Haaga-Helia
ammattikorkeakoulu Oy

DinnerChatter-ruokailusuunnittelusovelluksen Node.js-palvelimen kehitys

Topi Paretskoi

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2017

Tiivistelmä

Haaga-Helia ammattikorkeakoulu Oy

Tekijä(t)
Topi Paretskoi

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko
Ruokailusuunnittelusovelluksen Node.js-palvelimen kehitys

Sivu- ja liitesivumäärä
39 + 1

Opinnäytetyöprojektin tavoitteena oli kehittää Node.js-pohjainen palvelin Android-sovellukselle. Kehitysprojekti toteutettiin parityönä Olli Nyholmin kanssa, joka vastasi sovelluksen käyttäjäpuolesta. Sovellusta on tarkoitus käyttää ryhmien ruokailun suunnitteluun julkisissa ruokailupaikoissa, sallien käyttäjien luoda sovelluksessa ryhmiä ja niiden sisäisiä listoja ruokailupaikoista. Käyttäjät voivat sitten keskustella ruokailupaikoista ja tarvittaessa äänestää valinnan tekemiseksi. Ruokailupaikat saadaan näkyviin kartalle.

Opinnäytetyön Tietoperusta-luku käsittelee palvelinsovelluksessa käytettyjä teknologioita ja niiden käytön perusteita yleisesti puuttumatta suoraan niitten käyttöön sovelluksessa. Kehityksessä käytettiin Node.jsää sekä useita sen valmismoduuleja, kuten Express.jsää. Palvelimen tietokanta toteutettiin MongoDB:llä, joka yhdistetään sovellukseen Mongoose-moduulilla.

Sovellus-luvussa palvelinsovellus käsitellään aihekohtaisesti, alkaen sovelluksen teknologiavalintojen perustelusta ja palvelinosuuden tietotarpeiden suunnittelusta. Loppuluku käy läpi sovelluksen eri ominaisuuksien toteutusta.

Lopputulos ja retrospektiivi-luku käsittelee kehitysprojektin lopputulosta ja itse projektista tulleita mietteitä. Kappaleessa käydään läpi kehityksessä käytettyjen teknologioiden sopivuus lopputuloksen näkökulmasta, sekä sovelluksen nykytilan ongelmat ja jatkokehitysmahdollisuudet. Kappaleessa käsitellään myös lyhyesti sovelluksen käyttöönsaatto DigitalOceanin pilvipalvelimella oman SSL-sertifikaatilla suojatun verkkotunnuksensa alla. Lopuksi luvussa kerrataan kehittäjän oppimiskokemuksia projektista.

Asiasanat
Ohjelmistokehitys, Node.js, palvelimet, MongoDB

Sisällys

Käsitteet

HTTP on World Wide Webin sovellustason protokolla, joka määrittää miten data kommunikoidaan laitteiden välillä verkossa. Sen toiminta perustuu palvelimelle lähetettyyn pyyntöön ja palvelimen takaisin lähettämään vastaukseen.
(Pavan, P. 2013.)

Javascript on ohjelmointikieli, jota käytetään yleisesti HTML-verkkosivujen toiminnollisuuksien ja dynaamisuuden toteuttamiseen. (W3Schools 2017.)

C++ on keskitason olio-ohjelmointikieli. (TutorialsPoint 2017a.)

NoSQL-tietokanta on termi, jota käytetään yleisesti useista uusista tietokantateknologioista, jotka eivät vastaa toiminnaltaan vanhempaa SQL-tietokantamallia. Ne ovat tyypillisesti malliltaan taipuisampia kuin SQL-tietokannat. (Basho 2017.)

JSON, Javascript Object Notation, on kevyt datan siirtoon verkossa käytetty standardi. (JSON 2017.)

Nginx on ilmainen HTTP-palvelinohjelmisto ja käänteisvälityspalvelin. (Nginx 2017.)

PM2, Process Manager 2, on prosessimanageri Node.js-sovellusten hallintaan tuotantokäytössä. (Keymetrics 2017.)

Prosessorisäie (engl. processor thread) on prosessorin joltakin sovellukselta saama ohjesarja, jonka prosessori käsittelee. (Jon Stokes 2017.)

Monisäikeisyys (engl. multithreading) on prosessointimalli, jossa prosessori käsittelee useita säikeitä yhtäaikaisesti joko vuorottelemalla niitten välillä tai käsittelemällä niitä useammalla prosessoriytimellä. Sovelluksen näkökulmasta monisäikeisyys tarkoittaa, että sovellus luo tarvittaessa toiminnalleen useamman säikeen. (Jon Stokes 2017.)

1 Johdanto

Tämä opinnäytetyö käsittelee DinnerChatter-sovelluksen palvelinpuolen kehitystä. Projektin käyttäjäpuolesta vastasi Olli Nyholm, joka kirjoitti aiheesta erillisen opinnäytetyön. Kehitysprojektin tarkoituksena oli tuottaa ja julkaista Google Play Storeen ruokailusuunnittelumobiilisovellus hyödyntäen moderneja teknologioita. Sovellus on niin käyttäjä- kuin palvelinpuolellakin luotu kokonaan Javascriptillä.

Projektissa syntyneessä sovelluksessa käyttäjät luovat sähköpostikohtaisen tilin ja voivat luoda sekä liittyä ryhmiin. Ryhmien sisällä käyttäjät pystyvät luomaan listoja ruokailupaikoista jotka saadaan näkyviin Google Maps- kartalle. Listattuihin ruokailupaikkoihin voi lisätä kommentteja ja käyttäjät voivat keskustella niistä ryhmäkohtaisilla keskustelualueilla.

Pyrin tässä opinnäytetyössä aluksi kuvailemaan kehityksessä käytettyjä teknologioita. Tämän jälkeen käyn läpi niiden käyttöä sovelluksessa sekä ongelmakohtia niin, että mobiili-ja/tai verkkokehitysestä kiinnostuneet voivat sen avulla saada käsityksen tämän kaltaisen projektin toteuttamisesta. Lopuksi opinnäytetyössä käydään läpi lopputulosta yleisesti, käytettyjen teknologioiden soveltuvuutta ja sovelluksen käyttöönottoa.

Sovelluksen palvelinpuoli rakennettiin Javascriptiin perustuvaan Node.js-ympäristöön käyttäen NoSQL-tyyppistä MongoDB-tietokantaa. Ohjelmiston toimintojen rakentamisessa hyödynnettiin lukuisia Node.js-moduuleita. Sovelluskehityksessä käytettiin Github-kehitysjärjestelmää.

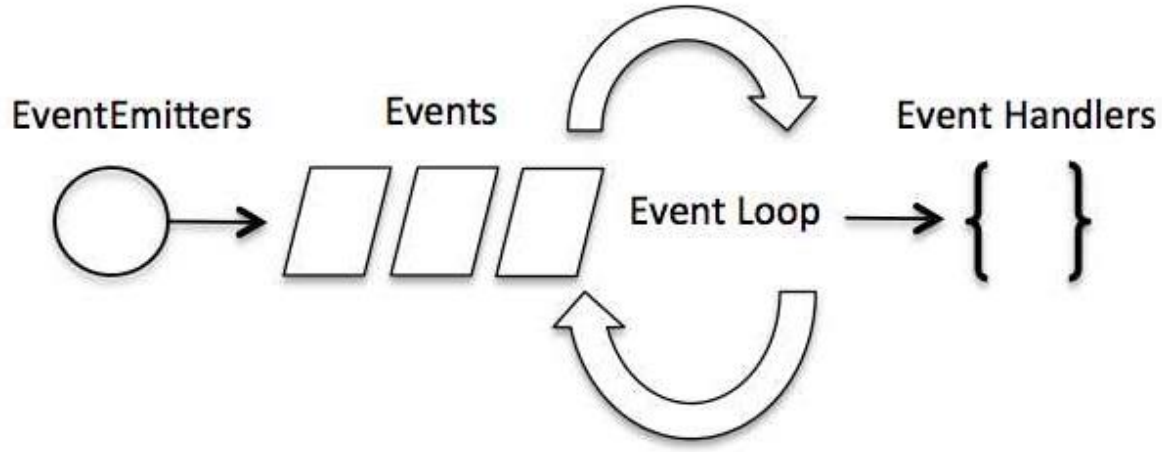
2 Tietoperusta

Tässä kappaleessa esitellään sovelluksen kehityksessä käytetyt teknologiat sekä niiden käytön perusteet.

2.1 Node.js

Node.js on Googlen V8 Javascript Enginen päälle rakennettu Javascript-ajoympäristö, joka on suunniteltu käyttöön palvelimella Javascriptille tyypillisen käyttäjäpuolen sijaan. Sen tavoitteena on tuottaa tehokkaasti skaalautuvia verkko-ohjelmistoja. Rakenteeltaan se on tapahtumavetoinen, asynkroninen ja ydintoiminnaltaan yksisäikeinen. (Node.js 2017a.)

Käytännössä Node-sovellus tuottaa esimerkiksi palvelimelle saapuvista käyttäjäpyynnöistä tapahtumia (events), jotka lisätään sovelluksen tapahtumajonoon. Tapahtumakierre (event loop) joka poimii tapahtumat jonosta ja tapahtumakäsittelijät (event handlers) joihin tapahtuma siirretään toimivat kaikki yhdessä prosessorisäikeessä, joten tapahtumia käsitellään yksi kerrallaan vuorotellen. Kuva 1 havainnollistaa tätä käsittelymallia. (Teixeira 2012)



Kuva 1. Node.js tapahtumakierre (TutorialsPoint 2017b.)

Noden tapahtumakäsittely on asynkronista, eli jos tapahtuman käsittelyssä on toiminto jota prosessorin täytyy odottaa, useimmiten siis levyn kirjoitus- tai lukutoiminto tai datan siirto verkossa, tapahtumakierre jättää käsittelyn ja siirtyy suorittamaan seuraavaa tapahtumaa. Käsittelyn keskeyttäneen toiminnon valmistuttua se luo valmistumistapahtuman joka lisää sen takaisin tapahtumalistalle, josta ohjelman tapahtumakierre sitten aikanaan poimii sen. (Future Processing 22.4.2015.)

Asynkronisen toiminnan mahdollistaa Libuv-niminen C++-kirjasto, joka ylläpitää erillistä säiekokoelmaa jolle tapahtumakierteen hylkäämät käsittelyt annetaan. Libuv-säikeille annettu käsittelypyyntö koostuu säikeessä suoritettavasta funktiosta, funktion datasta sekä funktiosta joka kokoaa Libuv-säikeen tulokset, jotka sitten palautetaan pääohjelmalle. Säiekokoelmassa on oletusasetuksilla neljä säiettä. Jos säikeet ovat kaikki käytössä, käsittelypyyntö lisätään säiekokoelman jonoon. (Future Processing 22.4.2015.)

Verkkopalvelimille tyypilliset funktiot ovat prosessorin käytöltään varsin kevyitä, käyttäen yleensä valtaosan elinkaarestaan joko levy- tai verkkoprosesseihin. Monisäikeisessä mallissa funktioille luotavat säikeet siis sitovat järjestelmän resursseja suurimman osan ajasta turhaan. Noden yksisäikeinen malli suunniteltiin verkkopalvelimien skaalautumisen helpottamiseksi ja yksinkertaistaa lisäksi ohjelmointia koska kehittäjän ei tarvitse puuttua säikeisiin ja niiden synkronointiin. Kääntöpuolena on huomioitava, että Node.js sopii huonosti sovelluksiin, jotka suorittavat prosessorille raskaita laskutoimintoja, koska ne sitovat sovelluksen ainoan Javascriptiä suorittavan säikeen pitkäksi aikaa, jolloin muu sovellus pysähtyy. (Vagg 24.6.2014.)

