Karjalainen Jani

Analysaattoreiden QT-käyttöliittymien uudistaminen web-teknologioita hyödyntäen

Insinööri (AMK)
Tieto- ja viestintätekniikka
Kevät 2024

KAMK • University of Applied Sciences

Tiivistelmä

Tekijä(t): Karjalainen Jani

Työn nimi: Analysaattoreiden QT-käyttöliittymien uudistaminen web-teknologioita hyödyntäen

Tutkintonimike: Insinööri (AMK), tieto- ja viestintätekniikka

Asiasanat: web-ohjelmointi, React, JavaScript, käyttöliittymä

Opinnäytetyön aiheena oli selvittää, kuinka analysaattoreiden nykyiset QT-pohjaiset käyttöliittymät voidaan korvata käyttämällä nykyaikaisia web-sovelluskehitysteknologioita. Opinnäytetyön toimeksiantajana toimi Valmet Automation Oy. Työn tavoitteena oli tutkia nykyaikaisia web-sovellusteknologioita ja toteuttaa demosovellus, jonka pohjalta voitaisiin lähteä kehittämään ja uudistamaan analysaattoreiden vanhoja käyttöliittymiä.

Työssä toteutettiin nykyaikainen web-sovellus käyttämällä JavaScript-ohjelmointikieltä. React-kirjaston avulla työhön tehtiin käyttöliittymä ja sovelluksen palvelinpuoli toteutettiin käyttämällä Node.js-ympäristöä. Käyttöliittymän ja palvelimen välinen kommunikaatio toteutettiin websocket-protokollaa hyödyntäen ja palvelimen ja vanhan sovellusrajapinnan välinen kommunikaatioprotokolla oli MQTT.

Kehitetty sovellus vastasi tavoitteita ja sovellukseen saatiin vanhasta rajapinnasta laitetietoja, arvoja sekä vanhan käyttöliittymän käyttöön määriteltyjä tekstejä. Sovelluksen palvelin kommunikoi nykyisen rajapinnan kanssa ja palvelimelta saatiin käyttöliittymäsovelluksen käyttöön tietoja ja tekstejä. Käyttöliittymään luotiin muutamia analysaattorisovelluksessa käytettyjä komponentteja, ja käyttöliittymästä valintaruutuja tai painikkeita painaessa tieto välittyi palvelimen kautta vanhaan rajapintaan ja sieltä edelleen analysaattoreiden käyttöön. Käyttöliittymään luotiin graafi, jonka avulla voidaan piirtää käyttöliittymään käyrä analysaattoreiden mittaustuloksista.

Työssä toteutetun web-sovelluksen toiminnasta voitiin todeta, että nykyiset käyttöliittymät on mahdollista korvata käyttäen nykyaikaisia web-teknologioita. Toteutettu käyttöliittymäsovellus ja palvelin antavat hyvät mahdollisuudet nykyisten käyttöliittymien korvaamiseen uusilla web-teknologioilla. Työn lopussa on pohdittu sovelluksen mahdollisia jatkokehitystoimenpiteitä.

KAMK • University of Applied Sciences

Abstract

Author(s): Karjalainen Jani

Title of the Publication: Renewing Analyzers' QT-based User Interface Utilizing Web Technologies

Degree Title: Bachelor of Engineering, Information and Communication Technologies

Keywords: web development, React, JavaScript, user interface

The subject of this thesis was to research how the current QT-based user interfaces can be replaced with modern web technologies. This thesis was commissioned by Valmet Automation Oy. The purpose of this thesis was to research modern web application technologies and to create demo software that can be used as a base template to develop and create new user interfaces for the current and new analyzers.

The work done in this thesis consists of developing a modern web application using JavaScript programming language. The user interface for the application was created with React library, and the server side was implemented using Node.js. The communication protocol between the user interface and the server was WebSocket and the communication protocol between the server and the analyzer application protocol interface was MQTT.

The developed application successfully met the goals of this thesis, as it received device information, values, and texts from the current application protocol interface. The new server communicates with the existing application protocol interface and updates data into the new user interface. Some components used in the old application were replicated to the new user interface, and when checkboxes or buttons were pressed in the new application, the server transmitted the new values to the existing application protocol interface where they could be used by the analyzers. Additionally, a graph was created for the new application to draw a curve that can be used to display measurement results.

From the new application created for this thesis, it is evident that it is possible to replace the current user interfaces using modern web technologies. The new user interface and server provide a good template for further research and development into replacing current user interfaces with modern web technologies. At the end of this thesis, there are some further development ideas for this application.

Sisällys

Symboliluettelo

npm	JavaScript-ohjelmointikielen paketinhallintatyökalu.
JSON	JavaScript Object Notation on tietomuoto, jota käytetään muodostamaan objekteja JavaScript-kielellä
Topic	Topicit ovat avaimia, joiden avulla viestejä lähetetään tilaajille MQTT-protokollassa
Broker	MQTT-broker on palvelin MQTT-viestien lähetykseen sovellusten välillä
React	JavaScript-kirjasto käyttöliittymien tekemiseen
Vue.js	JavaScript-kirjasto käyttöliittymien tekemiseen
Back-end	Sovelluksen palvelin, joka suorittaa toimintoja ja rajapintakutsuja
Front-end	Sovelluksen käyttöliittymä
Websocket	Protokolla, joka mahdollistaa kaksisuuntaisen reaaliaikaisen tiedonsiirron
Bundle	Sovelluksen JavaScript-tiedostot ja riippuvuudet pakattuna yhdeksi tiedostoksi

1 Johdanto

Tämän opinnäytetyön aiheena oli tutkia, kuinka nykyiset QT-pohjaiset käyttöliittymät analysaattoreissa voidaan toteuttaa nykyaikaisilla web-sovelluskehitysmenetelmillä. Työ keskittyy sovelluksen palvelinpuolen ja käyttöliittymän toteutukseen. Tässä työssä ei käsitellä uusien toimintojen toteuttamista nykyiseen rajapintaan, jotta se soveltuu web-sovelluksien käytettäväksi.

Työn tavoitteena oli löytää keino, jolla nykyisestä analysaattoreiden rajapinnasta saadaan laitetietoja, arvoja sekä tekstejä käytettäväksi uuteen selainpohjaiseen käyttöliittymään. Työssä toteutettiin demosovellus, johon vietiin analysaattoreista tietoja ja sovelluksen käyttöliittymästä voidaan ohjata analysaattorin toimintoja. Opinnäytetyön toimeksiantaja oli Valmet Automation Oy, Kajaanin toimipiste.

Valmet on kansainvälinen yritys, joka toimittaa ja kehittää palveluita sellu-, paperi- ja energiateollisuuteen. Valmetilla työskentelee yli 19 000 henkilöä ja yrityksellä on yli 220 vuoden historia. Yrityksen liikevaihto oli vuonna 2023 noin 5,5 miljardia. Pääkonttori sijaitsee Espoossa ja toimipisteitä on ympäri maailma. Liiketoiminta on jaettu viiteen linjaan, jotka ovat palvelut, virtauksensäätö, automaatiojärjestelmät, sellu ja energia sekä paperi. [1.]

Automaatiojärjestelmät keskittyvät automaatioratkaisujen toimittamiseen. Niitä voivat olla yksittäisiin mittauksiin liittyvät toteutukset tai koko tehtaan prosessiautomaatiojärjestelmät. Analysaattorit ovat osa automaatiojärjestelmiä ja niiden avulla suoritetaan erilaisia mittauksia sekä käsitellään mittaustuloksia. [1.]

2 Web-sovellus

Web-sovellus on sovellus, joka sijaitsee palvelimella ja sitä hallitaan selaimesta internetin välityksellä. Kun web-sovellus sijaitsee palvelimella, sitä ei tarvitse asentaa laitteelle ja sitä voidaan käyttää useilta eri laitteilta. Web-sovellus eroaa normaalista web-sivusta sen dynaamisuudella. Normaali web-sivusto on kiinteä ja siellä ei ole muuttuvia komponentteja. Web-sovelluksia voivat olla mm. nettipankit, sähköpostisovellukset ja erilaiset projektinhallintatyökalut. [2.]

Web-sovellus vaatii toimiakseen web-palvelimen, joka käsittelee käyttäjältä tulevat komennot. Kun käyttäjä tekee sovelluksessa pyynnön, se lähetetään selaimesta internetin välityksellä palvelimelle. Palvelin käsittelee käyttäjältä tulevan pyynnön ja suorittaa vaadittavat toiminnot. Toimintoihin voi kuulua esimerkiksi tiedonhaku tietokannasta. Sen jälkeen palvelin käsittelee prosessit ja luo vaadittavan paketin lähetettäväksi takaisin selaimelle. Selain päivittää käyttäjän näkymän vastaamaan pyyntöä. [3.]

2.1 Web-sovelluskehitys

Web-sovelluskehityksessä pyritään luomaan sovellus, joka toimii palvelimella ja sen käyttö tapahtuu web-selaimella. Web-sovelluskehitys voidaan jakaa useisiin osiin, joista tärkeimpiä ovat suunnittelu, back-end- ja front-end-kehitys sekä sovelluksen julkaisu ja ylläpito. [4.]

Web-sovellusta suunnitellessa määritellään sovelluksen vaatimukset ja sovelluksen toteuttamiseen käytettävät teknologiat. Front-end-kehityksessä toteutetaan sovelluksen käyttöliittymä ja käyttöliittymäsuunnittelu. Back-endin puolella toteutetaan sovelluksen palvelin ja sen suorittamat toiminnot, kuten tietokannasta tietojen haku ja sen vieminen käyttöliittymään. Back-endin puolella toteutetaan ja hallitaan myös käyttäjien hallinta. [4.]

2.2 HyperText Markup Language (HTML)

HTML on merkintäkieli, jota käytetään nettisivustojen tai sovelluksien luomiseen. HTML-tiedosto määrittelee, kuinka selain näyttää sivun tai sovelluksen käyttäjälle. HTML muodostuu elementeistä, joiden avulla käyttäjälle näkyvä sisältö voidaan jaotella osiin. Elementtejä ovat esimerkiksi sivun otsikko tai painike, jota painamalla sovellus suorittaa sille määritetyn toiminnon. [5.]

Yksistään HTML:n avulla ei saada kovinkaan näyttäviä ja toiminnallisia sovelluksia aikaan, vaan sen mukaan yhdistetään CSS, jolla saadaan muotoiltua elementit ja JavaScript hoitamaan toiminnallisuus. [5.]

HTML:n juuret menevät pitkälle ja ensimmäinen versio julkaistiin vuonna 1992. Kehitys jatkui vauhdikkaasti 1990-luvulla ja neljäs versio julkaistiin jo vuonna 1997. Tästä seurasi pidempi tauko ja HTML:n viides versio julkaistiin 2014. HTML5 sisälsi lukuisia elementtejä, joiden avulla HTML-tiedoston lukeminen helpottui. Näitä uusia elementtejä ovat esimerkiksi nav, jota käytetään luomaan navigointipalkki tai section, jota voidaan käyttää määrittelemään sivusto osioihin. [5.]

