Opinnäytetyö (AMK)

Tietojenkäsittely

2019

Jani Koski

REACT-SOVELLUKSEN FRONT END -ARKKITEHTUURI

– Case Wunder Finland Oy

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietojenkäsittely

2019 | 48 sivua

Jani Koski

REACT-SOVELLUKSEN FRONT END -ARKKITEHTUURI

- Case Wunder Finland Oy

Tämän opinnäytetyön tavoitteena oli toteuttaa toimeksiantoyritykselle ohjelmistoprojektien Git-versionhallinnan dataa visualisoiva verkkosovellus hyödyntäen React-ohjelmistokirjastoa. Tällä parannettaisiin ymmärrystä siitä, millaista teknistä työtä yrityksessä parhaillaan tehdään. Samalla oli tarkoitus pohtia, kuinka React sekä muut olennaiset työssä käytettävät teknologiat soveltuvat toimeksiannon mukaisen sovelluksen toteutukseen. Koska työ painottui pääosin sovelluksen näkyvään osaan eli front endiin, oli työn tarkoituksena myös tutustua verkkosovellusten käyttöliittymien toimintaan yleisesti sekä tutkia, kuinka niiden suorituskykyä voidaan optimoida.

Opinnäytetyön tutkimusmenetelmä oli konstruktiivinen, sillä johtopäätökset saatiin pääasiassa React-sovelluksen kehitystyöstä. Teoriaosuudessa perehdyttiin verkkosovellusten käyttöliittymien toimintaan yleisesti, suorituskyvyn optimointiin sekä työssä käytettävien teknologioiden ominaispiirteisiin.

Sovellukselle luotiin täysin toimiva pohja. Saatavilla olevan datan rakenne määriteltiin luomalla kevyt GraphQL-rajapinta, jota hyödynnettiin React-käyttöliittymässä. Datan visualisoinnissa hyödynnettiin Reactille kehitettyä Victory-kirjastoa, jonka kaaviokomponentit sopivat hyvin projektin tarpeisiin niiden helppokäyttöisyytensä vuoksi. Käyttöliittymä jäi kuitenkin melko kevyeksi, sillä saatavilla olevan datan muuntaminen visualisoitavaan muotoon oli aikaa vievää. Sovelluksen ulkoasu viimeisteltiin CSS:llä.

Lähdekoodi ladattiin yrityksen GitHub-ohjelmavarastoon jatkokehitysmahdollisuuksia varten.

ASIASANAT:

React, Javascript, Victory.js, GraphQL, Front end, Suorituskyky

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Business Information Technology

2019 | 48 pages

Jani Koski

FRONT END ARCHITECTURE IN REACT APPLICATIONS

- Case Wunder Finland Ltd

The goal of this thesis was to implement a React-based web application which visualizes Git version control data of Wunder Ltd software projects, and therefore gives a better awareness of updates around the company in terms of technical work. The main focus was on delving into the main features of the React software library along with a few other relevant tools that would be used for building the application, while considering how suitable they would be for the assignment, namely the visualization of version control data. Because the assignment's emphasis was on the client side of the application, generally known as front end, a slight emphasis was also given on investigating front end performance.

The research method for this thesis was constructive as developing the React application provided most of the conclusions. The theoretical part of the thesis covers how the client side of the web works, how its performance can be optimized, and the main technologies that were used for building the application.

As a result, a well functional base for the application was developed. A small GraphQL API was first implemented which defined the structure of the available data, and that data was then queried from the React front end. Victory.js, a data visualization library for React was added to handle the data visualization part of the application, as it provided easy-to-use chart components. However, turning the data, into effective visualizations was time-consuming and, therefore, the user interface was left to be quite small in size. The look of the application was finalized with CSS.

The source code was uploaded to the company's GitHub hosting storage where it is available and can be developed furthermore.

KEYWORDS:

React, Javascript, Victory.js, GraphQL, Front end, Performance

SISÄLTÖ

KUVAT

SANASTO

Commit	Muutos ohjelmistoprojektin Git-versionhallinnassa (Git-scm.com 2019)
CSS	Tyyliohje HTML-merkkauskielelle (W3schools.com 2019)
Elasticsearch	Hakukoneohjelmisto
Front end	Verkkosovellusten käyttöliittymäpuoli eli näkyvä osa
Git	Versionhallintatyökalu ohjelmistoprojekteille (Atlassian.com 2019)
GitHub	Hostingpalvelu Git-versionhallintaa käyttävälle lähdekoodille (Guides.github.com 2019)
GraphQL	Rajapintastandardi sekä kyselykieli
HTML	Verkkoselainten standardoitu merkkauskieli (W3schools.com 2019)
Javascript	Ohjelmointikieli HTML-merkkauskielelle ja verkkoselaimille (W3schools.com 2019)
Props	Reactin ominaisuus, jonka avulla voidaan siirtää dataa React-komponentista toiseen parametrien kautta (W3schools.com 2019). Selkokielen vuoksi käytän tässä työssä pääasiassa propseista termiä parametri
React	Javascript-kirjasto pääasiassa yhden sivun verkkosovelluksia varten
Victory.js	Javascript-kirjasto Reactille datan visualisointia varten

1 JOHDANTO

Tämän päivän ohjelmistokehityksessä Facebookin kehittämä React Javascript-kirjasto on noussut erittäin suosituksi vaihtoehdoksi verkkosovellusten käyttöliittymien kehittämiseen. Kesäkuussa 2018 se mainittiin 28 %:sta Yhdysvaltain työpaikkailmoituksista, jotka hakivat ohjelmistokehittäjiä, ja samaan aikaan se oli suosituin ladattu ohjelmisto NPM-rekisteristä (Kostrzewa 2018). Keskimääräisesti React-ohjelmistopakettia on ladattu viikoittain yli viisi miljoonaa kertaa tämän opinnäytetyön kirjoittamisen aikaan syksyllä 2019 (Npmjs.com 2019).

React-sovelluksissa liikkuu usein muitakin palasia kuin itse Reactin runko. Monet sovellukset hyödyntävät datan visualisointia jossain määrin, mikä yleensä vaatii erillisen kirjaston asennuksen. Lisäksi visualisoitava data on oikean elämän haasteissa lähes aina dynaamista ja voi tulla melkein mistä tiedonlähteestä tahansa. Edellä olevat palaset pitäisi jotenkin saada jäsenneltyä kohdalleen toimivan sovelluksen muodossa. Näiden lisäksi lähdekoodin tulisi olla optimoitu suorituskyvyn ja käyttökokemuksen parantamiseksi.

Opinnäytetyö tehdään toimeksiantona Wunder Finland Oy:lle, joka on digitaalisia palveluja ja konsultointia tarjoava ohjelmistoalan yritys. Tavoitteena on luoda React-pohjainen verkkosovellus, joka visualisoi yrityksen ohjelmistoprojektien Git-dataa, ja näin ollen antaa paremman ymmärryksen siitä, millaista teknistä työtä yrityksessä parhaillaan tehdään. Reactia käytetään oletuksena käyttöliittymien kehitykseen lähes kaikissa yrityksen alkavissa asiakasprojekteissa tämän raportin kirjoittamisen aikaan syksyllä 2019. Itse aihe on osana niin sanottua innovaatioprojektia, jonka tavoitteena tuoda esiin uusia ideoita, joilla voitaisiin edistää yrityksen toimintaa.

Työn rakenne muodostuu niin, että teoriaosiossa tutustutaan ensin verkkosovelluksen käyttöliittymän toimintaan yleisesti sekä tutkitaan, kuinka niiden suorituskykyä voidaan optimoida. Tämän jälkeen muodostetaan teoriaosion tärkein luku, jossa perehdytään kehitettävän sovelluksen käyttöliittymän teknologioihin. Näitä ovat mm. React-kirjasto, Create-React-App-kehitystyökalu sekä Victory-kaaviokirjasto. Teoriaosion viimeisessä luvussa tutustutaan myös lyhyesti työssä käytettäviin palvelinpuolen teknologioihin, joista tärkein on GraphQL rajapinnan muodostamista varten. Ohjelmistoprojektien alkuperäinen datan lähde on yrityksen Elasticsearch-hakukone. Toteutusosiossa

raportoidaan sovelluksen toteutus pohjautuen teoriaosuudessa käsiteltyihin teknologioihin.

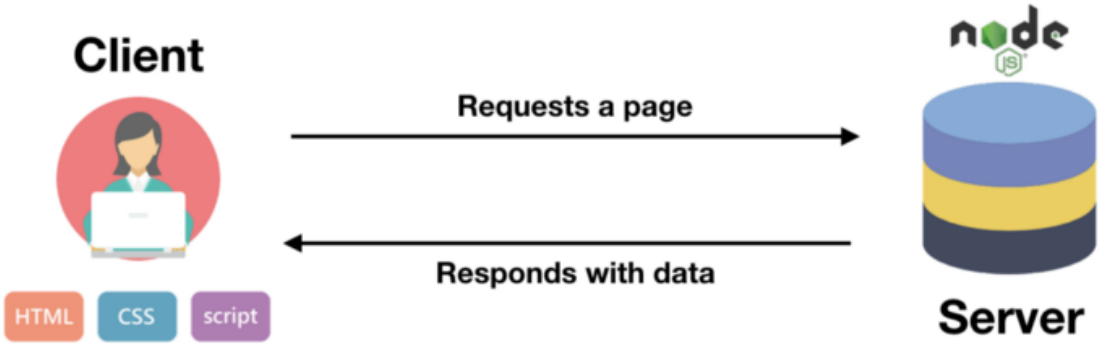
Opinnäytetyön tutkimusote on konstruktiivinen, sillä työssä kehitetään toimeksiannon mukaisesti dataa visualisoiva React-sovellus.

2 VERKKOKÄYTTÖLIITTYMÄN SUORITUSKYKY

Tässä luvussa perehdytään suorituskyvyn merkitykseen verkkosivustoilla sekä tutkitaan, kuinka verkkosivuston käyttöliittymän suorituskykyä voidaan parantaa staattisia tiedostoja optimoimalla (CSS, Javascript, kuvat). Toteutusvaiheessa staattiset tiedostot optimoidaan Create-React-App-kehitystyökalua hyödyntäen, josta käsittelen teoriaa luvussa 3.3.

Käyttöliittymän suorituskyvyn optimointia varten on kuitenkin oleellista ymmärtää niiden toiminta yleisellä tasolla verkkosivustoilla. Termillä front end viitataan juurikin verkkosivuston tai -sovelluksen näkyvään osaan eli käyttöliittymään. Käytännössä tämä siis tarkoittaa verkkoselainta. Kaikkien verkkoselaimien toiminta perustuu siihen, että niissä ajettavat standardoidut kielet HTML, CSS ja Javascript yhdessä määrittelevät sen, mitä käyttäjä näkee sivustolla. On myös tärkeää ymmärtää, että termillä back end viitataan palvelinpuolen asioihin, joita ovat mm. verkkopalvelimet ja tietokannat. (Upwork.com 2019.)

Kuva 1 (Gouirhate 2017) tarjoaakin havainnollistettuna selaimen ja palvelimen välisen toiminnan: käyttäjä lähettää kutsun palvelimelle, joka taas lähettää vastauksen määritellyn datan muodossa. Selaimessa HTML, CSS sekä Javascript muovaavat loppukäyttäjälle sivun näkyvän osan sekä interaktiivisuuden.



Kuva 1. Selain-palvelin-arkkitehtuuri (Gouirhaite 2017).