2.1.1 Node.jsän käyttö

Node.js on saatavilla asennusohjelmana Windows- ja OS X-käyttöjärjestelmille sekä binääreinä niille ja Linuxille. Kun Node.js on onnistuneesti asennettu, sitä käytetään komentoriviltä node-komennolla, joka ilman parametreja kutsuttaessa avaa Javascriptiä suorittavan Node.js-komentopäätteen. Ohjelmat käynnistetään kutsumalla niitä node-komennon parametrina.

Kuvassa 2 esitellään yksinkertainen Node.js-sovellus. Esimerkkisovelluksessa tiedostoon tuodaan Noden http-palvelinmoduuli ja pystytetään sen createServer-metodilla palvelin, jonka parametreina ovat pyynnön vastaanottava req-objekti sekä vastauksen lähettävä res-objekti. Palvelin käynnistetään kutsumalla palvelininstanssin listen-metodia jossa annetaan palvelimen portti. HTTP-palvelin pitää Node-sovelluksen päällä kunnes se sammutetaan. (Node.js 2017b.)

Kuvan 2 Res-objektin writeHead-metodi antaa vastausviestille headerin, tässä tapauksessa siis statuskoodin 200, joka ilmoittaa toiminnon onnistuneen. Vastausviesti päätetään end-metodilla, jota on kutsuttava joka vastauksen yhteydessä tai vastausta ei lähetetä ja pyynnön käsittely jumittuu. End-metodille voidaan antaa mukaan viesti joka kulkee vastauksen body-osuudessa – viesti voidaan lähettää myös erillisellä write-metodilla. Sovel-

luksen ollessa päälle selainnavigointi osoitteeseen http://localhost:8080/ tuottaa vastauksena end-metodissa lähetetyn tekstipätkän. (Node.js 2017b.)

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

var helloServer = http.createServer(function(req, res){
    res.writeHead(200);
    res.end("Hello world!");
});

helloServer.listen(8080);
```

Kuva 2. Yksinkertainen Node.js-palvelinsovellus

2.1.2 Moduulit ja NPM

Moduuli on yksikkö koodia, joka on koostettu jollain tavalla toisiinsa liittyvästä kokoelmasta koodia – tyypillisesti se toteuttaa jonkin toiminnollisuuden. Moduuliin viitataan koodissa yhtenä yksikkönä, jonka metodeja sen sisältämät funktiot ovat. Niiden tarkoitus on helpottaa koodin organisoimista ja sovelluksen muokkaamista jakamalla se toisistaan irrallisiin osiin jotka ohjelmisto sitten yhdistää. (Kim 2017.)

Nodessa jokainen tiedosto on moduuli ja moduuleilla on yksi-yhteen-yhteys, eli tarvittavat linkitykset tehdään tiedostokohtaisesti. Linkitykset tehdään kuvassa 2 nähdyllä require-komennolla, jolle annetaan polku kohdetiedostoon. Noden sisäänrakennetut moduulit, kuten http, tai siihen ulkopuolelta ladatut, node_modules-kansioon tallennetun moduulit kutsutaan ilman polkua. Kun moduuli on lisätty, sen ulospäin tarjoamat metodit ja muuttujat ovat kutsuttavissa. Moduuli luodaan lisäämällä tiedoston osia sen exports-objektiin. Objektin sisältämät osat ovat sitten saatavissa kaikissa tiedostoissa, joihin moduuli lisätään. (NodeJS 2017c.)

Funktioita voidaan lisätä exports-objektiin kahdella eri tavalla: joko yksitellen, kuten kuvassa 3 tai joukossa lisäämällä ne suoraan module.exports-objektin sisälle kuten kuvassa 4. Molemmissa tapauksissa moduulin metodit ovat saatavissa kutsumalla moduuli instanssina toiseen tiedostoon Noden require-komennolla ja kutsumalla niitä sen alta, kuten kuvassa 5, jossa kuvattu sovellus ajettaessa kutsuu returnAnimal-moduulin molempia metodeja ja kirjoittaa saadun tuloksen konsoliin. (NodeJS 2017c.)

```
//returnAnimal.js

exports.returnDog = function(){
    return "dog";
}

exports.returnCat = function(){
    return "cat";
}
```

Kuva 3. Metodien lisäys yksittäin

```
//returnAnimal.js
module.exports = {
    returnDog: function(){
        return "dog";
    },

    returnCat: function(){
        return "cat";
    }
};
```

Kuva 4. Metodien lisäys suoraan export-objektiin

```
var returnAnimal = require("./returnAnimal.js");

var animal = returnAnimal.returnCat();
console.log(animal);

var animal = returnAnimal.returnDog();
console.log(animal);
```

Kuva 5. Moduulin metodien käyttö

Nodella on laaja kokoelma käyttäjien luomia ja julkaisemia moduuleja, jotka ovat saatavissa NPM:n, Node Package Managerin, kautta. NPM on saatavilla osana Node.js-asennusohjelmia Windows- ja OS X- käyttöjärjestelmillä ja saatavilla erillisenä asennuksena niille sekä Linux-käyttöjärjestelmille. (NPM 2017a.)

NPM:ä käytetään komentoriviltä. Ohjelmistopaketit voidaan asentaa joko paikallisesti sovelluksen käyttöön, jolloin ne asentuvat kansiossa node_modules-alikansioon, tai globaalisti, jolloin ne ovat aina saatavilla. Globaali asennus on tarkoitettu lähinnä esimerkiksi komentolinjatyökaluina toimiville moduuleille. (NPM 2017b.)

Tyypillisesti sovelluksille määritetään package.json-tiedosto, johon listataan joitakin perustietoja sovelluksesta ja lista sen vaatimista ulkoisista moduuleista. NPM voi package.json-tiedoston perusteella ladata tarvittavat paketit, helpottaen sovelluksen käyttöönottoa. (NPM 2017c.)

2.2 Express.js

Express.js on kevytrakenteinen Node.js ohjelmistokehys joka on tarkoitettu helpottamaan verkkosovellusten ja ohjelmistorajapintojen kehittämistä. Express itsessään tarjoaa vain välttämättömimmät osat, mutta sen päälle on rakennettu useita moduuleja jotka laajentavat sen toiminnollisuutta. (Express.js 2017a.)

Sovelluksessa Express.jsää edustaa Application-objekti, yleensä nimeltään app, joka saadaan kutsumalla ylimmän tasoista express-funktiota. App-objekti on Javascript-funktio, joka kiinnitetään Node.js-palvelinobjektiin. Helpoimmin se on tehtävissä kuvassa 6 näytetyllä tavalla: listen-metodi kiinnittää app-objektin automaattisesti Noden HTTP-palvelinobjektiin ja avaa palvelimen parametrina annetussa portissa. Jos käyttöön halutaan HTTPS-portti tai molemmat täytyy app-objekti kiinnittää erikseen kutsuttaviin Noden HTTP- ja HTTPS-palvelinobjekteihin kuten kuvassa 7. Kuvassa HTTPS-palvelimelle annetaan lisäksi options-parametri, joka sisältää palvelimen SSL-asetukset. (Express.js 2017b.)

```
var express = require('express');
var app = express();
app.listen(3000);
```

Kuva 6. Express.js palvelimen käynnistys (Express.js 2017b.)

```
var express = require('express');
var https = require('https');
var http = require('http');
var app = express();

http.createServer(app).listen(80);
https.createServer(options, app).listen(443);
```

Kuva 7. Palvelimen käynnistys erillisillä HTTP- ja HTTPS-porteilla (Express.js 2017b.)

2.2.1 Reititys

Reitityksellä tarkoitetaan määrityksiä joiden mukaan Express-sovellus reagoi ohjelmapolkuihin saapuviin HTTP-pyyntöihin metodikohtaisesti. Samalle polulle, vaikkapa *www.exampleapp.com/hello,* voidaan esimerkiksi antaa erilliset käsittelijät GET- ja POST-metodeille. Virtaviivaistettu reititys on Express.jsän ydinominaisuus. (Express.js 2017c)

ExpressJS-reititys perustuu app-objektin reittimetodeihin, jotka toteuttavat normaaleja HTTP-metodeja. Niille annetaan parametreinä ohjelmapolku, johon saapuviin pyyntöihin reitti reagoi, sekä funktio joka sisältää Expressin Request- ja Response-objektit, joihin viitataan tyypillisesti nimillä req ja res. Req-objekti vastaanottaa polulle tulleen HTTP-pyynnön ja sen datan. Vastaus pyyntöön lähetetään kutsumalla jotain res-objektin metodeista. (Express.js 2017c)

Expressille on mahdollista luoda erillisiä reitittimiä kutsumalla express.Router-luokkaa, jotka sitten kiinnitetään sovelluksen app-objektiin. Ne käsittelevät omia reittejään oman juuripolkunsa alla ja niille voidaan määrittää muusta sovelluksesta erilliset väliohjelmat. (Express.js 2017c)

Esimerkki reitityksestä on nähtävissä kuvassa 8, jossa sovelluksen app-objektille on määritetty POST-metodireitti polulla /hello. Reitti tarkastaa onko palvelupyynnön datalla populoidun req-objektin body-osuudessa dataa name-avaimella. Jos dataa löytyy, lähetetään name-avaimen datalla personalisoitu vastaus kutsumalla res-objektin send-metodia. Jos dataa ei löydy lähetetään staattinen tervehdys. Send-metodilla voidaan String-datan lisäksi lähettää objekti, Array tai Buffer-objekti. (Express.js 2017b)

```
var bodyParser = require("body-parser");
app.use(bodyParser.json());

app.post('/hello', function(req, res){
    if(req.body.name){
        res.send("Hello, " + req.body.name + "!");
    }else{
        res.send("Hello, stranger!");
    }
});
```

Kuva 8 Reititys, pyyntö ja vastaus

Palvelupyynnön body-data täytyy parseroida req-objektiin. Kuvassa 8 reitin yläpuolella sovellukseen tuodaan body-parser-moduuli jonka JSON-metodi kiinnitetään app-objektiin.

Näin JSON-datan parserointi on käytettävissä sovelluksen kaikilla reiteillä. (Express.js 2017c)

Response-objektilla on useita metodeita, jotka mahdollistavat hyvin erilaisia toimenpiteitä reiteillä. Vastauksena voidaan lähettää esimerkiksi tiedosto, palvelimella luotava näkymä, siirtokäsky toiseen osoitteeseen tai statuskoodi. (Express.js 2017b)

2.2.2 Middleware

Express.jsän väliohjelmat ovat erillisiä funktiota, jotka ajetaan ennen niille määritettyjä reittejä. Niille annetaan reitin req- ja res-objektit, sekä kolmas parametri jota kutsuttaessa edetään seuraavaan väliohjelmaan: virallinen suositus on kutsua tätä parametria nimellä next. Jos ajettava väliohjelma ei päätä pyynnön käsittelyä res-objektin metodilla ja next()-kutsua ei anneta, käsittely jumittuu. (Express.js 2017d.)

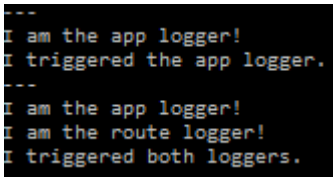
Väliohjelmat voidaan sitoa joko suoraan Application-objektiin, jolloin väliohjelma ajetaan ohjelman jokaisella polulla, tai yksittäisiin Router-objekteihin. Ne voivat suorittaa mitä tahansa funktioita, muokata res- ja req-objekteja, päättää pyyntö-vastausprosessin ja kutsua seuraavaa väliohjelmaa. (Express.js 2017d.)

Väliohjelmia voidaan hyödyntää esimerkiksi palvelintapahtumien kirjaamiseen, req-objektin datan parseroimiseen tai käyttäjien autentikoimiseen. Useat Node.js-moduulit toimivat väliohjelmina.

Kuvassa 9 sovellukseen luodaan kaksi väliohjelmaa, appLogger ja routeLogger, jotka kutsuttaessa kirjoittavat ilmoituksen konsoliin, sekä kaksi reittiä, /triggerOne ja /triggerBoth, jotka kutsuttaessa lähettävät vastauksen ja kirjoittavat oman ilmoituksensa konsoliin. appLogger kiinnitetään sovelluksen app-objektiin use-metodilla, jolloin se ajetaan sovelluksen kaikkia reittejä kutsuttaessa. routeLogger kiinnitetään suoraan /triggerBoth-reittiin, joka näin ollen ajaa kutsuttaessa molemmat väliohjelmat. Kuvassa 10 näytetään konsolinäkymä kun sovelluksessa kutsutaan ensin /triggerOne-reittiä ja sitten /triggerBoth-reittiä. (Express.js 2017d.)

