YKSISIVUISEN WEB-SOVELLUKSEN TOTEUTUS

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Ammattikorkeakoulututkinnon opinnäytetyö

Tietojenkäsittelyn koulutusohjelma

Syksy 2020

Kerkko Kiikeri

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietojenkäsittelyn koulutusohjelma
Hämeenlinnan korkeakoulukeskus

Tekijä Kerkko Kiikeri **Vuosi** 2020

Työn nimi Yksisivuisen web-sovelluksen toteutus

Työn ohjaaja/t Tommi Lahti

TIIVISTELMÄ

Tämän opinnäytetyön tavoite oli kehittää omia web-kehitys taitoja tehden yksisivuinen web-sovellus käyttäen moderneja, suosittuja ja itselle uusia tekniikoita ja teknologioita. Aiheen valintaa vaikutti eniten oma kiinnostus web-kehitykseen.

Opinnäytetyön teoriaosuudessa esitellään työlle oleellisia JavaScript-ominaisuuksia, jotka liittyvät tiedonsiirtoon sekä selaimessa että palvelimella. Osuudessa käydään läpi myös palvelimen toteutukseen käytettyjä teknologioita ja lisäohjelmistoja.

Opinnäytetyön käytännön osuudessa käydään läpi käytettyjen ohjelmistojen asennus ja käyttöönotto, sovelluksen sekä selainpuolelle että palvelimelle tehdyt ratkaisut ja niiden tarkoitus sovelluksessa. Lopuksi sovellusta testataan käyttämällä sen ominaisuuksia samalla tulostaen sen tietoliikennettä palvelimen terminaaliin.

Avainsanat Web-sovellus, palvelin, REST, JavaScript, Node.js

Sivut 24 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Business Information Technology
Hämeenlinna University Centre

Author Kerkko Kiikeri **Year** 2020

Subject Development of a single-page web application

Supervisors Tommi Lahti

ABSTRACT

The authors goal for this thesis was to improve his web-developing skills by making a single-page web application using methods and technologies that are modern, popular, and new to him. The main thing that made him choose this subject was his own interest towards web-developing.

The theory part of this thesis showcases essential JavaScript features about data transfer between the web browser and the server. The part also presents technologies and middleware that were used to create the server.

The practical part of this thesis includes installation and configuration of used software, the solutions for both front-end and back-end as well as the meaning of them within the web application. At the end the web application is being tested by using its features and printing its data transfer to the terminal of the server.

Keywords Web application, server, REST, JavaScript, Node.js

Pages 24 pages

SISÄLLYS

KÄSITELUETTELO

REST	Arkkitehtuurimalli tiedonsiirto rajapintojen toteuttamiseen
Väliohjelmisto	Termi palvelimella käytettäville lisäosille
Olio	Tietotyyppi, joka voi sisältää useita erilaisia muuttujia ja funktioita samanaikaisesti
JSON	JavaScript object notation, JavaScript-olio tiedostomuoto
MongoDB	MongoDB Inc:n kehittämä dokumenttikeskeinen tietokanta

1 JOHDANTO

Tämän opinnäytetyön tavoitteena on tutustua ja oppia käyttämään suosittuja web-kehitys teknologioita tekemällä niitä käyttäen web-sovellusta. Sovellus tulee sisältämään yksisivuisen nettisivun, palvelimen ja tietokannan. Valmista sovellusta testataan käyttäen sen ominaisuuksia samalla tulostaen sovelluksen tietoliikennettä palvelimen terminaaliin.

Teoriaosuudessa esitellään sovelluksen kannalta tärkeät osat käytetyistä teknologioista. Osuus sisältää teoriaa JavaScriptin opinnäytetyön kannalta tärkeistä ominaisuuksista sekä Node.js:n toiminnasta ja lisäosista.

Toteutuksessa käydään läpi tarvittavat asennukset, niiden kokoonpano, sovelluksen tekeminen ja käytetyt ratkaisut tietovirran kannalta priorisoidussa järjestyksessä. Palvelinosuudessa on sekä itse palvelimen toteutus että lisäosien tuomia metodeja ja käyttötapoja. Selainpuolen toteutus sisältää lyhyesti sovelluksen visualisoinnin, käyttäjän tiedon luonnin, sen muuttamisen HTML-elementeiksi sekä tiedon siirtämisen ja saamisen palvelimelta.

Lopuksi sovellusta testataan käyttäen siihen tehtyjä ominaisuuksia. Siinä testataan uusien sivujen luomista, kuvien lähetystä ja tekstin muokkausta.

Opinnäytetyö vastaa seuraaviin tutkimuskysymyksiin:

- Miten Node.js:llä luodaan palvelin?
- Miten luodaan REST-palvelu käyttäen Node.js ja express.js?
- Miten liikuttaa tietoa fetch apin ja REST-palvelun välillä?

2 YKSISIVUINEN WEB-SOVELLUS

Yksisivuisella tarkoitetaan nettisivua, joka jatkuvasti kokonaan uuden sivun palvelimelta lataamisen sijaan hakee palvelimelta tiettyjä tietoja käyttäjän syötteen mukaan, ja uudelleenrakentaa sivun paikallisesti selaimessa saatujen tietojen avulla (Neoteric 2016).

Web-sovellus tarkoittaa sovellusta, jonka käyttöliittymänä toimii selain. Web-sovelluksia on yksisivuisia ja monisivuisia. Suurin etu perinteiseen työpöytäsovellukseen verrattuna on yhteensopivuus; laite, joka tukee modernia selainta, tukee myös web-sovellusta. Yksisivuinen web-sovellus voi myös muistuttaa hyvin paljon mobiilisovellusta, jolloin ne voivat käyttää samaa palvelinta (Neoteric 2016).

Opinnäytetyön web-sovellus on ravintola menun luomiseen tarkoitettu sivu, jossa voi luoda, muokata ja poistaa tuotteita listasta. Listaan luotua tuotetta painamalla tuote avautuu muokkaustilaan, jossa voi muokata sen nimeä, kuvaa ja kuvausta.

