SAVONIA

OPINNÄYTETYÖ - **AMMATTIKORKEAKOULUTUTKINTO**
TEKNIIKAN JA LIIKENTEEN ALA

QARTOITUS ENTERPRISE PORTAL

Konsultointityökalu

TEKIJÄ/T: Eetu Leivo

SAVONIA-AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Tiivistelmä

Koulutusala Tekniikan ja liikenteen ala	
Koulutusohjelma/Tutkinto-ohjelma Tietotekniikan tutkinto-ohjelma	
Työn tekijä(t) Eetu Leivo	
Työn nimi Qartoitus Enterprise Portal	
Päiväys 16.04.2019	Sivumäärä/Liitteet 29/0
Ohjaaja(t) Lehtori Jukka Kinnunen, lehtori Mikko Pääkkönen	
Toimeksiantaja/Yhteistyökumppani(t) Qibbie Mobile Oy	

Tiivistelmä

Tämän opinnäytetyön aiheena oli suunnitella ja toteuttaa Qartoitus Enterprise Portal annettujen vaatimuksien perusteella. Työn tilasi Qibbie Mobile Oy, jossa olen toiminut ohjelmistosuunnittelijana helmikuusta, 2018. Portaali on suurin osa-alue, suuremmasta kokonaisuudesta, johon kuuluu 4 erillistä portaalia.

Työ tilattiin aiemman testiversion tuottaessa mielenkiintoa markkinoilla. Palvelu rakennettiin uudelleen alusta alkaen hiomalla ja parantamalla olemassa olevia ominaisuuksia ja lisäämällä uusia ominaisuuksia.

Työ on toteutettu käyttäen ReactJS ja Reduxia Frontend-tekniikoina, Play-framework (Java) toimii Rest API:na ja tietokantana on MongoDB.

Qartoitus Enterprise Portal on konsultointityökalu, jonka avulla tiedon keruu ja tarpeiden kartoitus on helppoa. Palvelu sisältää visuaalisen raportointityökalun sekä kartoitukseen vaadittavat helppokäyttöiset työkalut. Sen avulla on kätevä kerätä dataa, jonka avulla voidaan esimerkiksi keskittyä yrityksen ongelmakohtiin.

Avainsanat
ReactJS, Redux, Play-framework, MongoDB

SAVONIA UNIVERSITY OF APPLIED SCIENCES

THESIS
Abstract

Field of Study Technology, Communication and Transport			
Degree Programme Degree Programme in Information Technology			
Author(s) Eetu Leivo			
Title of Thesis Qartoitus Enterprise Portal			
Date	15 December 2020	Pages/Appendices	29/0
Supervisor(s) Lecturer Jukka Kinnunen & Lecturer Mikko Pääkkönen			
Client Organisation /Partners Qibbie Mobile Oy			

Abstract

The purpose of this thesis was to design and create Qartoitus Enterprise Portal. Qartoitus Enterprise Portal was designed to be a platform for consultants. The aim of the platform is to make it easy to gather and use data. Requirements were to contain visual reporting tools and create easy to use tools for gathering the required data. The objective of the platform is aimed to provide the tools necessary for the consultants so they can provide better guiding for the target company to improve and make better results.

The work started with a design phase that included choosing the right techniques, planning the tasks and putting them in the correct order. Techniques chosen were ReactJS & Redux in the frontend. Backend was created with Play-framework (Java) that was used as a REST-API and MongoDB for the database.

The design phase lasted for the first week and most of the time was spent on planning all the components required for the application. Care was taken to do designs that were easy to implement and fit all the requirements. The execution of the designs was done in weekly sprints, where the aim was to complete one main requirement. The work included handling the frontend and backend parts of the designs.

As a result of this thesis, Qartoitus Enterprise Portal was created in the time required with all the requirements given by the client Qibbie Mobile Oy. In addition, new requirements surfaced after this thesis and were commissioned by Qibbie Mobile Oy.

Keywords
ReactJS, Redux, Play-framework, MongoDB

ESIPUHE

Haluaisin kiittää Qibbie Mobile Oy:n henkilökuntaa luottamuksesta ja mahdollisuudesta työskennellä tämän mielenkiintoisen projektin parissa. Erityiskiitos Ari Oksaselle ja Jani Bäckille, että antoivat minulle mahdollisuuden tehdä tästä opinnäytetyön. Totetus aloitettiin vasta marraskuun puolivälissä ja ensimmäinen beta versio on nyt valmis.

Kiitos Henri Harjanteelle, joka toimi projektin vetäjänä ja suunnitteli portaalien näkymiä. Kiitos myös Miika Niemelle, joka toimii myös ohjelmoijana Qibbie Mobile Oy:ssa. Monta ongelmaa ratkaistiin pienen tai suuremman debaatin jälkeen.

Kiitos myös lehtoreille, Jukka Kinnuselle ja Mikko Pääkköselle, jotka toimivat opinnäytetyöni ohjaajina.

Erityisen suuri kiitos avopuolisolleni Moonalle, joka kannusti ja piiskasi minut tekemään tämän työn loppuun.

Kuopiossa 26.12.2018, päivitetty 04.11.2019
Eetu Leivo

SISÄLTÖ

1 JOHDANTO 7
2 SOVELLUS 8
2.1 Tarkoitus 8
2.2 Toiminnot 8
2.3 Suunnittelu 9
2.4 Rakenne 10
3 TEKNIIKAT 12
3.1 REST API 12
3.1.1 Play-Framework 12
3.1.2 MongoDB 12
3.2 Redux 13
3.2.1 Provider 13
3.2.2 Redux-Store 14
3.2.3 Connect 15
3.2.4 Actions 16
3.2.5 Funktiot mapStateToProps ja mapDispatchToProps 17
3.3 ReactJS 18
3.3.1 Container 19
3.3.2 Class 19
3.3.3 Functional 20
3.3.4 State 20
3.3.5 Props 21
4 TOTEUTUS 22
4.1 Kartoitustyökalu 22
4.1.1 Kartoitus 22
4.1.2 Kysymysryhmä 23
4.1.3 Kysymys 24
4.2 Lähetystyökalu 25
4.3 Seurantatyökalu 25
4.4 Yrityksen hallinta ja kumppanit 26
4.5 Raportointityökalu 26

4.5.1 Vertailutyökalu 27

5 YHTEENVETO JA POHDINTA 28

LÄHTEET 29

1 JOHDANTO

Qibbie Mobile Oy on perustettu 2016 ja se tuottaa ohjelmistoratkaisuja. Tärkeimpänä tuotteena ovat olleet Versoportal sekä Specialistportal, jotka tuotettiin Terveyspalvelu Versolle. Kokonaisuudessa on kyselymoottori, jota käytetään terveyskyselyiden suorittamiseen sähköisesti, joka säästää aikaa sekä resursseja.

Tästä kyselymoottorista lähti idea, että sehän ei ole alaan tai aiheeseen sidottuna, vaan sitä voitaisiin käyttää kartoittamaan yrityksen tai ihmisryhmien tarpeita. Tähän tarpeeseen luotiin testiversio, joka herätti mielenkiintoa.

Opinnäytetyönäni suunnittelin ja toteutin Qartoitus Enterprise Portalin. Portaalissa on useita eri toimintoja, jotka käydään läpi yksitellen tässä raportissa. Tämän raportin yhteydessä kutsutaan tätä kokonaisuutta palveluna.