2.1 Suorituskyvyn merkitys

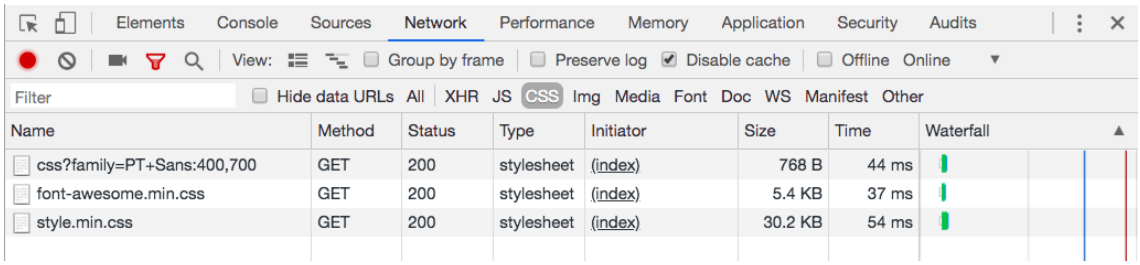
Tämän päivän verkkosivuilla sekä sovelluksissa on enemmän ominaisuuksia kuin koskaan aiemmin. Niin paljon, että monet sivustot tuskailevat suorituskyvyn kanssa, ja pahimmillaan suorituskykyongelmat voivatkin tehdä sivustosta täysin saavuttamattoman. Latausruudun katselemisen sijaan verkkokauppojen asiakkaat todennäköisesti haluavat ostaa tuotteita, blogisivustojen selaajat lukea blogeja ja sosiaalisen median käyttäjät olla vuorovaikutuksessa muiden käyttäjien kanssa. (Wagner 2019.) Suorituskyvyn rooli etenkin verkkokauppojen toiminnassa on näin ollen erittäin tärkeä.

Lisäksi Google jopa rankaisee huonon suorituskyvyn omaavia sivustoja hakutuloksissaan (Singhal & Cutts 2019).

2.2 CSS:n ja Javascriptin optimointi

Jokainen HTTP-kutsu palvelimelle kasvattaa sivuston latausaikaa, joten niin Javascript- kuin CSS-tiedostot on suositeltavaa yhdistää niin vähäiseen määrään kuin mahdollista tuotantoversiota varten, jolloin minimoidaan kutsujen määrä. Lisäksi koodin tiivistyksellä (minify) pienennetään tiedostokokoa, jolloin myös tietoa tarvitsee siirtää pienempi määrä. (Dorcey 2014, 24.)

Google Chrome -selaimen kehitystyökalujen Network-välilehden alta voi tarkkailla HTTP-kutsuja ja jopa suodattaa niitä tiedostotyyppien mukaan. Lisäksi nähtävillä on mm. tiedostokoko sekä latausaika millisekunteina. (Kuva 2.)



Kuva 2. Ladatut tyylitiedostot kehitystyökaluikkunassa (Google Chrome 2019).

Kuvan 2 esimerkkitapauksessa on suodatettu ladatut resurssit CSS:n mukaan. Samassa nähdään myös, miltä sivustolle ladattujen tyylitiedostojen olisi hyvä näyttää:

tyylitiedostoja ladataan vain muutama kappale, joista style.min.css-tiedosto viittaa selvästi, että kyseessä on sivuston päätyylitiedostot yhdistettynä yhdeksi tiivistetyksi tiedostoksi. Lähdekoodissahan tyylitiedostoja voisi olla jopa satoja, mutta yhdistämällä ne saadaan yhdeksi tiedostoksi. Kuvan 2 esimerkissä ladataan päätyylitiedoston lisäksi vain fontit, ja tässä tapauksessa kolmen ladatun tiedoston yhteenlaskettu latausaika on 135 millisekuntia. Samasta kuvasta voi myös hyvin laskea, kuinka millisekunnit kasvaisivat huomaamatta sekunneiksi, jos ladattavia resursseja olisi esimerkiksi satoja.

2.3 Kuvien optimointi

Kuvia tulisi osata käyttää oikein, sillä ne ovat yleensä verkkosivuille ladatuista tiedostoista eniten tilaa vieviä. Suuret kuvat luonnollisesti hidastavat sivun latausta, mutta vievät myös kaistanleveyttä sekä palvelintilaa (Huttunen 2019).

Tässä opinnäytetyössä oleellisinta on oikean kuvaformaatin käyttäminen, sillä käytetyt kuvat muodostuvat lähinnä logosta sekä kaavioista. Vektoriformaatti (SVG) säilyttää tarkkuutensa kaikilla resoluutioilla sekä zoomatessa tehden siitä ideaalin nimenomaan yksinkertaisille kuvioille, kuten logoille ja kaavioille. Koska SVG-kuvat muodostuvat XML-koodista, niin myös niiden optimointiin tiivistys ja pakkaus riittävät hyvin. (Grigorik 2019.)

Jos optimointia halutaan edistää entisestään, niin CSS Sprite -tekniikan avulla on mahdollista yhdistää useampi kuva yhdeksi, josta haluttu kuva valitaan CSS:ää käyttämällä (Dorcey 2014, 28).

3 REACT-KIRJASTO

React on Facebookin kehittämä erittäin suosittu, verkkosovellusten käyttöliittymien rakentamista varten luotu Javascript-kirjasto. Vuonna 2013 päivänvalon nähnyt kirjasto luotiin alun perin ratkomaan suurten tiedonsiirtopainotteisten sivustojen sekä dokumenttioliomalliin (DOM) liittyviä ongelmia. (Banks & Borcello 2017, 1.)

Banksin ja Borcellon (2017, 2) mukaan on myös tärkeää tiedostaa, että React on nimenomaan kirjasto käyttöliittymän rakentamista varten, eikä näin ollen kanna mukanaan kaikkia samoja elementtejä kuin perinteinen Javascript-kehikko (framework).

Tyypillisessä React-sovelluksessa React upotetaan render-metodin kautta haluttuun HTML-elementtiin, joka toimii näin ollen sovelluksen juurielementtinä. Kuvan 3 esimerkissä tulostetaan perinteinen Hello World.

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <h1>Hello world</h1>,
  document.getElementById('container')
);
```

Kuva 3. Hello Worldin tulostus Reactissa render-metodin avulla.

3.1 React-komponentit sekä JSX-syntaksi

React-kirjaston ydin perustuu komponentteihin, jotka yksinkertaistettuna ovat joko Javascript-luokkia (class) tai perinteisiä ohjelmia (function), joilla kuvataan käyttöliittymän eri lohkoja (Kagga 2018). Komponenttien avulla voidaan käyttöliittymän lähdekoodi jakaa itsenäisiksi ja uudelleenkäytettäviksi kokonaisuuksiksi, jolloin mm. sovelluksen rakenne on huomattavasti selkeämpää hahmotella (Reactjs.org 2019).

Yksi olennaisimmista React-kirjaston tarjoamista ominaisuuksista on myös tämän syntaksilaajennus nimeltä JSX, jonka perimmäinen tarkoitus on määritellä selaimeen tulostettavan osion rakenne (Reactjs.org 2019). Kuvasta 4 nähdään, kuinka JSX on lähes identtistä perinteisen HTML:n kanssa.

```
import React from 'react';
import ReactDOM from 'react-dom';

// JSX:
const jsxElement = (
  <div>
    <h1>React opinnäytetyö-esimerkki</h1>
    <p>Ja tässä lisää tekstiä</p>
  </div>
);

// Render
ReactDOM.render(jsxElement, document.getElementById('container'));
```

Kuva 4. JSX-syntaksi Reactissa.

3.2 Virtuaalinen DOM

Perinteisessä yhden sivun verkkosovelluksessa käyttäjän saapuessa sivulle, palvelin palauttaa vain yhden HTML-dokumentin, minkä jälkeen Javascript hallitsee sovelluksen toimintaa taustalla, yleensä AJAX-kutsujen avulla. Tämä mahdollistaa esimerkiksi sen, että jokin sovelluksen tietty osa voi kutsua dataa palvelimelta, ja päivittyä ilman, että koko sivu ladataan uudestaan. (Banks & Borcello 2017, 62.)

Ongelmallista on kuitenkin edelleen se, että kun jokin tietty osio sovelluksen DOM:sta päivittyy, joutuu selain uudelleen piirtämään kyseisen osion CSS:n, mikä on selkeästi ihmissilmälle havaittava prosessi. Tämä tarkoittaa sitä, että parhaan lopputuloksen saa vähentämällä ja optimoimalla DOM-muutokset niin, että vain ja ainoastaan tarvittava osa sovelluksesta päivittyy, jolloin CSS:n uudelleen piirtäminen minimoidaan. Kyseisen ongelman ratkaisu onkin Reactin tarjoaman virtuaalisen DOM:in taustalla. (Minnick 2019.)

Vaikka toteutusosiossa ei siis varsinaisesti nähdä virtuaalista DOM:ia, niin koin tarpeelliseksi raportoida sen tähän, koska kyseessä on yksi Reactin tärkeimpiä käsitteitä teknisellä tasolla.

3.3 Create-React-App

Create-React-App on Reactin kehittäjien luoma kehitystyökalu, joka helpottaa sekä nopeuttaa React-sovelluksen aloittamista, sillä tarvittavat moduulit ja konfiguraatiot on asennettu valmiiksi taustalle. Kehittäjä voikin näin ollen muutamalla komennolla ajaa kehitysympäristön pystyyn ja tämän jälkeen keskittyä olennaiseen eli sovelluksen kehittämiseen. Ilman vastaavanlaista kehitystyökalua kehittäjä joutuisi manuaalisesti asentaa ja konfiguroida mm. Webpack- sekä Babel-ohjelmistot, jotta lähdekoodi saadaan käännettyä, paketoitua, sekä tiivistettyä muotoon, joka on suorituskyvyllisesti optimaalinen sekä toimii vanhemmilla selaimilla. Create-React-Appissa nämä ovat valmiina, ja tarjolla oleva build-komento yhdistää kaikki staattiset tiedostot yhdeksi optimoiduksi Javascript-tiedostoksi tuotantoversiota varten. (Richey 2017.)

3.4 Victory.js

Victory on datan visualisointia varten luotu komponenttikirjasto Reactille. Tarjolla on valmiita React-komponentteja perinteisestä ympyrädiagrammista viivakaavioihin, joita on mahdollista yksilöidä esimerkiksi haluamillaan väreillä sekä animaatioilla. (Formidable.com 2019.)

Käyttö on hyvin yksinkertaista, sillä Victory-kirjaston verkkosivun dokumentaatiosta näkee kaikki saatavilla olevat komponentit. Haluttu komponentti tuodaan sisään (import) React-komponenttiin, ja sen tulostus tapahtuu lisäämällä komponentti render-funktion sisälle. Data lisätään komponentin parametrien (props) kautta, ja sen tulee koostua olioista muodostuvasta taulukosta. Jokainen olio saa kaksi avain-arvo-paria, jotka määrittelevät sen x- ja y-akselien arvot kaaviossa. (Formidable.com 2019.) Kuvan 5 esimerkissä tulostetaan pylväskaavio neljälle henkilölle: x-akselin arvot ovat henkilönimiä ja y-akselin arvot ovat numeraalisia arvoja. Kun nimeää x-akselin avaimen label-nimiseksi ja antaa sen arvoksi merkkijonon, osaa Victory myös tulostaa annetun merkkijonon selaimeen.

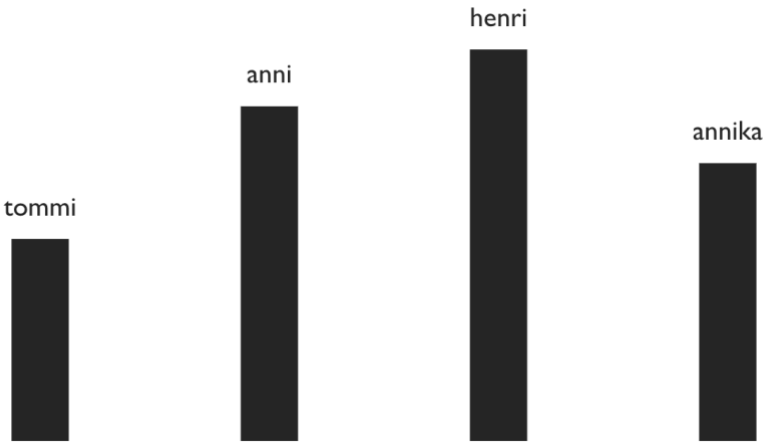
```
import React, { Component } from 'react'
import { VictoryBar } from 'victory'