```
var appLogger = function(req, res, next){
    console.log("---");
    console.log("I am the app logger!");
    next();
};

var routeLogger = function(req, res, next){
    console.log("I am the route logger!");
    next();
};

app.use(appLogger);

app.get("/triggerOne", function(req, res){
    res.send("I triggered the app logger.");
    console.log("I triggered the app logger.");
});

app.get("/triggerBoth", routeLogger, function(req, res){
    res.send("I triggered both loggers.");
    console.log("I triggered both loggers.");
});
```

Kuva 9 Kaksi väliohjelmaa



Kuva 10 Konsolinäkymä reittikutsuista

2.3 Body-parser

Body-parser on Node.js väliohjelma HTTP-pyyntöjen body-osion datan jäsentämiseen palvelimella käytettävään muotoon, esimerkiksi Express.jsän Request-objektin body-osion sisälle. Body-parseria voidaan käyttää JSON-, Raw-, Text- ja URL-encoded-tyyppisten body-osioiden käsittelyyn. Se ei pysty jäsentämään multipart-tyyppistä dataa. (Wilson 2017.)

Kuvassa 11 body-parserin JSON-luokka kiinnitetään ylimmän tason väliohjelmana, eli suoraan Express-sovelluksen Application-objektiin. Näin se on käytössä kaikilla sovelluksen reiteillä ilman lisätoimenpiteitä. Se voidaan samalla tavoin kiinnittää Router-objekteihin, jolloin se on käytössä vain kyseisen reitittimen poluilla. Kuvassa 12 samasta luokasta luodaan ensin objekti, joka sitten kiinnitetään yksittäiseen reittiin. (Wilson 2017.)

```
var app = express();
app.use(bodyParser.json());
```

Kuva 11. JSON-parserin kiinnitys Express-ohjelmaan

```
var jsonParser = bodyParser.json();

app.post('/hello2', jsonParser, function(req, res){
  console.log("Name: " + req.body.name);
    if(req.body.name){
        res.send("Hello, " + req.body.name + "!");
    }else{
        res.send("Hello, stranger!");
    }
});
```

Kuva 12. JSON-parserin kiinnitys reittiin.

2.4 Socket.io

Socket.io on Node.js-pohjainen ohjelmistokehys, joka on tarkoitettu tapahtumiin perustuvan, reaaliaikaisen ja kaksisuuntaisen kommunikaation rakentamiseen. Se hyödyntää pääasiassa WebSocket-protokollaa, mutta voi turvautua varasuunnitelmana polling-tekniikkaan. Se on myös saatavilla Java-, C++ ja Swift-kielille. Se tukee kaikkia yleisesti käytössä olevia selaimia. (Socket.io 2017a.)

Socket.io on kaksiosainen: sillä on erilliset ohjelmakirjastot palvelimelle sekä selaimessa käytettävälle käyttäjäpuolelle. Socket.io voidaan pystyttää joko omillaan tai lisätä osaksi johonkin toiseen palvelimeen. Virallinen dokumentaatio tarjoaa tukea Node.js-, Express.js- tai Koa-palvelimille. (Socket.io 2017a.)

Kuvassa 13 Socket.io lisätään osaksi Express.js-palvelinta. Socket.iota edustava io-objekti kiinnitetään sen luomisen yhteydessä Express.jsän app-objektilla jo määritettyyn server-objektiin. Sitten io-objekti säädetään kuuntelemaan connection-tapahtumaa, jonka tullessa lisää uuden käyttäjää edustavan socketin. (Socket.io 2017a.)

```
var app = require('express')();
var server = require('http').createServer(app);
var io = require('socket.io')(server);
io.on('connection', function(){ /* … */ });
server.listen(3000);
```

Kuva 13 Socket.io Express.js-palvelimella. (Socket.io 2017a.)

Kun socket on yhdistetty palvelimelle, ne voivat kommunikoida keskenään tuottamalla ja vastaanottamalla tapahtumia. Tapahtumat tuotetaan emit-metodilla ja ne vastaanotetaan on-metodilla, joille määritetään parametrina tapahtuman nimi sekä sen mahdollisesti kuljettama data. Kuvassa 14 käyttäjäpääte reagoi formin submit-tapahtumaan socket.emit-metodilla, tuottaen tapahtuman nimellä ’chat message’. Tapahtumaan lisätään formin viestinkentästä saatu data. Kuvassa 15 palvelin on säädetty odottamaan tapahtuma nimeltä ’chat message’ ja sellaisen tullessa kirjoittamaan viestin sisältö konsoliin. (Socket.io 2017b.)

```
<script src="/socket.io/socket.io.js"></script>
<script src="https://code.jquery.com/jquery-1.11.1.js"></script>
<script>
  $(function () {
    var socket = io();
    $('form').submit(function(){
      socket.emit('chat message', $('#m').val());
      $('#m').val('');
      return false;
    });
  });
</script>
```

Kuva 14. Socket tapahtuman luonti emit-metodilla. (Socket.io 2017b.)

```
io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    console.log('message: ' + msg);
  });
});
```

Kuva 15. Socket tapahtuman vastaanotto on-metodilla. (Socket.io 2017b.)

2.5 MongoDB

MongoDB on avoimen lähdekoodin NoSQL-dokumenttitietokanta. Data tallennetaan JSON-objekteja muistuttaviin dokumentteihin jotka koostuvat kenttä-arvo-pareista. Kenttiin voidaan säilöä eri datatyyppien lisäksi muita dokumentteja, listoja tai listoja dokumenteis-

ta. Dokumentit järjestetään dokumenttityypin mukaan kokoelmiin. Relaatiotietokantaan verrattuna kokoelmat vastaavat tauluja ja dokumentit rivejä tauluissa. (MongoDB 2017a)

Dokumenttien JSON-mallisuus on käytännöllinen ominaisuus Node.js alustaisella palvelimella, erityisesti jos myös käyttäjäpuoli on kirjoitettu esimerkiksi Angular.js:llä, koska se mahdollistaa helpon JSON-tyyppisen datamallinnuksen ohjelmiston kaikissa osissa.

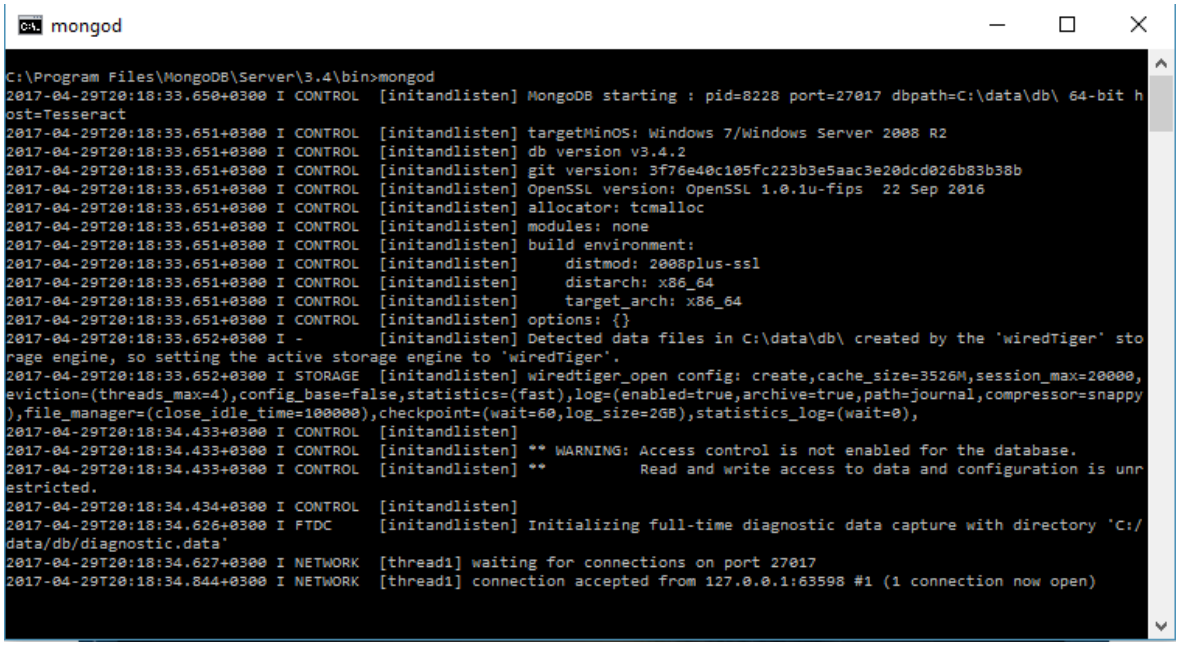
MongoDBeestä on saatavilla ilmaiset ja kaupalliset versiot Windows-, Linux- ja OS-alustoille. Asennukset voi tehdä joko ohjatulla asennusohjelmalla tai erillisten ohjeiden mukaisesti komentoriviltä. Kattavat ohjeet asennukseen ja käyttöönottoon löytyvät MongoDBeen verkkosivuilta.

MongoDBeen oletusasetuksissa data tallennetaan asennuslevyn juuressa olevaan polkuun \data\db – jos tätä oletuspolkua halutaan käyttää, täytyy se käyttöönoton yhteydessä luoda manuaalisesti. Jos polku halutaan vaihtaa, se voidaan tehdä komentorivillä kuvassa 16 näytetyllä tavalla tai vaihtaa ohjelman asetustiedossa. (MongoDB 2017b.)

```
C:\Program Files\MongoDB\Server\3.4\bin>mongod.exe --dbpath "C:\database\path"
```

Kuva 16. MongoDB-tietokannan sijainnin määritys

Kun tallennuspolku on määritetty, MongoDBeen voi käynnistää kutsumalla mongod.exe-ohjelmaa ilman argumentteja, kuten kuvassa 17, jossa tietokanta aukeaa oletusasetusporttiin 27017. (MongoDB 2017b.)



Kuva 17. MongoDBeen käynnistys.

MongoDBeen ollessa päällä siihen voi päästä käsiksi MongoDB Shell-komentopäätteellä, joka avataan komentosyötteessä mongo.exe-ohjelmistolla. Shelliin voidaan sitten syöttää MongoDB-komentoja tietokantojen käsittelemiseksi. (MongoDB 2017c.)

2.6 Mongoose

Mongoose on Node.js-moduuli joka on tarkoitettu MongoDB-tietokannan datamallinnuksen ja tietokantatoimintojen yksinkertaistamiseen. Data mallinnetaan MongoDB-dokumentteja vastaaviin Schema-objekteihin, joihin määritetään dokumentin datakentät. Se tarjoaa myös sisäänrakennetun validoinnin, automaattiset tyyppimuutokset sekä valmiit funktiot tietokantalauseille. (MongooseJS 2017a.)

Schema luodaan Mongoosen Schema-luokan instanssina, kuten kuvassa 15. Schemalle määritetään kenttä-avain-parit antamalla kentälle nimi ja datatyyppi. Kentille voidaan määrittää erilaisia ominaisuuksia, kuten require-sääntö, jonka avulla kentän täyttämisen määritetään pakolliseksi. Kuvan 18 lopussa Schemasta luodaan Car-niminen Model, joka on Schemaa vastaava konstruktori. Konstruktoria luodessa parametreiksi annetaan MongoDB-kokoelman nimi sekä luotu skeema. Kokoelman nimi syötetään yksikkömuodossa ja Mongoose käsittelee sen perusteella monikkomuotoisesti nimettyä kokoelmaa tietokannassa. (Mongoose 2017b.)

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

var carSchema = new mongoose.Schema({
    model:{
        type: String,
        required: true
    },
    manufacturer:{
        type: String,
        required: true
    }
});

var Car = mongoose.model('Car', carSchema);
```

