Tampereen ammattikorkeakoulu

Monoliitista mikropalveluiksi

Case maksupalvelu

Juuso Ahtiainen

OPINNÄYTETYÖ
Marraskuu 2020

Tietojenkäsittelyn tutkinto-ohjelma
Ohjelmistotuotanto

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittelyn tutkinto-ohjelma
Ohjelmistotuotanto

AHTIAINEN, JUUSO
Monoliitista mikropalveluiksi
Case maksupalvelu

Opinnäytetyö 32 sivua, joista liitteitä 0 sivua
Marraskuu 2020

Opinnäytetyö sai alkunsa yhteistyöyrityksen liian suureksi kasvaneen ohjelmiston ohjelmistoarkkitehtuurin ongelmista. Arkkitehtuurin aiheuttamat ongelmat olivat alkaneet vaikeuttamaan niin ohjelmiston jatkokehitystä kuin ylläpitoa. Kun ongelmaa tutkittiin, huomattiin että myös muilla yrityksillä oli liian suuriksi kasvaneista järjestelmistä johtuvia ongelmia. Opinnäytetyössä etsittiin syitä sekä ratkaisuja näihin esille nousseisiin ongelmiin ja niistä aiheutuviin haasteisiin. Työhön dokumentoitiin alkuasetelman syyt ja arkkitehtuuri- ja suunnitteluvalintojen perustelut sekä toteutustavat.

Opinnäytetyössä vertaillaan monoliittisen sekä mikropalveluiden ohjelmistoarkkitehtuurin vahvuuksia ja heikkouksia. Työssä selvitetään prosessia, kuinka monoliittisen arkkitehtuurin omaavan ohjelmiston ominaisuuden voi eritellä ja hajottaa omiksi mikropalveluiksi. Lisäksi havainnollistetaan esimerkkinä, kuinka yrityksen suureksi kasvaneen ohjelmiston maksupalvelun osuus käytännössä muutetaan mikropalveluiksi.

Opinnäytetyön lopputulos, dokumentoitu ohjeistus ohjelmiston ominaisuuksien erottelusta on hyödynnettävissä, kun etsitään tietoa laajan ohjelmiston hajauttamisesta mikropalveluiksi. Esimerkkinä käytetyssä hajauttamisessa rakennetut mikropalvelut eivät ole puhtaita oppikirjaesimerkkejä prosessista, vaan antavat realistisemman kuvauksen kompromisseista ja ratkaisuista, joihin voidaan päätyä työelämässä.

Asiasanat: monoliitti, mikropalvelu, maksujärjestelmä

ABSTRACT

Tampere University of Applied Sciences
Business Information Systems
Software Development

AHTIAINEN, JUUSO:
From Monolith to Microservices
Case Payment System

Bachelor's thesis 32 pages, appendices 0 pages
November 2020

This thesis' starting point was the client company's business software having grown big and its old programing architecture having started showing signs of problems. The problems, which were caused by the architecture, had started to limit the development and maintenance of the software. While researching the issues, it was revealed that other businesses faced similar problems with their business software. The thesis studies the causes and solutions for these problems and to the challenges caused by them. The starting point for the software is also documented, as well as how and why certain architectural or design decisions were made while fixing the issues.

The thesis presents the strengths and weaknesses of monolith and microservice architectures, how to recognize the needed properties to successfully separate a part of software to its own microservices, as well as how the process is carried out. As an example, a part of the aforementioned business software is separated into its own microservices and the process is explained step by step.

The theory presented in this thesis works as a source material for researching microservices, as well as to separate features from existing software to their own microservices. The case presented is not a clear textbook example of how the separation process is used and applied, but instead a more realistic representation on how the process could be handled in the business and what the result would be like.

