Emma Syynimaa

KÄYTTÖLIITTYMÄSOVELLUS REKISTERÖINTIASEMALLE

Tieto- ja viestintätekniikan koulutusohjelma

2019

samk
Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

KÄYTTÖLIITTYMÄSOVELLUS REKISTERÖINTIASEMALLE

Syynimaa, Emma
Satakunnan ammattikorkeakoulu
Tieto- ja viestintätekniikan koulutusohjelma
toukokuu 2019
Sivumäärä: 41

Asiasanat: ohjelmistokehitys, käyttöliittymät, sovelluskehykset

---

Tämän opinnäytetyön tarkoituksena oli tuottaa Cimcorp-konsernille automaattisen varastointi- ja keräilyjärjestelmän sisältämille rekisteröintiasemille käyttöliittymä.

Tavoitteena oli saavuttaa konsernin muiden käyttöliittymien kanssa tyylillisesti yhteensopiva responsiivinen web-käyttöliittymäsovellus, jonka tuli toimia vaivatta yhdessä varastonohjausjärjestelmän kanssa. Sovellus tuli toteuttaa käyttämällä määriteltyjä työkaluja, ja sen tuli noudattaa sille annettua vaatimusmäärittelyä, niin toiminnallisesti, kuin tyylillisestikin.

Tässä opinnäytetyössä käydään läpi työn lähtökohdat, käyttöliittymässä käytetyt tekniikat ja työkalut, sovelluksen rakenne ja toiminta, sovelluksen testaus sekä lopullinen saavutettu tulos.

Opinnäytetyö tehtiin Cimcorp-konsernin tuotekehitysosastolla.

USER INTERFACE FOR DEWRAPPING STATION

Syynimaa, Emma
Satakunnan ammattikorkeakoulu, Satakunta University of Applied Sciences
Degree Programme in information and communication teghnology
May 2019
Number of pages: 41

Keywords: software development, user interfaces, application frameworks

---

The purpose of this thesis was to provide Cimcorp Group with a user interface for the dewrapping stations contained in the automatic storage and retrieval system.

The goal was to achieve a responsive web interface application which could communicate with and worked smoothly together with the warehouse control system. The application had to be implemented using specified tools, and it had to comply with the required specification, both technically and stylistically.

This thesis covers the starting point of the work, the techniques and tools used in the interface, the structure and functions of the application, testing the application and the ultimate results.

The thesis was carried out at Cimcorp's product development facility.

SISÄLLYS

1 JOHDANTO

Cimcorp-konserni on vuonna 1975 perustettu logistiikka-automaatiojärjestelmien toimittaja. Konserni tytäryhtiöineen työllistää noin 400 henkilöä. Pääkonttori sijaitsee Suomessa, Ulvilassa, ja tytäryhtiöt Kanadassa, USA:ssa ja Intiassa. Cimcorp-konsernin osakekanta kokonaisuudessaan, on vuodesta 2014 asti ollut japanilaisen logistiikka-automaatio toimittajan Murata Machinery, Ltd:n omistuksessa. (Cimcorp intranet 2019)

Cimcorp toimittaa asiakkailleen muun muassa automaattisia varastointi- ja keräilyjärjestelmiä, jotka ovat räätälöitävissä asiakkaiden tarpeitten mukaan. Tämän opinnäytetyön tarkoituksena, oli tehdä konsernin uuteen, kehitteillä olevan automaattisen varastointi- ja keräilyjärjestelmän rekisteröintiasemille web-käyttöliittymäsovellus, joka toimii yhdessä varastonohjausjärjestelmän kanssa. (Cimcorp 2019)

Kehitteillä oleva varastointi- ja keräilyjärjestelmä koostuu useista varastomoduuleista, eli keskenään samankaltaisista, mutta toisistaan riippumattomista varastointitiloista, joissa jokaisessa on oma rekisteröintiasemansa. Varastomoduulit saattavat vaihdella keskenään, esimerkiksi lämpötilojensa osalta. Kaikki varastoon saapuneet lavat kulkevat oman moduulinsa rekisteröintiaseman kautta, ennen varsinaista varastointia varastoon.

Rekisteröintiasemalla operaattori, eli käyttäjä, poistaa varastoon saapuneen lavan muovikelmut, ja tarkistaa manuaalisesti, vastaako lavan sisältö rekisteröintinäytön näyttämää dataa. Jos kaikki on niin kuin pitääkin, operaattori hyväksyy lavan, ja se jatkaa matkaansa varastoitavaksi.

Jos saatu data on puutteellista tai virheellistä, eikä se ole rekisteröintiasemalla korjattavissa, operaattori hylkää lavan. Hylkäyksen jälkeen lava siirtyy hylkykuljettimelle, ja palaa takaisin ulos. (Rajakangas & Jantunen 2019, 15-21.)

2 LÄHTÖKOHDAT

2.1 Käyttöliittymän suunnittelu

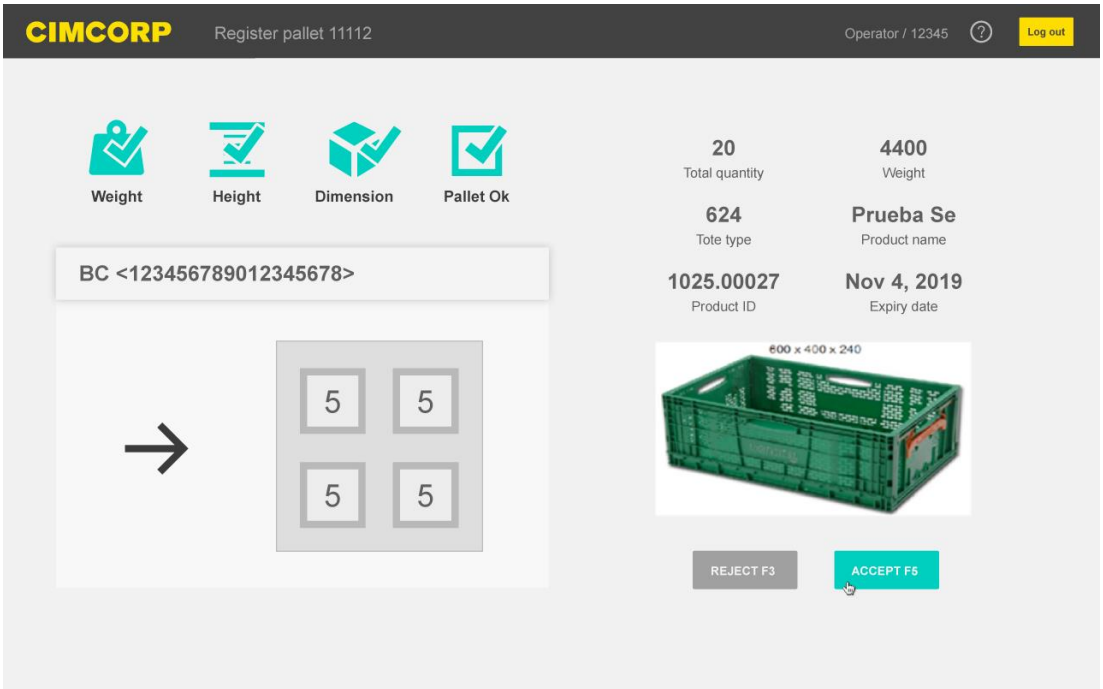
Sovellusta suunniteltaessa otettiin huomioon sekä asiakkaan, että Cimcorpin vaatimukset. Käyttöliittymän tuli oli olla selkeä ja yksinkertainen, sekä tyylillisesti yhteensopiva muiden varastonhallintajärjestelmän käyttöliittymien kanssa.

Asiakkaan vaatimusten mukaan, sovelluksen yläreunassa olevalla valintarivillä ei saanut olla muita painikkeita, kuin uloskirjautuminen ja kielenkääntö. Tällä pyrittiin pitämään sovelluksen käyttö yksinkertaisena, ja helposti opittavana.

Haasteena oli näyttää paljon tietoa, ja tiedon piti olla helposti havaittavissa. Profiilitarkistuksen tuloksia kuvaamaan suunniteltiin havainnollistavat ikonit. Nämä ikonit viestivät operaattorille värikoodeilla tarkistuksen tuloksen. Käyttöliittymän viestimä data on esitetty havainnollisesti, ja sijoitettu loogisesti ohjaamaan operaattorin toimintaa.

Käyttöliittymän suunnittelun toteutti Cimcorp-konsernin graafinen suunnittelija, Suvi Talvitie. Ulkoasu tuli toteuttaa alla olevan kuvan (kuva 1) mukaisesti, jotta se olisi ulkonäöltään yhteensopiva muiden varastonhallintajärjestelmän käyttöliittymien kanssa. (Latva henkilökohtainen tiedonanto 9.1.2019)

Tähän opinnäytetyöhön ei lukeutunut lainkaan käyttöliittymänsuunnittelua, vaan noudatettiin aiemmin kuvattuja ohjeita.