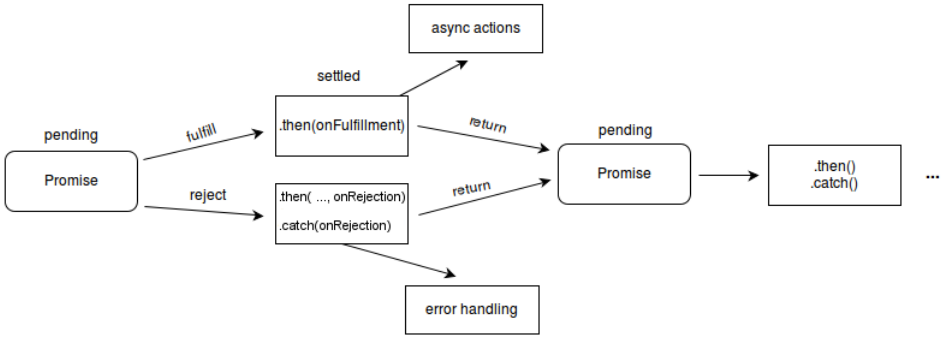
Opinnäytetyön web-sovellus sisältää muokkaustilan ravintolan henkilökunnalle. Web-sovellus ei sisällä ravintolan asiakkaiden käyttöön sopivaa katselutilaa tai ominaisuuksia. Web-sovelluksen käytöstä kerrotaan lisää kappaleessa 6.

3 JAVASCRIPT WEB-KEHITYKSESSÄ

JavaScript tunnetaan yleisesti selaimessa ajettavana kielenä, mutta Node.js:n, React Nativen ja Electron.js:n myötä kielellä voi kehittää myös palvelimia, mobiilisovelluksia ja työpöytäsovelluksia. Kielelle on myös monia kehyksiä, kuten React.js, Vue.js ja Angular.js, jotka laajentavat sen kehitysmahdollisuuksia. JavaScript oli suosituin ohjelmointikieli vuonna 2019 stackoverflow developer surveyssä seitsemättä vuotta putkeen (Adeyefa, O. 2019).

3.1 Promise

JavaScriptissä on web-kehitykselle tärkeitä ominaisuuksia, joista yksi on ”promise”. Promise on olio, jolla on tila, joka voi olla joko “pending”, “fulfilled” tai “rejected”. Jos promise odottaa pyydettävää tietoa, on tila “pending”, tiedon saatua tila on “fulfilled” ja virheen tapahtuessa “rejected” (Kuva 1). Promisen käyttö on tärkeää web-kehityksessä, koska tietoliikkeen nopeus on hyvin vaihtelevaa ja välillä tieto ei ole saatavilla, jotka ovat ongelmia, joiden ratkaisuun promise on suunniteltu (MDN web docs 2020c).



Kuva 1. Kaavio promisen toiminnasta. (MDN web docs 2020c)

3.2 Fetch api

Opinnäytetyön web-sovelluksessa hyödynnettävä fetch api on osa JavaScriptiä. Sen tarkoitus on lähettää REST-palvelulle pyyntöjä tiedostojen kera. Kysely pyynnön tehtyä fetch palauttaa promisen. Kun tieto on saapunut, promisen tila-arvo on ”fulfilled” ja se sisältää palvelimelta saadun olion (MDN web docs 2020a).

3.3 **Async**

Async-funktion ja await-avainsanan tarkoitus on pysäyttää funktion, kunnes await-avainsanalla merkatun työn palauttama promise on saatu selvitettyä. Await toimii vain async-funktion sisällä. Opinnäytetyön web-sovelluksessa await-avainsanaa käytetään, kun tieto liikkuu palvelimen ja selaimen välillä (MDN web docs 2020b).

4 NODE.JS

Node.js on Googlen v8 JavaScript-moottorin avulla rakennettu avoimen lähdekoodin JavaScript-suoritin. Sen pääkäyttötarkoitus on luoda palvelimia, mutta sitä voi käyttää myös terminaalin kaltaisena suorittimena. Node.js:n yksisäikeisyydestä huolimatta se suorittaa käskyjä tehokkaasti sen esteettömän tapahtumasilmukan ansiosta. Esteettömällä tapahtumasilmukalla tarkoitetaan sitä, että tehtävää, joka ei ole vielä valmis suoritettavaksi ei jäädä odottamaan, vaan se pusketaan sivuun, jotta muita valmiita tehtäviä voidaan suorittaa odottamisen sijaan. Tehtävän valmistuttua se lisätään takaisin tapahtumasilmukkaan suoritettavaksi (Nodejs.dev n.d.).

tarkoitetaan sitä, että tehtävää, joka ei ole vielä valmis suoritettavaksi ei jäädä odottamaan, vaan se pusketaan sivuun, jotta muita valmiita tehtäviä voidaan suorittaa odottamisen sijaan. Tehtävän valmistuttua se lisätään takaisin tapahtumasilmukkaan suoritettavaksi.

Node package manager (npm) on Node.js:n mukana tuleva pakettienhallintatyökalu. Npm toimii Node.js:n komentorivillä, jossa sillä voi ladata, asentaa ja poistaa sen asentamia paketteja. Asentaessa voi päättää, että asennetaanko paketti yleisesti vai tiettyyn projektiin. Näin voi jättää esim. testaukseen liittyvät paketit pois projektin tiedostoista (Pablo, R. 2019).

4.1 Express.js

Express.js on Node.js:lle tehty web-kehitys kehys. Express.js tarjoaa monia työkaluja palvelimen rakentamiseen, kuten reititinmetodin, jolla voi luoda REST-palvelun, sekä jäsentimen, jolla voi jäsentää tulevia JSON-tiedostoja käsiteltävään muotoon.

Express.js:n reititin tukee post-, delete-, patch-, put- ja get -metodeja. Kaikki metodit ottavat parametriksi osoitteen, jolla metodia kutsutaan palvelimen ulkopuolelta. Osoite voi olla pelkkä “/” jolloin reititintä kutsutaan sivun osoitteella. Tarpeen vaatiessa osoitteeseen voi lisätä muuttujia, esim. “/data/:id”, jossa tuleva pyyntö pyytää data -osoitteesta jotain id:llä, jota kaksoispisteen ansiosta voidaan käyttää palvelimella parametrinä esim. hakiessa tietoa tietokannasta (Express guide 2017).

Reititin metodi vaatii myös funktion, jolle annetaan parametrit req (request) ja res (response). Funktiossa päästään käsiksi pyynnön tietoihin käyttäen reqestia. Vastaanotettuun JSON-tiedostoon päästään käsiksi

käyttäen express.jsonin tarjoamaa body-jäsennintä, esim. “req.body._id”. Osoitteen parametreihin päästään käsiksi käyttäen “req.params” ja tiedostoihin “req.file”. Saadakseen funktio loppuun, on sen palautettava jotain käyttäen responsea (Express guide 2017).

