Opinnäytetyö (AMK)

Tietojenkäsittely

2018

Ville Vartiainen

TAUSTAJÄRJESTELMÄN RYHMÄTOIMINNALLISUUKSIEN KEHITTÄMINEN

– case Kyyti Group Ltd.

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietojenkäsittely

2018 | 34 sivua

Ville Vartiainen

TAUSTAJÄRJESTELMÄN RYHMÄTOIMINNALLISUUKSIEN KEHITTÄMINEN

- case Kyyti Group Ltd.

Opinnäytetyön tarkoituksena oli kehittää ryhmätoiminnallisuuksia yrityksen taustajärjestelmiin. Käyttäjien piti voida liittyä ryhmään sekä ryhmän jäsenyydellä tuli olla rajoitteita, esimerkiksi kysyttäviä lisätietoja tai vahvistetun sähköpostin verkko-osoite.

Opinnäytetyössä käydään ensin läpi toimeksiannon kuvaus. Sen jälkeen kuvataan käytettyjä teknologioita sekä olemassa olevaa järjestelmää. Lopuksi arvioidaan toimeksiannon lopputulosta.

Toimeksiannon lopputulos otettiin käyttöön tuotantopalvelimilla, ja se on edelleen aktiivisesti kehityksessä. Opinnäytetyössä kuvataan taustajärjestelmän toiminnallisuutta, mutta ei puututa esimerkiksi käyttöliittymään. Käyttöliittymä rakennetaan taustajärjestelmän päälle sen valmistuttua.

ASIASANAT:

Järjestelmäsuunnittelu, sovelluskehitys, JavaScript, sovellusarkkitehtuuri

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

BBA

2018 | 34 pages

Ville Vartiainen

DESIGN AND DEVELOPMENT OF GROUP FUNCTIONALITY FOR BACK-END SERVICES

- case Kyyti Group Ltd.

The purpose of this thesis was to develop and integrate group functionality to existing back-end system of Kyyti Group Ltd. Users had to be able to join different kind of groups and there have to be some constraints set to become an active member of the group.

First this thesis explains the requirements set for the system. Secondly, this thesis examines and presents the technologies and the current system and its state. After that, the development of the new system components is explained and the outcome is evaluated. This thesis explains the development of the back-end services but the scope of this thesis does not cover the user interface which will be implemented in addition to the API.

The final result is that the system was deployed to production and is in use. New features are being designed and built all the time.

KEYWORDS:

System design, software development, JavaScript, software architecture

SISÄLTÖ

SANASTO

Domain Driven Design	Järjestelmän osien jakaminen ymmärrettäviin, loogisesti jaoteltuihin komponentteihin niiden käyttötarkoituksen perusteella.
Microservice	Arkkitehturaalinen tyyli rakentaa esimerkiksi palvelimet pieninä, eroteltuina ja itsenäisinä komponentteinaan monoliittipalvelimen sijasta.
Mobility As A Service	Liikkuminen palveluna. Liikkumisen tarpeet tarjotaan asiakkaalle kokonaisuutena, esim. yhdistämällä julkisen liikenteen palveluita muihin liikkumispalveluihin.
Ride-sharing	Kyydin jakaminen. Samaan ajoneuvoon voidaan yhdistellä muitakin asiakkaita, joka tekee kapasiteetin käyttämisestä tehokkaampaa ja mahdollitaa halvemman hinnan asiakkaille.

1 JOHDANTO

Opinnäytetyön tarkoituksena oli suunnitella ja toteuttaa kampanjakoodi- sekä ryhmätoiminnallisuuksia Kyyti Group Oy:n taustajärjestelmiin. Kyyti palvelee sekä yksityis- että yritysasiakkaita kehittämällään Mobility As A Service -alustalla. Yrityksellä on taustajärjestelmän lisäksi kaksi mobiilisovellusta, jotka hyödyntävät taustajärjestelmän rajapintoja lähes kaikissa toiminnallisuuksissaan. Taustajärjestelmään on suunniteltu sekä integroitu myös yrityksen ja sen yhteistyökumppanien tarjoama kutsupohjainen kyydinjakamispalvelu, joka toimii nimellä Kyyti.

Yrityksen mobiilisovellukset tarjoavat käyttäjilleen erilaisia liikkumispalveluita, kuten esimerkiksi reittioppaan ja pysäkkiaikataulut julkiselle liikenteelle, reittioppaan autoilijoille sekä mahdollisuuden hakea ja tilata Kyyti-palvelun matkoja. Yrityksen tavoitteena on helpottaa ihmisten päivittäistä matkustamista yhdistämällä eri liikkumismuotojen reititys sekä lippujen ostaminen samaan sovellukseen. Tavoitteena yrityksellä on myös kehittää matkaketjuja, jotka mahdollistaisivat joustavan reitittämisen ja useamman lipputuotteen ostamisen samanaikaisesti, esimerkiksi taksimatka juna-asemalle ja junalippu määränpäähän.

Opinnäytetyön aihe tuli yrityksen tarpeesta toteuttaa asiakasyrityksille mahdollisuus tunnistautua sovelluksessa asiakasyrityksen työntekijänä, minkä jälkeen työntekijä olisi oikeutettu ostamaan yrityksen työntekijöille suunnattuja lipputuotteita ja palveluita.

Yrityksellä oli myös tulevaisuudessa tarve tarjota asiakkailleen erilaisia alennuksia ja bonuksia mahdollistavia kampanjakoodeja. Ryhmiin liittyminen olisi myös tarkoitus hoitaa ennalta määritettyjen koodien avulla, joten toiminnallisuuksien toteutus voitiin joiltain osin yhdistää. Aiheen laajuuden vuoksi kampanjatoiminnallisuuksien toteuttaminen ja suunnittelu joudutaan kuitenkin tässä opinnäytetyössä suurimmaksi osaksi sivuuttamaan.

Opinnäytetyössä käytettiin konstruktiivista tutkimusmenetelmää, sillä tarkoituksena oli uusien toiminnallisuuksien suunnittelu ja implementointi. Järjestelmään oli suunniteltu ryhmätoiminnallisuuksia. Suunnitelmien jälkeen itse järjestelmä oli kuitenkin muuttunut niin paljon, että käytännössä toimeksianto oli aloitettava lähes puhtaalta pöydältä. Yrityksen taustajärjestelmät, teknologiat ja toimintatavat olivat kuitenkin tulleet jo suurilta osin tutuiksi, sillä ennen opinnäytetyöni toimeksiannon suorittamista olin työskennellyt

yrityksessä harjoittelijana. Harjoittelujakson jälkeen ja opinnäytetyön tekemisen aikana työskentelin yrityksessä täysipäiväisesti.

Ryhmätoiminnallisuudet toteutettiin yrityksen palvelimien taustajärjestelmiin. Käyttäjän ja taustajärjestelmien välissä on kuitenkin usein web- tai mobiilisovellus, joka tarjoaa asiakkaille graafisen käyttöliittymän toiminnallisuuksien hyödyntämiseksi. Omaan työtehtävääni tai toimenkuvaani projektissa ei kuulunut käyttöliittymän suunnittelu tai toteutus, joten tässä opinnäytetyössä keskitytään vain taustajärjestelmän teknologioihin, toiminnallisuuksiin ja arkkitehtuuriin. Käyttöliittymä rakennettiin taustajärjestelmän päälle sen valmistuttua.

