Haaga-Helia
ammattikorkeakoulu Oy

Haaga-Helian oppilaille suunnattu verkkosivusto

Tomas Kukk

Opinnäytetyö
Tietojenkäsittelyn
koulutusohjelma
2019

Tiivistelmä

Haaga-Helia ammattikorkeakoulu Oy

Tekijä(t)
Tomas Kukk

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Raportin/Opinnäytetyön nimi
Haaga-Helian oppilaille suunnattu verkkosivusto

Sivu- ja liitesivumäärä
48 + 4

Projektin tavoitteena oli luoda toiminnallinen verkkosivusto Haaga-Helian oppilaiden käyttöön. Sivustolta löytyisi oppilaille oleellinen ja päivittäin tarvittava informaatio kuten lukujärjestys, ruokalistat ja opiskelijajuhlat. Lisäksi sivustolla olisivat linkit tärkeimpiin sovelluksiin kuten Moodleen, Peppiin sekä Lukkarikoneeseen.

Ensimmäisessä osiossa käyn läpi projektiin vaadittujen teknologioiden taustaa, ja selvitän miksi ne ovat olleet paras valinta tähän projektiin. Projektissa käytetyt teknologiat ovat React.js, Node.js, GraphQL ja Cypress.

Toisessa osiossa esittelen projektin luontia alusta loppuun. Näytän miten verkkosivuston palvelinpuoli, sekä käyttöliittymä luodaan. Ensimmäisenä käyn läpi palvelinpuolen ohjelmoinnin, jonka jälkeen esittelen käyttöliittymän tärkeimpien osien luomista.

Viimeisessä osiossa pohdin, ovatko projektin alussa asetetut tavoitteet saavutettu ja miten onnistunut projekti oli kaiken kaikkiaan. Käyn läpi haasteita joita esiintyi matkan varrella, sekä analysoin mitä olisi kannattanut tehdä toisin. Toisaalta tarkastelen myös, mikä on onnistunut erityisen hyvin. Lopuksi arvioin vielä mahdollisia jatkoaskeleita.

Asiasanat
React.js, Node.js, GraphQL, Cypress ja SPA

Sisällys

1 Johdanto

Projektin sovellus luotiin auttamaan Haaga-Helian oppilaiden arkea koulunkäynnissä. Lopputuloksena on sivusto, josta löytyy keskeisimmät tiedot koulunkäyntiä varten. Käyttöliittymään on tarkoitus saada mukavasti esille keskeisimmät asiat kuten ruokalistat, ryhmien lukujärjestykset ja tulevat opiskelijajuhlat. Toinen tärkeä ominaisuus on auttaa opiskelijoita pääsemään nopeasti koulun järjestelmiin käsiksi luomalla pikalinkit muun muassa Moodleen, Peppiin ja Lukkarikoneeseen.

Ensimmäisenä luodaan palvelinpuoli, johon kerätään tiedot ulkoisista REST-rajapinnoista. Jotta sovellus toimisi nopeasti ja luotettavasti, tieto haetaan omalta palvelimelta eikä suoraan muiden verkkosivustojen rajapinnoista. Tarkoituksena on tallentaa data JSON-tiedostoon, josta ne voidaan hakea nopeammin kuin ulkoisesta REST-palvelusta. Palvelinpuoli laaditaan Node.js teknologialla, hyödyntäen Express.js ja GraphQL kirjastoja.

Käyttöliittymän toiminta pitää olla yksinkertaista ja selkeää. Panostan erityisesti helppokäyttöisyyteen ja vain tarvittavan tiedon näyttämiseen. Uudet käyttäjät voivat näin ymmärtää sivuston toimintaperiaatteen nopeasti. Sivusto luodaan React.js kirjastoa käyttäen, koska React antaa hyvän tuen Single Page Application tyyppisille sivustoille. Tietoa voi näin ollen esittää tarpeen mukaan käyttäjän tekemien toimintojen myötä.

1.1 Käsitteet

AJAX (Asynchronous JavaScript and XML) = Tapa tehdä palvelimelle pyyntöjä ilman sivuston uudelleenlatausta (Developer mozilla a 2019).

API (Application program interface) = Sovelluksen sisältämät rutiinit, protokollat ja työkalut jotka määrittävät kuinka sen pitäisi toimia (Webopedia).

Call stack (Kutsupino) = Datarakenne, johon lisätään suoritettavat koodinpätkät jonoon (Medium 2017).

Callback (Takaisinkutsu) = Funktio, joka suoritetaan vasta toisen funktion suorituksen loputtua (Codeburst 2017).

DOM (Document Object Model) = Ohjelmointirajapinta, joka mahdollistaa selaimessa olevien elementtipuiden muokkaamisen (W3schools b 2019).

Endpoint (Päätepiste) = URL Osoite johon voidaan määrittää web palveluita (Biztalk360 2017).

E2E (End to End) = E2E testit varmistavat sovelluksen toiminnan oikeassa käyttöympäristössä, usein selaimessa (Technopedia 2019).

HTML (Hypertext Markup Language) = Hyperteksi, josta verkkosivustot koostuvat (W3schools a 2019).

JSON (JavaScript Object Notation) = Tekstisyntaksi jota voidaan käyttää datan välittämiseen kaikkien ohjelmointikielien välillä (Ecma international 2017).

JSX (JavaScript eXtension) = JavaScript lisäys, jolla voidaan luoda HTML:n näköisiä elementtejä (Fullstackreact 2019).

Middleware = Ohjelmisto joka toimii käyttöjärjestelmän ja sovelluksen välillä (Microsoft Azure).

Non-blocking = Menetelmä joka ei estä ohjelman etenemistä, ennen kuin joku osa on suoritettu loppuun (Yourdictionary 2019).

Promise (Lupaus) = Kertoo asynkronisen operaation tilan ja tuloksen (Developer mozilla b 2019).

Runtime = Periodi jonka aikana sovellus toimii, alkaa sovelluksen käynnistyessä ja loppuu sovelluksen toiminnan loppuessa (Techterms 2019).

SPA (Single Page Application) = Sivusto, joka koostuu vain yhdestä HTML tiedostosta. Sivuston sisältö vaihtuu usein käyttäjän tekemistä toiminnoista (Fullstackopen 2019.)

2 React.js

React on Facebookin luoma JavaScript-kirjasto, jota käytetään käyttöliittymien luomiseen verkkosivustoilla. Klassisesti verkkosivustot luodaan kirjoittamalla HTML:ää, jota selain tulkitsee. Tämän jälkeen HTML-elementit muutetaan sitten visuaaliseen muotoon, esim. tekstiksi tai listoiksi.

Reactia käytettäessä kirjoitetaan JavaScript koodia, joka käännetään selaimelle ymmärrettävään muotoon. Selain ymmärtää JavaScriptiä, mutta ei tue Reactissa käytettävää JSX-lisäystä. JSX:n lopputuloksena on HTML-elementtejä, jotka näytetään verkkosivustolla. Reactia käytettäessä on kuitenkin monia etuja normaalin HTML-koodin kirjoittamisen sijaan. Klassisessa tavassa ei ole mahdollisuutta muuttaa sivuston DOM:ia ilman sivuston uudelleenlatausta. Reactin avulla pääsemme muuttamaan DOM:ia ikään kuin lennosta. Tämä mahdollistaa ulkoasun muuttamisen joko käyttäjän tekemistä tapahtumista, kuten jonkun napin painalluksesta, tai palvelimen aiheuttamista muutoksista, kuten datan saamisesta palvelimelta.

Tämä kaikki on mahdollista virtuaalisen DOM-rakenteen avulla, jonka React luo. Kun sivustolla tehdään uudelleengenerointi (engl. render), React laskee edellisen ja nykyisen virtuaalisen DOM:in erot, jolloin koko sivuston rakennetta ei tarvitse rakentaa uudelleen. Sivuston DOM:ista vaihdetaan vain ne kohdat, jotka ovat muuttuneet vanhan ja nykyisen virtuaalisen DOM:in välillä. Tämä tapahtuu hyvin nopeasti ja käyttäjälle muutokset saattavat vaikuttaa välittömiltä. (Fedosejev 2015, 17-19.)

2.1 Reactin komponentit

Reactissa on mahdollista luoda kahdenlaisia komponentteja: funktionaalisia tai luokkakomponentteja. Näistä jälkimmäinen sisältää muutamia ominaisuuksia, joita funktionaalisiin komponentteihin ei voida määrittää. Suurin osa komponenteista luodaan yleensä kuitenkin funktionaalisina. Komponentit sisältävät JSX-elementtejä, jotka käännetään puhtaaksi JavaScriptiksi. JSX-elementit voivat sisältää valmiita HTML:n tyylisiä elementtejä kuten . Komponentit voivat sisältää myös toisia käyttäjän luomia komponentteja, jolloin syntyy komponenttien hierarkia. Tällä tavalla ei ole tarvittavaa kirjoittaa aina uutta komponenttia, vaan niitä voidaan käyttää uudelleen.

Komponentin sisältö saattaa myös muuttua niille vietyjen tietojen perusteella. Tiedon välitys komponenteille tapahtuu propsien avulla. Propseissa voi viedä tietoa, kuten merkkijonoja, lukuja tai, kuten useimmiten, JavaScript-objekteja.

Projektin juuressa on aina juurikomponentti, joka yleensä sisältää muita käyttäjän luomia komponentteja. Tämä juurikomponentti viedään index.html-tiedoston -elementtiin, jolloin kaikki sen alikomponentitkin generoituvat sivustolle. (Reactjs a 2019.)

2.2 Komponenttien tilat

Komponenteille voidaan määritellä tiloja, joiden arvot voivat muuttua sovelluksen toiminnan aikana. Jos komponentin tila muuttuu, se generoidaan uudelleen sivuston DOM:iin. Tämä saa sivuston ulkoasun mahdollisesti muuttumaan. Tämä muutos voi tapahtua joko suoraan komponentin sisäisen tilan muutoksen myötä, tai propsien välityksellä saatavan tilanvaihdoksen jälkeen.

Lähes koko Reactin olemassaolon aikana tilat on määritelty vain luokkakomponentteihin, eikä funktionaalisiin komponentteihin ole ollut edes mahdollista määrittää tiloja. Alle kaksi kuukautta tämän tekstin kirjoittamisesta julkaistiin React versio 16.8.0, jossa esiteltiin koukut (engl. hooks). Koukut mahdollistavat tilojen määrittelyn myös funktionaalisiin komponentteihin. Tämä tapahtuu hieman eri tavalla, kuin tilojen määritys luokkakomponentteihin. Tulevissa projekteissa suositellaan käyttämään koukkuja vanhan tyylin sijasta, mutta luokkakomponentin tilaa ei ole tarkoitus ottaa pois käytöstä ainakaan lähiaikoina. (Reactjs b 2019.)

2.2.1 Luokkakomponentin tila

Luokkakomponentin tila määritellään sen konstruktoriin tyylillä:

```
constructor(props) {
  super(props)
    this.state = { laskin: 0 }
}
```

Tässä tapauksessa tilaan määriteltiin yksi muuttuja nimeltä laskin, joka saa arvokseen kokonaisluvun 0. Tilan arvoa voi myöhemmin vaihtaa haluamansa mukaan. Kun tilan arvoa halutaan muuttaa, se tulee tehdä *setState*-funktiolla. Tilaa ei saa koskaan muuttaa suoraan esim. tyylillä: **this.state.<muuttuja> = arvo**, vaan tulee aina käyttää setState-funktiota.

Tilan muuttaminen on asynkroninen tapahtuma, joten sen ei pidä luottaa vaihtuvan heti. Tämä tarkoittaa sitä, että sen suorittamista ei jäädä odottamaan koodissa. Synkroniset tapahtumat sen sijaan tehdään ennen eteenpäin siirtymistä. setState-funktion sisälle on mahdollista määrittää takaisinkutsu (engl. callback), joka palauttaa tilan vaihtumisen.

Tämä mahdollistaa sen, ettei komponenttia generoida sivustolle ennen sen tilan vaihtumista. (Sengupta, Singhal & Corvalan. 2016, 55-78.)

Normaali setState-funktio voisi näyttää tältä:

```
this.setState({ laskin: this.state.laskin + this.props.inkrementoi });
```

Takaisinkutsua käyttävä setState-funktio voisi taas näyttää seuraavalta:

```
this.setState(( state, props) => ({
  laskin: state.laskin + props.inkrementoi
}));
```

2.2.2 Funktionaalisen komponentin tila

Funktionaalisten komponenttien koukut tarjoavat mahdollisuuden pilkkoa suurien ja kompleksien komponenttien logiikan pienempiin osiin, joilla on omat tilat. React kannustaa tähän ja aikomuksena on korvata ajan myötä luokkakomponenttien tilat kokonaan. Nämä tilat käyttäytyvät hyvin samalla tavalla kuin luokkakomponenttien tilat, mutta niiden määritteleminen ja muuttaminen seuraavat erilaista syntaksia.

Jos halutaan ottaa tila käyttöön funktionaalisessa komponentissa, on aluksi tuotava *useState*-funktio react-kirjastosta. Määritys ei tässä tapahdu konstruktorissa kuten luokkakomponentissa, sillä funktionaalisiin komponentteihin ei voi määritellä konstruktoria. Tila määritellään listana joka sisältää itse tilan arvon, sekä siihen käytettävää funktiota tilan muuttamiseen. Tilan määritys voisi olla tämän näköinen:

```
const [laskin, setLaskin] = useState(0);
```

useState-tila toimii hyvin samankaltaisesti kuin luokkakomponentin this.state. Ainoana erona on, että funktionaalisen komponentin *setLaskin*-funktio ei yhdistä uutta ja vanhaa tilaa, vaan korvaa sen aina uudella arvolla. Nyt jos haluaisimme muuttaa laskimen arvoa, sen voisi tehdä seuraavasti:

```
setLaskin(laskin + 1);
```

Projektin hallinta on helpompaa, kun käytetään koukkuja luokkakomponenttien tilojen sijaan. Tiloja voidaan määritellä monissa komponenteissa ja niiden hallinta toimii yleensä saman komponentin sisältä. Vanhan tyylin ymmärtäminen on kuitenkin tärkeää, ainakin vielä tänä päivänä. Melkein jokainen tuotannossa oleva Reactin tiloja käyttävä ohjelma on luonnollisesti rakennettu vanhalla tyylillä, sillä tämä on vielä hyvin uusi tapa lähestyä tiloja. (Reacjs b 2019.)

2.3 React Bootstrap

React hoitaa JavaScriptillä HTML-puolen sivustosta. Näin jää kuitenkin CSS eli tyylien lisääminen koskemattomaksi. Hyödyllinen ja yleinen työkalu tämän helpottamiseksi on React Bootstrap-kirjasto. Kuten React, React Bootstrap on myös avoimen lähdekoodin projekti. Tämä kirjasto auttaa värikkään ja käyttäjäystävällisen sivuston luomisessa.

Monilla elementeillä jotka määriteltäisiin normaaleina JSX-elementteinä, kuten <table> on olemassa omat määrittelyt React Bootstrapissa. Esimerkiksi <Table>-elementti on myös normaali JSX <table>, mutta siihen on liitetty mukaan monia tyylejä. Näiden sisältöä on helppo tarkastella avoimen lähdekoodin ansioista. (Github a 2019.)

React Bootstrap-kirjastolla on monia kilpailijoita. Material-ui on suosituin käyttöliittymien tyyleihin tarkoitettu kirjasto ainakin Github-tähtien mukaan, joita sillä on yli 45 000, kun React Bootstrapillä on 15 000 tähteä (Github a 2019; Github b 2019). Monesti parempi vaihtoehto saattaa kuitenkin olla Bootstrap, sillä paketeissa on suuria kokoeroja. Bootstrapin ollessa yli kolme kertaa pienempi, sillä on suuriakin vaikutuksia sivuston latausaikaan. Nämä paketit ovat molemmat kookkaita: pelkän React Bootstrap-kirjaston lataus sivustolla vie 3G-yhteydellä noin 0.8 sekuntia, kun material-ui vie yli 2 sekuntia. (Bundlephobia a 2019; Bundlephobia b 2019.)