4.2 **Mongoose.js**

Mongoose.js on Node.js:lle tehty lisäosa MongoDB:n käytön yksinkertaistamiseksi Node.js:n kanssa. Mongoose.js:n pääominaisuudet ovat kaavioiden ja mallien luonti ja MongoDB-tietokantaan tehtävät kyselyt. Kaavioissa ennalta määritellään mitä MongoDB-tietokantaan menevä JSON-olio tulee sisältämään. Kaaviolla voi asettaa olion arvoille vaatimuksia ja lisätä niihin tietoa. Tietokantakyselyt tehdään Mongoose.js:n tarjoamilla metodeilla. Näille metodeille annetaan olio, jonka mukaan tietoa kysytään tietokannasta (MDN web docs 2020d).

4.3 **Multer**

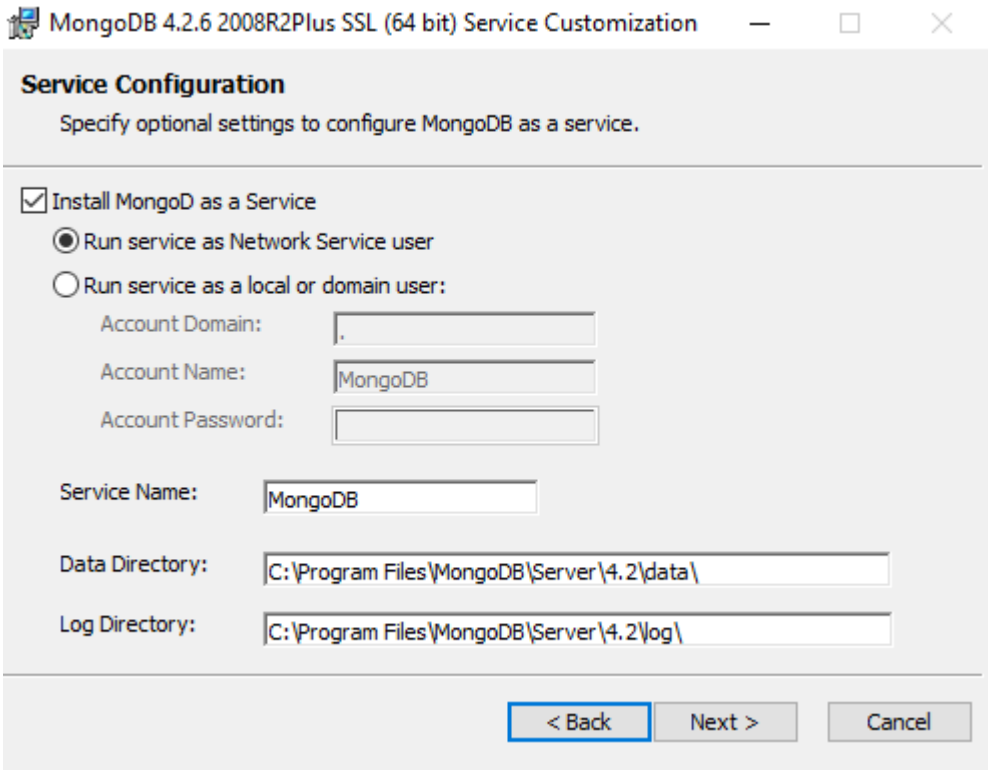
Multer on Node.js:lle tehty väliohjelmisto palvelimen vastaanottamien tiedostojen käsittelemiseen. Multer antaa vastaanotetuiden tiedostojen tiedot käsiteltäväksi koodista käsin, sekä tarjoaa monipuoliset metodit tiedon varastointiin. Näillä metodeilla voi mm. asettaa rajan vastaanotettavien tiedostojen koolle, siirtää tiedoston palvelimella olevaan kansioon ja uudelleennimetä tiedoston (expressjs/multer n.d.).

5 SOVELLUKSEN TOTEUTUS

Tässä luvussa käydään läpi sovelluksen toteutus sen tietoliikenteen kannalta priorisoidussa järjestyksessä.

5.1 Tietokanta

Paikallinen MongoDB-tietokanta luodaan asentamalla MongoDB Community Server. Se asennetaan Windows-palveluna (Kuva 2), joka käynnistää mongoDB:n kirjautuessa windowssiin.



Kuva 2. Paikallisen MongoDB-tietokannan asennus Windows-palveluna.

Palvelun käynnistyttyä tietokannan tiedostoille luodaan kansio C:\data\db, joka otetaan käyttöön komentorivillä. Komentorivillä annetaan mongod.exe:lle parametrinä tietokannalle luotu kansio komennolla: "C:\Program Files\MongoDB\Server\4.2\bin\mongod.exe" --dbpath="c:\data\db".

Työlle luodaan tietokanta ”db” avaamalla mongo.exe komentorivillä komennolla: "C:\Program Files\MongoDB\Server\4.2\bin\mongo.exe", ja syöttämällä sille komento ”use db”.

5.2 Palvelimen toteutus

Node.js-palvelimen toteutus alkaa sen asentamisella. Node.js:llä luodaan palvein käyttäen npm-komentoa ”npm init”, joka luo projektikansioon tarvittavat tiedostot sekä antaa vaiheittaisen kokoonpanokyselyn. Kokoonpanokyselyyn annetut arvot menevät ”package.json” nimiseen tiedostoon, joka sisältää myös palvelimen riippuvuudet.

Seuraavaksi palvelimelle asennetaan lisäosat Express.js, Mongoose.js ja Multer, komennolla ”npm install express mongoose multer”, joita tullaan käyttämään palvelimella. Npm sijoittaa lisäosat riippuvaisiksi automaattisesti package.json-tiedostoon.

Palvelimen ohjelmoinnin aloittamiseksi avataan npm init:n luoma app.js-tiedosto, joka on palvelimen pääprosessi. Asetetaan asennetut lisäosat vaadituiksi asettamalla ne vakioiksi, alustetaan Express.js sekä luodaan vakiot tietokantayhteyttä ja palvelimen porttia varten (Kuva 3).

```
2   const   express = require('express'),
3           mongoose = require('mongoose'),
4           app = express(),
5           url = 'mongodb://127.0.0.1:27017/db',
6           port = 3000;
7
```

Kuva 3. Asennetut paketit ja tarpeelliset arvot vakioina.

Palvelimelle pääsy vaatii portin, jota palvelin kuuntelee. Portti annetaan palvelimen express.listen-metodille, jonka parametriksi syötetään portti.

5.2.1 Tietokantayhteys

Yhteys paikalliseen MongoDB-tietokantaan luodaan Mongoose.js:n tarjoamalla metodilla mongoose.connect, jolle annetaan parametreiksi aikaisemmin luotu osoite. Tietokantayhteyttä voidaan testata metodilla mongoose.once, joka pyrkii yhdistämään tietokantaan ja sulkee yhteyden sen saatuaan.