const data = [
  { label: 'tommi', y: 32 },
  { label: 'anni', y: 53 },
  { label: 'henri', y: 62 },
  { label: 'annika', y: 44 }
]

export class App extends Component {
  render() {
    return (
      <VictoryBar
        data={data}
      />
    )
  }
}
```

Kuva 5. Victory-komponentin muodostaminen.

Kun tarkastellaan, miltä kaavio näyttää selaimessa, Victory on tulostanut SVG-formaatista muodostuvan kuvan, jossa on neljä pylvästä perustuen henkilöihin sekä näiden numeraalisiin arvoihin (Kuva 6).



Kuva 6. Esimerkki Victory-pylväskaaviosta.

4 PALVELINPUOLI

Vaikka tämän opinnäytetyön pääpaino onkin kehitettävän sovelluksen käyttöliittymässä, niin koin tarpeelliseksi myös raportoida oleellisimmat työssä käytettävät palvelinpuolen teknologiat. Tärkeässä roolissa on etenkin sovelluksen rajapinta.

Kehitettävän sovelluksen datan lähteenä toimii yrityksen tarjoama Elasticsearch-hakukone, jonka REST-rajapintaa hyödynnetään sovelluksessa. Tietoturvasyistä ei ollut kuitenkaan suositeltavaa lähettää kyselyitä käyttöliittymän puolelta suoraan Elasticsearchiin, joten väliin luodaan pieni GraphQL-yhdyskäytävä, joka ohjelmoidaan Javascriptilla Node.js-ajoympäristössä. Rajauksen vuoksi en tässä opinnäytetyössä käsittele Elasticsearch-kyselyiden muodostamista, enkä myöskään GraphQL-yhteyden muodostamista Elasticsearchiin.

4.1 Elasticsearch

Elasticsearch on vuonna 2010 julkaistu ilmainen avoimen lähdekoodin hakukoneohjelmisto, joka pohjautuu Apache Luceneen. Data lähetetään Elasticsearchille JSON-muodossa, joka varastoi sen ja lisää hakuviittauksen dokumenttiin klusterin indeksissä. Elasticsearch tarjoaa myös REST-rajapinnan, joka mahdollistaa datan monipuolisen käytön. (aws.amazon.com 2019.)

Itse kyselyt Elasticsearchiin tehdään JSON-syntaksia muistuttavalla kyselykielellä (Elastic.co 2019).

4.2 GraphQL

Graphql on Facebookin kehittämä uusi rajapintastandardi, jonka yksi valttikorteista on perinteisestä REST-metodologiasta poiketen vain yksi rajapinnan päätepiste, johon kutsut lähetetään. GraphQL-kyselykielellä muodostetaan JSON-syntaksia muistuttavia kyselyitä, joilla on mahdollista palauttaa rajapinnasta juuri se data, mitä halutaan. Tällä pyritäänkin paikkaamaan REST:n puutteita, jossa kutsuja lähetetään useaan rajapinnan päätepisteeseen, ja palautettava data on harvoin täysin siinä muodossa, jossa sitä tarvitaan. (howtographql.com 2019.)

Oleellisimpia asioita käytön kannalta ovat skeeman luominen sekä kyselyt. GraphQL määrittelee oman tyyppijärjestelmän, jonka mukaan skeema määritellään (howtographql 2019). Kuvan 7 esimerkissä luodaan ensin henkilötyyppi, jolle lisätään name- sekä age-kentät. Kenttien arvoksi määritellään datan tyypit, jotka tässä tapauksessa ovat merkkijono (String) sekä kokonaisluku (Integer). GraphQL-skeemaan lisätään myös aina juurityyppi eli Query, joka toimii skeeman tulokohtana. Query-tyypin alle määritellään vielä people-kenttä, jonka arvo on Person-kenttä taulukkomuodossa.

```
type Person {
  name: String
  age: Int
}

type Query {
  people: [Person]
}
```

Kuva 7. GraphQL-tyypin määritteleminen.

GraphQL-kyselyiden muodostaminen on myös yksinkertaista: JSON-syntaksia muistuttavan koodirivin avulla voidaan luoda kuvan 8 mukainen kysely, johon on määritelty samat kentät kuin kuvan 7 skeemassa määriteltiin, joten oikeassa sovelluksessa tämä palauttaisi juurikin henkilölistan. Tätä käytäntöä tullaan myös hyödyntämään toteutusosiossa.

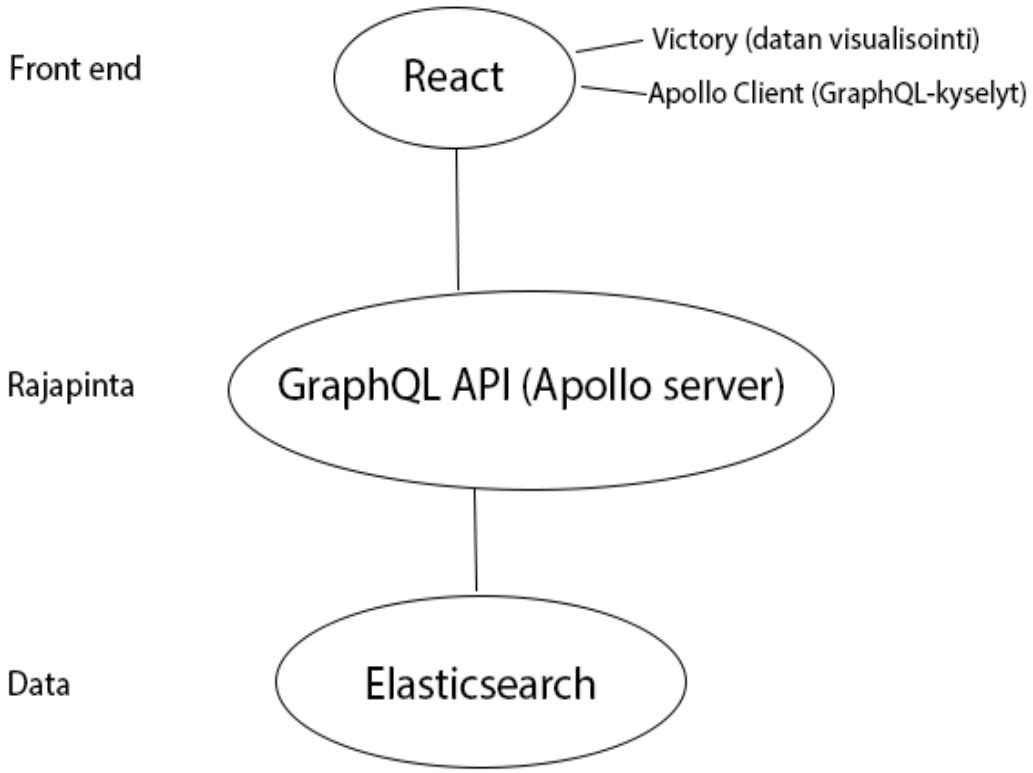
```
{
  people {
    name
    age
  }
}
```

Kuva 8. Yksinkertaisen GraphQL-kyselyn muodostaminen.

5 TOIMEKSIANNON TOTEUTUS

Tässä toteutusosiossa kehitetään siis React-sovellus, jonka runkoon lisätään Victory-kirjasto datan visualisointia varten, sekä Apollo Client rajapintayhteyttä ja GraphQL-kyselyitä varten. Apollo Clientista käsittelen hieman teoriaa luvussa 5.3.5, jossa luodaan GraphQL-kyselyt. Ulkoasun teemoitukseen käytetään jonkin verran CSS:ää. Pääosa työstä käsittelee käyttöliittymän kehitystä, ja tässä työssä raportoidaan sovelluksen etusivun toteutus.

Käyttöliittymän lisäksi välimaastoon luodaan pieni GraphQL-rajapinta, joka on yhteydessä React-käyttöliittymän lisäksi yrityksen tarjoamaan Elasticsearch REST-rajapintaan, josta varsinainen data on saatavilla. Niin kuin luvussa 4 mainitsin, en rajauksen vuoksi raportoi tähän opinnäytetyöhön GraphQL:n sekä Elasticsearchin välistä yhteyttä, vaikka tämän tuli todellisuudessa myös toteutettua kehitystyön aikana. Näin ollen tulen raportoimaan GraphQL-palvelimen pystyttämisen sekä skeeman luomisen. Koko sovelluksen arkkitehtuuri tulee näyttämään kuvan 9 mukaiselta.

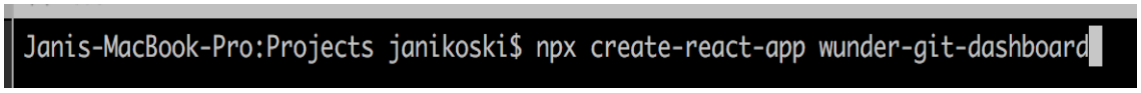


Kuva 9. Wunder Git Dashboard -sovelluksen arkkitehtuuri.

5.1 Alustavat toimenpiteet

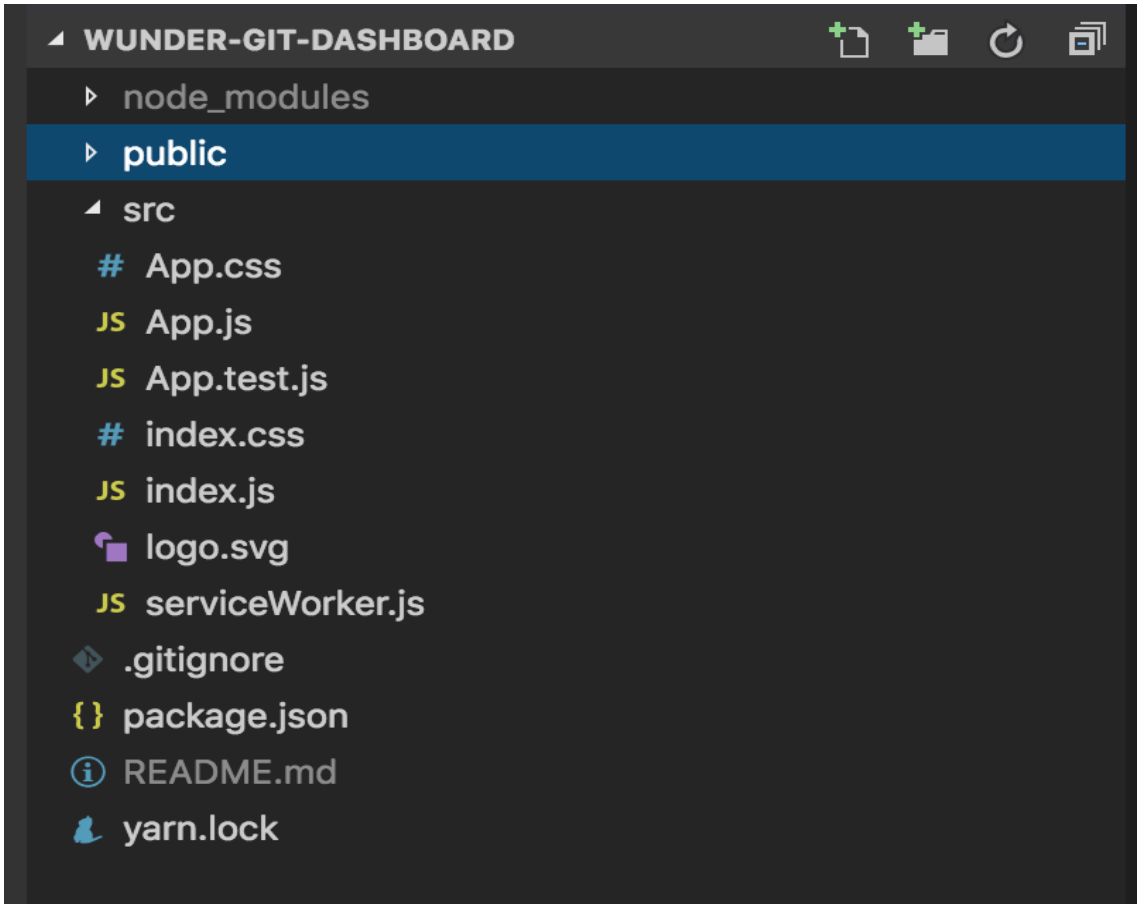
5.1.1 Käyttöliittymän alustus

Toteutus aloitetaan täysin tyhjästä, joten luodaan Create-React-Appin avulla React-kehitysympäristö. Ajetaan komentorivillä komento *npx create-react-app wunder-git-dashboard,* joka luo wunder-git-dashboard-nimisen hakemiston ja asentaa React-kehitysympäristön (Kuva 10).



Kuva 10. React-kehitysympäristön luominen Create-React-Appia käyttäen.

Create-React-App luo oletuksena kuvan 11 mukaisen hakemistorakenteen.



Kuva 11. Create-React-App-hakemistorakenne (Visual Studio Code 2019).

Create-React-Appin avulla sai siis luotua yhdellä komennolla kehitysympäristön, joka myös optimoi staattiset tiedostot automaattisesti niin kuin teoriaosuudessa käsiteltiin. Tämän toteutusosion aikana ei näin ollen käytetä enempää aikaa suorituskyvyn optimointiin.

Asennetaan seuraavaksi käyttöliittymässä hyödynnettävät paketit. Osa saatiin jo valmiina Create-React-Appin mukana, mutta näiden lisäksi asennetaan vielä GraphQL-kyselyitä, Victory-kaavioita sekä React-reititystä varten tarvittavat paketit. Haetaan näin ollen seuraavat paketit Node Package Manager -rekisteristä: apollo-boost, graphql-tag, react-apollo, react-router sekä victory-kirjasto. Asennus tapahtuu kuvan 12 mukaisesti komentoriviltä.