3 Node.js

Node.js toimii sovelluksissa palvelinpuolta ohjelmoidessa. Se on rakennettu Google Chromen V8-moottorin päälle ja ohjelmointikielenä toimii JavaScript. ECMAscriptin standardien päivittyessä nopeasti, Node lisää uusimpia toimintoja jatkuvasti lisää (Nodejs 2019.) Uusimmat versio tukevat jo lähes kaikkia ES6:en toimintoja (Node green 2019).

Nodea asennettaessa sen mukana ei tule suuria määriä valmiita paketteja, vaan ne tulee asentaa ohjelman kehittyessä ja niitä tarvittaessa. Paketinhallinnasta huolehtii Noden mukana tullut paketinhallintajärjestelmä npm. Projektin juuresta löytyy package.json-tiedosto, johon on määritelty sovelluksen riippuvuudet. Näiden riippuvuuksien kirjastot ladataan node_modules kansioon. Pakettien lataaminen onnistuu terminaalista komennolla **npm install <kirjaston nimi>**. (Mardan 2014, 8-20.)

Npm koki suosion menetystä yarn-paketinhallinan julkaisun myötä. Se luotiin nopeammaksi ja tietotuvallisemmaksi kuin npm. Molemmat toimivat hyvin samantyyppisellä syntaksilla, joten toisen käyttöönotto ei ole merkittävä muutos. (Sitepoint 2016.)

Npm ei halunnut hävitä kilpailua. Pian uusi julkaisu ratkaisi niitä ongelmia, jonka vuoksi yarn sai alun perin paljon suosiota. Nopeus parani huomattavasti ja tallennuspaikaksi määräytyi aina node_modules -kansio. Muutoksien myötä eroavaisuudet ovat vähentyneet huomattavasti ja paketinhallinnan valinta on tällä hetkellä lähinnä makuasia. (Iamturns 2018.)

3.1 Toimintaperiaate

Node.js toimii asynkronisesti takaisinkutsuja käyttäen. Vaikka toimintoja voidaan suorittaa JavaScriptissä vain yksi kerrallaan, takaisinkutsut mahdollistavat estämättömän (engl. non blocking) menetelmän. Kuvitellaan tilanne, jossa dataa haetaan tietokannasta. Voimme estämättömässä menetelmässä siirtyä eteenpäin koodin suorituksessa kun datan haku on käynnissä. Kun tieto on saapunut tietokannasta, takaisinkutsun määrittelemä funktio viedään kutsupinoon (engl. call stack), jonka jälkeen voidaan palata takaisin suorittamaan funktion koodia.

setTimeout on JavaScript-funktio, joka käyttää takaisinkutsua. Takaisinkutsut suoritetaan vasta sen jälkeen, kun priorisoitu koodi, eli synkroninen koodi, on suoritettu loppuun. Toisin sanoen, kutsupinon pitää olla tyhjä ennen kuin takaisinkutsujen koodit suoritetaan.

Takaisinkutsut viedään selaimen tarjoamaan web API:iin, joka mahdollistaa tämän toiminnan. Edellä kuvatun hoitaa Noden tapahtumasilmukka (engl. event loop), joka pitää huolen että takaisinkutsujen listalta viedään oikeat asiat oikeaan aikaan kutsupinoon. (Medium 2017.)

Takaisinkutsujen käyttö on kuitenkin vähentynyt ja siirtyminen lupauksiin (engl. promise) ja async/await:iin on lisääntynyt. Lupaukset toimivat hieman samalla periaatteella. Merkittävänä erona on kuitenkin *then*-funktio, jonka sisälle voidaan määrittää mitä tehdään kun lupaus on täytetty. Kyseisiä then-funktioita voidaan laittaa monia peräkkäin joista jokainen suoritetaan vasta edellisen then-funktion suorituksen jälkeen. Lupausten then-funktioiden jälkeen hoidetaan yleensä virheidenkäsittelyä, jos jokin menee pieleen. Tämä aiheuttaa kuitenkin sen, että saatamme tarvita virheidenkäsittelyä myös aikaisemmin funktiossa synkronisen koodin suorituksessa. Tällöin joutuisimme käsittelemään virheitä kahdesti peräkkäin, aluksi try-catch -blokilla ja sitten then-catch -blokilla.

Tämä ongelma ratkesi kun async/await julkaistiin. Async määritellään funktioon, jonka jälkeen await:ia voidaan käyttää. Jos await:ia koitetaan käyttää jossain funktiossa joka ei ole määritelty async:inä, syntyy virhe. Async await -funktiot palauttavat itsessään lupauksia, joten ne toimivat hyvin samalla tavalla kuin lupaukset itsessään. Koodin kirjoittaminen kuitenkin selkenee ja sen ymmärtäminen on helpompaa. Virheiden käsittely myös helpottuu, sillä koko funktio voidaan nyt määrittää try-catch -blokin sisään eikä then-catch -blokkia enää tarvita. (Vohr, S. 2.6.2018.)

Async await ei ole kuitenkaan kaikkien mielestä hyvä lisäys. Sen käyttö on melko rajoitettua eikä sitä voi käyttää koodin ylätasolla ilman funktiomääritystä. Atomisten kyselyiden luonti tietokantaan on myös hieman hankalaa, ja operaatioiden skaalautuessa syntyy jonkin verran hankaluuksia. (Kosev 23.8.2015.)

Esimerkkinä seuraavaksi lyhyt tiedosto, joka näyttää asynkronisen koodin toimintaa. On hyvä huomioida, että while-silmukka on synkroninen tapahtuma, joka pitää suorittaa loppuun ennen kuin koodissa voidaan siirtyä eteenpäin. Suoritetaan tiedosto komennolla **node <tiedoston nimi>**.

```
const aloitusAika = new Date();

setTimeout(() => {
  console.log('\t100 millisekuntia.. vai oikeasti',
  new Date() - aloitusAika, 'millisekuntia\n');
}, 100);

console.log('Minut suoritetaan ensimmäisenä!\n');
```

```
console.log('Aloitetaan while looppi\n');

while (new Date() - aloitusAika < 1000) {}

console.log('Loopin alusta on noin',
((new Date() - aloitusAika) / 1000).toFixed(4),
'sekuntia\n');

console.log('Minä olen synkroninen tuloste\n');
```

Ensimmäisessä rivissä talletetaan muuttujaan *aloitusAika* sen hetkinen Date-objekti, jonka JavaScript tarjoaa automaattisesti. setTimeout-funktio määrittää takaisinkutsun aikaisintaan 100 millisekunnin päähän. Tämän jälkeen tulostetaan konsoliin muutama tekstin pätkä. While-silmukka ajaa tyhjää sen aikaa kun on syntynyt sekunnin ero sen hetkisen ajan ja aloitusajan välillä. Tämän jälkeen tulostetaan konsoliin suunnilleen se aika joka on kulunut while-silmukan alustamisesta. Viimeisenä koodissa on normaali synkroninen console.log-funktio.

Vaikka setTimeout-funktion alustamisesta on kulunut yli sekunti silmukan jälkeen ja siihen on asetettu takaisinkutsun ajaksi 100 millisekuntia, sen tulostus näkyy vasta tiedoston viimeisen rivin jälkeen. Tämä osoittaa sen, kuinka kutsupino pitää tyhjentää ennen kuin voidaan siirtyä takaisinkutsuihin. Tässä tapauksessa ei siis ole merkitystä ensimmäisen setTimeout-funktion console.log tulosajalle, onko ensimmäisessä funktiossa määritetty 0, 10, 100 vai 1000 millisekunnin viive. Esimerkkituloste näyttää tältä, jossa takaisinkutsua käyttävä tuloste on sisennettynä kauemmas reunasta (kuva 1).

```
Minut suoritetaan ensimmäisenä!

Aloitetaan while looppi

Loopin alusta on noin 1.0000 sekuntia

Minä olen synkroninen tuloste

        100 millisekuntia.. vai oikeasti 1002 millisekuntia
```

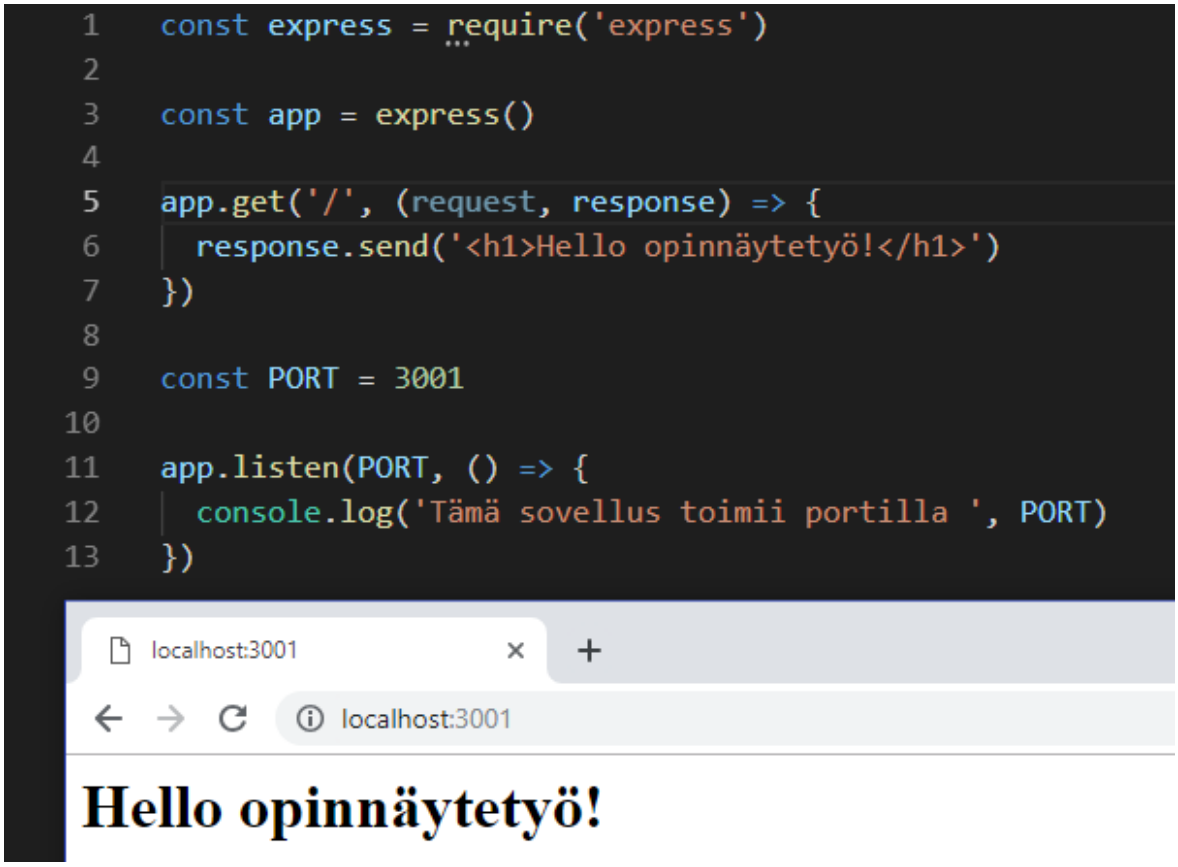
Kuva 1. Asynkronisen koodin tuloste

Noden asynkroninen toimintaperiaate palvelee hyvin nykytilaa jossa käytämme internetiä selaimilla. Hyvät puolet tulevat esille varsinkin kun käyttäjämäärä kasvaa suureksi. Tässä

Node päihittää muita teknologioita kuten PHP:n ja Python-Webin suurillakin lukemilla. Toinen hyvä puoli Nodessa on IO intensiiviset sivustot, eli sellaiset missä luetaan tai kirjoitetaan suuria määriä dataa. (Lei, Ma & Tan 2014.)

3.2 Express.js

Nodessa on valmiina http-kirjasto, mutta sen käyttö on vähäistä. Monet muut kirjastot palvelevat samaa asiaa, ja ne ovat helppokäyttöisimpiä. Suurin ja käytetyin on Express.js. Kyseinen kirjasto antaa paljon avuliaita työkaluja, joita muuten joutuisi itse määrittelemään koodiin normaalia http kirjastoa käytettäessä. Express antaa mahdollisuuden tehdä MVC (Model, View ja Control) -tyyppisiä ohjelmia, jossa jokainen osa-alue on tarkasti erotelta-vissa. Mallien luomiseen (Model) olisi mahdollista käyttää esimerkissä GraphQL-kirjastoa, jolla tehdään GraphQL:ää varten malleja. Päätepisteiden (engl. endpoint) määrittely (Control) on myös yksinkertaistettua http-kirjastoon verrattuna. Esimerkkikuvassa näkyy yksinkertaisen Express sovelluksen koodi (kuva 2).



Kuva 2. Yksinkertainen Express sovellus

Suuremmissa sovelluksissa on yleistä tuoda moduuleita päätiedostoon, jossa kaikki kasataan yhteen. Päätiedoston käynnistyessä se saa ulkoiset moduulit käyttöön. Näitä moduuleita ovat esim. erilaiset mallit ja kontrollerit. Sovellus saattaa olla yhteydessä myös tietokantaan ja sen määrittely tehdään usein päätiedostoon. Express ottaa sitten käyttöön

muun muassa kontrollerit ja mahdolliset jäsentimet (engl. parser). Viimeisenä Express määritellään kuuntelemaan jotain porttia. (Mardan 2014, 33-36.)

4 GraphQL

GraphQL on Facebookin luoma JavaScript-kirjasto. Kirjasto on ollut käytössä Facebookilla vuodesta 2012 lähtien, mutta julkiseen käyttöön se tuli vasta vuonna 2015. Määrittelyn mukaan GraphQL on kyselykieli (GraphQueryLanguage) sekä runtime. Kyseinen runtime ei vaadi mitään tiettyä kieltä käyttöön palvelinpuolelle, vaan tämä päätös on hyvin vapaa sovelluksen kehittäjän kannalta. Myöskään tietokantaa ei tarvitse valita GraphQL:n mukaan, sillä se ei ota kantaa tietokannan toteutukseen. Itse kieli muistuttaa läheisesti JSON-datatyyppiä. (Buna 2016, 6-8.)

Toimintaperiaate on hyvin erilainen kuin laajassa käytössä olevat REST-rajapinnat. Jotta ymmärtäisimme GraphQL:n erikoisuuden, esitellään lyhyesti miten REST-API:t toimivat. REST-rajapinnoissa määritellään eri päätepisteitä, joihin lähetetään http-pyyntöjä. Palvelin suorittaa sille määritellyt tehtävät päätepisteestä riippuen, sekä mahdollisesti vastaa tähän pyyntöön. Http-pyyntöjä joita REST-API:t käyttävät, ovat muun muassa GET, POST, PUT, PATCH ja DELETE. (Restfulapi.)

GraphQL ei noudata samoja periaatteita vaan se käyttää jokaisen pyynnön tekemiseen POST-pyyntöä. GraphQL ei kuitenkaan ole riippuvainen http:stä, vaan sillä voisi käyttää vaikka sähköpostia. Yleinen määritys on kuitenkin POST-pyyntö. Itse operaatio voi olla joko kysely, mutaatio tai subskriptio (engl. query, mutation ja subscription). Näistä ensimmäisellä haetaan tietoa palvelimelta ja toisella muokataan, lisätään tai poistetaan tietoa. Subskriptiolla voidaan määrittää reaaliaikainen muutosten kuuntelu palvelimelle. Subskriptiot pitää itse määritellä palvelinpuolella. Suuri etu REST-rajapintoihin verrattuna on vapaus valita, mitä tietoja haluamme hakea. REST-päätepiste palauttaa aina kaiken tiedon, mutta GraphQL:n kyselellä voimme valita haluamamme tiedot. (Buna 10-18.)