5.2.2 Väliohjelmistot

Seuraavaksi annetaan palvelimelle käyttöön väliohjelmistoja. Väliohjelmistot otetaan käyttöön express.use-metodilla (Kuva 4). Express.json on tulevia JSON-tiedostoja jäsentävä väliohjelmisto. Se mahdollistaa tulevien JSON-tiedostojen tietojen käytön palvelimella. Express.static:lla luodaan palvelimelle kansio, jonka tiedostoihin sovellusta käyttävä henkilö pääsee käsiksi, eli kansio tulee sisältämään kaikki selainpuolella käytettävät

tiedostot. Väliohjelmiston ensimmäinen parametri viittaa selaimessa käytettävään url-osoitteeseen, josta kansion sisältöön pääsee käsiksi. Myös reitittimet sisältävä tiedosto lisätään väliohjelmistoksi.

```
21  app.use(express.urlencoded({ extended: true }));
22  app.use(express.json());
23  app.use('/listItems', listItemRoute);
24  app.use('/site', express.static('site'));
```

Kuva 4. Väliohjelmistojen lisäys pääprosessiin.

5.2.3 Mongoose.js-malli

Tiedon lisäämiseksi, muokkaamiseksi ja poistamiseksi tarvitaan malli, jota tietokantaa muokkaavat metodit noudattavat. Mongoose.js tarjoaa metodin mongoose.Schema MongoDB-mallin luomiseksi. Malli on JSON-olio tyyppinen, jolle annetaan avaimia (arvon nimi) ja niille arvoja (tietotyyppi, vaatimuksia). Aluksi asetetaan Mongoose.js vaadituksi asettamalla se vakioon. Sovellus tarvitsee kentät ”title”, ”imgurl” ja ”texts”, joiden kaikkien arvot ovat String-tyyppisiä. Lopuksi tehdään kaaviosta malli, annetaan sille nimi ja annetaan se vietäväksi (Kuva 5).

```
3   const listItemSchema = mongoose.Schema({
4       title: String,
5       imgurl: String,
6       texts: String
7   }, {versionKey: false
8   });
9
10
11  module.exports = mongoose.model('ListItems', listItemSchema);
```

Kuva 5. Mongoose.js scheman ja mallin luonti.

5.2.4 Reitittimet

REST-palvelu luodaan Express.js:n tarjoamalla express.Router-metodilla. Reitittimille luodaan uusi tiedosto, joka alkaa vaatimuksilla: Express.js reititintä varten, Multer-väliohjelmiston kuvatiedostojen käsittelyyn, aikaisemmin luotu Mongoose.js-malli tietojen tietokantaan viemistä varten sekä path-väliohjelmiston kuvatiedostojen tietotyypin saamiseen.

Kuvatiedostojen säilömistä ja uudelleen nimeämistä varten tarvitaan kansio ja metodi, joka käsittelee kuvien uudelleen nimeämisen. Kansion luotua käytetään multer.diskStorage-metodia, jolle annetaan kansion sijainti ja metodi tiedoston uudelleen nimeämistä varten. Tiedostot uudelleen nimetään estämään käyttäjien lähettämien mahdollisesti saman nimisten

tiedostojen tuomia ongelmia. Metodi palauttaa nimen, joka koostuu ”img-” alusta, tämänhetkisestä kellonajasta sekä tiedoston päätteestä (Kuva 6).

```
const storage = multer.diskStorage({
    destination: './site/img',
    filename: (req, file, cb) => {
        cb(null, file.fieldname + "-" + Date.now() +
        path.extname(file.originalname));
    }
});
```

Kuva 6. Multer-väliohjelmistolla varaston luominen ja kuvien uudelleen nimeäminen.

Kuvatiedostojen vastaanottoa käsitteleväksi metodiksi luodaan upload-metodi, joka ottaa arvoksi aikaisemmin luodun diskStorage-metodin (Kuva 7).

```
const upload = multer({
    storage: storage
});
```

Kuva 7. Multer-väliohjelmistolla luodulle upload-metodille annetaan kuvassa 6 luotu varasto.

Express.js:n reititin tarjoaa metodit: get, post, delete, patch ja put, joita käytetään REST-palvelun tekemiseen. Aloitetaan get-metodilla. Get kutsutaan, kun sovellus tarvitsee tietoa palvelimelta. Metodi hakee kaikki Mongoose.js-mallin mukaiset oliot tietokannasta käyttäen sen find-metodia, jonka jälkeen palauttaa ne käyttäjälle yhtenä JSON-tiedostona (Kuva 8).

```
router.get('/', async (req, res) => {
    try {
        const items = await ListItem.find();
        res.json({items});
    } catch (err) {
        res.json({ message: err });
    }
});
```

Kuva 8. Reititin, joka hakee tietoa tietokannasta ja lähettää sen selaimeen.

Seuraavaksi tehdään get-metodi, joka palauttaa yhden mallin mukaisen olion sen id:n mukaan. Id saadaan selaimelta parametrinä, ja sitä käytetään Mongoose.js:n metodissa findById.

Uuden olion luominen tietokantaan tapahtuu post-metodissa. Metodissa luodaan uusi Mongoose.js-mallin mukainen olio, jolle annetaan arvot metodin saamasta JSON-tiedostosta. JSON-tiedostoon päästään käsiksi

käyttäen express.json-jäsennintä. Olion saamiseksi tietokantaan käytetään Mongoose.js:n save-metodia (Kuva 9).

```
45  router.post('/', async (req,res) => {
46      const listItem = new ListItem({
47          title: req.body.title,
48          imgurl: req.body.imgurl,
49          texts: req.body.texts
50      });
51      try{
52          const savedListItem = await listItem.save();
53          res.json(savedListItem);
54      }catch(err){
55          res.json({message: err});
56      }
57  });
```

Kuva 9. Olion luonti Mongoose.js-mallilla ja sen tallennus tietokantaan.

Tietokannasta olioita poistaessa käytetään reitittimen delete-metodia. Metodi ottaa id:n parametrinä saadusta osoitteesta, ja käyttää sitä Mongoose.js:n metodissa deleteOne.