Kuva 1. Suunnitelma käyttöliittymän ulkoasusta (Talvitie 2019)

2.2 Toiminnalliset vaatimukset

Alkuperäisen suunnitelman mukaan, työhön lukeutui käyttöliittymän toteutuksen vaihtoehtoisten ratkaisujen tutkiminen, Java desktopin ja web-käyttöliittymän välillä. Ennen työn aloittamista, tuotekehitysosasto oli kuitenkin jo päätynyt näistä kahdesta vaihtoehdosta web-käyttöliittymään, sen mukautuvuuden vuoksi. On mahdollista, että käyttöliittymää tullaan jatkossa käyttämään mobiililaitteilla, joten web-käyttöliittymä oli tässä tilanteessa parempi ratkaisu.

Tavoitteena oli saada selkeä ja toimiva käyttöliittymäsovellus rekisteröintiasemille, joka toimisi sujuvasti yhdessä muiden ohjelmistojen kanssa.

Käyttöliittymän tuli olla responsiivinen, eli käytettävissä erikokoisissa näytöissä ilman, että ulkonäkö muuttuisi käyttöä haittaavaksi.

Sovellukseen tuli sisältyä myös viivakoodin lukemisen mahdollistava ominaisuus. Tämä ominaisuus on tarpeellinen tilanteissa, joissa operaattorin tarvitsee syystä tai toisesta, lukea viivakoodi uudelleen. (Rajakangas & Jantunen 2019, 15-21.)

2.3 Toiminta

Ennen varsinaiselle rekisteröintiasemalle saapumista, jokainen varastoon saapuva lava kulkee automaattisen tarkistuspisteen kautta. Tarkistuspisteellä luetaan saapuneen lavan viivakoodi, sekä tehdään lavan profiilitarkistus antureita käyttämällä. Luetusta viivakoodista saadaan lavan tuotetiedot, joiden mukaan saapuva lava osataan ohjata oikeaan varastomoduuliin.

Profiilitarkistus koostuu lavan pinojen korkeuden mittauksesta, lavan kokonaismassan punnituksesta, lavalla olevien pinojen ulkosivujen tarkistamisesta sekä itse lavan kunnon tarkistamisesta. Jokainen saapuva lava on tuotepuhdas, eli sisältää ainoastaan yhtä tuotetta.

Viivakoodinluku ja profiilien tarkistaminen ovat täysin automatisoituja, eikä tarkistuspisteellä vaadita ihmisen läsnäoloa.

Jos tarkistuspisteen viivakoodinluku tai profiilien tarkistaminen epäonnistuu, ei lava jatka matkaansa rekisteröintiasemalle, vaan siirtyy hylkäyskuljettimelle. Jos taas kaikki tarkistukset saadaan toimitettua oikein, lähetetään lavasta saatu data API:n (Application programming interface), eli ohjelmointirajapinnan kautta tietokantaan, ja lava siirtyy kuljettimella rekisteröitäväksi. Rekisteröintiasemalla käyttöliittymä tekee kyselyn API:n kautta, ja näyttää saamansa datan operaattorille rekisteröintinäytössä.

Operaattori tarkistaa lavan tietojen oikeellisuuden näytöltä, ja joko hylkää tai hyväksyy lavan.

Jos näytön näyttämä data on virheellistä, mutta virhe on kuitenkin korjattavissa, korjaa operaattori tiedot asiakkaan toiminnanohjausjärjestelmässä, ja lukee viivakoodin uudelleen. Lavan päivitetyt tiedot siirtyvät järjestelmään, ja lava voidaan hyväksyä.
Jos lava joudutaan hylkäämään, antaa operaattori hylkäykseen johtaneen syyn käyttöliittymässä. Rekisteröinnin tulos lähetetään takaisin API:n kautta tietokantaan.

Rekisteröinnin tuloksen mukaan, lava siirtyy joko hylkäyskuljettimelle tai jatkaa matkaansa varastoon. (Rajakangas & Jantunen 2019, 15-21.)

2.4 Työn aikataulutus

Käyttöliittymän valmistumiselle ei annettu tarkkaa aikataulua. Joustavan aikataulun mukaan, sen tuli olla valmis kesään 2019 mennessä.

3 KÄYTETYT TYÖKALUT JA TEKNIIKAT

Käyttöliittymää tehdessä käytettiin useita erilaisia työkaluja ja tekniikoita, niistä merkittävimmät esitellään tässä opinnäytetyössä. Alla esiteltävät työkalut ja tekniikat eivät ole ainoastaan tähän työhön valittuja, vaan ovat jatkuvassa käytössä tuotekehityksen web-tiimissä.

3.1 Visual Studio Code

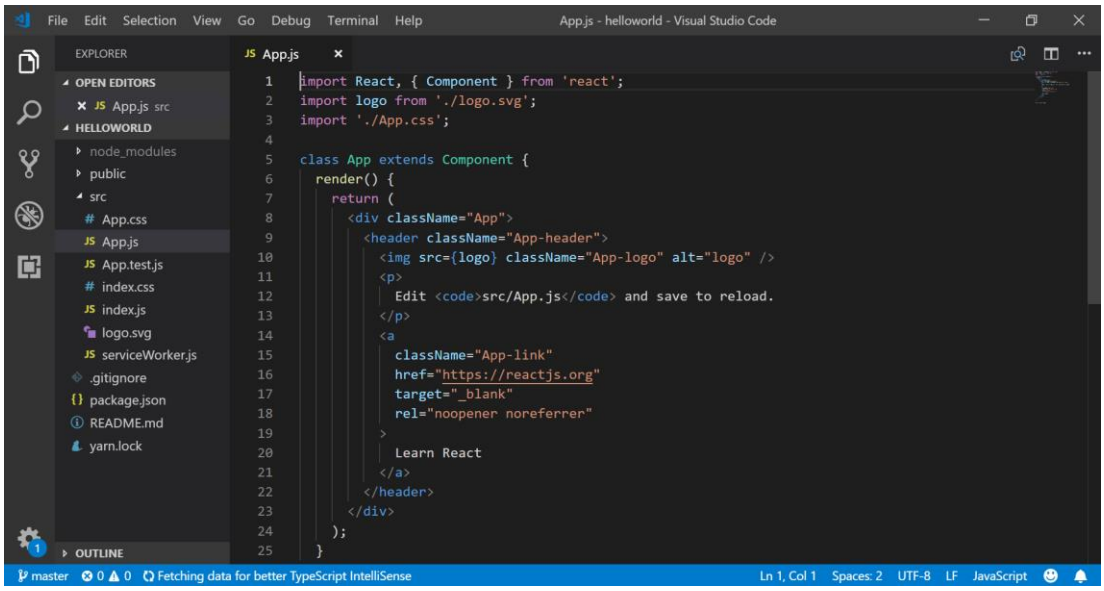
Työssä käytettiin kehitysympäristönä Microsoftin avoimeen lähdekoodiin perustuvaa tekstieditoria, Visual Studio Codea. Ominaisuuksiltaan editori on kevyt ja tehokas ohjelmisto, joka on saatavilla yleisimpiin käyttöjärjestelmiin. Visual Studio Code tukee kehitystoimintoja, kuten virheenkorjausta, versionhallintaa ja laajennuskelpoisuutta, mutta ei tue monimutkaisempia työprosesseja, kuten kattavammat ohjelmistoympäristöt tekevät. (GitHub vscode n.d.)

Ilman laajennuksia editorissa on valmiina sisäänrakennettuna tuki JavaScriptille, TypeScriptille, Node.js:lle, mutta laajennusten vuoksi, on mahdollista ohjelmoida myös muillakin kielillä, kuten esimerkiksi PHP:llä, Pythonilla, C++:lla, sekä C#:lla. (Microsoft Visual Studio Code 2019)

Visual Studio Code toimii MIT-lisenssin alla, mikä tarkoittaa sen olevan ilmainen sillä ehdolla, että alkuperäinen tekijänoikeusilmoitus säilyy jokaisessa ohjelmassa ja ohjelman kopiossa. (GitHub vscode license 2015)

Visual Studio Code on suosittu tekstieditori etenkin web-ohjelmoinnissa, ja esimerkiksi useissa web-pohjaisen ohjelmoinnin tutoriaaleissa kannustetaan Visual Studio Coden käyttöön. Esimerkiksi Stephen Grider, David Joseph Katz sekä Jonas Schmedtmann käyttävät opetusalusta Udemyn opetusmateriaaleissaan Visual Studio Codea. (Grider n.d.; Katz n.d.; Schmedtmann n.d.)

Alla esimerkki (kuva 2) Visual Studio Coden käyttöliittymästä tummalla teemavärillä.



Kuva 2. Visual Studio Coden käyttöliittymä

3.2 ReactJS

Käyttöliittymän ollessa web-pohjainen, ohjelmointikielenä toimi JavaScript. Tässä työssä ei kuitenkaan käytetty puhdasta JavaScriptiä, vaan sovelluskehystä. Sovelluskehyksenä käytettiin Reactia, joka on Facebookin kehittämä suosittu JavaScript-kirjasto käyttöliittymien rakentamiseen.

React on itseään selittävä, helppolukuinen sovelluskehys, joka tekee interaktiivisten, eli vuorovaikutteisten, käyttöliittymien luomisesta helpompaa.

