Mikko Laitila

Sisällönhallintasovelluksen luominen moderneilla full stack -tekniikoilla

Opinnäytetyö
Kevät 2017
SeAMK Tekniikka
Tietotekniikan koulutusohjelma

SeAMK
SEINÄJOEN AMMATTIKORKEAKOULU
SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

SEINÄJOEN AMMATTIKORKEAKOULU

Opinnäytetyön tiivistelmä

Koulutusyksikkö: Seinäjoen Ammattikorkeakoulu

Tutkinto-ohjelma: Tietotekniikan koulutusohjelma

Suuntautumisvaihtoehto: Tietoverkkotekniikka

Tekijä: Mikko Laitila

Työn nimi: Sisällönhallintasovelluksen luominen moderneilla full stack -tekniikoilla

Ohjaaja: Petteri Mäkelä

Vuosi: 2017 Sivumäärä: 52 Liitteiden lukumäärä: 0

Opinnäytetyö toteutettiin LEDiMedia Oy:n toiveesta. Työn tarkoituksena oli perehtyä moderneihin full stack -tekniikoihin, kuten ReactJS-, Redux-, NodeJS- ja ExpressJS-tekniikkaan ja toteuttaa niitä käyttäen uusi sisällönhallintasovellus. Sisällönhallintasovelluksesta oli tarkoitus saada nykyistä versiota dynaamisempi ja modernimpi.

Työn teoriaosuudessa käydään läpi tärkeimpiä työssä käytettäviä full stack -tekniikoita, sekä työssä käytettävä tietokanta MongoDB. Teoriaosuudessa esitellään kyseiset tekniikat ja niiden käyttämisestä näytetään lyhyitä esimerkkejä. Teoriaosuuden lopuksi käydään läpi web-ohjelman rakennetta pientä esimerkkisovellusta apuna käyttäen. Käytännön osuudessa esitellään tehtyä ohjelmaa ja sen toimintoja.

Työn lopputuloksena syntyi ohjelma, jolla käyttäjä voi luoda mainoskokonaisuuksia esimerkiksi yhtä jalkapallo-ottelua varten ja lisätä kyseiseen kokonaisuuteen näytettävät mediat.

Avainsanat: ReactJS, NodeJS, Front-end, Back-end, Web development, Sisällön-hallinta, Full Stack

SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

Thesis abstract

Faculty: School of Technology

Degree programme: Information Technology

Specialisation: Networking Technology

Author: Mikko Laitila

Title of thesis: Creation of a content management -web application using modern full stack -techniques

Supervisor: Petteri Mäkelä

Year: 2017 Number of pages: 52 Number of appendices: 0

This thesis was commissioned by a company called LEDiMedia. The aim was to look into the modern full stack -techniques like ReactJS, Redux, NodeJS, ExpressJS and develop a new content management -web application by using them. The new content management -web application would be more dynamic and modern than the current version.

The theory part of this thesis presents the most important full stack -techniques that are used, and MongoDB, which is the database used in this thesis. Also some short examples of how to use those techniques are shown. After the theory part, there is a chapter about the structure of a web application which is presented by using a small web application as an example. The practical part presents the web application that was made during this thesis.

The result of this thesis was a web application that allows users both to create an advertising complex, for example, for a football match and to add media to that complex.

Keywords: ReactJS, NodeJS, Front-end, Back-end, Web development, Content management, Full Stack

SISÄLTÖ

Kuva-, kuvio- ja taulukkoluettelo

Käytetyt termit ja lyhenteet

API	Ohjelmointirajapinta. Määritelmä, jonka mukaan eri ohjelmat voivat keskustella keskenään.
Asynkroninen	Tapahtumat eivät ole ajallisesti toisistaan riippuvaisia.
DNS	Internetin nimipalvelujärjestelmä. Muuntaa verkkotunnukset IP-osoitteiksi.
Full Stack	Sisältää front-endin eli käyttäjälle näkyvän osuuden ja back-endin eli palvelinpuolen.
HTTP	HyperText Transfer Protocol. Protokolla, jota selaimet ja WWW-palvelimet käyttävät tiedonsiirtoon.
JavaScript	Web-ympäristössä käytettävä komentosarjakieli, jolla lisätään web-sivuille dynaamisia toimintoja.
JSON	JavaScript Object Notation. Yksinkertainen avoimen standardin tiedostomuoto.
NoSQL-tietokanta	Perinteisestä relaatiomallista poikkeava tietokanta. Tauluja ei määritetä kiinteästi, skaalautuu hyvin.
Objekti	Joukko loogisesti yhteenkuuluvaa tietoa ja toiminnallisuutta.
Relaatiotietokanta	Tietokanta, jossa taulut määritellään. Taulujen välille muodostetaan yhteyksiä avaimilla.
Renderöinti	Näkymän luominen mallista tietokoneohjelman avula.
Socket	Tekniikka, jota käytetään kaksisuuntaiseen kommunikaatioon verkossa.
SQL	Kyselykieli, jolla voidaan tehdä hakuja, muutoksia ja lisäyksiä relaatiotietokantaan.

SSL	Secure Sockets Layer. Standardi turvallisuusteknologia, jolla muodostetaan salattu yhteys web-palvelimen ja selaimen välillä.
Syntaksi	Ohjelmointikielen oikeinkirjoitus. Kertoo sen, miltä ohjelmointikieli näyttää.
TCP/IP	Usean internetliikenteessä käytettävän protokollan yhdistelmä.
URL	Merkkijono, jota käytetään osoittamaan WWW-sivuja.
XML	Extensible Markup Language. Merkintäkielien standardi, jolla tiedon merkitys on kuvattavissa tiedon sekaan.

1 JOHDANTO

1.1 Työn tausta

Seinäjoella sijaitsevalla OmaSP-stadionilla on tällä hetkellä käytössä LEDiMedian kehittämä sovellus kentällä sijaitsevien laitamainosten hallintaan. Sovellus on ohjelmoitu toimimaan pelkästään OmaSP-stadionilla ja se on tehty vanhoilla tekniikoilla. Käyttöön tarvitaan uusi sovellus, jolla voitaisiin toteuttaa monipuolisia mainostuskokonaisuuksia erilaisilla näyttö- tai paneelikokoonpanoilla.

1.2 Työn tavoite

Työn tavoitteena on perehtyä uusimpiin full stack -tekniikoihin, erityisesti Reactiin ja NodeJS-kirjastoon, ja niitä käyttäen toteuttaa uusi, dynaaminen sisällönhallintasovellus monipuoliseen mainostamiseen.

1.3 Työn rakenne

Luvussa kaksi kerrotaan front-endin ohjelmoinnissa käytettävistä tekniikoista. Luvussa kolme kerrotaan back-endin ohjelmoinnissa käytettävistä tekniikoista. Luvussa neljä käydään läpi web-ohjelman rakennetta esimerkkisovelluksen avulla. Luvussa viisi esitellään opinnäytetyötä varten tehtyä ohjelmaa. Luvussa kuusi on yhteenveto ja pohdintaa.

1.4 Yritysesittely

LEDiMedia on Seinäjoella perustettu, valtakunnallisesti toimiva videotekniikkaan keskittyvä yritys. LEDiMedia vuokraa ja myy mainos- ja infonäyttöjä. Näyttölaitteiden lisäksi LEDiMedialla kehitetään erilaisia ohjelmia sisällönhallintaa varten. (LEDiMedia, [Viitattu 13.4.2017].)

2 FRONT-END-OSA

2.1 React

React on Facebookin kehittämä JavaScript-kirjasto käyttöliittymien ohjelmointia varten. Reactilla voidaan helposti rakentaa interaktiivisia käyttöliittymiä. Suunnittelemalla yksinkertaiset näkymät kutakin ohjelman tilaa kohden voidaan varmistaa, että React päivittää ja renderöi oikeat komponentit datan muutoksen myötä. (React, 2017.)

Reactin toiminta perustuu komponentteihin. Komponenteilla on omat tilat, joita kutsutaan stateiksi. Ohjelma koostuu useista eri komponenteista, joiden avulla voidaan muodostaa monipuolisia ja interaktiivisia käyttöliittymiä. Komponenttien logiikka on kirjoitettu JavaScriptillä, mikä mahdollistaa monipuolisen datan liikuttamisen ohjelmassa. Reactissa on myös mahdollista käyttää palvelinpuolen renderöintiä NodeJS-kirjaston avulla. Mobiiliohjelmien tekeminen onnistuu React Nativella. (React, 2017.)

2.1.1 Yksinkertainen komponentti

Reactin komponenteissa käytetään render-metodia. Render-metodille annetaan dataa ja se palauttaa sen, mitä halutaan näyttää ohjelmassa. Reactilla ohjelmoidessa on mahdollista käyttää XML-tyylistä syntaksia nimeltään JSX. JSX on JavaScriptin lisäosa, joka laajentaa sen syntaksia. JSX-lisäosaa suositellaan käytettäväksi Reactin kanssa käyttöliittymän ulkonäön kuvailemiseen. JSX-lisäosan käyttö ei kuitenkaan ole pakollista. Kuviossa 1 on esimerkki yksinkertaisen React-komponentin luomisesta ja keltaisella korostetussa osassa on komponentti selaimessa avattuna eli renderöitynä. Annettu data voidaan näyttää render-metodissa this.props-komennon avulla. (React, 2017.)

Kuvio 1. Yksinkertainen React-komponentti (React, 2017).

2.1.2 Tilallinen komponentti

Komponenteissa voidaan säilöä tiladataa sisäisesti. Tiladataa voidaan käyttää this.state-komennon avulla (kuvio 2). Tiladata on hyvä vaihtoehto datan välittämiselle. Kun komponentin tiladatassa tapahtuu muutoksia, render-metodia kutsutaan automaattisesti uudestaan. Uudelleenrenderöinnin tapahtuessa kaikki komponentin osat, joita muutokset koskevat, päivittyvät selaimessa. Esimerkiksi kuviossa 2 tehdyssä tiladataa sisältävässä komponentissa on toteutettu yksinkertainen laskuri. Laskurin numero tulee komponentin tiladatasta, jota kasvatetaan yhdellä sekunnin välein. Tiladata kasvaa yhdellä joka sekunti, jolloin selaimessa näkyvä laskuri kasvaa yhdellä sekunnin välein uudelleenrenderöinnin ansiosta (kuviossa 2 keltaisella korostettu). (React, 2017.)

```
class Timer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {secondsElapsed: 0};
  }

  tick() {
    this.setState((prevState) => ({
      secondsElapsed: prevState.secondsElapsed + 1
    }));
  }

  componentDidMount() {
    this.interval = setInterval(() => this.tick(), 1000);
  }

  componentWillUnmount() {
    clearInterval(this.interval);
  }

  render() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
}

ReactDOM.render(<Timer />, mountNode);
```

Seconds Elapsed: 1415

Kuvio 2. React-komponentti, jolla on tila eli state (React, 2017).