Reitittimen patch-metodissa yli kirjoitetaan jo tietokannassa olevien olioiden arvoja. Olion tiedot päivitetään Mongoose.js:n updateOne-metodilla, johon syötetään MongoDB:n päivitys operaation syntaksia (Kuva 10). Päässtääkseen käsiksi olioon, vaatii metodi sekä osoitteesta saatavan id:n, että olion, joka sisältää päivitettävät tiedot.

```
71  router.patch('/texts/:itemId', async (req, res) => {
72      try {
73          const updatedListItem = await ListItem.updateOne(
74              { _id: req.params.itemId },
75              { $set: {
76                  texts: req.body.texts
77              }});
78          res.json(updatedListItem);
79      } catch (error) {
80          console.log(error);
81      }
82  });
```

Kuva 10. Mongoose.js:n updateOne-metodin käyttöä.

Myös kuvien vastaanottaminen vaatii oman reititinmetodinsa. Kuvia vastaanottaessa käytetään post-metodia, ja kuva annetaan aikaisemmin luodulle upload-metodille jo post-metodin parametrinä ”upload.single”. Jotta kuva saadaan näkymään oikeassa osoitteessa, luodaan reititin metodin sisälle myös updateOne-metodi. Siinä käytetään parametrinä saatua id:tä ja kuvatiedoston mukana tullutta reitti arvoa muuttamaan id:tä vastaavan olion imgurl-arvoa (Kuva 11).

```
99  router.post('/upload/:itemId', upload.single("img"), async (req,res) => {
100     try {
101         const updatedListItem = await ListItem.updateOne(
102             { _id: req.params.itemId },
103             { $set: {
104                 imgurl: req.file.path
105             }});
106         res.json(updatedListItem);
107     } catch (error) {
108         console.log(error);
109     }
110 });
```

Kuva 11. Reititin, joka vastaanottaa kuvan ja lisää sen osoitetta vastaavaan tuotteeseen.

5.3 **Selainpuolen toteutus**

Selainpuolen toteutukseen tarvittavat HTML-, CSS- ja JavaScript-tiedostot luodaan express.static:lle asetettuun kansioon, jotta sivulle pääsee palvelimen osoitteella.

5.3.1 Staattisen sivun visualisointi

Toteutus alkaa visuaalisella muotoilulla ja staattisen HTML-sivun tekemisellä. Sivu jaetaan kolmeen osaan: ylätunnisteeseen, vasemman ja oikeanpuolisiin ruutuihin (Kuva 12).

Kuva 12. Sivu jaettu kolmeen osaan.

Luodaan sivulle myöhemmin scriptin luomaa sisältöä muistuttava sisältöä, jotta ne voidaan visualisoida ja sijoitella tässä vaiheessa (Kuva 13).

Kuva 13. Ruutuihin lisätty staattista sisältöä sijoittelua varten.

5.3.2 Olioiden luonti ja lähettäminen palvelimelle

Koska sivun sisältö perustuu olioiden sisältämiin tietoihin, aloitetaan selainpuolen koodaus olion luovalla funktiolla (Kuva 14). Funktio luo Mongoose.js-mallin mukaisen olion ja antaa sen avaimille vakio arvot. Oliot lähetetään palvelimelle fetch apia käyttävällä SetObj-funktiolla (Kuva 15). SetObj-funktiota kutsutaan await-avainsanan kanssa, joka takaa sen, ettei tämänhetkistä funktiota jatketa ennen kuin SetObj-funktio on saanut työnsä valmiiksi. Lopuksi kutsutaan Load-funktiota (Kuva 16), joka saa palvelimelta kaikki oliot ja näyttää ne vasemmassa ruudussa. Olio täytyy tallentaa tietokantaan ennen, kun sen tietoja käytetään HTML- elementtien luomisessa, koska olion mennessä tietokantaan MongoDB luo sille uniikin id:n, jota käytetään parametrinä sivulta kutsuttavissa metodeissa.

```
87  async function AddObj() {
88
89      let obj = {
90          title: "title",
91          imgurl: "",
92          texts: ""
93      }
94
95      await SetObj(obj);
96      Load();
97  }
```

Kuva 14. Tuoteolion luova AddObj-funktio.

SetObj-funktio (Kuva 15) käyttää fetch apia lähettääkseen post-pyynnön palvelimelle. Fetch ottaa parametreiksi reitittimen osoitteen, olion, joka sisältää pyynnön tyypin ja lähetettävän tiedostotyypin sekä itse tietokantaan vietävän olion. Pyynnön valmistuttua kutsutaan Promise.then-metodia, jossa lähetetään pyyntö palvelimelle.

```
23  async function SetObj(obj){
24      await fetch(url,{
25          method: 'POST',
26          headers: {
27            'Accept': 'application/json',
28            'Content-Type': 'application/json'
29          },
30          body: JSON.stringify(obj)
31      })
32      .then((res) => res.json())
33      .catch((err) => { console.error("Error: ", err);});
34  }
```

Kuva 15. Palvelimelle olion lähettävä SetObj-funktio.

5.3.3 Oliot palvelimelta HTML-elementeiksi

Sovelluksen käyttäjän tullessa sivulle halutaan, että kaikki sisältö tulee valittavaksi vasemmanpuoliseen ruutuun. Luodaan siis tätä käsittelevä Load-funktio (Kuva 16), joka hakee kaikki mallin mukaiset oliot fetch apia käyttävällä GetAllObjs-funktiolla (Kuva 18) ja tallentaa ne muuttujaan. Ennen sisällön luomista saatujen olioiden avulla poistetaan kaikki vasemman ruudun lapsielementit ClearLeftCont-funktiolla (Kuva 19). Sisältö luodaan silmukalla, jossa kutsutaan CreateListItem-funktiota (Kuva 20) kaikilla muuttujan olioilla. Vasempaan ruutuun luodun listan loppuun luodaan nappi uusien olioiden luomista varten.

```
146   async function Load() {
147       let objs = await GetAllObjs();
148
149       ClearLeftCont();
150
151       if(objs.items!==null){
152           for(const item in objs.items){
153               CreateListItem(objs.items[item]);
154           }
155       }
156
157       CreateAddBtn();
158   }
```

Kuva 16. Sivulle sisältöä kokoava Load-funktio.

Load-funktiota kutsutaan sivun latauduttua käyttäen DOMContentLoaded-tapahtumankuuntelijaa (Kuva 17), jota kutsutaan, kun kaikki sovelluksen staattinen sisältö on latautunut.

```
250
251   document.addEventListener("DOMContentLoaded", () => { Load(); });
252
```

Kuva 17. Load-funktiota kutsuva tapahtumankuuntelija.