React on toiminnaltaan komponenttipohjainen. Useimmiten App-nimiseksi nimitetty komponentti on ohjelman pääkomponentti. App-komponentti sisältää lapsikomponentteja, ja lapsikomponentit voivat sisältää omia lapsikomponentteja. Komponentit voivat vastaanottaa dataa, ja antaa sitä edelleen eteenpäin omille lapsikomponenteilleen. Tätä komponentin lapsilleen antamaa dataa kutsutaan propseiksi. Komponentit toimivat eräänlaisina ohjelman rakennuspalikoina.

Reactin tehokas toiminta perustuu siihen, että datan muuttuessa, ei kaikkia komponentteja päivitetä, vaan vain ainoastaan ne komponentit, joille se on tarpeellista.

Reactissa komponentteja on kahdenlaisia, luokallisia (kuva 3) sekä funktionaalisia (kuva 4) komponentteja. (React n.d.; GitHub react n.d.; React tutorial n.d.)

```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}
```

Kuva 3. Esimerkki luokallisesta komponentista. (React tutorial n.d.)

```
function Square(props) {
  return (
    <button className="square" onClick={props.onClick}>
      {props.value}
    </button>
  );
}
```

Kuva 4. Esimerkki funktionaalisesta komponentista. (React tutorial n.d.)

Aikaisemmin ainoastaan luokallisille komponenteille oli mahdollista määritellä komponentin tila, eli state (kuva 5), mutta Reactin syksyllä 2018 julkaisemien ”Hooksien” myötä myös funktionaalisille komponenteille pystyi asettamaan tilan (kuva 6). (React Introducing Hooks n.d.)

```
class Square extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: null,
    };
  }

  render() {
    return (
      <button className="square" onClick={() => alert('click')}>
        {this.props.value}
      </button>
    );
  }
}
```

Kuva 5. Esimerkki luokallisesta komponentista, jolle on asetettu tila. (React n.d.)

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Kuva 6. Luokallinen komponentti, jolle on asetettu tila käyttämällä useState-hooksia. (React Introducing Hooks n.d.)

Myös React toimii MIT-lisenssin alla, eli on ilmainen sillä ehdolla, että alkuperäinen tekijänoikeusilmoitus säilyy jokaisessa ohjelmassa ja ohjelman kopiossa. (GitHub react licence 2018)

3.2.1 JSX

JSX on huomattavasti HTML-kieltä huomattavasti muistuttava JavaScriptin syntaksi. Sen käyttö yhdessä React-kirjaston kanssa ei ole pakollista, mutta sitä suositellaan vahvasti. Syntaksi tekee Reactilla tehdystä ohjelmasta huomattavasti helppolukuisemman ja itseään selittävän. Alla esimerkit JSX:llä kirjoitetusta koodista (kuva 7) sekä HTML:llä kirjoitetusta koodista (kuva 8) (React tutorial n.d.)

```
class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.toWhat}</div>;
  }
}

ReactDOM.render(
  <Hello toWhat="World" />,
  document.getElementById('root')
);
```

Kuva 7. Esimerkki JSX-syntaksista (React tutorial n.d.)

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1, shrink-to-fit=no"
    />
    <meta name="theme-color" content="#000000" />
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <title>React App</title>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

Kuva 8. Esimerkki HTML-syntaksista

3.3 Redux

Käyttöliittymän tilanhallintaan käytettiin Reduxia, JavaScriptin tilanhallintakirjastoa. Redux on Reactin kanssa yhteensopiva, mutta sitä ei ole suunniteltu erityisesti Reactin

kanssa käytettäväksi, vaan se sopii käytettäväksi myös muiden JavaScript-sovelluskehysten, sekä puhtaan JavaScriptin kanssa.

Reactia voidaan käyttää myös ilman erillistä tilanhallintakirjastoa, mutta monimutkaisissa ohjelmissa tilanhallinnan toteuttaminen Reduxilla tekee ohjelmoinnista helpompaa ja nopeampaa, sillä se mahdollistaa koko ohjelmalle yhden yhteisen tilavaraston, storen, joka on jokaisen komponentin käytettävissä ja muokattavissa.

Ilman Reduxia, Reactissa jokaisella komponentilla on joko oma tila, tai ei ole tilaa ollenkaan. Komponentin tila on helposti käytettävissä ainoastaan komponentilla itsellään, tai siirrettävissä komponentin omille lapsikomponenteille käyttöön. Tila on mahdollista siirtää komponentin isäntäkomponentille, mutta se on huomattavasti monimutkaisemmin toteutettavissa. (Grider Introduction to Redux n.d.)

Reduxin store on olio, joka tuo ohjelman käyttämät actioncreator-funktiot ja reducer-funktiot yhteen, ja pitää sisällään koko ohjelmiston tilaa. Storen pitämää tilaa voidaan muokata ainoastaan kutsumalla actioncreator-funktiota (kuva 9), mikä palauttaa action-olion.

```
props.setTimer(true);
```

Kuva 9. Esimerkki actioncreator-funktion kutsumisesta

Actioncreator-funktiossa täytyy olla vähintäänkin määriteltynä actionin tyyppi (type) ja halutessa myös määriteltynä kuorma (payload) (kuva 10).

```
export const setTimer = (timerOn) => ({
    type: SET_TIMER,
    payload: timerOn
});
```

Kuva 10. Esimerkki actioncreator-funktiosta

Reduxin reducer-funktiot määrittävät, miten ohjelmiston tila muuttuu lähetettyjen action-olioiden mukaisesti. Alla olevassa esimerkissä (kuva 11) timerReducer-reducer

saa actioncreatorin palauttaman action-olion, ja muuttaa tilan, eli staten, actionin tyypin mukaan. (Redux 2018)

```
export function timerReducer(state = '', action) {

    switch(action.type) {
        case SET_TIMER:{
            return {
                ...state,
                timerOn: action.payload,
            }
        }
        default: {
            return state;
        }
    }
}
```

Kuva 11. Esimerkki timerReducer-reducerista

3.4 Redux-Saga

Käyttäessä Redux- ja React-kirjastoja yksinään, voidaan käyttää ainoastaan synkronisia actioncreatoreita.

Synkroninen actioncreator voi palauttaa ainoastaan heti saatavilla olevaa dataa, esimerkiksi ohjelman sisään kovakoodattua, tai käyttäjän antamaa dataa. Tämä johtuu siitä, että datakyselyä tehdessä, ei actioncreator odota vastausta, vaan aloittaa suorittamaan ohjelman seuraavaa vaihetta.

Asynkroninen actioncreator taas palauttaa dataa, joka ei ole välittömästi saatavilla, vaan vaatii saapumiseen ennalta tietämättömän ajan. Tällaista dataa ovat ohjelman ulkopuolelle tehtyjen datakyselyjen tulokset. (Grider Middlewares in Redux n.d.)

Redux-Saga on kirjasto, joka mahdollistaa muun muassa asykronisen datan haun. Koska käyttöliittymän esittämä data haetaan API:n kautta, on tämänkaltainen kirjasto ohjelman toiminnan kannalta välttämätön.

Redux-Saga käyttää toiminnassaan JavaScriptin generaattorifunktioita. Generaattorifunktiot eroavat JavaScriptin tavallisista funktioista siten, että kutsuessa generaattorifunktiota, koko funktion sisältöä ei ajeta, vaan se sisältää yhden tai useamman pysähdyspaikan, jotka merkitään avainsanalla yield. Ensimmäistä kertaa funktiota kutsuessa koodi ajetaan ensimmäiseen pysäytyspaikkaan. Toista kertaa kutsuessa toiseen, ja niin edelleen.

Generaattorifunktioiden hyöty saavutetaan sillä, että koodin jokainen rivi on varmasti suoritettu loppuun asti, ennen siirtymistä seuraavalle riville. (Phillips Javascript generator functions n.d.)

Funktiota on kutsuttava niin, että varmasti jokainen sen sisällä oleva yield-rivi ajetaan, kunnes suoritettavaa koodia ei ole enää jäljellä. Tämä varmistetaan käyttämällä generaattorifunktioita watchersagaa ja workersagaa. (Phillips first saga – watchers, workers, and ”takeEvery” n.d.)

Asynkroniseen datanhakuun olisi voinut käyttää vaihtoehtoisena kirjastona myös middleware-kirjastoa, esimerkiksi Redux-Thunkia. Redux-Sagan generaattorifunktiot, eli sagat, ovat kuitenkin tehokkaampia, hallittavampia sekä helpommin testattavissa. (Redux-Saga n.d.; Grider Middlewares in Redux n.d.)

3.5 Git

Cimcorp-konsernin tuotekehitysosastolla versionhallinta toteutetaan käyttämällä Git-versionhallintajärjestelmää. Git on ilmainen, avoimeen lähdekoodiin perustuva järjestelmä, joka mahdollistaa ohjelmointiprojektien hajauttamisen ja haarautumisen useaan halutessa toisistaan riippumattomaan haaraan (branch) (kuva 12).

Luotuja haaroja voidaan kehittää, kopioida, yhdistää ja poistaa.