2.2 React-Router

React-Router on monipuolinen reititykseen käytettävä kirjasto, joka on rakennettu Reactin päälle. React-Routerin avulla voidaan varmistaa uusiin näkymiin siirtyminen sulavasti. Näkymiin siirryttäessä React-Router pitää URL-osoitteen ajan tasalla, sen mukaan mitä sivulla näytetään. React-Routerille määritellään komponentit, jotka renderöidään URL-osoitteen perusteella. (React-Router, 2017a.)

2.2.1 Reittien määritys

Reittien määrittämisellä React-Routerissa tarkoitetaan reitittäjän ohjeistamista, siten että se osaa ajaa oikean osan koodista URL-osoitteen perusteella. Kuviossa 3 luodaan yksinkertainen reititys, jonka perusteella ohjelma osaa renderöidä oikean komponentin. (React-Router, 2017b.)

```
render((
  <Router>
    <Route path="/" component={App}>
      <Route path="about" component={About} />
      <Route path="inbox" component={Inbox}>
        <Route path="messages/:id" component={Message} />
      </Route>
    </Route>
  </Router>
), document.body)
```

Kuvio 3. Yksinkertainen reittien määrittäminen (React-Router, 2017b).

Kuvion 3 määritysten perusteella sivulle mentäessä renderöidään vakiona App-komponentti. Jos URL-osoitteen lopussa lukee /about, ohjelma renderöi About-komponentin ja niin edelleen (kuvio 4). (React-Router, 2017b.)

URL	Components
/	App
/about	App -> About
/inbox	App -> Inbox
/inbox/messages/:id	App -> Inbox -> Message

Kuvio 4. Määritetyt säännöt. Mikä komponentti renderöidään missäkin URL-osoitteessa (React-Router, 2017b).

2.3 Redux

Redux on JavaScript-ohjelmia varten luotu säilö, jonne voidaan laittaa kaikki ohjelman tilat. Esimerkiksi Reactissa voidaan käyttää Reduxia sen sijaan, että käytettäisiin kunkin komponentin tiloja. Reduxilla ohjelmaan saadaan selvyyttä, koska kaikki tilat ovat tallessa yhteisessä säilössä, storessa. Reduxia voidaan kuvailla kolmella pääperiaatteella:

- Koko ohjelman tila yhdessä storessa.
- Tilaa voidaan vain lukea, muutokset actioneilla.
- Muutokset tilaan tehdään puhtailla funktioilla, reducereilla. (Redux, 2017.)

JavaScriptillä tehtävien ohjelmien vaatimustasot ovat nykyään aiempaa korkeampia, ja ohjelmista tulee koko ajan monimutkaisempia. Koodin täytyy kyetä käsittelemään yhä enemmän tiloja, jotka saattavat sisältää esimerkiksi pyyntöjä palvelimelle. Redux on työkalu, joka mahdollistaa kaiken tämän. (Redux, 2017.)

2.3.1 Actionit

Actionilla tarkoitetaan tilan vaihtamiseen käytettävää objektia, joka kuvastaa, mitä tapahtui. Actionien avulla varmistetaan, että mikään ei koskaan pääse kirjoittamaan suoraan tilaan. Actionit sisältävät tietoa, jotka lähettävät dataa storeen. Actionit ovat storen ainoa tiedonlähde. Actioneja voidaan lähettää store.dispatch-funktiolla. (Redux, 2017.)

Kuviossa 5 on luotu kampanjoiden hakuun käytettävä funktio. Funktio sisältää kolme actionia:

- FETCH_CAMPAIGNS_PENDING, haetaan kampanjoita
- FETCH_CAMPAIGNS_FULFILLED, kampanjat haettu
- FETCH_CAMPAIGNS_REJECTED, haku keskeytetty

Actioneita lähetetään sen mukaan, kuinka palvelin vastaa. PENDING-action lähetetään heti funktiota kutsuttaessa ilmaisemaan haun aloittaminen (kuvio 5). Palvelimen vastatessa haetuilla kampanjoilla, lähetetään FULFILLED-action, ja sen mukana palvelimen antamat kampanjat storeen. Datan lähetys tapahtuu dispatch-funktion sisällä olevassa payload-kohdassa (kuvio 5). Jos palvelin antaa virheilmoituksen, lähetetään REJECTED-action, ja sen mukana virheilmoitus storeen (kuvio 5).

```
export function fetchCampaigns () {
    return function (dispatch) {
        dispatch({ type: "FETCH_CAMPAIGNS_PENDING" });
        Axios.get(config.nodeServer + "getCampaigns")
            .then((response) => {
                dispatch({ type: "FETCH_CAMPAIGNS_FULFILLED", payload: response.data });
            })
            .catch((error) => {
                dispatch({ type: "FETCH_CAMPAIGNS_REJECTED", payload: error });
            });
    };
}
```

Kuvio 5. Esimerkki palvelinpyynnön sisältävän actionin luomisesta.

2.3.2 Reducerit

Reducerit ovat puhtaita funktioita, jotka määrittävät kuinka action on muuttanut tilaa. Reducer ottaa edellisen tilan ja tulleen actionin, minkä jälkeen se palauttaa seuraavan tilan. Jokaista osa-aluetta kohden suunnitellaan omat reducerit. (Redux, 2017.)

Kuviossa 6 on luotu reducerit kullekin kampanjoiden hakua koskevalle actionille. Switch-case-lauseessa muutetaan tiettyjä storen arvoja. Esimerkiksi haun onnistuttua laitetaan haetut kampanjat storeen campaigns-arrayhyn, ja muutetaan arvo fetched eli haettu todeksi (kuvio 6). Eri arvoja muuttelemalla voidaan reactin puolella määrittää renderöitäviä komponentteja. Esimerkiksi kuvion 6 tapauksessa arvon fetched ollessa tosi, voidaan renderöidä kolmen sekunnin ajaksi teksti success, jotta käyttäjä tietää haun onnistuneen.

```
switch (action.type) {

    case "FETCH_CAMPAIGNS_PENDING": {
        return {
            ...state,
            fetching: true
        };
    }
    case "FETCH_CAMPAIGNS_REJECTED": {
        return {
            ...state,
            fetching: false,
            error: action.payload
        };
    }
    case "FETCH_CAMPAIGNS_FULFILLED": {
        return {
            ...state,
            fetching: false,
            fetched: true,
            campaigns: action.payload,
            changed: false,
            created: false
        };
    }
```

Kuvio 6. Kampanjoiden hakua varten luodut reducerit.

2.4 Webpack

Webpack on moduulien bundlaamiseen eli niputtamiseen käytettävä työkalu. Webpackin luoma paketti on JavaScript-tiedosto, joka sisältää kaikki ominaisuudet, tiedostot ja toiminnot, mitkä kuuluvat yhteen. Paketti on tiedosto, joka tulee tarjoilla palvelimelta esimerkiksi selaimelle, kun selain tekee pyynnön yhdestä tiedostosta. Paketti voi sisältää JavaScriptiä, CSS-tyylejä, HTML-tiedostoja ja melkein mitä tahansa muita tiedostoja. (Angular, 2017.)

Webpack käy ohjelman lähdekoodin läpi ja etsii sieltä import-komentoja, rakentaa riippuvuuskaavion (mitä kaikkia moduuleja ohjelma tarvitsee), sekä luo yhden tai useamman paketin. Lisäosien ja sääntöjen avulla Webpack voi esikäsitella ja minimoida myös muita tiedostoja JavaScriptin lisäksi. Tällaisia muita tiedostoja ovat esimerkiksi TypeScript, SASS ja LESS. Koodin minimoimisella tarkoitetaan kaikkien turhien merkkien poistamista koodista, mutta kuitenkin kaiken toiminnallisuuden säilyttämistä ennallaan. Minimoimisella saadaan tiedoston kokoa pienemmäksi. (Angular, 2017.)

3 BACK-END-OSA

3.1 NodeJS

NodeJS on avoimen lähdekoodin alusta, jolla voidaan luoda nopeita ja skaalautuvia web-ohjelmia JavaScriptiä käyttäen. NodeJS käyttää tapahtumaperäistä, tukkeutumatonta I/O-mallia, mikä tekee siitä kevyen ja tehokkaan alustan web-ohjelmille. NodeJS voi käsitellä useita yhtäaikaisia yhteyksiä, mikä tekee siitä loistavan välineen reaaliaikaisille, runsaasti dataa sisältäville ohjelmille. (Teixeira 2013, 3.)

NodeJS-kirjastolla voidaan rakentaa esimerkiksi HTTP-välityspalvelin, DNS-palvelin, SMTP-palvelin, IRC-palvelin tai mikä tahansa verkkointensiivinen prosessi. NodeJS-kirjastolla ohjelmoidessa käytetään JavaScriptiä. JavaScript on voimakas kieli, jolla voidaan helposti kirjoittaa verkotettuja, tapahtumaperäisiä ohjelmia. (Teixeira 2013, 3.)

Yksi NodeJS-kirjaston tärkeimmistä ominaisuuksista on kolmannen osapuolten avoimen lähdekoodin moduulien käyttö. NodeJS tarjoaa käyttöön Node Package Managerin (NPM), jolla on helppo asentaa, käsitellä ja käyttää mitä tahansa moduulia. Kyseisiä moduuleja löytyy paljon ja arkistot kasvavat edelleen. NPM mahdollistaa moduulien asennuksen eristetysti, eli ohjelmassa voi olla esimerkiksi kaksi eri versiota samasta moduulista, eivätkä ne aiheuta konfliktia. (Teixeira 2013, 3.)

3.1.1 Tärkeimmät moduulit

NodeJS-kirjastoon on tarjolla suuri määrä erilaisia moduuleja eri tarkoituksiin. Require-funktiolla voidaan ottaa moduuleja käyttöön NodeJS-ohjelmassa. Seuraavassa on lyhyt lista tärkeimmistä moduuleista:

- net: TCP-asiakkaiden ja -palvelinten luomista varten.
- http: HTTP-palveluiden luomista ja käyttämistä varten.
- fs: Tiedostojen käyttämistä ja manipuloimista varten.