2.3 Cascading Style Sheets (CSS)

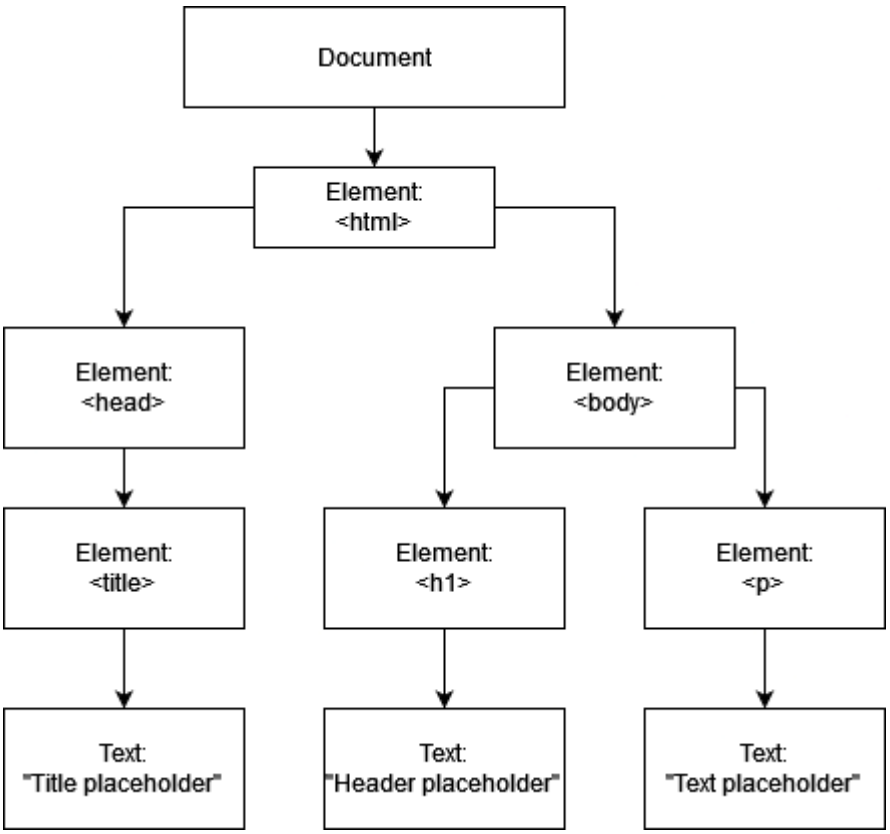
CSS on kieli, jota käytetään HTML-tiedostojen muotoiluun. CSS:n avulla saadaan HTML-sivuista näyttäviä ja sen avulla muutetaan tekstien kokoja, fontteja ja koko sivuston tai sovelluksen ulkoasua. CSS:ää voidaan käyttää HTML-tiedoston sisällä muotoilemaan yksittäinen lause tai otsikko, mutta yleisesti sen avulla luodaan web-sovelluksista yhtenäisiä näkymien välillä käyttämällä erillistä CSS-tiedostoa. Erilliseen tiedostoon luotua CSS-muotoilua voidaan käyttää jokaiseen HTML-näkymään ja tekemällä sovelluksen ulkoasuun muutoksia, ne päivittyvät automaattisesti jokaiseen HTML-elementtiin. [6.]

2.4 Document Object Model (DOM)

DOM eli Document Object Model on web-sovelluksissa käytettävä rajapinta. DOM on yleisin rajapinta web-sovelluksissa ja nettisivuissa ja se mahdollistaa koodin suorittamisen selaimessa. Se esittää nettisivun tai sovelluksen objekteina ja kaikki sovelluksen käyttämät metodit ja arvot ovat sijoitettuna objekteihin. Objekteja ja niiden sisältöä voidaan sovelluksessa käsitellä käyttämällä JavaScriptiä. DOM on luotu käyttäen useita rajapintoja, joilla saadaan tuki esimerkiksi HTML-dokumenteille. [7.]

DOM on osa web-rajapintaa, jota käytetään sivujen ja web-sovelluksien tekemiseen, ja se suunniteltiin riippumattomaksi tietyistä ohjelmointikielistä, vaikka suurin osa käyttääkin rajapintaa JavaScriptin avulla. Document Object Modelin käyttämiseen ei tarvita omia riippuvuuksia, vaan siihen pääsee käsiksi esimerkiksi suoraan HTML-tiedostosta JavaScriptiä käyttämällä. Yksi esimerkki

sen käytöstä on lisätä nappiin toimintaa käyttämällä addEventListener()-metodia. [7.] Kuvassa 1 DOM kuvattuna kaaviona.



Kuva 1. Document Object Model kaaviona.

2.5 Single page application (SPA)

SPA on nykyaikainen tapa luoda web-sovellus. SPA on käytössä melkein jokaisella nykyaikaisella nettisivulla ja se mahdollistaa sovelluksen näkymän muuttamisen ilman perinteistä tapaa, jossa koko sivu täytyy ladata uudelleen jonkin tiedon muuttuessa. SPA-sovelluksia ovat yleiset ja joka-päiväiset sivustot, kuten nettipankit, Google Maps ja Netflix. Single page application -sovellukset mahdollistavat dynaamisen ja sulavan käyttäjäkokemuksen, kun jokaista sovelluksen elementtiä voidaan dynaamisesti päivittää. [8.]

SPA-sovelluksessa on vain yksi päänäkymä, jossa suuri osa komponenteista pysyy samanlaisena ja vain tietty osa sovellusta päivittyy. Esimerkiksi sivuston yläosassa usein sijaitseva navigointi-palkki pysyy samanlaisena, mutta sen alla oleva näkymä vaihtuu, kun navigoidaan eri osiin sivus-

tolla. Se päivittää vain sen elementin tietoja ja näkymää, joita käyttäjän valitsema toiminto päivittää. Tällä saadaan sovelluksesta myös nopeampi käyttää, kun suurta osaa sivustoa ei tarvitse ladata uudelleen jatkuvasti ja päivitetään esimerkiksi pelkkä tekstinäkymä. SPA hyödyntää myös virtuaalista Document Object Modelia, jota käydään React-osiossa alempana tarkemmin. [8.]

SPA-sovellusta kehitettäessä myös sovelluksen kehittämistä voidaan nopeuttaa jakamalla sovelluksen osat komponentteihin. Eri kehittäjät voivat kehittää tiettyjä osia sovelluksesta ja lopulta ne voidaan liittää yhteen näkymään. Sovellusta käytettäessä myös palvelimelle tulevat pyynnöt vähenevät ja palvelimen ei tarvitse aina näyttää kokonaista uutta sivua käyttäjälle, koska vain yksi osa sovellusta tarvitsee ladata uudestaan. [8.]

3 React

React on JavaScript-kirjasto, jolla voidaan luoda web-sovelluksiin käyttöliittymä komponentteja tai kokonaisia käyttöliittymiä. Komponentit voivat siis olla yksittäinen nappi HTML-sivulla, tai komponentin avulla voidaan luoda koko HTML-sivu. React-komponentit pystyvät päivittymään ja näyttämään dataa, ilman että koko HTML-sivu täytyy ladata uudestaan. Tämän takia Reactin avulla voidaan luoda sulavia ja toimivia käyttöliittymän osia tai kokonaisia käyttöliittymiä web-sovellukseen. [9, s. 6–7.]

React-koukut mahdollistavat erilaisten ominaisuuksien käytön komponenteissa. Reactissa on valmiita koukkuja, joita yhdistelemällä päästään haluttuun lopputulokseen. Yleisin koukku React-sovelluksessa on tilakoukku. Reactissa jokaisen komponentin tila määritellään useState- tai useReducer-koukkujen avulla. UseState-koukun avulla määritellään tilamuuttuja, jota voidaan vaihtaa koukkua käyttämällä. [10.]

Toinen Reactissa usein käytetty valmis koukku on useEffect. UseEffect-koukkua käytetään esimerkiksi muodostamaan websocket-yhteys komponenttia renderöitäessä ja katkaisemaan yhteys, kun komponenttia ei enää näytetä sivulla. [10.]

Kuvassa 2 määritellään counter-tilamuuttuja, joka alustetaan arvoon 0. Sen jälkeen luodaan 2 nappia, joiden avulla tilaa voidaan muuttaa yhdellä ylöspäin tai alaspäin, ja nappien alla on teksti, joka näyttää laskurin tilan numeroina.

```
import './App.css'
import { useState } from 'react';

function App() {

  const [counter, setCounter] = useState(0);

  return (
    <>
      <button onClick={() => {setCounter(counter + 1)}}>Increase counter</button>
      <button onClick={() => {setCounter(counter - 1)}}>Decrease counter</button>
      <p>Counter {counter}</p>
    </>
  )
}

export default App
```

Kuva 2. UseState-koukun käyttö yksinkertaisen laskurin toteuttamisessa.

3.1 Virtual Document Object Model

VDOM on virtuaalinen kopio sovelluksen sen hetkisestä DOM:ista, joka on muistissa Reactin DOM-kirjaston avulla. React käyttää VDOM:ia päivittämään sovelluksen komponentteja sulavasti ja tehokkaasti. Kun komponenttien ominaisuudet tai komponentin tila muuttuu, React päivittää VDOM:in ja vertailee sitä vanhan DOM:in kanssa ja määrittelee vain ne muutokset, jotka eroavat vanhasta. Vasta sen jälkeen React päivittää muutokset DOM:iin. Tämä johtaa sovelluksen päivittymisen nopeutumiseen, kun oikeaa DOM:ia ei tarvitse muuttaa useasti. [11.]

React käyttää tehokasta algoritmiä virtuaalisten ja todellisen DOM:in vertailemiseen. Sen avulla määritetään, mitkä osat todellisesta DOM:ista tarvitsee päivittää. Jos komponentti ei ole muuttunut, React voi käyttää aiemmin luotua virtuaalista DOM:ia ja sen ei tarvitse luoda täysin uutta. [11.]

3.2 JavaScript XML

JSX on syntaksi JavaScriptiin, joka mahdollistaa HTML-tyylisen koodin kirjoittamisen JavaScript-tiedoston sisällä. JSX on yleisin käytössä oleva tapa luoda React-komponentteja. Normaalissa web-kehityksessä HTML-, CSS- ja JavaScript-tiedostot ovat omina tiedostoinaan, mutta web-sovelluskehityksen kehittyessä ja web-sovellusten mennessä dynaamisempaan suuntaan React-komponentit luodaan yhteen tiedostoon, johon sisältyy HTML-elementit ja JavaScriptin toiminnallisuus. React käyttää JSX-syntaksia yhdistäen JavaScriptin ja HTML-merkintäkielen. [12.]

JSX-syntaksilla on muutamia sääntöjä, joita tulee noudattaa. Ensimmäinen on se, että palautetaan vain yksi elementti. Jos elementissä on useita komponentteja, ne täytyy yhdistää pääkomponenttiin esimerkiksi käyttäen div-elementtiä, joiden sisällä on useita elementtejä. Toinen sääntö on se, että jokainen elementti täytyy sulkea. Kolmas sääntö liittyy attribuuttien nimeämiseen, jossa tulisi käyttää camelCase-tyyliä. [12.]

3.3 Components

Komponentti React-sovelluksessa on yksi osa sovelluksen käyttöliittymästä. React-sovelluksessa käyttöliittymä on jaettu useisiin komponentteihin, joista käyttöliittymä rakennetaan. Jokainen

komponentti sijaitsee omassa tiedostossaan ja niillä on omat funktiot ja toiminnallisuudet. Komponentteihin jaettu käyttöliittymä helpottaa ja nopeuttaa sovelluksen kehitystä, koska jokaisella komponentilla voi olla oma kehittäjänsä. Jokainen komponentti on myös uudelleenkäytettävissä React-sovelluksessa. Komponenteista rakennettu sovellus myös helpottaa sen ylläpitämistä ja toimintojen lisäämistä. Samoja komponentteja voidaan käyttää myös eri React-sovellusten välillä. [13.]

Komponentit voidaan jakaa kahdenlaisiin komponentteihin, joita ovat toiminnallinen komponentti ja luokkakomponentti. Toiminnallinen komponentti on yksinkertainen JavaScript-funktio, joka määrittelee, mitä komponentti näyttää käyttöliittymässä. Toiminnallisella komponentilla ei ole minkäänlaisia tilanhallintaominaisuuksia. Luokkakomponentit kirjoitetaan käyttäen JavaScript ES6-luokkia. Nämä komponentit ovat laajempia ja niillä voi olla tiloja, joiden avulla komponentin sisällä olevia tietoja päivitetään tai muutetaan. [14.]

Toiminnallisia komponentteja käytetään, kun komponentti ei tarvitse tilanhallintaa ja toiminnallisen komponentin koodi on myös selkeämpi ja yksinkertaisempi. Toiminnallisia komponentteja käytetäänkin yksinkertaisien osien tekemiseen käyttöliittymässä, kuten otsikointiin ja yksinkertaisiin nappeihin. [14.]