Key words: monolith, microservice

SISÄLLYS

1 JOHDANTO 5
2 OHJELMISTOARKKITEHTUURI 6
3 MONOLIITTI 8
4 MIKROPALVELU 11
5 PALVELUN HAJOITTAMINEN MIKROPALVELUIKSI 14
6 CASE MAKSUPALVELU 18
6.1 Esittely 18
6.2 Tekniikat 19
6.3 Lähtökohdat 20
6.4 Suunnittelu 21
6.4.1 Latauslaite 22
6.4.2 Lataustapahtuma 23
6.4.3 Automaattinen maksatus 24
6.5 Toteutus 25
6.5.1 Hintatiedot 25
6.5.2 Maksatus 26
6.5.3 Vanha palvelu 27
6.6 Uusien palveluiden käyttöönotto 28
7 POHDINTA 30
LÄHTEET 32

1 JOHDANTO

Usein yrityksissä törmätään bisnestoiminnan kasvettua ongelmiin bisnekseltä kriittisen ohjelmiston kanssa. Ohjelmisto on rakennettu vastaamaan bisneksen tarpeita, mutta bisnes on kasvanut nopeasti ja sen tarpeet ohjelmistolle ovat muuttuneet nopeaan tahtiin. Näin ollen ohjelmiston kehitys ei ole pysynyt mukana tahdissa vaan on jouduttu usein tekemään nopeita ratkaisuja, jotta tarpeisiin on voitu vastata.

Näin kehitetyn ohjelmiston jatkokehityksessä tulee usein vastaan hetki, jolloin tehdyt ratkaisut ohjelmiston kehityksessä aiheuttavat suuria haasteita tai suoraan estävät ohjelmiston jatkokehittämisen. Vanhan ohjelmiston sisäiset riippuvuudet ovat monimutkaistuneet ja ominaisuuden muuttaminen tai uuden luominen voi aiheuttaa dominoefektin, joka vaikuttaa koko ohjelmiston toimintaan.

Tällainen hetki tuli vastaan yrityksessä, kun sen bisneslogiikan kannalta kriittisen ohjelmiston jatkokehittämisessä alettiin kohdata jatkuvasti haasteita. Opinnäytetyö nousi vastaukseksi haasteita varten tehdyn tiedon keräämisen talteen kirjaamiseen. Myös taustatiedot minkälaisesta tilanteesta on päädytty valittuihin ratkaisuihin, on esitelty, jotta näihin valintoihin voidaan tarvittaessa palata.

Opinnäytetyö esittää ratkaisuksi tähän ongelmaan ohjelmiston toiminnallisuuksien hajauttamista yhtenäisestä koodikannasta useiksi mikropalveluiksi. Se esittelee monoliittisen ohjelmistoarkkitehtuurin sekä mikropalveluarkkitehtuurin malleja sekä käy lävitse näiden tarjoamia etuja sekä haasteita kehitykselle. Tapa tunnistaa ominaisuuden riippuvuudet monoliittisessa ohjelmistossa sekä prosessi, kuinka se voidaan eriyttää omiksi mikropalveluiksensa, esitellään lukijalle. Opinnäytetyössä myös esitellään, kuinka maksupalvelu irrotettiin omiksi mikropalveluiksensa yrityksen vanhasta monoliittisesta koodikannasta, antaen näin esimerkin prosessista käytössä oikeassa tapahtumassa.

2 OHJELMISTOARKKITEHTUURI

Ohjelmistoarkkitehtuuri on käsitteenä laaja, eikä sille ole yhtenäistä määritelmää. Se mitä käsitteellä tarkoitetaan, vaihtelee puhujan sekä keskustelun kontekstin mukaisesti. Johtuen termin laajuudesta, sekä sen sisällön epäselvyydestä, pitää muistaa pitää mielessä konteksti, jossa termiä käytetään.

Ohjelmistoarkkitehtuurin perustana on ohjelmiston jakaminen osituksiin tiettyyn periaatteeseen nojaten. Tämä periaate voi perustua moniin asioihin kuten toiminnallisuuteen, hajautukseen, ohjelmiston skaalautuvuuteen tai tietoturvaan perustuen. Tämän pohjalta ohjelmisto voidaan jakaa osituksiin ja näistä osituksista voidaan strukturoida ohjelmiston kehitys.