- dns: DNS-palvelun käyttöä varten.
- events: Tapahtumalähettimien luomista varten.
- stream: Datavirtojen luomista varten.
- os: Paikallisen käyttöjärjestelmän statistiikan tutkimista varten.
- assert: Käytetään pysyvien muuttujien testaamiseen.
- util: Monenlaisten hyötyohjelmien käyttöön. (Teixeira 2013, 12.)

Moduuli voidaan tuoda ohjelmassa paikalliseen muuttujaan require-funktion avulla (kuvio 7) (Teixeira 2013, 12).

```
var fs = require('fs');
```

Kuvio 7. Moduulin tuominen paikalliseen muuttujaan "fs" (Teixeira 2013, 12).

3.1.2 NodeJS-kirjastolla ohjelmoiminen

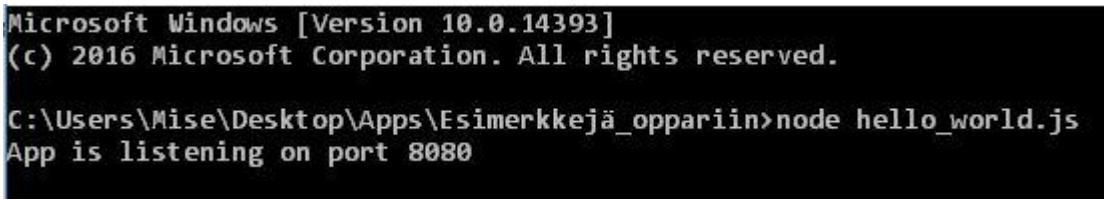
Kuviossa 8 on esimerkki yksinkertaisesta Hello World -ohjelmasta, joka on kirjoitettu NodeJS-kirjastoa käyttäen. Ohjelmassa luodaan HTTP-palvelin, joka vastaa kirjoittamalla selaimeen ”Hello World”, kun käyttäjä menee osoitteeseen localhost:8080. Ohjelman luoma palvelin kuuntelee siis porttia 8080.

```
var http = require("http");
var server = http.createServer();
server.on("request", function(req,res){
    res.writeHead(200, {"content-type":"text/plain"});
    res.write("Hello World");
    res.end();
});

var port = 8080;
server.listen(port);
server.once("listening", function(){
    console.log("App is listening on port "+port);
})
```

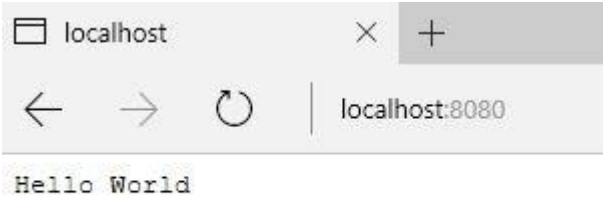
Kuvio 8. "Hello World"-ohjelma NodeJS-kirjastoa käyttäen (Teixeira 2013, 9).

Luotu HTTP-palvelin käynnistetään avaamalla Windowsin komentokehote kansiossa, jossa HTTP-palvelinta varten luotu hello_world.js-tiedosto sijaitsee. Komennolla node käynnistetään palvelin (kuvio 9).



Kuvio 9. Hello World -ohjelman käynnistäminen komentokehotteessa.

Kun ohjelma on käynnistetty, se ilmoittaa komentokehotteessa kuuntelevansa porttia 8080 (kuvio 9). Kun käyttäjän menee selaimessa osoitteeseen localhost:8080, ohjelma vastaa kirjoittamalla selaimeen ”Hello World” (kuvio 10).



Kuvio 10. Hello World -ohjelman vastaus.

3.2 Express

Express on pieni ja joustava runko NodeJS-kirjastolla tehtyjä web-ohjelmia varten. Express tarjoaa kokoelman voimakkaita ominaisuuksia web- ja mobiili-ohjelmia varten. Useat suositut rungot perustuvat Expressiin. Express tarjoaa suuren määrän HTTP-toimintoja sekä väliohjelmistoja, mikä mahdollistaa voimakkaiden API-rajapintojen luomisen vaivattomasti. Expressin tarjoamat ominaisuudet eivät sekoita NodeJS-kirjaston toimintoja. (Express, [Viitattu 1.2.2017].)

3.2.1 Routing eli reititys

Yksi Expressin tärkeimmistä toiminnoista on routing eli reititys. Reitityksellä määritetään kuinka NodeJS-ohjelma vastaa käyttäjän pyyntöihin, jotka tulevat tiettyihin osoitteisiin. Kuviossa 11 otetaan käyttöön Express ja tehdään yksinkertainen reititys. Käyttäjän mennessä kotisivulle ohjelma vastaa ”hello world”. (Express, [Viitattu 6.2.2017].)

```
var express = require('express')
var app = express()

// respond with "hello world" when a GET request is made to the homepage
app.get('/', function (req, res) {
  res.send('hello world')
})
```

Kuvio 11. Yksinkertainen esimerkki reitittämisestä Expressissä (Express, [Viitattu 6.2.2017]).

3.2.2 Express Router

Express.Router-luokkaa käytetään luomaan modulaarisia, koottavia reittien käsittelijöitä. Router on väliohjelmisto ja reititysjärjestelmä, minkä vuoksi sitä kutsutaan usein mini-ohjelmaksi. Kuviossa 12 luodaan router-moduuli, jossa käytetään väliohjelmistoa ja määritellään reittejä. (Express, [Viitattu 6.2.2017].)

```
var express = require('express')
var router = express.Router()

// middleware that is specific to this router
router.use(function timeLog (req, res, next) {
  console.log('Time: ', Date.now())
  next()
})
// define the home page route
router.get('/', function (req, res) {
  res.send('Birds home page')
})
// define the about route
router.get('/about', function (req, res) {
  res.send('About birds')
})

module.exports = router
```

Kuvio 12. Routerin luominen (Express, [Viitattu 6.2.2017]).

Routerin luomisen jälkeen se täytyy vielä ottaa käyttöön pääohjelmassa. Router tuodaan pääohjelmaan require-komennolla (kuvio 13). Router otetaan käyttöön app.use-komennolla (kuvio 13). (Express, [Viitattu 6.2.2017].)

```
var birds = require('./birds')

// ...

app.use('/birds', birds)
```

Kuvio 13. Routerin käyttöönotto pääohjelmassa (Express, [Viitattu 6.2.2017]).

3.3 MongoDB

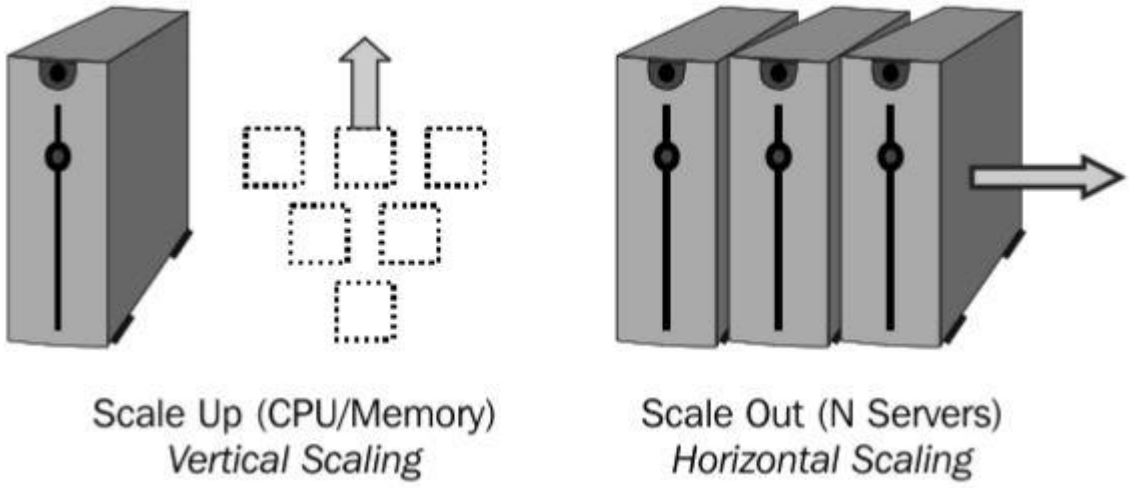
Nykyisin yritykset tallentavat valtavat määrät monipuolista dataa. Datan säilyttämiseen käytetään useita eri metodeja. Datan säilyttämiseen ei ole yleistä tapaa, jolla voitaisiin hoitaa kaikki erilaiset tietolähteet. Kukin datan säilyttämisen tekniikka on kehittynyt vastaamaan silloisen ajanjakson tarpeita. Datan määrän, monipuolisuuden ja nopeuden kasvaessa, uusia säilyttämisratkaisuja kehitetään jatkuvasti. Yhdessä organisaatiossa saattaa esiintyä useita eri tietolähteitä:

- yksinkertaiset taulukot (tekstitiedostot ja Microsoft Excel)
- ilmaiset relaatiotietokannat (Oracle, SQL server ja DB2)
- avoimen lähdekoodin relaatiotietokannat (MySQL ja PostGreSQL)
- modernit, verkko-painotteiset tietolähteet (XML, JSON, verkkopalvelut ja APIt)
- analyyttiset tietokannat (Vertica, Greenlplum ja InfoBright)
- laitteiden luomat tietolähteet (sensorien data, ohjelman logit ja palvelimen logit)
- NoSQL-tietokannat (MongoDB, Redis, Cassandra, Hbase ja CouchDB) (Borland 2014, 6.)

Yritykset sijoittavat paljon datan säilyttämistekniikoihin ja henkilöstöön, joka osaa taltioida, säilöä ja käsitellä dataa. MongoDB on saavuttanut suuren suosion ja nykyään se lukeutuu NoSQL-tietokantojen kärkinimiin. (Borland 2014, 6.)

3.3.1 Historiaa

Perinteisissä relaatiotietokannoissa datan määrän kasvamisen käsittely sisältää yleensä ylöspäin skaalauksen, eli yhdelle tietokantapalvelimelle lisätään keskusmuistia ja keskusyksiköitä (kuvio 14). Tämä tapa on kallis ja yhden palvelimen skaalauksella on rajansa. Google ja Amazon päättivät ratkaista kyseiset dataan liittyvät ongelmat kehittämällä omat hajautetut tietokantansa, joita on helppo skaalata ulos satojen tai tuhansien palvelimien kesken (kuvio 14). Kyseiset hajautetut tietokannat saivat aikaan NoSQL-tietokantaliikkeen, jonka ansiosta myös MongoDB syntyi. (Borland 2014, 6-7.)