Seuraavassa esimerkissä on kuvattu miten GraphQL:n kysely toimii. Kuvitellaan että tarpeena on tallentaa henkilötietoja tietokantaan seuraavassa muodossa:

```
{
    name: "Tomas Kukk",
    phone: "040-1234567",
    street: "Datakatu B 13 ",
    city: "Helsinki",
}
```

Yllä näkyvien tietojen lisäksi meillä on id-kenttä, jonka GraphQL täyttää puolestamme. Person-skeema näyttää seuraavalta:

```
type Person {
    name: String!
```

```
    phone: String
    street: String!
    city: String!
    id: ID!
}
```

Määritellään seuraavaksi muutama kysely, joilla voimme hakea tietoja palvelimelta:

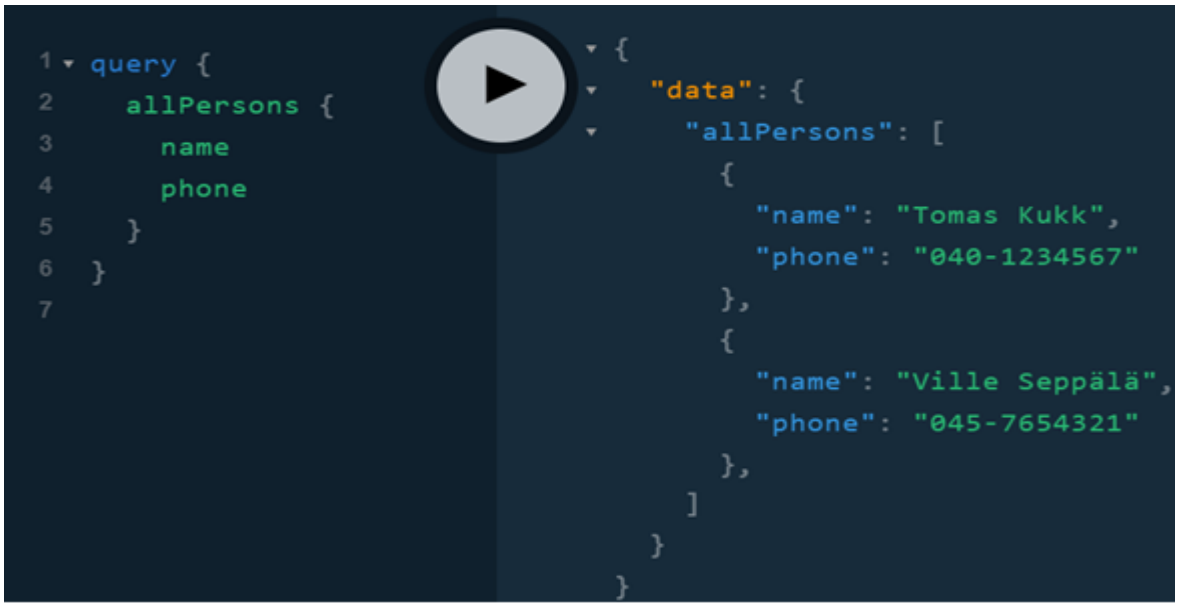
```
type Query {
    allPersons: [Person!]!
    findPerson(name: String!): Person
}
```

Viimeisenä joudumme määrittelemään resolverit jokaiselle kyselylle. Resolverit kertovat, mitä haluamme kyselyn tekevän. Niitä ei käsitellä tässä sen enempää, mutta yllä kyselyiden resolverit ovat seuraavat:

```
Query: {
    allPersons: () => persons,
    findPerson: (root, args) => persons.find(p => p.name === args.name),
}
```

Nyt sovelluksen pitäisi toimia sujuvasti, joten tarkastellaan GraphQL:n kykyä valita haluamme tietoa pienessä mittakaavassa. Käynnistettäessä sovellus Apollo-serverin kanssa, se toimii normaalin asetuksen mukaan portilla 4000 (Apollographql a).

Osoitteeseen localhost:4000 tulee käyttöön hyödyllinen käyttöliittymä, jossa on mahdollista kokeilla äsken esiteltyä palvelinpuolta. Jos tarpeena on löytää kaikkien henkilöiden nimet ja puhelinnumerot piittaamatta muusta tiedosta, se tapahtuisi seuraavan kyselyn avulla. Kuvassa vasemmalla itse kyselyn muoto ja oikealla palvelimen vastaus (kuva 3).



Kuva 3. GraphQL allPersons kysely esimerkki

Seuraavassa vielä tilanne jossa tarpeena on löytää yhden henkilön kaikki tiedot. Tiedot haetaan henkilön nimen mukaan (kuva 4).

Kuva 4. GraphQL findPerson kysely esimerkki

Kyselyiden lisäksi on olemassa mutaatioita ja subskriptioita. Niitä ei käsitellä tässä työssä kovin laajalti, mutta seuraavassa kappaleessa esitellään niiden perusteet.

Mutaatiot hoitavat tiedon lisäämistä ja päivittämistä, eli korvaavat REST:in PUT- ja POST-pyynnöt. Subskriptiolle ei ole olemassa korvaavaa asiaa REST-rajapinnoissa, vaan sen ominaisuus on vain GraphQL:ssä. Subskriptioiden avulla on mahdollista määrittää tilaus palvelimelle. Tämä tilaus tuo dataa palvelimelta automaattisesti jos kyseisen tilauksen mukaiseen skeemaan on lisätty uusi objekti. Edelliseen esimerkkiin olisi voitu määrittää siis subskriptio henkilölle, jolloin uuden henkilön lisäys saataisiin heti tietoon. Subskriptioiden tuoma lisäys voi olla arvokas sovelluksissa, joissa lisäykset dataan halutaan nähdä mahdollisimman nopeasti ja vaivattomasti. (Apollographql b; GraphQL.)

5 Cypress

Cypress on E2E testausta varten tehty kirjasto, jolla on mahdollisuus kirjoittaa ylimmän tason testejä JavaScriptillä. Selenium on ollut pitkään suosittu kirjasto E2E-testaukseen, mutta siinä piilee ongelmia SPA:n testaukseen liittyen. Cypress toimii itse selaimen sisällä eikä tee komentoja selaimen ulkopuolelta, kuten Selenium. Tämä mahdollistaa mm. AJAX-pyyntöjen kuuntelun ja modifioinnin. Testauksessa ei ole turhaa viivettä, vaan se toimii reaaliaikaisesti, kuten sovellus toimisi itse käyttäjän käytössä.

Aikaisemmin E2E-testit ovat mahdollistaneet lähinnä vain käyttöliittymän testausta. Cypress mahdollistaa kuitenkin myös palvelimelle lähtevien pyyntöjen testauksen, sekä kuuntelemaan palvelimen vastauksia. Koska kirjasto asennetaan lokaalisti tietokoneelle, myös kuvakaappausten ottaminen, videoiden kuvaaminen ja tiedostojen hallinta on mahdollista. (Cypress 2019.)

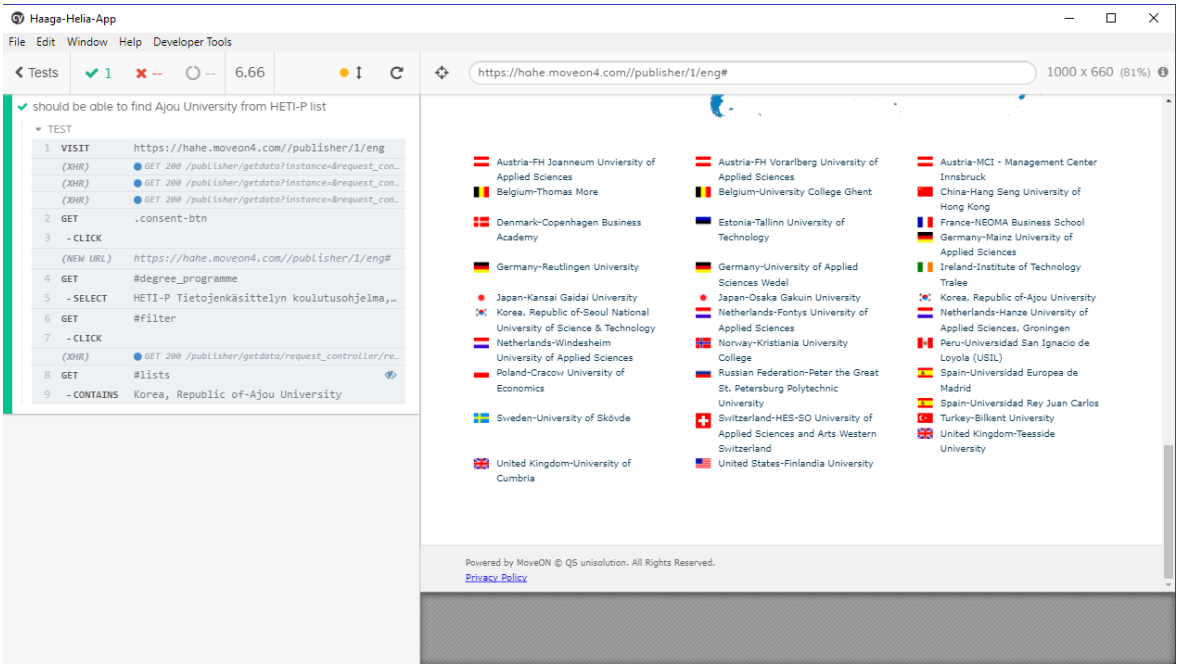
Seuraavassa on testi, joka on luotu Cypressillä. Esimerkin testissä testataan että Haaga-Helian opiskelijavaihtosivustolta löytyy Ajou University sen jälkeen, kun on valittu avoimet kohteet Tietojenkäsittelyn koulutusohjelmalle (HETI-P). Esimerkin koodi:

```
it('should be able to find Ajou University from HETI-P list', () => {
  cy.visit('https://hahe.moveon4.com//publisher/1/eng');
  cy
    .get('.consent-btn');
    .click();
  cy
    .get('#degree_programme');
    .select('HETI-P Tietojenkäsittelyn koulutusohjelma, päivä (408)');
  cy
    .get('#filter');
    .click();
  cy
    .get('#lists');
    .contains('Korea, Republic of-Ajou University');
});
```

Aluksi Cypress menee oikeaan osoitteeseen. Tämän jälkeen vahvistetaan sivustolle evästeiden käyttö etsimällä nappi, jolla on class-arvo *consent-btn* ja klikkaamalla sitä. Seuraavaksi testi hakee select-kentän, jonka id on *degree_programme*. Select-kentästä valikoituu tietojenkäsittelyn koulutusohjelma. Lopussa sivustolla tehdään haku klikkaamalla nappia, jonka id on *filter*. Viimeisenä sivulta haetaan lista, jonka id on *lists*. Viimeiseksi tarkistetaan, löytyykö Ajou University tekstin perusteella.

Cypressin käynnistyessä aukeaa käyttöliittymä, jossa näkyy projektin testit. Valitsemalla juuri tuotu testi, uudessa avautuvassa ikkunassa näkyvät testin vaiheet, testiin käytetty aika ja lopputulos.

Seuraavassa kuvassa on esitelty testin lopputulos käyttöliittymässä (kuva 5). Oikealla on itse selain, jossa testi suoritetaan reaaliajassa. Vasemmalla nähdään kuinka Cypress näkee AJAX (XHR) -pyyntöjen lähtevän ja jää odottamaan niiden vastausta. Koulu etsitään listalta vasta sen jälkeen, kun sivusto on generoitunut uudestaan palvelimen vastatessa.



Kuva 5. Cypress käyttöliittymän näkymä esimerkkitestin jälkeen

6 Verkkosivuston toteutus

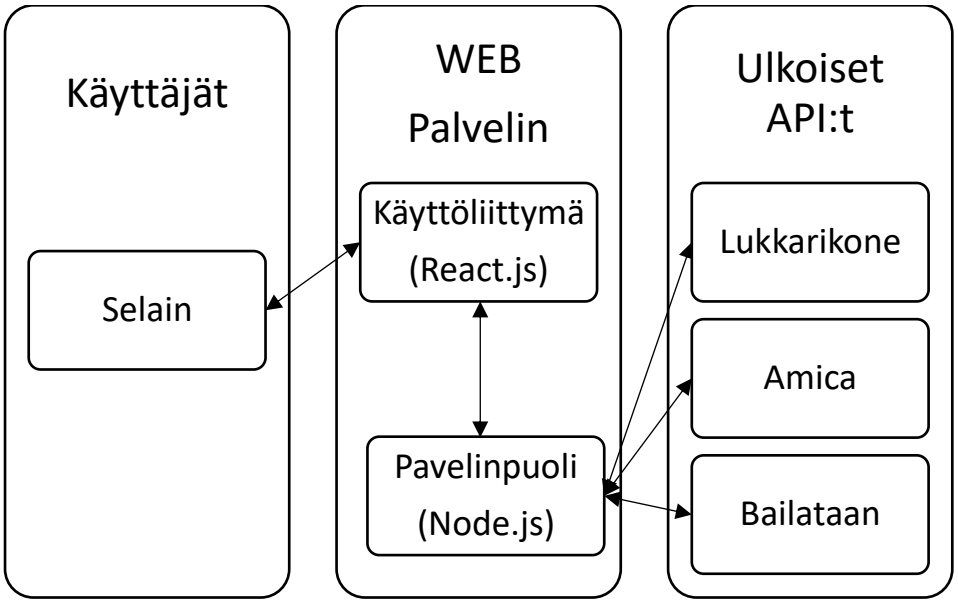
Ensimmäisen osuuden kappaleissa on kerrottu Node.js, React.js, GraphQL ja Cypress-kirjastoista, joilla kaikilla ohjelmointikielenä toimii JavaScript. Tämä on suurin syy jokaisen teknologian valinnalle. JavaScriptillä on mahdollista luoda moderneja sivustoja, jotka toimivat nopeasti ja luotettavasti. Toisena syynä teknologioiden valinnalle on tavoitteeni oppia käyttämään niitä paremmin.

React ja Node sopivat hyvin yhteen. Molemmat tukevat ES6 ja uudempiin standardeihin perustuvaa JavaScriptiä, sekä käyttävät asynkronisia tapahtumia. Molemmat näistä käyttävät samanlaista tapaa ottaa käyttöön moduuleita, vaikka syntaksi onkin hieman erilainen. Paketinhallintana toimii molemmissa npm, joten siitä ei aiheudu ylimääräistä päänvaivaa.

Pyrin luomaan sovelluksen mahdollisimman moderneja teknologioita käyttäen. GraphQL saattaa olla tulevaisuudessa laajasti käytetty teknologia, joten projektin luominen sen avulla on perusteltua. Cypress taas saattaa uhata Seleniumia tulevaisuuden E2E-testauksessa, joten valitsen sen.

Kutsun tulevissa kappaleissa molempia muuttujatyyppejä ”const” ja ”let” yksinkertaisesti muuttujiksi. Näissä on kuitenkin eroa, sillä const on vakioarvoinen muuttuja, kun let-muuttujan arvoa voidaan myöhemmin vielä muuttaa.

Alla olevassa kuvassa on vielä esitelty miten sovellus tulee toimimaan (kuva 6).



Kuva 6. Sovelluksen toimintaperiaate

7 Palvelinpuoli

Alustetaan projektin palvelinpuoli komennolla *npm init*, joka luo meille package.json-tiedoston siihen syötettyjen tietojen mukaan. Sovellus käynnistetään komennolla *npm start*, joka käynnistää tiedoston index.js. Ladataan tarpeelliset riippuvuudet, jotta pääsemme aloittamaan projektin luomisen. Ajetaan seuraavat komennot terminaalissa:
* npm install **axios** *
* npm install **express** *
* npm install **cors** *

7.1 Päätiedosto index.js

Index.js on tiedosto, joka käynnistää koko ohjelman. Tänne määritellään siis itse ohjelman toteutus, joka saa käyttöönsä monia moduuleita. Luodaan tiedostopohja, jossa voidaan myöhemmin ottaa käyttöön eri päätepisteet. Tiedostoon määritellään tarvittavat asiat:

```
const http = require('http');
const express = require('express');
const app = express();
const bodyParser = require('body-parser');
const cors = require('cors');

app.use(cors());

const server = http.createServer(app);

const PORT = 3001;

server.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});
```

Yllä olevan tiedoston sisältö on kaikki mitä tarvitsemme toistaiseksi. Luomme express-sovelluksen, joka ottaa käyttöön cors eli cross-origin resource sharing middlewaren. Näin voimme myöhemmin tehdä pyyntöjä muista osoitteista. Luodaan muuttuja *server* käyttämällä http-kirjastoa. Näin voimme ottaa myöhemmin käyttöön kätevästi päätepisteet syntaksilla *app.use(<päätepiste>, <reititin>)*. Määritetään viimeisenä server kuuntelemaan porttia 3001, jossa sovelluksemme toimii.