Kuva 12. Esimerkki haarautetusta projektista. (Git n.d.)

Haaran paikalliseen kopioon tehdyt muutokset eivät automaattisesti päivity alkuperäiseen haaraan, vaan se täytyy sitouttaa (commit) siihen. Haaroja voidaan yhdistää (merge) toisiinsa, ja tarvittaessa myös poistaa kokonaan. Virheiden sattuessa, haara voidaan myös palauttaa takaisin aikaisempaan toimivaan versioon.

Projektin haarautuminen mahdollistaa sen, että useat kehittäjät voivat kehittää samaa projektia. Muita Git-versionhallintajärjestelmän hyötyjä ovat nopea suorituskyky ja helppokäyttöisyys. Edellä kuvatut ominaisuudet tukevat luotettavaa ohjelmistokehitystä, ja tekevät Gitistä miellyttävän työkalun. (Git n.d.)

Tässä ohjelmointiprojektissa haaroja oli vain yksi, sillä kehittäjiä oli vain yksi.

3.6 Muut

Ennen ohjelmiston käyttämän varsinaisen API:n valmistumista, käytettiin POST- Ja GET-kyselyjen rakenteiden testaamiseen JSONPlaceholderia. JSONPlaceholder on tekaistu API, joka mahdollistaa oman valedatan käytön. (JSONPlaceholder n.d.)

Sovelluksen komponenttien tyylittelyyn käytettiin Sassia. Sass on css-laajennuskieli, joka tekee tyylitiedostojen ylläpitämisestä selkeämpää ja helpommin hallittavaa. (Sass n.d.)

API:lle lähetettyjen kyselyiden toteuttamiseen käytettiin axiosta. Axios on lupaukseen perustuva HTTP-asiakas selaimelle ja node.js:lle. (GitHub axios n.d.)

Sovelluksen kielenkääntö toteutettiin käyttämällä i18n -käännösmoduulia. (npm i18n n.d.)

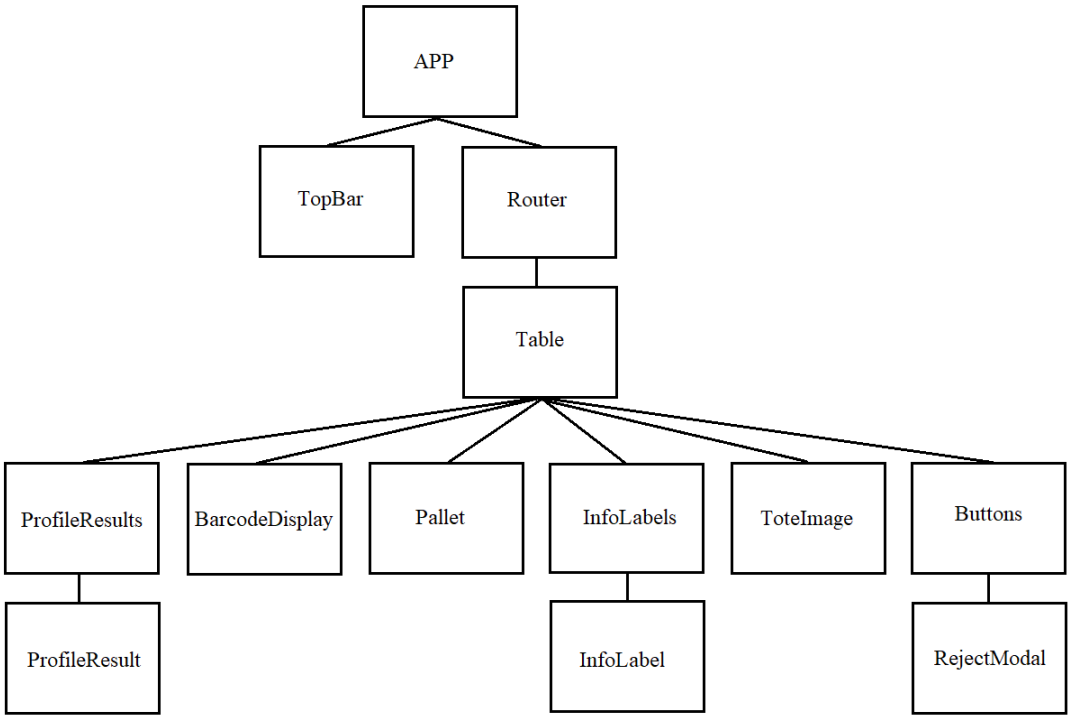
4 SOVELLUKSEN RAKENNE JA TOIMINTA

4.1 Sovelluksen rakenne

Kuten jo edellisessä luvussa mainittiin, React on toiminnaltaan komponenttipohjainen sovelluskehys, jonka ylimmän tason komponentti nimetään usein nimellä App. Yleisesti hyväksyttyä nimeämiskäytäntöä käytettiin myös tässä sovelluksessa.

Kuten alla olevassa komponenttipuussa (kuva 13) on ilmaistu, on App-komponentilla kaksi lapsikomponenttia. TopBar-komponentti on jo aikaisempaan sovellykseen kehitetty, eikä sitä, eikä sen lapsikomponentteja esitellä tässä opinnäytetyössä.

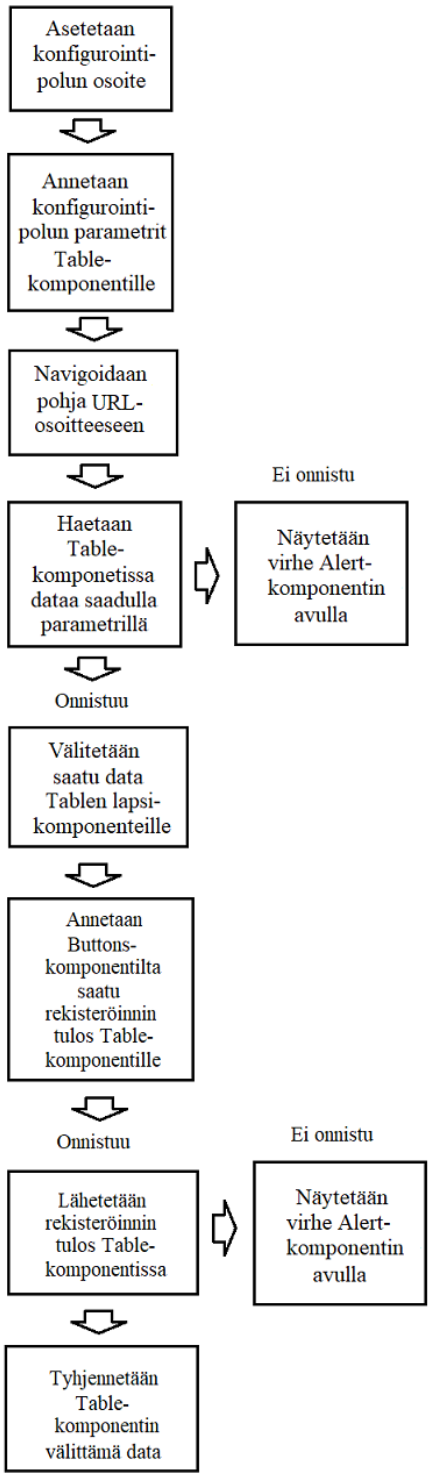
ProfileResults-komponentilla on neljä ProfileResult-lapsikomponenttia, ja InfoLabels-komponentilla kuusi InfoLabel-lapsikomponenttia. Selkeyden vuoksi, on kaavioon kuitenkin piirretty kummallekin komponentille vain yhdet lapsikomponentit.



Kuva 13. Kaavio sovelluksen komponenttipuusta

4.2 Komponenttien toiminta

Alla olevassa kuvassa (kuva 14), on esitettynä sovelluksen toiminta yksinkertaistettuna.



Kuva 14. Kaavio komponenttien toiminnasta

4.2.1 Router

Reach Router on ohjelman sisällä navigoinnin mahdollistava riippuvuuskirjasto, joka sisältää muun muassa tässäkin sovelluksessa käytetyn Router-komponentin. Router-komponentti mahdollistaa lapsikomponenteilleen URL-polun asettamisen. Lapsikomponenteille annettu polku (path) lisätään sovelluksen käyttämän pohja URL-osoitteen loppuun.

Routerin sisältämät lapsikomponentit näkyvät käyttäjälle vain niiden määriteltyä URL-polkua käyttämällä.

Tässä sovelluksessa Table-komponentille on määritelty kaksi polkua. Toinen polku käyttää ohjelman pohja URL-osoitetta, ja toinen polku on niin kutsuttu konfigurointi-polku, joka sisältää kaksi URL-parametria. URL-parametrit ovat käyttäjän ohjelmalle antamia parametreja, jotka annetaan navigoidessa WWW-osoitteeseen. Esimerkiksi osoitteessa www.base.com/123/up annetut URL-parametrit ovat 123 ja up. Näillä parametreilla asetetaan näytön positio ja lavan kulkusuunta.