Opinnäytetyö oli ensimmäinen suurempi järjestelmäkokonaisuus, jota olin alusta asti päässyt itsenäisesti suunnittelemaan ja toteuttamaan. Tavoitteena oli ensisijaisesti saada aikaan järjestelmä, joka täyttää yrityksen sekä asiakkaan tarpeet ja jota on helppo jatkokehittää. Tavoitteena oli myös lisätä omaa ymmärrystä ohjelmoinnista, sovelluskehityksen prosesseista sekä tietojärjestelmien suunnittelusta.

Web-teknologiat kehittyvät erittäin nopeasti, joten opinnäytetyön lähteinä on suurimmaksi osaksi käytetty internetistä löytyviä lähteitä. Internetistä saatava tieto oli mielestäni huomattavasti paremmin ajan tasalla kuin alaan liittyvä kirjallisuus. Tähän vaikuttaa osittain se, että yrityksen käyttämistä teknologioista kuten microserviceistä ei vielä ole olemassa minkään näköistä tarkkaa ohjeistusta, vaan jokainen yritys saattaa toteuttaa niitä omiin tarkoituksiinsa sopivilla tavoilla.

Opinnäytetyössä kerrotaan toimeksiannosta, järjestelmän taustatiedoista sekä käytössä olevista teknologioista sekä työkaluista. Kaikkia järjestelmän osia ja teknologioita ei voida opinnäytetyössä esitellä niiden salaisuuden sekä laajuuden vuoksi, joten vain oleellisimmat teknologiat ja järjestelmät käydään läpi. Toteutukseen keskittyvässä osiossa kerrotaan projektin valmistumisesta ja järjestelmän toiminnallisuuksista, sekä niihin liittyvistä haasteita. Lisäksi perustellaan projektin toteutuksien arkkitehtuurisia ratkaisuja ja valintoja. Opinnäytetyön lopuksi arvioidaan projektin lopputulos sekä pohditaan, miten järjestelmää tulisi tulevaisuudessa kehittää.

.

2 TEKNOLOGIAT

2.1 JavaScript-ohjelmointikieli

JavaScript on dynaamisesti ja heikosti tyypitetty korkean tason skriptikieli, joka on HTML- ja CSS-kielien ohella yksi kolmesta World Wide Webin tärkeimmistä teknologioista. JavaScriptiä käytetään suurimmaksi osaksi web-sovellusten dynaamisten ja interaktiivisten komponenttien luomiseen, mutta nykyisin sillä voidaan web-sovellusten lisäksi toteuttaa myös palvelin- ja työpöytäsovelluksia. (Mozilla Developer Network 2018a.)

JavaScriptin toiminnallisuus pohjautuu ECMAScript-standardiin. Standardi määrittelee yleisesti, miten skriptikielen tulisi toimia. JavaScript on tunnetuin standardia toteuttavista skriptikielistä, vaikka muitakin standardin toteuttavia skriptikieliä on olemassa. Nykyisin lähes kaikki modernit selaimet tukevat uusimpia standardin määrittelemiä ominaisuuksia. Standardista julkaistaan uusi versio pääsääntöisesti vuosittain. (Ecma International 2018.)

JavaScriptiä ei tule myöskään nimestään huolimatta sekoittaa keskenään ohjelmointikieli Javan kanssa, vaan JavaScriptiin on suurimmaksi osaksi vaikuttanut C-, Self- sekä Scheme-kielet. (Mozilla Developer Network 2018a.)

2.2 Node.js-palvelinympäristö

Tavallisesti selaimet suorittavat JavaScriptiä asiakaskoneella, mutta Node.js mahdollistaa myös palvelinpuolen ohjelmoinnin. JavaScriptiä ollaan siis ottamassa käyttöön web-sovellusten lisäksi kaikenlaisissa muissakin käyttötarkoituksissa.

Node.js toimii yhdellä prosessorin säikeellä, ja se pystyy ohjelman ajoa pysäyttämättömien I/O-kutsujensa ansiosta käsittelemään todella montaa samanaikaista yhteyttä sekä suorittamaan koodia asynkronisesti ja jopa paralleelisti. Tästä muodostuu suuri ero Node.js ja PHP-kielen välillä, sillä PHP vaatii edeltävän koodirivin suorittamista loppuun ennen ohjelman jatkamista, kun taas Node.js ympäristössä koodin suorittamisessa voidaan edetä sillä aikaa, kun esimerkiksi odotetaan HTTP-kutsun vastausta ulkoiselta palvelimelta. (Node.js Foundation 2018.)

Node.js perustuu Chromen V8 JavaScript-moottoriin, joka on kirjoitettu käyttäen C++-kieltä. Moottori kääntää kirjoitetun JavaScriptin natiiviksi konekieleksi ennen sen ajamista. (Node.js Foundation 2018.)

Node.js käyttää myös Node Package Manager (npm) -nimistä pakettienhallintarekisteriä, josta löytyy todella paljon valmiita kirjastoja erilaisten tehtävien suorittamiseen, joten pyörää ei tarvitse aina keksiä uudelleen. Valmiita moduuleita ja kirjastoja löytyy esimerkiksi vaikkapa tekstin lokalisointiin tai GPS-koordinaattien etäisyyden laskemiseen. (npm, Inc. 2018.)

2.3 Microservice-arkkitehtuuri

Aikaisemmin palvelimet oli tapana tehdä suurina ja yksittäisinä kokonaisuuksina, joita kutsutaan monoliiteiksi. Tällöin yksi palvelin saattaa toteuttaa yksin useita tai jopa kaikkia järjestelmän toimintoja. Monoliittipalvelin on jossain määrin helpommin ymmärrettävissä ja hallinnoitavissa kuin useampi yksittäinen palvelin, mutta monet yritykset kuten esimerkiksi Netflix ovat siirtyneet toteuttamaan niin kutsuttua microservice-arkkitehtuuria. (NGINX Inc. 2018.)

Microservice-arkkitehtuurin hyödyntäminen mahdollistaa ohjelmakokonaisuuden pilkkomisen ja abstraktoinnin yksittäisiksi ja itsenäisemmiksi komponenteiksi, joista jokaisella on oma rajattu tarkoituksensa. Tämä tarkoittaa, että esimerkiksi tuotteen hinnan laskemiseen ja luomiseen voi olla kaksi erilaista palvelinta jotka ovat yhteydessä toisiinsa; toinen on vastuussa hinnoittelusta ja toinen on vastuussa tuotteen tietojen lisäämisestä ja tallettamisesta tietokantaan. Microservice-arkkitehtuuri hyödyntää läheisesti Domain Driven Design (DDD) -periaatteita, jotka tarkoittavat toiminnallisuuksien erottelua eri paikkoihin niiden käyttötarkoituksen mukaan. Jokaisen arkkitehtuurin komponentin tulisi voida toimia järjestelmässä itsenäisesti, mutta ne voivat myös kommunikoida keskenään. Arkkitehtuuri mahdollistaa teoriassa myös sen, että jokainen microservice voisi olla teknologiselta toteutukseltaan erilainen eikä kaikkia järjestelmän osia tarvitse kirjoittaa edes samalla ohjelmointikielellä. (Fowler & Lewis 2014.)

Microservice-arkkitehtuurin hyödyntäminen tekee järjestelmän skaalautumisesta helpompaa. Kehitystiimin kaikkien jäsenten ei tarvitse tehdä töitä yhden palvelimen parissa

tai edes ymmärtää järjestelmää kokonaisuutena, vaan he voivat itsenäisesti kehittää lisätoiminnallisuuksia järjestelmään eri microserviceinä. Tämän lisäksi järjestelmän eri osia on helppo kehittää, päivittää tai jopa korvata kokonaan uusilla. (SmartBear Software 2018.)