7.2 Juhlat

Juhlien tarjoaminen käyttöliittymälle tehdään GraphQL:n avulla. Ladataan siis riippuvuudet, joita tarvitsemme tämän käyttämiseen:
* npm install **express-graphql** *

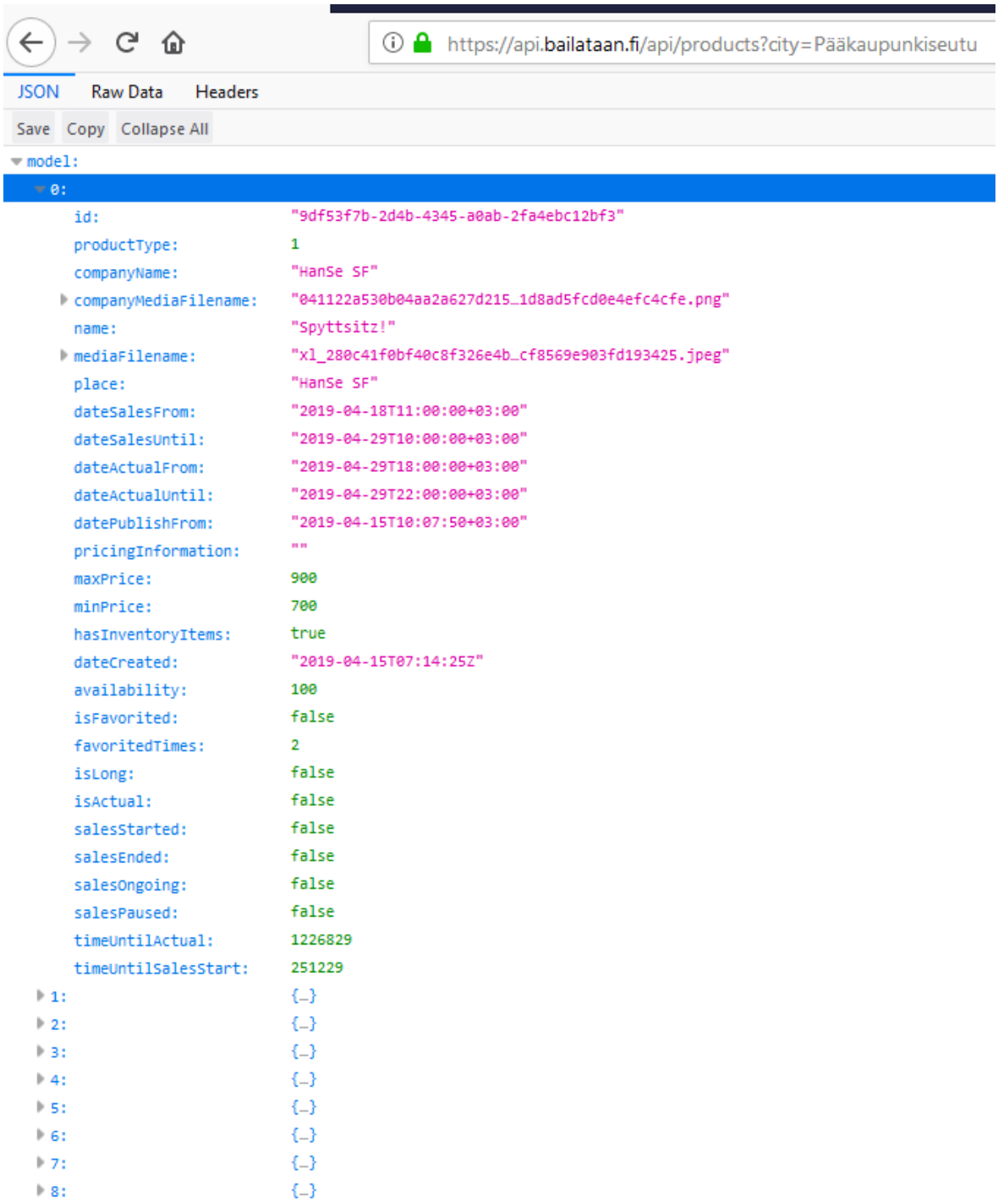
* npm install **graphql** *

Jotta saamme juhlat tarjottua käyttöliittymälle, ne pitää aluksi hakea Bailataan.fi sivuston API:sta. Tähän käytämme axios-kirjastoa, jonka avulla tehdään GET-pyyntö osoitteeseen, josta löytyy haluamamme juhlat JSON-muodossa. Juhlia lisätään sen verran harvoin, ettei ole luontevaa tehdä pyyntöä Bailataanin API:in joka kerta kun käyttäjä tulee sivustolle. Luodaan siis tätä varten ratkaisu, jolla voidaan hakea tiedot kerran päivässä API:sta ja tallettaa tämä vastaus muuttujaan. Näin emme myöskään kuormita heitä jatkuvilla pyynnöillä. Luodaan siis tiedosto Bailataancontroller.js kansioon controllers ja katsotaan tiedostoa, joka toteuttaa yllä mainitut asiat:

```
let lastDate = new Date().getDate();

let events;

const thisDay = () => new Date().getDate();

const updateData = (async () => {
  const response = await axios.get
  ('https://api.bailataan.fi/api/products?city=Pääkaupunkiseutu') ;
...
```

Ohjelman käynnistyessä tallennamme muuttujaan *lastDate* kyseisen päivämäärän. Alustetaan muuttuja *events*, jotta sitä voidaan käyttää myös *updateData*-funktion ulkopuolella. Luodaan tämän jälkeen muuttujafunktio *thisDay*, joka palauttaa kyseisen päivän päivämäärän. Tämä on tärkeää luoda funktiona, jotta saadaan oikea päivämäärä vaikka ohjelma olisi ollut toiminnassa montakin päivää. Jos käynnistäisimme sovelluksen kuun yhdeksäs päivä ja antaisimme sen olla toiminnassa 10 päivää, olisi lastDate = 9 ja thisday-funktion kutsu palauttaisi arvon 19.

Tiedoston pääasiallinen async-funktio on updateData. Await:in avulla saamme lupauksen palautettua *response*-muuttujaan, kun siihen saadaan vastaus. Haetaan siis muuttujaan response pääkaupunkiseudun juhlat JSON:ina. Vastauksen data on kutakuinkin tämän näköinen (kuva 7).



Kuva 7. Bailataan API-pyynnön vastaus

GET-pyynnön datasta löytyy model objekti, jonka sisällä on objekteja jotka sisältävät yksittäisen juhlan tiedot. Myöhemmin GraphQL osoittautuu avuliaaksi ”ystäväksi”, sillä emme tarvitse kaikkia kenttiä käyttöömme joita objektien sisältä löytyy. Funktio updateData jatkuu:

```
...
  events = JSON.parse(response.data.model);
  lastDate = thisDay();
})();
```

Koodissa muutetaan seuraavaksi vastaus oikeaan muotoon, jotta se näyttäytyy JSON-objektina. Tallennetaan se muuttujaan events, jonka alustimme aluksi. Muuttujaan tallennetaan siis vastauksen data.model eli model objektin sisältämät objektit.

Funktion viimeisellä rivillä on tärkeä toiminto, jossa tallennamme lastDate-muuttujaan kyseisen päivämäärän. Tämän arvon sijoittamisen käyttötarkoitus nähdään myöhemmin resolverin määrittelyssä. Funktiota updateData pitää kutsua aina kun ohjelma käynnistetään, joten se on luotu IIFE (Immediatly Invoked Function Expression) -tyyppisenä. IIFE-funktio alkaa sululla *(* ja päättyy sulkuun ja funktion kutsuun*)();*. Tämä mahdollistaa funktion kutsun automaattisesti sen jälkeen kun se on huomattu ohjelman suorituksessa.

7.2.1 GraphQL

Edellisen osan käsittelyssä olevasta tiedostosta Bailataancontroller löytyy vielä resolveri, jossa kerrotaan mitä GraphQL-kyselyn pitää palauttaa. Esitellään tämä kuitenkin vasta sen jälkeen kun sille luotu sille vaadittu skeema ja kysely. Skeema ja kysely luodaan GraphQL-kirjastosta löytyvästä funktiosta *buildSchema*. Skeemassa on Bailataanin kentät ja kyselyissa määritelty yksi kysely: *allEvents*. Tämä palauttaa listana jokaisen Event-objektin. Seuraavaksi esitellään tiedosto Event.js joka löytyy models-kansiosta:

```
const { buildSchema } = require('graphql') ;

const typeDefs = buildSchema(`

  type Event {
    id: String
    name: String
    productType: Int
    companyName: String
    place: String
    dateSalesFrom: String
    dateSalesUntil: String
    dateActualUntil: String
    dateActualFrom: String
    datePublishFrom: String
    pricingInformation: String
    maxPrice: Int
    minPrice: Int
    hasInventoryItems: Boolean
    dateCreated: String
    availability: Int
    isFavorited: Boolean
    favoritedTimes: Int
    isLong: Boolean
    isActual: Boolean
```

```
    salesStarted: Boolean
    salesEnded: Boolean
    salesOngoing: Boolean
    salesPaused: Boolean
    timeUntilActual: Int
    timeUntilSalesStart: Int
  }

  type Query {
    allEvents: [Event!]!
  }
`);

module.exports = typeDefs;
```

Siirrytään vielä tiedostoon Bailataancontroller, jossa on määritelty resolveri yllä näytetylle kyselylle. Resolveri näyttää seuraavalta:

```
const resolvers = {
  allEvents: () => {
  if (thisDay !== lastDate()) {
    updateData();
  }
  return events;
 }
}
```

Resolverin funktiossa tarkistetaan if-lauseen avulla, onko muuttujan lastDate ja funktion thisDay palauttama arvo sama. Jos arvo täsmää, palautetaan events-muuttuja. Jos arvot eivät kuitenkaan täsmää, palautus tapahtuu vasta funktion updateData kutsumisen jälkeen

Määritellään nyt index.js-tiedostoon päätepiste, josta nämä tiedot voi hakea GraphQL kyselyllä. Päätetään päätepisteen olevan /api/graphql. Viedään graphqlHTTP:lle vielä määrittelemämme skeema ja resolveri.

```
app.use('/api/graphql', graphqlHTTP({
    schema: schema,
    rootValue: resolvers
}));
```

7.2.2 Bailataan API vs GraphQL API

Verrataan vielä miten kysely määrittelemääni palvelinpuolelle ja Bailataanin API:iin eroaa. Emme tarvitse kaikkea tietoa, jota Bailataanin API meille tarjoaa. Täten GraphQL on oiva valinta, sillä meillä on valta valita mitä tietoa tarvitsemme ja milloin. Verrataan vastauksiin

kuluneita aikoja ja tiedostokokoja. Ensimmäisenä näkyy kolme pyyntöä Bailataanin API:iin (kuva 8).

Kuva 8. Bailataan API:n vastauksien tiedot

Kuten kuvassa näkyy, vastauksen koko on 108.6 kilotavua ja aikaa kuluu reilu 100 millisekuntia. Lähetetään seuraavaksi pyyntö tehdylle palvelimelle GraphQL:llä. Jotta vertaaminen olisi realistista, vastaukset ovat lähetetty ulkoiselle palvelimelle lokaalin sijaan. Palvelin on osoitteessa https://hhapp.info. Lähetetään palvelimelle pyynnöt seuraavalla määrityksellä:

```
query {
  allEvents {
    id
    name
    dateActualFrom
  }
}
```

Kuva 9. GraphQL-pyyntö sovelluksen backendiin

Huomaamme heti eroja vastauksessa. Koska emme hakeneet pyynnössä kuin kolme kenttää: id, name ja actualDateFrom, on vastauksen koko huomattavasti pienempi. Aikaa on kulunut myös hieman vähemmän pyynnön saamiseen (kuva 9).

Yllä oleva vertailu on suuntaa antava eikä siitä voi vetää johtopäätöksiä, onko toinen nopeampi kuin toinen. Sen sijaan varmaa on vastauksen koko. Se tulee olemaan aina pienempi tehdessä pyyntö GraphQL-päätepisteeseen, jos emme hae jokaista kenttää skeeman mukaisesta objektista.

7.3 Ruokalistat

Ruokalistojen lisääminen palvelinpuolelle onnistuu ilman uusien riippuvuuksien lataamista. Toimintaperiaate tulee olemaan samanlainen siten, että haluamme kerran päivässä päivittää tiedot. Nyt kuitenkin teemme hieman REST:iä muistuttavan rajapinnan, joka tarjoaa haluamamme tiedot. JSON-vastaus Amican API:sta tallennetaan paikalliseen JSON-tiedostoon, jonka sisältö tarjotaan päätepisteessä. Käyn tässä läpi vain Malmin ruokalistan toteutuksen, mutta myös Pasilan ja Haagan ruokalistat ovat saatavilla. Niiden toteutus on täysin samanlainen kun Malmin tapauksessa.

Tarkastellaan pääasiallista async-funktiota ja sen toimintaa. Luodaan tiedosto Amicacontroller.js. Määritetään aluksi amicaRouter Expressin Router-funktiota hyödyntäen:

```
const amicaRouter = require('express').Router();
```

Esitellään ensimmäiseksi tilanne, jossa ruokalistaa ei ole vielä tallennettu paikalliseen JSON-tiedostoon kyseisenä päivänä:

```
amicaRouter.get('/malmi/:lang', async (req, res) => {
    try {
        const lang = req.params.lang;
        ...
        const response = await axios.get
        (`https://www.amica.fi/api/restaurant/menu/week?language=${lang}
        &restaurantPageId=7498&weekDate=${tomorrowAsJSON()}`);

        let resStringify = JSON.stringify(response.data);
        fs.writeFileSync(`${lang}amicamalmi${thisDayAsJSON()}.json`,
        resStringify);

        if (fs.existsSync(`${lang}amicamalmi${yesterDayAsJson()}.json`)) {
        fs.unlink(`${lang}amicamalmi${yesterDayAsJson()}.json`, (err) => {
            if (err) throw err;
        })}
        res.send(response.data);
    } catch(exception) {
        console.log(exception);
    }
});
```

Samaan tyyliin kun juhlien haussa, tallennetaan vastaus response-muuttujaan. GET-pyynnön mukana on tullut parametrina *lang*, jonka arvo on fi tai en. Tämän perusteella haetaan joko suomenkielinen tai englanninkielinen versio ruokalistasta. Lopussa on funktion *tomorrowAsJSON*-kutsu. Tämä palauttaa oikean muotoisen merkkijonon, jotta saamme oikean päivän ruokalistan haettua. tomorrowAsJSON-funktio palauttaa seuraavan päivän

päivämäärän. Tämä ominaisuus auttaa hakemaan sunnuntaina jo seuraavan viikon listat, jos ne löytyvät Amican API:sta.

Tämän jälkeen luodaan uusi tiedosto *fs*-kirjastoa käyttäen, (yksi harvoista kirjastoista, joka tulee oletuksena Node.js projektin mukana) joka sisältää saamamme datan. Ennen datan lähettämistä poistamme edellisen päivän tiedoston, jos sellainen on ollut olemassa. Lopuksi vielä lähetämme vastauksena saamamme datan.