Kuva 18. Mongoose Schema ja Model

Model-konstruktoreita käytetään Document-objektien luomiseen, joiden metodeilla sitten käsitellään vastaavaa kokoelmaa MongoDB-tietokannassa. Kuvassa 19 luodaan newCar-niminen instanssi Car-Modelista, määritetään sen data ja tallennetaan se kutsumalla sa-

ve-metodia. Funktion car-niminen callback palauttaa tietokantaan tallennetun Car-dokumentin ja se kirjataan konsoliin. (Mongoose 2017c)

```
var newCar = new Car({
    model: "Scirocco",
    manufacturer: "Volkswagen"
});

newCar.save(function(err, car){
    if(err){
        //errorhandling
    }else{
        console.log(car);
    }
});
```

Kuva 19. Mongoose-dokumentin luonti ja käyttö

2.7 JSON Web Token

JSON Web Token avoin datasiirtostandardi joka perustuu digitaalisesti allekirjoitettuun JSON objektiin. Käytännössä se on kolmiosainen merkkisarja joka rakennetaan syötetyn datan, koodausalgoritmin ja koodaussalaisuuden perusteella. Sen kolme osaa ovat:

- Header, joka sisältää tiedon tokenin luomiseen käytetystä algoritmista sekä tokenin tyypistä.
- Payload, joka sisältää tokenin varsinaisen datan JSON-muotoisina avain-arvo-pareina.
- Signature, joka on koodaussalaisuuden perusteella tuotettu allekirjoitus, jonka avulla JWT-standardi varmentaa tokenin luotettavuuden.

(JWT 2017.)

JWT:t eivät ole itsessään salattuja ja niiden sisältämä data on helppo saada selkokielisek-si syöttämällä se oikealla algoritmilla dekoodaukseen. JWT ei siis turvaa kuljettamansa datan salassapitoa. JWT:n sisältämä allekirjoitus tuotetaan siihen syötetyn datan, algoritmin ja koodaussalaisuuden, siis tokenin tuottajan tiedossa olevan salaisen merkkisarjan, perusteella. Jos token luodaan tai sen sisältöä muokataan ilman tätä salaisuutta, allekirjoitus ei läpäise tokenin luojan tarkastusta. JWT:llä voidaan siis varmentaa kuljetetun datan luoja sekä datan koskemattomuus. (JWT 2017.)

Datasiirron lisäksi JWT:tä voidaan hyödyntää käyttäjän autentikoinnissa palvelimelle tai linkitetyille palveluille. Käytännössä tämä toteutetaan antamalla käyttäjälle onnistuneen kirjautumisen yhteydessä token joka sisältää tiedot käyttäjälle palvelussa annetuista oi-

keuksista. Tokeniin voidaan usein tallentaa kaikki tarvittava data, joten uusia tietokanta-hakuja käyttäjän varmentamiseksi ei tarvitse tehdä. Tarkastuksen läpäisevä JWT voi itsessään palvella perustason avaimena järjestelmään ja sen kantama oikeusdata tarkastetaan kun käyttäjä yrittää päästä käsiksi oikeudet vaativiin osiin järjestelmässä. (JWT 2017.)

Nodelle on saatavissa Node JSON Web Token-moduuli, joka sisältää tarvittavat metodit Tokenin luomiselle erilaisilla asetuksilla sekä Tokenin tarkastukseen. Kuvan 20 esimerkissä määritetään muuttujat secret ja username, jotka ovat molemmat tekstipätkiä. Token luodaan token-nimisenä muuttujana kutsumalla moduulin sign-metodia, jolle annetaan parametreina ensin kuljetettavaksi dataksi username-muuttujan sisältö ja sitten secret-muuttuja tokenin luontiin käytettävänä salaisuutena. Lopuksi Tokenin eliniäksi määritetään 24 tuntia käyttämällä expiresIn-asetusta. Token voidaan nyt toimittaa HTTP-vastauksessa käyttäjälle. (JWT 2017.)

```
var jwt = require("jsonwebtoken");

var secret = "supersecretstring";
var username = "ExampleUser";

var token = jwt.sign({username: username}, secret, {
    expiresIn: '24h'
});
```

Kuva 20. Tokenin luonti

Token tarkastetaan moduulin verify-metodilla, jolle annetaan parametreina tarkastettava token ja sovelluksessa käytetty salaisuus. Kuvan 21 esimerkissä tarkastus on tehty osana Express.js-väliohjelmaa. Jos tarkastus palauttaa err-vastauksen, eli tarkastuksessa on tapahtunut virhe, sovellus päättää käsittelyn lähettämällä käyttäjälle 401 Unauthorized-vastauksen sekä JSONin joka kertoo Tokenin tarkastuksen epäonnistuneen. Onnistunut tarkastus tuottaa decoded-objektin, joka sisältää dekoodatun, siis selkokielisen, version Tokenista. Se lisätään osaksi HTTP-pyynnön sisältävää req-objektia, jotta se on saatavilla käsittelyn myöhemmissä vaiheissa. Käsittelyn annetaan sitten edetä seuraavaan vaiheeseen next()-komennolla. (JWT 2017.)

```
jwt.verify(token, secret, function(err, decoded){
    if(err){
        console.log("Token authentication failed.");
        return res.status(401).send({
            success: false,
            message: "Token authentication failed."
        });
    }else{
        console.log("Token authenticated.")
        req.decoded = decoded;
        next();
    }
});
```

Kuva 21. Tokenin tarkastus

2.8 Multer

Multer on Node-moduuli joka on tarkoitettu multiform/form-data-tyyppisen datan käsittelyyn: käytännössä sitä käytetään lähinnä tiedostojen siirtoon. Se on rakennettu form-datan siirtoon tarkoitetun Busboy-moduulin päälle. (Multer 2017.)

Multer lisää käsittelemiinsä Request-objekteihin body- ja file-osuudet: body-osuuteen siirretään HTTP-pyynnön HTML-formin mahdollinen data ja file-osuus vastaanottaa HTTP-pyynnössä lähetetyn tiedoston. Multer-instanssille määritetään tallennuspolku sekä mahdolliset rajoitteet käsiteltäville tiedostoille, esimerkiksi hyväksytyt tiedostotyypit tai raja tiedostojen koolle. (Multer 2017.)

Kuvassa 22 luodaan Multer-instanssi nimeltä upload, jolle määritetään ensin tallennuspolku ja sitten fileFilter-asetus, jota vasten kaikki tällä Multer-instanssilla tallennettavat tiedostot tarkistetaan. Lähetetystä tiedostosta otetaan ext-muuttujaan talteen sen tiedostopääte käyttämällä Node.jsän sisäänrakennettua path-moduulia. Ext-muuttujaa verrataan sitten kuvatiedostojen päätteisiin. Jos muuttuja ei vastaa mitään niistä, Multer tuottaa error-viestin, joka ilmoittaa, että kyseinen instanssi hyväksyy vain kuvatiedostoja. Jos tarkastus läpäistään, Multer jatkaa toimintaansa normaalisti. Instanssille on lisäksi määritetty limits-asetuksiin kuvatiedoston maksimikoko. (Multer 2017.)

```
var upload = multer({
    dest:'./public/uploads/',
    fileFilter: function (req, file, cb) {
        var ext = path.extname(file.originalname.toLowerCase());
        if(ext !== '.png' && ext !== '.jpg' && ext !== '.gif' && ext !== '.jpeg') {
            return cb(new Error('Only images are allowed.'), false);
        }
        cb(null, true);
    },
    limits:{
        fileSize: 1024*1024
    }
});
```

Kuva 22. Multerin asetukset

Luotu objekti voidaan sitten lisätä haluttuihin reitteihin. Reittiin kiinnityksen yhteydessä kutsutaan jotain objektin metodeista määrittämään millaista dataa multer reitillä ottaa vastaan. Kuvassa 23 ensimmäinen reitti määritetään vastaanottamaan yksi tiedosto, jonka tyypiksi määritetään avatar ja toinen reitti käsittelemään tiedostoista koostuvan Array-objektin. (Multer 2017.)

```
app.post('/profile', upload.single('avatar'), function (req, res, next) {
  // req.file is the `avatar` file
  // req.body will hold the text fields, if there were any
})

app.post('/photos/upload', upload.array('photos', 12), function (req, res, next) {
  // req.files is array of `photos` files
  // req.body will contain the text fields, if there were any
})
```

Kuva 23. Multerin kiinnitys reitteihin (Multer 2017.)

Multerin Request-objektiin lisäämä field-osuus sisältää arvoinaan erinäisiä tietoja lähetetystä tiedostosta, esimerkiksi sen alkuperäisen, koon, polun johon tiedosto tallennettiin ja niin edelleen. (Multer 2017.)

2.9 Bcryptjs-moduuli

Bcryptjs on Node.js-moduuli joka toteuttaa Bcrypt-nimistä salausfunktiota, joka suolaa kohdemerkkisarjan. Node.js-palvelimella moduuli hyödyntää Noden sisäänrakennettua crypto-moduulia luotettavien satunnaisnumeroiden tuottamiseksi suolausta varten. Suolan luomiseksi sille annetaan lisäksi työkierrosten määrä joissa suola luodaan. Luotu suola-arvo voidaan sitten lisätä sovelluksen hash-metodissa määritettyyn merkkisarjaan. (Wirtz 2017.)

Esimerkki suolauksesta on nähtävissä kuvassa 24, jossa ensin luodaan suola bcryptjsän genSalt-metodilla, jonka parametreiksi annetaan luontikierrosten määrä sekä asynkroninen funktio, joka palauttaa salt-nimellä tuotetun suolan. Seuraavaksi kutsutaan hash-metodia, jolle annetaan suolattava merkkisarja, suola ja funktio joka palauttaa suolatun version merkkisarjasta nimellä hash. Kuvassa 25 compare-metodia käytetään kahden merkkisarjan tarkastukseen tuotettua suolattua merkkisarjaa vasten. Ylempi merkkisarjoista on sama, jota käytettiin hashattyn arvon luontiin, joten se läpäisee tarkastuksen ja res-palautteen arvoksi annetaan true. Alempi arvo ei läpäise tarkastusta ja funktio palauttaa false. (Wirtz 2017.)

```
var bcrypt = require('bcryptjs');
bcrypt.genSalt(10, function(err, salt) {
    bcrypt.hash("B4c0/\/", salt, function(err, hash) {
        // Store hash in your password DB.
    });
});
```

Kuva 24. Bcryptjs suolauksen käyttö (Wirtz 2017.)

```
// Load hash from your password DB.
bcrypt.compare("B4c0/\/", hash, function(err, res) {
    // res === true
});
bcrypt.compare("not_bacon", hash, function(err, res) {
    // res === false
});
```

Kuva 25. Merkkisarjojen vertaus compare-metodilla (Wirtz 2017.)

2.10 Nodemailer

Nodemailer on Node.js-moduuli joka on tarkoitettu sähköpostien lähettämiseen. Se ei vaadi muita moduuleja toimiakseen. Viesteissä voidaan lähettää tekstiä ja HTML-sisältöä. Viestit lähetetään turvallisesti TLS/STARTTLS-protokollilla. (Nodemailer 2017.)