Ajatus microserviceistä ei ole uusi, mutta vasta viimeisen muutaman vuoden aikana yritykset ovat alkaneet toden teolla käyttämään niitä. Tämän vuoksi jokaisella yrityksellä on myös hieman erilaisia tapoja toteuttaa microservice-arkkitehtuuria, eikä mitään kultaista keskitietä tai standardia toteuttamiseen vielä ole olemassa. (Chris Richardson 2017.)

2.4 JSON

JavaScript Object Notation (JSON) on tapa esittää dataa merkkijonona. JSON nimensä mukaisesti vastaa hyvin läheisesti JavaScript-objektien rakennetta. JSON ei kuitenkaan ole millään tavalla sidottu JavaScriptiin, vaan nykyisin hyvin monista ohjelmointikielistä löytyy työkaluja ja kirjastoja merkkijonojen lukemiseen tai niiden luomiseen. JSON-objekteja voi myös tallettaa helposti moderneihin tietokantoihin kuten PostgreSQL. (JSON.org 2018.)

JSONia käytetään useimmiten sovellusten väliseen kommunikointiin verkossa, sillä sen sisältämä data on ihmiselle helposti luettavassa, ymmärrettävässä sekä kirjoitettavassa muodossa. Data esitetään JavaScript-objektien tavoin avain-arvo-pareina tai taulukkotyylisenä tietotyyppinä. (Mozilla Developer Network 2018b.)

JSONia pidetään vähemmän työläänä lukea ja kirjoittaa kun vaihtoehtoisia datan esitystyyppejä kuten YAML tai XML. JSON ei kuitenkaan sisällä kuitenkaan kommentteja tai vahvempia tyypityksiä tai datatyyppejä.

2.5 REST-rajapinta

Representational State Transfer (REST) on arkkitehtuurinen tyyli rakentaa rajapintoja HTTP-kutsujen päälle. REST-rajapinnan avulla voidaan esimerkiksi hakea tai muokata palvelimella olevaa dataa ennalta määriteltyjen URL-osoitteiden ja parametrien sekä

HTTP-kutsun metodin perusteella. REST-arkkitehtuuri kuvaa palvelin-asiakas asetelmaa, jossa asiakasohjelman ei tarvitse tietää järjestelmän yksityiskohtia, vaan järjestelmästä tiedon hakeminen tai muokkaaminen tapahtuu ennalta määriteltyjen kutsujen ja kontrolloitujen transaktioiden avulla. HTTP-protokollan takia REST-arkkitehtuurin järjestelmät eivät ylläpidä omaa tilaa (state). Jos yksittäinen järjestelmäkutsu epäonnistuu, niin muutoksia järjestelmään ei pitäisi tapahtua vaan käyttäjä saa tiedon virheestä. Tilattomuutensa vuoksi järjestelmä on kaatuessa myös helposti palautettavissa. (Pivotal Software 2018.)

HTTP-kutsuilla on erilaisia metodeja. GET-kutsuilla haetaan dataa rajapinnasta, eikä niiden tulisi muokata rajapinnassa olevaa tietoa millään tavalla. Saman GET-kutsun suorittaminen useamman kerran peräkkäin pitäisi siis taata tismalleen sama tulos. (Stack Overflow 2018.)

POST-, PUT- ja DELETE-kutsuilla taas on tarkoitus tehdä erilaisia ja toimintoja rajapinnassa. POST- ja PUT-kutsuilla voidaan lisätä tai muokata dataa ja DELETE-kutsulla taas poistaa dataa. (Stack Overflow 2018.)

2.6 PostgreSQL

PostgreSQL on avoimen lähdekoodin olio-relaatiotietokantapalvelin, jonka tarkoituksena on varastoida tietoa sekä välittää sitä tarvittaessa tietokantaan kytkettyihin ohjelmistoihin ja palveluihin. (PostgreSQL Development Group 2018.)

PostgreSQL on tullut tunnetummaksi pyrkimällä noudattamaan erilaisia standardeja, sekä tukemalla monipuolisia tietotyyppejä. PostgreSQL noudattaa hyvin ACID-mallia (Atomicity, Consistency, Isolation, Durability), jolla tarkoitetaan tietokantatransaktioiden ja toimintojen toimivuutta ja ennalta-arvattavuutta virheistä tai järjestelmän kaatumisesta huolimatta. (PostgreSQL Tutorial 2018.)

Muihin tietokantoihin verrattuna PostgreSQL on transaktioissaan hieman hitaampi, mutta sitä pidetään todella vakaana tietokantajärjestelmänä. PostgreSQL sisältää myös monia erilaisia tietotyyppejä ja mahdollisuuden kirjoittaa omia funktioita ja jopa lisäosia tietokantaan, joten sen toiminnallisuutta pystyy tarvittaessa laajentamaan helposti. (PostgreSQL Tutorial 2018.)

3 TOIMEKSIANNON KUVAUS

Yrityksen mobiilisovelluksen käyttämiin taustajärjestelmän palvelimiin tuli suunnitella, toteuttaa ja integroida ryhmä- sekä kampanjatoiminnallisuuksia. Toiminnallisuudet tulisi suunnitella siten, että niitä olisi tarvittaessa helppo jatkokehittää sekä laajentaa tarvittaessa järjestelmän skaalautuessa. Taustajärjestelmän käyttämä microservice-arkkitehtuuri helpottaa skaalautumista sekä uusien ominaisuuksien sekä palveluiden implementointia ja integrointia.

Ryhmätoiminnallisuuksien vaatimuksena oli, että käyttäjän on mahdollista liittyä ryhmään syöttämällä mobiilisovellukseen jokin ennalta määrätty koodi. Ryhmäjäsenyys tulisi myös koodin käyttämisen jälkeen vahvistaa joko automaattisesti sähköpostin verkkotunnuksen perusteella ja/tai ryhmäjäsenyyteen vaadittujen lisätietojen antamisen jälkeen. Ryhmätoiminnallisuuksien hallinnointiin tarvittiin myös rajapinnat, joiden tulisi mahdollistaa tiettyyn ryhmään kuuluvien käyttäjien listaaminen tai yksittäisen käyttäjän poistaminen ryhmästä.

Tulevaisuudessa kampanjakoodien lunastaminen tulisi ryhmään liittymisen tavoin toimimaan syöttämällä sovellukseen koodi, jonka jälkeen käyttäjälle talletetaan jonkin ennalta määritellyn kampanjan mukainen bonus tililleen. Tämä tulee ottaa huomioon myös suunnitellessa ryhmäkoodien toiminnallisuutta, sillä koodeille pitää voida tämän myötä määrittää tyyppi, joka kertoo mitä koodilla on tarkoitus tehdä.

Projektin ajallisten rajoitteiden takia emme ehtineet tekemään kovin tarkkoja suunnitelmia tai kuvauksia järjestelmästä, joten projekti muovautui hyvin paljon ajan kanssa. Ajallisten rajoitteiden lisäksi yrityksen sovelluskehitys tapahtui ketterin menetelmin noin kaksi kertaa viikossa pidettävien palaverien siivittämänä, joissa toiminnallisuuksia iteroitiin ja suunniteltiin muiden sovelluskehittäjien kanssa sekä varmistettiin, että kaikki sillä hetkellä kehityksessä olevat toiminnallisuudet pysyisivät helposti ylläpidettävinä ja jatkokehitettävinä.