Seuraavassa esimerkissä käsitellään tilannetta, jossa tiedosto on jo olemassa:

```
amicaRouter.get('/malmi/:lang', async (req, res) => {
    try {
        const lang = req.params.lang;
        if (fs.existsSync(`${lang}amicamalmi${thisDayAsJSON()}.json`)) {
            let data = fs.readFileSync
            (`${lang}amicamalmi${thisDayAsJSON()}.json`);
            let parsedData = JSON.parse(data);
            res.send(parsedData);
            return;
        }
    ...
```

Tässä tarkistetaan onko tämän päivän tiedosto olemassa ja palautetaan sen sisältö, jos se löytyy. Tämä tehdään lukemalla tiedoston sisältö muuttujaan data, joka muutetaan oikeaan muotoon JSON.parsea käyttäen. Return lauseella päästään funktion sisältä ulos.

Viimeisenä tarpeena on lisätä index.js-tiedostoon määritys tälle päätepisteelle:

```
app.use('/api/amica', amicaRouter)
```

7.4 Lukkari

Lukujärjestysten eli lukkarien tarjoaminen käyttöliittymälle on hankalampi tehtävä, sillä Lukkarikone ei tarjoa REST-rajapintaa. Lukkarikone tekee pyyntöjä erilaisiin PHP-sivustoihin, joiden avulla data haetaan sivustolle. Joudumme palauttamaan käyttöliittymälle yhden PHP-sivuston sisällön, jotta saamme lukkarit näkyviin.

Tärkein toiminto on tarjota käyttöliittymässä hakutoiminto ryhmätunnuksella, joka palauttaa kyseisen ryhmän lukkarin. Tämän lisäksi tulisi pystyä liikkumaan viikoissa eteen ja taaksepäin. Lukkarikoneeseen pitää tehdä monta pyyntöä ryhmätunnuksella hakua varten, täten tulee tallentaa ensimmäisen pyynnön *PHPSESSIID*-eväste. Kyseistä evästettä tarvitaan myös, jotta voimme selata eri viikkoja Lukkarikoneessa.

7.4.1 Ryhmätunnuksella haku

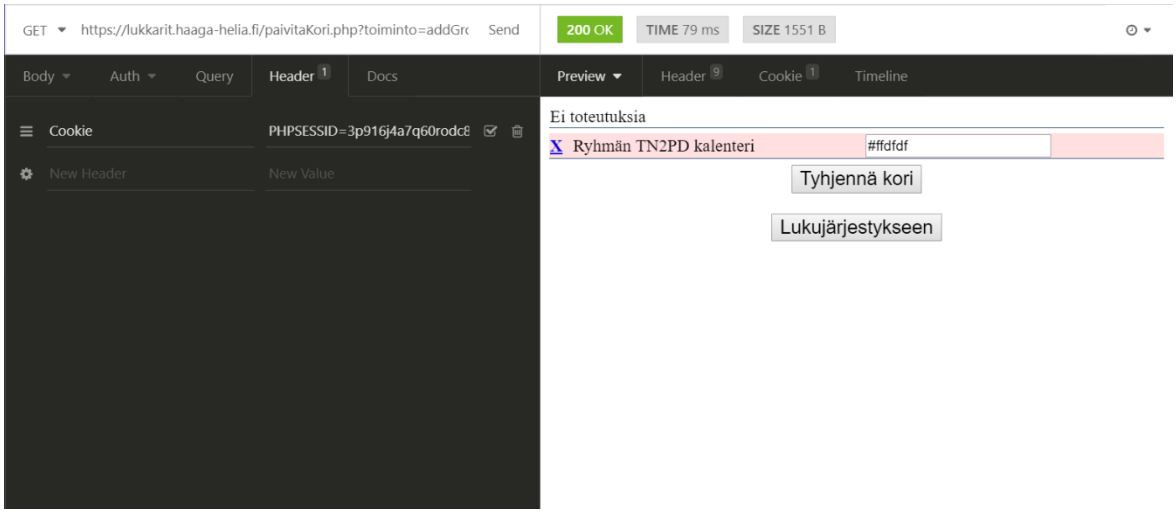
Seuraavassa on kuvattu miten pyynnöt tehdään. Tarkastellaan ensimmäiseksi ryhmätunnuksella hakua. Luodaan Lukkarirouter.js tiedosto ja luodaan muuttuja *lukkariRouter*. Määritellään lukkariRouterille GET-päätepiste, johon pyyntö voidaan lähettää:

```
lukkariRouter.get('/:tunnus', async (request, response) => {
  try {
    const tunnus = request.params.tunnus;
    const haeRyhmaKalenteri = await axios.post
    (`https://lukkarit.haaga-helia.fi
    /haeRyhmaKalenteri.php?ryhmanimi=${tunnus}`);

    let sessionCookie = (haeRyhmaKalenteri.headers
    ['set-cookie'][0]).split(';')[0] ;
    const config = {
      headers: { 'Cookie': sessionCookie }
    }
    const paivitaKori = await axios.get
    (`https://lukkarit.haaga-helia.fi/paivitaKori.php?
    toiminto=addGroup&code=${tunnus.toUpperCase()}&viewReply=true`, config);

    const kalenteri = await axios.get
    (`https://lukkarit.haaga-helia.fi/kalenteri.php`, config);

    response.send(`${sessionCookie}\n` + kalenteri.data);

  } catch(exception) {
    console.log(exception);
  }
});
```

Pyynnön parametrina tuleva tunnus talletetaan muuttujaan tunnus. Tämän jälkeen tehdään POST-pyyntö Lukkarikoneeseen kyseisellä tunnuksella ja talletetaan vastaus muuttujaan *haeRyhmaKalenteri*.

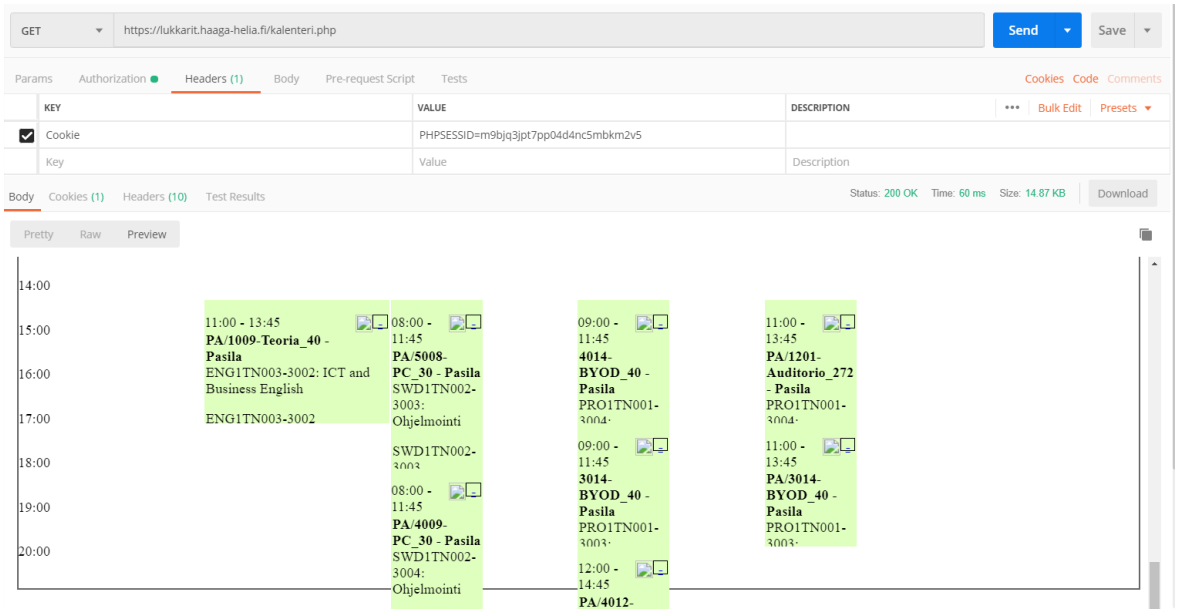
Evästeen saaminen vastauksen ylätunnisteesta (engl. header) toteutetaan string.split -funktiolla. Tämän jälkeen luodaan tulevia pyyntöjä varten config-muuttuja, jonka ylätunnisteisiin määritellään evästeen arvoksi tallentamamme eväste.

Seuraavaksi päivitetään tulokset GET-pyynnöllä paivitaKori.php sivustolle. Pyyntöön on nyt liitetty edellä tehty config. Kyseinen pyyntö palauttaa vastauksen, joka näyttää kutakuinkin tältä (kuva 10).



Kuva 10. paivitaKori.php GET pyynnön esimerkkivastaus

Joudumme kuitenkin luomaan vielä yhden pyynnön, josta saamme itse kalenterin datan. Tämä toteutuu lähettämällä GET-pyyntö Lukkarikoneen kalenteri.php päätepisteeseen. Tämän pyynnön vastauksessa saamme itse kalenterin tiedot. Viimeisenä palautamme omassa vastauksessa evästeen, sekä kalenterin datan eli kalenteri.php sivuston HTML:n. Esimerkkivastaus GET-pyynnölle (kuva 11).



Kuva 11. kalenteri.php GET-pyynnön esimerkkivastaus

Toteutus saattaa vaikuttaa hieman epämääräiseltä. Jos kuitenkin ohittaisimme jonkin näistä pyynnöistä, ei Lukkarikoneen kalenteri.php sivustolla näkyisi haluamiamme kurs-seja. Jokainen pyyntö on kriittinen, jotka myös Lukkarikone tekee aina ryhmätunnuksella hakiessa.

7.4.2 Viikkojen selaaminen

Seuraavassa on kuvattu miten viikkojen selaaminen onnistuu. Määritetään uusi päätepiste lukkariRouterille, joka vastaa viikkojen selaamisesta. Uuden päätepisteen koodi:

```
lukkariRouter.get('/:week/:cookie', async (request, response) => {
  try {
    const week = request.params.week;

    const cookie = request.params.cookie;

    const config = {
      headers: { 'Cookie': cookie }
  }
    const kalenteri = await axios.get
    (`https://lukkarit.haaga-helia.fi/kalenteri.php?date=${week}`, config);

    response.send(kalenteri.data);

  } catch(exception) {
    console.log(exception);
  }
});
```

Palvelimelle tulevassa GET-pyynnössä on parametrina jokin viikko, jonka kurssit haluamme hakea. Tämän lisäksi parametrina tulee käyttäjän eväste, jolla hän on suorittanut ryhmähaun aikaisemmin. Luodaan taas config, jossa on evästeemme. Tehdään uusi pyyntö kalenteri.php päätepisteeseen, jossa parametrina on kyseinen viikko. Palautetaan sitten kyseisen viikon kalenterin data vastauksessa.

Lisätään viimeiseksi index.js-tiedostoon muuttujalle app määritys, jotta sillä on käytössä myös lukkariRouterin päätepisteet:

```
app.use('/api/lukkari', lukkariRouter);
```

8 Käyttöliittymä

Luodaan käyttöliittymälle React.js pohja. Tähän kätevä työkalu on create-react-app kirjasto (toimii komentona * npx create-react-app <ohjelman nimi> *), joka huolehtii monista asioista koodaajan puolesta. Kyseinen kirjasto luo pohjan, jonka päälle on helppo lähteä rakentamaan, eikä tarvitse huolehtia sovelluksen kääntämisestä selaimelle ymmärrettävään versioon. Komento *npm run build* luo yhden kansion, josta löytyy käyttöliittymän koodi minimoituina JavaScript-tiedostoina, joita selain ymmärtää.

Ladataan sovelluksen sisälle tarvitsemamme riippuvuudet, jotka ovat jo tiedossa.
* npm install **axios** *
* npm install **react-bootstrap** *
* npm install **graphql** *
**** npm install ***apollo-boost ***

Lisätään package.json-tiedostoon myös määritys siitä, että palvelinpuoli toimii portilla 3001. Tehdään kyseiseen tiedostoon proxy:

```
"proxy": "http://localhost:3001"
```

Nyt axiosin avulla lähtevät pyynnöt ohjautuvat portille 3001, kun käyttöliittymämme taas toimii portilla 3000.

Create-react-app luo meille monia tiedostoja automaattisesti. App.js-tiedosto toimii projektin pääkomponenttina, johon kootaan kaikki tulevat komponentit. Index.js sen sijaan hoitaa itse generoinnin index.html-tiedostoon, jotta se ei sekoita meitä App.js-tiedostossa. Luodaan src-kansion sisään vielä components, img ja services -kansiot.

Index.js tiedosto on hyvin yksinkertainen:

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

ReactDOM.render(<App />, document.getElementById('root'));
```

Juurikomponentti App generoidaan siis index.html:n -elementtiin, jolla on id *root*.

8.1 Navigointi

Yksi verkkosivuston tärkeimmistä toiminnoista on tarjota linkit koulunkäyntiä varten käytettyihin sivustoihin. Näitä ovat mm. Moodle, Peppi ja Lukkarikone. Luodaan kaksi funktionaalista komponenttia Nav ja NavItem. Ensimmäisen sisällä on yksittäiset NavItem-komponentit, jolle on viety propseina kuva, overlaytrigger ja linkki. NavItem-komponentti näyttää tältä:

```
const NavItem = props => (
  <OverlayTrigger
    placement="bottom"
    overlay={(
      <Tooltip id="hhtooltip">
        {props.trigger}
      </Tooltip>
)}
  >

    <NavLink
      className="navLink"
      href={props.link}
      rel="noopener noreferrer"
      target="_blank"
    >

      <Image
        className="navImg"
        src={props.image}
        roundedCircle
      />
    </NavLink>
  </OverlayTrigger>
);
```

Overlaytrigger on käyttäjäkokemusta parantava mukava pieni lisäys. Hiiren ollessa kuvan päällä, sen alapuolelle tulee esiin teksti missä on sovelluksen nimi. Kyseisen komponentin sisällä olevat komponentit <NavLink> ja <Image> ovat React-Bootstrapin tarjoamia. Tämänkin komponentin voisi vielä pienentää pienempiin osiin, mutten näe siihen tarvetta.

Nav-komponentin sisään laitetaan NavItemit, joita tarvitaan. Esimerkissä näkyy komponentin koodi, josta on riisuttu kaikki paitsi yksi NavItem pois (<Nav> on jälleen React-Bootstrapin tarjoama).

```
const Navigation = () => {
  return (
    <Nav className="justify-content-center">
      <NavItem
```

```
        id="moodleLink"
        trigger="Moodle"
        link="https://hhmoodle.haaga-helia.fi"
        image={moodlePicture}
      />
    </Nav>
    )
}
```

Sivustolle saadaan näkyviin nyt navigointi sen yläreunaan (kuva 12.)

Kuva 12. Sivuston navigointinäkymä

8.2 Kielen valinta

Jotta sovellus palvelisi myös muita kuin suomenkielisiä henkilöitä, on siihen lisättävä toiminto joka mahdollistaa sivuston näkemisen myös englanniksi. Ladataan meitä avustava riippuvuus localized-strings, jonka avulla voimme määrittää sivustolla olevalle tekstille englanninkieliset vastineet:
* npm install **localized-strings** *

Luodaan uusi tiedosto Langstrings.js, johon määritellään muuttuja strings, joka luo uuden localizedstrings-objektin. Esimerkissä näkyy kaksi sanaa, jotka määritellään strings-muuttujaan:

```
const strings = new LocalizedStrings({
  en: {
    lunchmenu: 'lunch menu',
    notAvYet: 'Not available yet',
  },
  fi: {
    lunchmenu: 'lounaslista',
    notAvYet: 'Ei saatavilla vielä',
  },
});
```

Nyt voimme käyttää näitä sanoja komponenteissa tuomalla strings-muuttujan ja kirjoittamalla esim *{strings.lunchmenu}*. Luodaan App.js-komponentille tila, joka kertoo valittuna olevan kielen. Aloituskielenä toimii selaimen oletuskieli.