```
npm i apollo-boost graphql-tag react-apollo react-router-dom victory
```

Kuva 12. Käyttöliittymän Node-moduulien asennus.

Asennuskomento luo client-hakemistoon node-modules-nimisen hakemiston, johon paketit on nyt asennettu. Pakettien lisäksi saman hakemiston juuresta löytyy nyt package.json-tiedosto, josta näkee asennetut paketit sekä niiden versioinnin (Kuva 13).

```
"dependencies": {
  "apollo-boost": "^0.3.1",
  "graphql": "^14.1.1",
  "graphql-tag": "^2.10.1",
  "react": "^16.8.6",
  "react-apollo": "^2.5.2",
  "react-dom": "^16.8.6",
  "react-router-dom": "^5.0.0",
  "react-scripts": "3.0.1",
  "victory": "^32.1.0"
},
```

Kuva 13. Package.json, josta näkee asennetut paketit sekä niiden versioinnin.

Tässä raportissa ei käsitellä React-reitittimen käyttöä, mutta Apollo-Boost on tärkeä työn kannalta, sillä se asentaa valmiin Apollo Client -konfiguraation. Apollo Client on GraphQL asiakasohjelma Javascriptille, jonka tehtäviin kuuluvat mm. välimuistitus sekä käyttöliittymän automaattinen päivittäminen. Kun kehittäjän ei tarvitse kuluttaa aikaa

edellä mainittuihin asioihin, voi tämä lähinnä keskittyä olennaiseen eli GraphQL-kyselyiden kirjoittamiseen. (apollographql.com 2019.)

Kehitysympäristö saadaan käyntiin ajamalla komento *npm start* komentorivillä, minkä jälkeen oletusselain avaa osoitteen localhost:3000, jossa sovellus on nyt käynnissä. Sovelluksessa on oletuksena jonkin verran CSS-tyylejä ja HTML-merkkausta, jotka kaikki poistetaan tässä vaiheessa, koska luonnollisesti tilalle lisätään omat.

5.1.2 Rajapinnan alustus

Rajapinnan kehitys aloitetaan muuttamalla jo olemassa olevaa kansiorakennetta. Luodaan jo luotuun wunder-git-dashboard-hakemistoon client-niminen hakemisto, jotta arkkitehtuurissa tulisi eroteltua selvästi käyttöliittymä ja rajapinta toisistaan. Tämän jälkeen kopioidaan leikkaa- ja liitä-toiminnallisuudella Create-React-Appin hakemistorakenne client-hakemistoon. Sovelluksen juurihakemisto eli wunder-git-dashboard on tästä tästä eteenpäin varattu palvelinpuolen (tässä tapauksessa rajapinnan) koodille ja client-hakemisto käyttöliittymän koodille.

GraphQL-Rajapinnalle tarvitaan myös NPM-rekisteristä Node-moduuleja, joten alustetaan nämä luomalla ensin packacke.json-tiedosto. Tämä onnistuu ajamalla komentorivillä *npm init* wunder-git-dashboard-hakemiston juuresta (Kuva 14).

```
npm init
```

Kuva 14. Npm init.

Ladataan seuraavaksi itse moduulit. Tämän raportin kannalta tärkein on Apollo Server, joka on yksinkertaistettuna Javascript-kirjasto GraphQL-rajapintojen luomista varten, ja jonka avulla on mahdollista luoda yhteys käytännössä mihin tahansa datan lähteeseen (Apollographql.com 2019). Kuvasta 15 nähdään moduulien asennuskomento kokonaisuudessaan.

```
Janis-MacBook-Pro:wunder-git-dashboard janikoski$ npm i @elastic/elasticsearch apollo-boost apollo-server
dotenv graphql nodemon apollo-link-context
```

Kuva 15. Rajapintaan liittyvien pakettien asennus.

Asennuskomennon jälkeen NPM luo jälleen node modules -hakemiston sekä package.json-tiedoston, ja tällä kertaa tietysti projektin juurihakemistoon, josta

asennuskomento ajettiin. Rajapintaa varten tarvittavat moduulit on nyt asennettu, ja kehitystyö voidaan aloittaa.

5.2 Rajapinnan toteutus

5.2.1 GraphQL-skeema

Sovelluksessa hyödynnettävän datan rakenne määritellään luomalla projektille skeema, jotta sitä voidaan myöhemmin visualisoida käyttöliittymässä. Tämä onnistuu luomalla projektin juureen schema.js-niminen tiedosto, johon tuodaan gql-funktio Apollo Server -paketista, joka taas mahdollistaa GraphQL-tyyppimäärittelyjen kirjoittamisen (Kuva 16).

Kuva 16. Gql-funktion tuominen skeemaan (Visual Studio Code 2019).

Koska tarkoituksena on visualisoida niin Git commit -viesteihin kuin itse projekteihin liittyvää dataa, luodaan molemmille omat tyypit skeemaan: Commit sekä Repo. Näille lisätään vielä tarvittavat kentät, joita ovat mm. viestin nimi, lähettejä sekä projektin nimi. (Kuva 17.) Keskinäistä suhdetta Commit- sekä Repo-tyypeillä ei ole, ja tässä raportissa keskitytään pääosin Commit-tyyppiin.

```
type Commit {
  author: String
  message: String
  repository: String
  time: String
}

type Repo {
  name: String
  primaryLanguage: String
}
```

Kuva 17. Commit- sekä Repo-tyyppien määrittely.

Käyttöliittymässä olisi tarkoitus voida suodattaa Commit-viestejä joko viimeisen päivän tai viikon ajalta, joten lisätään myös näille omat tyypit skeemaan: CommitsToday ja CommitsThisWeek. Kentät näille ovat commit-kokonaislukumäärä (total) sekä jo luotu Commit-tyyppi taulukkomuodossa, sillä commit-viestejä tulisi listata tietty määrä. (Kuva 18.)

```
type CommitsToday {
  total: Int
  commits: [Commit]
}

type CommitsThisWeek {
  total: Int
  commits: [Commit]
}
```

Kuva 18. CommitsToday- sekä CommitsThisWeek-tyyppien määrittely.

Viimeisenä skeemaan luodaan Query-tyyppi, johon lisätään kentät tämän päivän sekä viikon commiteille, ja näiden arvoksi asetetaan jo luodut CommitsToday- sekä CommitsThisWeek-tyypit. Näin voidaan hyödyntää mahdollisimman tehokkaasti Commit-tyyppiä sekä tämän kenttiä. Viimeinen kenttä eli repos on projektien listausta varten, ja sen arvoksi annetaan Repo-tyyppi taulukkomuodossa. Skeema on nyt valmis. (Kuva 19.) Tässä vaiheessa mainitsen, että tämän opinnäytetyön raportissa en tule kaikkia skeemaan määriteltyjä kenttiä hyödyntämään käyttöliittymän toteutusosiossa, vaikka todellisuudessa kaikkia hyödynnettiin sovelluksessa.