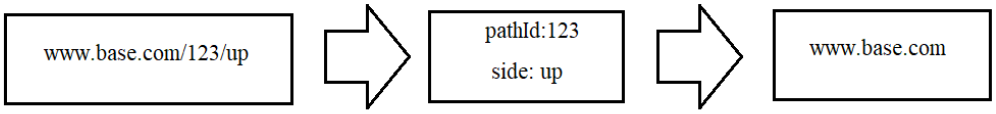
Komponenttiin määritellyssä polussa olevat parametrit tunnistetaan niiden edessä olevista kaksoispisteistä (kuva 15).

```
<Router>
<Table path=":pathId/:side"/>
<Table path="/"/>
</Router>
```

Kuva 15. Table-komponentille annettu polku

Kaksi annettua polkua tarkoittavat, että Table-komponentti on näkyvissä kahdessa eri osoitteissa. Kuitenkin, jos navigoidaan ensin pohja URL-sivulle, ei Table-komponentilla ole vielä mitään näytettävää, sillä sovellus ei ole saanut tarvitsemiaan parametreja. PathId-parametria käytetään myöhemmin datan hakemiseen, ja jos sitä ei ole määritelty, ei ohjelma osaa hakea dataa oikeasta paikasta. Jos dataa ei ole, ei ole mitään, mitä näyttää.

Alla olevassa kaaviossa (kuva 16) kuvataan sovelluksen reititystä. Kaavion ensimmäisessä osiossa on konfiguraatiopolku parametreineen. Toisessa osiossa kuvataan URL-osoitteesta saatuja parametreja. Kolmannessa osiossa on pohja URL-polku, jonne siirrytään automaattisesti, kun tarvittavat parametrit on saatu.



Kuva 16. Sovelluksen reititys

Ensin on siis navigoitava konfigurointipolkuun, ja annettava vaaditut parametrit. Router antaa saadut parametrit propseina Table-komponentille. Kuten jo aikaisemmassa luvussa mainittiin, propseiksi kutsutaan dataa, jota komponentti lähettää lapsikomponenteilleen.

Kun Table-komponentti on saanut parametrit, kutsutaan setPosition-actioncreatoria, ja asetetaan nämä parametrit sen argumenteiksi. SetPosition luo actionin, jonka seurauksena TableReducer-reducer palauttaa tilan, jossa on asetettuna sovelluksen käyttämä positio sekä lavan kulkusuunta. Tämän jälkeen sovellus navigoi automaattisesti takaisin pohja URL-osoitteeseen (kuva 17).

```
useEffect(() => {
    if (props.pathId && props.side)
        props.setPosition(props.side, props.pathId);
        navigate('/');
}, [props.pathId, props.side])
```

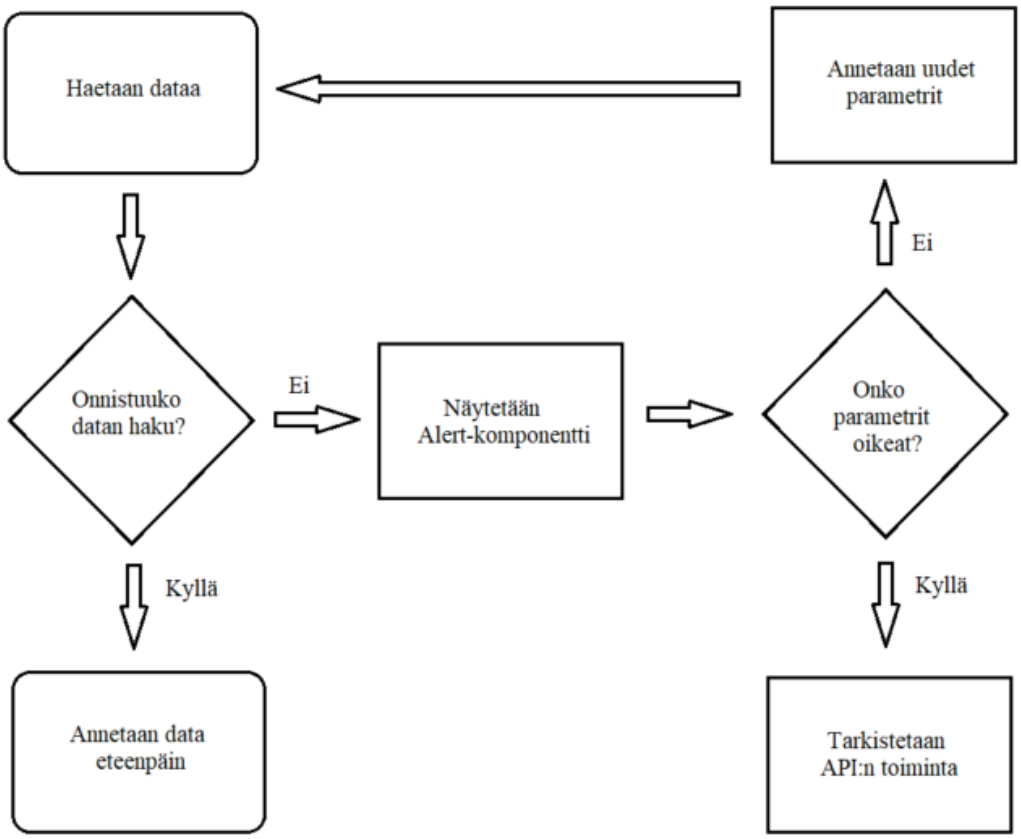
Kuva 17. Navigointi pohja URL-osoitteeseen

Sovelluksessa päädyttiin käyttämään kahta polkua, sillä näin loppukäyttäjä, eli operaattori, näkee sovelluksen käyttämänä osoitteena ainoastaan pohja URL-osoitteen. Tällä pyritään ehkäisemään se, ettei konfiguroituja parametrejä muuteta vahingossa virheellisiksi.

Koska sovellus on tyypiltään SPA (Single Page Application), ei muunlaista navigointia ohjelman sisällä tarvita. (Reach Router n.d.; Reach Router URL Params n.d.)

4.2.2 Table

Alla olevassa vuokaaviossa (kuva 18) kuvataan Table-komponentin sisällä tapahtuvaa datan hakua.



Kuva 18. Vuokaavio datan hausta

Table-komponentissa haetaan dataa TableReducerin palauttaman tilan position avulla. Table-komponentti kutsuu getData-actioncreatoria, ja asettaa saamansa position sen argumentiksi.

Koska data haetaan sellaiselta API:lta, jonka sisältö muuttuu usein, toistetaan getData-actioncreatorin kutsumista joka kolmas sekunti, eli dataa niin kutsutusti pollataan (kuva 19).

```
function getData() {
  props.getData(props.pos);
}

useEffect(() => {

  savedGetData.current = getData;
});

useEffect(() => {

  function tick() {
    savedGetData.current();
  }
    let id = setInterval(tick, 3000);
    return () => clearInterval(id);

}, [props.pos]);
```

Kuva 19. Datan pollaaminen Table-komponentissa

Kun data haetaan sovelluksen ulkopuolelta, on kyse asynkronisesta datan hausta. Kuten aikaisemmassa luvussa kerrottiin, tässä sovelluksessa asynkroniseen datan haun hallintaan käytettiin Redux-Saga -kirjastoa.

WatchGetData-saga ottaa jokaisen lähetetyn actionin, jonka tyyppi on GET_DATA, ja kutsuu workersaga getDataa (kuva 20). GetData-saga kutsuu getData-funktiota, ja antaa kuormana saamansa position sen argumentiksi. GetData-funktio yrittää tehdä GET-kyselyn GetData-sagalta saamallaan positiolla.

```
export function* getData(action) {
    try {
    const result = yield call(api.getData, {pos:action.payload});
    yield put(getDataSuccess({result : result.data}));
    }catch(e){
        yield put(usersError({
            error: I18n.t('table.errors.getData')
        }))
    }
}


export function* watchGetData() {
    yield takeEvery(GET_DATA, getData)
}
```

Kuva 20. watchGetData -watchersaga ja getData workersaga

Jos kysely onnistuu, kutsutaan getDataSuccess-actioncreatoria, ja asetetaan saatu tulos sen argumentiksi. Jos kysely ei onnistu, kutsutaan usersError-actioncreatoria, ja asetetaan sen argumentiksi virheviesti.

Käytetystä actioncreatorista riippuen, TableReducer-reducer palauttaa tilan, jossa joko tyhjätään tilan error-viesti ja asetetaan datahaun tuottama tulos tilan resultille, tai asetetaan kuormana saatu error-viesti tilan errorille (kuva 21). Virhetilanteessa ei siis aseteta tilan resultille arvoa lainkaan.

```
case GET_DATA_SUCCESS: {
    return {
        ...state,
        error: '',
        result: action.payload.result,
    }
}
case USERS_ERROR:{
   return {
       ...state,
       error: action.payload.error
   }
}
```

Kuva 21. TableReducer-reducerin palauttamat tilat

Jos datan haku on onnistunut, Table-komponentti saa API:n kautta haetun datan TableReducerin palauttamasta resultista, ja antaa sen eteenpäin kuudelle näkyvälle lapsikomponenteilleen propseina.