```
this.state = { lang: strings.getLanguage() }
```

Nyt voimme lisätä Header-komponenttiin napin, jonka sisällä on kuva Englannin tai Suomenlipusta. Komponentille tuodaan siis äskeisessä näytetty tila, jonka avulla päätetään kumpi lippu pitää näyttää:

```
const langToShow = selectedLang === 'fi' ? engFlag : finFlag;

<button
  type="button"
  className="langButton"
  name="lang"
  onClick={() => { handleLangClick(); }}
>
  <img name="lang" src={langToShow} alt="English" />
</button>
```

Napin eli lipun painaminen laukaisee funktion handleLangClick-kutsun, joka sijaitsee App.js-komponentissa:

```
handleLangClick = () => {
  strings.setLanguage(this.state.lang === 'fi' ? 'en' : 'fi');
  this.setState({ lang: this.state.lang === 'fi' ? 'en' : 'fi' })
}
```

Yllä muutetaan strings muuttujan kieli lipun painalluksen mukaiseksi. Vaihdetaan myös komponentin *lang* tila samaksi. Tämä laukaisee sivuston uudelleengeneroinnin, jonka mukana kaikki sivun tekstit vaihtuvat lipun painalluksen mukaiseksi.

8.3 Lukkari

Lukkarin lisääminen sivustolle on hieman hankalampi tehtävä. Koska saamme palvelinpuolelta HTML:ää, se pitää generoida itsessään sivulle. Kuten teoriaosuudessa mainittiin, React julkaisi uuden version helmikuussa 2019 jossa esiteltiin koukut. Näillä voimme luoda tiloja myös luokkakomponentteihin. Toteutetaan lukkari siis funktionaalisena komponenttina, joka käyttää koukkuja.

Luodaan uusi komponentti Lukkari.js ja tuodaan käyttöömme tarvitsemamme kirjastot. Komponentista tulee melko laaja, joten esittelen vain mielenkiintoisimmat kohdat. Komponentin juureen määritellään tilat joita tarvitsemme:

```
const [waitMessage, setMessage] = useState('');
const [lukkariState, setLukkari] = useState('<div></div>');
const [lukkariCookie, setCookie] = useState(null);
const [weekLukkari, setWeek] = useState(new Date());
const [groupId, setGroupId] = useState('');
```

Alustamme siis viisi eri tilaa, joita tulemme tarvitsemaan. *waitMessage* kertoo kun lukkarin haku on käynnissä. *lukkariState*-tilaan tallennamme HTML-vastauksen palvelimelta. *lukkariCookie* säilyttää meidän evästeen ja *weekLukkari* viikon, joka on lukkarissa valittuna. *groupId* pitää sisällään ryhmätunnuksen hakukentän arvon.

Luokkakomponenteissa *componentDidMount*-funktio tekee generoinnin aina kun sivuston DOM:iin ladataan kyseinen komponentti. Uuden version myötä *useEffect*-funktio ajaa samaa asiaa funktionaalisissa komponenteissa. useEfffect-funktiossa on myös muita ominaisuuksia, mutta tarvitsemme nyt vain componentDidMount:in vastinetta. Seuraavassa on kuvattu miten tämä tapahtuu:

```
useEffect(() => {
  let id = 'tn2';
  const userLukkari = window.localStorage.getItem('lukkariTunnus');
  if (userLukkari) {
    id = userLukkari;
  }
  lukkariService.findByGroupId(id)
    .then((response) => {
      setLukkari(response);
      setCookie(response.split('\n')[0]);
      setMessage(`${strings.timetablefor} ${id} ${strings.lukkari}`);
    })
    .then(() => {
      highLightThisDayLukkari();
      scrollIntoCurrentDayLukkari();
      setMoodleCourseTarget();
    })
    .catch(() => {
      setLukkari('<div><h3>There was an error fetching data from
      lukkarikone</h3></div>');
    });
}, []);
```

useEffectin sisällä tarkistamme aluksi, onko käyttäjän selaimen paikallisessa varastossa (engl. localstorage) aiempaa hakua vastaava tunnus. Jos tällainen löytyy, teemme ensimmäisen haun sivustolle mentäessä tämän mukaan, muuten haemme tyhjän lukujärjestyksen. Näemme myös kuinka helppoa on muuttaa koukkujen tilaa verrattuna luokkakomponentin *setState*-funktioon.

Lopussa on vielä lisätoimintoja: *setMoodleCourseTarget*-funktiokutsun avulla painaessa lukkarissa olevaa kurssia, selain avaa uudessa välilehdessä kyseisen kurssin Moodle sivun.

Funktion toisena parametrina on tyhjä taulukko. Tällöin efekti suoritetaan vain yhden kerran sivua käyttäessä. Ilman tyhjän taulukon antamista efektiä suoritettaisiin jatkuvasti, kun DOM:issa ilmenee muutoksia.

Lukujärjestystä hakiessa teemme uuden *lukkariService.findGroupById(id)*-haun. Samalla poistamme selaimen paikallisesta varastosta vanhaa hakua vastaavan tunnuksen ja lisäämme tilalle uuden. Tarkastellaan kuinka paikalliseen varastoon tallentuu ryhmätunnus merkkijonona (kuva 13).

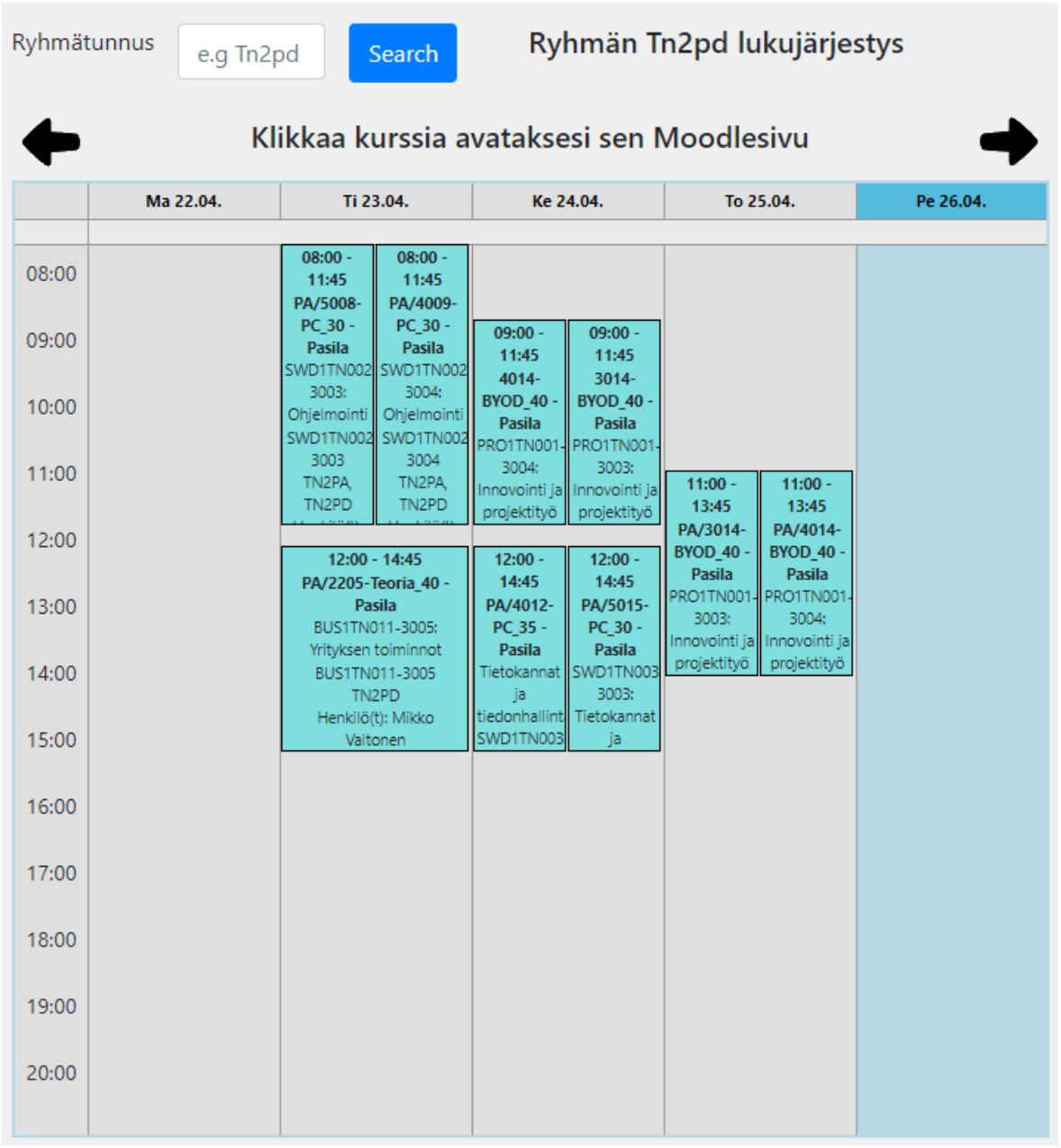
```
> window.localStorage.getItem('lukkariTunnus')
"tn2pd"
```

Kuva 13. Selaimen paikalliseen varastoon tallentuva lukkaritunnus

Näiden toimintojen lisäksi komponentissa on vielä lukkarin viikonvaihdosta vastuussa oleva funktio ja pientä säätöä vastauksen HTML:n kanssa:

```
let changedHtml = lukkariState.replace('<table ',
  '<table class="table table table-responsive-lg
  table-striped table-bordered table-hover"');
changedHtml = changedHtml.replace(/src/g, 'ref');
changedHtml = changedHtml.replace(/viewEvent/g,
  'console.log(event.target.textContent);console.log');
```

Tässä lisäämme taulukolle class-ominaisuuksia, jolloin React-Bootstrap tulee avuksi. Kahdessa myöhemmässä teemme regex:in avulla muutoksia HTML:ään, jotta saadaan virheitä aiheuttavat kohdat poistettua. Alla on kuva miltä lukkari näyttää sivustolla (kuva 14).



Kuva 14. Lukujärjestys sivustolla

Lukkarin toteutus ei ole tyylikkäin mahdollinen koodin puolesta. Lukkarikone ei kuitenkaan tarjoa parempaa rajapintaa, joten tämä on paras mahdollinen ratkaisu jonka löysin. Jos Lukkarikone tarjoaisi tulevaisuudessa kurssit REST-rajapinnassa, voisi toteutuksesta tehdä hieman helpomman. Tällöin olisi myös mahdollista toteuttaa lukkarin tallennus tietokantaan käyttäjittäin.

8.4 Juhlat

Toteutetaan seuraavaksi juhlien esittäminen sivustolla. Juhlat haetaan apolloClient:iä ja GraphQL:ää käyttäen. Tämä toteutetaan services-kansion Bailataan.js-tiedostossa. Esitellään tiedoston sisältö:

```
const client = new ApolloClient({
  uri: '/api/graphql',
});

const query = gql`
{
    allEvents {
        name,
        id,
        dateActualFrom,
        availability
    }
}
`;

const getAllKideApp = async () => {
  const response = await client.query({ query });
  return response.data.allEvents;
};

export default { getAllKideApp };
```

getAllKideApp-funktio palauttaa kaikkien tapahtumien nimet, id:t, päivämäärät sekä lippujen jäljellä olevan määrän.

Käytetään jälleen funktionaalista komponenttia ja koukkuja. Luodaan Bailataan.js komponentti. Juhlat haetaan taas komponentin generoituessa DOM:iin useEffect-funktion avulla:

```
const [events, setEvents] = useState(null);

useEffect(() => {
  bailataanService.getAllKideApp()
    .then((events) => {
      setEvents(events);
    });
}, []);
```

Jotta sivusto pysyy järkevänä, haluamme näyttää vain yhden päivän juhlat kerrallaan. Tehdään viikonpäiväpainike toiseen komponenttiin, joka kertoo valitun päivän. Tämä tieto talletetaan juurikomponentin App.js tilaan, koska tulemme tarvitsemaan tätä myös ruokalistoja näytettäessä. Keskityn tässä suurimpien komponenttien luontiin, joten viikonpäivänpainikkeiden luomista ei käydä läpi.

Tuodaan nyt komponentille muuttuja selectedDay propseina, joka kertoo valitun päivän. Alla on kuvattu, miten suodatus tapahtuu:

```
let filteredEvents = [];
```

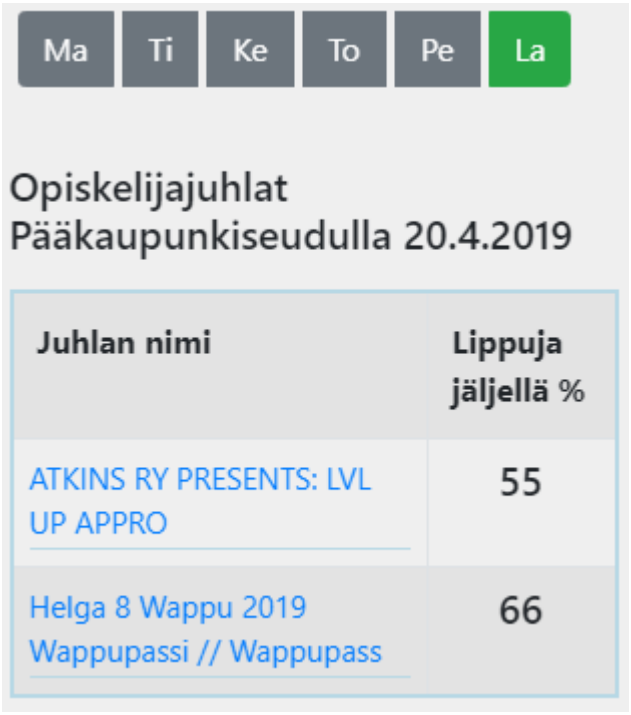
```
if (events) {
  filteredEvents = events.filter(e =>
    e.dateActualFrom.includes(nextDayFromSelectedDayAsJSON))
    .map(event => (
      <tr key={event.id}>
        <td>
          <a
            className="cool-link"
            rel="noopener noreferrer"
            target="_blank"
            href={`https://bailataan.fi/events/${event.id}`}
          >
            {event.name}
          </a>
        </td>
        <td className="ticketsLeft">
          <h5>{event.availability}</h5>
        </td>
      </tr>
    ));
}
```

Aluksi alustetaan taulukko, johon tullaan tallettamaan suodatetut juhlat. Tämän jälkeen käytetään JavaScriptin filter ja map -funktioita, jotka suodattavat ja käyvät läpi kaikki juhlat. Tämä kaikki tehdään if-lauseen sisällä, jotta ei törmätä virheeseen ennen kuin juhlat on saatu palvelimelta. Filteredevents-taulukkoon jää nyt jäljelle vain oikean päivän juhlat JSX-elementteinä. Näihin on myös lisätty href-attribuutti, jolloin juhlaa painaessa päädytään kyseisen juhlan sivustolle Bailataanissa. Kuten ylhäällä näkyy, nämä tiedot (name, id, dateActualFrom ja availability) ovat ainoat joita tarvitsemme. Olisi ollut hieman turhaa käyttää REST-rajapintaa, joka palauttaa kaikki kentät. Voimme jatkossa myös muokata helposti kyselyn muotoa, jos tarvitsemme uusia kenttiä käyttöön.

Komponentti palauttaa seuraavat tiedot hieman riisuttuna:

```
<Table striped bordered hover className="table">
  <tbody>
    <tr>
      <th>{strings.partyName}</th>
       <th>
         {strings.ticketsLeft}
         {' '}
       </th>
    </tr>
    {filteredEvents}
  </tbody>
</Table>
```

Seuraavassa kuvassa esitellään miltä tulos näyttää sivustolla (kuva 15). Painamalla viikonpäivää, saamme näkyviin kyseisen päivän juhlat. Lisäksi se viikonpäivä joka on meneillään, on tehostettu vihreällä taustalla. Sunnuntai on jätetty pois, sillä silloin on harvemmin juhlia



Kuva 15 Juhlat verkkosivustolla

8.5 Ruokalistat

Ruokalistojen toteutus tapahtuu samalla tavalla kuin juhlien, mutta data saadaan hieman REST:iä muistuttavasta rajapinnasta. Toteutus ja ruokalistojen data tallennetaan taas funktionaalisen komponentin useState-tilaan. useEffect:issä on nyt kuitenkin yksi pieni ero edellisiin verrattuna:

```
const [pasilaAmicaFood, setPasila] = useState([]);

useEffect(() => {
  amicaService.getAllPasila(language)
    .then((foodList) => {
      setPasila(foodList.LunchMenus);
    });
}, [language]);
```

useEffect-funktiolle on annettu toisena parametrina language-muuttuja, joka on juurikomponentin tila *lang*. Tämä kertoo useEffect-funktiolle, että sen pitäisi tehdä uudelleengenerointi, jos muuttujan language arvo vaihtuu. Tämä käytännössä tarkoittaa, että kieltä vaihdettaessa tehdään palvelinpuolelle uusi pyyntö englannin tai suomen kielellä. Toteutuksessa on myös Haagan ja Malmin ruokalistat, jotka on jätetty pois esittelystä.

Services kansion Amica.js tiedoston getAllPasila-funktio näyttää tältä:

```
const getAllPasila = async (lang) => {
  const response = await axios.get(`${urlPasila}/${lang}`);
  return response.data;
};
```

Olemme luoneet sivustolle napit, jotka vaihtavat tilaa *location* kyseisen kampuksen mukaiseksi. Switch case -lausekkeella tarkistetaan, mikä ruokalista kuuluu näyttää milloinkin sivustolla:

```
const listToShow = () => {
  switch (selectedLocation) {
    case 'Malmi':
      return malmiAmicaFood;
    case 'Pasila':
      return pasilaAmicaFood;
    case 'Haaga':
      return haagaAmicaFood;
    default:
      return pasilaAmicaFood;
  }
};
```

Ruokalistojen data koostuu hierarkkisesti monesta tasosta, joten ruokalistalle on luotu kaksi alikomponenttia. Ensimmäinen alikomponentti saa suodatetun ruokalistan juurikomponentilta. Alikomponentin sisällä luodaan vielä uusi alikomponentti RuokalistaDiet, joka hoitaa allergeenien generoinnin:

```
const foodListToRender = foodList.Meals.map(meal => (
  <tr key={getRandomKey()}>
    <td>{meal.Name}</td>
    <td className="dietsStyle">
      {<RuokalistaDiet
        key={getRandomKey()}
        foodListDiet={meal.Diets}
      />}
    </td>
  </tr>
));
```

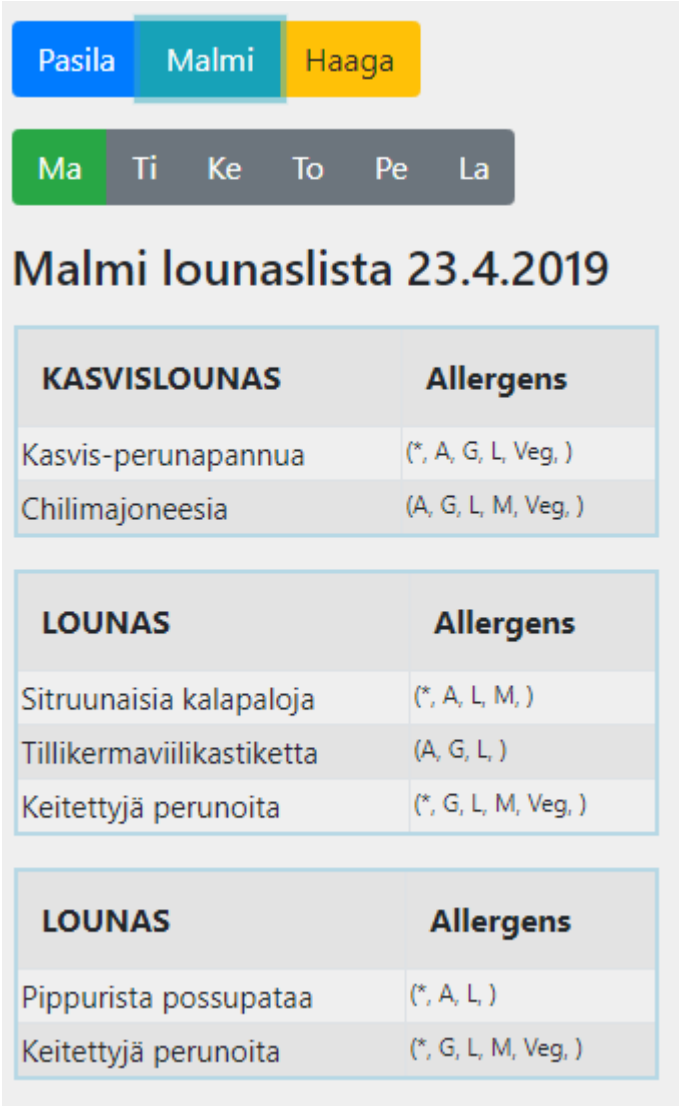
Koska datasta ei löydy näppärästi mitään uniikkia arvoa, pitää meidän määrittää avaimen arvoksi laskettu arvo jonka voidaan olettaa olevan uniikki.

```
const getRandomKey = () => Math.random().toString(36).substring(7);
```

RuokalistaDiet komponentti taas näyttää seuraavalta:

```
const RuokalistaDiet = ({ foodListDiet }) => (
  <div>
      (
    {foodListDiet.map(diet => `${diet}, `)}
)
  </div>
);
```

Seuraavassa kuvassa esitellään miltä tulos näyttää sivustolla (kuva 16). Samat viikonpäivänapit kuin juhlien suodattamisessa, toimivat myös ruokalistojen suodattamisessa.



Kuva 16. Ruokalistanäkymä verkkosivustolla

8.6 Testaus Cypresillä

Nyt kun sivusto on valmis, määritetään sille E2E-testejä. Cypress on tähän hyvä työkalu, sillä se toimii aidosti selaimessa ja sillä voimme myös manipuloida AJAX-pyyntöjä. Ladataan Cypress ja lisätään package.json-tiedoston scripts kohtaan meitä avustava komento, jotta saamme ajettua Cypressin helposti:

* npm install –save-dev **cypress** *

```
"cypress:open": "cypress open",
```

Voimme nyt kirjoittaa ensimmäiset testimme. Tämä tapahtuu Cypress-kansion sisällä olevaan integration-kansioon. Luodaan uusi tiedosto hhapp.spec.js johon testit kirjoitetaan.

Testejä on monia ja näytän tässä vain muutaman niistä. Lukkaria testattaessa voidaan hakea uniikilla ryhmäkoodilla lukkaria ja tarkistaa, löytyykö sivustolta kyseisen ryhmätunnuksen sisältämää kurssia.

```
context('LukkariTests', () => {
  beforeEach(() => {
    cy.visit('http://localhost:3000');
  });

  it('should be able to search tn2pd timetable and find PRO1TN001-3003
course', () => {
    cy.get('#groupInput')
      .type('tn2pd');
    cy.get('#searchButton')
      .click();
    cy.contains('PRO1TN001-3003');
  });
```

Selaimen paikallista varastoa testatessa tarkistetaan että ryhmätunnuksella haku tallentaa varastoon lukkariTunnus-esineen (engl. item), joka saa ryhmän tunnuksen arvokseen:

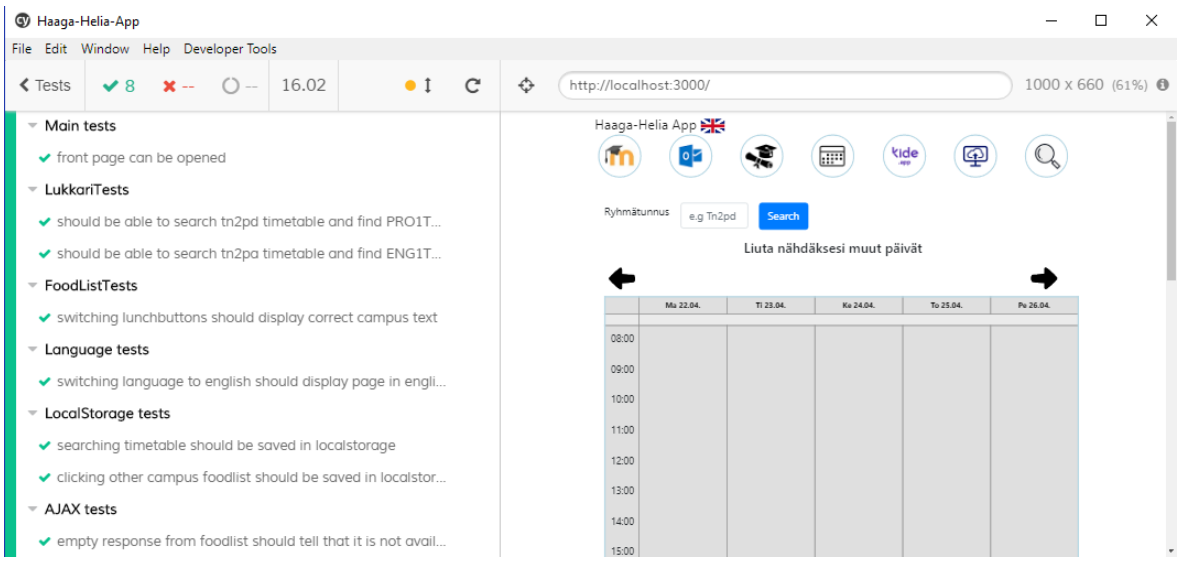
```
context('LocalStorage tests', () => {
  beforeEach(() => {
    cy.visit('http://localhost:3000');
  });

  it('searching timetable should be saved in localstorage', () => {
    cy.get('#groupInput')
      .type('tn1pb1');
    cy.get('#searchButton')
      .click()
      .should(() => {
        expect(localStorage.getItem('lukkariTunnus')).to.eq('tn1pb1');
      });
  });
```

Viimeisenä esitellään vielä testi, jossa manipuloimme /api/amica päätepisteistä tulevia pyyntöjä palauttamaan tyhjän listan. Tämän pitäisi näyttää ruokalistojen kohdalla *Ei saatavilla vielä* tekstin:

```
context('AJAX tests', () => {
  beforeEach(() => {
```

```
    cy.visit('http://localhost:3000');
  });

  it('empty response from amica backend should tell that it is not
  availaible yet', () => {
    cy.server();
    cy.route({
      method: 'GET',
      url: '/api/amica',
      response: [],
    });
    cy.contains('Ei saatavilla vielä');
  });
});
```

Nyt kun ajamme komennon **npm run cypress:open,** aukeaa Cypressin käyttöliittymä, jossa näemme projektin testitiedostot. Painetaan hhapp.spec.js-tiedostoa, jonka jälkeen aukeaa ikkuna jossa testit ajetaan. Ikkunassa näkyy vasemmalla puolella ajettavat testit ja oikealla selain jossa ne ajetaan (kuva 17). Testien suorittamisen jälkeen nähdään ne jotka onnistuivat ja jotka eivät. Cypress antaa mahdollisuuden myös tarkastella selaimen DOM-rakennetta testien jokaisen kohdan välillä.



Kuva 17. hhapp.spec.js Cypress testien näkymä ajon jälkeen

9 Julkaisu

Julkaisu on melko yksinkertainen prosessi koodin ja ohjelman kasaamisen kannalta. Palvelimen konfigurointi oikein saattaa tuottaa hieman enemmän ongelmia. Luodaan nyt build-kansio käyttöliittymän projektista. Tämä tapahtuu ajamalla komento **npm run build**. Build-kansio ilmestyy hetken kuluttua projektin juureen.

Viedään edellä saatu kansio palvelinpuolen projektin juureen, sekä kirjoitetaan palvelinpuolen index.js-tiedostoon seuraava rivi, jolla saamme käyttöliittymän käyttöön:

```
app.use(express.static('build'));
```

Luodaan vielä .env-tiedosto, johon voimme määritellä ympäristömuuttujia. Tänne liitämme portin numeron jota haluamme käyttää. Config.js tiedostossa sen sijaan on määritelty port-muuttuja käyttämään env-tiedostossa määriteltyä PORT:ia. Index.js-tiedosto muuttuu seuraavasti:

```
server.listen(config.port, () => {
    console.log(`Server running on port ${config.port}`);
})
```

Config.js-tiedostossa on seuraava rivi:

```
let port = process.env.PORT
```

Ja env-tiedosssa:
PORT=3001

Nyt kun kloonaamme projektin ulkoiselle palvelimelle, teemme uuden env-tiedoston (joka ei tule Githubista, sillä se on liitetty tiedoston .gitignore listalle) jossa määrittelemme portille arvon 8080.

9.1 Palvelimelle

Jotta nettisivun saadaan julkiseen käyttöön, se on laitettava luotettavalle palvelimelle. Minun valintani on DigitalOcean:in palvelin, sillä opiskelijana olen saanut sen käyttöön 50€ bonuksen. Käytän 5€ kuukaudessa maksavaa palvelinta. Olen ostanut myös verkkotunnuksen *hhapp.info*.

Palvelimella pitää tehdä jonkin verran konfigurointia, josta näytän vain osan. Aluksi tehtiin uusi käyttäjä palvelimelle ja ladattiin nginx (http palvelin, jonka avulla sovellus toimii internetissä).

Lisätään nyt palvelimella /etc/nginx/sites-enabled/default -tiedostoon määritys, joka ohjaa kaikki http ja https -pyynnöt meidän ohjelmalle (eli portille 8080). Tämän lisäksi kerrotaan, mitkä ovat minun palvelimeni käytössä olevat verkkotunnukset (kuva 18).

```
server_name www.hhapp.info hhapp.info;