Luokkakomponentteja käytetään monimutkaisempiin toteutuksiin, kun komponentti tarvitsee tilanhallintaa. Luokkakomponentit ovat hyviä esimerkiksi taulukoiden näyttämiseen ja niiden päivittämiseen dynaamisesti tilan vaihtuessa. [14.]

3.4 Redux

Redux on kirjasto sovelluksen komponenttien tilanhallintaa. Sitä voidaan kutsua keskitetyksi ”kaupaksi” sovelluksessa ja se sisältää jokaisen komponentin tilan. Tilanhallintaa tarvitaan esimerkiksi piilottamaan tai näyttämään tiettyjä osioita sovelluksessa. Reduxin käyttö tulee erityisen tärkeäksi suuria ja laajoja sovelluksia kehittäessä, kun useiden komponenttien tila voi vaihdella ohjelman eri vaiheissa. [15.]

React-sovelluksessa on vain yksi kauppa koko sovelluksen käyttöön. Kaupan tehtävänä on hallita koko sovelluksen komponenttien tiloja. Kaupassa oleviin tiloihin päästään käsiksi getState-funktiolla ja tiloja voidaan muuttaa käyttämällä dispatch-funktiota, jonka avulla määritetään tapahtuma kyseiselle tilalle. Tapahtumien avulla määritellään, mitä tilalle tehdään, kun dispatch-funktiota

kutsutaan. Tapahtumat voivat olla synkronisia, eli tapahtuma suoritetaan heti, tai tapahtumat voivat olla myös asynkronisia. Asynkronisten tapahtumien avulla suoritetaan esimerkiksi fetch-kutsuja ja näitä käytetään yleisesti esimerkiksi Ladataan-tekstin näyttämiseen, ennen kuin fetch-kutsu on hakenut kaiken datan. Reducerit käsittelevät tapahtumia ja päivittävät tiloja sekä määrittelevät, mitä erilaisten tapahtumien kuuluu tehdä. Reducerit ovat siis funktioita, jotka ottavat parametreinä tilan ja tapahtuman ja toteuttavat saadulle tilalle halutun tapahtuman. [15.]

Redux-koodin kirjoittaminen on monimutkaista ja vaatii usein hankalaa logiikkaa ja tilojen käsittelemisessä voi helposti tapahtua virheitä. Tätä helpottamaan on kehitetty nykyaikainen Redux-toolkit-kirjasto. Redux-toolkitin avulla monia tapahtumia on pelkistetty sekä se hoitaa monimutkaisen tilanhallinnan pelkistämällä kaupan, tapahtumien ja reducereiden luontia. [16.]

4 Sovelluksen toteutus

Opinnäytetyö alkoi toimeksiantajan kanssa pidetyssä palaverissa, missä käytiin läpi opinnäytetyön toteutusta. Palaverissa myös rajattiin opinnäytetyötä ja asetettiin vaatimuksia opinnäytetyön toteutumisesta. Opinnäytetyön alkaessa perehdyttiin ja tutustuttiin web-sovelluskehityksen menetelmiin ja tekniikoihin, joita työn toteuttamisessa tarvittiin. Työ aloitettiin suunnittelemalla, mitä tekniikoita käyttäen sovellusta lähdetään kehittämään ja mitä toimintoja sovellus tarvitsee. Suunnittelun jälkeen aloitettiin rakentamaan sovelluksen back-end-palvelinta ja toteuttamaan sinne tarvittavia toimintoja. Siitä jatkettiin käyttöliittymäsovelluksen toteuttamiseen.

4.1 Sovelluksen suunnittelu

Sovelluksen suunnittelu alkoi määrittelemällä, minkälaista tietoa ja dataa sovelluksen back-end-palvelimen täytyy käsitellä ja kuinka ne saadaan käyttöliittymäsovellukseen. Nykyisestä rajapinnasta uuteen käyttöliittymäsovellukseen täytyy tuoda laitetietoja, mittaustuloksia, komponenttien tekstejä useilla kielivalinnoilla sekä arvoja toimintojen ohjaukseen. Nykyisissä sovelluksissa ohjelmistojen komponenttien attribuutit ja tekstit ovat määriteltyinä XML-tiedostoihin. XML-tiedostot sisältävät mm. napeissa ja valintaruuduissa olevat tekstit ja otsikot. XML-tiedostoihin on myös määritelty tekstit eri kielille. Sovellukseen täytyy saada myös nappien, valintaruutujen, select-valikoiden ja muiden komponenttien arvot, joita muuttamalla ohjelmisto suorittaa haluttuja toimintoja. Nykyisestä rajapinnasta ei ole mahdollista suoraan hakea näitä tietoja uuteen web-pohjaiseen sovellukseen. Nykyiseen rajapintaan täytyy toteuttaa uusia toimintoja, joilla tiedot ja mittaustulokset voidaan lähettää uuden web-sovelluksen back-end-palvelimelle.

Tämä opinnäytetyö keskittyy web-sovelluksen ja sen back-end-palvelimen toteuttamiseen. Sovelluksen suunnittelun alkuvaiheissa tutkittiin mahdollisia sovelluskehityksen tekniikoita. Tässä työssä toteutetun demosovelluksen back-endin toteuttamiseen päädyttiin käyttämään Node.js-ympäristöä, joka mahdollistaa JavaScript-koodin suorittamista palvelimella. Palvelinpuolella päädyttiin käyttämään Express-viitekehystä. Yhtenä syynä back-endin toteuttamiseen näillä menetelmillä oli se, että molemmista oli hieman kokemusta ja se nopeuttaa palvelinpuolen toteuttamista. Web-sovelluksen front-endin, eli käyttöliittymäsovelluksen, toteuttamisen päävaihtoehtoina olivat React ja Vue. Näistä kahdesta päädyttiin toteuttamaan käyttöliittymäsovellus käyt-

täen React-kirjastoa. React tuntui helpommalta ja selkeämmältä aloittelevalle web-sovelluskehittäjälle. React on myös käyttäjämäärältään huomattavasti suurempi, joten tukea ja mahdollisia ratkaisuja löytyy netistä enemmän ja mahdollisissa ongelmatapauksissa valmiita ratkaisuja löytyy helpommin.

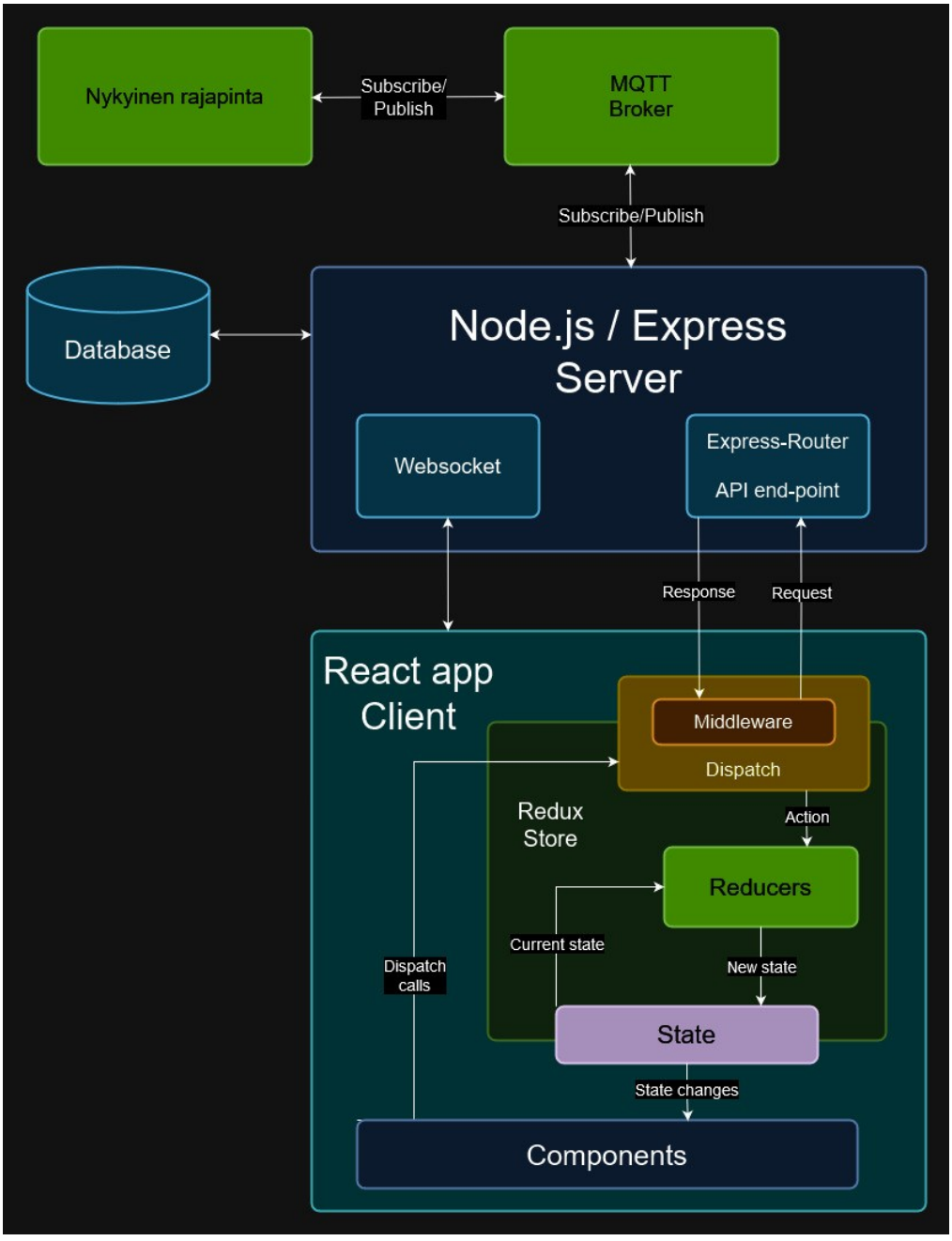
Back-end-palvelimen täytyy vastaanottaa vanhasta rajapinnasta tulevat tiedot. Palvelimen täytyy käsitellä tiedot ja tarjoilla ne käyttöliittymäsovellukselle. Palvelimen ja nykyisen rajapinnan välinen kommunikaatio päädyttiin toteuttamaan käyttämällä MQTT-teknologiaa. MQTT perustuu subscribe/publish-menetelmään, jossa palvelin muodostaa yhteyden MQTT-brokeriin, joka käsittelee sinne rajapinnasta lähetettyjä tietoja topicien avulla. Palvelimen täytyy siis tilata jokainen topic, joita rajapinnasta lähetetään. Topicit tulevat sisältämään JSON-objekteja, jotka muodostetaan vanhassa rajapinnassa. Jokaisen topicin sisältämä tieto tulee olemaan esimerkiksi napin tekstit, arvot ja mahdolliset attribuutit. Palvelimen täytyy käsitellä topicien sisältämä data ja tarjoilla se käyttöliittymäsovelluksen käyttöön. Palvelimen täytyy myös pystyä lähettämään käyttöliittymälle tärkeitä tietoja esimerkiksi mittaustuloksia nopeasti ja aina niiden muuttuessa. Tähän tiedonsiirtoon päädyttiin käyttämään websocket-yhteyttä palvelimen ja käyttöliittymäsovelluksen välillä. Se mahdollistaa nopean datasiirron palvelimelta käyttöliittymäsovellukseen.

Palvelimen täytyy myös pystyä hallitsemaan käyttäjien kirjautumistietoja ja tämä voidaan toteuttaa rakentamalla tietokanta palvelimen kaveriksi. Palvelimen täytyy myös sisältää API-päätepisteitä, joista käyttöliittymä voi hakea kaikki topicien tiedot käyttöliittymäsovelluksen käynnistyessä. Tämä API-endpoint toteutettiin tekemällä palvelimelle polkuja käyttämällä Express-router kirjastoa. Kirjaston avulla luotiin polku, johon voidaan tehdä pyyntö käyttöliittymästä. Router vastaa kutsuun lähettämällä käyttöliittymälle objektin, joka sisältää jokaisesta topicista haetut tiedot. Objektin data järjesteltiin topicien nimien mukaan ja jokaisen nimen alla on kaikki kyseisen topicin sisältämät tiedot.