4 JÄRJESTELMÄ JA TYÖKALUT

4.1 Yrityksen taustajärjestelmä

Olemassa oleva taustajärjestelmä on rakennettu suurimmaksi osaksi yrityksen oman microservice-arkkitehtuurin ja sen käyttöä helpottavien yksityisten npm-pakettien avulla**.** Microservicet kommunikoivat keskenään omien sisäisten rajapintojensa Remote Procedure Callien (RPC) avulla.

Mobiilisovelluksen ja varsinaisten microserviceiden välisen kommunikoinnin mahdollistaa niiden välissä toimiva gateway eli yhdyskäytävä, joka ottaa vastaan sovelluksen lähettämät HTTP-pyynnöt ja toimittaa niistä olennaisen tiedon oikealle microservicelle käsiteltäväksi. Palvelin taas vastaa kutsuun sovellukselle gatewayn kautta. Microservicet pyörivät Google Cloud Platform -palvelun palvelimilla pakattuna erillisiin Docker-containereihin, joiden hallinta tapahtuu avoimen lähdekoodin Kubernetes-järjestelmällä. Microserviceiden ajaminen containereiden sisällä mahdollistaa koodin toiminnan samalla tavalla kaikissa palvelinympäristöissä. (Docker Inc. 2018.)

Järjestelmään on ehditty kehittää suuri määrä erilaisia toiminnallisuuksia mahdollistavia microserviceitä, mutta olennaisimpana toimeksiantoon liittyvänä järjestelmänä on **users**, jonka tarkoituksena on mahdollistaa kaikki suoraan käyttäjään liittyvät toiminnot, kuten esimerkiksi käyttäjän profiilin hakeminen tai päivittäminen. Käyttäjien ryhmiin liittyvät tietokannat voidaan luoda ja integroida valmiiseen microserviceen.

Microserviceillä on omat tietokantansa, joihin ne tallettavat ja joista ne hakevat toimiinsa tarvittavaa tietoa. Tätä tietoa voidaan esimerkiksi prosessoida tai lähettää sellaisenaan toiselle järjestelmäkomponentille tai yhdyskäytävän kautta rajapinnan käyttäjälle.

Käyttäjän tunnistaminen järjestelmässä tapahtuu käyttämällä JSON Web Tokeneita (JWT). Kirjautuessa käyttäjän istunnon tunniste tallennetaan järjestelmään ja käyttäjätieto kulkee microserviceiden välillä liikkuvien kutsujen sisällä. Tämän tiedon perusteella voidaan tarkistaa mitä toimintoja ja rajapintakutsuja käyttäjän annetaan käyttää.

4.2 Kehityksessä käytetyt työkalut

4.2.1 Git-versionhallintajärjestelmä

Git on erittäin suosittu avoimen lähdekoodin hajautettu versionhallintajärjestelmä, joka mahdollistaa projektin tilan tallentamisen ja seurannan sekä monen sovelluskehittäjän samanaikaisen työskentelyn projektin parissa. Järjestelmä pitää siis hyvää huolta siitä, että ristiriitoja ja eroavaisuuksia projektin rakenteessa ei syntyisi eri sovelluskehittäjien välille. (Git 2018.)

Jokaisella Git-projektilla on oma säilönsä eli **repository** projektin sisältämälle datalle. Säilö voidaan kopioida palvelimelta sovelluskehittäjän tietokoneelle, jonka jälkeen hän voi tehdä projektiin haluamiaan muutoksia ja siirtää muutokset taas palvelimelle talteen.

Gitiä voi asennuksen jälkeen käyttää joko komentoriviltä tai siihen tarkoitetulla erikseen asennettavalla graafisella käyttöliittymällä. Toimeksiannon yritys käyttää yhteisenä keskitettynä säilönä GitHubia.

4.2.2 WebStorm-kehitysympäristö

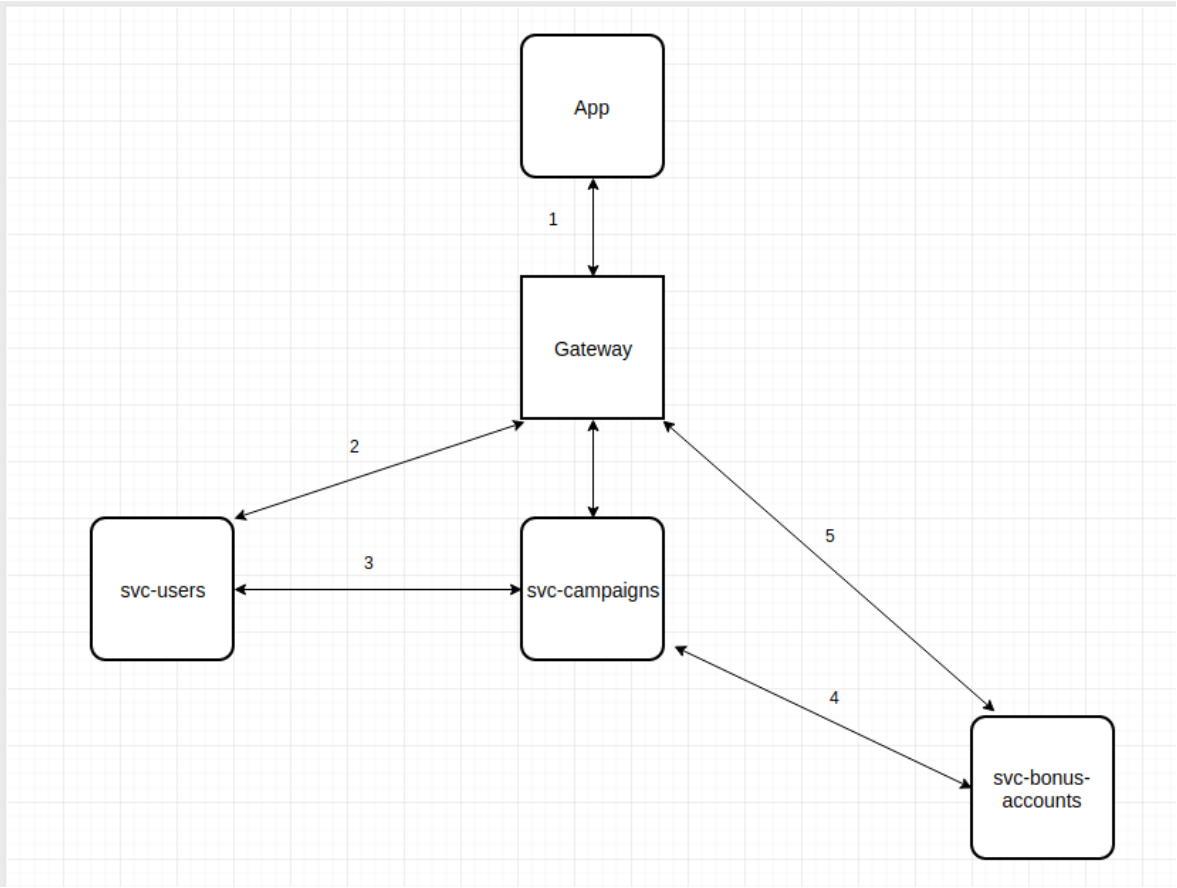
Webstorm on JetBrainsin kehittämä Integrated Development Environment (IDE) websovellusten kehittämiseen JavaScriptillä. Tekstieditorin lisäksi se tarjoaa erilaisia työkaluja helpottamaan sovelluskehitystä, joista esimerkkinä integroitu terminaali sekä debugger-työkalu. Näiden lisäksi sovelluksesta löytyy todella paljon asetuksia millä itse tekstieditoria sekä sen toiminnallisuuksia pystyy muokkaamaan. (JetBrains 2018.)