Näkyvien lapsikomponenttien lisäksi, on Table-komponentilla myös yksi oletuksena piilotettu Alert-komponentti. Tämä komponentti on näkyvissä ainoastaan virhetilanteissa, ja sen tehtävänä on ohjata käyttöliittymän käyttäjää, kertomalla missä virhe on tapahtunut.

Toinen vaihtoehto datan hakemiseen olisi ollut WebSocketin käyttäminen. WebSocket on ohjelmointirajapinta, joka mahdollistaa käyttäjän selaimen ja palvelimen reaaliaikaisen vuorovaikutuksen, luomalla jatkuvan yhteyden niiden välille. (MDN web docs the WebSocket API 2019)

Tässä ohjelmassa ei kuitenkaan koettu reaaliaikaista vuorovaikutusta tarpeelliseksi, sillä käyttäjän selaimelta on tarpeellista lähettää API:n kautta dataa vasta, kun käyttäjä on rekisteröinyt lavan. Datan ajastettu hakeminen koettiin järkevämmäksi tässä tapauksessa.

4.2.3 ProfileResults

ProfileResults-komponentti saa Table-komponentilta propseina tietoa lavan profiilitarkistuksesta, ja jakaa ne neljälle ProfileResult-lapsikomponentilleen.

Kukin ProfileResult -komponentti näyttää profiilitarkistuksen kohteen ikonin, sekä sen otsikon. Jos joku profiilitarkistuksen kohteista ei ole käytössä, näytetään sen ikoni ja otsikko tyhjänä.

Jos rekisteröintiasemalle jostain syystä saapuu lava, jonka profiilitarkistuskohde on ollut käytössä, mutta mittaus on epäonnistunut, näytetään tarkistusta kuvaava ikoni punaisena. Tällaista tilannetta ei kuitenkaan pitäisi pystyä tapahtumaan, sillä profiilitarkistuksen epäonnistuessa lavan pitäisi siirtyä hylkäyskuljettimelle, jo ennen rekisteröintiasemalle saapumista.

4.2.4 BarcodeDisplay

BarcodeDisplay-komponentti näyttää Table-komponentilta saamansa lavan viivakoodin tekstikentän sisällä.

4.2.5 Pallet

Pallet-komponentin tehtävänä on piirtää rekisteröitävä lava pinoineen ylhäältäpäin kuvattuna. Komponentti saa propseina tiedon lavan kulkusuunnasta, lavalla olevien pinojen koordinaatit suhteessa lavan keskipisteeseen, lavalla olevien muovilaatikoiden tyypin, sekä tiedot lavan ja pinojen orientaatioista.

Lavan orientaatiolla tarkoitetaan lavan mahdollisia pyörähdyksiä viivakoodin luvun jälkeen, ja pinojen orientaatiolla sitä, miten päin pinot ovat suhteessa lavaan. Vaikka lava olisi kuljettimella ollessaan pyörähtänyt 180 astetta matkalla rekisteröintiasemalle, osaa komponentti piirtää pinot oikeaan järjestykseen, oikeisiin paikkoihin ja oikein päin.

Komponentti laskee kunkin pinon sijainnin, ja piirtää ne lavalle annetun muovilaatikon koon mukaan.

Piirron toteuttamiseen suunniteltiin aluksi käytettäväksi JavaScriptin canvas-kirjastoa, Konvaa. Konva ei kuitenkaan toiminut responsiivisesti, joten piirto päädyttiin toteuttamaan JSX-syntaksilla, käyttämällä div-lohkoja.

Jos lavan profiilitestissä pinojen ulkosivujen tarkistamisessa on löytynyt virhe, ja lava jostain syystä on saapunutkin rekisteröintiasemalle, ilmaistaan lavan virheellinen sivu punaisella viivalla.

Pallet-komponentin ulkonäkö muuttui hieman alkuperäisestä suunnitelmasta. Alkuperäisen suunnitelman mukaan, sekä piirrettävä lava että sen pinot olisivat neliön muotoisia. Näytössä kuitenkin haluttiin tuoda lavan oikeat mitat esiin, sen operaattorille tuoman selkeyden vuoksi.
Myös lavan kulkusuunta muuttui. Alkuperäisen suunnitelman mukaan, lavan kulkusuunta näytöllä olisi joko oikealta vasemmalle, tai vasemmalta oikealle. Lopulliseen näyttöön vaihtoehtoisiksi kulkusuunniksi tuli kuitenkin ylhäältä alaspäin, tai alhaalta ylöspäin.

4.2.6 InfoLabels

InfoLabels-komponentti saa Table-komponentilta propseina tietoa lavan tuotetiedoista ja sisällöstä. ProfileResults-komponentin tavoin, jakaa komponentti saamansa propsit omille lapsikomponenteilleen.

InfoLabels-komponentilla on kuusi InfoLabel-lapsikomponenttia, joista kukin näyttää saamansa tiedon otsikon ja arvon.

4.2.7 ToteImage

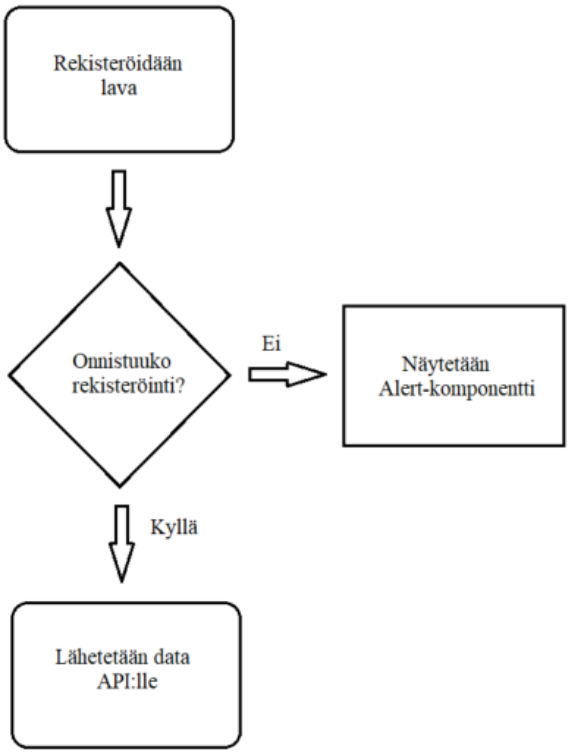
ToteImage-komponentin tehtävänä on näyttää rekisteröitävällä lavalla olevien muovilaatikoiden kuva. Komponentti saa propseina tiedon muovilaatikoiden tyypistä, ja käy läpi sovellukseen talletetut tyypit. Jos tyyppi löytyy, palauttaa komponentti kuvan oikeasta muovilaatikosta, ja jos tyyppiä ei löydy, palautetaan muovilaatikon kuvanpaikka tyhjänä.

4.2.8 Buttons

Buttons-komponentti koostuu kahdesta painikkeesta, Acceptista ja Rejectista, sekä rejectModal-lapsikomponentista. RejectModal-komponentti on normaalitilassa suljettuna, ja aukeaa ainoastaan, jos painetaan Reject-painiketta.

RejectModal-komponentissa valitaan valikosta hylkäysperuste, ja vahvistetaan hylkääminen.

Kun lava on rekisteröity, lähettää Buttons-komponentti rekisteröinnin tuloksen isäntäkomponentilleen, Tablelle. Rekisteröinnin tulos koostuu tuloksesta ja hylkäysviestistä. Jos lava on hyväksytty, on hylkäysviesti tyhjä. Vuokaaviossa (kuva 22) on rekisteröinti kuvattuna yksinkertaistetusti.



Kuva 22. Vuokaavio rekisteröinnistä

Table-komponentti kutsuu confirmPallet-actioncreatoria, ja asettaa sen argumentiksi lavan ID:n, Buttons-komponentilta saamansa rekisteröinnin tuloksen, lavan viivakoodin ja rekisteröintiaseman position (kuva 23).

```
const handleConfirm = ({ accepted, msg }) => {
    props.confirmPallet({id, accepted, msg, bc, pos});
}
```

Kuva 23. ConfirmPallet-actioncreatorin kutsuminen.

WatchConfirmPallet-saga ottaa viimeisen lähetetyn actionin, jonka tyyppi on CONFIRM_PALLET, ja kutsuu workersaga confirmPalletia. ConfirmPallet-saga kutsuu registrationResult-funktiota, ja antaa sille argumentiksi actionin kuorman. RegistrationResult-funktio yrittää tehdä API:lle POST-kyselyn confirmPallet-sagalta saamillaan arvoilla.