GetAllObjs-funktiossa (Kuva 18) fetch-metodille annetaan reitittimen osoite ja olio, joka sisältää reitin tyypin ”get” sekä vastaanotettavan tiedostotyypin JSON. Funktion lopussa palautetaan palvelimelta saatu tiedosto.

```
5   async function GetAllObjs(){
6       const objs = await fetch(url,{
7           method:"GET",
8           header: {"Content-Type": "application/json"}})
9           .then(res => res.json())
10          .catch(error => console.log(error));
11      return objs;
12  }
```

Kuva 18. Fetch apia hyödyntävä GetAllObjs-funktio.

ClearLeftCont-funktio (Kuva 19) käy läpi kaikki vasemman ruudun lapsielementit silmukassa ja poistaa ne. Listan poisto ja uudelleen luonti on helppo tapa pitää elementtien järjestys sekä estää mahdolliset päällekkäisyydet.

```
133   function ClearLeftCont(){
134       while (leftcontainer.firstChild){
135           leftcontainer.removeChild(leftcontainer.firstChild);
136       }
137   }
```

Kuva 19. HTML-elementtejä poistava ClearLeftCont-funktio.

CreateListItem-funktio (Kuva 20) suoritetaan yksi kerrallaan jokaiselle palvelimelta saadulle oliolle. Funktio luo pääelementin lisäksi paragrafi elementin otsikolle, napin sekä itsensä poistamiselle, että laajentamiselle oikeaan ruutuun. Elementeille annetaan aikaisemmin sijoitellessa ja tyylejä tehdessä luodut luokat, paragrafille oliolta saatu otsikko, poistamisnapille lisätään onclick-tapahtuma, joka kutsuu DeleteItem-funktiota oliolta saadulla id:llä ja laajennusnapille onclick-tapahtuma, joka kutsuu ExpandItem-funktiota myös oliolta saadulla id:llä. Paragrafi ja napit annetaan pääelementille lapsielementeiksi ja pääelementti vasemman ruudun lapsielementiksi.

```
 99   function CreateListItem(obj) {
100       const listItem = document.createElement("div"),
101       listItemHeader = document.createElement("p"),
102       delListItemBtn = document.createElement("Button"),
103       expandBtn = document.createElement("Button");
104
105       listItem.classList.add("item", "listItem");
106       listItemHeader.classList.add("itemHeader");
107       delListItemBtn.classList.add("delItemBtn", "fa", "fa-trash-o");
108       expandBtn.classList.add("expandBtn");
109
110       listItemHeader.innerHTML = obj.title;
111       delListItemBtn.setAttribute("onclick", "DeleteItem('"+obj._id+"');");
112       expandBtn.setAttribute("onclick", "ExpandItem('"+obj._id+"');");
113
114       listItem.appendChild(listItemHeader);
115       listItem.appendChild(delListItemBtn);
116       listItem.appendChild(expandBtn);
117
118       leftcontainer.appendChild(listItem);
119   }
```

Kuva 20. Vasempaan ruutuun sisältöä luova CreateListItem-funktio.

Load-funktion lopussa kutsuttava CreateAddBtn-funktio (Kuva 21) luo pääelementin ja nappielementin. Elementeille annetaan luokat tyylejä varten, napille annetaan luokat ”fa” ja ”fa-plus” joka antaa elementille ikonin. Napille lisätään onclick-tapahtuma, joka kutsuu AddObj-funktiota (Kuva 14). Nappi lisätään pääelementin lapsielementiksi ja pääelementti vasemman ruudun lapsielementiksi tehden siitä sen pohjimmaisen elementin.

```
121  function CreateAddBtn(){
122      const addElement = document.createElement("div"),
123      addBtn = document.createElement("Button");
124
125      addElement.classList.add("contAdd");
126      addBtn.classList.add("fa", "fa-plus", "contAddBtn", "listItem");
127      addBtn.setAttribute("onclick", "AddObj()");
128
129      addElement.appendChild(addBtn);
130      leftcontainer.appendChild(addElement);
131  }
```

Kuva 21. CreateAddBtn-funktio, joka luo napin vasemman ruudun pohjalle.

Olion sisällön laajentaminen oikeanpuoliseen ruutuun tapahtuu ExpandItem-funktiolla (Kuva 22), jota kutsutaan aikaisemmin luoduilla napeilla. Funktio saa napilta id:n, jota käytetään hakiessa kutsuvaa elementtiä vastaava olio palvelimelta käyttäen GetOne-funktiota. Olion saatua funktio tyhjentää oikeanpuolisen ruudun ClearRightCont-funktiolla. ClearRightCont-funktio on muuten identtinen ClearLeftCont-funktion kanssa, mutta se tarkistaa ja tyhjentää oikeaa ruutua vasemman sijaan. Sisältö oikeanpuolen ruutuun rakennetaan CreateExpadableObj-funktiolla ja sille GetOne-funktion hakemalla oliolla. ResizeTextarea-funktio muuttaa tarvittaessa tekstikentän kokoa.

```
228  async function ExpandItem(id){
229      const obj = await GetOne(id);
230      ClearRightCont();
231      CreateExpandableObj(obj);
232      ResizeTextarea();
233  }
```

Kuva 22. ExpandItem-funktio, joka luo sisältöä oikeaan ruutuun muiden funktioiden avulla.

GetOne-funktio toimii muuten samalla tavalla, kun GetAllObjs-funktio (Kuva 18), mutta url-osoitteen loppuun lisätään parametrinä saatu id, näin se käyttää automaattisesti palvelimen reititintä, joka hakee yhden olion sen id:n mukaan.

CreateExpandableObj-funktio luo oikeanpuoleiseen ruutuun elementtejä CreateListItem-funktion (Kuva 20) ja CreateAddBtn-funktion (Kuva 21) tavoin. Se luo elementit otsikolle, kuvalle, kuvan lähettävälle form:lle sekä yleiselle tekstikentälle.

5.3.4 Olioiden tietojen päivittäminen ja poisto

Sekä otsikko, että yleinen tekstikenttä käyttää samaa fetch apia hyödyntävää UpdateText-funktiota tekstin lähettämiseksi palvelimelle. Funktio tarvitsee siten parametreiksi tyypin, id:n ja itse teksti sisällön. Palvelimella on

reitittimet erikseen kummallekin tyypille, joten tyyppi lähetetään palvelimelle url-osoite parametrinä id:n kanssa. Funktion lopussa kutsuttava Load-funktio päivittää vasemman ruudun listItem-funktiossa olevan otsikon.