Web-sovelluksen käyttöliittymän täytyy hakea tiedot palvelimelta ja näiden tietojen avulla käyttöliittymän komponentteihin saadaan nimet, otsikot ja esimerkiksi select-komponentin sisältämät vaihtoehdot valinnoille. Käyttöliittymään tulevia tärkeitä tietoja varten sovelluksen täytyy hallita ja päivittää tärkeitä tietoja jatkuvasti. Tärkeitä tietoja varten käyttöliittymäsovelluksen ja palvelimen välillä on websocket-yhteys. Käyttöliittymäsovelluksen täytyy myös hallita palvelimelta tulevaa dataa sekä niiden tiloja. Käyttöliittymän tilojen hallinta toteutettiin rakentamalla käyttöliittymäsovellukseen tilakone käyttämällä Redux-toolkit-kirjastoa. Redux-tilanhallinta mah-

dollistaa tilojen ja niiden arvojen hallintaa siten, että ne ovat käytettävissä kaikkialla sovelluksessa. Käyttöliittymän täytyy myös luoda dynaamisia komponentteja palvelimelta tulevasta datasta. Näitä komponentteja ovat esimerkiksi painonapit, checkbox- ja select-komponentit.

Käyttöliittymäsovelluksen komponentit lähettävät palvelimelle tietoja, kun esimerkiksi nappia on painettu ja palvelimen täytyy lähettää nämä muutokset nykyisen rajapinnan käyttöön, jossa toteutetaan haluttuja toimintoja. Näitä toimintoja ovat esimerkiksi mittausprosessin aloitus. Kuvassa 3 on sovellus kuvattuna tilakaaviona.



Kuva 3. Web-sovelluksen tilakaavio.

4.1.1 MQTT

MQTT on viestiprotokolla, joka on kehitetty vuonna 1993. Sitä käytetään yleisesti laitteiden väliseen kommunikaatioon. MQTT-protokolla on yleinen standardi mm IoT-laitteissa, koska se on hyvin kevyt ja se käyttää hyvin vähän resursseja. MQTT:n käyttöönotto laitteiden välillä on myös helppoa, koska MQTT-kirjastoja on monille koodikielille ja yksinkertaiseen kommunikaatioon ei tarvita kauhean suurta määrää koodia. MQTT-protokolla toimii subscribe/publish-mallilla eli se sisältää julkaisijan (publisher) ja tilaajan (subscriber). [17.]

Toiminta voidaan jakaa kolmeen pääkomponenttiin, joita ovat MQTT-client, MQTT-broker ja MQTT-yhteys. Clientillä tarkoitetaan laitetta tai sovelluksen osaa, joka julkaisee tai tilaa tietoja topiceista. Jos client lähettää viestejä, se toimii julkaisijana; ja jos client vastaanottaa viestejä, se toimii tilaajana. Client voi myös sekä julkaista että tilata tietoja. Jokaista laitetta, joka kommunikoi MQTT-protokollan avulla, voidaan kutsua MQTT-clientiksi. [17.]

Broker on eräänlainen koordinaattori, joka hallitsee viestien lähetyksen clienttien välillä. Brokerin tehtäviin kuuluu julkaistujen viestien vastaanottaminen ja clienttien hallinta. Clientit tilaavat topiceja ja brokerin tehtäviin kuuluu lähettää topiceihin tulevat viestit oikeille clienteille. Brokerin tehtävänä on myös hallita clienttien yhteyksiä. [17.]

Clientit luovat MQTT-yhteyden lähettämällä CONNECT-viestin brokerille ja se varmistaa yhteyden. Client ja broker ovat yhteyksissä TCP/IP-protokollan avulla ja clientit eivät koskaan ole suoraan yhteyksissä toisiinsa, vaan kommunikointi tapahtuu brokerin välityksellä. [17.]

MQTT-topic on eräänlainen ”avain”, jota broker käyttää viestien välitykseen clienttien välillä. Topicit on jaoteltu vähän samaan tyyliin kuin tiedosto- tai kansiohierarkia ja topicit määritelläänkin käyttämällä kenoviivaa. Opinnäytetyössä käytetyt topicit on luotu käyttämään ”product_name/.p/xx/yy”-muotoa, jolloin esimerkiksi checkboxin piilotukseen vaadittavat tiedot saadaan topicista ”.p/struct__CheckBox/HideValue” ja saman checkboxin käytöstä poistamiseen saadaan tiedot topicista ”.p/struct__Checkbox/DisableValue”. [17.]

4.1.2 Websocket-protokolla

Websocket-protokolla mahdollistaa jatkuvan, kaksisuuntaisen kommunikaation web-clientin ja web-palvelimen välityksellä käyttäen TCP-yhteyttä. Protokolla standardisoitiin vuonna 2011.

Websocketteja käytetään silloin, kun tarvitaan reaaliaikaista datansiirtoa web-sovellusten ja web-palvelimien välityksellä. Niiden käyttö on yleistä mm. chat-palveluissa ja paikannussovelluksissa. [18.]

Websockettien käyttö muodostuu kolmesta vaiheesta ja ensimmäinen vaihe on websocket-yhteyden muodostaminen. Yhteys muodostetaan http request/response -menetelmällä clientin ja palvelimen välillä. Kun yhteys on muodostettu, client ja palvelin voivat siirtää dataa kahdensuuntaisesti tekstimuodossa tai binääridatana. Ja kun yhteyttä ei enää tarvita, yhteys terminoidaan lähettämällä close-viesti. [18.]

Websockettien hyöty tulee siitä, että se on nopeampi kuin http-pyynnöt ja dataa voidaan lähettää molempiin suuntiin clientin ja palvelimen välillä. Tässä opinnäytetyössä websocketteja käytetäänkin tämän takia back-end-palvelimen ja käyttöliittymäsovelluksen tärkeiden tietojen, kuten mittaustuloksien ja arvojen, välitykseen käyttöliittymälle. [18.]

4.2 Back-end

Back-end-palvelimen rakentaminen aloitettiin luomalla uusi kansio projektille. Opinnäytetyössä käytettiin npm-paketinhallintaa, jonka avulla asennettiin demosovelluksessa käytettäviä kirjastoja. Palvelimen luominen aloitettiin siirtymällä uuteen kansioon ja luomalla projektille runko suorittamalla komento *npm init -y*. Tämä komento luo kansioon uuden npm-projektin sekä package.json-tiedoston. Tiedosto sisältää projektin tietoja, kuten nimen, version ja dependencies-määrittelyt. Lisäämällä *-y* voidaan vastata *npm init* -komennon kysymyksiin automaattisesti kyllä ja määrittelyt voidaan itse lisätä jälkeenpäin. Komento on myös mahdollista suorittaa ilman *-y* lisäystä, jolloin komentoriville tulee projektin tietoja kysymyksinä ja niihin täytyy vastata. Package.json-tiedoston dependencies-määrittelyt ovat tässä vaiheessa tyhjänä. Seuraavaksi asennettiin muutamia kirjastoja, joita palvelimen rakentamiseen tarvittiin. Ensiksi asennettiin Express-viitekehys komennolla *npm install express*. Kun käytetään npm-paketinhallintaa asentamaan kirjastoja, ne lisätään automaattisesti package.json-tiedoston määrittelyihin. Projektiin asennettiin myös mqtt- ja Socket.io-kirjastot. Projektiin lisättiin myös yksi sovelluskehitystä helpottava kirjasto nimeltä nodemon. Tämä asennettiin projektiin suorittamalla komento *npm install –save-dev nodemon*, mikä asentaa nodemonin pelkästään kehitysvaiheissa käytettäväksi. Lisäksi package.json sisällä olevaa scripts-kohtaa täytyy muokata, että nodemon saadaan käyttöön. Tiedoston scripts-kohtaan lisätään rivi, jossa määritellään *”dev”: ”nodemon index.js”*. Nyt projekti

voidaan käynnistää kehitystilassa suorittamalla komento npm run dev. Nodemon vahtii projektissa olevia tiedostoja ja niihin tallennettuja muutoksia ajon aikana ja käynnistää automaattisesti palvelimen uudestaan, kun jokin tiedoston on muuttunut ja tallennettu. Tämä nopeuttaa palvelimen sovelluskehitystä, koska kehittäjän ei tarvitse manuaalisesti käynnistää palvelinta aina uudestaan.

Projektikansioon luotiin tiedosto index.js, joka ajetaan palvelimen käynnistyessä. Projektikansioon luotiin myös tiedosto app.js, joka hallitsee palvelimen toiminnallisuuden. App-tiedoston luominen aloitettiin sisällyttämällä siihen express-viitekehys sekä muita kirjastoja sovelluksen käytettäväksi. Web-sovelluksen palvelimelle täytyi luoda polku, josta dataa haetaan käyttöliittymän käyttöön HTTP GET -metodilla. Polkuihin voidaan määritellä myös POST-metodi, jonka avulla palvelimelle voidaan lähettää tietoja selaimesta. Web-sovellus tarjoillaan selaimeen käyttämällä vakiopolkua ”http://localhost:3001”. Lisäpolkuja palvelimelle saadaan luotua Expressistä löytyvän Router-funktion avulla.

Projektiin lisättiin routes-kansio, joka sisältää polkujen muodostamiseen vaadittavat tiedostot. Routejen tekeminen aloitettiin luomalla data.js-tiedosto. Tiedostoon sisällytettiin Router-funktio, joka määriteltiin dataRouter-muuttujaan. Tiedostoon lisättiin muutamia uusia polku käyttämällä get-funktiota. Funktion sisällä voidaan määritellä, mitä se lähettää vastaukseksi käyttöliittymälle silloin, kun se tekee pyynnön kyseiseen polkuun. Poluista tullaan tarjoilemaan käyttöliittymälle tietoja, joita palvelin saa MQTT-topiceista. Polkuja lisättiin demosovellukseen aluksi kaksi. Toisesta käyttöliittymän käyttöön tarjoillaan JSON-muodossa olevaa testidataa käyttöliittymän toimintojen testaukseen ja toisesta voidaan tarjoilla laitetietoja. Kuvassa 4 näkyvät polkujen sekä dataRouterin määrittely.

```
const dataRouter = Router()

dataRouter.get('/data', (request, response) => {
    response.json(data);
});

dataRouter.get('/topicdata', (request, response) => {
    response.json(topicData)
});

export default dataRouter
```

Kuva 4. Datarouter ja polkujen määrittely.

Sovelluksen palvelimen täytyy hallita MQTT-yhteyttä, jonka avulla palvelin saa dataa nykyisten analysaattoreiden rajapinnasta. Projektiin lisättiin uusi mqttClient-kansio. Kansion sisälle luotiin mqttClient.js-tiedosto, joka sisältää MQTT-yhteyden muodostamisen sekä topicien tilaamisen. Tiedoston koodaus aloitettiin sisällyttämällä mqtt-kirjasto sekä luomalla muuttujat MQTT_HOST, MQTT_PROTOCOL ja MQTT_PORT. Nämä muuttujat asetettiin käyttämään node-prosessin ympäristömuuttujia ja niiden perään luotiin myös vakiomuuttujat, jos ympäristömuuttujia ei ole määritelty. Näistä kolmesta muuttujasta rakentuu osoite MQTT-yhteydelle. Tiedoston sisälle luotu connectMqtt-funktio hallitsee yhteyden muodostamisen. Kuvassa 5 näkyy yhteyden muodostus sekä määriteltyjen topicien tilaaminen.

```
const MQTT_HOST = process.env.MQTT_HOST || 'localhost'
const MQTT_PROTOCOL = process.env.MQTT_PROTOCOL || 'mqtt'
const MQTT_PORT = process.env.MQTT_PORT || 1883

const mqttUrl = `${MQTT_PROTOCOL}://${MQTT_HOST}:${MQTT_PORT}`;

export const topicData = {};

function connectMqtt() {
    return new Promise((resolve, reject) => {
        const mqttClient = mqtt.connect(mqttUrl);

        mqttClient.on('connect', () => {
            console.log('mqtt client connected');

            mqttClient.subscribe(subscribePrefix, (error) => {
                if (error) {
                    console.log('Subscribe error: ', error);
                    reject(error);
                } else {
                    console.log(`Subscribed to all topics in ${subscribePrefix}`);
                    resolve(mqttClient);
                }
            });

        });
```

Kuva 5. Yhteyden muodostaminen ja topicien tilaaminen.