Esimerkki Nodemailerin käytöstä löytyy kuvasta 26. Nodemailerille määritetään ensin Transporter-tyyppinen transporter-objekti viestin lähettämistä varten, jolle annetaan käy-

tettävä sähköpostipalvelu ja tarvittavat tunnukset sen käyttöön. Objektille voidaan myös antaa erikseen jokin dataprotokolla; oletusasetuksena Nodemailer käyttää SMTP-protokollaa. Lähetettävän viestin lähettäjä, vastaanottaja, otsikko ja sisältö määritetään mailOptions-nimiseen Arrayhin. Viesti lähetetään kutsumalla transporter-objektin sendMail-metodia, jolle annetaan viestin asetukset. Metodi on asynkroninen ja palauttaa valmistuessaan joko error- tai info-objektin. (Nodemailer 2017.)

```
'use strict';
const nodemailer = require('nodemailer');

// create reusable transporter object using the default SMTP transport
let transporter = nodemailer.createTransport({
    service: 'gmail',
    auth: {
        user: 'gmail.user@gmail.com',
        pass: 'yourpass'
    }
});

// setup email data with unicode symbols
let mailOptions = {
    from: '"Fred Foo 👻" <foo@blurdybloop.com>', // sender address
    to: 'bar@blurdybloop.com, baz@blurdybloop.com', // list of receivers
    subject: 'Hello ✔', // Subject line
    text: 'Hello world ?', // plain text body
    html: '<b>Hello world ?</b>' // html body
};

// send mail with defined transport object
transporter.sendMail(mailOptions, (error, info) => {
    if (error) {
        return console.log(error);
    }
    console.log('Message %s sent: %s', info.messageId, info.response);
});
```

Kuva 26. Nodemailerin käyttöönotto ja viestin lähetys (Nodemailer 2017.)

3 Sovellus

Tässä kappaleessa käsitellään kehitysprojektissa syntyneen palvelinsovelluksen eri ominaisuuksia sekä syitä niissä tehtyihin teknisiin ratkaisuihin.

3.1 Teknologiavalinnat

Sovelluksen teknologiavalinnat tehtiin pääasiassa käyttötapaukseen sopivuuden sekä moderniuden pohjalta, mutta merkittävänä tekijänä oli myös pyrkimys valita kehittäjälle vieraita ratkaisuja oppiarvon parantamiseksi.

Node.js on suunniteltu nimenomaisesti verkkosovellusten palvelinpuolen toteuttamiseen ja vaikutti teknisenä ratkaisuna erittäin kiinnostavalta. Se on lisäksi kehittäjien suosima ja ilmeisesti vahvasti työllistävä teknologia, joten se valittiin nopeasti osaksi sovellusta. Express.js-moduuli valikoitui mukaan luonnollisena jatkeena Nodelle toteuttamaan sovelluksen ytimessä olevaa reititystä. Socket.io vaikutti loogiselta valinnalta sovelluksen chat-ominaisuuden kehittämiseen ja sen tapahtumapohjainen ja muuten vapaa rakenne vaikutti myös lupaavalta muiden reaaliaikaisten toimintojen toteuttamiseen. JSON Web Token-teknologia vaikutti projektin alussa keveytensä takia erittäin sopivalta mobiilisovellukseen.

MongoDBeen valintaan sovelluksen tietokannaksi vaikutti pitkälti kehittäjän henkilökohtainen kiinnostus tutustua NoSQL-tietokantoihin. MongoDB on ilmainen, helposti käyttöönotettava ja integroituu sulavasti Node-palvelimeen Mongoose-moduulin avulla. Sen merkittävin kilpailija tietokantavalinnassa oli Redis, joka tippui pois sovelluksen suhteellisen yksinkertaisen tietokantarakenteen sekä mahdollisten palvelinkoneen muistirajoitteiden vuoksi.

Muut teknologiavalinnat ovat Node.js-moduuleja, jotka valittiin ominaisuuksien kehittämisen yhteydessä prosessin virtaviivaistamiseksi.

3.2 Tietokanta ja datamallit

Palvelinsovelluksen suunnittelemiseksi projektissa määritettiin ensin sovelluksen tarvitsema data. Kehitysprojektin tavoitteena oli tuottaa Cordova-pohjainen sovellus Androidille. Käyttäjät luovat sovellusta käyttääkseen sähköpostiosoitteellaan tilin tai kirjautuvat Google-tunnuksillaan. Kirjautuneet käyttäjät voivat luoda ryhmiä ja kutsua niihin muita käyttäjiä. Ryhmien sisällä on ryhmäkohtainen keskustelualue sekä lista käyttäjien valitsemista ruokapaikoista, jotka voi saada näkyviin kartalle.

Tämän kuvauksen perusteella sovelluksen täytyy tallentaa dataa ainakin käyttäjästä ja käyttäjäryhmistä sekä siitä, mihin ryhmiin käyttäjät kuuluvat. Ryhmäkohtaiset listat ruokailupaikoista ovat myös olennaista dataa sovelluksen toiminnalle. Keskusteluhistorian tallentaminen ei ole sovelluksen tavoitteiden kannalta välttämätöntä, mutta parantaa käyttäjäkokemusta, joten se otettiin sovellukseen mukaan.

Sovelluksen MongoDB-tietokannan käyttö tapahtuu Mongoose-moduulin kautta, joka mallintaa dataa Model-luokkina jotka vastaavat rakenteeltaan MongoDB-kokoelmien JSON-rakennetta. Koska MongoDB-tietokanta luo ja mukauttaa kokoelmia dynaamisesti ilman jäykkää mallia, sovelluksen tietokanta luodaan käytännössä käytönaikana näillä Modeleilla.

3.2.1 userModel

Kentän nimi	Asetukset	Kuvaus
_id	ObjectID	MongoDB:n dokumentille automaattisesti generoima ObjectID-arvo. Käytetään sovelluksessa viitteenä käyttäjädokumenttiin nimellä userID.
userEmail	String, required, unique	Käyttäjätiliä luodessa annettu sähköpostiosoite. Käytetään sovelluksessa viitteenä käyttäjädokumenttiin kirjautumisessa sekä käyttäjien välillä.
username	String, required	Käyttäjätilin muille käyttäjille näkyvä nimi sovelluksessa. Muutettavissa.
userPassword	String, required	Käyttäjätilin salasana. Muutettavissa.
verified	Boolean, required	Boolean-arvo joka määrittää onko käyttäjätili aktivoitu. Aktivoimattomat tilit eivät toimi.
GAuth	Boolean, Required	Boolean-arvo joka määrittää onko käyttäjätili luotu Googlen Login-palvelun avulla.
groups	Array	Taulukko joka sisältää _id-arvot ryhmistä joiden jäsen käyttäjä on.
invites	Array	Taulukko, joka sisältää _id-arvot ja ryhmänimet käsittelemättömistä ryhmäkutsuista.

Taulukko 1. userModelin rakenne

userModel-luokka mallintaa sovelluksessa käyttäjää. Sen rakenne on nähtävissä taulukossa 1. Käyttäjädokumentit sisältävät listan ryhmistä joiden jäsen käyttäjä on ylimääräisten tietokantahakujen välttämiseksi.

userModelille on lisäksi määritetty kaksi funktiota. Ensimmäinen ajetaan aina kun userModeliin perustuvaa dokumenttia yritetään tallentaa tai päivittää ja se tarkastaa onko dokumentin salasanaa muokattu. Jos on, uusi salasana suolataan Bcryptjsällä. Toinen funktio on määritetty Modelin metodiksi nimellä comparePassword. Sitä käytetään käyttäjädokumentin suolatun salasanan vertaamiseen kirjautumisyrityksessä annettuun salasanaan Bcryptjsän compare-metodilla.

3.2.2 groupModel

Kentän nimi	Asetukset	Kuvaus
_id	ObjectID	MongoDB:n dokumentille automaattisesti generoima ObjectID-arvo. Käytetään sovelluksessa viitteenä ryhmädokumenttiin nimellä groupID.
groupName	String, required	Käyttäjäryhmän käyttäjille näkyvä nimi. Muutettavissa.
groupAdmin	String, required	Ryhmän luoneen käyttäjän _id-arvo. Määrittää ryhmän hallintaoikeudet.
groupDesc	String, Required	Käyttäjäryhmän kuvaus. Muutettavissa.
members	Array	Kokoelma _id-arvoja käyttäjistä jotka ovat ryhmän jäseniä.
places	Array	Kokoelma Googlen PlaceID-arvoja sijainneista, joita käyttäjät lisäävät ryhmän sijaintilistaan.
voting	Array	Sisältää äänen lähettäneen käyttäjän userID-arvon ja äänestetyn paikan placeID-arvon.

Taulukko 2. groupModelin rakenne

groupModel-luokka mallintaa sovelluksessa käyttäjien luomia ryhmiä. Sen rakenne on nähtävissä taulukossa 2. Ryhmädokumentit sisältävät listan käyttäjistään ylimääräisten tietokantahakujen välttämiseksi.

3.2.3 messageModel

Kentän nimi	Asetukset	Kuvaus
_id	ObjectID	MongoDB:n dokumentille automaattisesti generoima ObjectID-arvo. Käytetään sovelluksessa viitteenä viestidokumenttiin nimellä messageID.
groupID	String, required	_id-arvo ryhmästä, johon viesti kuuluu.
author	String, Required	_id-arvo käyttäjästä, jolle viesti kuuluu.
msg	String, Required	Viestin sisältö.
username	String, required	Viestin lähettäneen käyttäjän käyttäjänimi.
date	String, required	Viestin luomispäivämäärä.
time	String, Required	Viestin luomiskellonaika.

Taulukko 3. messageModelin rakenne

messageModel-luokka mallintaa yksittäisiä keskustelualueen viestejä ja sen rakenne on nähtävissä taulukossa 3. Viestit erotettiin groupModel-mallista omaansa, koska niitä syntyy todennäköisesti ryhmää kohden merkittävä määrä, mikä voisi vaikuttaa negatiivisesti sovelluksen Group-dokumenttien käsittelyyn. Dokumenttiin tallennetaan käyttäjän käyttäjänimi ylimääräisen tietokantahaun välttämiseksi.

3.3 Sovelluksen rakenne

Sovelluksen juurikansiossa on käynnistykseen käytetty päätiedosto ja package.json-tiedosto. Sovelluksen muut osat on jaettu user-, group- ja chat-kansioihin, jotka sisältävät vastaavat Modelit ja Controllerit. Lisäksi sovelluksessa on config-kansio, joka sisältää sovelluksen asetustiedostoja, ja public-kansio, joka sisältää sovelluksessa käytettyjä staattisia tiedostoja.

Server.js on sovelluksen päätiedosto joka käynnistää sovelluksen Express.js-palvelimen ja sisältää sen reitittimet ja reitit. Se kutsuu sovelluksen toiminnollisuudet toteuttavia Cont-

roller-tiedostoja ja liittää niiden metodit reitteihin. Sovellukselle on määritetty kaksi reititintä, joista ensimmäinen on sovelluksen app-objekti, joka hallitsee sovelluksen tilin luontiin ja kirjautumiseen käytettyjä reittejä. authRouter-reititin toteuttaa sovelluksen muut, kirjautumista ja käyttöoikeuksia vaativat reitit. Reittien sekä niitä toteuttavien funktioiden nimeämisessä pyrittiin kuvaaviin nimiin koodin luettavuuden parantamiseksi. Tästä syystä sovelluksessa ei esimerkiksi esiinny Express.jsällä mahdollisia reitityksiä, joissa sama reittinimi palvelee useampaa tarkoitusta HTTP-metodista riippuen.

Config-kansiossa on päätiedoston käyttämä, kaikki sovelluksen asetukset kokoava config.js-tiedosto sekä sen kutsumat erilliset asetustiedostot jotka sisältävät sovelluksen Token-salaisuuden ja sähköpostitunnukset. Config.js-tiedostossa on myös tarvittaessa valmiina säädökset SSL-sertifikaatin käyttöönottoon, jos sovellusta itsessään käytetään HTTPS-palvelimena Nginxin sijasta.