4.2.3 Postman

Postman on työkalu web-rajapintojen suunnitteluun ja testaamiseen. Sillä voi tallentaa ja lähettää erilaisia HTTP-pyyntöjä rajapintoihin ja tarkastella niiden vastauksia. Tämän lisäksi sovelluksella voi helposti tehdä erilaisia automaattisia monitoreita ja testejä rajapinnan toimivuuden valvomiseksi. (Postdot Technologies, Inc. 2018.)

5 TOTEUTUS

Ennen toimeksiannon aloittamista tarvittiin vain pieni ja yksinkertaistettu kuvaus uusista järjestelmäkomponenteista ja niiden sijoittamisesta järjestelmään. Järjestelmään tultaisiin käytännössä tarvitsemaan aluksi vain uusi campaigns-microservice.



Kuva 1. Yksinkertaistettu kuvaus uusien toiminnallisuuksien sijoittamisesta järjestelmään.

Kuvan 1 numeroidut vuorovaikutukset selitettynä:

1. Yrityksen mobiiliapplikaatio on yhteydessä yhdyskäytävään, joka välittää käyttäjän pyynnön sekä istunnon tiedot taustajärjestelmille. Taustajärjestelmät eivät myöskään ole suoraan yhteydessä sovellukseen, vaan vastaukset sovellukselle tulevat yhdyskäytävän kautta.

2. Yhdyskäytävä on yhteydessä users-serviceen ja hakee sieltä käyttäjän tiedot, jotka välitetään tarvittaessa kutsuissa eteenpäin muille microserviceille. Tiedot tulevat pitämään sisällään integraation jälkeen myös tiedot käyttäjän ryhmistä.

3. Ryhmiin liittyminen tulee tapahtumaan uudella microservicellä, campaignsilla, joka saa yhdyskäytävältä käyttäjän sovelluksesta lähetetyn koodin. Koodin tarkistamisen jälkeen tieto välitetään eteenpäin users-servicelle, jossa käyttäjä liitetään ryhmään.

4. Kampanjakoodien lunastaminen tapahtuu myös campaigns-servicen kautta. Tässä tapauksessa campaigns-service lähettää käytetyn koodin tiedot toiselle servicelle, bonus-accountsille, jolloin bonus-serevice lisää käyttäjälle koodiin sidotun bonuksen tai edun.

5. Microservicet tarjoavat tarvittavat Create Read Update Delete (CRUD) -toiminnot yhdyskäytävän kautta rajapinnan käyttäjälle.

Tämän yksinkertaistetun järjestelmän osien ja vuorovaikutusten kuvaamisen jälkeen voidaan uudet järjestelmäkomponentit rakentaa ja integroida olemassaolevaan taustajärjestelmään. Ensin toteutetaan koodin syöttäminen sekä ryhmätoiminnallisuus uuden campaigns-servicen avulla sekä integroidaan se users-serviceen.

5.1 **Microservice kampanjakoodeille**

Aluksi luodaan uusi microservice, jonka tarkoituksena on pitää kirjaa sekä hallinnoida erilaisia toiminnallisuuksia mahdollistavia koodeja. Palvelinta ei tarvitse lähteä ohjelmoimaan ja konfiguroimaan tyhjästä, vaan yrityksellä on käytössään **service-template,** joka voidaan kopioida uuden microservicen pohjaksi.

Kehitystyössä ei kuitenkaan käytännössä tarvitse koskea kuin src-hakemiston sisältöön, joka sisältää sovelluksen varsinaisen lähdekoodin. Tietokantamuutosten hallinnointiin yrityksessä käytetään työkalua, mutta toteutuksen kuvauksessa kuvataan vain tarvittavat komennot tietokantataulujen luomiselle eikä työkaluun liittyviä komentoja tai konfiguraatiota.

Kuvassa 2 nähdään esimerkki uuden microservicen kansiorakenteesta. Service-template pitää sisällään uuden microservicen palvelimen, tietokantayhteyden sekä monitoroinnin peruskonfiguraatiot.

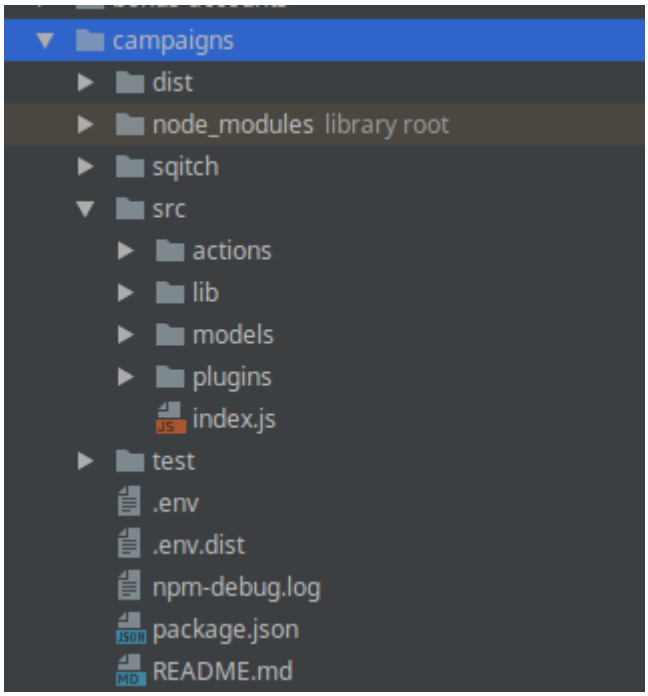
/src/actions pitää sisällään tietokantaan tai HTTP-kutsuihin liittyviä moduuleita.

/src/lib-hakemistoon voidaan tallettaa muita apufunktioita- ja moduuleita, niin sanottuja kirjastofunktioita.

/src/models sisältää tietokannan Object-Relational Mapping (ORM)-mallit. Nämä mallit mahdollistavat tietokannan ja koodin välisen informaation rakenteen yhteneväisyyden sekä datan validointia. Näiden mallien rakennetta ei opinnäytetyössä selvennetä sen enempää, sillä ne muistuttavat hyvin paljon tietokantojen luomisessa käytettyjä komentoja.

/src/plugins sisältää microservicen omat toiminnot sekä rajapinnat.

/index.js on palvelimen käynnistyspiste, joka sisältää microservicen käynnistämiseen, konfiguraatioiden lukemiseen sekä tietokantayhteyksien luomiseen liittyvän koodin.



Kuva 2. Servicen kansiorakenne.

5.2 Koodin syöttämisen rajapintakuvaus

Ennen varsinaisen toiminnallisuuden ohjelmointia voidaan tehdä valmiiksi yhdyskäytävän rajapintakuvaus tarvittaville toiminnoille.

Yhdyskäytävään lisätään koodin 1 mukainen moduuli ja otetaan se käyttöön. Moduulissa on konfiguraatio uudelle POST-reitille, jota mobiiliapplikaatio kutsuu käyttäessään ryhmä- tai kampanjakoodin.

Rajapinnan konfiguraatiossa käyttäjän tunnistaminen asetetaan pakolliseksi, lisätään rajapinnalle tunnisteet (tags), kuvaus (description) sekä määritellään POST-pyynnön osoite ja tarvittavat parametrit.