Kuvio 14. Palvelimen skaalaus ylös (scale up) ja ulos (scale out) (Borland 2014, 7).

3.3.2 MongoDB-tietokannan tietomalli

Hierarkkiset JSON-dokumentit muodostavat MongoDB-tietokannan tietomallin perustan. JSON-dokumentit ovat kielestä riippumattomia tekstitiedostoja, jotka esittävät dataa. JSON-dokumenteissa on kaksi ensisijaista datarakennetta: sisäkkäiset kokoelmat nimi/arvo-pareja ja järjestetyt listat, kuten arrayt. Kuviossa 15 on esimerkki MongoDB-dokumentista, joka on JSON-muodossa. JSON sisältää objektin, jossa on elokuvan tiedot, sisäkkäin asetetun objektin, jossa on tuotantoyhtiön tiedot ja arrayn, jossa on näyttelijät. MongoDB-tietokannan dokumenteissa jokainen objekti on erotettu keskenään pilkulla. Objektit ovat aaltosuluilla suljettuja kokoelmia avain-arvo-pareista. (Borland 2014, 7-8.)

```
{
        "movie": "Forrest Gump",
        "rating": "PG-13",
        "duration_min": 142,
        "production_company": {
                "name": "Parmount Pictures",
                "streetAddress": "5555 Melrose Ave",
                "city": "Los Angeles",
                "state": "CA",
                "postalCode": 90038
        },
        "cast": [
                    {
                         "character": "Forrest Gump",
                         "person": "Tom Hanks"
                    },
                    {
                         "character": "Jenny Curran",
                         "person": "Robing Wright"
                    }
        ]
}
```

Kuvio 15. MongoDB-dokumentti JSON-muodossa (Borland 2014, 8).

MongoDB-tietokannalla on oma kyselykieli, josta on rakennettu tehokas tapa hakea, käsitellä ja päivittää JSON-dokumentteja. Dokumentti-painotteinen datan säilöntä on hyvä vaihtoehto SQL-pohjaisille relaatiotietokannoille ja se tarjoaakin ainutlaatuisia hyötyjä:

- Mahdollistaa nopean ja helpon horisontaalisen skaalauksen ryhmittelemällä toisiinsa liittyvät datat dokumenttikokoelmiin erillisten tietokantataulujen sijaan.
- Mahdollistaa nopeamman ja helpomman sovelluskehityksen, koska data tarjotaan JSON-muodossa. JSON on tuettu lähes kaikkien modernien ohjelmointikielien keskuudessa.
- Sallii datan lisäämisen tietokantaan ilman ennalta määritettyä tietokantamallia. Ohjelmistokehittäjien on joustavampaa määritellä ja manipuloida tietokantamallia, koska heidän ei tarvitse olla riippuvaisia yhdestä tietokantaylläpitäjästä, joka huolehtii tietokantamallin muuttamisesta. (Borland 2014, 8.)

3.4 Mongoose

Mongoose on MongoDB-tietokannan ja NodeJS-kirjaston kanssa käytettävä työkalu, jolla voidaan mallintaa objekteja. Mongoose on suunniteltu toimivaksi asynkronisessa ympäristössä. Mongoose tarjoaa selkeän skeema-pohjaisen ratkaisun ohjelman datan mallintamiseen. (Mongoose, [Viitattu 1.2.2017].)

3.4.1 Skeema ja malli

Skeema määrittää dokumenttien muodon MongoDB-kokoelmassa. Mongoosen käyttäminen alkaa skeeman luomisella. Skeemassa annetaan avaimet ja niitä vastaavien arvojen tyypit. Näitä kutsutaan skeematyypeiksi. Sallitut skeematyypit ovat:

- String
- Number
- Date
- Buffer
- Boolean
- Mixed
- ObjectId
- Array (Mongoose, [Viitattu 1.2.2017].)

Kuviossa 16 on esimerkki yksinkertaisen skeeman luomisesta mongoosessa (Mongoose, [Viitattu 1.2.2017]).

```
var mongoose = require("mongoose");
var Schema = mongoose.Schema;

var animalSchema = new Schema({
        Name: String,
        Race: String,
        Age: Number,
        Weight: Number
});
```

Kuvio 16. Esimerkki yksinkertaisesta skeemasta.

Skeeman määrittelyn jälkeen se täytyy muuntaa malliksi, jota voidaan käyttää ohjelmoidessa. Muuntaminen tapahtuu mongoose.model-komennolla (kuvio 17). (Mongoose, [Viitattu 1.2.2017].)

```
var Animal = mongoose.model("Animal", animalSchema);
```

Kuvio 17. Mallin luominen.

MongoDB-tietokantaan voidaan tallentaa dokumentteja mallien avulla. Tallentaminen tapahtuu save-komennolla. Kuvio 18 on esimerkki uuden dokumentin luomisesta ja tallentamisesta tietokantaan.

```
var newAnimal = new Animal({
        Name: "Puppe",
        Race: "Koira",
        Age: 5,
        Weight: 15
});

newAnimal.save(function (err, createdAnimal) {
        if(err){ console.error(err); }
        res.json(createdAnimal);
});
```

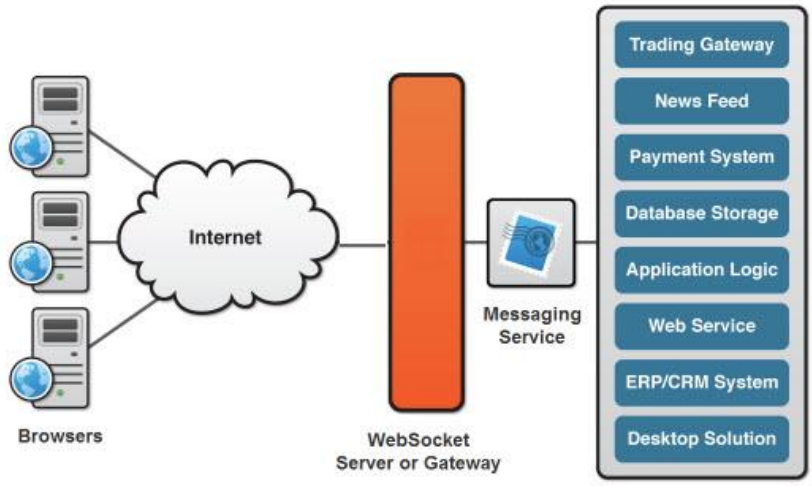
Kuvio 18. Dokumentin luominen ja tallentaminen tietokantaan.

3.5 WebSocket ja Socket.IO

WebSocket ja Socket.IO ovat reaaliaikaisten web-sovellusten tekemiseen käytettäviä tekniikoita. Kyseiset tekniikat mahdollistavat kaksisuuntaisen kommunikaation etäisännän kanssa (esimerkiksi selaimen ja palvelimen välillä). Kaksisuuntainen kommunikaatio mahdollistaa sen, että molemmat osapuolet voivat lähettää toisilleen dataa yhtä aikaa. Esimerkiksi reaaliaikainen keskusteluohjelma voidaan toteuttaa kyseisillä tekniikoilla. (WebSocket, 2017.)

3.5.1 WebSocket

WebSocket-tekniikka mahdollistaa kaksisuuntaisen kommunikaation yhden socketin kautta verkon ylitse (kuvio 19). Tämä vähentää huomattavasti verkon kuormitusta ja viivettä verrattuna vanhoihin tapoihin matkia kaksisuuntaista, suoraa kommunikaatiota etäisäntien kanssa. (WebSocket, 2017.)



Kuvio 19. WebSocketin arkkitehtuuri (WebSocket, 2017).

Yksi WebSocketin ominaisuuksista on sen kyky kulkea palomuurien ja välityspalvelimien lävitse, mikä on yleisesti yksi web-sovelluksen ongelmakohdista. WebSocket havaitsee välityspalvelimen läsnäolon ja luo automaattisesti tunnelin sen läpäisyä varten. (WebSocket, 2017.)

Tunnelin luominen tapahtuu antamalla välityspalvelimelle HTTP CONNECT -lausunto, joka pyytää välityspalvelinta avaamaan TCP/IP-yhteyden tiettyyn isäntään ja porttiin. Kun tunneli on luotu, kommunikaatio onnistuu välityspalvelimen lävitse häiriöttä. Koska HTTP/S toimii samalla periaatteella, WebSocket voi hyödyntää samaa HTTP CONNECT -lausuntoa turvallisen yhteyden luomiseen SSL-teknologian kautta. (WebSocket, 2017.)

3.5.2 WebSocket-protokolla

WebSocket-protokolla suunniteltiin toimimaan hyvin olemassa olevan Web-infrastruktuurin kanssa. Tämän vuoksi yksi suunnittelun periaatteista oli, että WebSocket-yhteys on aluksi HTTP-yhteys. Protokollan vaihtumista HTTP-yhteydestä WebSocketiksi kutsutaan WebSocket-kättelyksi. Selain lähettää palvelimelle pyynnön, jossa se ilmoittaa haluavansa vaihtaa protokollan HTTP-yhteydestä WebSocketiksi. Tämä käy ilmi pyynnön upgrade-headerista (kuvio 20). (WebSocket, 2017.)

```
GET ws://echo.websocket.org/?encoding=text HTTP/1.1
Origin: http://websocket.org
Cookie: __utma=99as
Connection: Upgrade
Host: echo.websocket.org
Sec-WebSocket-Key: uRovscZjNol/umbTt5uKmw==
Upgrade: websocket
Sec-WebSocket-Version: 13
```

Kuvio 20. WebSocket-kättelyn pyyntövaihe (WebSocket, 2017).

Mikäli palvelin ymmärtää WebSocket-protokollan, se hyväksyy protokollan vaihdoksen. Tämä käy ilmi upgrade-headerista (kuvio 21). Tässä vaiheessa HTTP-yhteys suljetaan ja korvataan WebSocket-yhteydellä käyttäen samaa taustalla olevaa TCP/IP-yhteyttä. WebSocket-yhteys ottaa vakiona käyttöön samat portit kuin HTTP(80) ja HTTPS(443). (WebSocket, 2017.)

```
HTTP/1.1 101 WebSocket Protocol Handshake
Date: Fri, 10 Feb 2012 17:38:18 GMT
Connection: Upgrade
Server: Kaazing Gateway
Upgrade: WebSocket
Access-Control-Allow-Origin: http://websocket.org
Access-Control-Allow-Credentials: true
Sec-WebSocket-Accept: rLHCkw/SKsO9GAH/ZSFhBATDKrU=
Access-Control-Allow-Headers: content-type
```

Kuvio 21. WebSocket-kättelyn hyväksyminen (WebSocket, 2017).