Palvelimen täytyy käsitellä suuria määriä topiceja sekä niissä olevaa dataa. MqttClientin täytyy jollain tapaa käsitellä kaikki topiceista tuleva data ja muodostaa jokaisen topicin datasta objekti käyttöliittymälle tarjoiltavaksi. Topiceista tuleva data järjestetään yhteen objektiin topicin nimen mukaan ja näiden alle tulee kyseisen topicin data. Tämä kokonainen objekti voidaan tarjota käyttöliittymälle palvelimen API end -pointista, josta käyttöliittymä saa tietoja käyttöönsä sovelluksen käynnistyessä. Topiceihin tuleva uusi data päivittyy objektiin aina, kun uusi viesti tulee mqttClient-tiin mahdollistaen sen, että aina käyttöliittymä sovelluksen käynnistyessä uudelleen on tarjoiltava data aina uusinta ja käyttöliittymä pysyy synkronoituna palvelimen tietojen kanssa. Kuvassa 6 näkyy tiedostoon lisätty asynkroninen initializeMqttClient-funktio, joka palauttaa mqttClient-objektin yhteyden muodostamisen jälkeen.

```
async function initializeMqttClient() {
    try {
        await connectMqtt();
        return mqttClientInstance;
    } catch (error) {
        console.error(`Error initializing MQTT client: ${error}`);
    }
}
```

Kuva 6. InitializeMqttClient-funktio.

Websocket-yhteyttä tarvitaan, kun back-endistä täytyy viedä reaaliajassa mittaustuloksia ja muuta dataa käyttöliittymään. Websocket-yhteyttä varten palvelimelle luotiin socket.js-tiedosto, joka hallitsee websockettien välillä tapahtuvaa tiedonsiirtoa. Tiedostoon sisällytettiin websocket-palvelimen muodostamiseen vaadittu Server-funktio Socket.io-kirjastosta. Palvelimen app.js-tiedostossa luotiin HTTP-palvelin käyttämällä Express-kirjastoa, jota käytetään uuden websocket-objektin luomiseen socket.js-tiedostossa. Websocket-yhteyttä varten muodostettiin asynkroninen funktio initializeSocket(server), joka ottaa parametrinaan aikaisemmin luodun HTTP-palvelimen. Tämän avulla luotiin kuvan 7 mukaisesti uusi io-objekti websocket-yhteyden hallitsemiseen. Tätä objektia voidaan käyttää socket.js-tiedoston sisällä kaksisuuntaiseen tiedonsiirtoon palvelimen ja käyttöliittymäsovelluksen välillä. Palvelimelle lisättiin myös mahdollisuus lähettää mqttClientin luoma dataobjekti käyttöliittymälle myös websocketin välityksellä. Socket-yhteys mahdollistaa suuren objektin lähetyksen nopeammin kuin esimerkiksi fetch-kutsu palvelimen API-endpointtiin.

Websockettien testaukseen muodostettiin hyvin yksinkertainen simulaatiodataa luova funktio hyödyntäen Math.random-funktiota. SimulationData-funktio palauttaa objektin, joka sisältää satunnaisen desimaaliarvon ja aikaleiman. Palvelimella on simulationOnOff-kanava, joka kuuntelee käyttöliittymästä tulevia viestiä ja käynnistää simulaation. Kuvassa 7 nähdään, kuinka simulaatiodataa lähetetään websocketin välityksellä käyttöliittymäsovellukselle sekunnin välein, kunnes käyttöliittymästä tulee uusi käsky, joka lopettaa simulaation.

```
io = new Server(server);

io.on('connection', (socket) => {
    console.log('New client connected');


    socket.on(channels.simulationOnOff, (msg) => {
        if(msg === 'Start') {
            isEmitting = true;
            if(isEmitting) {
                emitInterval = setInterval(() => {
                    let simuData = simulationData();
                    socket.emit(channels.simuData, simuData);
                }, 1000)
            }
        }
        if(msg === 'Stop') {
            isEmitting = false;
            clearInterval(emitInterval);
        }

    });
```

Kuva 7. Yhteyden muodostus ja simulaatiodatan lähetys käyttöliittymälle.

Käyttöliittymästä tulevia viestejä tarvitaan analysaattoreiden hallitsemiseen, joten palvelimen täytyy käsitellä myös muita käyttöliittymästä tulevia käskyjä. Palvelimelle luotiin uusi websocket-kanava, joka käsittelee käyttöliittymältä tulevia arvojen muutoksia. Palvelin vastaanottaa käyttöliittymältä tulevan JSON-muodossa olevan viestin ja muodostaa siitä väliaikaisen uuden objektin. Uusi objekti sisältää käyttöliittymästä tulevan uuden arvon tai tekstin sekä topicin, johon arvo täytyy muuttaa. Ensiksi objektissa olevan topicin avulla vertaillaan, löytyykö olemassa olevasta kaikkien topicien tietoja sisältävästä objektista sen nimistä topicia. Jos vertailtava topic löytyy, sen tiedoista muodostetaan uusi objekti käyttöliittymästä saadun arvon perusteella ja julkaistaan se topiciin analysaattoreiden rajapintaan käytettäväksi. Kuvassa 8 nähdään käyttöliittymästä tulevan viestin käsittely ja julkaiseminen sekä mahdollisen virheellisen viestin hallinta.

```
socket.on(channels.messageFromClient, (msg) => {

    try {
        let message = JSON.parse(msg);

        if(topicData.hasOwnProperty(message.topic)) {

            let topicObject = topicData[message.topic];
            topicObject.v = message.newValue;

            if(mqttClient) {
                mqttClient.publish(message.topic, JSON.stringify(topicObject));
            }
        } else {
            console.log(`Topic ${message.topic} not found in topicData`);
        }

    } catch (error) {
        console.error(`MQTT Parse error in socket.js: ${error}`);
    }

});
```

Kuva 8. Käyttöliittymästä tulevien viestin käsittely ja julkaiseminen.

Nyt palvelin pystyy kaksisuuntaiseen tiedonsiirtoon käyttöliittymän välillä ja pystyy välittämään niitä eteenpäin nykyisen rajapinnan ja analysaattoreiden käyttöön. Palvelin on tähän asti toiminut yhteistyössä käyttöliittymäsovelluksen kehityspalvelimen kanssa, mutta valmiin julkaisun täytyy olla yhtenäinen. Alkuvaiheissa luotuun app.js-tiedostoon täytyy lisätä ominaisuuksia, jotta käyttöliittymäsovellus voidaan tarjota selaimeen palvelimen kautta. Expressistä löytyy väliohjelmisto express.static, jonka avulla palvelin voi tarjoilla nettiselaimeen staattisia tiedostoja. Palvelin määriteltiin tarjoamaan staattisia tiedostoja ja tarjoilemaan nettiselaimeen pakattu React-sovellus, jos sellainen asetetusta polusta löytyy. Jos sovellusta ei ole oikeassa polussa, sovellus tarjoaa nettiselaimeen ennalta määritellyn HTML-tiedoston otsikolla ja yhdellä lauseella kertoen, että sovellusta ei löydy.

4.3 Käyttöliittymä

Vite on front-end-kehitykseen suunniteltu työkalu, joka sisältää useita ominaisuuksia helpottamaan ja nopeuttamaan sovelluskehityksen vaiheita. Ominaisuuksiin kuuluu Hot Module Replacement (HMR), joka mahdollistaa koodin muuttamisen kehityksen aikana ja päivittää sovelluksen näkymät ilman uudelleenkäynnistymistä. Vite myös mahdollistaa ES-moduulien käyttämisen se-

laimessa ilman erillistä bundlausta sovelluskehityksen aikana, mikä mahdollistaa kehityspalvelimen nopeaan käynnistymiseen. Vite käsittelee ja tukee JSX-tiedostoja suoraan ilman konfiguroimista. [19.]

Front-endin tekemisessä päädyttiin käyttämään Reactia. React projektin luomiseen käytettiin Vite-työkalua, joka mahdollistaa nopean projektin pystyttämisen ja siinä on valmis pohja React-sovellukselle. Viten avulla luotu React-sovellus on valmiiksi konfiguroitu perustarpeisiin.

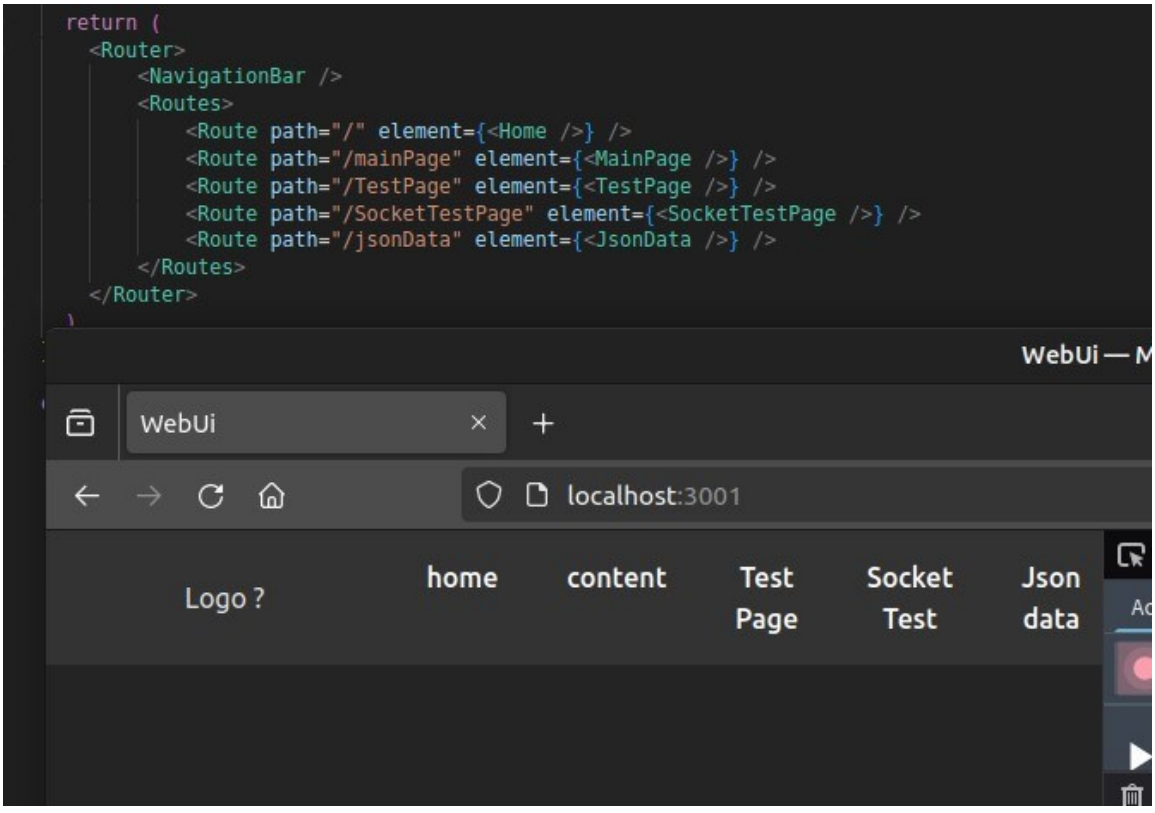
Sovelluksen luominen aloitettiin komennolla *npm create vite@latest project-name -- --template react*. Komento luo React-sovelluksen Viten React-template-pohjaa käyttäen. Projektin sisälle rakentuu valmiiksi src-kansio, joka sisältää sovelluksen lähdekoodin. Vite luo kansioon package.json-tiedoston, joka sisältää projektin tietoja ja sen vaatimat riippuvuudet. Viten avulla projektiin myös sisältyy vite.config.js-tiedosto, jossa on erilliset asetukset projektin käyttöön.