Rajapinta tarvitsee koodin käyttämisen mahdollistavassa kutsussa koodin käyttäneen käyttäjän tunniste (userId) sekä koodi, jonka käyttäjä haluaa käyttää (campaignCode).

```
import Joi from '../../lib/joi';

export default {
  '/code/use': {
    POST: {
      action: 'role:campaigns, code:use',
      config: {
        auth: {
          required: true
        },
        description: 'Use campaign codes and join groups',
        tags: ['api', 'campaign'],
        validate: {
          payload: Joi.object({
            userId: Joi.string()
              .allow([null, 'me'])
              .guid({version: 'uuidv4'}),
            campaignCode: Joi.string().required()
          })
        }
      }
    }
  }
};
```

Koodi 1. Yhdyskäytävän moduuli uudelle rajapinnalle.

Yhdyskäytävä validoi käyttäjän syöttämän tiedon kirjastolla nimeltä Joi, jonka avulla voidaan määritellä tarkemmin muun muassa eri parametrien tyypit JSON-skeeman avulla.

User ID on käyttäjän identifiointiin tarkoitettu uniikki UUID-merkkijono. Käyttäjä voi kuitenkin hankalan merkkijonon sijasta antaa rajapinnalle tässä parametrissä sanan ”me”, jonka avulla yhdyskäytävä osaa automaattisesti asettaa käyttäjän vaikealukuisen tunnisteen automaattisesti seuraaviin kutsuihin.

Koodin perusteella yhdyskäytävä pystyy lisäosan ansioista generoimaan kuvan 2 kaltaisen Swagger-dokumentaation rajapinnasta. Dokumentaation voi kirjoittaa myös erikseen käyttämällä esimerkiksi YAMLia, mutta dokumentaation automaattinen generointi käytettävästä koodista on erittäin toimiva ratkaisu ja vähentää työtä dokumentoinnin osalta. Dokumentaatio pysyy näin myös paremmin ajan tasalla.

Kuva 2. Swagger-rajapintakuvaus koodin käyttämiselle.

5.3 Tietokantataulujen luominen koodeille

Rajapintakuvauksen ja koodin käyttämiseen vaadittavan yhdyskäytävän HTTP-reitin luomisen jälkeen voidaan tietokantaan tehdä taulu erilaisille koodeille. Sekä ryhmiin liittyminen, että kampanjakoodien lunastaminen tapahtuu koodien avulla, joten niiden koodit voidaan yhdistää samaan tauluun, kun koodit erotellaan toisistaan jollain parametrilla toisessa kolumnissa. Näin koodeihin voidaan lisätä tulevaisuudessa lisää toiminnallisuuksia, mutta silti koodit on talletettu keskitetysti yhden microservicen hallinnoimaksi. Muualla tämän taulun koodeihin referoidaan sen tietokantarivin **id**-tunnisteen perusteella.

Tietokannan tauluissa on pääsääntöisesti aina pääavaimeksi asetettu uniikki **id** sekä **created_at** ja **updated_at** kentät. Tämän lisäksi tietokannan pitää luonnollisesti sisältää itse koodi (**code)**, jonka käyttäjä syöttää sovellukseen sekä koodille asetettava tyyppi **(type)**, jotta tiedetään, onko koodilla tarkoitus lunastaa kampanja vai liittää käyttäjä ryhmään. Koodin tyyppi voi näin aluksi olla siis joko **group** tai **campaign**.

Tietokantataulu voidaan luoda koodin 2 mukaista SQL-lauseketta käyttäen.

```
CREATE TABLE campaigns.code (
  id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  code TEXT NOT NULL,
  type TEXT NOT NULL,
  created_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(),
  updated_at TIMESTAMP WITH TIME ZONE
);
ALTER TABLE campaigns.code OWNER TO campaigns;
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE campaigns.code TO cam-paigns_rw;
```

Koodi 2. Tietokannan taulun luomiseen vaadittava SQL-lauseke.

Tietokantaa ja tauluja luodessa on otettava huomioon järjestelmän muiden komponenttien käyttämä tietokantarakenne ja pääsynhallinta. Järjestelmän komponenttien tietokannat on järjestelty niin, että jokaisella tietokantaa tarvitsevalla microservicellä on pääsääntöisesti omat skeemansa sekä käyttäjätilinsä skeemojen ja taulujen hallinnointia varten. Taulu kampanjakoodeille luodaan tietokantaan campaigns-skeeman alle, jonka omistusoikeudet annetaan **campaigns**-nimiselle käyttäjälle ja kirjoitus- ja lukuoikeudet **campaigns_rw**-käyttäjälle.

Useamman tietokantakäyttäjän luomisen syynä on niin sanottu ”Principle of Least Privilege”. Tämä tarkoittaa sitä, että järjestelmän osilla on suora pääsy vain ja ainoastaan siihen informaatioon, jota ne toimintoihinsa tarvitsevat. (Beyond Trust 2018)

Oikeuksien jakaminen useammalle eri käyttäjälle tekee tietokannasta hieman vaikeammin hallinnoitavan, mutta turvallisemman. Yksittäisellä microservicellä ei tarvitse olla suoraa pääsyä muiden microserviceiden dataan, vaan ne voivat tarvittaessaan pyytää sitä sisäisten rajapintojen kautta valmiiksi määritetyillä tietokantakutsuilla. Jos hyökkääjä pääsisi käsiksi campaigns-serviceen tai sen tietokantakäyttäjiin, ei hän kuitenkaan siis pystyisi lukemaan esimerkiksi käyttäjien tunnuksia tai salasanoja tietokannan toisesta taulusta.

5.4 Koodin käyttäminen campaigns-servicessä

Campaigns-servicen toiminnallisuutta voidaan luoda uuteen moduuliin polussa **src/plugins/campaigns.js**. Tiedoston ja toiminnallisuuden implementoinnin jälkeen moduuli pitää vielä ottaa käyttöön **src/plugins/index.js** -tiedostossa.

Aikaisemmin yhdyskäytävälle tehtiin uusi HTTP-reitti, jonka **action**-parametrin arvoksi asetettiin merkkijono **’role: campaigns, code:use’**. Merkkijono toimii ikään kuin sisäisen rajapinnan toiminnon nimenä ja pattern-matchingin avulla yhdyskäytävä osaa kutsua oikeaa toimintoa.

Microservicelle lisätään toiminto kutsumalla **seneca.addSecureAsync** -funktiota, jonka kautta voidaan asettaa toiminnolle ”nimi” sekä funktio, joka suoritetaan kun toiminnallisuutta halutaan käyttää. Toimintoa voidaan toisessa microservicessä kutsua **seneca.actAsync** -funktiolla.

```
export default function () {
 const seneca = this;
  const pluginName = 'servicePlugin';
  seneca.addSecureAsync(
    'role:campaigns, code:use',
    async ({_credentials, userId, campaignCode: code}) => {
        const codeData = await getCode({code: code.trim()});
        return seneca.actAsync(
          `role:campaigns, entity:code, type:${codeData.type}, cmd:use`,
          {
            _credentials,
            userId,
            codeId: codeData.id
          }
        );
    }
  );
  return {pluginName};
```

Koodi 4. Campaigns-moduulin sisältö.

Yhdyskäytävältä toiminnolle tulee siis parametrina **_credentials**, joka sisältää käyttäjän istunnon tunnistetiedot. Käyttäjä, jolle koodi lunastetaan, tunnistetaan **userId**-parametrin perusteella ja **campaignCode** sisältää käytetyn koodin.