3.5.3 Socket.IO

Socket.IO on hyvin samankaltainen kaksisuuntaiseen kommunikaatioon käytettävä tekniikka, kuin WebSocket. Socket.IO toimii samalla lailla eri selainten kesken, mikä helpottaa sen käyttämistä. Socket.IO tarjoaa myös paljon monipuolisemman API-rajapinnan WebSockettiin verrattuna. (Rai 2013, 87.)

3.5.4 Socket.IO API

Socket.IO-tekniikan käyttäminen on hyvin samanlaista kuin WebSocketin käyttäminen. Kuviossa 22 on yksinkertainen esimerkki Socket.IO-tekniikan käyttämisestä ohjelmassa. Websocketissa käsittelijä liitetään komentoon käyttämällä toimintoja onopen, onmessage ja onclose, kun Socket.IO-tekniikassa käytetään toimintoa ”on”. (Rai 2013, 90.)

```
<script>
  var socket = io.connect("http://localhost:8080");

  socket.on("connect", function() {
    socket.send("Client socket connected");
  });

  socket.on("message", function(data) {
    console.log("Received a message from the server!", data);
  });

  socket.on("disconnect", function() {
    console.log("The client socket disconnected!");
  });

</script>
```

Kuvio 22. Yksinkertainen Socket.IO-esimerkki (Rai 2013, 89-90).

Socket.IO-tekniikassa voidaan käyttää itse luotuja tapahtumia. Vaikka tapahtuma on itse luotu, sama API käsittelee sen. Uusi, itse luotu tapahtuma, voidaan lähettää käyttämällä emit-toimintoa (kuvio 23). Emit-toiminnossa voidaan lähettää dataa yhteydessä olevalle tai oleville asiakkaille (kuvio 23). Datan vastaanotto tapahtuu kuuntelemalla luotua tapahtumaa (tässä tapauksessa nimellä myevent) on-toiminnon avulla (kuvio 24). (Rai 2013, 90.)

```
socket.emit("myevent", {"eventData": "..."});
```

Kuvio 23. Socket.IO, datan lähetys (Rai 2013, 90).

```
socket.on("myevent", function(event){...});
```

Kuvio 24. Socket.IO, datan vastaanottaminen (Rai 2013, 90).

Socket.IO tarjoaa seuraavia ominaisuuksia:

- Viestien jakaminen nimiavaruuksiin
- Yhteyksien kanavointi
- Yhteyskatkon havaitseminen
- Yhteyden uudelleen muodostaminen
- Kyky lähettää viestejä kaikille yhteydessä oleville asiakkaille. (Rai 2013, 90).

Kaikkien näiden ominaisuuksien vuoksi Socket.IO tarvitsee oman protokollan ja mekanismin toimiakseen (Rai 2013, 90).

3.5.5 Socket.IO socket

Minkä tahansa socketin tavoin, Socket.IO-tekniikan socketilla on useita eri tiloja elämänkaaressaan. Kyseiset tilat riippuvat yhteyden tilasta. Mahdolliset tilat ovat seuraavat:

- muodostetaan yhteyttä
- yhteys muodostettu
- katkaistaan yhteyttä
- yhteys katkaistu. (Rai 2013, 90-91).

Socket luodaan kun asiakas lähettää yhteyspyynnön palvelimelle ja niiden välinen kättely aloitetaan. Kun kättely on saatu loppuun, yhteys avataan käyttäen sitä kuljetusmekanismia, joka on kättelyn aikana sovittu. Tämän jälkeen socketin tilaksi muuttuu yhteys muodostettu. (Rai 2013, 90-91.)

Joissakin tapauksissa palvelin voi vaatia asiakkaalta säännöllisiä viestejä tarkistaakseen socketin olemassaolon. Tämä riippuu palvelimen asetuksista. Mikäli kyseistä viestiä ei tule, socketin yhteys katkaistaan ja sen tilaksi muuttuu: yhteys katkaistu. Jos yhteys katkaistaan, asiakas käynnistää uudelleenyhdistämisen. Jos yhteys onnistutaan palauttamaan sovitun aikamääreen aikana, lähettämättä jääneet viestit lähetetään eteenpäin. Jos yhteyttä ei onnistuta palauttamaan, asiakas lähettää uuden yhteyspyynnön palvelimelle ja aloittaa uuden kättelyn. (Rai 2013, 91.)

3.5.6 Socket.IO-yhteys

Socket.IO-yhteys alkaa kättelyllä. Kättelyä lukuun ottamatta kaikki muut tapahtumat ja viestit protokollassa siirretään socketin kautta. Socket.IO on tarkoitettu käytettäväksi web-ohjelmien kanssa, joten siihen liittyy oletus, että nämä ohjelmat kykenevät aina käyttämään HTTP-protokollaa. Tämän oletuksen takia kättely tapahtuu HTTP-protokollan kautta. (Rai 2013, 91)

Yhteyden muodostamiseksi asiakas lähettää POST-pyynnön kättelyn osoitteeseen. Palvelin vastaa tähän pyyntöön yhdellä kolmesta mahdollisesta tavasta:

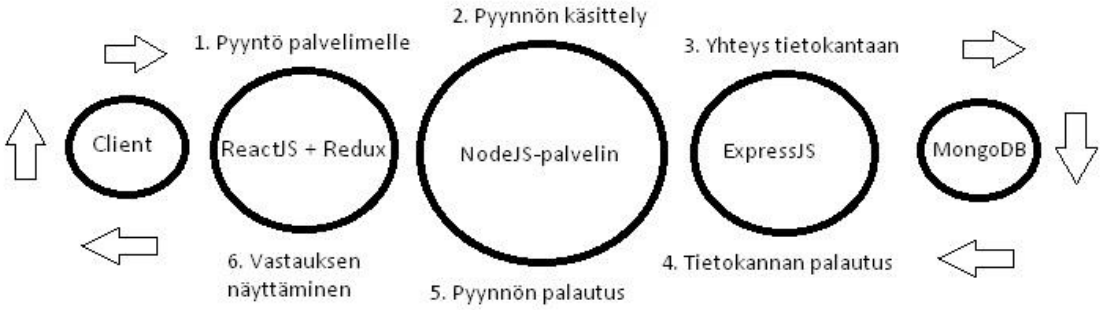
- 200 OK
- 401 Luvaton
- 503 Palvelu ei saatavissa. (Rai 2013, 92.)

Kättelyn onnistuttua palvelin ja Socket.IO-asiakas alkavat kommunikoida keskenään käyttäen palvelimen tarjoamaa ja asiakkaan tukemaa kuljetusmekanismia (Rai 2013, 92).

4 Web-ohjelman rakenne

Moderni web-ohjelma koostuu useista eri tekniikoista ja kielistä. Aikaisemmin esitellyt tekniikat ovat tärkeitä web-ohjelmaa luodessa, mutta ohjelmoidessa voidaan käyttää vielä paljon muitakin tekniikoita. Tässä luvussa käydään läpi web-ohjelman rakennetta yksinkertaisen esimerkkiohjelman avulla. Kyseisellä esimerkkiohjelmalla voidaan tehdä operaatioita tietokantaan, joka sisältää kuvitteellisia oppilaita ja heidän tietojaan. Oppilaita voidaan lisätä, poistaa, hakea nimellä ja päivittää. Esimerkkiohjelmasta käydään läpi oppilaan luomisen prosessi.

Kuvio 25 kuvastaa web-ohjelman arkkitehtuuria. Front-end eli käyttäjälle näkyvä osa on tehty Reactilla. Reduxia ei tässä esimerkissä otettu käyttöön, koska dataa liikkuu niin vähän. Palvelinpuoli eli back-end on ohjelmoitu NodeJS-kirjastoa ja Expressiä käyttäen. Tietokantana toimii MongoDB.



Kuvio 25. Esimerkkiohjelman arkkitehtuuri.

4.1 Client.js

Client.js on tiedosto, jossa määritellään React Router eli renderöitävät komponentit URL-osoitteen mukaan (kuvio 26). Tiedostoon tuodaan komponentit mukaan import-komennolla. Require-komennolla liitetään tyylitiedosto ohjelmaan (kuvio 26).

```
import React from "react";
import ReactDOM from "react-dom";
import {Router, Route, IndexRoute, browserHistory} from "react-router";


import Layout from "./components/Layout";
import Header from "./components/header";
import Frontpage from "./components/frontpage";
import GetData from "./components/getdata";
import Uploader from "./components/uploader";
import MyParent from "./components/getdata2";
import UpdateData from "./components/updateData";
import Delete from "./components/delete";


require("../styles/stylesheet.scss");


const app = document.getElementById("app");
ReactDOM.render(
    <Router history={browserHistory}>
        <Route path="/" component={Layout}>
            <Route path="Header" component={Header}/>
            <Route path="Frontpage" component={Frontpage}/>
            <Route path="GetData" component={GetData}/>
            <Route path="Uploader" component={Uploader}/>
            <Route path="GetDataV2" component={MyParent}/>
            <Route path="UpdateData" component={UpdateData} />
            <Route path="Delete" component={Delete} />
        </Route>
    </Router>,app);
```

Kuvio 26. Client.js-tiedoston lähdekoodi.

Kohdassa history otetaan käyttöön browserhistory-niminen toiminto React-Routerista (kuvio 26). Browserhistoryn avulla saadaan pidettyä URL siistinä selaimessa ja sen avulla voidaan esimerkiksi siirtyä uudelle routelle ohjelmallisesti. Route-kohdissa määritellään renderöitävät komponentit kullekin URL-osoitteelle (kuvio 26). Tässä tapauksessa juuressa (kohta path=”/”) renderöidään komponentti Layout (kuvio 26).