D'Souza ja Wills määrittelevät ohjelmistoarkkitehtuurin rajoiksi, joiden tarkoituksena on estää ohjelmiston ylläpitäjiä sekä toteuttajia olemasta tarpeettoman luovia (D'Souza, Wills, 1998). Tältä pohjalta ohjelmistoarkkitehtuuria voidaan kuvata säännöstöksi, josta ei voi poiketa ilman painavaa syytä, sillä sen on tarkoitus määrittää ohjelmiston ytimen, jonka olisi tarkoituksena pysyä samana koko sen kehityksen sekä ylläpidon ajan. Ohjelmiston eri osia voidaan rakentaa ja muokata vapaasti tarpeiden mukaan, kunhan ne täyttävät valitun ohjelmistoarkkitehtuurin asettamat vaatimukset.

Vastaavasti IEEE:n standardi arkkitehtuurien kuvaamisesta määrittelee ohjelmistoarkkitehtuurin osaksi järjestelmän perusorganisaatiota, jossa esitetään periaatteet, jotka ohjaavat järjestelmän suunnittelua ja kehitystä sekä esitellään järjestelmän osat, niiden väliset suhteet ja niiden suhteet ympäristöön (IEEE 2000).

Olennaista IEEE:n määritelmässä on se, että se ei pidä sisällään pelkästään ohjelmiston rakennetta, vaan myös ohjelmiston käyttäytymistä sekä sen suhteita ympäristöön. Vastaavasti se saattaa, kontekstin mukaan, pitää sisällään myös ajonaikaisia muutoksia ohjelmiston, tai mahdollisesti ohjelmiston sisältämien osien, niin ikään komponenttien, ajonaikaisia muutoksia. Arkkitehtuuri saattaa koskea myös koodin rakennetta, käytettäviä loogisia ratkaisuja tai valittuja algoritmeja, esimerkiksi määritellä tietyn tavan lajitella tietoa (Koskimies, Mikkonen 2005).

Näin ollen, riippuen esitetystä kontekstista, ohjelmistoarkkitehtuurilla voidaan siis kuvata ja ohjeistaa erittäin tarkasti ohjelmiston toimintaa. Aina valitun ohjelmointikielen ja -tekniikoiden osalta sekä itse logiikan tasolle asti. Puhuttaessa eri ohjelmistoarkkitehtuureista tarkoitetaan sillä kuitenkin useasti vain periaatteita, joiden pohjalta ohjelmistoa kehitetään ja ylläpidetään. Eikä sillä oteta suoraan kantaa tapaan toteuttaa halutut periaatteet.

Esimerkiksi, valittu arkkitehtuuri saattaa asettaa rajoituksia tiedon tallennukselle sekä saatavuudelle ohjelmassa, mutta ei ota kantaa missä muodossa tieto pitää tallentaa tai vaihtoehtoisesti se ei esitä rajoitusta valittuun tietokantatekniikkaan.

Puhuttaessa ohjelmistoarkkitehtuurista, tarkoitetaan sitä säännöstöä, jonka pohjalta ohjelmistoa ja sen osia rakennetaan. Ja jonka pohjalta saadaan määriteltyä yhtenäiset vaatimukset virheidenkäsittelylle sekä skaalautuvuudelle ja muille vastaaville ominaisuuksille. Se ei itsessään ota kantaa käytettävään ohjelmointikieleen tai valittuihin loogisiin ratkaisuihin. Vaan sillä kuvataan abstraktilla tasolla haluttua ohjelmiston rakennetta, jotta saavutetaan halutut toiminnallisuudet sekä tiedostetaan ohjelmiston mahdolliset heikkoudet (Koskimies, Mikkonen 2005).