Tämän jälkeen koodin tiedot haetaan aikaisemmin luomastamme tietokannan taulusta ja talletetaan **codeData**-muuttujaan. Koodin tyypin perusteella käyttäjän ja koodin tiedot ohjataan edelleen seuraavalle toiminnolle kutsumalla **actAsync**-funktiota sisäisessä rajapinnassa, jonka sisältö nähdään koodissa 5.

```
seneca.addSecureAsync(
  'role:campaigns, entity:code, type:group, cmd:use',
   async ({_credentials, userId, codeId}) => {
     svcUtilsAuth.requireUser(_credentials, {id: userId});
     const {groupId} = await getGroupCode({codeId});
     return seneca.actAsync('role:groups, entity:user, type:code, cmd:join', {
       _credentials,
       userId,
       groupId
     });
   }
);
```

Koodi 5. Campaigns-servicen ryhmätoiminnallisuus.

5.5 Tietokantataulujen luominen ryhmille

Tietokantataulut ryhmille voidaan luoda samalla tavalla kuin muutkin tietokantataulut. Ryhmille ei kuitenkaan tarvitse tehdä uutta microserviceä, vaan toiminnallisuus ja oikeudet tietokantaan voidaan luovuttaa users-servicelle. Periaatteessa groups-service voisi olla omakin microservicensä, mutta niin tarkka DDD-jaottelu ei välttämättä ole tarpeellista.

Ryhmiin liittyminen tapahtuu koodilla, joten koodin **code_id** tulee tallettaa ryhmän tietokantatauluun. Taulut ovat kuitenkin eri tietokantaskeemassa ja eri käyttäjien hallinnoima, joten suoraa relaatiota koodien tietokantatauluun ei ole mahdollista tehdä.

Ryhmätoiminnallisuuksien vaatimuksena on myös, että ryhmän täysivaltaiselle jäsenyydelle voidaan asettaa erilaisia rajoitteita. Ryhmäjäsenyyden vahvistamiseksi voidaan esimerkiksi kysyä käyttäjältä vaadittuja lisätietoja ja/tai tarkistaa käyttäjän sähköpostiosoitteen verkko-osoite. Tämän takia taululle lisätään ominaisuudet **required_user_details** sekä **email_domains**.

Ryhmien taulun lisäksi tarvitaan vielä taulu, johon voidaan tallettaa käyttäjien ryhmäjäsenyydet. Tämä onnistuu yksinkertaisesti koodin 6 mukaisella tavalla; tauluun ei tarvitse tallettaa kuin käyttäjän tunniste **user_id**, ryhmän tunniste **group_id** sekä ryhmäjäsenyyden tila **status**.

```
CREATE TABLE groups.membership (
 id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
  user_id UUID NOT NULL,
  group_id UUID NOT NULL REFERENCES groups.group(id),
  status CITEXT NOT NULL,
  created_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(),
  updated_at TIMESTAMP WITH TIME ZONE
);
```

Koodi 6. Taulu ryhmäjäsenyyksien tallettamiseen.

5.6 Ryhmään liittyminen

Taulujen luomisen jälkeen luodaan varsinainen toiminnallisuus ryhmään liittymiselle. Tämä tapahtuu yksinkertaisesti luomalla users-serviceen uusi sisäisen rajapinnan toiminto, joka kutsuu koodin 7 mukaista funktiota.

Funktion parametrit voidaan syöttää funktiolle **params**-objektissa, joka pitää sisällään ryhmäjäsenyyteen vaaditut **user_id, group_id** sekä **status** -tiedot. Nämä syötetään tietokantaan uudelle riville. Käyttäjän ryhmäjäsenyyden tila on aluksi **pending**. Käyttäjälle siis palautetaan tieto **groupJoined** sekä **status**, joka kertoo ryhmäjäsenyyden tilan.

Lisäksi ryhmän tiedoissa tulee tieto vaadittavista lisätiedoista tai vaadittavasta sähköpostiosoitteen verkko-osoitteesta. Tämän vastauksen perusteella sovellus osaa tarvittaessa kysyä käyttäjältä lisätietoja, yritysasiakkaan tapauksessa esimerkiksi työntekijänumeroa.

```
export async function joinGroup(params) {
const joinedGroup = await GroupMembership.query()
    .omit(['groupId'])
    .eager('group')
    .insert(params)
    .catch(databaseErrorHandler);
  joinedGroup.group.status = joinedGroup.status;
  joinedGroup.status = 'groupJoined';
  return joinedGroup.toJSON();
}
```

Koodi 7. Ryhmäjäsenyyden syöttäminen tietokantaan.

Tietokannan ja koodin välissä toimii **src/models** hakemistossa tietokannan ORM-malli, joka mahdollistaa käytetyn syntaksin tietokantakyselyiden suorittamiselle. ORM-mallit myös validoivat dataa samoin kuin tyypitetty tietokantakin. Koodissa näkyvä query-syntaksi perustuu tunnettuun **knex.js**-kirjastoon.

Nyt ryhmään voidaan siis periaatteessa jo liittyä koodia käyttämällä. Tämä tapahtuu lähettämällä POST-kutsu parametreineen ulkoisen rajapinnan osoitteeseen, josta saadaan vastaukseksi seuraava JSON-merkkijono:

```
{
        ”status”: ”groupJoined”,
        ”group”: {<Ryhmän tiedot>}
}
```

Ryhmäjäsenyyden tila voidaan tarkistaa tietokannasta. Valitaan tietokantarivit **groups.membership**-taulusta:

```
-[ RECORD 1 ]-------------------------------------
id         | 84b6ac90-5e0d-11e8-9c2d-fa7ae01bbebc
user_id    | 84b6af74-5e0d-11e8-9c2d-fa7ae01bbebc
group_id   | 84b6b1d6-5e0d-11e8-9c2d-fa7ae01bbebc
status     | pending
created_at | 2018-03-05 19:16:48.338213+03
updated_at | 2018-03-05 19:16:48.950751+03
```

5.7 Lisätietojen syöttäminen ja jäsenyyden aktivoiminen

Jos kaikki ehdot eivät vielä ole täyttyneet, niin ryhmäjäsenyys on tilassa **pending** ja se pitää aktivoida. Ryhmän tietokantataulussa **additional_details** sisältää vaatimuksen lisätiedosta **employeeId**. Tämä tarkoittaa, että käyttäjän on syötettävä vielä työntekijänumeronsa ryhmäjäsenyyden aktivoimiseksi. Tähän täytyy taustalle luoda tietokantataulu ja toiminnallisuus. Tietokantataulu voidaan luoda koodin 8 mukailemalla tavalla.

```
CREATE TABLE users.group_info (
 id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
   user_id UUID NOT NULL REFERENCES users.user_account(id) ON UPDATE CASCADE ON DELETE CASCADE,
   group_id UUID NOT NULL,
   additional_details JSONB NOT NULL,
   created_at TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(),
   updated_at TIMESTAMP WITH TIME ZONE NULL
);
ALTER TABLE users.group_info OWNER TO users;
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE users.group_info TO us-ers_rw;
CREATE UNIQUE INDEX ON users.group_info(user_id, group_id);
```

Koodi 8. Lisätietotaulu käyttäjälle

Taulu liittää käyttäjän ja ryhmän tunnisteet **additional_details**-kolumniin, jonka tyyppi on JSONB. Tämä on PostgreSQL kannan tietotyyppi, johon voi tallettaa JSON-objekteja. Kolumni tulee pitämään sisällään key-value -pareilla käyttäjän ryhmään liittyvät lisätiedot.