Käyttöliittymäsovelluksen src-kansion sisälle luotiin lisää kansioita, jotta projektin kasvaessa tiedostot olisivat sijoiteltuna niiden toimintojen mukaan omiin ryhmiinsä. Projektiin luotiin api-, features-, pages- ja utils-kansiot. Api-kansion sisältä löytyy socket.js-tiedosto, joka sisältää websocket-objektin. Features-kansio sisältää redux-logiikan ja jos logiikalla on oma React-komponentti, se löytyy myös features-kansiosta. Pages-kansiossa on React-komponentit, jotka esittävät yhden näkymän toiminnallisuudet erillisen sivun muodossa. Utils-kansiosta löytyy apufunktioita, kuten topicin datasta tietojen parsimiseen käytettävät funktiot. Projektiin luotiin myös app-kansio, jonka sisällä on redux-tilahallinnan store.js-tiedosto ja Reactin päänäkymän muodostava App.jsx-tiedosto.

Sovelluskehityksen ensimmäisenä vaiheena oli testata, käynnistyykö sovellus ja se käynnistetään ajamalla komento *npm run dev*. Tämä komento käynnistää React-sovelluksen käyttäen Viten kehityspalvelinta ja tässä nähtiinkin Viten avulla luodun kehityspalvelimen nopea käynnistyminen. Sovellus käynnistyikin hieman yli 200 millisekunnin jälkeen. Web-selaimen avaamalla ja siirtymällä kehityspalvelimen osoitteeseen ”http://localhost:5173” saadaan näkyviin Viten avulla luodun React-sovelluksen perusnäkymä. Seuraavaksi muokattiin app.jsx-tiedostoa ja poistettiin sieltä perusnäkymä ja aloitettiin suunnittelemaan React-sovellukseen navigointia eri näkymien välille. Components-kansioon luotiin uusi kansio navigationBar ja sen sisälle luotiin NavigationBar.jsx- ja NavigationBar.css-tiedostot. Navigointipalkin luominen aloitettiin käyttämällä React-router-kirjastoa ja tiedostoon luotiin linkityksiä muutamalle testisivulle. CSS-tiedoston sisälle lisättiin navigointipalkin muotoiluun tyylejä, jotta näkymien linkitykset eivät olisi päällekkäin. CSS-

tiedoston avulla myös luotiin navigointipalkkiin paikka, johon voidaan lisätä joko pieni kuva tai sovelluksen nimi.

Navigointipalkki täytyi liittää sovelluksen App.jsx-tiedostoon ja sinne sisällytettiin React-router-kirjastosta muutamia toimintoja. Routerin lisääminen sovellukseen tapahtuu ”käärimällä” kaikki App.jsx-tiedoston näkymät Router-toiminnon sisään. Ensimmäiseksi sisään lisättiin aikaisemmin luotu NavigationBar ja sen jälkeen Routes-toiminnon sisään määriteltiin haluttuja polkuja ja elementiksi jokaiseen polkuun tulee haluttu näkymä pages-kansion sisältä. Sovellukseen määriteltiin Home-elementti, joka tulee olemaan sovelluksen perusnäkymä sovelluksen käynnistyessä. Lisäksi näkymiin lisättiin muutamia ylimääräisiä sivuja palkin testaamiseen. Nyt sovelluksessamme on päänäkymä ja jatkuvasti näkyvillä oleva navigointipalkki, josta löytyy painikkeet näkymien vaihtamiseen kuvan 9 näkymän mukaisesti. Painamalla TestPage-nappia navigointipalkista, näkymä vaihtuu ja TestPagen sisältö tulee näkyviin ja SocketTestPagea painamalla sisältö taas vaihtuu sen sisältämään näkymään. Tässä vaiheessa näkymät olivat tyhjiä ja niihin luodaan sisältö seuraavaksi.



Kuva 9. Navigointipalkin liittäminen sovellukseen ja navigointipalkki sovelluksessa.

React-komponentteihin perehtyminen aloitettiin TestPage-näkymässä. Ensimmäiseksi näkymään luotiin muutama painonappi ja yksi select-kenttä. Tämän näkymän luomisen tarkoituksena oli

opetella ja tutustua perusteisiin React-komponenttien käytöstä. Painonappien avulla tutustuttiin Reactin koukkujen käyttämiseen. Näkymään luotiin count-tilamuuttuja lisäämällä koodiin rivi *const [count, setCount] = useState(0);*. Tämä asettaa count-muuttujan arvoksi 0 ja luo setCount-funktion, jonka avulla tilamuuttujan arvoa voidaan päivittää. Muuttujan päivittäminen myös päivittää komponentin näkymän, kun count-muuttujan arvoa muutetaan. TestPage-näkymään luoduille painikkeille asetettiin funktiot, joista toinen lisää count-muuttujan arvoa yhdellä ja toinen asettaa sen arvon takaisin nollaksi. Näkymään luotiin myös koodirivi *{counter !== 0 && Button clicked {counter} times. }*, joka sisältää jsx-elementin, mikä näytetään vain arvon ollessa erisuuri kuin 0. Select-kenttä luotiin aluksi samaan tyyliin käyttämällä tilamuuttujaa, joka päivittyy select-kentän arvoa muutettaessa ja näyttää sen näkymässä.

Komponenttien rakentaminen dynaamiseksi aloitettiin Select-komponentista. Select-kenttään täytyy saada nimi ja valikon arvot lisättyä datasta, joita käyttöliittymäsovellukseen tuotaisiin nykyisestä rajapinnasta. Components-kansion sisälle luotiin Select.jsx-tiedosto, joka tulee sisältämään toiminnallisuuden luoda näkymiin select-valikoita halutuilla arvoilla ja nimellä ja sen täytyy skaalautua arvojen määrän mukaisesti. Tiedostoon luotiin funktio, joka ottaa vastaan propseina label, options, value ja onChange ja palauttaa jsx-elementin. Sen avulla luodaan haluttuun näkymään select-komponentti, jonka arvoina on rajapinnasta tuodut arvot ja nimenä rajapinnasta tuotu nimi. Select-komponentin luomiseen käytettiin HTML-kielestä tuttuja <label>-, <select>- ja <option>-elementtejä. Ensimmäiseksi label-elementtiin liitetään rajapinnan datasta nimi, jonka jälkeen select- ja option-elementteihin haetaan arvot käyttäen JavaScriptin map-toimintoa. Kuvassa 10 näkyy select-elementin muodostamiseen käytetty funktio.

```
function Select ({label, options, value, onchange}) {

    return (
        <label>
            {label}
            <select value={value} onChange={onchange}>
                {options.map((option, index) => (
                    <option key={index} value={option.value}>
                        {option.name}
                    </option>
                ))}
            </select>
        </label>
    )
}

export default Select
```

Kuva 10. Select-komponentti.

Käyttöliittymäsovelluksen ja komponenttimäärän kasvaessa sovelluksen tilojenhallinnasta alkoi tulla hankalaa ja monimutkaista ylläpitää. Tässä apuun otettiin jo aikaisemmin esitelty Redux-toolkit. Sovellukseen täytyi saada Redux-store kaikkien komponenttien ja tietojen tilojen hallintaan. Projektiin asennettiin kirjastot React-redux ja Redux-toolkit. Tietojen hakeminen palvelimelta sovellukseen tapahtuu Axios-kirjaston avulla ja websocketin välityksellä. Redux-toolkitin avulla luodaan tilalle tapahtuma ja reducer saman tiedoston sisälle. Reduxin sisällyttäminen osaksi React-sovellusta aloitettiin socketConnectionSlice.js-tiedostosta, joka määrittelee websocket-yhteyden tilan. SocketConnection-tilaan määriteltiin connecting-muuttuja sekä socketConnection-muuttuja. Connecting-muuttuja alustettiin arvoon true ja socketConnection arvoon false. Näiden avulla sovellukseen voidaan määrittää toimintoja ja tarkastella, onko websocket-yhteyttä vai ei. Yhteyden tilojen hallintaan luotiin reducer, joka sisältää connected- ja disconnected-tapahtumat kuvan 11 mukaisesti.

```
const socketConnectionSlice = createSlice({
    name: 'socketConnection',
    initialState,
    reducers: {
        connected: (state, actions) => {
            state.connecting = false
            state.socketConnection = actions.payload
        },
        disconnected: (state, actions) => {
            state.connecting = true
            state.socketConnection = actions.payload
        }
    }
})
```

Kuva 11. Reducer websocket-yhteyden tilanhallintaan.

Kaupan muodostaminen tapahtui lisäämällä projektiin uusi tiedosto store.js. Tiedoston sisällä muodostettiin kauppa käyttäen Redux-toolkitin configureStore-funktiota, johon sisällytetään nykyiset ja tulevat reducerit. Jotta Redux-kauppa saadaan koko sovelluksen käytettäväksi, se täytyy määritellä main.jsx-tiedostossa, joka on sovelluksen päätiedosto. Websocket-yhteyden muodostamiseen tarvitaan myös yhteyden muodostava funktio. Käyttöliittymän websocket-yhteys muodostettiin käyttämällä Socket.io-client-kirjastoa uudessa socket.js-tiedostossa. Tiedostoon määriteltiin samalla myös funktio, jonka avulla käyttöliittymästä voidaan lähettää tilojen muutoksia palvelimelle käsiteltäväksi ja lähetettäväksi edelleen rajapinnan kautta analysaattoreihin. Funktio emitValueChange ottaa parametreina topicin sekä muutettavan arvon ja muodostaa näistä payload-paketin lähetettäväksi kuvan 12 mukaisesti.

Websocket-yhteys luotiin App.jsx-tiedostossa useEffect-koukun sisällä. UseEffect-koukkua käytettäessä sen sisällä olevat toiminnot suoritetaan silloin, kun käyttöliittymäsovellus renderöidään ensimmäisen kerran selaimessa. Koukun sisällä muodostetaan websocket-yhteys ja käytetään dispatch-funktiota muuttamaan yhteyden tila Redux-tilakoneeseen. Koukun sisällä hallitaan lisäksi seuraavaksi käsiteltävien dataobjektien sekä simulaatiodatan tilojen hallinta ja vastaanotto websocketeista. Sovelluksen sammuessa websocket-yhteydet katkaistaan.

```
export const socket = io(import.meta.env.VITE_WS_URL)

export function emitValueChange(t, v) {

    try {
        let payload = {topic: t, newValue: v}
        socket.emit('client-message', JSON.stringify(payload));
    } catch (error) {
        console.log(`socket.js Error: ${error}`);
    }
}
```

Kuva 12. Funktio arvojen lähettämiseen palvelimelle.