Opinnäytetyöni suunnitteluvaiheessa suunnittelin ja määrittelin sovelluksen toimintoja yhdessä Henri Harjanteen kanssa. Varsinainen toteutus oli minun vastuullani. Käytetyt tekniikat ja työskentelytavan sain valita itse. Lopputuotteena syntyi Qartoitus Enterprise Portalin beta-versio, jonka kehitystä jatketaan tammikuusta alkaen. Tässä raportissa kerron opinnäytetyöni työvaiheista, toteutuksesta, kehitysmenetelmistä ja ominaisuuksista.

Koska kyseessä on Qibbie Mobile Oy:n palvelu, tässä dokumentaatiossa ei tulla esittelemään oikeita esimerkkejä koodista, tai edes kuvia palvelusta. Tekniikat osuudessa on yksinkertaisia esimerkkejä, miten React ja Redux toimivat.

2 SOVELLUS

2.1 Tarkoitus

Opinnäytetyön tarkoituksena oli luoda portaali, jossa voi luoda ja lähettää kartoituksia sekä seurata niiden edistymistä. Kokonaisuudessa on myös raportointityökalu, joka analysoi kartoituksen automaattisesti sen tulosten perusteella. Kartoituksen vastaukset ovat anonyymejä. Palvelulle on havaittu selkeää tarvetta ja asiakkaita palvelulle on jo valmiiksi.

2.2 Toiminnot

Palvelu sisältää monia eri toimintoja ja se on yksi pääkokonaisuus suuremmasta kokonaisuudesta. Palvelu muodostuu useasta komponentista. Komponentit ovat ReactJS:ällä käytettäviä osia, jotka mahdollistavat niiden uudelleen käytön ja kokonaisuuksien pilkkomisen pienempiin osa-alueisiin. (Katso 2.4 Rakenne). Palvelua suunniteltaessa pyrittiin rakentamaan komponentit niin, että niitä voisi käyttää useassa eri kohteessa eri tarpeisiin.

Palvelulla on seuraavat päätoiminnot:

- Kartoitustyökalu
- Raportointityökalu
- Lähetystyökalu
- Seurantatyökalu
- Yrityksen hallinta ja kumppanit

2.3 Suunnittelu

Palvelu suunniteltiin marraskuun alussa 2018 käydyssä palaverissa, jossa käytiin läpi tarpeita sekä tarvittavat ominaisuudet. Käytettävät tekniikat päätettiin myöhemmin käydyssä määrittelypalaverissa. Määrittelypalaverissa myös rajattiin ominaisuudet, joita pystyttiin toteuttamaan määritetyn aikataulun mukaan. Ensimmäisen beta-version täytyi valmistua 21.12.2018.

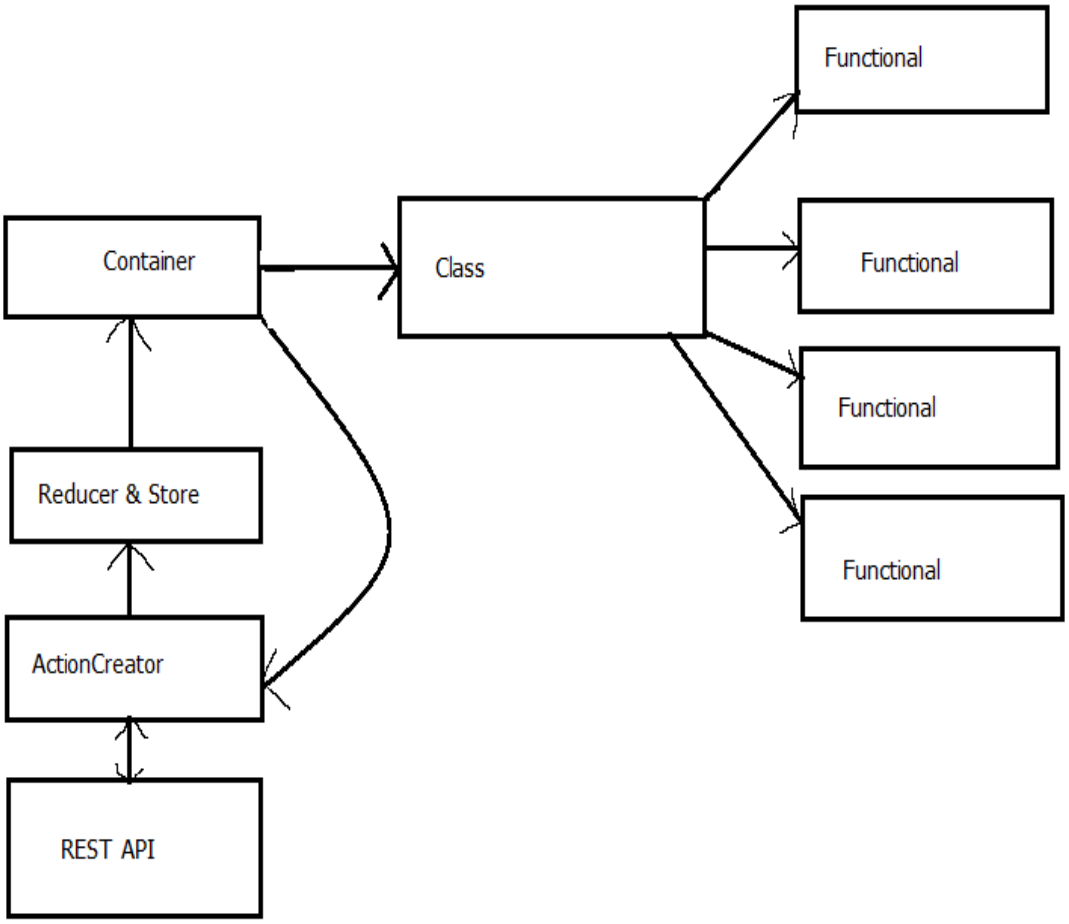
Aikataulurajoituksesta johtuen monia ominaisuuksia jouduttiin jättämään pois, mutta tässä dokumentissa mainitut ominaisuudet on rakennettu tähän beta-versioon (Katso kohta Toiminnot).

Huomioitavaa	Syy
Ominaisuuksien rajaaminen	Toteutusvaiheessa vain vähän päälle kuukausi aikaa, pois jätetyt ominaisuudet lisätään myöhemmin
Palvelun kuormitus	Palvelussa voi olla samaan aikaan useita käyttäjiä ja jokaiselle täytyy pystyä tarjoamaan dataa nopeasti
Valitut tekniikat	Tekniikat valittu niin, että päästään haluttuun päämäärään. Responsiivisuuteen ja datan nopeaan liikenteeseen.

TAULUKKO 1. Huomioon otettavat asiat.

2.4 Rakenne

Palvelun rakenne on laadittu niin, että jokaisella osa-alueella on omat komponentit. Kokonaisuudet voivat myös käyttävää mahdollisuuksien mukaan uudelleen käytettäviä komponentteja, joissa vain näytettävä data muuttuu. Listat, listassa näytettävät elementit, painikkeet jne. ovat helposti hyödynnettävissä edellä mainitusta syystä.



KAAVIO 1. Esimerkki rakenteesta. Top to Bottom.

Kaavio 1 ei näytä miten todellinen toteutus on tehty, tämän tarkoitus on vain havainnollistaa toimintamallia. Mallissa Redux hoitaa kutsut ja vastaanottaa dataa REST API:lta, ActionCreator sisältää määritellyt API rajapinnat, joita komponentti voi kutsua. Ne voivat myös sisältää erilaisia funktioita, joilla muokataan redux-storessa olevaa tilaa ilman että kutsuja lähetetään REST API:lle.
Redux on liitetty Containerille käyttäen hyväksi Reduxin tarjoamaa *connect-funktiota.*

Kun ActionCreatorissa määritelty tapahtuma on valmis, tieto lähetetään *redux-storelle* ja tila päivittyy. Container-komponentti päivittyy uudella tilalla, jolloin se lähettää tiedot eteenpäin tässä tapauksessa Class-komponentille.

