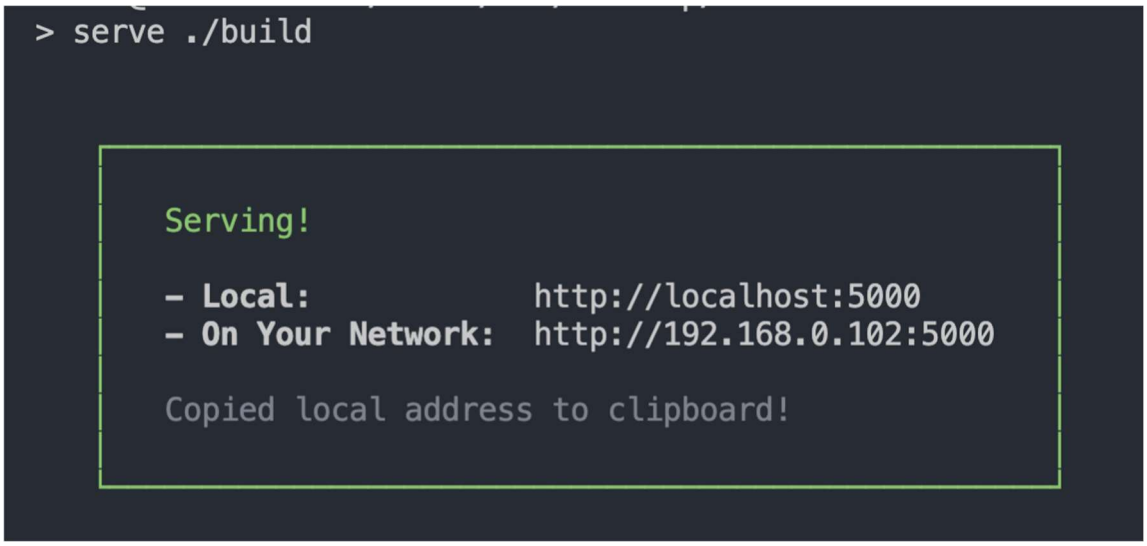
Esimerkkikoodi 19. Tuotantoversion lähdekoodia ei ole tarkoitettu ihmisen luettavaksi. Sen vuoksi sieltä voidaan poistaa kaikki kommentit, syntaksin lukua helpottavat sisennykset ja rivinvaihdot, sekä muuttujien nimet voidaan muuttaa niin lyhyiksi kuin mahdollista. Tämän johdosta käyttäjän selaimeen lähetettävän datan (suoritettava koodi) koko saadaan pidettyä mahdollisimman matalana. Tästä hyötyvät erityisesti hitaalla yhteydellä varustetut päätelaitteet, koska ladattavan datan määrä on pienempi.

Selaimet pyrkivät nopeuttamaan sivun latausta tallentamalla kertaalleen ladatut tiedostot välimuistiin. Välimuisti voi aiheuttaa ongelmia sovelluksen eri versioiden välillä, koska selain ei tiedä milloin web-sovelluksen lähdekoodia on muutettu ja pyrkii se aina käyttämään jo välimuistissa saatavilla olevia tiedostoja. Tällaiseen väliuistiongelmaa löytyy onneksi ratkaisu ja se on valmiiksi konfiguroitu Create-react-appillä luodussa projektissa. Sovelluksen koontivaiheessa kaikkien tiedostojen nimeen liitetään tiiviste (hash) itse tiedoston sisällöstä, jos tiedosto on muuttunut, muuttuu myös sen nimessä oleva tiiviste. (17) Nyt selain näkee muuttuneen tiedoston uutena tiedostona ja lataa sen aina palvelimelta. Kuvasta 7 nähdään tuotantoversion tiedostojen nimet ja koot pakattuina.

Kuva 7. Tiedostot nimetään seuraavalla tavalla [tunniste].[hash].[?chunk].ext. Tunniste on joko main, numero, tai vendor code joka viittaa avoimen lähdekoodin kirjaston tekiään. Hash eli tiiviste lasketaan tiedostosta itsestään. Chunk merkitsee että tiedosto on osa suurempaa kokonaisuutta.

7.2 Julkaisu

Projektin tuotantoversio tarvitsee http-palvelimen, jotta se voidaan julkaista kaikkien käytettäväksi internettiin. Serve on pieni ja yksinkertainen http-palvelin, joka on saatavilla npm-paketinhallinnasta. Se ei vaadi konfigurointia tai osaamista palvelimen perustamiseen. Kuvasta 8 voidaan todeta, että tuotantoversion julkaisu onnistuu yhdellä ainoalla komennolla.



Kuva 8. Palvelin käynnistetään komennolla serve <hakemisto>. Terminaali näkymä onnistuneen http-palvelimen käynnistyksen jälkeen.

Kuva 9.

8 Yhteenveto

Kehitysprojektin perustaminen saattaa alkuun kuulostaa hyvin suoraviivaiselta ja jopa helpolta tehtävältä. Projektin perustamisen jälkeen, kun varsinaisen tuotteen kehitystyö alkoi, huomattiin useita asioita, joihin olisi voinut kiinnittää enemmän huomiota. Tästä johtuen projektin pohjaa ei ole vielä sellaisenaan uudelleenkäytetty, mutta pienillä muutoksilla sitä on hiottu jo paljon parempaan suuntaan uusien kehitysprojektien myötä.