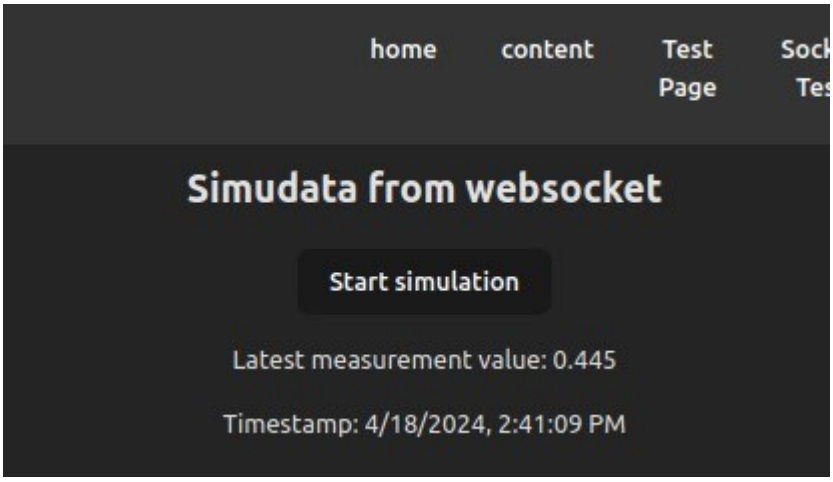
Reduxiin luotiin lisää tiloja, joiden avulla hallitaan palvelimelta saatuja dataobjekteja, jotka sisältävät analysaattoreiden rajapinnasta tuota dataa, kuten tekstejä ja muuttujien arvoja. Käyttöliittymäsovellukseen piti toteuttaa funktioita näiden tietojen käsittelyyn. Suurien dataobjektien tilojen tallentamiseen toteutettiin tähän demosovellukseen kaksi mahdollisuutta. Toinen hakee dataobjektit rajapinnasta käyttäen Axios-kirjastoa. Axios on HTTP-client JavaScript käyttöön ja sen avulla voidaan luoda asynkronisia HTTP-kutsuja. Tässä sovelluksessa kirjastoa käytettiin osana Reduxia. Reduxiin luotiin uusi tila, johon määriteltiin muuttujat loading, topicData ja error. Tila määriteltiin topicDataSlice.js-tiedostossa, jossa määritellään myös asynkroninen HTTP-kutsu. Asynkronisen tapahtuman määrittelemiseen Reduxiin käytetään createAsyncThunk-väliohjelmistoa, joka löytyy Redux-toolkitistä. Sen avulla saadaan luotua ratkaisematon (pending), ratkaistu (fulfilled) sekä hylätty (rejected) tapahtumat. Näiden tapahtumien avulla muutetaan määriteltyjä tiloja, joko Loading-tilaan tai onnistuessaan päivitetään topicData palvelimesta haetulla datalla kuvan 13 mukaisesti.

```
export const fetchTopicData = createAsyncThunk('topicData/fetchTopicData', () => {
    return axios
        .get('/api/topicdata')
        .then(response => response.data)
})

const topicDataSlice = createSlice({
    name: 'topicData',
    initialState,
    extraReducers: builder => {
        builder.addCase(fetchTopicData.pending, state => {
            state.loading = true
        })
        builder.addCase(fetchTopicData.fulfilled, (state, action) => {
            state.loading = false
            state.topicData = action.payload
            state.error = ''
        })
        builder.addCase(fetchTopicData.rejected, (state, action) => {
            state.loading = false
            state.topicData = []
            state.error = action.error.message
        })
    }
})
```

Kuva 13. Asynkroninen HTTP-kutsu tapahtumissa.

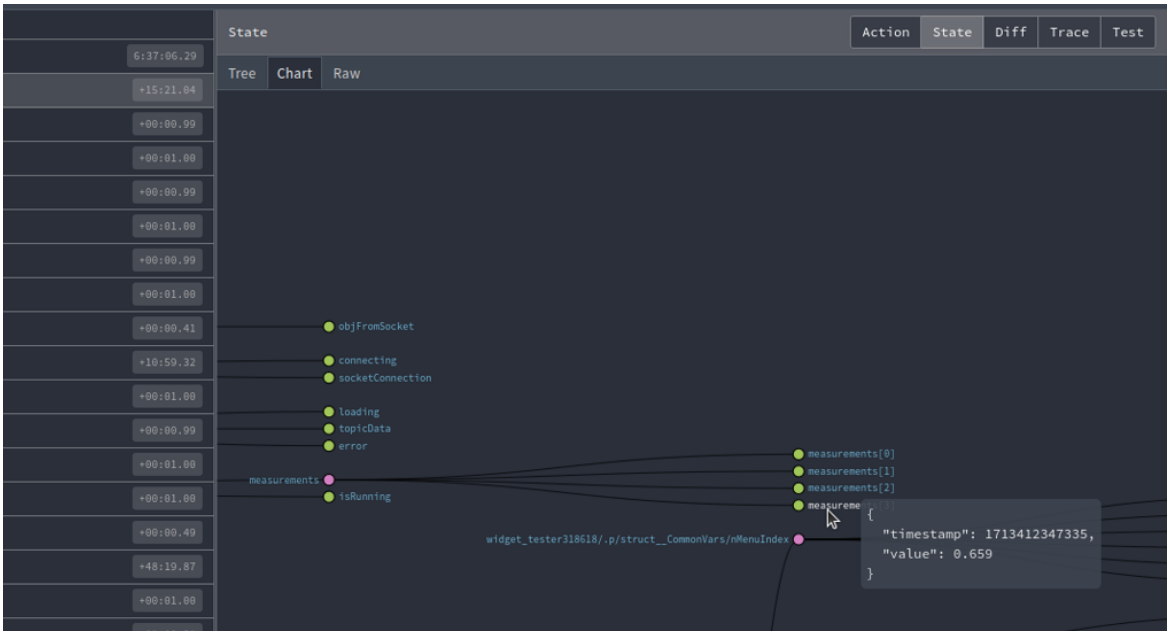
Topicien datan hakemiseen palvelimelta luotiin myös samankaltainen tilankäsittelijä, mutta käyttäen websockettia. Websocketin avulla sovelluksen toiminta nopeutuu huomattavasti ja palvelin ei myöskään rasitu niin paljoa. Tätä websockettia käytetään dataobjektien tietojen hakemiseen sovelluksessa ja HTTP-kutsulla toteutettu haku jäi demosovelluksessa vain malliesimerkiksi.

Redux-tilakoneeseen luotiin tila aikaisemmin palvelimen puolella luodulle simulaatiodatalle. Simulaatiodatan hallintaan määriteltiin 2 tilamuuttujaa. Measurements-muuttuja on taulukko, johon tallennetaan palvelimelta saatu simulaatiodatan arvo ja aikaleima. IsRunning-muuttujan avulla tutkitaan, onko simulaatio päällä vai ei. Simulaatiodata yhdistettiin osaksi SocketTest-komponenttia käyttämällä useSelector-koukkua, jonka avulla luotiin tilamuuttujat päivittymään aina, kun simulaatiodatan arvot muuttuvat. Komponenttiin luotiin osio otsikolla ”Simudata from websocket”, jonka alle luotiin painonappi ja teksti näyttämään viimeisimmän mittaustuloksen ja mittauksen aikaleiman. Painonappi lähettää palvelimelle Start- tai Stop-viestin riippuen siitä, mikä on simulaatiodatan isRunning-muuttujan tila. Simulaatiodatan ollessa käynnissä nappi lähettää Stop-viestin ja simulaatiodatan ollessa pois päältä lähetetään Start. Kuvassa 14 nähdään simulaatiodata pysäytettynä sekä viimeisin mittaustulos.



Kuva 14. Simulaatiodatan näyttäminen web-sovelluksessa.

Redux DevTools on selaimiin asennettava lisäosa, jonka avulla voidaan tarkastella sovelluksessa olevia tiloja. Työkalu on erittäin kätevä ja sisältää monia ominaisuuksia helpottamaan sovelluskehitystä. Työkalun avulla pystyy esimerkiksi tarkastelemaan tiloja graafisessa muodossa ja tilojen muutoksia voidaan tarkastella yksitellen. Redux DevToolsin avulla tilojen muutoksia voidaan yksitellen selata taaksepäin tai eteenpäin ja nähdä muutokset suoraan käyttöliittymässä. Työkalu näyttää jokaisen tilan arvot yksittäin kuvan 15 mukaisesti.



Kuva 15. Redux DevTools -näkymä ja simulaatiodatan arvo sekä aikaleima.

Useiden komponenttien liittäminen samalle sivulle alkoi kasvattamaan tiedoston pituutta. Komponenttien muodostamiseen vaadittuja toimintoja siirrettiin osaksi komponenttia. Select-komponentin luomiseen vaaditut attribuutit määriteltiin aikaisemmin osana jokaista näkymää. Tämä aiheutti koodiin useita toistoja ja koodista haluttiin luoda selkeämpää. Komponenttia muutettiin ja komponentin propseina välitettiin pelkästään topic, jonka tiedoilla komponentti luodaan. Komponentin sisällä määriteltiin selectValue-, text- ja options-muuttujat ja muuttujiin haettiin tiedot käyttämällä apufunktioita getValue(), getText() sekä getSelectValues(). Komponentin sisällä hoidettiin valinnan vaihto ja muuttuneen arvon välittäminen palvelimelle. Tämän muutoksen avulla tarvittavaa koodia saatiin vähennettyä ja samalla koodin lukeminen helpottui ja useiden samojen komponenttien käyttö yhdessä näkymässä oli selkeämpää. Kuvassa 16 nähdään päivitetty Select-komponentti.

```
function SelectComponent ({topic}) {

    const topicData = useSelector((state) => state.wsTopicData.wsTopicData);

    const [selectValue, setSelectValue] = useState('')
    const [text, setText] = useState('');
    const [options, setOptions] = useState([]);

    const setSelect = () => {
        setSelectValue(getValue(topicData[topic]));
        setText(getText(topicData[topic]));
        setOptions(getSelectValues(topicData[topic]));
    }

    const handleChange = event => {
        setSelectValue(event.target.value);
        emitValueChange(topic, event.target.value)
    }

    useEffect(() => {
        setSelect();
    }, [topicData])

    return (
        <label>
            {text}
            <select value={selectValue} onChange={handleChange}>
                {options.map((option, index) => (
                    <option key={index} value={option.value}>
                        {option.name}
                    </option>
                ))}
            </select>
        </label>
    )

}

export default SelectComponent;
```

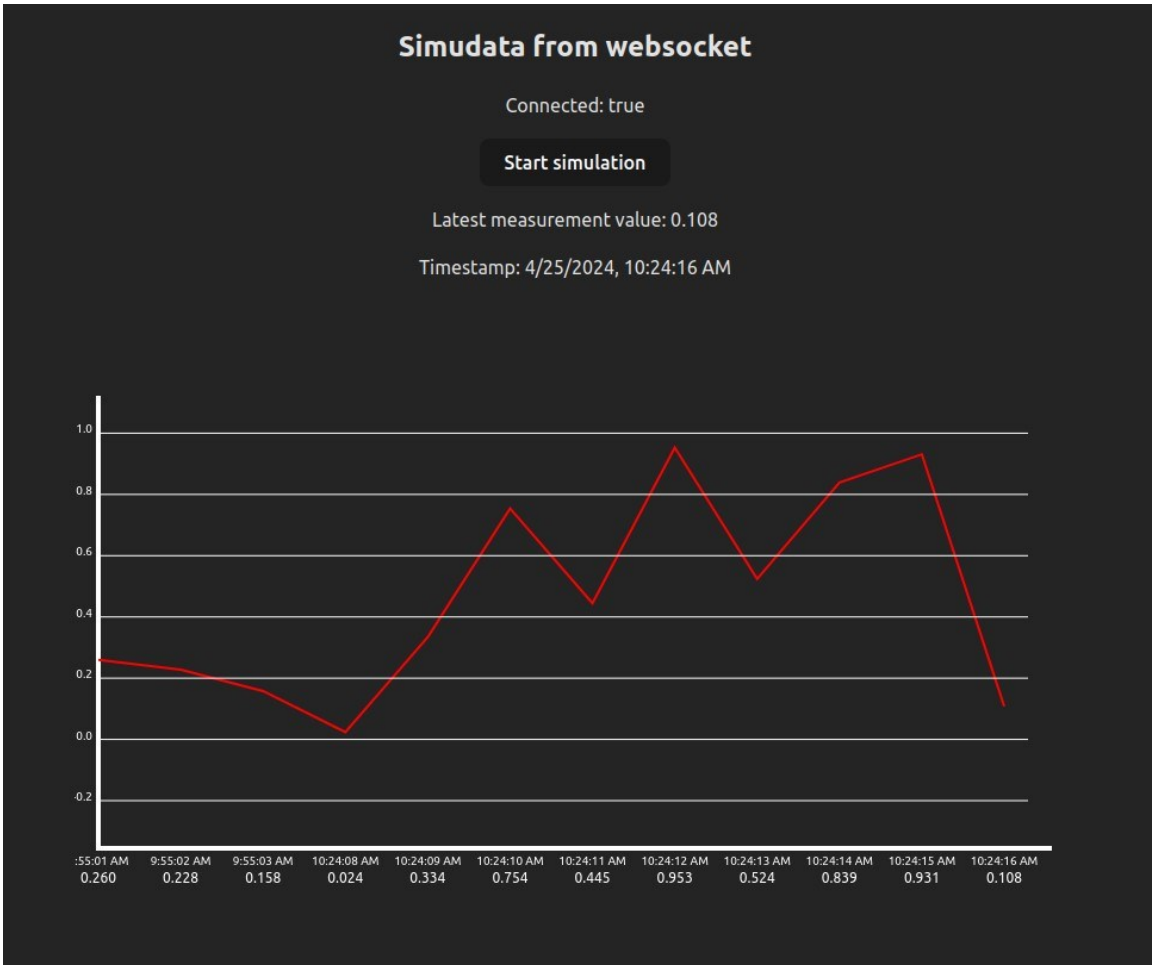
Kuva 16. Päivitetty Select-komponentti.