Class-komponentilla on tässä tapauksessa useita Functional-komponentteja ns. *children-komponentteina*, joiden tehtävänä on renderoida pelkästään html, määritellyllä ja saadulla datalla. Class-komponentille on myös tuotu props:ien mukana dispatch-funktioita, jotka on määritelty Container-komponentissa.

Container-komponentissa määritellyt dispatch-funktiot, osoittavat ActionCreatorissa oleviin *actioneihin*, jotka kuuluvat sen vastuualueelle. Näitä hyväksi käyttäen redux-storen tietoja voidaan päivittää niiden tarvitsemilta osa-alueilta ja kun redux-store saa uuden tilan ne valuvat ylhäältä alaspäin numerojärjestyksessä:

1. Container
2. Class
3. Functional

Tätä periaatetta kutsutaan myös sanoilla *Top to Bottom*.

3 TEKNIIKAT

3.1 REST API

REST API:n tehtävä on tarjoilla dataa Restful periaatteella palvelulle sen sitä pyytäessä, käyttäjän oikeuksien sallimissa rajoissa. Palvelu on rakennettu käyttäen Play-framework versiota 1.5.3 ja tietokantana toimii MongoDB. Play-framework ja MongoDB osaltaan valittiin aikaisempien kokemuksien perusteella.

MongoDB valittiin sen takia, että se on ns. noSQL kanta, eli **schema less,** siihen voidaan tallentaa dataa vapaasti ja tarpeen mukaan. Lisäksi se skaalautuu paremmin kuin SQL-kannat.

REST API:ssa hyödynnetään @before funktiota suojaamaan rajapinnat. Näin alustettu funktio suoritetaan aina ensin ennen kuin mihinkään rajapintoihin edetään. Jokainen kutsu menee validaattorin läpi, joka tarkastaa tokenin aitouden (Jjwt, Java JWT: JSON Web Token). Jos tämä epäonnistuu, dataa ei tarjoilla eteenpäin. Jos kutsu menee rajapinnalle asti, haetaan tai muokataan dataa MongoDB:stä ja palautetaan se ActionCreatorille.

3.1.1 Play-Framework

Play-framework on open-source web application framework, joka on kirjoitettu käyttäen kieltä scala, mutta sitä voidaan myös käyttää muilla kielillä kuten Java. Play-framework käyttää MVC-mallia ja palvelu on rakennettu tätä periaatetta käyttäen. Ensimmäinen versio julkaistiin 2007.

Play-framework toimii palvelimena tässä työssä. Palvelin käsittelee pyynnöt ja suorittaa vaadittavat operaatiot MongoDB:ssä käyttäjän oikeuksien mukaan. Jokaisella kokonaisuudella on omat kokonaisuudet rakennettuna palvelimelle.

3.1.2 MongoDB

MongoDB on MongoDB inc. luoma noSQL-tietokanta. MongoDB tallentaa tietoa ja käsittelee sitä JSON-tyylisissä dokumenteissa. Sen etuina on sen skaalautuvuus, koska siihen voi tallentaa vapaasti dataa millä tahansa arvoilla ja MongoDB ymmärtää datatyypin automaattisesti. Näin ollen ns. schemaa ei tarvitse luoda ennen kuin dataa voidaan tallentaa. Koska palvelu vaatii suurien datamäärien käsittelyä, MongoDB soveltuu tähän erinomaisesti nopeutensa ansiosta. Ensimmäinen versio ilmestyi helmikuun 11. päivä, 2009.

MongoDB vastaa palvelussa tiedon tallentamisesta ja sen tarjoilusta palvelimen niitä pyytäessä. Vahvuutena on skaalautuvuus ja nopeus sekä ketterässä kehityksessä joutuu useasti tekemään nopeita muutoksia, jolloin uusien tietojen lisääminen olemassa oleviin paikkoihin on vaivatonta.

3.2 Redux

Redux on open-source javascript-kirjasto, jolla hallitaan sovelluksen tilaa. Sitä käytetään useimmiten kirjastojen kuten ReactJS tai Angularin kanssa luomaan datan ennalta-arvattavuutta. Sen etuina on sen pieni koko. Kehittäjänä toimivat Dan Abramov sekä Andrew Clark ja sen ensimmäinen versio ilmestyi kesäkuun 2. päivä, 2015.

Redux tarjoaa ns. datan ennalta-arvattavuutta. Siinä päätetään itse mitä dataa se tarjoaa ja mille osa-alueille. Tässä palvelussa Redux suorittaa kaikki kutsut REST API:lle, tarjoaa dataa vain Container-komponenteille ja ne tarjoavat datan varsinaisille ReactJS komponenteille.

Itse käyttöliittymässä tapahtuvat muutokset jätettiin näiden komponenttien vastuulle. Kun käyttöliittymässä tapahtuvat muutokset tallennetaan, ne lähetetään takaisin Container-komponenteille, jotka tarjoavat datan takaisin ActionCreatoreille joiden tehtävä on lähettää data REST API:lle tallennettavaksi. Kun *actionit* ovat valmiit redux-store päivittyy, joka johtaa sovelluksen uuteen tilaan.

3.2.1 Provider

Provider-komponentin tehtävä on jakaa redux-storen tilaa sen sisälle määritetyille komponenteille, jotka ovat liitettynä redux-storeen *connect*-funktiolla. Käytännössä kaikki komponentit sovelluksessa voisivat olla liitettynä redux-storeen.

Normally, you can't use a connected component unless it is nested inside of a `<Provider>`. (Redux Documentation 15.12.2020). Siis normaali tilanteessa liitetyn komponentin käyttö ei onnistu ilman Provider-komponenttia ja tästä syystä Provider-komponentti usein määritellään ylimmälle tasolle ja se ympäröi koko sovelluksen.

```
ReactDOM.render(<Provider store={store}><Router><App /></Router></Provider>, document.getElementById('root'));
```

KUVA 1. <Provider> komponentin käyttö.

Kuvassa 1. Provider-komponentille viedään propseina *redux-store*, joka on määritelty muuttujaan *store.* Store sisältää koko redux-storen ja se ympäröi koko sovelluksen, jolloin se tarjoaa sovelluksessa kaikille komponenteille mahdollisuuden käyttää *connect*-funktiota.

3.2.2 Redux-Store

Reducer on funktio, jonka tehtävä on havaita muutokset sovelluksen tilassa. Reducer käyttää hyväksi *actioneita*, jotka määrittävät mitä tulisi päivittää ja miten. Ensisijainen päivittäminen tapahtuu reducerissa, mutta sovellus saa tilakseen sen palauttaman arvon. Reducerista tai reducereista taas koostuu varsinainen Redux-store. Liian isot reducerit voivat aiheuttaa ylläpito ongelmia ja jokaiselle osa-alueelle on hyvä olla oma reducer, jos sovellus on laajempi.

```
1   import * as ActionTypes from '../actions/example/Actions/ActionTypes';
2
3   const initialState = {
4       dataList: [],
5       exampleTextOne: "Tällä komponentilla on pääsy omaan tilaan",
6       exampleTextTwo: "Esimerkki 2"
7   }
8
9   const reducer = (state = initialState, action) => {
10      switch(action.type) {
11          case ActionTypes.STORE_EXAMPLE:
12              return {
13                  ...state,
14                  dataList: [...action.payload]
15              }
16          case ActionTypes.HANDLE_CHANGE:
17              return {
18                  ...state,
19                  ...action.payload
20              }
21          default:
22              return state
23      }
24  }
```

KUVA 1. Esimerkki reducer.