4.2 Layout

Käyttäjän mennessä ohjelman etusivulle, tässä tapauksessa osoitteeseen localhost:8080/, renderöidään Layout-komponentti. Layout-komponentissa renderöidään navigaatiopalkki, jossa on linkit eri sivuille. React Router tarjoaa link-elementin, jolla voidaan kätevästi navigoida ja pitää URL ajan tasalla. URL-osoitteen mukainen komponentti renderöidään tässä tapauksessa this.props.children-kohdassa (kuvio 27), eli navigaatiopalkki on koko ajan näkyvissä sivun yläreunassa (kuvio 28).

```
import React from "react";
import _ from "lodash";
import Axios from "axios";
import {Link} from "react-router";
import Header from "./header";

export default class Layout extends React.Component {
    constructor() {
        super();
        this.state = {
            title: "ReactiSivu !",
            data: null
        };
    }

    render() {
        return (
            <div>
                <Header title="React Sivu !"/>
                <div>
                    <ul class="nav nav-tabs">
                        <li><Link to="Frontpage">Etusivu</Link></li>
                        <li><Link to="GetData">Hae kannasta</Link></li>
                        <li><Link to="Uploader">Lisää kantaan</Link></li>
                        <li><Link to="GetDataV2">Hae kannasta v.2</Link></li>
                        <li><Link to="UpdateData">Päivitä kantaa</Link></li>
                        <li><Link to="Delete">Poista kannasta</Link></li>
                    </ul>
                </div>
                {this.props.children}
            </div>
        );
    }
}
```

Kuvio 27. Layout-komponentin lähdekoodi.

Kuvio 28. Sivun yläreunassa oleva navigaatiopalkki.

4.3 Oppilaan lisääminen

Kuviossa 29 on oppilaan lisäämiseen käytettävä sivu avattuna selaimessa. LISÄÄ TIETOKANTAAN -painike aktivoituu, kun käyttäjä on syöttänyt kaikkiin kenttiin tietoa.

Kuvio 29. Oppilaan lisäämiseen käytettävä sivu.

Kuviossa 30 on oppilaan lisäämiseen tehdyn komponentin lähdekoodista render-funktio, eli kuviossa 29 näkyvien kenttien ja painikkeen luominen.

```
render() {
    return (
        <div className="component">
            <br />
            <h4>Lisää oppilaita tietokantaan</h4>
            <br></br>
            <form id="uploadForm">
                <p><input onChange={this.nameChange.bind(this)} placeholder="Nimi" name="nimi"></input></p>
                <p><input onChange={this.ikaChange.bind(this)} placeholder="Ikä" name="ika"></input></p>
                <p><input onChange={this.ryhmaChange.bind(this)} placeholder="Ryhmä" name="ryhma"></input></p>
                <p><input onChange={this.OPChange.bind(this)} placeholder="Opintopisteitä" name="opintopisteita"></input></p>
                <p><input onChange={this.valmistuuChange.bind(this)} placeholder="Valmistuu" name="valmistuu"></input></p>
                <p><button disabled={this.state.invalidData} class="btn btn-success" type="submit" onClick={this.handleClick.bind(this)}>LISÄÄ TIETOKANTAAN</button></p>
            </form>
        </div>
    );
}
```

Kuvio 30. Oppilaan lisäämisen käytettävän komponentin render-funktio.

Kuviosta 30 nähdään, että LISÄÄ TIETOKANTAAN -painikkeeseen liitetään handleClick-funktio, kun käyttäjä painaa painikkeesta (kohta onClick). Kun kaikki kentät on täytetty ja käyttäjä painaa painikkeesta, handleClick-funktio ottaa tiedot kaikista kentistä (kuvio 31). Kun tiedot ovat tallessa, ne välitetään eteenpäin postToDb-funktiolle ja kentät tyhjätään selaimessa form.reset-komennolla (kuvio 31).

```
handleClick(e) {
    e.preventDefault();
    let form = document.getElementById("uploadForm");
    let data = serialize(form, { hash: true });
    this.postToDb(data);
    form.reset();
}
```

Kuvio 31. HandleClick-funktion lähdekoodi.

PostToDb-funktio käsittelee sille välitetyn datan ja lähettää sen eteenpäin palvelinpuolelle. Palvelimelle lähettämiseen on käytetty axios-nimistä moduulia. Axios lähettää tiedot Axios.post-komennolla tässä tapauksessa osoitteeseen http://localhost:3000/addStudent (kuvio 32). Virheen havaitsemiseen on käytetty catch-komentoa (kuvio 32).

```
postToDb(data) {
    Axios.post("http://localhost:3000/addStudent", {
        nimi: data.nimi,
        sukupuoli: data.sukupuoli,
        ikä: data.ika,
        ryhmä: data.ryhma,
        opintopisteitä: data.opintopisteita,
        valmistuu: data.valmistuu
    })
        .catch((error) => {
            console.log(error);
        });
}
```

Kuvio 32. PostToDb-funktion lähdekoodi.

4.4 Oppilaan skeema

Oppilaalle voidaan antaa seuraavia tietoja: nimi, sukupuoli, ikä, ryhmä, opintopisteitä ja valmistuu. Kyseiset tiedot on määritelty skeemaa luodessa (kuvio 33). Malli luodaan komennolla mongoose.model ja samalla määritetään kokoelman nimi MongoDB-tietokannassa. Tässä tapauksessa kokoelman nimeksi tulee students (kuvio 33).

```
var mongoose = require("mongoose");

oppilasSchema = new mongoose.Schema({
    nimi: String,
    sukupuoli: String,
    ikä: Number,
    ryhmä: String,
    opintopisteitä: Number,
    valmistuu: Number
});

var oppilas = module.exports = mongoose.model("students",oppilasSchema);
```

Kuvio 33. Oppilaan skeeman ja mallin luominen.

4.5 Tietokantaan lisääminen

Oppilaan lisääminen MongoDB-tietokantaan tapahtuu tässä ohjelmassa osoitteessa http://localhost:3000/addStudent. Komento app.post käsittelee kyseiseen osoitteeseen tulleen post-pyynnön. Palvelinpuolelle lähetetyt tiedot saadaan talteen req.body-komennolla (kuvio 34).

```
app.post("/addStudent", function (req, res) {
    var nimi = req.body.nimi;
    var spuoli = req.body.sukupuoli;
    var ikä = req.body.ikä;
    var ryhmä = req.body.ryhmä;
    var opintopisteitä = req.body.opintopisteitä;
    var valmistuu = req.body.valmistuu;
```

Kuvio 34. Post-pyynnön käsittely osoitteessa http://localhost:3000/addStudent.

Kun tiedot on otettu talteen, luodaan uusi oppilas mallin mukaisesti ja tallennetaan se tietokantaan komennolla save (kuvio 35). Tallentamisen jälkeen palvelin vastaa juuri tallennetun oppilaan tiedoilla komennolla res.json (kuvio 35).

```
var student = new oppilas({
    nimi: nimi, sukupuoli: spuoli, ikä: ikä, ryhmä: ryhmä, opintopisteitä: opintopisteitä,
    valmistuu: valmistuu
});
student.save(function (err, student) {
    if (err) return console.error(err);
});
res.json(student);
```

Kuvio 35. Uuden oppilaan luominen ja tallentaminen.

Tallentamisen jälkeen oppilas näkyy MongoDB-tietokannassa. Tietokantaa voidaan tutkia suoraan konsolista tai, kuten tässä tapauksessa, apuohjelman avulla. Apuohjelmana käytetty Robomongo näyttää tietokannan kokoelmat ja dokumentit graafisesti ja selkeästi. Skeemassa määriteltyjen kenttien lisäksi oppilaalle tulee _id eli uniikki tietokanta-id ja _v eli versioavain (kuvio 36).

_id	ObjectId("58d0f064e349c9060c89f747")	ObjectId
nimi	Jaska Jokunen	String
sukupuoli	Mies	String
ikä	23	Int32
ryhmä	tite14	String
opintopisteitä	155	Int32
valmistuu	2018	Int32
_v	0	Int32

Kuvio 36. Uusi oppilas tietokannassa.

5 Sisällönhallintasovellus

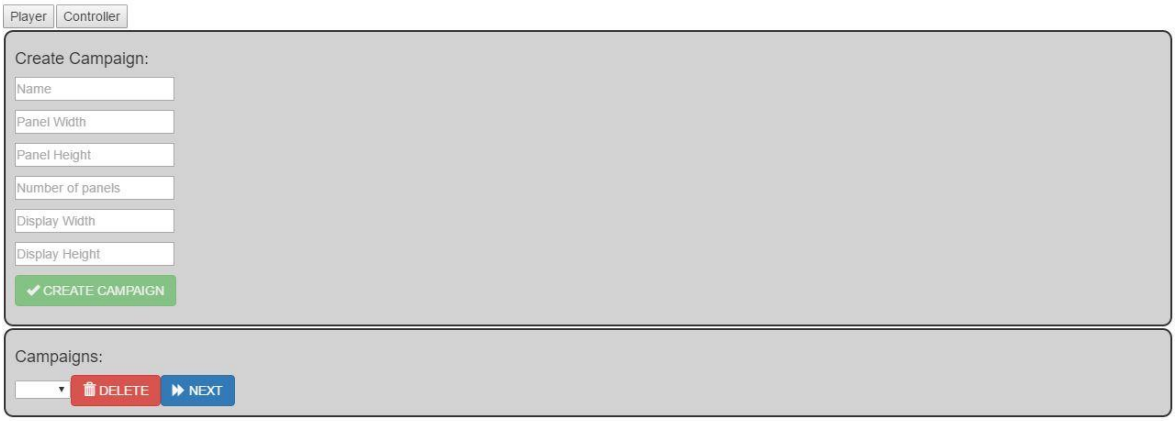
Tämän työn tarkoituksena oli perehtyä moderneihin web-ohjelmoinnissa käytettäviin tekniikoihin ja kieliin, sekä niitä käyttäen luoda sisällönhallintasovellus monipuoliseen mainostamiseen. Front-End-ohjelmointi toteutettiin käyttäen Reactia ja Reduxia, Back-End-ohjelmointiin käytettiin NodeJS-kirjastoa ja Expressiä. Tietokantana työssä toimi MongoDB. Edellä mainittujen tärkeimpien tekniikoiden lisäksi työssä käytettiin useita pienempiä moduuleja, joiden avulla tietyt toiminnot saatiin suoritettua helpommin. Versionhallinta toteutettiin Bitbucket.org-nimisellä verkkosivustolla. Ohjelmointi toteutettiin käyttämällä Visual Studio Code -editoria. Tietokannan tarkasteluun käytettiin RoboMongo-nimistä apuohjelmaa. Palvelinpuolen ohjelmoinnissa apuvälineenä käytettiin Postman-nimistä ohjelmaa, jolla voidaan lähettää tietoja NodeJS-ohjelmalle ja tutkia sen reaktioita.

5.1 Sovelluksen tarkoitus

Sisällönhallintasovelluksen tarkoitus on mahdollistaa monipuolinen mainostaminen erilaisilla näyttökokoonpanoilla. Käyttäjä voi luoda esimerkiksi yhden illan tapahtumaan kokonaisuuden, joka pyörittää mainoksia näytöillä käyttäjän haluamalla tavalla. Käyttäjä voi lisätä ja poistaa medioita, sekä määritellä niiden sijainnit esimerkiksi kymmenien ledipaneelien laitamainoskokoonpanossa.