Lisätietojen syöttämisen jälkeen ryhmäjäsenyyden tila tarkistetaan uudelleen. Jos kaikki vaadittavat tiedot on täytetty sekä sähköpostiosoitteen verkko-osoite täsmää, voidaan käyttäjän ryhmäjäsenyyden tila asettaa aktiiviseksi kutsumalla koodin 9 funktiota käyttäjän ja ryhmän id-tunnisteella sekä **status** arvolla ”**active**”. Funktio on tarkoituksella monikäyttöinen, eikä sitä voida ulkoisesta rajapinnasta kutsua suoraan.

```
 export async function setMembershipStatus({userId, groupId, status}) {
return GroupMembership.query()
    .patch({status})
    .where('user_id', userId)
    .andWhere('group_id', groupId)
    .returning('*')
    .catch(databaseErrorHandler);
```

Koodi 9. Käyttäjän ryhmäjäsenyyden statuksen päivittäminen tietokantaan.

5.8 Ryhmän näyttäminen käyttäjän profiilissa

Ryhmän pitäisi vielä tulla näkyviin käyttäjän hakiessa profiiliaan. Tämä tarkoittaa luotujen toiminnallisuuksien integroimista vanhaan järjestelmään. Ryhmän tiedot lisätään käyttäjän profiiliin users-servicen **get user** -rajapintafunktioon koodin 10 kaltaisen pätkän avulla.

```
// Fetch user memberships
const userGroupMemberships = await listUserMemberships({userId: id});
// Fetch user group info
const userGroupInfos = await listUserGroupInfos({userId: id});
user.groups = userGroupMemberships.map(membership => {
  const group = membership.group;
  group.membershipStatus = membership.status;
  for (const info of userGroupInfos) {
    if (membership.groupId === info.groupId) {
      group.additionalDetails = info.additionalDetails || null;
    }
  }
  return group;
});
```

Koodi 10. Ryhmän lisääminen käyttäjän profiiliin.

Koodi hakee käyttäjän ryhmäjäsenyyksien tiedot sekä käyttäjän asettamat lisätiedot sekä lisää nämä käyttäjäprofiiliin oikein parsittuna niin, että ryhmäjäsenyys sekä lisätiedot liitetään toisiinsa **group id** -parametrin perusteella. Näin ryhmäjäsenyys saadaan näkyviin käyttäjän profiilissa.

Tämän jälkeen ryhmäjäsenyyden perusteella voidaan luoda lähes mitä tahansa lisäpalveluita, kun ryhmäjäsenyys voidaan tarkistaa käyttäjän pyynnöissä liikkuvien istunnon tietojen perusteella. Tarvitsee siis vain hakea käyttäjän profiili ja tarkistaa kuuluuko käyttäjä vaadittuun ryhmään, sekä onko ryhmäjäsenyyden status aktiivinen.

6 YHTEENVETO

Opinnäytetyön ja toimeksiannon tavoitteena oli toteuttaa ja integroida ryhmätoiminnallisuuksia yrityksen järjestelmään. Valmistunut järjestelmä täytti sille asetetut vaatimukset ja järjestelmä otettiin käyttöön tuotantopalvelimella asti. Toiminnallisuutta käyttävät yritys itse sekä yrityksen yritysasiakkaat.

Järjestelmää kehitetään jatkuvasti. Tulevaisuudessa erilaisten rajoitteiden, esimerkiksi tuotteiden ja maksutapojen rajoittaminen ryhmäkohtaisesti tulisi olla mahdollista. Myös muita uusia toiminnallisia vaatimuksia ryhmille on alettu miettimään ja suunnittelemaan, mutta niiden lisääminen jälkikäteen pitäisi kuitenkin olla todella helppoa; tarvitsee vain tarkistaa käyttäjäprofiilista käyttäjän ryhmät sekä ryhmäjäsenyyden tila.

Projektin valmistumisessa tuli melko kiire, joten koodin tyyliä ja ymmärrettävyyttä voisi parantaa jonkin verran. Lisäksi rajapintoja ryhmien ja koodien hallinnointiin tarvitaan, mutta niitä ei tässä opinnäytetyössä ehditty toteuttamaan. Nämä ovat myös tulevien ryhmätoiminnallisuuksien suunnitelmissa. Ryhmiä ei siis vielä voi esimerkiksi luoda rajapinnan kautta, vaan ne pitää syöttää tietokantaan manuaalisesti SQL-lausekkeiden avulla.

Yrityksessä ja toimeksiannossa käytetyt teknologiat kehittyvät jatkuvasti, eikä kaikkien niiden käytöstä ole olemassa tarkkaa ohjeistusta. Microservice-arkkitehtuuri mahdollistaa kuitenkin joustavan sovelluskehityksen ja uusien toiminnallisuuksien lisäämisen ketterästi. Järjestelmässä on monia abstraktiotasoja jotka mahdollistavat eri järjestelmäkomponenttien kuten tietokannan tai kokonaisen microservicen vaihtamisen lennosta toiseen ilman käyttökatkoksia.

Suunnitelmien ja vaatimuksien dokumentointiin ei käytetty paljoa aikaa projektityöskentelyssä käytettävien ketterien menetelmien vuoksi. Käytännössä järjestelmä kehittyi siis pikkuhiljaa iteroimalla ja vaatimuksia tarkentamalla. Valmis järjestelmä saatiin kuitenkin otettua käyttöön ilman sen suurempia ongelmia.

LÄHTEET

Beyond Trust, inc. What is least privilege? Viitattu 07.05.2018 https://www.beyondtrust.com/blog/what-is-least-privilege/

Chris Richardson. What are microservices? Viitattu 22.05.2018 http://microservices.io/

Docker Inc. 2018. What is a Container. Viitattu 22.05.2018 https://www.docker.com/what-container

Ecma International 2018. What is Ecma. Viitattu 24.05.2018 https://www.ecma-international.org/

Fowler & Lewis 2014. Microservices. Viitattu 03.05.2018 https://martinfowler.com/articles/microservices.html

Git 2018. About Git. Viitattu 04.05.2018 https://git-scm.com/

JetBrains 2018. WebStorm. Viitattu 04.05.2018 https://www.jetbrains.com/webstorm/

JSON.org 2018. Introducing JSON. Viitattu 04.05.2018 https://www.json.org/

Mozilla Developer Network 2018a. JavaScript. Viitattu 03.05.2018 https://developer.mozilla.org/bm/docs/Web/JavaScript

Mozilla Developer Network 2018b. JSON. Viitattu 04.05.2018 https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON

NGINX Inc. Microservices at Netflix: Lessons for architectural design. Viitattu 22.05.2018 https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

Node.js Foundation 2018. About Node.js. Viitattu 03.05.2018 https://nodejs.org/en/about/

Npm, Inc. 2018. About npm. Viitattu 03.05.2018 https://www.npmjs.com/about

Pivotal Software, Inc. Understanding Rest. Viitattu 24.05.2018 https://spring.io/understanding/REST

PostgreSQL Global Development Group 2018. About. Viitattu 04.05.2018 https://www.postgresql.org/about/

PostgreSQL Tutorial. What is PostgreSQL? Viitattu 04.05.2018 http://www.postgresqltutorial.com/what-is-postgresql/

Postdot Technologies, Inc. Postman. Viitattu 04.05.2018 https://www.getpostman.com/