UpdateText-funktiota kutsutaan SetText-funktiosta (Kuva 23). Sitä kutsutaan puolen sekunnin viiveellä käyttäen JavaScriptin setTimeout-metodia. Jotta tekstiä ei tallennettaisi jokaisen muutoksen tapahtuessa, käytetään keypress-tapahtumankuuntelijaa (Kuva 23), joka kutsuu setTimeout-metodille clearTimeout-metodia, joka nollaa setTimeout-metodissa kuluneen ajan. Näin sovelluksen käyttäjän täytyy olla puoli sekuntia kirjoittamatta ennen, kun teksti lähetetään palvelimelle.

```
242  let saveTextTimeout;
243  function SetText(obj){
244      saveTextTimeout = setTimeout(() => {UpdateText(obj);}, 500);
245  }
246
247  window.addEventListener('keypress', () => {
248      window.clearTimeout(saveTextTimeout);
249  }, true);
```

Kuva 23. Tekstiä vastaanottava SetText-funktio ja sen toiminnan pysäyttävä tapahtumankuuntelija.

Olioiden poistaminen tietokannasta tapahtuu vasemman ruudun listItem-elementeissä olevista roskakori ikoneista. Nappi kutsuu DeleteItem-funktiota, jossa fetch api lähettää reitittimelle delete-pyynnön id:n kanssa. Poistamisen jälkeen funktio tarkistaa, että onko poistettava olio tällä hetkellä aukaistuna oikeassa ruudussa ja tyhjentää oikean ruudun ClearRightCont-funktiolla, jos on. Funktio kutsuu Load-funktiota ladatakseen vasemmanpuolen ruudun listan uudelleen.

5.3.5 Kuvien lähetys ja käsittely palvelimella

Kuvan lähettäminen palvelimelle tapahtuu HTML-form:lla ja siitä kutsuttavalla UploadImage-funktiolla. CreateExpandableObj-funktiossa luotu form-elementti sisältää input-kentän, johon voi valita tiedoston sekä submit-napin, johon on lisätty tapahtumankuuntelija, joka kutsuu UploadImage-funktiota (kuva 24). Form-elementin vakiotoiminto on lähettää saadut tiedot johonkin submit-napilla. Tätä ei kuitenkaan haluta, kun tieto halutaan lähettää käyttäen fetch apia. Toiminto estetään tapahtuman metodilla Event.prefentDefault.

UploadImage-funktio hakee HTML-dokumentista sekä olion id:n johon kuva halutaan liittää että itse kuvatiedoston ja luo FormData-olion. Kuva lisätään FormData-olioon ja se lähetetään reitittimelle käyttäen fetch apia. Kuvan lähetettyä kutsutaan RefreshImgSpace-funktiota, joka hakee uuden

kuvan tiedot palvelimelta pienellä viiveellä, koska kuvan käsittely palvelimella vie hetken.

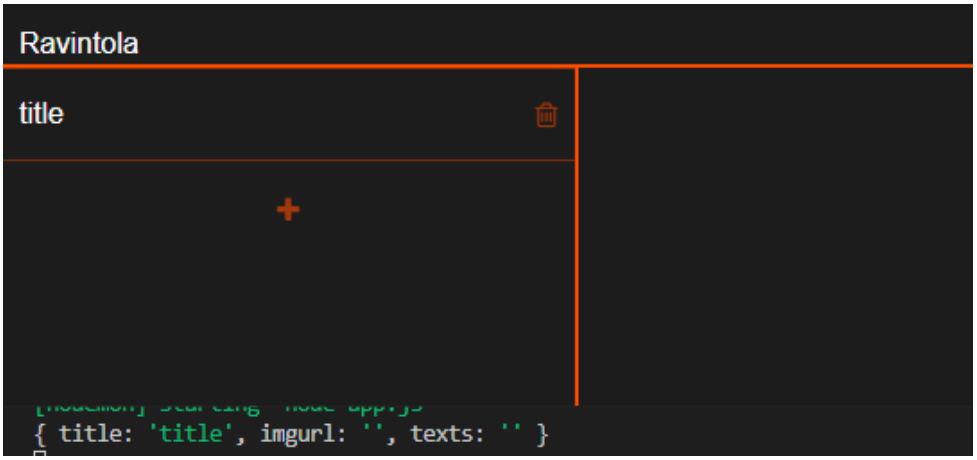
```
async function UploadImage(ev){
    ev.preventDefault();

    const inpFile = document.getElementById("selectImg");
    const fd = new FormData();
    const id = document.getElementById("idelem").value;
    if(inpFile.files[0] === undefined){
        alert("choose an image to submit");
    }else{
        fd.append("img", inpFile.files[0]);

        await fetch(url + "upload/" + id, {
            method: "POST",
            body: fd
        })
        .then((res) => res.json())
        .then(setTimeout(() => {RefreshImgSpace(id);}, 2000))
        .catch((err) => console.log(err));
    }
}
```

Kuva 24. UploadImage-funktio, joka lähettää kuvan palvelimelle.

6 WEB-SOVELLUKSEN TESTAUS

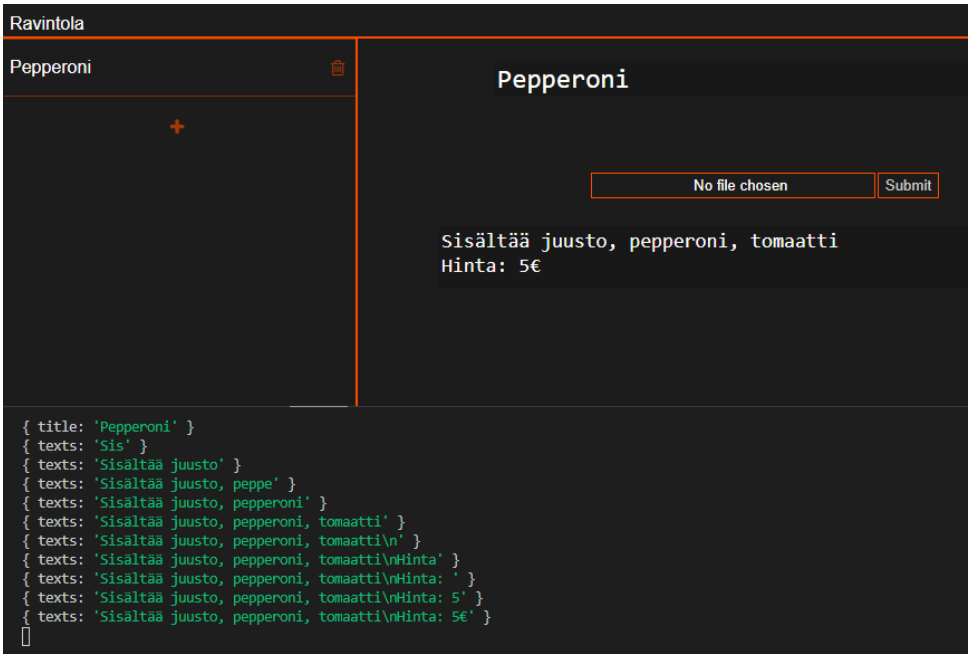
Testiskenaariossa luodaan tuote web-sovellukseen käyttäen sen ominaisuuksia sekä seurataan tietoliikennettä palvelimella.