location / {
        proxy_pass http://localhost:8080;
```

Kuva 18. /etc/nginx/sites-enabled/default tiedosto

Olemme ottaneet myös käyttöön UFW-palomuurin, joka sallii kaikki palvelimelle tulevat http-yhteydet. Kaikki http-pyynnöt ohjataan myös käyttämään https-protokollaa.

Nyt kaikki on kunnossa ja sovelluksemme pitäisi toimia palvelimen osoitteessa sen käynnistyessä. Siirrytään palvelimella käyttäjän kotikansioon ja kloonataan projekti Githubista. Tämän jälkeen luodaan env-tiedosto, jonka sisältö on yksinkertaisesti:
PORT=8080

Tarvitsemme vielä moduulit projektia varten, sillä niitä ei ollut Githubissa. Ajetaan siis seuraava komento, jolla saamme kaikki package.json-riippuvuudet käyttöömme:
* npm **install ***

Ennen sovelluksen käynnistämistä tehdään vielä verkkotunnukselle tarvittavat määrittelyt (kuva 19).

DNS records

Type	Hostname	Value
A	www.hhapp.info	directs to 209.97.182.163
A	hhapp.info	directs to 209.97.182.163

Kuva 19. Verkkotunnukselle määrittelyt

Olemme siis määritelleet sekä www alkuiset, että ilman www alkua tulevat pyynnöt menemään palvelimelle.

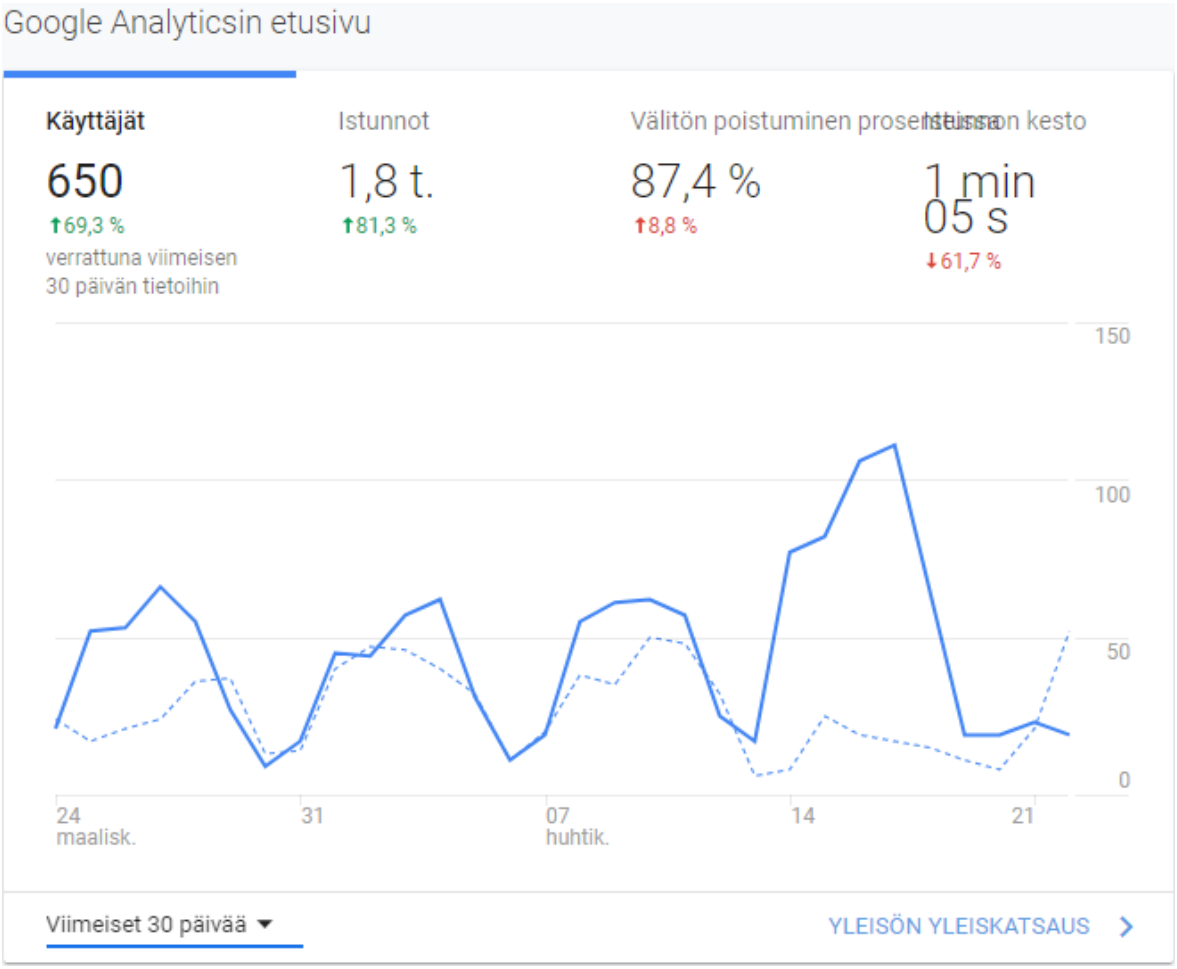
Nyt kun käynnistämme sovelluksen ja menemme sivustolle https://hhapp.info, on näkymä seuraava (selainta hieman zoomatessa ulos) (kuva 20).

Kuva 20. Tuotannossa oleva verkkosivu

9.2 Seuranta

Verkkosivuston seuranta on mielenkiintoista, sillä sen avulla tiedämme kuinka monen oppilaan käytössä sivusto on. Tuotannossa olevassa versiossa on Google Analytics:in antama <script>-kenttä index.html sivustolla, joka seuraa sivuston aktiivisuutta.

Verkkosivusto on ollut jo julkisena hetken, joten voimme tarkastella sen käyttäjämääriä. Alla on kuvattu viimeisen kuukauden tietoja (kuva 21).



Kuva 21. Google Analytics viimeiset 30 päivää

Sivustolla voidaan todeta olevan aktiivisuutta. Kuukauden aikana on tullut lähes kaksi tuhatta istuntoa 650:ltä eri käyttäjältä. Tähän ei oteta mukaan omasta IP-osoitteestani tulevia käyntejä, sillä olen tehnyt siihen tarvittavan määrittelyn Analytics sivustolla.

Kuviosta voidaan myös tulkita että käynnit painottuvat selvästi arkipäiville, jolloin käyttäjät luultavasti vierailevat sivustolla koulupäivän aikana. Viikonloppuna on hiljaisempaa, joka on loogista ottaen huomioon sivuston käyttötarkoituksen.

10 Pohdinta

Projekti onnistui mielestäni kaiken kaikkiaan hyvin. Loin toiminnaltaan yksinkertaisen ja selkeän käyttöliittymän, joka helpottaa Haaga-Helian opiskelijoiden arkea heidän hakiessaan tietoa mm. lukujärjestyksistä, ruokalistoista sekä juhlista. Toki on monia asioita joita tekisi mieli lisätä ja muokata paremmaksi, mutta se on ikuinen prosessi. Olen tyytyväinen, että sain käytettyä uusia teknologioita joista en tiennyt ennen ollenkaan. Cypress, GraphQL ja Reactin koukut olivat aivan vieraita asioita ennen projektin luontia. Tämä oli myös ensimmäinen todellinen projekti Reactilla ja Nodella, joten sain varmuutta näiden kirjastojen käyttöön.

Parhaiten onnistui mielestäni palvelinpuoli ja sen toiminta. Olen tyytyväinen kerran päivässä hakemiseen ja GraphQL-kyselyn luomiseen juhlien haussa. Käyttöliittymässä helpointa oli juuri juhlien luonti. Projektin varrella tuli myös ongelmia, joiden uniikkeihin ratkaisuihin olen ollut tyytyväinen. En ole tosin varma olisiko näihin ongelmiin ollut joitain parempia ja toiminnan kannalta viisaampia tapoja. Reactissa on niin paljon opittavaa, että jäi varmasti monia toimintoja käyttämättä, joista en tiedä tai joita en osaa vielä käyttää oikeaoppisesti

Eniten jäi ehkä harmittamaan lukkarin yksinkertaisuus ja käyttöliittymän tyylit. En ole kovin hyvä visuaalisesti, joka kostautui ehkä hieman käyttöliittymän estetiikassa. Koitin panostaa kuitenkin yksinkertaisuuteen ja helppokäyttöisyyteen. Tämä onnistui mielestäni kohtalaisen hyvin.

Lukkarista jäi uupumaan joitakin toimintoja, joita olisin halunnut lisätä. Olin alun perin tehnyt käyttäjien hallintaan jo toimivan palvelinpuolen ja alustanut myös käyttöliittymää siihen liittyen. Hylkäsin kuitenkin tämän, sillä lukkarien tallentaminen tuntui aika mahdottomalta tehtävältä.

Verkkosivustosta jäi käsittelemättä joitain osia. Niitä ovat mm. Header, muut linkit osio ja Footer. Css:ää en käynyt ollenkaan läpi, sillä se on aika yksinkertaista eikä siellä kummempia yllätyksiä ole. Responsiivisuutta ei myöskään käsitelty, vaikkakin sivusto on optimoitu myös tableteille ja puhelimille. Keskityin esittelyissä mielestäni olennaiseen.

Olen saanut tälle sivustolle jo melko hyvin käyttäjiä, joka on ollut mukava todeta. Projektin laajempi tavoite oli tehdä sivusto, joka on avuksi opiskelijoiden koulunkäynnissä. Tämän takia ajattelen, että pääsin tavoitteisiin melko hyvin.

10.1 Jatkokehitys

Vaikka projekti on tuotannossa, ei se ole mielestäni ikinä lopullisesti valmis. Lisäyksiä haluan tehdä periaatteessa niin kauan, kun niiden lisääminen auttaa jotakin henkilöä. Sivuston alalaidassa on palautelinkki, josta toivon saavani kehitysideoita ja palautetta.

Puhelimelle optimointi voisi jatkua vielä luomalla sovellus Play kauppaan ja Applen storeen saataville. Tämä avaisi myös mahdollisuuden luoda Android järjestelmille widget:in, joka voisi näyttää lukkarin kyseisen päivän esim. puhelimen aloitusruudulla. Toteutus voisi toimia WebView:in avulla, jolloin siihen ei tarvitsisi käyttää kovin paljoa aikaa.

Lukkareiden osalta haluaisin vielä parantaa käytettävyyttä. Tällä tarkoitan lukkarien jonkinlaista tallentamista tietokantaan ja omien tapaamisten lisäämistä kalenteriin. Oman REST-Rajapinnan rakentaminen olisi mahdollisuus, mutta erittäin työläs tapa. Jos Lukkarikone tulee tarjoamaan paremman rajapinnan tulevaisuudessa, niin tallentaminen olisi varmasti helpompaa.

Tämän lisäksi lukkareihin olisi hyvä saada mahdollisuus hakea yksittäisiä kursseja. Tämä on kuitenkin mielestäni relevanttia vain, jos tallentaminen onnistuu. Kukaan tuskin haluaa hakea kymmentä kurssia yksittäin joka kerta kun saapuu sivustolle.

Jenkinsin avulla jatkuva integraatio olisi mukava lisäys, jonka eteen olen jo tehnyt hieman työtä. Tällöin ei tarvitsisi joka kerta itse tehdä tiedostojen siirtoa palvelimelle. Riittäisi että tehtäessä uusi commit Githubiin, palvelin saisi uusimman version automaattisesti. Jenkins voisi myös suorittaa Cypress-testit, joiden onnistuessa uusin versio menisi tuotantoon.

Lukkarikoneen ollessa myös Metropolian ja Laurean käytössä, ei olisi mahdoton ajatus laajentaa sovellusta myös heidän käyttöönsä. Toki tällöin pitäisi luoda uudet sivustot, mutta niiden luonti olisi melko helppo tehtävä samankaltaisuuden takia. Tämä on kuitenkin realistinen ajatus vain, jos sivustosta saisi vielä hienomman ja käyttäjäystävällisemmän.

Lähteet**:**

Apollographql a. GraphQL Playground. Luettavissa: https://www.apollographql.com/docs/apollo-server/features/graphql-playground. Luettu: 29.4.2019.

Apollographql b. Subscriptions. Luettvissa: https://www.apollographql.com/docs/react/advanced/subscriptions. Luettu: 10.5.2019.

Buna S. Learning GraphQL and Relay. Packt Publishing. Birmingham. Luettavissa: https://books.google.fi/books?id=j6XWDQAAQBAJ&printsec=frontcover&source=gbs_atb#v=onepage&q&f=false. Luettu: 13.4.2019.

Biztalk360. What is Web Endpoint and how it can be monitored? Luettavissa: https://blogs.biztalk360.com/web-endpoint-monitoring-biztalk360/. Luettu: 29.4.2019.

Bundlephobia a 2019. react-bootstrap. Luettavissa: https://bundlephobia.com/result?p=react-bootstrap@1.0.0-beta.6. Luettu: 4.4.2019.

Bundlephobia b 2019. material-ui. Luettavissa: https://bundlephobia.com/result?p=material-ui@0.20.2. Luettu: 4.4.2019.

Cypress 2019. Testing has been broken for too long. Luettavissa: https://www.cypress.io/how-it-works/. Luettu: 11.4.2019.

Codeburst 2017. What the heck is callback? Luettavissa: https://codeburst.io/javascript-what-the-heck-is-a-callback-aba4da2deced. Luettu: 14.5.2019.

Developer Mozilla a 2019. Getting started: What's AJAX? Luettavissa: https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX/Getting_Started. Luettu 29.4.2019.

Developer Mozilla b 2019. Promise. Luettavissa: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. Luettu: 29.4.2019.

Ecma international. The JSON Data Interchange Syntax. Luettavissa: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf. Luettu 29.4.2019.

Fedosejev, A. 2015. React.js Essentials. Packt Publishing. Birmingham. Luettavissa: https://books.google.fi/books?hl=en&lr=&id=Rhl1CgAAQBAJ&oi=fnd&pg=PP1&dq=react.js&ots=JjwtoEBQSL&sig=_OXa4LZ5Fzfsh123sKH7SSzMQHA&redir_esc=y#v=onepage&q&f=false. Luettu: 2.4.2019

Fullstackopen 2019. Web-sovelluksen toimintaperiaatteita. Luettavissa: https://fullstackopen.com/osa0/web_sovelluksen_toimintaperiaatteita#single-page-app. Luettu: 29.4.2019.

Fullstackreact 2018. What is JSX?. Luettavissa: https://www.fullstackreact.com/30-days-of-react/day-2/. Luettu 29.4.2019.

Github a 2019. React-bootstrap/src/Table.js. Luettavissa: https://github.com/react-bootstrap/react-bootstrap/blob/master/src/Table.js Luettu: 4.4.2019.

Github b 2019. Mui-org/material-ui. Luettavissa: https://github.com/mui-org/material-ui. Luettu: 4.4.2019.

Graphql. Mutations. Luettavissa: https://graphql.org/learn/queries/#mutations. Luettu: 10.5.2019.

Iamturns 2018. Matt Turnbull Why Are You Still Using Yarn in 2018?. Luettavissa: https://iamturns.com/yarn-vs-npm-2018/. Luettu: 29.4.2019.

Kosev, G. 23.8.2015. ES7 async functions - a step in the wrong direction. Luettavissa: https://spion.github.io/posts/es7-async-await-step-in-the-wrong-direction.html. Luettu: 2.5.2019.

Lei K, Ma Y & Tan Z. 2014. Performance Comparison and Avaluation of Web Development Techonlogies in PHP, Python and Node.js. Luettavissa: https://www.researchgate.net/profile/Kai_Lei8/publication/286594024_Performance_Comparison_and_Evaluation_of_Web_Development_Technologies_in_PHP_Python_and_Nodejs/links/5c284b06458515a4c700b891/Performance-Comparison-and-Evaluation-of-Web-Development-Technologies-in-PHP-Python-and-Nodejs.pdf. Luettu: 10.4.2019.

Mardan A. Practical node.js. 2014. Apress New York.

Medium 2017. Gaurav Pandvia. Understanding JavaScript Function Executions – Call Stack, Event Loop, Tasks & more. Luettavissa: https://medium.com/@gaurav.pandvia/understanding-javascript-function-executions-tasks-event-loop-call-stack-more-part-1-5683dea1f5ec. Luettu: 9.4.2019.

Microsoft Azure. What is Middleware? Luettavissa: https://azure.microsoft.com/en-gb/overview/what-is-middleware/. Luettu: 5.5.2019.

Node green 2019. Node.js ES2015 Support. Luettavissa: https://node.green/ Luettu: 26.04.2019.

Nodejs 2019. ECMAScript 2015 (ES6) and beyond. Luettavissa: https://nodejs.org/en/docs/es6/. Luettu: 26.04.2019.

Reactjs a 2019. Components and Props. Luettavissa: https://reactjs.org/docs/components-and-props.html. Luettu: 2.4.2019.

Reactjs b 2019. Introducing Hooks. Luettavissa: https://reactjs.org/docs/hooks-intro.html. Luettu: 4.4.2019.

Restfulapi. HTTP Methods. Luettavissa: https://restfulapi.net/http-methods/. Luettu: 15.5.2019.

Sengupta D, Singhal M & Corvalan D. Getting Started with React. Packt Publishing. Birmingham.

Sitepoint. Tim Severien Yarn vs npm: Everything You Need To Know. Luettavissa: https://www.sitepoint.com/yarn-vs-npm/. Luettu: 29.4.2019.

Technopedia 2019. End-to-End Test. Luettavissa: https://www.techopedia.com/definition/7035/end-to-end-test. Luettu: 29.4.2019.

Techterms 2019. Runtime. Luettavissa: https://techterms.com/definition/runtime. Luettu: 29.4.2019

Vohr, S. 2.6.2018. Software Engineer. Asynchrony: Under the Hood. JSConf EU 2018. Seminaariesitys. Berlin. Katsottavissa: https://www.youtube.com/watch?v=SrNQS8J67zc. Katsottu: 2.5.2019.

Webopedia. API – application program intrerface. Luettavissa: https://www.webopedia.com/TERM/A/API.html. Luettu: 7.5.2019.

W3Schools a 2019. HTML Introduction. Luettavissa: https://www.w3schools.com/html/html_intro.asp Luettu: 29.4.2019.

W3Schools b 2019. JavaScript HTML DOM. Luettavissa: https://www.w3schools.com/js/js_htmldom.asp Luettu: 29.4.2019.

Yourdictionary 2019. Non-blocking. Luettavissa: https://www.yourdictionary.com/non-blocking. Luettu: 4.5.2019.