Reducer käsittelee *actioneita*, switch-case periaatteella, mutta tässä voi käyttää myös muita tapoja rajaamaan sopivaa tapausta. Reducer vaatii parametreinä *state* ja action parametrit. State on reducerin nykyinen tila ja action sisältää tulevien tapahtumien tiedot.

Action voi sisältää siis tyypin ja datan, näistä ensimmäinen on pakollinen. Kun reducer havaitsee, että jokin komponentti on lähettänyt (*dispatch)* kutsun, reducer vertaa sitä olemassa määriteltyihin tapauksiin. Kun ehto täyttyy, reducerin tilaa päivitetään määritellyllä tavalla ja palautuva arvo päivittyy sovelluksen tilaksi. Oletuksena palautetaan olemassa oleva tila.

Reduxissa täytyy muuttaa tilaa niin, ettei alkuperäiseen tila-objektiin kosketa. Kuva 1. osoittaa miten tilaa voidaan asettaa eri *actioneissa*. On äärimmäisen tärkeää, että reducer pidetään puhtaana ja siinä ei tehdä operaatioita, jotka aiheuttavat sivuoireita. Se vain määrittää uuden tilan ja palauttaa

sen. Reducerissa ei siis tehdä pyyntöjä REST API:lle. Tällä saavutetaan toimintavarmuus sekä datan ennakoitavuus.

Redux-store luodaan hyödyntäen funktiota *createStore*, joka alustaa Reduxin tilan sille annetulla reducerilla. Sovelluksessa tulisi olla vain yksi reducer.

createStore(Creates a Redux store that holds the complete state tree of your app. There should only be a single store in your app.
(React Redux Documentation 15.12.2020.)

Jos sovelluksessa halutaan käyttää useita reducereita, tulisi siihen hyödyntää funktiota *combineReducers*. Funktio tarjoaa mahdollisuuden yhdistää sovelluksessa olevat reducerit yhdeksi isoksi Redux-storeksi. Palautuva arvo tarjotaan *createStore*-funktiolle parametrinä.

createStore(Creates a Redux store that holds the complete state tree of your app. There should only be a single store in your app. use `combineReducers` to create a single root reducer out of many. (Redux documentation 15.12.2020.).

3.2.3 Connect

Container-komponentit palvelussa ovat yhdistettynä redux-storeen käyttäen connect-functiota, joka palauttaa uuden, yhdistetyn komponentin.

React Redux provides a `connect` function for you to connect your component to the store. (Redux 6.x Documentation 14.12.2020.)

Connect on siis Container, joka ympäröi sille parametrinä annetun komponentin antaen sille mahdollisuuden suorittaa tapahtumia ja pääsyn dataan redux-storessa. Connect-funktiolle voi myös antaa parametreinä **mapStateToProps-** ja **mapDispatchToProps-** funktiot

- **mapStateToProps** – On ensimmäinen parametri. Tällä funktiolla määritetään määritetään liitetyn komponentin tarvitsemat muuttujat.

  `mapStateToProps` is used for selecting the part of the data from the store that the connected component needs. (Redux 6.x Documentation 14.12.2020.).

  - Kutsutaan jokaisella kerralla, kun redux-storen tila muuttuu
  - Saa arvokseen koko redux-storen tilan, mutta palauttaa vain määritellyt muuttujat
- **mapDispatchToProps** – On toinen parametri. Funktio siis antaa komponentille mahdollisuuden käyttää dispatch-funktiota. Dispatch mahdollistaa redux-storen tilan muutokset.

`dispatch` is a function of the Redux store. You call `store.dispatch` to dispatch an action. This is the only way to trigger a state change. (Redux 6.x Documentation 14.12.2020.).

- Perusarvona liitetty komponentti saa dispatch funktion props:ien mukana. Komponentti voi suorittaa tapahtumia *props.dispatch:n avulla*
- Jos **mapDispatchToProps** on connect-funktion parametrinä, sen avulla voi määrittää funktioita, jotka lähettävät pyyntöjä redux-storelle ja nämä funktiot voidaan antaa props:eina valittuihin komponentteihin

```
export default connect(mapStateToProps, mapDispatchToProps)(ExampleContainer);
```

KUVA 2. *connect-funktion* toteutus.

Kun komponentti on yhdistetty redux-storeen, komponentilla on pääsy koko redux-storeen. Komponenteilla tulisi kuitenkin olla määritellyt rajat, mikä data ja mitkä kutsut kuuluvat niiden osa-alueelle. Tähän tarkoitukseen soveltuvat mapStateToProps- ja mapDispatchToProps-funktiot.

3.2.4 Actions

Actionsit toimivat ainoana informaation lähteenä redux-storelle. Sen tehtävänä on kuljettaa informaatiota sovellukselta redux-storelle. Ne koostuvat tavallisista javascript-objekteista, joiden täytyy sisältää tyyppi, jonka perusteella toiminta tapahtuu itse reducerissa (store).

```
1  export const STORE_EXAMPLE = "STORE_EXAMPLE";
2  export const HANDLE_CHANGE = "HANDLE_CHANGE";
```

KUVA 2. Action types.

Toteutuksessa on käytetty erillisiä tiedostoja, nimeltään ActionTypes. Kyseessä on perinteinen javascript-tiedosto, jotka sisältävät reducerissa olevien *actioneiden tyypit.* Tämä ei ole vaadittava toimenpide, mutta se selkeyttää kokonaisuutta. Rajaamalla actionit omaan ActionTypes-tiedostoon, helpottaa ylläpidettävyyttä sekä selkeyttää koodia. Kuvasta 1. Voi havaita miten näitä on hyödynnetty itse reducerissa.

```
1   import * as ActionTypes from './ActionTypes';
2
3   export const storeExample = params => {
4       return {
5           type: ActionTypes.STORE_EXAMPLE,
6           payload: params
7       }
8   };
9
10  export const handleChange = params => {
11      return {
12          type: ActionTypes.HANDLE_CHANGE,
13          payload: params
14      }
15  };
```

KUVA 3. Action creator.

Varsinainen *action* tapahtuu, kun Kuvassa 3. määriteltyjä funktioita kutsutaan käyttämällä hyväksi *connect*-funktion (3.2.1 Connect) tarjoamaa *dispatch-funktiota*. Kuvassa 3. esiintyvistä funktioista voi havainnollistaa, että funktiot palauttavat tavallisen javascript-objektin, joiden täytyy sisältää tyyppi (ActionType). Javascript-objekti voi myös sisältää dataa tapahtuman sitä vaatiessa, mutta data ei ole välttämätöntä. Javascript-objekti palautuu *dispatch*-funktiolle. Dispatchin tehtävä on toimittaa tämä objekti *reducerille* joka hoitaa itse *actionin*.

3.2.5 Funktiot mapStateToProps ja mapDispatchToProps

Parametreinä *connect*-funktiolle voi antaa funktiot mapStateToProps ja mapDispatchToProps. Näiden parametrien ansiosta redux-store voidaan paloitella komponentin vaatimiin osa-alueisiin. Kyseessä on funktioita, joille voidaan määrittää mihin dataan komponentilla on pääsy sekä mitä actioneita se voi kutsua. Tällä saavutetaan ennalta arvattavuutta sekä varmistetaan, että komponentit saavat niiden vaatiman datan ja niille on määritelty asiaankuuluvat ActionCreatorit.

```
const mapStateToProps = state => {
    return {
        testOne: state.example.exampleTextOne
    }
}

const mapDispatchToProps = dispatch => {
    return {
        onTextChange: (value) => dispatch(ActionCreator.storeExample(value))
    }
}
```

KUVA 5. mapStateToProps-funktio ja mapDispatchToProps-funktio.