Aloitetaan luomalla listaan uusi tuote plusnapilla. Palvelin tulostaa tuotteen tiedot terminaaliin (Kuva 25).



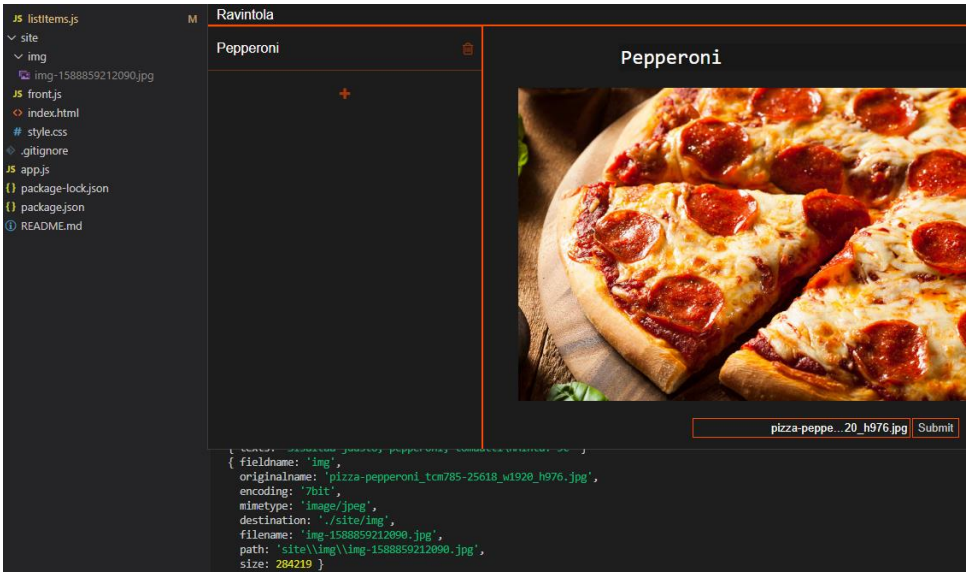
Kuva 25. Sovelluksessa luotu tuote ja sen olio palvelimella.

Avataan tuote oikeaan ruutuun painamalla sitä ja lisätään tekstiä kenttiin. Funktion, joka lähettää tekstin palvelimelle viiveellä tietoliikenteen vähentämiseksi voi nähdä toiminnassa palvelimen saamista olioista (Kuva 26). Teksti ei näin päivity kirjain kerrallaan.



Kuva 26. Tekstin lisäys kenttiin.

Valitaan tuotetta vastaava kuva paikalliselta kovalevyltä ja painetaan Submit-nappia. Kuva ilmestyy uudelleennimettynä projektin tiedostoihin, sekä sen tietoihin päästään käsiksi palvelimella. Kuvan käsiteltyä palvelin lähettää sen selaimelle (Kuva 27).



Kuva 27. Kuvan lähetys palvelimelle.

7 YHTEENVETO

Tässä opinnäytetyössä tutustuttiin paikallisen yksisivuisen web-sovellus kokonaisuuden ohjelmointiin. Sovelluksen selainpuoli koodattiin ilman erillisiä JavaScript-kirjastoja käyttäen vain perinteisiä selainpuolen koodaus kieliä. Palvelinta tehdessä tutustuttiin Node.js-ympäristöön, sen tarjoamiin väliohjelmistoihin ja niillä tehtäviin ratkaisuihin. Tietokanta toteutettiin paikallisesti MongoDB:n community palvelimella. Aiheen ja teknologioiden valintaan vaikutti oma kiinnostus moderniin web-kehitykseen ja käytettyjen teknologioiden suosio alalla. Yksinkertaisen web-sovelluksen ohjelmointi onnistui mielistäni hyvin; sovellusta testatessa se toimi virheettömästi ja nopeasti.

Opinnäytetyön päätavoitteet olivat selainpuolen koodaus sovellusmaiseksi ja Node.js-palvelin. Kummastakaan ei ollut aiempaa kokemusta, mutta aiemmat web-kehitys projektit helpottivat oppimista.

Opinnäytetyön eniten hankaluuksia aiheuttanut osa oli selainpuolen ohjelmointi. Se ohjelmoitiin täysin aikaisemman ohjelmointikokemuksen voimin käyttämättä edes suuntaa antavia ohjeita ilman aikaisempaa kokemusta yksisivuisista web-sovelluksista. Ohjelmiston suunnitteluun meni paljon aikaa ja koko selainpuolen ohjelmointi aloitettiin tyhjästä kolmesti.

Opinnäytetyötä tehdessä ei tullut pelkästään opittua uutta, vaan myös kehitettyä suunnittelutaitoja projektin kolmen toisistaan riippuvan osan ansiosta.

LÄHTEET

Adeyefa, O. (2019) *JavaScript Everywhere – Web, Mobile and desktop.* Haettu 4.3.2020
https://medium.com/@sainttobs/javascript-everywhere-web-mobile-and-desktop-68131878d22d

MDN web docs (2020a) *Using Fetch*. Haettu 5.3.2020
https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

MDN web docs (2020b) *Async function*. Haettu 5.3.2020
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

MDN web docs (2020c) *Promise*. Haettu 9.3.2020
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Pablo Regen (2019) *Node.js: what it is, when and how to use it, and why you should.* Haettu 5.3.2020
https://www.freecodecamp.org/news/node-js-what-when-where-why-how-ab8424886e2/

Express Guide (2017) *Routing*. Haettu 9.3.2020
https://expressjs.com/en/guide/routing.html

Neoteric (2016) *Single-page application vs. multiple-page application.* Haettu 9.3.2020
https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58

MDN web docs (2020d) *Express Tutorial part 3: Using a Database (with Mongoose).*
Haettu 9.3.2020
https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/mongoose

Expressjs/multer (n.d.) multer readme.
Haettu 9.3.2020
https://github.com/expressjs/multer

Nodejs.dev (n.d.) Introduction to Node.js
Haettu 9.3.2020
https://nodejs.dev/