```
const typeDefs = gql`
  type Commit {
    author: String
    message: String
    repository: String
    time: String
  }

  type Repo {
    name: String
    primaryLanguage: String
  }

  type CommitsToday {
    total: Int
    commits: [Commit]
  }

  type CommitsThisWeek {
    total: Int
    commits: [Commit]
  }

  type Query {
    commitsToday: CommitsToday
    commitsThisWeek: CommitsThisWeek
    repos: [Repo]
  }
`
```

Kuva 19. GraphQL-skeema.

5.2.2 GraphQL-palvelin

Projektin juureen luodaan server.js-niminen tiedosto, joka toimii GraphQL-rajapinnan palvelimena. Tähän tiedostoon tuodaan kolme eri asiaa:

- Apollo Server -kirjasto.
- GraphQL-rajapinnan skeema, joka luotiin edellisessä luvussa.
- GraphQL-resolverit, joiden luontia tässä työssä ei käydä läpi. Nämä vastaavat skeeman määrittelyn varsinaisesta toteutuksesta, joten tämä osa sovelluksesta on yhteydessä Elasticsearchiin. Varsinainen data tulee siis tätä kautta.

Luodaan myös ilmentymä ApolloServer-oliosta, johon lisätään listen-metodi. Kun palvelin ajetaan käyntiin, tulostetaan komentoriville englanniksi selkeästi teksti, joka viittaa palvelimen käynnistymiseen.

Kokonaisuudessaan GraphQL-palvelimen koodi nähdään kuvassa 20.

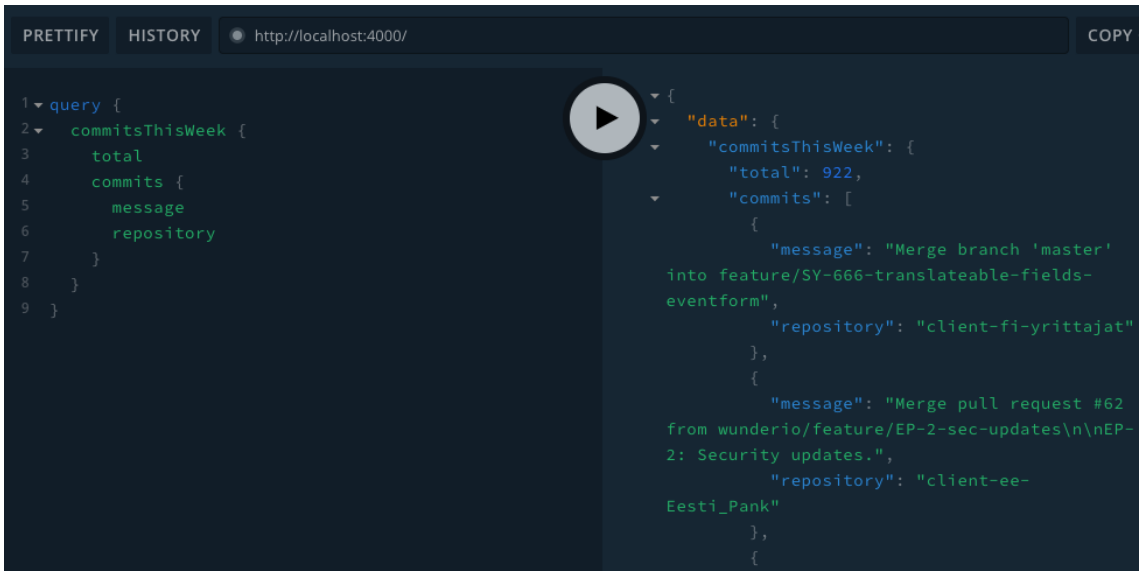
```
const { ApolloServer } = require('apollo-server')
const typeDefs = require('./schema')
const resolvers = require('./resolvers')

const server = new ApolloServer({ typeDefs, resolvers })

server.listen().then(({ url }) => {
  console.log(`🚀  Server ready at ${url}`)
})
```

Kuva 20. GraphQL-palvelimen koodi.

Komentoriviltä voi nyt ajaa *npm run server*, minkä jälkeen GraphQL-palvelin käynnistyy osoitteessa localhost:4000. Kun navigoi selaimessa kyseiseen osoitteeseen, päätyy GraphiQL-työkalun näkymään, jossa voi testata kyselyitä ja tarkastella, millaista dataa ne palauttavat. Vasemmassa sarakkeessa luodaan kysely ja oikeassa nähdään vastaus. Kuvassa 21 testataan commitsThisWeek-kyselyn toimivuutta, ja kuten kuvasta nähdään, oikeanlaista projekteihin liittyvää dataa palautuu selvästi oikeanpuoleiseen sarakkeeseen. Tätä dataa tullaan hyödyntämään käyttöliittymän visualisoinnissa.



Kuva 21. GraphQL-kyselyn testaaminen GraphiQL-työkalussa.

Rajapinta on valmis, joten seuraavassa luvussa voidaan aloittaa käyttöliittymän toteutus.

5.3 Käyttöliittymän toteutus

5.3.1 Ylätunniste

Käyttöliittymän toteutus on helpointa aloittaa yksinkertaisimmasta React-komponentista eli ylätunnisteesta, sillä se ei sisällä juurikaan varsinaista logiikkaa. Luodaan client/src/compoenents-hakemistoon Header-niminen hakemisto, ja tähän hakemistoon Header.js-tiedosto. Kyseiseen tiedostoon lisätään ylätunnisteen koodi.

Koska kyseessä on React-komponentti, ensimmäisellä koodirivillä tuodaan tietysti React itse React-kirjastosta (Kuva 22).

```
import React from 'react'
```

Kuva 22. Reactin import.

Ylätunnisteen React-komponentti tulee palauttamaan vain staattista HTML:ää. Lisätään ylätunnisteeseen Header-tagit, joiden sisään lisätään Wunderin Logo sekä ”Git dashboard" -teksti h1-tagien sisään. Koska komponentti on tilaton (stateless), käytetään funktionaalista syntaksia, joka on yksinkertaisesti funktio, joka palauttaa tarvittavan merkkauksen JSX-muodossa. Viimeisenä funktion eteen lisätään avainsanat export default, jotta sitä voidaan myöhemmin hyödyntää toisessa komponentissa. (Kuva 23.)

```
export default function Header() {
  return (
    <header>
      <div className="logo-wrapper">
        <a href="/" className="logo"></a>
      </div>
      <h1>Git Dashboard</h1>
    </header>
  )
}
```

Kuva 23. Header React-komponentti, jonka syntaksi on funktionaalinen.

Viimeistellään komponentti lisäämällä sille hieman tyylejä. Create-React-Appin hakemistorakenteesta löytyy App.css-tiedosto, johon tyylit on oletuksena lisätty. Poistan kaikki oletustyylit ja luon tilalle omat.

Header-elementti saa taustavärikseen harmaan. Lisäksi keskitetään ns. lapsielementit flex- ja center-arvoja käyttäen. (Kuva 24.)

```
header {
  display: flex;
  align-items: center;
  background-color: grey;
}
```

Kuva 24. Header-komponentin tyylit.

Ylätunnisteessa oleva h1-teksti muutetaan vielä valkoiseksi, jotta se erottuisi selkeämmin harmaasta taustasta. Lisäksi logo lisätään paikalleen käyttäen CSS:n background-ominaisuuksia, ja sille määritellään sopivat leveys- sekä korkeussuhdanteet pikseleinä. (Kuva 25.)

```
h1 {
  color: white;
}

.logo {
  display: block;
  width: 200px;
  height: 70px;
  background-image: url('./wunder-logo.svg');
  background-repeat: no-repeat;
}
```

Kuva 25. h1- sekä logoelementtien tyylit.

Ylätunniste on nyt valmis (Kuva 26).

Kuva 26. Valmiin ylätunnisteen ulkoasu.

5.3.2 Pylväskaavio

Pylväskaaviota varten luodaan varten Charts-niminen hakemisto components-hakemiston alle, jolla pyritään viittaamaan selkeästi, että kyseinen hakemisto on kaaviokomponentteja varten. Luotuun hakemistoon tehdään vielä ActiveProjects.js-tiedosto, joka tulee pitämään sisällään pylväskaavion koodin. ActiveProjects-nimellä pyritään kertomaan, että kyseisen komponentin tarkoitus on esittää commit-määriin perustuen yrityksen aktiivisimpia projekteja joko viimeisimmän päivän tai viikon mukaan.

Komponentin luominen aloitetaan perinteiseen tapaan tuomalla ensin React jo luotuun tiedostoon. Tuodaan myös Victory-kirjastosta VictoryBar-, VictoryChart- sekä VictoryAxis-komponentit, joita kaikkia tullaan hyödyntämään pylväskaavion toteutuksessa. (Kuva 27.)

```
import React, { Component } from 'react'
import { VictoryBar, VictoryChart, VictoryAxis } from 'victory'
```

Kuva 27. ActiveProjects.js-tiedostoon tuodaan itse React sekä tarvittavat Victory-komponentit.

Tehdään ActiveProjects.js:stä luokkapohjainen komponentti, joka palauttaa aluksi vain hieman JSX-merkkausta render-metodin kautta. Lisätään siis div-elementti ja tarvittavat luokat tyylittelyä varten, sekä h2-elementti komponentin otsikointia varten. (Kuva 28.)

```
class ActiveProjects extends Component {
  render() {
    return (
      <div className="home__active-projects active-projects">
        <h2>Activity by project</h2>
      </div>
    )
  }
}
```

Kuva 28. Yksinkertainen luokkapohjainen React-komponentti, joka toimii pohjana pylväskaaviolle.

Kun pohja komponentille on luotu, lisätään merkkaukseen mukaan aiemmin tuodut VictoryChart-, VictoryBar- sekä VictoryAxis-elementit, jotka muodostavat pylväskaavion rungon (Kuva 29).

```
<VictoryChart>
  <VictoryBar />
  <VictoryAxis />
  <VictoryAxis />
</VictoryChart>
```

Kuva 29. Pylväskaavion merkkaus ilman ominaisuuksia.

Määritellään seuraavaksi komponentin ulkoasu lisäämällä tälle hieman tyylejä sekä pieni animaatio. Uloimman komponentin eli VictoryChartin parametreissä voidaan hyödyntää Victory-kirjastosta saatuja domainpadding- ja animate-parametrejä, joista domainpadding määrittelee pylväiden välin pikseleinä ja animate animaation pituuden millisekunteina. Lisätään siis pylväiden väliksi viisitoista pikseliä ja animaation pituudeksi kaksituhatta millisekuntia. (Kuva 30.)

```
<VictoryChart
  domainPadding={15}
  animate={{
    duration: 2000,
  }}
>
```

Kuva 30. VictoryChart-komponentin parametrit, joissa määritellään pylväiden väli sekä animaation pituus.

Aiemmin luodut kaksi VictoryAxis-komponenttia mahdollistavat akselien tarkemman määrittelyn. Otsikot kaavion X- ja Y-akseleille tulisi nimetä Project- sekä Commits-nimisiksi ja määritellä, kumpaa käytetään Y-akselilla. Lisätään otsikot label-parametrin kautta, sekä määritellään commit-akselista Y-akseli lisäämällä tälle ylimääräinen parametri nimeltä dependentAxis (Kuva31).

```
<VictoryAxis
  label="Project"
/>

<VictoryAxis
  dependentAxis
  label="Commits"
/>
```

Kuva 31. Akseleiden määrittely Victory-komponentissa.

Pylväskomponentin eli VictoryBarin parametrit määritellään viimeisenä. Tälle lisätään kaksi parametriä: style sekä data, joista ensimmäistä käytetään nimensä mukaisesti tyylittelyyn ja jälkimmäiseen lisätään visualisoitava data. Oliomaista rakennetta käyttäen voidaan lisätä tarvittavat tyylimääritellyt style-parametrin sisään, ja tässä tapauksessa vaihdetaan pylväiden taustaväri harmaaksi ja otsikoiden fonttikooksi kymmenen pikseliä. Tärkein osuus tapahtuu kuitenkin data-parametrin kautta, johon lisätään itse data, jota ei siis vielä tässä vaiheessa ole olemassa. Data tullaan saamaan myöhemmin activeProjects-komponentin parametrien kautta, joten sen arvoksi voidaan antaa jo tässä vaiheessa this.props.activeProjects. Kyseinen muuttuja tulee pitämään sisällään myöhemmin datan rakenteen. Kuvasta 32 nähdään VictoryBar-komponentti kokonaisuudessaan.