Jos kysely onnistuu, kutsutaan jälleen getDataSuccess-actioncreatoria, ja asetetaan sen argumentiksi tyhjä result. Tällöin TableReducer-reducer päivittää tilan errorin ja resultin molemmat tyhjiksi. Jos kysely ei onnistu, kutsutaan usersError-actioncreatoria, ja asetetaan sen argumentiksi virheviesti. Kyselyn epäonnistuessa, TableReducer ei päivitä tilan resultia tyhjäksi, vaan säilyttää tiedot (kuva 24).

```
export function* confirmPallet(action) {
    try {
       yield call(api.registrationResult, {
           id : action.payload.id,
           accepted : action.payload.accepted,
           msg : action.payload.msg,
           bc : action.payload.bc,
           pos : action.payload.pos }
           );
        const result = '';
        yield put(getDataSuccess({result}));
    }catch(e){
        yield put(usersError({
            error: I18n.t('table.errors.confirm')
        }))
    }
}


export function* watchConfirmPallet() {
    yield takeLatest(CONFIRM_PALLET, confirmPallet)
}
```

Kuva 24. watchConfirmPallet -watchersaga ja confirmPallet workersaga

5 OHJELMISTOTESTAUS

Jos ohjelmistoprojekti on luotu käyttämällä Create React App -generaattoria, käyttää se automaattisesti Jest-testauskehystä testiajoonsa. Create React App on tuettu tapa luoda yksisivuisia React-sovelluksia, eikä sen käyttäjän tarvitse murehtia sellaisten työkalujen asentamisesta ja määrittämisestä, kuten Webpack tai Babel, vaan voi suoraan keskittyä olennaiseen – ohjelmiston kehittämiseen. Myös tämä ohjelmisto on luotu käyttämällä Create React Appia.

Testitiedostot tulee luoda joko ”__tests__ ”- nimiseen kansioon, tai yksittäiset tiedostot tulee nimetä ”.test.js”- tai ”.spec.js” -päätteillä. Esimerkiksi App.test.js on validi nimi testitiedostolle. Testitiedostojen oikeanlainen nimeäminen on tärkeää, sillä Jest etsii ja ajaa testattavia tiedostoja niiden nimen mukaan.

Varsinainen testien ajaminen tapahtuu syöttämällä komentokehotteeseen joko ”npm run test” - tai ”npm test” -komento, jolloin Jestin testienkatselutila käynnistyy. Jestin ollessa katselutilassa, ajetaan testit aina uudestaan, jos joku ohjelmiston tiedostoista tallennetaan. (Create React App Testing 2019)

5.1 Komponenttien testaus

Sovellusta testatessa, todettiin tärkeämmäksi testata sen toiminnallinen puoli, kuin kirjoittaa sen jokaiselle komponenteille useita yksikkötestejä. Tämän vuoksi komponentteja päädyttiin testaamaan käyttämällä ainoastaan snapshot-testausta.

Snapshot-testauksessa tarkistetaan, että komponentti toimii oikein. Kun testi ajetaan, tallennetaan samalla snapshot-tiedosto. Joka kerta, kun testit ajetaan uudestaan, verrataan saatua tulosta aikaisempaan tiedostoon. Jos tiedostot eivät vastaa toisiaan, testi ei ole hyväksytty, vaikka komponentti toimisikin moitteettomasti. Tällöin voidaan aikaisempi snapshot-tiedosto päivittää uuteen, tai selvittää, mikä komponentissa on muuttunut, ja korjata se.

Snapshot-testaaminen on erittäin hyödyllistä silloin, kun halutaan varmistaa, ettei käyttöliittymä muutu odottamatta. Alla olevassa kuvassa (kuva 25) on esimerkki Table-komponentin snapshot-testistä. (Snapshot Testing with Jest n.d.)

```
it('Table renders correctly', () => {
  expect(<Table/>).toMatchSnapshot();
});
```

Kuva 25. Table-komponentin snapshot-testi

5.2 Actioncreatorien testaus

Actioncreatoreita testatessa testattiin, että sen palauttamalla action-oliolla on oikea odotettu tyyppi ja mahdollinen kuorma.

Testissä kutsutaan actioncreator-funktiota, ja sen palauttama action-olio asetetaan testissä käytetyn action-muuttujan arvoksi. Sitten testataan, että action-olion tyyppi on oikea, ja että sen kuorma on asetettu oikein. Alla olevassa kuvassa (kuva 26) testataan getDataSuccess-actioncreatoria.

```
describe('getDataSuccess', () => {
    const action = getDataSuccess('Payload');

    it('has the correct type and payload', () => {
        expect(action.type).toEqual('GET_DATA_SUCCESS');
        expect(action.payload).toEqual('Payload');
    });
});
```

Kuva 26. getDataSuccess-actioncreatorin testi

5.3 Reducerin testaus

TableReducer- reducerin toiminnassa testattiin, että se palauttaa tilan odotetun laisesti, saamansa actionin tyypin mukaan. Testissä käytetyn action-muuttujan arvoksi asete-

taan olio, jolla on tyyppi ja mahdollinen kuorma. Sitten testataan, osaako reducer käsitellä saamaansa action-olion oikein, ja palauttaa tilan odotetun laisesti. Alla olevassa kuvassa (kuva 27) on esimerkki Tablereducerin testistä, jossa sille annetaan GET_DATA_SUCCESS -tyyppinen action.

```
it('handles actions of type GET_DATA_SUCCESS', () => {
    const action = {
        type: GET_DATA_SUCCESS,
        payload: {
            result:['test']
        }
    };

    const newState = TableReducer([], action);
    expect(newState).toEqual({error:'', result : ['test']});

});
```

Kuva 27. TableReducer-reducerin testi, jossa testataan, käsitteleekö se oikein getDataSuccess-actioncreatorin palauttamaa actionia.

TableReduceria testatessa testattiin myös, miten reducer toimii, jos se saa action-olion, jolla on sille tuntematon tyyppi (kuva 28). Tällaisessa tilanteessa, ei reducerin pitäisi tehdä tilaan muutoksia.

```
it('handles action with unknown type', () => {
    const newState = TableReducer([], {type: 'Something'});
    expect(newState).toEqual([]);
});
```

Kuva 28. TableReducer-reducerin testi, jossa testataan, käsitteleekö se oikein tuntemattomia actioneitä.

5.4 Sagojen testaus

Sovelluksessa käyttettyjen sagojen testaukseen käytettiin Redux Saga Test Plan -testauspakettia. Paketti tukee sekä yksikkötestausta, että integraatiotestausta. Yksikkötestauksen yksi haittapuoli on, että yksinkertainen sagan uudelleen järjestäminen saattaa

rikkoa testin, sillä testi on suoraan yhdistettynä testattavaan sagaan. Tämän vuoksi ohjelmassa päädyttiin käyttämään integraatiotestausta.

Integraatiotestauksessa käytetään testauspaketin expectSaga-funktiota, jolle annetaan argumenttina testattava saga ja testattavan sagan mahdolliset omat argumentit. Funktion palauttama arvo on mahdollista ketjuttaa Redux-Sagan sisältämillä effects-funktioilla, kuten esimerkiksi call- ja put-funktioilla. (Redux Saga Test Plan n.d.)

Alla olevassa testissä (kuva 29) testataan getData-sagan toiminta tilanteessa, jossa virhettä ei tapahdu. Sagan lähettämä API-kysely on mallinnettu käyttämällä provider-funktiota, jonka argumenteiksi on annettu matcher-funktio ja testiä varten luotun result-muuttuja. Result-muuttujan tarkoituksena on jäljitellä oikean kyselyn tuottamaa tulosta.

Testissä testataan, kutsuuko saga actioncreator-funktiota, jonka tyypiksi on määritelty GET_DATA_SUCCESS, ja asettaako se saamansa tuloksen actioncreatorin argumentiksi. Testin lopussa oleva run-funktio käynnistää testin.

```
it('getData', () => {
  const result = {data: {result : {}}}
  return expectSaga(getData, 'GET_DATA')
    .provide([
      [matchers.call.fn(api.getData), result]
    ])
    .put({
      type: 'GET_DATA_SUCCESS',
      payload: {
        result : result.data
      },
    })
    .run();
});
```

Kuva 29. GetData-sagan toiminnan testi

Sovelluksen käyttämien sagojen toiminta testattiin myös mahdollisten virhetilanteiden osalta.

Alla olevassa testissä (kuva 30) testataan getData-sagan toiminta virhetilanteessa. Provider-funktiota käytetään mallintamaan tilanne, jossa sagaa suorittaessa tapahtuu virhe. Virhetilanteissa suoritetaan sagan catch-lohkon sisältö.