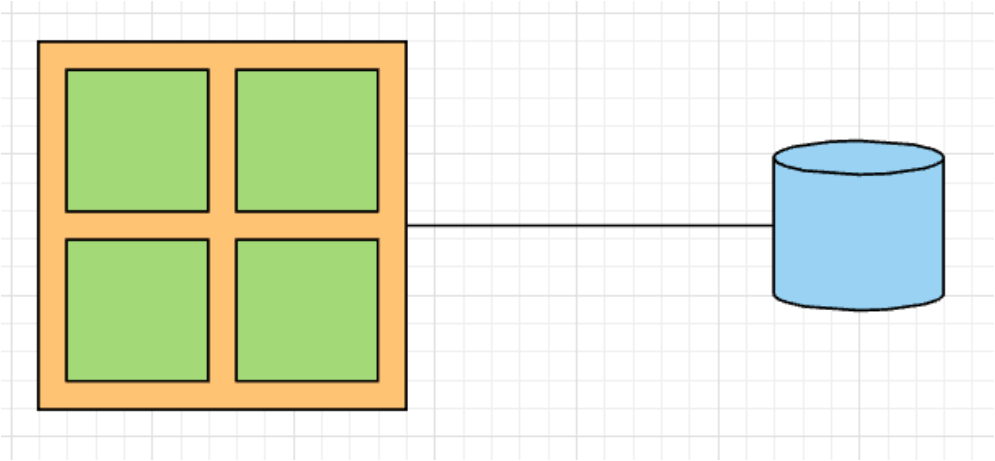
3 MONOLIITTI

Monoliittisessa arkkitehtuurimallissa koko ohjelmisto on rakentunut yhden prosessin ympärille, eikä sen osia voida suorittaa itsenäisesti. Se sisältää itsessään kaiken loogisen kokonaisuuden sekä toiminnallisuuden. Se myös jakaa samat palvelinresurssit, kuten tietokannan, muistialueen ja tiedostot. Usein sen tuotoksena on paketti, joka on itsenäinen kokonaisuus ja sisältää kaiken tarvittavan käyttöönottoa varten, mutta ei aina. Esimerkiksi Java-ympäristöön toteutetussa palvelussa tämä paketti olisi yksi JAR- tai WAR-tiedosto (Syrjä, 2016).

Huomattavaa on, että monoliittinen arkkitehtuuri ei ole esteenä ohjelmiston modulaariselle suunnittelulle. Monoliitin sisällä sen loogiset osat voidaan suunnitella ja toteuttaa toimimaan itsenäisinä, loogisina kokonaisuuksina ja kommunikoimaan erikseen määritettyjen rajapintojen kautta. Tällöin moduulia käyttävällä logiikalla ei ole väliä mitä moduulissa itsessään tapahtuu, jos rajapinnan kautta tapahtuvat pyynnöt ja sen saamat vastaukset eivät muutu. (Dragon ym. 2017)

Vastaavasti monoliittinen arkkitehtuuri voi käyttää sen ulkopuolisia palveluita, kuten kolmannen osapuolen tarjoaman tietokannan, joka toimii erillisenä kokonaisuutena. Kuitenkin nämä osat toimivat samassa prosessissa yhdessä, jakaen resursseja, ja näin olleen ovat aina riippuvaisia toisistansa. (Dragon ym. 2017)

Monoliittisella arkkitehtuurilla tarkoitetaan siis yhtenäisen koodikannan omaavaa ohjelmistoa. Se toimii yhdessä prosessissa itsenäisesti ja jakaa resurssit, esimerkiksi tietokantayhteyden kolmannen osapuolen tarjoamaan tietokantapalveluun, kaikkien toiminnallisuuksiensa kesken. Kuviossa 1 on esitetty monoliittisen arkkitehtuurin mallia. Se esittää yksittäisen prosessin, jonka alaisena eri ominaisuudet toimivat ja jakavat yhteisen tietokantayhteyden kolmannen osapuolen palveluun.