Public-kansio on jaettu kolmeen alikansioon. Socket.io-kansiossa on käyttäjäpuolen tarvitsema socket.io.js-tiedosto. Uploads-kansio sisältää käyttäjien sovellukseen tallentamat avatar- ja ryhmäkuvat. Www-kansiossa on neljä sovelluksesta erillistä HTML-sivua: sovelluksen erillisenä kotisivuna toimiva index.html, käyttöehdot sisältävä tos.html ja käyttäjätilin sähköpostivahvistuksessa käytetyt verify.html ja fail.html.

3.4 Tunnistautuminen ja käyttäjäoikeudet

Sovellus tunnistaa käyttäjän JSON Web Tokenin perusteella, jonka käyttäjä saa palvelimelta onnistuneen kirjautumisen yhteydessä. Token sisältää käyttäjän userID-arvon sekä omat luonti- ja vanhentumisaikansa. Token luodaan palvelimella säilötyn salaisen avaimen avulla.

Token lähetetään HTTP-pyynnöissä kaikille tunnistautumista vaativille, authRouter-reitittimen toteuttamille reiteille. Token tarkastetaan ennen reiteille pääsyä kuvassa 27 näkyvässä tokenVerifier-nimisessä väliohjelmassa, joka varmistaa tokenin digitaalisen allekirjoituksen luontisalaisuutta vasten. Jos Tokenia ei ollut tai se ei läpäise allekirjoitustarkastusta, tokenVerifier päättää pyynnön käsittelyn Unauthorized-vastauksella.. Jos tarkastus läpäistään tokenVerifier antaa palvelupyynnön edetä kutsumalla käsittelyn seuraavaa osaa. Dekoodattu Token lisätään osaksi Expressin req-objektia, jotta sen sisältö on myöhemmin saatavilla.

```
var tokenVerifier = function(req, res, next){
    if(!req.headers["authorization"]) return res.sendStatus(401);

    var token = req.headers["authorization"].replace(/^Bearer\s/, '');

    if(token){
        jwt.verify(token, config.secret, function(err, decoded){
            if(err){
                console.log("Token authentication failed.");
                return res.status(401).send({
                    success: false,
                    message: "Token authentication failed."
                });
            }else{
                console.log("Token authenticated.")
                req.decoded = decoded;
                next();
            }
        });
    }else{
        return res.status(401).send({
           success: false,
           message: "No token was provided.",
        });
    }
};

authRouter.use(tokenVerifier);
```

Kuva 27. tokenVerifier-väliohjelma

Käyttäjäoikeudet tarkastetaan erillisillä väliohjelmilla reiteillä, jotka käsittelevät jotakin käyttäjää tai ryhmää. Jos palvelupyyntö ei läpäise tarkastuksia, väliohjelmat päättävät pyynnön käsittelyn lähettämällä käyttäjälle Unauthorized-vastauksen. Jos tarkastukset läpäistään ne kutsuvat käsittelyn seuraavaa osaa. Kuvan 28 userChecker-väliohjelma tarkastaa palvelupyynnössä lähetetyn userID-arvon Tokenista löytyvää userID-arvoa vasten: käyttäjä voi siis käytännössä hakea ja muokata vain omaa käyttäjädataansa.

```
var userChecker = function(req, res, next){
    if(!req.decoded) return res.sendStatus(401);

    console.log("Checking requested user against authentication...");
    var userID = req.decoded.userID;

    if(userID == req.body.userID || userID == req.headers["userid"]){
        console.log("Requested user matches authentication.");
        next();
    }else{
        console.log("Requested user does not match authentication.");
            return res.status(401).send({
                success: false,
                message: "Requested user does not match authentication.",
            });
    }
};
```

Kuva 28 userChecker-väliohjelma

Ryhmiä käsittelevät reitit turvaa kuvan 29 groupChecker-väliohjelma, joka tarkastaa tietokannasta löytyykö palvelupyynnössä lähetettyä groupIDtä vastaavan ryhmän jäsentiedoista Tokenista saatu userID-arvo. Koska ryhmäkäyttöoikeuksien tarkastus vaatii erillisen tietokantahaun, sovellus ei toteuta täysin JWT:n periaatteita, joiden mukaan token voi sisältää kaikki tarvittavat käyttöoikeustiedot. Ratkaisu tehtiin, koska Tokenin viiveetön päivittäminen mahdollisesti usein muuttuvien jäsenyystietojen mukana osoittautui hankalaksi.

```
var groupChecker = function(req, res, next){
    if(!req.decoded) return res.sendStatus(401);
    var userID = req.decoded.userID;
    if(req.body.groupid){
        var groupID = req.body.groupid;
    }else if(req.headers["groupid"]){
        var groupID = req.headers["groupid"];
    }
    if(groupID){
        console.log("Checking members in: " + groupID);
        Group.findOne({_id:groupID}, "members.memberID", function(err, results){
            if(err){
                console.log(err);
                return res.status(500).send({
                    success: false,
                    message: "Database error.",
                });
            }
            if(results){
                var i = 0;
                while(i < results.members.length){
                    var memberID = results.members[i].memberID;
                    var matches = false;
                    if(userID == memberID){
                        console.log("User " + userID + "is a member of " + groupID);
                        matches = true;
                        next();
                        break;
                    }
                    i++;
                };
                if(matches == false){
                    console.log("User is not a member of that group.");
                    return res.status(401).send({
                        success: false,
                        message: "User is not a member of that group.",
                    });
                }
            }else{
                console.log("No such group.");
                return res.status(404).send({
                    success: false,
                    message: "No such group.",
                });
            }
        });
    }else{···
    }
```

Kuva 29. groupChecker-väliohjelma

3.5 Käyttäjät

Sovelluksen userModel-mallia, siis käyttäjätietoja, käsittelevää logiikka toteuttaa user-Controller. Se sisältää kaikki sovelluksen userModel-mallia käsittelevät funktiot: ryhmien jäsenyyden käsittely toteutetaan groupControllerissa.

userController käyttää useimmissa toiminnoissaan viitteenään käyttäjädokumentteihin userID-arvoa, joka vastaa tietokannassa MongoDBeen dokumentin luomisen yhteydessä automaattisesti generoimaa ObjectID-kenttää. userID-arvo on uniikki ja sovelluksen ulkopuolella merkityksetön, joten se on käytännöllinen sekä tietoturvaltaan paras käyttäjäviite siirrettäväksi käyttäjälaitteen ja palvelimen välillä. userEmail-arvoa käytetään viitteenä käyttäjädokumenttiin funktioissa, esimerkiksi sisäänkirjautumisessa, joissa käyttäjä syöttää kohdedokumentin tiedot.

3.5.1 Tilin luonti ja kirjautuminen

Sovelluksen paikallinen käyttäjätilin luonti sekä kirjautuminen toteutetaan userControllerissa funktioissa signUp ja login. SignUp-funktiolle lähetetään tilin luomiseksi sähköpostiosoite, käyttäjänimi sekä salasana. Funktio tarkastaa ensin löytyykö User-kokoelmasta jo dokumentti samalla sähköpostiosoitteella ja jatkaa vain jos sellaista ei ole, muodostaen lähetetystä datasta newUser-objektin, joka tallennetaan tietokantaan. Uudelle käyttäjädokumentille asetetaan verified-kentän arvoksi false ja käyttäjän sähköpostiin lähetetään varmistuslinkki, joka kutsuu verify-funktiota.

Verify-funktio saa lähetetyn linkin parametreista käyttäjän sähköpostiosoitteen ja userID-arvot ja hakee tietokannasta niitten perusteella ei-varmistetun käyttäjän. Jos käyttäjädokumentti löytyy, sen verified-kentän arvoksi muutetaan true, mikä aktivoi käyttäjätilin.

Login-funktio vaatii sähköpostiosoitteen sekä salasanan. Sovellus tarkastaa löytyykö tietokannasta annetulla sähköpostiosoitteella aktivoitu käyttäjätili ja vertaa lähetettyä salasanaa tietokannassa olevaan suolattuun salasanaan Bcryptjs-moduulin comparePassword-metodilla. Jos tarkastus läpäistään, sovellus luo ja lähettää käyttäjälle Tokenin. Jos käyttäjää ei löydy tai salasanat eivät vastaa toisiaan, käyttäjälle lähetetään vastauksena ilmoitus tapahtuneesta.

Paikallisen tilin lisäksi sovellus sallii Google-tilin käytön. Kirjautuminen Googlen palveluihin on toteutettu käyttäjäpuolella, joka saa kirjautumisen onnistuessa Googlen käyttämän idTokenin, joka lähetetään palvelimelle googleAuth-funktioon. Funktiossa idToken tarkastetaan uudestaan Googlen omalla tunnistautumistyökalulla. Jos tarkastus onnistuu, sovel-

lus tekee tietokantahaun idTokenista saadulla sähköpostiosoitteella. Jos sillä on jo sovelluksessa tili, käyttäjälle luodaan ja lähetetään sovelluksen oma Token. Jos tiliä ei ole, se luodaan automaattisesti idTokenista saadun sähköpostiosoitteen ja käyttäjänimen perusteella ja käyttäjälle lähetetään Token. Googlen kautta luodun tilin tietokantadokumentin GAuth-kentän arvoksi annetaan true, mikä käytännössä tarkoittaa, että tilille voi kirjautua vain Googlen Login-palvelun kautta.

3.5.2 Käyttäjädatan haku ja muokkaus

Käyttäjä voi sovelluksessa hakea tarkemmat tiedot vain omasta käyttäjätilistään. Toiminnollisuuden toteuttaa getProfile-funktio, joka hakee käyttädokumentin lähetetyn userID-arvon perusteella. Mahdolliset kutsut ryhmiin haetaan käyttäjädokumentin invites-listasta getInvites-funktiolla userID-arvolla.

Käyttäjä voi halutessaan muokata tilinsä käyttäjänimeä tai salasanaa, sekä vaihtaa sovelluksessa näkyvän profiilikuvansa. Toiminnot ovat funktioissa changeUsername, changePassword ja setAvatar, joille annetaan kohdedokumentin userID-arvo sekä uusi arvo muutettavalle tiedolle. changePassword-funktio vaatii lisäksi vanhan salasanan, joka tarkastetaan ennen muutoksen tekoa.

3.6 Ryhmät

groupController vastaa ryhmiä sekä jäsenyyttä käsittelevästä ohjelmistologiikasta. Se viittaa toiminnoissaan ryhmiin groupID-nimisellä arvolla, joka vastaa MongoDBeen dokumenteille luomisen yhteydessä satunnaisesti tuottamia, uniikkeja ObjectID-arvoja.

3.6.1 Ryhmien haku ja käsittely

createGroup-funktio luo ryhmän palvelupyynnössä annetun ryhmänimen ja kuvauksen perusteella. Ryhmien nimet eivät ole uniikkeja, joten funktio ei aja tarkastuksia tietokannassa. Ryhmälle voidaan vaihtoehtoisesti antaa luomisen yhteydessä kuva, joka vastaanotetaan palvelimelle setGroupImage-funktiolla, joka tallentaa kuvan palvelimelle erilliseen groups-kansioon ja nimeää kuvan ryhmän groupID-arvolla.

Ryhmän nimi ja kuvaus voidaan myöhemmin vaihtaa alterGroup-funktiolla, joka ajettaessa tarkastaa palvelupyynnössä tulleen datan perusteella kumpaa arvoa sen on tarkoitus muokata. Kuvanvaihdosta vastaa edelleen setGroupImage-funktio. Ryhmän poisto tapahtuu deleteGroup-funktiolla, joka tuhoaa ryhmän sekä sen viestihistorian tietokannasta ja poistaa sille mahdollisesti annetun kuvan palvelimelta.

getGroups-funktio hakee perustiedot kaikista ryhmistä joiden jäsenenä userID:llä tunnistettu käyttäjä on. Yksittäisen ryhmän kaikki tiedot haetaan getGroup-funktiolla ryhmän groupID-arvolla.