Avoimen lähdekoodin kirjastojen valinnassa Axios, Material-ui ja easy-peasy ovat osoittautuneet hyviksi valinnoiksi. Ne ovatkin vakiinnuttaneet paikkansa myös kaikissa uusissa projekteissa. Sen sijaan Formikin on huomattu aiheuttavan hieman soveltuvuusongelmia suurissa ja monimutkaisissa lomaketoteutuksissa. Formikille ei kuitenkaan ole löytynyt korvaavaa toteutusta, joka ratkaisisi kaikki sen puutteet mutta samalla tarjoaisi yhtä laajat ominaisuudet. Yhtenä vaihtoehtona on suunniteltu oman toteutuksen luontia Formikin inspiroimana.

Varsinainen kehitysympäristö on toiminut lähes moitteetta kaikilla kehitystiimin jäsenillä. Tiimin käytössä ovat kaikki suurimman käyttöjärjestelmät, sekä laaja kokoelma erilaisia koodieditoreita ja sitä kautta vielä sitäkin laajempi valikoima erilaisia editoriliitännäisiä.

Vaikka insinöörityön ohessa luotua projektipohjaa ei ole suoraan otettu käyttöön sellaisenaan uusissa kehitysprojekteissa, on siitä ollut silti merkittävä apu uusia projekteja perustettaessa, ja se onkin toiminut mallina kaikille uusille projekteille.

Vaikka projektilla ei ollut varsinaista määritystä tai tarkkaan kuvattuja tavoitteita, voidaan projektia sanoa onnistuneeksi, koska se saavutti sille toivotut tarpeet ja antoi koko kehitystiimille paremman käsityksen siitä, minkälaisessa ympäristössä haluamme jatkossakin työskennellä.

Lähteet

1 Download the Node.js source code, 2020. Verkkoaineisto. Nodejs.org. <https://nodejs.org/en/download>. Luettu 18.05.2020.

2 Introducing JSX, 2020. Verkkoaineisto. React-dokumentaatio <https://reactjs.org/docs/introducing-jsx.html>. Luettu 04 05 2020.

3 Recommended Toolchains, 2020. Verkkoaineisto. React-dokumentaatio <https://reactjs.org/docs/create-a-new-react-app.html#recommended-toolchains>. Luettu Haettu 5 4 2020.

4 D. Grander, 2018. Verkkoaineisto. Malicious code found in npm package event-stream downloaded 8 million times in the past 2.5 months <https://snyk.io/blog/malicious-code-found-in-npm-package-event-stream/>. Luettu 05 05 2020.

5 Integrating with jquery chosen plugin, 2019. Verkkoaineisto. React-dokumentaatio <https://reactjs.org/docs/integrating-with-other-libraries.html#integrating-with-jquery-chosen-plugin>. Luettu 05 05 2020.

6 Who's using Material-UI?, 2020. Verkkoaineisto. Material-ui.com. <https://material-ui.com/> [Online]. https://material-ui.com/. Luettu 05 05 2020.

7 Fully fledged solutions, 2020. Verkkoaineisto. React-dokumentaatio <https://reactjs.org/docs/forms.html#fully-fledged-solutions>. Luettu 05 05 2020.

8 J. Luong, 2019. Verkkoaineisto. When prop-drilling slows down my React apps <https://www.techynovice.com/when-prop-drilling-slows-down-my-react-apps>. Luettu 06 05 2020.

9 Reducing boilerplate, 2020. Verkkoaineisto. Redux-dokumentaatio <https://redux.js.org/recipes/reducing-boilerplate>. Luettu 06 05 2020.

10 A. Imran, 2018. Verkkoaineisto. How to work with React the right way to avoid some common pitfalls <https://www.freecodecamp.org/news/how-to-work-with-react-the-right-way-to-avoid-some-common-pitfalls-fc9eb5e34d9e>. Luettu 06 05 2020.

11 E. Freeman ja E. Robson, 2016. Verkkoaineisto. 5 reasons to finally learn design patterns <https://www.oreilly.com/content/5-reasons-to-finally-learn-design-patterns>. Luettu 07 05 2020.

12 R. Kotze, 2018. Verkkoaineisto. Higher-order components vs Render Props <https://www.richardkotze.com/coding/hoc-vs-render-props-reacthtml>. Luettu 10 05 2020.

13 J. Palmer, 2020. Verkkoaineisto. Formik versiohistoria <https://github.com/jared-palmer/formik/blob/efe128c31a953194e6d801a51eb4d0764499c3af/packages/formik/src/withFormik.tsx#L128>. Luettu 10 05 2020.

14 D. Abramov, 2019. Verkkoaineisto. React v16.8: The One With Hooks <https://reactjs.org/blog/2019/02/06/react-v16.8.0.html>. Luettu 07 05 2020.

15 It's hard to reuse stateful logic between components, 2020. Vekkoaineisto. React-dokumentaatio <https://reactjs.org/docs/hooks-intro.html#its-hard-to-reuse-stateful-logic-between-components>. Luettu 05 05 2020.

16 Do Hooks replace render props and higher-order components?, 2020. Verkkoaineisto. React-dokumentaatio <https://reactjs.org/docs/hooks-faq.html#do-hooks-replace-render-props-and-higher-order-components>. Luettu 07 05 2020.

17 Optimizing performance, 2020. Verkkoaineisto. React-dokumentaatio <https://reactjs.org/docs/optimizing-performance.html#use-the-production-build> Luettu 08 05 2020.

18 J. Evans, 2019. Verkkoaineisto. Creating a Production Build <https://create-react-app.dev/docs/production-build>. Luettu 08 05 2020.