Valintaruutujen luomiseen käytetty CheckBoxComponent muutettiin edellä esitellyn Select-komponentin kaltaiseksi ja nyt molempia komponentteja voidaan käyttää sovelluksen sivujen luomiseen helpommin. Apufunktioita luotiin sovelluksen utils-kansioon index.js-tiedoston sisälle ja apufunktioiden avulla halutusta topicista palautetaan vaadittuja arvoja, joita käytetään komponentin sisällä. Apufunktiot ottavat parametreina dataobjektin, joka sisältää valitun topicin tiedot. Kuvassa 17 näkyvässä getSelectValues-funktiossa määritellään väliaikainen taulukko ja tarkastellaan, sisältääkö välitetty dataobjekti attribuutin SEL. SEL-attribuutti objektissa tarkoittaa, että objekti sisältää valintaruudun arvot. Lopuksi funktio palauttaa arvot komponentin käyttöön.

```
export function getSelectValues(dataObject) {

    let values = []

    try {
        if (dataObject.v_attr[0] === properties.SEL) {
            for(let i = 1; i < dataObject.v_attr.length; i++) {
                values.push({name: dataObject.v_attr[i]})
            }
        }
    } catch (error) {
        console.log('utils/index getSelectValues error: ' + error)
    }

    return values
}
```

Kuva 17. GetSelectValues-funktio.

Aikaisemmin luodun simulaatiodatan näyttämiseen haluttiin luoda graafi, joka piirtää simulaatiodatan arvot käyränä. Graafin luominen simulaatiodatan käyttöön aloitettiin määrittelemällä sen koko. Graph-komponenttiin viedään parametrina measurements-taulukko, joka sisältää simulaatiodatan arvot ja aikaleiman. Komponentin sisällä taulukosta tallennetaan viimeisimmät 12 arvoa ja arvojen aikaleimat JavaScriptin slice()-funktion avulla. Graafin luomiseen komponentin sisällä määriteltiin funktio, jonka avulla graafiin saadaan vaakatasoon apuviivoja sekä niiden tekstit. Graafin käyrän piirtämiseen käytettiin SVG-path-elementtiä ja sen sisältämiä moveTo- sekä lineTo-attribuutteja. Komponentti graafin luomiseen jäi opinnäytetyössä demovaiheeseen, mutta sen toiminta saatiin toteutettua. Graafin alalaidassa näkyvät simulaatiodatan arvot ja aikaleima jäävät hieman piiloon vasemmasta laidasta. Graafi päivittyy jatkuvasti simulaatiodatan päivittyessä ja näyttää graafina viimeisimmät 12 arvoa ja mittauksien aikaleiman kuvan 18 mukaisesti.



Kuva 18. Simulaatiodata graafina.

Sovellusta oli pääasiassa käytetty kehityspalvelimen avulla ja sovelluksen näyttäminen selaimeen back-endin avulla vaatii käyttöliittymäsovelluksen sekä sovelluksen riippuvuuksien pakkaamisen yhdeksi tiedostoksi. Sovelluksen ja sen riippuvuuksien pakatusta versiosta käytetään nimitystä bundle. Käyttöliittymän pakkaamiseen määriteltiin sovelluksen vite.config.js-tiedostoon polku ja kansion nimi, johon haluttu bundle viedään sovellusta pakattaessa. Sovelluksen package.json-tiedostoon täytyi määritellä myös komento, jonka avulla sovellus pakataan. Tiedostoon luotiin komento *cd src/client && vite build –config ../../vite.config.js*, joka suorittaa *vite build*-komennon käyttöliittymäsovelluksen kansiossa käyttäen vite.config.js-tiedoston määrittelyjä. Samaan tiedostoon luotiin myös komento, joka suorittaa build-komennon sekä käynnistää back-end-palvelimen. Tämän avulla käyttöliittymä pakataan aina uudestaan muutoksien varalta ja käynnistetään palvelin, josta sovellus tarjoillaan selaimeen käytettäväksi.

5 Yhteenveto

Opinnäytetyön tavoitteena oli tutkia nykyaikaisia web-teknologioita sekä suunnitella ja toteuttaa web-sovellus, jonka avulla voidaan lähteä kehittämään uusia käyttöliittymäsovelluksia analysaattoreihin. Työn alkuvaiheissa työhön kuului hyvin paljon web-teknologioihin ja käytettyihin menetelmiin tutustumista. Sovelluksen suunnittelussa täytyi huomioida analysaattoreiden nykyisten käyttöliittymien käyttämät parametrit ja suunnitella sovellus toimimaan niitä käyttäen. Sovellukselle luotiin palvelin, joka pystyy kaksisuuntaiseen kommunikaatioon analysaattoreiden nykyisen rajapinnan kanssa. Sovelluksen käyttöliittymään saatiin luotua erilaisia näkymiä ja näkymiin luotiin toimivia käyttöliittymäkomponentteja nykyisestä rajapinnasta saatujen tietojen avulla. Käyttöliittymäsovelluksesta valintaruutuja tai nappeja painellessa palvelin välittää tiedot rajapintaan ja käyttöliittymässä tapahtuvat muutokset näkyvät myös vanhoissa sovelluksissa samaan aikaan. Käyttöliittymään luotiin myös graafi, joka piirtää käyrän palvelimella kehitetystä simulaatiodatasta.

Työssä toteutettuun demosovellukseen jäi myös jatkokehitysideoita, joista yksi on erillinen tietokanta. Tietokannan avulla sovellukseen voitaisiin luoda käyttäjienhallinta ja ominaisuuksia voitaisiin näyttää käyttäjätason mukaisesti ja käyttöoikeuksilla voitaisiin määritellä, mihin toimintoihin käyttäjällä on pääsy. Tämä on tarpeellista laajoissa tehdasympäristöissä toimivalle sovellukselle. Sovelluksen palvelimen toteutuksessa käytetty Express-kirjasto on myös mahdollista korvata monia nykyaikaisia teknologioita käyttäen, mutta opinnäytetyössä käytetty aika ei riittänyt toteuttamaan sovellusta useilla teknologioilla. Express-kirjasto oli myös valmiiksi tuttu ja nopeutti palvelimen kehitystyötä ja siihen on saatavilla paljon valmiita funktioita helpottamaan palvelimen rakentamista. Työn alkuvaiheissa tutustuttiin myös Vue.js-kirjastoon ja kirjaston avulla toteutettiin pieniä testisovelluksia, mutta lopullinen sovellus päädyttiin toteuttamaan käyttämällä React-kirjastoa. Vue.js-kirjaston käyttö osoittautui monimutkaisemmaksi ja sen avulla yksinkertaisten komponenttien luominen palvelimelta tulevilla tiedoilla sekä tilojen hallitseminen oli hankalampaa.

Työssä toteutettuun sovellukseen ei toteutettu sovellustestaukseen vaadittavia komponenttitestejä. Sovelluksen osien testaamiseen voidaan käyttää esimerkiksi Jest-kirjastoa. Vite sisältää myös komponenttien ja sovelluksen osien testaamiseen vaadittavia ominaisuuksia. Testejä voidaan kirjoittaa sovellukselle ja testien avulla pystytään testaamaan esimerkiksi, onko komponenttien

tekstit oikeat tai renderöidäänkö komponentti näkymään. Jatkon kannalta testejä olisi sovelluksen kasvaessa tarpeen luoda, jotta kehityksessä päästään korjaamaan mahdollisia ongelmia heti kehitysvaiheessa.

Toteutetun sovelluksen perusteella nykyisten analysaattoreiden käyttöliittymät on mahdollista korvata käyttämällä nykyaikaisia web-teknologioita. Sovelluksen avulla voidaan lähteä kehittämään ja suunnittelemaan analysaattoreiden käyttöliittymien uudistamista nykyaikaisilla web-teknologioilla. Työssä päästiin asetettuihin tavoitteisiin ja se tarjoaa hyvän pohjan jatkokehitykselle ja käyttöliittymien uudistamiselle.

Lähteet

1. Valmet lyhyesti. Valmet. [Internet]. [viitattu 25.4.2024]. Saatavilla: https://www.valmet.com/fi/valmet-yrityksena/valmet-lyhyesti/

2. What is Web Application (Web Apps) and its Benefits. TechTarget. [Internet]. [viitattu 25.1.2024]. Saatavilla: https://www.techtarget.com/searchsoftwarequality/definition/Web-application-Web-app

3. Web Applications | What is Web Application – Javatpoint. Javatpoint. [Internet]. [viitattu 25.1.2024]. Saatavilla: https://www.javatpoint.com/web-application

4. Web Application Development: Process, Tools & Examples. Browserstack. [Internet.] [viitattu 29.2.2024]. Saatavilla: https://www.browserstack.com/guide/web-application-development-guide

5. What is HTML and How Does Hypertext Markup Language Work? Theserverside. [Internet]. [viitattu 5.2.2024]. Saatavilla: https://www.theserverside.com/definition/HTML-Hypertext-Markup-Language

6. What is CSS? The Ultimate Intro – Code Institute Global. Codeinstitute. [Internet}. [viitattu 8.2.2024]. Saatavilla: https://codeinstitute.net/global/blog/what-is-css-and-why-should-i-learn-it/

7. Introduction to the DOM – Web APIs | MDN. Developer.mozilla. [Internet]. [viitattu 12.2.2024]. Saatavilla: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

8. What Is a Single Page Application? Bloomreach. [Internet]. [viitattu 14.2.2024]. Saatavilla: https://www.bloomreach.com/en/blog/2018/what-is-a-single-page-application

9. Chiarelli A. *Beginning React: Simplify Your Frontend Development Workflow and Enhance the User Experience of Your Applications with React*. 1st edition. Birmingham; Mumbai: Packt Publishing, 2018.

10. Built-in React Hooks. React [Internet]. [viitattu 11.4.2024]. Saatavilla: https://react.dev/reference/react/hooks

11. React's Virtual DOM. The virtual DOM (VDOM) is a concept. Medium. [Internet]. [viitattu 28.2.2024]. Saatavilla: https://medium.com/@BharathkumarV/reacts-virtual-dom-17fdcb290a10

12. Writing Markup with JSX – React. React.dev. [Internet]. [viitattu 14.2.2024]. Saatavilla: https://react.dev/learn/writing-markup-with-jsx

13. ReactJS Components – Javatpoint. Javatpoint. [Internet]. [viitattu 29.2.2024]. Saatavilla: https://www.javatpoint.com/react-components

14. Functional Components Vs Class Components in React JS. Geekster. [Internet]. [viitattu 29.2.2024]. Saatavilla: https://blog.geekster.in/functional-components-vs-class-components/

15. Redux Fundamentals, Part1: Redux Overview. Redux. [Internet]. [viitattu 14.2.2024]. Saatavilla: https://redux.js.org/tutorials/fundamentals/part-1-overview

16. Redux Fundamentals, Part 8: Modern Redux with Redux Toolkit. Redux. [Internet]. [viitattu 11.4.2024]. Saatavilla: https://redux.js.org/tutorials/fundamentals/part-8-modern-redux

17. What is MQTT? MQTT Protocol Explained. AWS. [Internet]. [viitattu 20.3.2024]. Saatavilla: https://aws.amazon.com/what-is/mqtt/

18. WebSocket API and protocol explained. Ably. [Internet]. [viitattu 26.3.2024]. Saatavilla: https://ably.com/topic/websockets

19. Why Vite. Vite [Internet]. [viitattu 2.4.2024]. Saatavilla: https://vitejs.dev/guide/why.html