3.6.2 Jäsenyyden hallinta

Ryhmän luonut käyttäjä voi lähettää jäsenkutsuja ryhmään muille käyttäjille. Invite-funktio ottaa palvelupyynnöstä vastaan kohdekäyttäjän sähköpostiosoitteen, kohderyhmän groupID-arvon sekä ryhmän nimen, joka tallennetaan groupIDen kanssa kutsuttavan käyttäjän tietokantadokumentin invites-kenttään erillisten tietokantahakujen välttämiseksi ryhmän nimeämiseksi kun kutsu näytetään käyttäjälle. Jos kohdekäyttäjä on jo ryhmän jäsen, funktio ilmoittaa siitä. Jos käyttäjälle on jo lähetetty kutsu jota ei vielä ole käsitelty, sovellus ei erikseen ilmoita asiasta.

Jos käyttäjä hyväksyy kutsun, acceptInvitation-funktio poistaa sen käyttäjän dokumentista userID-arvon avulla ja lisää käyttäjän ryhmän jäseneksi. Ennen hyväksymistä funktio tarkastaa kutsun olemassaolon tekemällä tietokantahaun käyttäjän userID- ja kutsun groupID-arvoilla. declineInvitation-funktio vain poistaa kutsun käyttäjän tiedoista. Vastauksesta ei kummassakaan tapauksessa erikseen ilmoiteta kutsun lähettäjälle.

3.7 Chat

Sovelluksen keskustelualue on toteutettu Socket.io-moduulilla ja sen määritykset löytyvät chatController-tiedosta. Käyttäjän kirjautuessa sisään sovellukseen hänelle luodaan socket-objekti, joka vastaa käyttäjää Socket.ion toiminnassa. Luonnin yhteydessä socketin ID-arvo kirjataan käyttäjän userID-arvon kanssa socketUsers-listalle, jotta sockettiin voidaan tarvittaessa viitata käyttäjän userID:llä. Jos sovellus havaitsee disconnect-tapahtuman, eli käyttäjä on kirjautunut ulos tai sulkenut sovelluksen, käyttäjä poistetaan socketUsers-listalta.

Keskustelualueet on toteutettu ryhmäkohtaisesti käyttäen Socket.ion rooms-toiminnollisuutta viestin jakamiseksi erillisille kanaville. Käyttäjän siirtyessä johonkin ryhmään listalta hänet lisätään ryhmän keskusteluhuoneeseen. Samalla tehdään kutsu joka palauttaa getMessages-funktion tietokannasta ryhmän groupID-arvolla hakeman viestihistorian.

Jos käyttäjä poistetaan listalta, groupControllerin removefromGroup-funktio kutsuu chatControllerin removeSocket-funktiota, joka käy läpi socketUsers-listan poistetun käyttä-

jän userID-arvolla. Jos käyttäjä on sovelluksessa, arvo löytyy ja funktio käyttää siihen kiinnitettyä socketID-arvoa käyttäjän poistamiseen ryhmän keskusteluhuoneesta. Lisäksi funktio luo removedFromGroup-tapahtuman, joka siirtää käyttäjän näkymän pois ryhmästä takaisin listasivulle.

Käyttäjien viestit lähetetään palvelimelle message-tapahtumana, jonka vastaanottaessaan palvelin ottaa viestissä lähetetyn datan ja muodostaa niistä uuden Message-dokumentin, joka tallennetaan tietokantaan. Tallennuksen onnistuttua viesti lähetetään eteenpäin keskusteluhuoneen muille jäsenille. Toteutus kirjoittaa tietokantaan joka viestin kohdalla, mikä aiheuttaa suhteessa merkittävän määrän tietokantakutsuja: ratkaisu voi siis olla tarpeettoman raskas.

3.8 Ruokailupaikat

Ruokailupaikkojen hakeminen sovellukseen on toteutettu Google Maps-palvelun avulla, jonka käyttöehdoista johtuen palvelusta saadusta datasta voidaan palvelimelle tallentaa vain sijainnin placeID-arvo. Tästä johtuen sovelluksen sijaintikäsittely rajoittuu paikkojen tallentamiseen ryhmäkohtaisille listoille savePlace-funktiolla, niitten poistamiseen listoilta deletePlace-funktiolla sekä ryhmien placeID-arvojen hakemiseen getPlaces-funktiolla. PlaceIDt lähetetään käyttäjäpuolella, jossa niitä vastaavat sijaintiedot haetaan Googlen palvelun avulla. Funktiot käyttävät toiminnassaan ainoastaan groupID- ja placeID-arvoja. Ruokailupaikkojen lisäys ja poisto tuottavat Socket.io-tapahtumia, jotka päivittävät muiden ryhmän jäsenten listat.

Ruokailupaikkojen valinnassa käytetty äänestys jäi ensimmäisessä julkaisuversiossa keskeneräiseksi teknisistä ongelmista johtuen, mutta palvelinsovelluksessa on pääpiirteittäin Socket.iolla toteutetut toiminnollisuudet sen toteuttamiseen. Käyttäjän äänestäessä käyttäjäsovellus lähettää palvelimelle tapahtuman, jonka vastaanottaessaan palvelinsovellus tallentaa äänen ryhmän tietokantadokumenttiin. Kun tallennus on suoritettu palvelin lähettää äänen tiedot sisältävän tapahtumaviestin ryhmän muille jäsenille. Äänestyksen päättämiseksi palvelin reagoi erilliseen päättymistapahtumaan, jonka saadessaan se lähettää ryhmän jäsenille äänestyksen lopputuloksen ja poistaa sitten äänestyksen tietokannasta. Ryhmän Socket.io-huoneeseen kesken äänestyksen liittyvät saavat nykyisen äänestysdatan getPlaces-funktiolta.

3.9 Sähköpostiviestien lähetys

Sovelluksen sähköpostitarpeista vastaa Nodemailerilla toteutettu emailService, joka käyttää sovelluksen Gmail-tiliä viestien lähettämiseen. Moduulin ainoa funktio on tällä hetkellä

sendConfirmation, jota kutsutaan kun uusi käyttäjätili luodaan. Se ottaa vastaan käyttäjän sähköpostitunnuksen ja userID-arvon ja rakentaa niitten perusteella linkin joka kutsuu groupControllerin verify-funktiota. Linkki lisätään osaksi sähköpostiviestiä ja viesti lähetetään.

Jatkokehityksessä moduuliin tullaan lisäämään ainakin metodi, jolla käyttäjälle lähetetään sähköpostiviesti salasanan uusimiseksi.

3.10 Virheenkäsittely

Aikarajoitteiden johdosta sovelluksen virheenkäsittelyä ei ehditty tehdä keskitetysti. Virhekäsittely on siis toteutettu paikallisesti jokaisen funktion sisällä, mikä tuottaa turhaa koodin toistoa ja heikentää sovelluksen ylläpidettävyyttä. Virheenkäsittelyn ainoat sovelluksen kaatavat osuudet liittyvät sovelluksen käynnistykseen, joiden on parasta antaa keskeyttää sovelluksen toiminta toiminnanaikaisten virhetilanteiden välttämiseksi.

Sovelluksen toiminnanaikainen virheenhallinta kirjaa virheviestit sovelluksen virhelogiin ja ilmoittaa virheestä käyttäjälle vastausviestillä. Sovelluksen funktiot tarkastavat ajonsa alussa, että hakupyynnössä on lähetetty funktioiden tarvitsema data ja keskeyttävät datan puuttuessa käsittelyn vastauskoodilla 400, joka ilmoittaa palvelupyynnön olevan virheellinen. Jos tietokantatoiminnoissa tapahtuu virhe, palvelin lähettää käyttäjälle sisäisen virheen vastauskoodin 500 ja ilmoituksen tietokantaongelmasta. Jos palvelupyynnössä viitattua dataa ei löydy, palvelin ilmoittaa asiasta käyttäjälle funktiokohtaisella viestillä. Jos käyttäjä yrittää päästä käsiksi dataan, johon hänellä ei enää ole käyttöoikeuksia, palvelin vastaa pyyntöön vastauskoodilla 401, joka merkitsee, että pyyntö on yrittänyt suorittaa toiminnon johon sillä ei ole oikeuksia.

4 Lopputulos ja retrospektiivi

DinnerChatter-sovellus on kokonaisuudessa opinnäytetyöprojektin lopussa myöhäisessä alphassa. Palvelinpuolelle on toteutettu äänestystä lukuun ottamatta kaikki projektisuunnitelmassa mainitut ominaisuudet, mutta monet vaativat vielä optimointia ja lisätestausta. Sovelluksen kehitys oli suhteellisen hidasta johtuen lähinnä kehittäjien kokemattomuudesta, joka näkyi myös työmäärän aliarviointina kehityksen alkupuolella.

Myöhäisestä julkaisusta johtuen sovellukselle ei ole saatavilla käyttäjäpalautetta tai tietoa mahdollisista laajemman käyttäjäjoukon aiheuttamista teknisistä ongelmista. Sovellus ei myöskään tällä hetkellä tee millään tavalla rahallista tuottoa, mutta kehityksen edettyä hieman pidemmälle siihen voitaisiin käyttäjämääristä riippuen hankkia mainostusta.

4.1 Käytettyjen teknologioiden soveltuvuus

Kaiken kaikkiaan Node.js-alustainen palvelin osoittautui erinomaiseksi valinnaksi. Kehitetty sovellus on odotetusti prosessorivaatimuksiltaan suhteessa kevyt verrattuna sen tekemiin tietokantahaku- ja kirjoitustoimintoihin, joten Node.jsän IO-toiminnan tehokkuutta painottava arkkitehtuuri palvelee sovelluksen tarkoituksia erittäin hyvin. Node.jsälle on saatavissa paljon dokumentaatiota ja sen käyttöönotto on varsin helppoa. Aiempi kokemus normaalin Javascriptin kanssa nopeutti uuden teknologian omaksumista. Palvelinsovelluksen kehitystä helpotti myös merkittävästi Node.js-alustalle saatavissa oleva laaja kokoelma eri tarkoituksiin suunniteltuja valmiita moduuleja, joiden käyttöönotolla voitiin lyhentää sovelluksen eri toiminnollisuuksien kehitysaikaa.

Sovelluksen ydintoiminta perustuu ExpressJSään, jolla palvelinsovellukseen kehitettiin käyttäjäsovelluksen palvelupyynnöt käsittelevät reititykset ja joka palveli tässä tarkoituksessa mainiosti. Se on rakenteeltaan intuitiivinen ja tästä syystä nopea oppia. Tarvittavien reitittimien ja reittien määrittäminen voitiin sen avulla tehdä varsin nopeasti ja tehokkaasti ja reittien sitominen palvelinsovelluksen funktioihin oli helppoa. Sen reiteille määritettävät väliohjelmat osoittautuivat käytännölliseksi tavaksi toteuttaa käyttäjän tunnistus ja käyttöoikeuksien hallinta.

Pelkän SQL-taustan perusteella MongoDB:n NoSQL-malliin siirtyminen oli aluksi hieman hankalaa, mutta sen arkkitehtuuri on pohjimmiltaan varsin yksinkertainen ja JSON-pohjainen datamallinnus yksinkertaisti osaltaan käyttöönottoa Javascriptiä käyttävässä sovelluksessa. Mongoose-moduulin datamallinnusmenetelmät ja valmiit tietokantafunktiot tekivät MongoDB:n yhdistämisestä Node-sovellukseen huomattavasti helpompaa.

Socket.io palveli myös tarkoituksensa sovelluksen keskustelualueiden kehityksessä kiitettävästi, lukuun ottamatta joitain ongelmakohtia liittyen esimerkiksi tiettyyn käyttäjäsockettiin viittaamiseen sovelluksen omalla userID-arvolla. Sitä käytetään myös kautta sovelluksen näkymien reaaliaikaiseen päivitykseen.