5.2 Kampanjan luominen

Etusivulla kohdassa Create Campaign käyttäjä luo kampanjan esimerkiksi yhtä jalkapallo-ottelua varten. Käyttäjä määrittää kampanjalle mainosnäyttöjen resoluutiot, sekä määrän ja näytön resoluution, jolta mainokset luetaan ledipaneeleihin (kuvio 37). Käyttäjän painaessa Create Campaign -painikkeesta sovellus poimii tiedot kentistä ja lähettää ne palvelinpuolelle käyttäen axios-nimistä moduulia. Palvelinpuolella NodeJS lisää luodun kampanjan MongoDB-tietokantaan. Kohdassa Campaigns (kuvio 37), käyttäjä voi halutessaan valita aiemmin luodun kampanjan ja siirtyä seuraavalle sivulle hallitsemaan sen sisältöä.

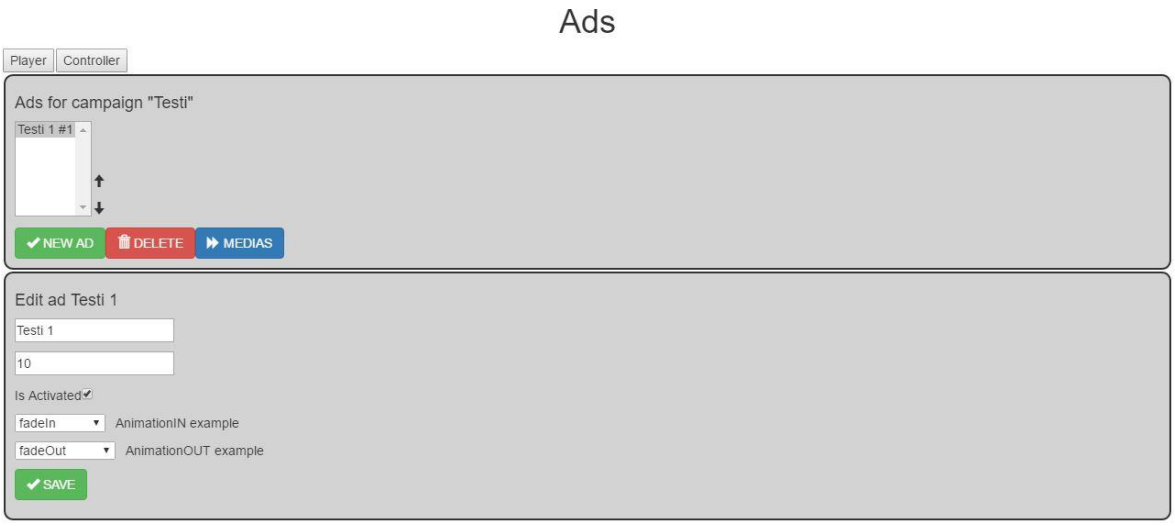


Kuvio 37. Sovelluksen etusivu. Kampanjan luominen.

Sovellus hakee kaikki kampanjat tietokannasta aina kun etusivu avataan tai päivitetään ja näyttää kampanjat dropdown-valikossa kohdassa Campaigns, mikäli aiemmin luotuja kampanjoita löytyy (kuvio 37). Kyseisessä kohdassa käyttäjä voi myös poistaa aiemmin luotuja kampanjoita. Kampanjan poistaminen poistaa kaikki siihen liittyvät dokumentit tietokannasta sekä mediakansiot.

5.3 Mainoksen luominen

Kun käyttäjä on luonut uuden kampanjan tai valinnut aikaisemmin luodun kampanjan, siirrytään Ads-sivulle eli mainosten hallintaan. Sivun ylemmässä osassa voidaan luoda uusi mainos, poistaa mainoksia, siirtyä hallinnoimaan valitun mainoksen medioita tai vaihtaa valitun mainoksen järjestysnumeroa nuolipainikkeilla (kuvio 38). Jos käyttäjä painaa New Ad -painikkeesta tai valitsee mainoksen, sivun alempi osa tulee näkyviin (kuvio 38). Medias-painikkeella päästään hallitsemaan valittuun mainokseen liittyviä medioita (kuvio 38).

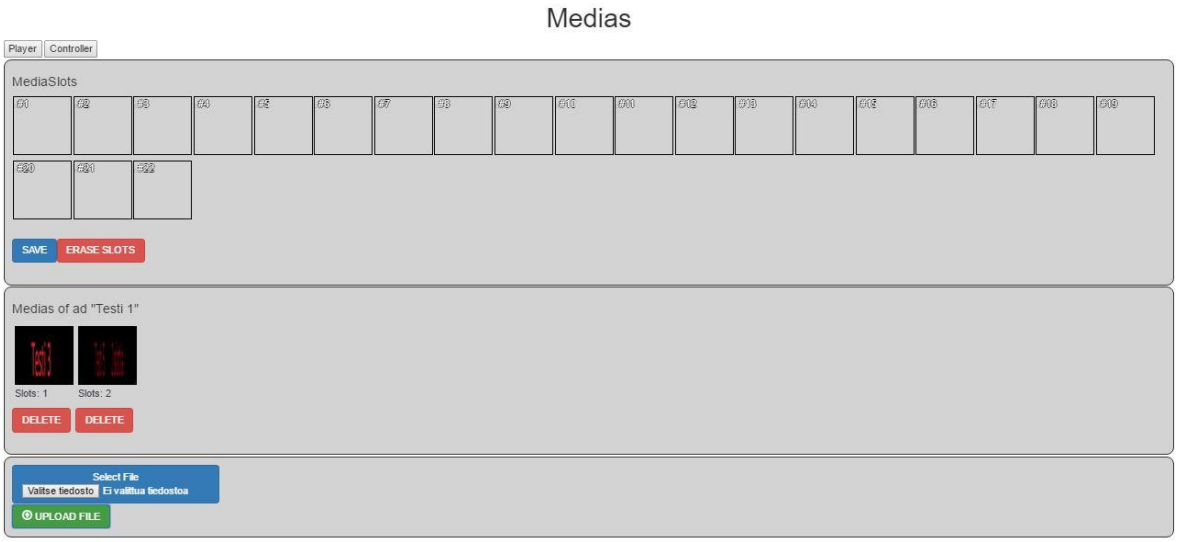


Kuvio 38. Mainosten hallinta.

Alemmassa osassa voidaan tallentaa mainos tietokantaan. Käyttäjä voi nimetä mainoksen, antaa sen keston sekunteina, valita onko mainos käytössä vai ei, sekä päättää animaatiot, joilla mainos tulee näkyviin ja lähtee pois (kuvio 38). Save-painikkeen painalluksen jälkeen kerätään jälleen tiedot kentistä ja tallennetaan mainos tietokantaan. Delete-painike poistaa valitun mainoksen, sekä siihen liittyvät dokumentit tietokannasta ja mediakansiot.

5.4 Medioiden hallinta

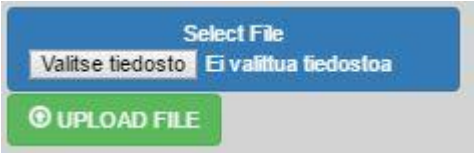
Medioiden hallintasivu koostuu kolmesta osiosta. Käyttäjä voi lisätä mediatiedostoja haluamalleen mainokselle ja sijoitella ne ledipaneeleihin (kuvio 39). Sivun ylimmässä osassa olevat laatikot kuvastavat ledipaneeleja. Käyttäjä on aiemmin saanut määrittää paneelien resoluutiot ja määrän. Sovellus laskee montako paneelia, eli tässä tapauksessa slottia, yksi mainos vie. Sovellus sallii vain sellaisia medioita, joiden resoluutio sopii paneelien kanssa yhteen, eli leveys ja korkeus ovat jaollisia paneelien leveydellä ja korkeudella.



Kuvio 39. Medioiden hallinta

5.4.1 Median lisääminen

Medias-sivun alaosassa on median lisäämisen käytettävät painikkeet. Valitse tiedosto -painikkeesta avautuu ikkuna, josta käyttäjä voi etsiä tiedostoja tietokoneeltaan ja valita lisättävän median (kuvio 40). Upload File -painikkeella kyseinen tiedosto tallennetaan palvelimelle omaan kansioonsa, sekä siitä tehdään myös pienempi versio eli thumbnail. Tietokantaan tallentuu median polut sen näyttämistä varten, paneeleihin sijoittamiseen liittyviä tietoja, sekä kampanja ja mainos, joiden alle media kuuluu.



Kuvio 40. Median lisääminen.

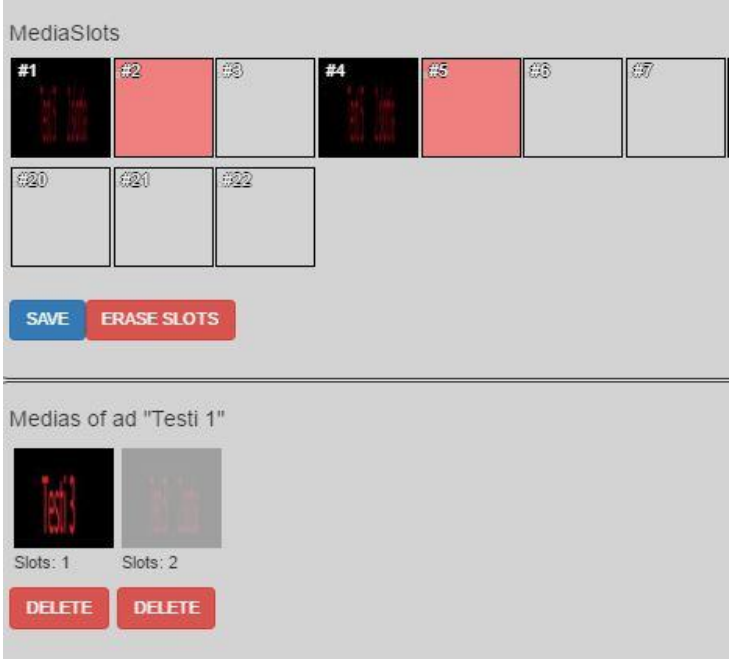
5.4.2 Median valitseminen

Medias-sivun keskiosassa näytetään kaikista mainokselle kuuluvista medioista pienennetyt versiot eli thumbnailit. Käyttäjä voi painaa sivulla median thumbnailista, jolloin se tulee valituksi (kuvio 41, vaalennettu), ja sen voi sijoittaa sivun yläosassa sijaitseviin laatikoihin eli slotteihin. Median alla oleva teksti Slots ilmoittaa montako slottia kyseinen media vie (kuvio 41). Delete-painikkeella voidaan poistaa kyseinen media. Poistamisen yhteydessä poistetaan media ja sen thumbnail kansioista, sekä mediaan liittyvä dokumentti tietokannasta.