```
<VictoryBar
  style={{ data: { fill: "grey" }, labels: {fontSize: 10} }}
  data={this.props.activeProjects}
/>
```

Kuva 32. VictoryBar-komponentin parametrit, johon myös itse data lisätään.

Pylväskaavion ulkoasu on tässä vaiheessa valmis, mutta dataa ei vielä ole määritelty. Itse komponenttia myöskään ei ole renderöity, joten selaimen puolella ei voi tarkastella, miltä se näyttää. Nämä vaiheet toteutetaan myöhemmin etusivu-komponenttia käsittelevässä luvussa.

5.3.3 CommitCount-komponentti

Luodaan CommitCount-niminen komponentti, joka tulee esittämään committien lukumäärän, ja tätä lukumäärää käyttäjä voi myös suodattaa käyttöliittymän puolella. Components-hakemiston juureen lisätään näin ollen uusi tiedosto nimeltä CommitCount.js, johon luodaan luokkapohjainen React-komponentti. Tämä Komponentti palauttaa parametrien kautta h2-elementin, jonka teksti tulee vaihtumaan riippuen käyttäjän valinnasta. Sama tehdään myös committien lukumäärälle. Lisätään vielä lopuksi muutama luokka, jotta komponenttia on mahdollista teemoittaa. Kokonaisuudessaan edeltävät toimenpiteet nähdään kuvasta 33.

```
import React, { Component } from 'react'

class CommitCount extends Component {
  render() {
    return (
      <div className="home__commit-count commit-count">
        <h2>{this.props.commitText}</h2>
        <div className="commit-count__count">{this.props.commitCount}</div>
      </div>
    )
  }
}
```

Kuva 33. CommitCount-komponentti.

Seuraavassa luvussa luodaan etusivu-komponentti, jonka kautta renderöidään selaimeen jo aikaisemmin luotu pylväskaavio (ActiveProjects) sekä nyt luotu CommitCount-komponentti. Näiden lisäksi luodaan dataa käsittelevä logiikka, jotta sitä on mahdollista hyödyntää jo luoduissa elementeissä.

5.3.4 Etusivu-komponentin alustus

Etusivulle olisi tarkoitus lisätä kahdessa edellisessä luvussa luodut komponentit, sekä kaksi eri painiketta, joita käyttämällä käyttäjä voi suodattaa dataa joko päivän tai viikon mukaan.

Aloitetaan luomalla sivukohtaisia React-komponentteja varten Page-niminen hakemisto components-hakemiston alle, ja luodaan tähän hakemistoon Home.js-niminen tiedosto viitaten etusivuun. Luodaan myös Home.js:stä luokkapohjainen React-komponentti, ja määritellään tälle ensimmäisenä merkkaus. Etusivun ylälaitaan lisätään h2-teksti, joka viittaa commit-aktiivisuuteen. Lisäksi lisätään painike-elementit, joihin myöhemmin lisätään tarvittava Javascript-logiikka suodattamista varten. Viimeisenä lisätään kaikille elementeille luokat teemoitusta varten. (Kuva 34.)

```
<h2 className="page-title">Commit activity</h2>
<div className="datesort">
  <span className="datesort__label">Sort by:</span>
  <button className="datesort__day">Day</button>
  <button className="datesort__week">Week</button>
</div>
```

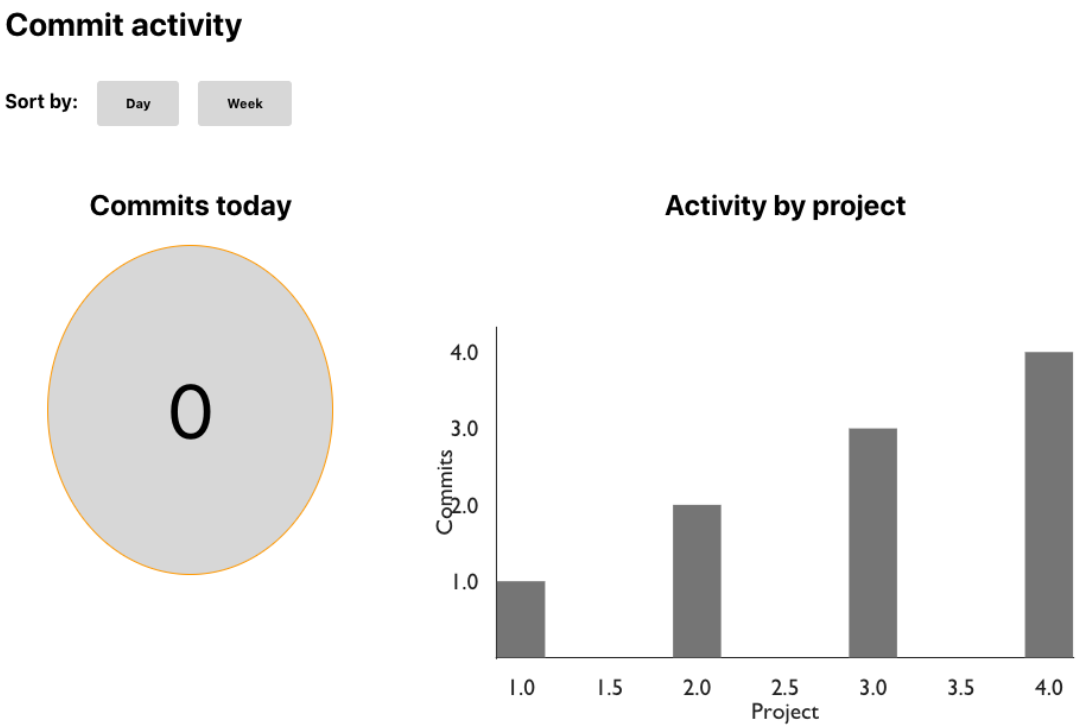
Kuva 34. Etusivun ylälaidan elementit.

Tämän jälkeen voidaan lisätä paikalleen jo aiemmin luodut CommitCount- sekä ActiveProjects-komponentit. Lisätään myös jo tässä vaiheessa molempien komponenttien parametrien arvot. Commit-teksti, lukumäärä sekä aktiiviset projektit tullaan kaikki varastoimaan etusivu-komponentin tilassa, joten ne voidaan lisätä komponentteihin näiden parametrien kautta. (Kuva 35).

```
<CommitCount commitText={this.state.commitText} commitCount={this.state.commitCount}/>
<ActiveProjects activeProjects={this.state.activeProjects}/>
```

Kuva 35. CommitCount- ja ActiveProjects-komponenttien renderöinti, sekä näiden parametrit.

Pienen tyylittelyn (CSS) jälkeen voidaan luoduille komponenteille lisätä testidataa, ja kuvasta 36 tarkastella, miltä etusivu näyttää selaimessa.



Kuva 36. Etusivun ulkoasu ilman oikeaa dataa.

Kuten kuvasta 36 nähdään, ulkoasu näyttää hyvältä, mutta oikea data puuttuu vielä, ja se tullaankin saamaan GraphQL-kyselyiden avulla.

5.3.5 GraphQL-kyselyiden muodostaminen

Tuodaan Home.js tiedostoon Reactin lisäksi

- Apollo Client, jolla määritellään GraphQL-rajapinnan osoite
- jo luotu ActiveProjects-komponentti
- Gql-funktio GraphQL-kyselyitä varten.

Nämä nähdään tarkemmin kuvasta 37.

```
import React, { Component } from 'react'
import ApolloClient from 'apollo-boost'
import ActiveProjects from '../Charts/ActiveProjects'
import gql from 'graphql-tag';

const client = new ApolloClient({
  uri: "http://localhost:4000/",
});
```

Kuva 37. Home.js-komponentin importit, sekä GraphQL-rajapinnan osoitteen määrittely.

Luodaan seuraavaksi GraphQL-kyselyt viimeisimmän päivän sekä viikon commiteille, joiden rakenne määriteltiin aiemmin GraphQL-rajapinnan skeemassa. Lisätään molemmille omat muuttujat: commitsToday ja commitsThisWeek. Edellä mainittuihin muuttujiin lisätään itse kyselyt gql-metodin avulla, joista saadut vastaukset ovat commit-viestien kokonaismäärä sekä projektin repository. Molempia vastauksia tullaan hyödyntämään myöhemmin sovelluksessa. Kokonaisuudessaan kyselyt näyttävät kuten alla olevassa kuvassa 38.

```
const commitsToday = gql`
  query {
    commitsToday {
      total
      commits {
        repository
      }
    }
  }
`

const commitsThisWeek = gql`
  query {
    commitsThisWeek {
      total
      commits {
        repository
      }
    }
  }
`
```

Kuva 38. GraphQL-kyselyt tämän päivän sekä viikon commiteille.

5.3.6 Etusivu-komponentin tila

Kun käyttäjä suodattaa etusivulla committeja joko viimeisimmän päivän tai viikon mukaan, kolmen asian pitäisi etusivulla kuvastaa tätä valintaa selkeästi:

- CommitCount-komponentista tulevassa otsikossa tulee lukea joko “Commits Today” tai “Commits this week”.
- CommitCount-komponentista tulevan committien kokonaislukumäärän tulee vaihtua käyttäjän valinnan mukaan.
- Pylväskaavion tulosten tulee vaihtua käyttäjän valinnan mukaan.

Edellä mainitut asiat voidaan määritellä sovelluksen tilaan (state), joka on yksi keskeisempiä React-käsitteitä. Tärkeintä on tässä vaiheessa ymmärtää, että state on yksinkertaisesti Javascript olio, jonka avaimilla pyritään kuvastamaan komponentin tilaa. Kun minkä tahansa avaimen arvo muuttuu setState-metodin toimesta, React renderöi komponentin uudestaan. (Reactjs.org 2019.) Luodaan tässä vaiheessa komponentin tilan alustavat arvot, joita tullaan myös myöhemmin muuttamaan tiettyjen ehtojen täyttyessä.

Aloitetaan luomalla luokkapohjainen komponentti etusivulle, ja määritellään tälle alustava tila sekä arvot. Tila tulee muuttumaan riippuen siitä, suodattaako käyttäjä dataa päivän vai viikon perusteella. Tilassa käytettävät avaimet ovat:

- commitText, jonka alustava arvo on ’Commits today’ viitaten tämän päivän committeihin, mutta se voi muuttua käyttäjän valinnan mukaan.

- commitCount, johon tullaan lisäämään commit-viestien kokonaismäärä, ja jonka arvo voi vaihtua päivän tai viikon aikavälin mukaan. Alustava arvo on tässä vaiheessa tyhjä merkkijono, mutta myöhemmin se saadaan GraphQL-kyselystä.

- activeProjects, joka tulee pitämään sisällään repository GraphQL-kyselyn arvon. Tämä käsitellään myöhemmin, mutta alustava arvo on null eli ei tiedossa.

Kuvasta 39 nähdään, kuinka React-komponentin tila määritellään olio-syntaksin mukaisesti this.state-määritelmän alle.