Kuten kuvasta 5. voi havaita, molemmat funktiot palauttavat javascript-objektin. mapStateToProps-funktio saa parametrinä redux-storen tilan kokonaisuudessaan, palautuvassa javascript-objektissa on kuitenkin vain example-lohkosta muuttuja ”exampleTextOne”. *Connect*-funktio toimittaa kyseisen

muuttujan komponentille sen *props:*eihin ja alustaa sen muuttujaan *props.testOne,* näin ollen komponentilla on pääsy tähän osaan redux-storesta.

Your `mapStateToProps` function should return a plain object that contains the data the component needs:

- Each field in the object will become a prop for your actual component
- The values in the fields will be used to determine if your component needs to re-render

(React redux documentation 15.12.2020).

Komponentille tuotaviin toimintojen määrittämiseen voidaan käyttää funktiota mapDispatchToProps. Funktio mapDispatchToProps saa parametrikseen *dispatch.* Tätä hyödyntäen voidaan määritellä komponentin vaatimat toiminnot ja ne käyttävät valmiiksi *dispatch*-funktiota.

Providing a `mapDispatchToProps` allows you to specify which actions your component might need to dispatch. It lets you provide action dispatching functions as props. Therefore, instead of calling `props.dispatch(() => increment())`, you may call `props.increment()` directly. There are a few reasons why you might want to do that (React redux documentation).

Funktiolla saavutetaan useita hyötytekijöitä:

- Kun Lähetystoiminta kapseloidaan itse toimintaan, tekee se toteutuksesta paljon selkeämmän. Luodaan ainoastaan toiminnan lähetys ja redux-store hoitaa itse tiedovirran.
- Mahdollisuus siirtää toiminnan logiikka *children*-komponenteille, jotka hyvin usein eivät ole liitettyinä. Tämä mahdollistaa useamman komponentin toimintojen lähetyksen ja komponentit pidetään tietämättöminä itse Reduxista.

3.3 ReactJS

ReactJS on javascript-kirjasto, jolla luodaan responsiivisia ja toimivia websovelluksia. ReactJS käyttää JSX (Javascript XML) joka on jatke javascript-syntaxille. Yhtenä etuna on Virtual DOM (Document Object Modal). React luo muistiin itselleen DOM:in selaimesta, vertailee eroja ja sitten tehokkaasti päivittää selaimen DOM:in samalle tasolle. Koska näkymää päivitetään tarpeen mukaan, tämä lisää suorituskykyä huomattavasti.

ReactJS oikeaoppinen käyttö vaatii kykyä pilkkoa isoja kokonaisuuksia pienempiin osa-alueisiin. Ylläpidettävyyden ja ominaisuuksien lisäämisen helpottuminen on rajoitettujen kokonaisuuksien ansiosta helpompaa, koska komponenttien muutokset eivät vaikuta muihin kokonaisuuksiin.

ReactJS sovellukset koostuvat usein Container-, Class- ja Functional-komponenteista. Jokaisella näistä on eri rooli kokonaisuudessa:

- Container-komponentti hallitsee dataa ja jakaa sitä muille sovelluksen komponenteille. Containerin tehtävä ei siis ole luoda sovelluksen näkymää, vaan sen tehtävä on auttaa muita komponentteja toteuttamaan varsinainen näkymä tulevalla datalla. (Stateful)

- Class-komponentti hoitaa pienempiä kokonaisuuksia, joita ei välttämättä haluta hoitaa reduxin kautta sekä renderoi html, määritellyllä datalla. (Stateful)
- Functional-komponentteja käytetään näyttämään sovelluksessa näytettävää dataa, eikä se voi hallita omaa tilaa (Stateless)

3.3.1 Container

Container-komponenttien vastuulla on hallita tilaa ja miten asiat toimivat. Usein Container-komponentit eivät itsessään luo näkymää, vaan data ja toiminnallisuudet tarjoillaan eteenpäin. Usein nämä komponentit ovat Stateful, eli komponentilla on kyky hallita omaa tilaa pystyäkseen tallentamaan vastaanotettavaa dataa.

Container-komponentit voidaan määrittää esimerkiksi seuraavasti:

- Hallitsevat miten asiat toimivat
- Eivät usein luo itse näkymää
- Tarjoavat dataa ja toiminnallisuutta muille komponenteille
- Ovat usein stateful

3.3.2 Class

Class-komponentteja käytetään, kun vaatimuksena on hallita komponentin tilaa. Class-komponentit ovat usein ES6-luokkia, joiden on jatkettava React-kirjaston tarjoamaa Component-luokkaa sekä käytettävä render-funktiota palauttamaan React-elementtejä. Tämä myös mahdollistaa tilan hallinnan sekä React-elämänkaaren eli voit halutessasi muokata näiden toiminnallisuutta. Usein tälle ei ole tarvetta ja niiden väärä käyttö aiheuttaa ongelmia, jotka ovat vaikeasti paikallistettavissa.

Class-komponentit voidaan määrittää esimerkiksi seuraavasti:

- Kyky hallita tilaa
- Jatkettava React.Component-luokkaa
- Käytettävä render-funktiota
- Mahdollistaa React-lifecyclet

Lifecycle menetelmät tarjoavat mahdollisuutta suorittaa tapahtumia määritellyissä pisteissä komponentin ”elämänkaarella”. Tärkeimmät menetelmät:

- shouldComponentUpdate – Antaa ohjelmoijalle mahdollisuuden estää turhat uudelleen renderoinnit komponentissa jos päivitykselle ei nähdä tarvetta.
- componentDidMount – Funktiota kutsutaan kun komponentti on lisätty selaimen DOM:in ja tätä käytetään yleensä tekemään kutsut palvelimelle.
- componentWillUnmount – Funktiota kutsutaan juuri ennen kuin komponentti tuhotaan DOM:ista, käytetään siivoamaan pois turhaan resursseja vievät asiat kuten *eventListener.*

- render – Kaikista tärkein, sillä se on pakollinen. Render-funktiota kutsutaan, kun tila muuttuu ja tämän pitäisi myös näkyä käyttäjälle.

3.3.3 Functional

Functional-komponentit koostavat sovelluksestaa ylivoimaisesti suurimman osan. Komponentit eivät voi hallita omaa tilaansa ja niillä ei ole omaa React-elämänkaarta. Näiden komponenttien vastuulla on vain näyttää data, joka komponentille tarjotaan.

Functional-komponentit voidaan määrittää esimerkiksi seuraavasti:

- Helpottavat ylläpitoa ja testausta, sillä komponentit ovat tavallisia javascript-funktioita, joten niillä ei ole omaa tilaa eikä elämänkaarta
- Vähemmän koodia
- Auttavat luomaan parempia ratkaisuja, kun tilan hallintaan ei ole mahdollisuutta
- Uudelleen käytettävyys
- Mahdolliset hyödyt myös sivuston suorituskykyyn

Functional-komponenttien vahvuus on siinä, että niitä voi vapaasti käyttää **useassa eri paikassa**. Niiden tehtävä on vain näyttää tarvittava data, muutokset tehdään pääkomponenteissa, joissa tilaa halutaan muuttaa.