Testissä testataan, kutsuuko saga actioncreator-funktiota, jonka tyypiksi on määritelty USERS_ERROR, ja asetetaanko virheviesti actioncreatorin argumentiksi.

```
it('getData handles errors', () => {
    return expectSaga(getData)
      .provide({
        call() {
          throw new Error('Not Found');
        },
      })
      .put.actionType('USERS_ERROR')
      .run();
  });
```

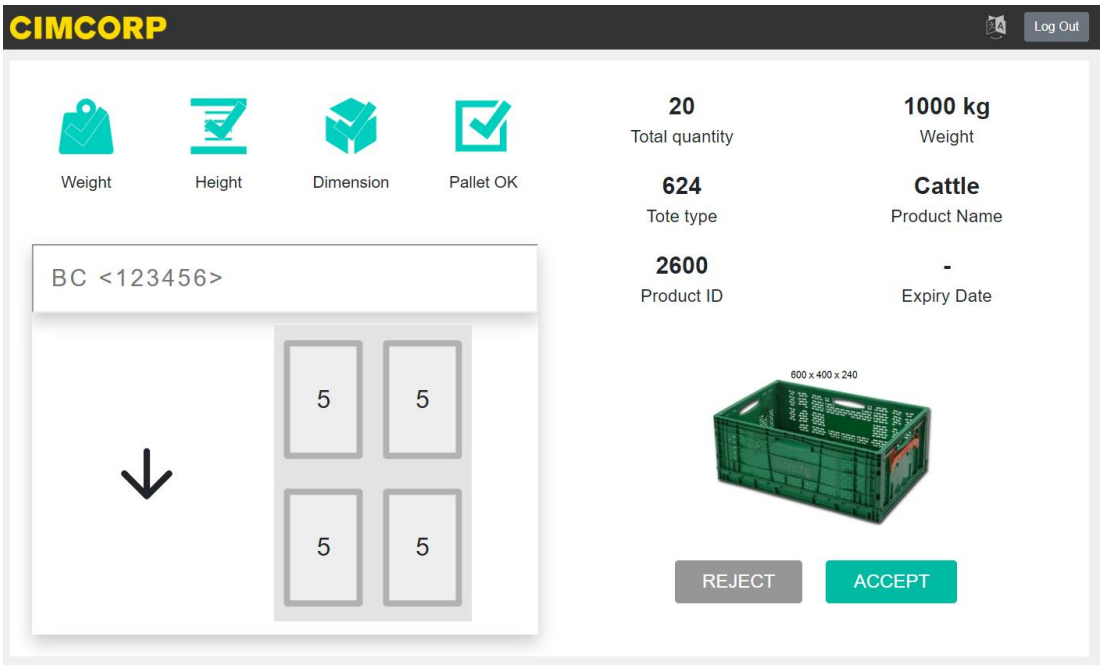
Kuva 30. GetData-sagan toiminnan testi virhetilanteessa

6 YHTEENVETO

6.1 Lopputulos

Alla olevassa kuvassa (kuva 31) on esitettynä käyttöliittymäsovelluksen lopullinen ulkoasu. Tulos vastaa hyvin pitkälti työn lähtökohdissa esiteltyä suunnitelmaa, lukuun ottamatta Pallet-komponentille tehtyjä muutoksia.

Responsiivisuutta pidettiin sovelluksen mukautuvuuden kannalta tärkeänä asiana, ja myös se saatiin toteutettua odotetun laisesti. Ainoastaan älypuhelimien pienissä näytöissä, ei sovelluksen ulkoasu vastaa suunnitelmaa, sillä sitä ei ole suunniteltu käytettäväksi missään tilanteessa niin pienillä laitteilla.



Kuva 31. Kuva valmiista käyttöliittymästä

Sovelluksessa jäi kehitettäväksi vielä viivakoodinluvun mahdollistava ominaisuus. Sen valmistelu on vielä toistaiseksi kesken, mutta sen odotetaan olevan käyttökunnossa kevään aikana.

Käyttöliittymäsovellusta tullaan testaamaan myös käytännössä tulevan kesän aikana, Cimcorp-konsernin automaatiohallissa.

6.2 Pohdinta

Jo opiskelupaikan vastaanottaessani tiesin, että haluaisin ehdottomasti tehdä opinnäytetyöni yritykselle. Halusin tehdä jotain, jolla olisi tarkoitus, ja joka pääsisi tuotantokäyttöön. Koin, että näin näkisi omien käsiensä jäljen paremmin, ja syventäisin omaa osaamistani.

Olen pitänyt Cimcorp-konsernia erittäin mielenkiintoisena yhtiönä sitä saakka, kun kuulin siitä ensimmäisen kerran. Kuulosti aivan uskomattomalta, että on suuryritys, jonka pääkonttori kuitenkin sijaitsee pienessä satakuntalaisessa kaupungissa. Konsernin toimittamien pitkälle automatisoitujen tuotteiden soveltuvuus niin monelle erilaiselle alalle, lisäsi mielenkiintoani yritystä kohtaan entisestään. Tuotteita toimitetaan niin elintarvike- ja juomateollisuuteen, rengasteollisuuteen, postinkäsittelyyn sekä monille muille teollisuuden aloille.

Tämän vuoksi olinkin erittäin innoissani, että pääsin tekemään juuri Heille opinnäytetyöni.

Pidin opinnäytetyön aihevalinnasta paljon, sillä olen pitänyt web-ohjelmistokehitystä aina hyvin mielenkiintoisena. Minulla oli siitä vain hieman kokemusta, ja halusin oppia siitä lisää. Kaiken kaikkiaan pidin opinnäytetyön tekemistä Cimcorpilla hyvin miellyttävänä, sillä tuotekehityksessä käytettävät tekniikat olivat ajankohtaisia, eivätkä niin sanottuja vanhentuneita toimintamalleja. Osastolla seurattiin ja otettiin käyttöön uusia päivittyneitä tekniikoita ajan tasalla.

Koko opinnäytetyöprosessin aikana opin paljon uutta, niin web-ohjelmistokehityksestä kuin työelämässä yleisesti käytettävistä toimintatavoista ja työkaluista. Uskon kaikella oppimallani olevan varmasti arvoa tulevaisuuttani ajatellen.

LÄHTEET

Cimcorp. 2019. Viitattu 21.4.2019.
https://www.cimcorp.com/fi/cimcorp/cimcorp-konserni

Cimcorp intranet. 2018. Viitattu 21.4.2019.
Saatavissa ainoastaan yrityksen työntekijöille.

Create React App Testing. 2019. Viitattu 28.4.2019.
https://facebook.github.io/create-react-app/docs/running-tests

Git. n.d. Viitattu 27.4.2019.
https://git-scm.com/about

GitHub axios. n.d. Viitattu 28.4.2019.
https://github.com/axios/axios

GitHub react. n.d. Viitattu 24.4.2019.
https://github.com/facebook/react

GitHub react licence. 2018. Viitattu 24.4.2019.
https://github.com/facebook/react/blob/master/LICENSE

GitHub vscode. n.d. Viitattu 22.4.2019.
https://github.com/Microsoft/vscode

GitHub vscode license. 2015 Viitattu 22.4.2019.
https://github.com/Microsoft/vscode/blob/master/LICENSE.txt

Grider, S. n.d. Introduction to Redux. Viitattu 24.4.2019.
https://www.udemy.com/react-redux/learn/v4/t/lecture/12531392?start=0

Grider, S. n.d. Middlewares in Redux. Viitattu 24.4.2019.
https://www.udemy.com/react-redux/learn/v4/t/lecture/12586864?start=0

Grider, S. n.d. Modern React with Redux. Viitattu 22.4.2019.
https://www.udemy.com/react-redux/learn/v4/overview

JSONPlaceholder. n.d. Viitattu 28.9.2019.
https://jsonplaceholder.typicode.com/

Katz, D. n.d. React – Mastering Test Driven Development. Viitattu 22.4.2019.
https://www.udemy.com/react-tdd/learn/v4/overview

Latva, M. 2019. Product Manager, Cimcorp. Ulvila. Henkilökohtainen tiedonanto 9.1.2019.

MDN web docs the WebSocket API. 2019. Viitattu 27.4.2019.
https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

Microsoft Visual Studio Code. 2019. Viitattu 22.4.2019.
https://code.visualstudio.com/docs

npm i18n. n.d. Viitattu 28.4.2019.
https://www.npmjs.com/package/i18n

Phillips, T. n.d. first saga – watchers, workers, and ”takeEvery”. Viitattu 26.4.2019.
https://www.udemy.com/redux-saga/learn/v4/t/lecture/12335844?start=0

Phillips, T. n.d. Javascript generator functions. Viitattu 26.4.2019.
https://www.udemy.com/redux-saga/learn/v4/t/lecture/12335844?start=0

Rajakangas, T & Jantunen, J. 2019. Functional Specification Multipick System. Ulvila: Cimcorp.

Reach Router. n.d. Viitattu 27.4.2019.
https://reach.tech/router

Reach Router URL Params. n.d. Viitattu 27.4.2019.
https://reach.tech/router/example/url-params

React. n.d. Viitattu 24.4.2019.
https://reactjs.org/

React Introducing Hooks. n.d. Viitattu 24.4.2019.
https://reactjs.org/docs/hooks-intro.html

React tutorial. n.d. Viitattu 24.4.2019.
https://reactjs.org/

Redux 2018. Viitattu 24.4.2019.
https://redux.js.org/

Redux Saga. n.d. Viitattu 26.4.2019.
https://redux-saga.js.org/

Redux Saga Test Plan. n.d. Viitattu 28.4.2019.
http://redux-saga-test-plan.jeremyfairbank.com/

Sass. n.d. Viitattu 28.9.2019.
https://sass-lang.com/

Schmedtmann, J. n.d. Advanced CSS and Sass. Viitattu 22.4.2019.
https://www.udemy.com/advanced-css-and-sass/learn/v4/overview

Snapshot Testing with Jest. n.d. Viitattu 28.4.2019.
https://jestjs.io/docs/en/snapshot-testing.html#snapshot-testing-with-jest

Talvitie, S. 2019. Suunnitelma käyttöliittymän ulkoasusta. Ulvila: Cimcorp.