```
this.state = {
  commitText: 'Commits today',
  commitCount: '',
  ActiveProjects: null
}
```

Kuva 39. Tilan määritteleminen React-komponentissa.

5.3.7 Päivä- sekä viikko-ohjelmien luominen

Seuraavaksi luodaan ohjelmat, jotka ajetaan riippuen siitä, haluaako käyttäjä suodattaa dataa etusivulla viimeisimmän päivän tai viikon mukaan.

Aloitetaan päiväohjelman luomisella, eli luodaan commitsToday-niminen Javascript-ohjelma, joka tulee pitämään sisällään tarvittavan logiikan. Ohjelman tulee palauttaa commit-dataa JSON-muodossa, joten nyt on aika hyödyntää aiemmin luotuja GraphQL-kyselyitä. Koska rajapinnan url määriteltiin jo aiemmin client-muuttujaan Apollo-kirjaston kautta, voidaan nyt luoda client.query-metodi, jonka agrumentin arvoksi lisätään haluttu GraphQL-kysely. Tässä tapauksessa haluan käyttää jo luodun commitsToday-kyselyn muuttujaa, joten koodirivi kokonaisuudessaan näyttää kuten kuvassa 40.

```
client.query({ query: commitsToday })
```

Kuva 40. Query-metodin käyttö.

Koska client.query-metodi palauttaa Javascript-lupauksen (Promise), Javascriptin then-metodilla voidaan määritellä mitä tehdä, kun palvelin palauttaa datan. Ensimmäiseksi tutkitaan miltä vastaus näyttää, ja millaisessa muodossa palautettu data tulee. Then-metodin res-parametri sisältää GraphQL-kysylystä saadun tietosisällön, joten console.log-metodin avulla voidaan tulostaa tietosisältö selaimen konsoliin (Kuva 41).

```
client.query({ query: commitsThisWeek })
  .then(res => {
    console.log(res)
```

Kuva 41. GraphQL-kyselyn tietosisällön tulostaminen.

Kun sovellus avataan selaimessa, kehitystyökalujen alta voidaan tutkia, miltä vastaus näyttää (Kuva 42).

```
▼ Object i
  ▼ commitsToday:
    ▼ commits: Array(25)
      ▶ 0: {repository: "client-fi-novita-novitaknits", __typename: "Commit"}
      ▶ 1: {repository: "client-fi-tre3-science-gatsby", __typename: "Commit"}
      ▶ 2: {repository: "client-fi-tre3-science-gatsby", __typename: "Commit"}
      ▶ 3: {repository: "client-fi-tre3-science-gatsby", __typename: "Commit"}
      ▶ 4: {repository: "client-fi-tre3-science-backend", __typename: "Commit"}
      ▶ 5: {repository: "client-fi-tre3-science-backend", __typename: "Commit"}
      ▶ 6: {repository: "client-fi-tre3-science-backend", __typename: "Commit"}
```

Kuva 42. CommitsToday-kyselyn tietosisältö tulostettuna selaimen konsoliin (Google Chrome 2019).

Kuten kuvasta 42 nähdään, vastauksena saadaan CommitsToday-olio, joka sisältää taulukkomuodossa kaikki commitit olioina. Toisin sanoen vastaus näyttää lähes identtiseltä kuin varsinainen GraphQL-vastaus. Saatu vastaus ei kuitenkaan ole sellaisenaan hyödynnettävissä Victory-kaaviossa, joten sitä tullaan käsittelemään seuraavassa luvussa.

5.3.8 Datan pilkkominen ja etusivun viimeistely

Teoriaosuudessa Victory-kirjastoa käsittelevässä luvussa käsiteltiin, kuinka datan täytyy olla olioista koostuva taulukko, jonka avain-arvo-parit muodostuvat x- sekä y-akselien määrittelyistä. Muuten Victory ei osaa piirtää kaaviota oikein. Edellisessä luvussa saatua GraphQL-vastausta joudutaan siis pilkkomaan, jotta se saadaan haluttuun muotoon ja on siten hyödynnettävissä Victory-kaaviossa.

Lisätään commiteille sekä committien lukumäärälle omat muuttujat, jotta näitä voidaan pilkkoa helpommin (Kuva 43).

```
let commits = res.data.commitsToday.commits;
let commitCount = res.data.commitsToday.total;
```

Kuva 43. GraphQL-kyselystä saatu tietosisältö lisättynä muuttujiin.

CommitCount-muuttujaan ei tässä vaiheessa kosketa, mutta pilkotaan commits-muuttujan dataa, joka on olioista muodostuva taulukko, joka listaa viimeisimmässä 25:ssa commitissa käytetyn repositoryn. Ensimmäisenä pitäisi saada poimittua jokaisen repositoryn ilmentymä mahdollisimman helposti. Javascriptin reduce-metodin avulla voidaan käydä läpi jokainen taulukosta löytyvä olio ja palauttaa saadulla tiedolla käytännössä mitä tahansa muuta. Tässä tapauksessa palautetaan olio, jonka avain-arvo-parit muodostuvat itse repositorysta sekä repositoryn ilmentymien lukumäärästä.

```
const repoOccurances = commits.reduce(function(obj, v) {
  obj[v.repository] = (obj[v.repository] || 0) + 1;
  return obj;
}, {})
```

Kuva 44. Reduce-metodin-avulla voidaan taulukosta palauttaa tässä tapauksessa olio.

Data on nyt oliomuodossa repoOccurances-muuttujassa, mutta sitä on pilkottava lisää, jotta se saadaan haluttuun muotoon eli olioista muodostuvaan taulukkoon.

Käytetään Object.keys-metodia, jonka avulla saadaan ensin repoOccurances-oliosta jokainen avain taulukkoon. Tämän jälkeen map-metodin avulla iteroidaan eli käydään läpi repositoryt sekä näiden avaimet. Lopputuloksena palautetaan oliosta koostuva taulukko, jonka avain-arvo-parit muodostuvat repositoryn ilmentymien lukumäärästä y-avaimessa sekä itse repositorysta label-avaimessa. (Kuva 45.)

```
const repos = Object.keys(repoOccurances).map(function(key) {
  return {
    y: repoOccurances[key],
    label: key
  }
})
```

Kuva 45. Object.keys- sekä map-metodien avulla saadaan palautettua data olioista muodostuvasta taulukossa.

Data on käytännössä nyt repos-muuttujassa jo siinä muodossa, että sitä voidaan hyödyntää kaaviossa. Rajataan kuitenkin vielä tulokset kymmeneen, sekä järjestetään ne laskevaan järjestykseen.

Luodaan Javascriptin sort-metodin avulla takaisinkutsufunktio (callback), joka palauttaa datan olioiden y-arvon mukaan, laskevassa järjestyksessä. Lisätään tämä ohjelma kuvan 46 mukaisesti reposSorted-muuttujaan.

```
const reposSorted = repos.sort(function(a, b) {
  return b.y- a.y;
})
```

Kuva 46. Olioista koostuva taulukko muodostetaan laskevaan järjestykseen sort-metodin avulla.

Viimeistellään data rajaamalla se kymmeneen slice-metodin avulla, jonka jälkeen se voidaan lisätä data-muuttujaan (Kuva 47).

```
const data = reposSorted.slice(0, 10)
```

Kuva 47. Data rajattuna kymmeneen tulokseen slice-metodin avulla.

Kun selaimen konsoliin tulostetaan data-muuttuja, nähdään että taulukko listaa kymmenen oliota laskevassa järjestyksessä perustuen y-avaimen arvoon (Kuva 48).

Kuva 48. Data viimeistellyssä muodossa eli olioista muodostuvasta kaaviosta laskevassa järjestyksessä (Google Chrome 2019).

Data on nyt pilkottu tarvittavaan muotoon, ja koska data-muuttuja lisättiin pylväskaaviokomponenttiin jo luvussa 5.3.2, Victory-osaa piirtää sen selaimeen oikein. Ohjelma pitää kuitenkin vielä viimeistellä määrittelemällä logiikka komponentin tilan vaihtumiselle sekä tietenkin ohjelman ajolle.

Kaikki kolme etusivu-komponentin alustavassa tilassa olevaa avainta pitäisi päivittää riippuen siitä, valitseeko käyttäjä päivä- vai viikkosuodatuksen. Lisätään tässä vaiheessa Reactin setState-metodi, jotta tilan alustavia arvoja voidaan päivittää. Metodiin lisätään samat avaimet, jotka määriteltiin aiemmin komponentin alustavaan tilaan. Asetetaan commitText:in arvoksi ’Commits Today’, sillä kyseessä on päiväohjelma. CommitCount-avaimen arvoksi asetetaan commitCount-muuttuja, jonka arvo saatiin GraphQL-kyselystä. Viimeisenä asetetaan activeProjects-avaimen arvoksi data-muuttuja, jonka arvo on juuri pilkotun datan tulos. Kuvassa 49 nähdään edellä mainitut määrittelyt kokonaisuudessaan.

```
this.setState({
  commitText: 'Commits today',
  commitCount: commitCount,
  activeProjects: data
})
```

Kuva 49. Etusivu-komponentin tilan määritteleminen setState-metodin avulla.

CommitsToday-ohjelma on nyt valmis, ja kokonaisuudessaan se näyttää kuten kuvassa 50.

```
commitsToday() {
  client.query({ query: commitsToday })
    .then(res => {
      let commits = res.data.commitsToday.commits;
      let commitCount = res.data.commitsToday.total;

      const repoOccurances = commits.reduce(function(obj, v) {
        obj[v.repository] = (obj[v.repository] || 0) + 1;
        return obj;
      }, {})

       const repos = Object.keys(repoOccurances).map(function(key) {
        return {
          y: repoOccurances[key],
          label: key
        }
      })

      const reposSorted = repos.sort(function(a, b) {
        return b.y- a.y;
      })

      const data = reposSorted.slice(0, 10)

      this.setState({
        commitText: 'Commits today',
        commitCount: commitCount,
        activeProjects: data
      })
  })
}
```

Kuva 50. commitsToday-ohjelma kokonaisuudessaan.

Viikon commiteille riittää, että luo käytännössä täysin identtisen ohjelman, mutta nimeää sen commitsThisWeek-nimiseksi. Query-metodin query-avaimen arvoksi vaihdetaan commitsThisWeek, joka hyödyntää kyseistä GraphQL-kyselyä. Viimeisenä vaihdetaan setState-metodissa commitText-avaimen arvoksi ’Commits this week’.

CommitsToday- sekä CommitsThisWeek-ohjelmat ovat nyt valmiit, ja viimeinen tehtävä on ajaa ne tiettyjen ehtojen täyttyessä. CommitsToday tulisi ajaa, kun käyttäjä lataa sivun tai painaa day-painiketta etusivulla. CommitsThisWeek ajetaan, kun käyttäjä painaa week-painiketta. Reactin componentDidMount-metodissa voidaan yksinkertasesti kutsua commitsToday-ohjelma, joten se ajetaan automaattisesti sivun latauksen yhteydessä (Kuva 51).