4.2 Ongelmat ja jatkokehitys

Jatkokehityksen ensimmäinen prioriteetti on projektisuunnitelmassa mainitun äänestystoiminnon saattaminen käyttökuntoon. Palvelinpuolella on jo pääasiassa valmiudet äänestyksen toteuttamiseen, mutta ominaisuutta ei vielä ole otettu käyttöön ja se on testaamaton. Sovelluksesta puuttuu myös vielä joitakin käyttäjän käyttökokemusta parantavia ominaisuuksia, kuten varsin olennainen mahdollisuus käyttäjätilin salasanan uusimiseen sähköpostin välityksellä. Tämä toiminnollisuus on varsin helppo toteuttaa ja tulee olemaan yksi jatkokehityslistan ensimmäisistä projekteista.

Keskeisten toiminnollisuuksien toteutuksen jälkeen palvelinpuolen tärkeimmäksi jatkokehitysprojektiksi nousee todennäköisesti optimointi. Rajaavin tekijä sovelluksen skaalautumisessa tulee oletettavasti olemaan tietokantatoiminta. Tietokantatoiminnot eivät sovelluksessa nykyisellään ole yksissään erityisen raskaita, mutta niitä tehdään varsin usein ja esimerkiksi viestihistorian kaltaiset nopeasti kasvavat tallenteet voivat tulevaisuudessa osoittautua raskaiksi jos hakuja ei rajata. Käytännöllisin ratkaisu palvelinresurssien säästämiseksi olisi todennäköisesti vanhempien viestien automaattinen poisto kun jokin viestimäärä saavutetaan.

Tietokannan tehokkuus hyötyisi myös taulujen indeksoinnista, jota ei kehitysprojektin aikarajoissa ehditty tekemään. Tietokantatoimintojen määrää voitaisiin todennäköisesti myös rajata muuttamalla joittenkin kokoelmien rakennetta. Esimerkiksi jäsenyystietojen tallennus tehdään sovelluksen julkaisuversiossa käyttäjä- ja ryhmäkokoelmiin johtuen sovelluskehityksen alussa tehdyistä ja koodiin aikarajoitteiden takia jääneistä ratkaisuista. Jäsenyystietojen siirto erilliseen kokoelmaan saattaisi osoittautua tehokkaammaksi ratkaisuksi. Sovelluksesta voi myöhemmässä optimoinnissa löytyä tarpeettomia tietokantatoimintoja ja sovelluksen käyttäjäpuolta saattaa olla mahdollista muokata tietokantatoimintojen harventamiseksi.

Mahdollisiin myöhemmin kehitettäviin uusiin toiminnollisuuksiin voisi esimerkiksi kuulua Google Maps-palvelun käytön laajentaminen niin, että kartalla näytettäisiin reittiohjeet äänestyksen voittaneeseen ruokailupaikkaan. Käyttäjät voisivat myös vaihtoehtoisesti

antaa sovelluksen jakaa Google Maps-palvelun avulla sijaintinsa tapaamisten organisoinnin helpottamiseksi – tällainen toiminto tosin lisää sovelluksen tietoturvatarpeita, eikä muitten toiminnollisuuksien kanssa välttämättä ole erityisen hyödyllinen.

Socket.ion käyttöä sovelluksessa voitaisiin laajentaa erilaisten reaaliaikaisten toimintojen toteuttamiseksi. Sitä käytetään jo joissain sovelluksen näkymissä niitten päivittämiseksi muitten käyttäjien muokatessa näkymien dataa.

4.3 Palvelin ja palvelinsovelluksen käyttöönotto

Palvelinsovellusta varten vuokrattiin pilvipalvelin DigitalOcean-palvelusta. Käyttöönottoa varten sovellukselle ostettiin myös verkkotunnus nimellä oppariteam.com NameCheapiltä. Tietoliikenne käyttäjälaitteiden ja palvelimen välillä tapahtuu HTTPS-protokollalla, jonka toteuttamiseen vaadittu SSL-sertifikaatti saatiin ilmaiseksi LetsEncrypt-projektin CertBot-sovelluksella.

Sovelluksen palvelinkone käyttää Ubuntu 16.04-käyttöjärjestelmää. Testausta ja alustavaa käyttöönsaattoa varten hankitun palvelimen toimintaominaisuudet ovat:

- 1GB välimuistia
- 1x 1.8GHz prosessoriydin
- 30GB SSD-kovalevy
- 2TB datasiirtonopeus

Palvelinkoneella on käytössä Nginx-palvelin, joka vastaanottaa HTTPS-pyyntöjä osoitteessa dinnerchatter.oppariteam.com ja siirtää ne osoitteeseen localhost:8080, jota palvelinsovellus kuuntelee. Palvelimen SSL-suojaus on siis toteutettu Nginx-palvelimen asetuksena. Palvelimen palomuuri hyväksyy yhteydenottoja vain sovelluksen käyttämään Nginxin HTTPS-porttiin ja OpenSSH:n porttiin, jota käytetään palvelimen hallintaan.

4.4 Oma oppimiseni

Projekti oli ensimmäinen kehitysprojekti, jossa olin yksin vastuussa palvelinpuolen kehitysestä tai muusta suuresta osuudesta projektia. Aikaisempi kokemukseni sovelluskehitysprojekteissa on myös pääasiassa rajoittunut sovelluksen käyttäjäpuolelle. Projekti oli jo tästä syystä erittäin hyvä oppimiskokemus sovelluskehityksestä yleisesti ja erityisesti verkkokehityksestä. Sovellus on myös ensimmäinen julkaisuni, jota varten järjestin myös ensimmäistä kertaa palvelimen sekä muut sovelluksen käyttöönottoon vaadittavat, varsinaisen kehityksen ulkopuoliset tarpeet.

Koska valitsin palvelinkehitykseen tarkoituksellisesti itselleni aiemmin pääasiassa tuntemattomia teknologioita, projekti oli alusta loppuun valtava oppimiskokemus moderneista verkkokehitysmenetelmistä. Projekti oli kaiken kaikkiaan mielestäni suurin ja paras yksittäinen oppimistapaus opiskeluni aikana ja suosittelen vastaavaa projektia lämpimästi verkkokehityksestä kiinnostuneille.

Lähteet

Basho 2017. NoSQL Databases Explained | What is a NoSQL Database | Basho. Luettavissa: http://basho.com/resources/nosql-databases/. Luettu 16.4.2017.

Express.js 2017a. Express – Node Application Framework. Luettavissa: https://expressjs.com/. Luettu 14.4.2017.

Express.js 2017b. Express 4.x – API Reference. Luettavissa: https://expressjs.com/en/4x/api.html. Luettu 14.4.2017.

Express.js 2017c. Express routing. Luettavissa: https://expressjs.com/en/guide/routing.html. Luettu 14.4.2017.

Express.js 2017d. Writing middleware for use in Express apps. Luettavissa: https://expressjs.com/en/guide/writing-middleware.html. Luettu 14.4.2017.

Future Processing 22.4.2015. On problems with threads in Node.js. Luettavissa: https://www.future-processing.pl/blog/on-problems-with-threads-in-node-js/. Luettu 12.4.2017.

JSON 2017. JSON. Luettavissa: http://www.json.org/. Luettu 18.5.2017.

JWT 2017. JSON Web Token Introduction – jwt.io. Luettavissa: https://jwt.io/introduction/. Luettu 15.4.2017.

Jon Stokes 2017. Ask Ars: What is a CPU thread? Luettavissa: https://arstechnica.com/business/2011/04/ask-ars-what-is-a-cpu-thread/. Luettu 17.4.2017.

Keymetrics 2017. PM2: Production process manager for Node.js apps with a built-in load balancer. Luettavissa: https://github.com/Unitech/pm2. Luettu 19.5.2017.

Kim, C. S. 2015. Understanding module.exports and exports in Node.js. Luettavissa: https://www.sitepoint.com/understanding-module-exports-exports-node-js/. Luettu 13.4.2017.

MongoDB 2017a. Introduction to MongoDB. Luettavissa: https://docs.mongodb.com/getting-started/shell/introduction/. Luettu 17.4.2017.

MongoDB 2017b. Install MongoDB Community Edition on Windows. Luettavissa: https://docs.mongodb.com/getting-started/shell/tutorial/install-mongodb-on-windows/. Luettu 17.4.2017.

MongoDB 2017c. MongoDB Shell (mongo) – Getting started with MongoDB 3.0.4. Luettavissa: https://docs.mongodb.com/getting-started/shell/client/. Luettu 17.4.2017.

MongooseJS 2017a. MongoDB object modelling designed to work in an asynchronous environment. Luettavissa: https://github.com/Automattic/mongoose. Luettu 17.4.2017.

MongooseJS 2017b. Mongoose Schemas v4.9.8. Luettavissa: http://mongoosejs.com/docs/guide.html. Luettu 17.4.2017.

MongooseJS 2017c. Mongoose Models v4.9.8. Luettavissa: http://mongoosejs.com/docs/models.html. Luettu 17.4.2017.

Multer 2017. expressjs/multer Node.js middleware for handling multipart/form-data. Luettavissa: https://github.com/expressjs/multer. Luettu 16.4.2017.

NPM 2017a. 01 – What is NPM?. Luettavissa: https://docs.npmjs.com/getting-started/what-is-npm. Luettu 13.4.2017.

NPM 2017b. 04 – Installing npm packages locally. Luettavissa: https://docs.npmjs.com/getting-started/installing-npm-packages-locally. Luettu: 13.4.2017.

NPM 2017c. 05 – Using a package.json. Luettavissa: https://docs.npmjs.com/getting-started/using-a-package.json. Luettu: 13.4.2017.

Nginx 2017. Welcome to NGINX Wiki | NGINX. Luettavissa: https://www.nginx.com/resources/wiki/. Luettu 16.5.2017.

Node.js 2017a. About. Luettavissa: https://nodejs.org/en/about/. Luettu 11.4.2017.

Node.js 2017b. HTTP. Luettavissa: https://nodejs.org/api/http.html. Luettu 11.4.2017.

Node.js 2017c. Modules. Luettavissa: https://nodejs.org/api/modules.html. Luettu 11.4.2017.

Nodemailer 2017. Nodemailer. Luettavissa: https://nodemailer.com/about/. Luettu 12.5.2017.

Pavan, P. 2013. HTTP: The Protocol Every Web Developer Must Know – Part 1. Luettavissa: https://code.tutsplus.com/tutorials/http-the-protocol-every-web-developer-must-know-part-1--net-31177. Luettu 17.5.2017.

Socket.io 2017a. Real-time application framework (Node.js server). Luettavissa: https://github.com/socketio/socket.io. Luettu 17.5.2017.

Socket.io 2017b. Socket.IO – Chat. Luettavissa: https://socket.io/get-started/chat. Luettu 17.5.2017.

Teixeira, P. 2017. Professional Node.js: Building Javascript Based Scalable Software. John Wiley & Sons. s. 15-17.

TutorialsPoint 2017a. C++ Overview. Luettavissa: https://www.tutorialspoint.com/cplusplus/cpp_overview.htm. Luettu: 19.5.2017.

TutorialsPoint 2017b. Node.js – Event Loop. Luettavissa: https://www.tutorialspoint.com/nodejs/nodejs_event_loop.htm. Luettu 14.5.2017.

Vagg, R. 24.6.2014. Why Asynchronous? Luettavissa: https://nodesource.com/blog/why-asynchronous/. Luettu 12.4.2017.

W3Schools 2017. Javascript Tutorial. Luettavissa: https://www.w3schools.com/js/. Luettu 16.5.2017.

Wilson, D. 2017. body-parser. Luettavissa: https://www.npmjs.com/package/body-parser. Luettu 15.4.2017.

Wirtz, D. 2017. bcryptjs. Luettavissa: https://www.npmjs.com/package/bcryptjs. Luettu 20.4.2017.

Liitteet

Liite 1. Linkit sovelluksen sivuille

Github:

https://github.com/TehOlli/Ruokasoftaoppari

Google Play Store:

https://play.google.com/store/apps/details?id=com.app.dinnerchatter