Kuvio 41. Media valittuna.

5.4.3 Median sijoittelu

Median valitsemisen jälkeen se voidaan sijoittaa sivun yläosassa oleviin laatikoihin. Kun media on sijoitettu, käyttäjän klikkaamaan laatikkoon tulee kyseisen median thumbnail kuvastamaan, että media alkaa siitä kohdasta (kuvio 42). Thumbnailin jälkeen sovellus maalaa laatikoita punaiseksi kuvastamaan montako slottia sijoitettu media vie (kuvio 42). Save-painikkeella voidaan tallentaa medioiden sijoittelu tietokantaan. Erase Slots -painikkeella voidaan tyhjentää kaikki sijoittelut.

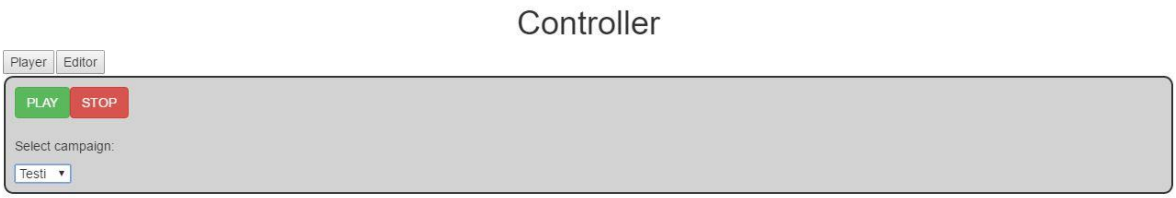


Kuvio 42. Media sijoitettuna paneeleille.

Tietokantaan tallentuu tässä vaiheessa kullekin medialle varatut slotit (kuvion 42 tapauksessa [”1-2”, ”4-5”]), sekä median alkamiskohdat ruudulla. Esimerkiksi jos media alkaa slotista yksi, sen alkamiskohta ruudulla on [0,0]. Alkamiskohtien laskemista varten tehty funktio katsoo, mistä slotista media alkaa ja laskee sen mukaan sijainnin ruudulla. Kyseinen funktio ei vielä toimi täysin oikein ja projekti on siinä kohtaa menossa.

5.5 Player ja Controller

Player- ja Controller-sivut ovat projektissa vasta suunnitteluasteella, joten kaikki toiminnallisuudet puuttuvat. Controller-sivulta käyttäjä voi valita aiemmin luomansa kampanjan ja pistää sen pyörimään. PLAY-painikkeesta kootaan JSON-tiedosto, johon haetaan medioiden näyttämiseen tarvittavat tiedot tietokannasta (kuvio 43). Tämän jälkeen controller lähettää Socket.IO-tekniikkaa käyttäen JSON-tiedoston playerille, joka osaa sitä käyttäen näyttää mainoksia.



Kuvio 43. Controller-sivu.

5.6 Ongelmat

Suuri osa projektin ongelmista johtui kokemattomuudesta. Etenkin Reactin kanssa tuli esille erilaisia ongelmia, koska osa yksinkertaisimmistakin toiminnoista täytyy toteuttaa Reactissa eri tavalla esimerkiksi perinteiseen JavaScriptiin verrattuna. Myös tiedon hankkiminen oli välillä vaikeaa, koska työ sisältää useita eri moduuleja. Tietoa etsiessä kävi usein ilmi, että löydetty ratkaisu koski jotain vanhempaa versiota, eikä se enää toiminut. Voidaankin tiivistää, että ongelmat johtuivat nopeasti muuttuvasta ympäristöstä ja kokemattomuudesta. Mihinkään suuriin teknisiin ongelmiin työn aikana ei törmätty.

6 Yhteenveto ja pohdinta

Opinnäytetyön tavoitteena oli perehtyä moderneihin web-ohjelmien luomiseen käytettäviin full stack -tekniikoihin, sekä toteuttaa niitä käyttäen uusi sisällönhallintasovellus LEDiMedialle. Sovelluksella pystyttäisiin luomaan erilaisia mainoskokonaisuuksia esimerkiksi jalkapallo-ottelujen ajaksi. Tärkein uudistus vanhaan versioon nähden oli sovelluksen dynaamisuus, eli sovelluksen piti toimia erilaisten näyttö- tai paneelikokoonpanojen kanssa.

Työ koostui hyvin pitkälti uusien asioiden opiskelusta ja tiedonhausta. Aluksi mietittiin tärkeimmät tekniikat front-endin ja back-endin ohjelmointia varten. Kustakin tekniikasta käytiin läpi perusteet erilaisten harjoitusten avulla. Kun katsottiin, että perusteet oli opittu, aloitettiin itse web-ohjelman kehittäminen. Työn lopputuloksena syntyi web-ohjelma, jossa tärkeimpinä tekniikoina esiintyy: ReactJS, Redux, NodeJS ja ExpressJS. Tietokannaksi työhön valikoitui MongoDB. Web-ohjelmaa saatiin tehtyä siihen asti, että käyttäjä voi luoda tapahtumia varten mainoskokonaisuuksia ja lisätä näytettäviä mainosmedioita. Kehitysvaiheeseen jäivät medioiden näyttäminen, kontrolleri, sekä player, joka toistaa luotuja mainoskokonaisuuksia.

Opinnäytetyön tekijällä ei ollut ollenkaan aiempaa kokemusta web-ohjelmien kehittämisestä. Web-ohjelman kehittämisen eteneminen pysähtyi usein, koska ongelmia esiintyi suhteellisen paljon. Ongelmat johtuivat useimmiten kokemattomuudesta tai nopeasti muuttuvasta ympäristöstä (versiot päivittyvät nopeasti). Sinnikkäällä tiedonhaulla ongelmiin löytyi yleensä ratkaisu.

Opinnäytetyön tekemisestä sai todella paljon käytännön kokemusta moderneilla tekniikoilla ohjelmoinnista, sekä web-ohjelmien kehittämisestä yleisesti. Erityisesti uusimmista tekniikoista, kuten ReactJS ja Redux, sai hyvän valttikortin työmarkkinoita silmällä pitäen.

Modernit full stack -tekniikat mahdollistavat kevyiden ja nopeiden web-ohjelmien luomisen. Esimerkiksi ReactJS mahdollistaa yhden sivun web-ohjelmien luomisen, eli kullekin sivulle ei tarvitse tehdä omaa HTML-tiedostoa. Tämä tarkoittaa sitä, että

ohjelma osaa tiettyjen asioiden mukaan näyttää oikeat asiat samalla sivulla. Tämän ansiosta ohjelmat toimivat miellyttävän nopeasti selaimessa.

Opinnäytetyössä käytettävien tekniikoiden osaajille on tälläkin hetkellä paljon kysyntää ja tulevaisuudessa varmasti vielä enemmän, kun yritykset siirtyvät vanhemmista tekniikoista pois. Web-ohjelmien kehittäminen on trendien mukana menevä ala. Tulevien vuosien aikana mennään varmasti paljon eteenpäin ja uusia tekniikoita nousee suosioon.

LÄHTEET

Angular. 2017. Webpack: An introduction. [www-dokumentti]. Angular. [Viitattu 14.3.2017]. Saatavissa: https://angular.io/docs/ts/latest/guide/webpack.html

Borland, B. 2014. Pentaho Analytics for MongoDB. [Verkkokirja]. Packt Publishing. [Viitattu 30.1.2017]. Saatavissa Ebrary-tietokannasta. Vaatii käyttöoikeuden.

Express. Ei päiväystä. Express Routing. [www-dokumentti]. Node.js Foundation. [Viitattu 6.2.2017]. Saatavissa: http://expressjs.com/en/guide/routing.html

Express. Ei päiväystä. Express, fast, unopinionated, minimalist web framework for Node.js. [www-dokumentti]. Node.js Foundation. [Viitattu1.2.2017]. Saatavissa: http://expressjs.com/

LEDiMedia. Ei päiväystä. LEDiMedia. [www-dokumentti]. LEDiMedia Oy. [Viitattu 13.4.2017]. Saatavissa: http://ledimedia.fi/

Mongoose. Ei päiväystä. Mongoose Guide. [www-dokumentti]. LearnBoost, Inc. [Viitattu 1.2.2017]. Saatavissa: http://mongoosejs.com/docs/guide.html

Rai, R. 2013. Socket.io Real-time Web Application Development. [Verkkokirja]. Packt Publishing. [Viitattu 18.1.2017]. Saatavissa Ebrary-tietokannasta. Vaatii käyttöoikeuden.

React. 2017. React, a JavaScript library for building user interfaces. [www-sivu]. Facebook Incorporation. [Viitattu 20.2.2017]. Saatavissa: https://facebook.github.io/react/

React-Router. 2017a. React-Router Introduction. [www-dokumentti]. GitHub Incorporation. [Viitattu 2.3.2017]. Saatavissa: https://github.com/ReactTraining/react-router/blob/master/docs/Introduction.md

React-Router. 2017b. React-Router route configuration. [www-dokumentti]. GitHub Incorporation. [Viitattu 2.3.2017]. Saatavissa: https://github.com/ReactTraining/react-router/blob/master/docs/guides/RouteConfiguration.md

Redux JS. 2017. Actions basics. [www-dokumentti]. GitBook. [Viitattu 7.3.2017]. Saatavissa: http://redux.js.org/docs/basics/Actions.html

Redux JS. 2017. Three Principles. [www-dokumentti]. GitBook. [Viitattu 7.3.2017]. Saatavissa: http://redux.js.org/docs/introduction/ThreePrinciples.html

Teixeira, P. 2013. Instant Node.js Starter. [verkkokirja]. Packt Publishing. [Viitattu 6.2.2017]. Saatavissa Ebrary-tietokannasta. Vaatii käyttöoikeuden.

WebSocket. 2017. About HTML5 WebSocket. [www-dokumentti]. Kaazing corporation. [Viitattu 16.1.2017]. Saatavissa: http://websocket.org/aboutwebsocket.html