```
componentDidMount() {
  this.commitsToday()
}
```

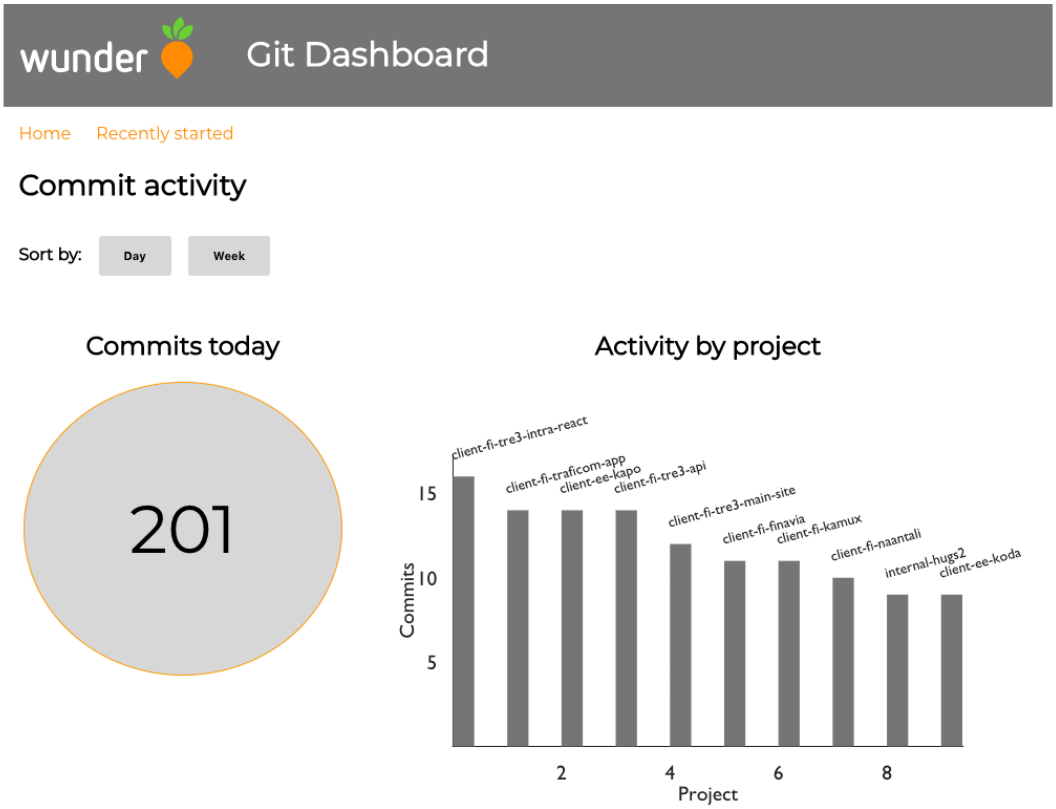
Kuva 51. componentDidMount-metodin käyttö.

Lisätään vielä molempien ohjelmien kutsut näille tehtyihin painikkeisiin, jolloin valittu ohjelma ajetaan vain, kun käyttäjä painaa valittua painiketta. Tämä onnistuu lisäämällä painikkeiden attribuuteiksi onClick-metodit, joiden arvoksi määritellään ohjelmien kutsut (Kuva 52).

```
<button onClick={this.commitsToday} className="datesort__day">Day</button>
<button onClick={this.commitsThisWeek} className="datesort__week">Week</button>
```

Kuva 52. Painikkeisiin lisätään onClick-metodit, joiden arvoksi asetetaan sekä päivä- että viikko-ohjelmat.

Selaimesta voidaan nyt katsoa, miltä etusivu näyttää oikean datan kanssa (Kuva 53). Kuvassa näkyy myös navigaatio, jota en tässä raportissa käsitellyt. Etusivu näyttää oletuksena viimeisimmän päivän commitit, mutta painikkeista voi niitä suodattaa myös viikon mukaan, juuri niin kuin oli tarkoitus.



Kuva 53. Valmis etusivu.

6 YHTEENVETO

Opinnäytetyön tavoitteena oli luoda React-pohjainen verkkosovellus, joka visualisoi toimeksiantajan ohjelmistoprojektien Git-versionhallinnan dataa. Samalla oli tarkoitus pohtia, kuinka työssä käytettävät teknologiat soveltuvat toimeksiannon mukaisen sovelluksen kehitykseen. Työn tarkoituksena oli myös tutkia, kuinka verkkosovellusten käyttöliittymien suorituskykyä voidaan optimoida.

Muulta osin sain käytännössä vapaat kädet toteuttamiseen, esimerkiksi sovelluksen ulkoasun teemoittamisessa ja tarjolla olevan datan visualisoinnissa. Varsinaista aikarajaa ei myöskään annettu, mutta tärkeää oman aikataulun osalta oli, että sovellus valmistui hyvissä ajoin kesän aikana, jotta pystyin keskittymään raportin kirjoittamiseen.

Kehitettävälle sovellukselle luotiin täysin toimiva pohja. Teoriaosuudessa selvitettiin, että suorituskyvyn optimoimiseksi on suositeltavaa yhdistää staattiset tiedostot yhdeksi tiivistetyksi tiedostoksi. Suorituskyvyn optimointiin ei kehitystyössä kuitenkaan varsinaisesti käytetty aikaa, sillä React-käyttöliittymä alustettiin hyödyntäen Create-React-App-kehitystyökalua, joka optimoi staattiset tiedostot automaattisesti. Kyseinen kehitystyökalu sopi hyvin etenkin pienen skaalan ohjelmistoprojektia varten, sillä tärkeää oli saada kehitystyö aluilleen mahdollisimman nopeasti.

Ohjelmistoprojektien data oli saatavilla yrityksen Elasticsearch-hakukoneesta. Dataa varten luotiin kevyt rajapinta hyödyntäen GraphQL-rajapintastandardia, sillä tietoturvasyistä johtuen ei ollut suositeltavaa tehdä kyselyitä suoraan React-käyttöliittymästä Elasticsearchiin. Jatkoa varten rajapinta voidaan tarvittaessa yhdistää myös muihin datan lähteisiin. Opinnäytetyön aiheen rajauksen vuoksi en tähän työhön raportoinut Elasticsearchin ja GraphQL:n välisen yhteyden muodostamista.

Käyttöliittymän runkona toimi siis React Javascript-kirjasto, joka soveltui toimeksiannon mukaiseen tehtävään ongelmitta. Reactin komponenttimaisen luonteen vuoksi oli etusivulle luodut elementit helppo erotella toisistaan sovelluksen arkkitehtuurissa. Lisäksi Reactin JSX-syntaksi nopeutti huomattavasti käyttöliittymään renderöitävän osion määrittelyssä.

Datan visualisointia varten Reactin runkoon lisättiin Victory-kirjasto, jonka kaaviokomponentit soveltuivat hyvin projektin tarpeisiin, sillä kaavioiden ulkoasuun ja animaatioihin ei liittynyt tarkkoja vaatimuksia. Victory-kirjaston käyttö oli myös hyvin

yksinkertaista: käytännössä kirjasto tarvitsi vain asentaa, jonka jälkeen haluamansa kaaviokomponentin sai tuotua Reactin runkoon import-komennon avulla. Saatavilla olevaa dataa joutui kuitenkin pilkkomaan melkoisesti, jotta sitä oli mahdollista visualisoida tehokkaasti. Tästä syystä käyttöliittymä jäi melko kevyeksi. Käyttöliittymän ulkoasu viimeisteltiin CSS:llä.

Sovelluksen toteuttamiseen meni noin kaksi kuukautta. Projektin alussa osaaminen käytetyistä teknologioista oli hyvin rajallinen, joten näihin perehtyminen vei useamman viikon ennen kuin varsinaista kehitystyötä oli mahdollista aloittaa.

Lähdekoodi lisättiin saataville yrityksen GitHub-ohjelmistovarastoon mahdollisia jatkokehitysideoita varten.

LÄHTEET

Apollographql.com 2019. Introduction, What is Apollo Client? Viitattu 16.9.2019 https://www.apollographql.com/docs/react/.

Apollographql.com 2019. Introduction, what is Apollo Server and what does it do? Viitattu 16.9.2019 https://www.apollographql.com/docs/apollo-server/.

Atlassian.com 2019. What is Git. Viitattu 21.9.2019 https://www.atlassian.com/git/tutorials/what-is-git.

Aws.amazon.com. 2019 Elasticsearch. Viitattu 5.6.2019 https://aws.amazon.com/elasticsearch-service/what-is-elasticsearch/.

Banks, A & Porcello, E. 2017. Learning React. Sebastopol, California: O'Reilly Media.

Banks, A & Porcello, E. 2017. Learning React. Sebastopol, California: O'Reilly Media.

Dorcey, T. 2014. Web Page Size, Speed and Performance. Sebastopol, California: O'Reilly Media.

Elastic.co 2019. Introducing the Query Language. Viitattu 5.6.2019 https://www.elastic.co/guide/en/elasticsearch/reference/6.1/_introducing_the_query_language.html.

Formidable.com 2019. Getting started with Victory. Viitattu 2.7.2019 https://formidable.com/open-source/victory/docs/.

Git-scm.com 2019. Viitattu 21.9.2019. https://git-scm.com/docs/git-commit.

Gouirhate, N. 2017. Build a simple chat app with node.js and socket.io. Viitattu 15.8.2019 https://medium.com/@noufel.gouirhate/build-a-simple-chat-app-with-node-js-and-socket-io-ea716c093088.

Grigorik, I. 2019. Image Optimization. Viitattu 26.5.2019 https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/image-optimization.

Guides.github.com 2019. Viitattu 21.5.2019. What is Github? https://guides.github.com/activities/hello-world/.

Howtographql.com 2019. Basic tutorial - introduction. Viitattu 2.7.2019 https://www.howtographql.com/basics/0-introduction/.

Howtographql.com 2019. Core concepts. Viitattu 2.7.2019 https://www.howtographql.com/basics/2-core-concepts/.

Huttunen, K. 2018. Kuvan pienentäminen netistä löytyvän kuvankäsittelyohjelman avulla. Viitattu 26.5.2019 https://www.zoner.fi/kuvan-pienentaminen/.

Kagga, J. 2018. Understanding React Components. Viitattu 15.5.2019 https://medium.com/the-andela-way/understanding-react-components-37f841c1f3bb.

Kostrzewa, D. 2018. Is React the best Javascript framework in 2019? Viitattu 18.8.2019 https://hackernoon.com/is-react-js-the-best-javascript-framework-in-2018-264a0eb373c8.

Minnick, C. 2016. The Real Benefits of the Virtual DOM in React.js. Viitattu 26.5.2019 https://www.accelebrate.com/blog/the-real-benefits-of-the-virtual-dom-in-react-js/.

Npmjs.com 2019. React. Viitattu 21.9.2019 https://www.npmjs.com/package/react.

Reactjs.org 2019. Components and props. Viitattu 16.5.2019 https://reactjs.org/docs/components-and-props.html.

Reactjs.org 2019. State and lifecycle. Viitattu 28.9.2019 https://reactjs.org/docs/state-and-lifecycle.html.

Reactjs.org 2019. Introducing JSX. Viitattu 26.9.2019 https://reactjs.org/docs/introducing-jsx.html.

Richey, B. 2017. Learning React With Create-React-App (part 1). Viitattu 26.5.2019 https://medium.com/in-the-weeds/learning-react-with-create-react-app-part-1-a12e1833fdc.

Singhal, A & Cutts, M. 2010. Using site speed in web search ranking. Viitattu 14.5.2019 https://webmasters.googleblog.com/2010/04/using-site-speed-in-web-search-ranking.html.

Upwork.com 2019. A Beginner's Guide to Front-End Development. Viitattu 26.5. 2019 https://www.upwork.com/hiring/development/beginners-guide-to-front-end-development/.

Wagner, J. 2019. Why Performance Matters. Viitattu 12.5.2019 https://developers.google.com/web/fundamentals/performance/why-performance-matters/.

W3scools.com 2019. CSS tutorial. Viitattu 21.9.2019 https://www.w3schools.com/css/.

W3schools.com 2019. Javascript tutorial. Viitattu 21.9.2019 https://www.w3schools.com/js/.

W3schools.com 2019. React Props. Viitattu 17.9.2019 https://www.w3schools.com/react/react_props.asp.