3.3.4 State

State on komponentin tila. Tilaan voidaan asettaa vaaditut muuttujat, joiden tilaa halutaan hallita. Tilaa ei muuteta suoraan asettamalla uusi arvo muuttujaan, vaan tässä apuna käytetään *this.setState*-funktiota. Funktiolle parametrinä on aina javascript-objekti, jonka sisälle on määritelty mitä tietuetta päivitetään ja millä arvolla. Jos tilaa muutetaan funktiossa, sen täytyy olla yhdistettynä komponenttiin, jotta sillä on pääsy this-ulottuvuuteen. Täällä sijaitsee komponentin tila.

```
handleChange(valueToChange, value) {
    this.setState({
        [valueToChange]: value
    });
}
```

KUVA 6. handleChange näyttää yhden tavan tilan muutokselle.

```
handleChange(valueToChange, value) {
    this.state.exampleTextOne = value;
}
```

KUVA 7. Esimerkki siitä miten tilaa ei saa muuttaa.

3.3.5 Props

Props on komponentille tulevat näyttämötarpeistot. Luodun komponentin havaitessaan React toimittaa komponentille JSX-tarpeistot ja *childrenit* näyttämötarpeistona. *Childrenit* palauttavat html-elementtejä, jolloin React päivittyy.

```
import React from 'react';

const ExampleFunctional = props => {
    return (
        <div>
            <input type="text" value={props.text} onChange={(e) => props.inputHandler(props.name, e.target.value)} style={{width: "100%"}}/>
        </div>
    )
}

export default ExampleFunctional;
```

KUVA 8. Functional-komponentti jolle on toimitettu näyttämötarpeistoa.

Näyttämötarpeisto on siis keino, miten dataa voi siirtää eri komponenteille näytettäväksi. Tarpeiston mukana on myös mahdollista toimittaa funktioita sekä tarjota mahdollisuutta *actioneille*. Kuvassa 8. Functional-komponentille on toimitettu arvo (props.text) ja funktio (props.inputHandler) tämän arvon muuttamiseen *tilaan*.

Huomattavaa on, että näyttämötarpeiston muutokset eivät aiheuta sivuston päivitystä. Ainoastaan *tilan* muutos aiheuttaa sen, että React vertailee selaimen DOM:in ja sen omassa muistissa olevan Virtual DOM:in eroja.

Yksinkertaisesti tässä tapauksessa prosessi etenee seuraavasti:

1. Käyttäjä muuttaa kenttää
2. Tieto lähetetään pääkomponentille propsissa olevalla viittauksella
3. Uudet tiedot asetetaan komponentin tilaan
4. Komponentti renderoituu uudelleen
5. Propsien viittaus päivittyy

4 TOTEUTUS

Esimerkkien tehtävä oli pohjustaa tehtyjä ratkaisuja ja miten niitä on käytetty varsinaiseen toteutukseen. Tässä käydään läpi jokainen pääkokonaisuus ja miten aikaisempia on hyödynnetty rakentamaan toimiva kokonaisuus. Johdannossa jo mainittiin asiasta, mutta tämä osuus ei tule sisältämään esimerkkejä todellisesta koodista tai kuvia palvelusta. Pyritään kuitenkin kertomaan sen mitä voi, palvelun toiminnasta ja sen ominaisuuksista.

4.1 Kartoitustyökalu

Kartoitustyökalussa voidaan luoda kartoituksia (kyselyitä), jotka toimivat sähköisesti erillisessä palvelussa. Nämä kyselyt voidaan aktivoida lähetystyökalussa, kohteettomina tai yrityskohtaisesti. Luodut kartoitukset listataan näkymään, kun kartoitustyökaluun navigoidaan. Näitä kun voi olla n-kpl. Niistä on tehty oma komponentti, jota voidaan silmukassa renderoida tarvittava määrä, tarvittavalla datalla ja tarvittavilla ominaisuuksilla per kortti.

Olemassa olevan kartoituksen tai rakenteilla olevan kartoituksen näkymässä sen sisällä olevat kysymysryhmät ja kysymykset listataan sivun vasemmassa laidassa olevassa listassa. Listan renderoinnissa on käytetty hyödyksi map-funktiota (ECMAScript), jolla voidaan käsitellä listassa olevaa dataa.

- Kysymysryhmä
  - Kysymys
  - Kysymys
  - Kysymys

Tästä on helppo huomata, että nämä voidaan yhdistää hyvin omaan komponenttiin, jolle tarvitsee vain lähettää kysymysryhmän data ja se renderoi määritellysti kyseisen listanäkymän. Lisäksi jokainen kenttä saa tarvittavat funktiot toiminnallisuuteen liittyen. Kun kysymysryhmiä on n-kpl, ne renderoituvat automaattisesti samanlaisiksi kuin ne on määritelty. Jokaisen kysymysryhmän alla on n-kpl kysymyksiä, kun kysymykset renderoituvat ja seuraava kysymysryhmä löytyy, prosessi alkaa alusta. Tämä prosessi jatkuu niin pitkään, kuin kysymysryhmiä on tarjolla.

4.1.1 Kartoitus

Kartoitukselle annetaan ensin nimi ja selite. Tämän jälkeen kartoitus on valmiina luotavaksi. Sen alle luodaan kysymysryhmät, joiden alle taas luodaan kysymykset. Jo olemassa olevaa kartoitusta on mahdollisuus muokata samassa näkymässä.

4.1.2 Kysymysryhmä

Kysymysryhmällä on nimi ja selite, lisäksi sille annetaan raja-arvot mitkä muodostavat yhteyden kartoituksen analyysille. Näitä raja-arvoja voi olla n-kpl, eli ne ovat vapaasti määriteltävissä. Ainoana rajauksena on, että pisterajat eivät voi kuitenkaan ylittää sataa-pistettä (Pisteeet ovat lähempänä keskiarvoa tässä tapauksessa).

Nämä raja-arvot skaalautuvat automaattisesti niitä lisätessä. Kun raja-arvoilla on tilat 0-50 ja 51-100, uuden raja-arvon lisäys johtaa raja-arvoihin 0-50, 51-98 ja 99-100. Tämän jälkeen rajat ovat säädettävissä siten, että muiden raja-arvojen ylärajaa muutettaessa muiden alaraja-arvot muuttuvat sen mukaan. Yllä mainitussa tilanteessa 51-98, ei voida nostaa ylemmäksi, koska se pakottaisi viimeisen raja-arvon tilanteeseen 100-101.

Käytännössä raja-arvoilla on teoreettinen raja kuinka monta niitä voidaan luoda. Eli noin 50, mutta tässä tapauksessa mentäisi niin äärimmäisyyksiin, että tätä rajaa tuskin saavutetaan.

yleisin tilanne on seuraava:

1. 0-25
2. 26-65
3. 66-100

Jokaisen raja-arvon kohdalle lisätään vielä palaute, joka annetaan raportointityökalussa, jos kysymysryhmän keskiarvo osuu tälle alueelle. Lisäksi raportoinnissa käytetään värikoodeja kertomaan tulos nopeasti lukijalle.

Kuten havaitaan, data on palautteiden luonnissa samankaltaista. Aina on väri, pisterajat sekä palauteteksti. Näin ollen voidaan luoda kokonaisuus, joka voidaan lisätä aina haluttuun paikkaan n-kertaa. Jokaisen palutteen lisäämisen yhteydessä tämä kokonaisuus ilmestyy käyttäjälle ruutuun, samankaltaisena kuin aikaisemmin, ainoastaan arvot ovat erona.

Koko palvelun toteutuksessa on käytetty ReactJS periaatteita, pieniä kokonaisuuksia, joilla on selkeä tehtävä. Palvelurakennetta suunnitellessa on tärkeää havaita, mitkä ovat ne kentät, joita esiintyy n-kpl. Kun nämä on selkeästi kartoitettu, voidaan luoda pieniä kokonaisuuksia, joita liitetään haluttuun paikkaan tarpeen mukaan. Samaa näkymää siis käytetään uuden kysymysryhmän luontiin, sekä olemassa olevan muokkaukseen. Kun kysymysryhmän kentät ovat täytettyinä, on aika luoda sille kysymyksiä.

4.1.3 Kysymys

Kysymysryhmässä on voinut määrittää palautteen, jolla on tietyt raja-arvot. Kysymyksen luonnissa käytetään näitä raja-arvoja luomaan valmiiksi määritellyt rajat, jota hyväksi käytetään tarjoamaan analyysiä kysymyksen tuloksen ollessa raja-arvon sisällä. Analyysiin kuuluu väri, joka toimii indikaattorina tilanteelle, raja-arvot ja analyysi.

Kuten havaita saattaa, data-kentät ovat samanlaiset kuin kysymysryhmän palaute-tekstikentissä. Kun kokonaisuus oli oikein suunniteltu ja tämä mahdollisuus oli otettu huomioon, voidaan lisätä sama kokonaisuus tähän kuten kysymysryhmässä. Ainut ero on, että ainoastaan analyysin tekstikenttä on muutettavissa.

(State and Props) Aikaisemmin käytiin läpi ReactJS Propsit, joita hyödynnetään nyt tässä kohdassa viemään lisätietokenttä, joka aiheuttaa sen, että väriä tai pisterajoja ei voi muuttaa, vaan ne renderoidaan sellaisenaan minkä arvon ne perivät.

Analyysit näytetään raportointityökalussa, kun kysymyksen keskiarvo osuu tälle alueelle. Lisäksi väri taas ilmaisee nopeallakin vilkaisulla, millä alueella kysymys on.

Kysymykselle voi luoda n-kpl, vastausvaihtoehtoja. Vastausvaihtoehdolla on seuraavat kentät:

- Väri
- Vastausvaihtoehto
- Pisteet
- Palaute

Tässäkin tapauksessa voidaan havaita, että kentät pysyvät samoina, arvot vain muuttuvat, joten tästäkin on luotu oma kokonaisuus, jonka lisääminen palvelussa haluttuun paikkaan on mahdollista. Jokainen kerta, kun käyttäjä haluaa lisätä uuden vaihtoehdon, uusi komponentti renderoidaan aikaisempien alapuolelle. Samaa näkymää käytetään jo olemassa olevan kysymyksen muokkaamiseen ja arvot päivitetään oikeisiin paikkoihin automaattisesti jo olemassa olevilla arvoilla.

Pisteiden ja vastauksien lukumäärän avulla muodostetaan keskiarvot, joiden perusteella tarvittavat kentät täytetään raportointityökalussa. Väriä käytetään taustavärinä osoittamaan mikä vastausvaihtoehto on kyseessä. Palaute-kentästä otetaan tiedot, jotka näytetään raportoinnissa, jos tämä vaihtoehto oli enemmistö.

4.2 Lähetystyökalu

Lähetystyökalussa voidaan nimensä mukaisesti lähettää kartoitus joko yritykselle tai anonyyminä (kohteettomana). Luonnin yhteydessä kartoitukselle luodaan koodi, joka vastaa tähän lähetettyyn kyselyyn, joka taas voidaan kirjoittaa erillisessä palvelussa olevaan kenttään. Koodin ollessa oikein, kartoitus aktivoituu ja siihen voidaan vastata.

Vaadittavat kentät ovat kartoitukselle annettava nimi, kuten ”Kevään suurkysely 2019”, kohde (jos yritykselle), lähetettävät kartoitukset ja aktiivinen aika.

4.3 Seurantatyökalu

Työkalu listaa aikaisemmin lähetetyt kartoitukset ja niiden tilan. Työkalussa on erilliset näkymät, kohteettomille ja kohteellisille kartoituksille. Tilaa määritellään värikoodien avulla.

- Punainen – Ei ole vielä aktiivinen tai ei vastauksia, päättynyt ja ei vastauksia
- Keltainen – Aktiivinen ja sisältää vastauksia
- Vihreä – Päättynyt ja sisältää vastauksia

Listauksessa näytetään kartoituksen tila, nimi, koodi, aktiivinen aika ja mahdolliset toiminnot. Toiminnoista voidaan perua kartoitus. Jos se oli aktiivinen, se päättyy. Jos se ei vielä ollut aktiivinen, se katsotaan päättyneeksi. Jos kartoituksessa on vastauksia, toiminnoista voi suoraan siirtyä myös raportointityökaluun, joka näyttää kartoituksen raportin.

Kartoituksia voi suodattaa värikoodien mukaan työkalussa sijaitsevista painikkeista, joiden värit vastaavat erillisiä tiloja. Kun käyttäjä painaa Vihreää-nappia, näytetään ainoastaan kartoitukset, joiden tila on Vihreä. Jos yksikään suodatin ei ole päällä, palvelu näyttää kaikki lähetetyt kartoitukset. Suodatuksessa on käytetty hyväksi filter- ja map-funktiota (ECMAScript).

Jokainen ehdon täyttänyt objekti palautetaan näkymään ja ne, jotka eivät täytä ehtoa poistetaan näkymästä. Filter-funktio luo siis uuden listan ehdon täyttäneistä arvoista, jonka jälkeen map-funktiolla lista käydään läpi.

Toistaiseksi alkuperäisenä listana on aina kaikki lähetetyt kartoitukset. Myöhemmin kartoituksien määrän kasvaessa, voimme hajottaa tämän erillisiin pyyntöihin, jolloin dataa ei ole niin paljon renderoitavana, eikä koko listaa tarvitse aina käydä läpi eri tilojen aikana. Tämä on yksi tulevaisuuden optimointitarpeista, mutta toistaiseksi sille ei ole tarvetta.

4.4 Yrityksen hallinta ja kumppanit

Työkalussa voi hallita yrityksen tietoja ja lisätä esimerkiksi lisää käyttäjiä palveluun. Työkalu myös mahdollistaa sen, että konsulttiyritys rekisteröi asiakasyrityksen palveluun tätä kautta, jolloin he voivat lähettää heille tarvittaessa kartoituksia. Heillä on myös mahdollisuus luoda käyttäjiä yrityksen palvelun puolelle pelkkää sähköpostia käyttäen. Enterprise-puolella ei voida muokata, tai nähdä yrityksen käyttäjätunnuksia tämän prosessin jälkeen.

4.5 Raportointityökalu

Raportointityökalussa näytetään erilaisia raportteja kartoituksen tuloksista. Ensin valitaan yritys, jonka tuloksia halutaan tarkastella, jonka jälkeen palvelu hakee kaikki kartoitukset, joissa on raportoitavaa. Listauksessa ei näy kartoitukset, joissa ei ole vastauksia ollenkaan. Näytettävään raporttiin voi valita n-kpl kartoituksia, jotka yhdistetään yhdeksi kokonaisuudeksi.

Nykyisellään työkalussa on kolme eri näkymää, joita voi vapaasti muuttaa. Näkymät on luotu niin, että tarvittava data saadaan nopeasti hyödynnettyä ja se on helppo lukuista.

Kahdessa ensimmäisessä näkymässä näytetään tulokset graafisesti, eli väripalkein ja mille alueelle tulos asettuu. Siitä on karsittu kaikki ylimääräinen pois, eli vain kategorian nimi, selite ja kysymysryhmien nimet sekä tulokset näkyvät selkeästi.

Kolmas näkymä tarjoaa enemmän dataa käyttäjälle, jos haluaa selkeän kuvan mistä tulos on muodostunut, mitkä on lisätyt muistiinpanot (jatkotoimenpiteet) ja kaikkien osa-alueiden palautteet.

Näkymät on rakennettu niin, että tieto on jo käyttöliittymällä näkymiä vaihdellessa, sitä vain käsitellään eri näkymissä eri tavoin. Tämä mahdollistaa käyttöliittymän nopean renderoinnin. Tämä siksi, että tietoa ei haeta erikseen näkymille jokaisella kerralla, kun näkymää vaihdetaan. Tämä säästää resursseja ja lisää käyttökokemusta, kun käyttäjä ei joudu odottamaan kutsun päättymistä näkymiä vaihdellessa. Todellisuudessa MongoDB ja Play-framework toteuttaa nykyisellään kutsut niin nopeasti, että tämäkin olisi täysin mahdollista tehdä ilman, että käyttökokemus kärsii.

Raportin laskennassa MongoDB nopeus on yksi tärkeimmistä avaintekijöistä, miksi raportin lasku ja käyttäjälle näyttäminen tapahtuu niin nopeastin. Todisteena raportin laskennan ja renderoitumisen välisestä nopeudesta oli, että testikäyttäjiltä tuli jopa virheilmoitus, että näkymä ei päivittynyt, jos raportteja yhdistetään. Todellisuudessa, koska näkymässä vain osa kentistä päivittyy, käyttäjät eivät huomanneet tätä prosessia, joten harkitsimme jopa keinotekoista ”hidastamista”. Tämä olisi kuitenkin ollut turha prosessi, joten ilmoitusta raportin uudelleen luomisesta parannettiin niin, että se olisi selkeämpi ja kertoisi että kaikki raportit, jotka ovat valittuina olisivat selkeästi ilmoituksessa esillä.

4.5.1 Vertailutyökalu

Vertailutyökalu sisältyy raportointityökaluun. Sen avulla voidaan verrata kahta, tai useampia samankaltaista kartoitusta. Esimerkiksi kartoitus A on lähetetty jouluna 2018 ja tarve olisi nähdä tulokset selkeästi vertailtuna kartoitukseen B, joka lähetettäisiin kesällä 2019, työkalu muodostaa helppolukuisen näkymän.

Aikaisemmin kuin tulokset näytettiin keskiarvoina ja niiden perusteella lasketulla tuloksella. Työkalu taas vaihtaa näkymän niin, että tuloksesta näkyy suoraan, onko muutos ollut positiivista vai negatiivista ja miten suurta vaihtelua on tapahtunut.

Työkalussa ei ole mitään raja-arvoa, miten monta kartoitusta otetaan kummallekin puolelle vertailua. Kun käyttäjä suorittaa valintoja kartoituksista A, lista kartoituksista B suodattuu automaattisesti, niin että käyttäjän on mahdollista nähdä, onko niillä yhtäläisyyksiä ja näkymästä poistuu kartoitukset, joista tätä yhteyttä ei löydetä.

Kun varsinaista vertailua aloitetaan suorittamaan, suoritus tapahtuu siten, että jos kartoituksien A ja kartoituksien B välillä löydetään yhtäläisyyksiä, ne otetaan vertailuun mukaan ja loppuosa mistä yhtäläisyyksiä ei löydy hylätään.

5 YHTEENVETO JA POHDINTA

Opinnäytetyöni tarkoituksena oli suunnitella ja luoda Qartoitus Enterprise Portal, annettujen vaatimuksien perusteella. Portaalissa on mahdollista luoda kartoituksia, lähettää niitä ja lopulta analysoida tulokset. Analysoinnissa helpottaa nopeasti tulokset kertova raportointi, joka on helppo lukuinen ja antaa tarvittavan tiedon heti, eikä sitä tarvitse etsiä.

Työ eteni odotetusti ja saavutin kaikki tavoitteet, jotka palvelulle oli alkuun asetettu. Aikataulu oli erittäin raju, mutta hyvällä suunnittelulla ja keskittymällä kokonaisuuksiin oikeassa järjestyksessä työ eteni huomattavan nopeasti.

Redux oli minulle täysin tuntematon ennen projektin alkua. Olin kuullut siitä vain ohimennen, mutta koskaan sitä en ollut käyttänyt. Nykyään pohdin, että miten sitä on ennen ilman pärjännyt.

Tämän näkisin suurimpana haasteena työn suorituksessa. Uusi tekniikka, jota soveltaa tiukan aikataulun aikana, kokonaan uuden palvelun luomiseen. Itse olin tosin päätöksen takana ottaa se käyttöön ja se lisäsi mielenkiintoa projektia kohtaan, kun joka päivä oppi jotain uutta. Suosittelen kaikille vahvasti tutustumaan aiheeseen ja kokeilemaan sen käyttöä.

Nykyisellään tätä päivittäessä 16.4.2019, palvelu on kehittynyt huomattavasti tästä dokumentaatiosta ja olen mukana edelleen kehityksessä.

LÄHTEET

REACT, React-DOM (Luettu 15.12.2020, Viitattu 15.12.2020) Saatavissa:

https://reactjs.org/docs/react-dom.html

REACT, Component and props (Luettu 15.12.2020, Viitattu 15.12.2020) Saatavissa:

https://reactjs.org/docs/components-and-props.html

REACT, State and lifecycle (Luettu 16.11.2018) Saatavissa:

https://reactjs.org/docs/state-and-lifecycle.html

REACT, Thinking in react (Luettu 16.11.2018) Saatavissa:

https://reactjs.org/docs/thinking-in-react.html

REDUX, Core Concepts (Luettu 1.12.2018) Saatavissa:

https://redux.js.org/introduction/core-concepts

REDUX, Store (Luettu 1.12.2018, Viitattu 14.12.2020) Saatavissa:

https://redux.js.org/api/store

REDUX, Create store (Luettu 1.12.2018, Viitattu 14.12.2020) Saatavissa:

https://redux.js.org/api/createstore

REDUX, Combinereducers (Luettu 14.12.2020, Viitattu 14.12.2020) Saatavissa:

https://redux.js.org/api/combinereducers

REACT-REDUX, connect (Luettu 14.12.2020, Viitattu 14.12.2020) Saatavissa:

https://react-redux.js.org/6.x/api/connect

REDUX, QUICK-START. Virallinen dokumentaatio. (Luettu 1.12.2018) Saatavissa:

https://redux.js.org/introduction/getting-started

REACT-REDUX, Provider (Luettu 14.12.2020, Viitattu 14.12.2020) Saatavissa:

https://react-redux.js.org/api/provider

REACT-REDUX, Connect: Extracting data with mapStateToProps (Luettu 14.12.2020, Viitattu 14.12.2020)

https://react-redux.js.org/6.x/using-react-redux/connect-mapstate

REACT-REDUX, Connect: Dispatching actions with mapDispatchToProps (Luettu 14.12.2020, Viitattu 14.12.2020)

https://react-redux.js.org/6.x/using-react-redux/connect-mapdispatch

REACT-REDUX, Quick-start. Luettu (16.11.2018). Saatavissa:

https://react-redux.js.org/introduction/quick-start

REDUX-THUNK, GITHUB. Dokumentaatio ja esimerkit. (Luettu 1.12.2018) Saatavissa:

https://github.com/reduxjs/redux-thunk

REACTSTRAP. Dokumentaatio ja esimerkit. (Luettu 1.12.2018) Saatavissa:

https://reactstrap.github.io/

MONGODB, DOCUMENTATION. MongoDB 4.0 virallinen dokumentaatio (Luettu 16.04.2019). Saatavissa:

https://docs.mongodb.com/manual/

PLAY 1.5.x, DOCUMENTATION. Play-frameworkin virallinen dokumentaatio, versioille 1.5.x. (Luettu 1.12.2018) Saatavissa: https://www.playframework.com/documentation/1.5.x/home