KUVIO 1. Monoliittisen arkkitehtuurin kuvaus

Johtuen monoliittisen arkkitehtuurin tarpeesta toimia yhtenä kokonaisuutena, ohjelmiston kasvaessa kasvaa usein myös sen käyttämä koodikanta suureksi. Tämä luo ongelmia sen hallitsemisessa, lisäten mahdollisuuksia loogisten riippuvuuksista johtuvien ongelmien syntymiseen sekä vaikeuttaen näin niiden huomaamista. Vastaavasti suuri koodikanta luo vaikeuden ohjelmiston kehittäjälle hallita kokonaisuutta, mikä myös kasvattaa mahdollisuutta virheisiin. (Dragon ym. 2017)

Esitettyjä ongelmia voidaan yrittää minimoida suunnittelemalla ja kirjoittamalla sovellus modulaariseksi, jolloin kehittäjän ei tarvitse osata hallita koko koodikantaa. Tällöin hänelle riittää vain yhden osan, moduulin, hallitseminen mikä on muutostyön alla. Tämä ei kuitenkaan poista mahdollisia syntyneitä loogisia riippuvuuksia eri moduulien väliltä, joita voi tulla huomaamatta tai tarkoituksella. Ja nämä riippuvuudet voivat aiheuttaa paljonkin töitä, joko suoraan tai kerronnallisesti. Modulaarinen ohjelmisto ei siis suinkaan itsessään ratkaise kaikkia ongelmia mitä monoliittisessa ohjelmistossa on.

Useasti ennen ohjelmiston uuden version julkaisua se testataan, joko manuaalisesti testaamalla tai kehittäjien kirjoittamilla, automatisoiduilla testeillä. Näiden testien tarkoituksena on varmistaa ohjelmiston haluttu toimivuus. Ohjelmistoon tehty muutos voi mahdollisesti vaikuttaa kerrannaisesti useisiin moduuleihin ohjelmistossa, jonka myötä useita testejä pitää korjata vastaamaan nykyistä todellisuutta. Vaikka nämä testit olisi automatisoitu, vie niiden ajaminen aikaa sekä

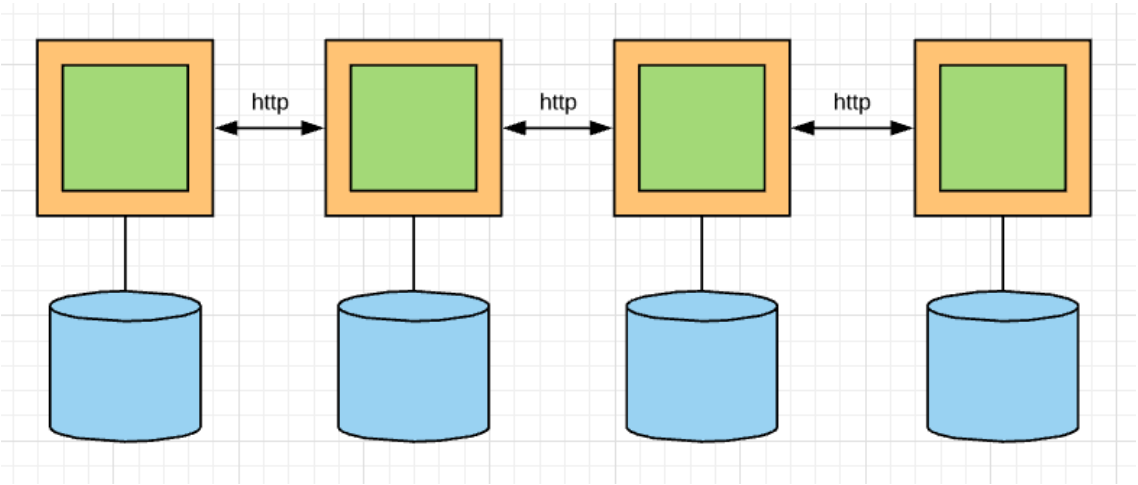
resursseja. Usein myös pelkästään muokatun osion testien ajaminen ei täytä testien vaatimuksia.

Vastaavasti uuden version käynnistäminen tuotannossa voi viedä paljon aikaa ja resursseja, ohjelman koon mukaan. Muutosten lisääminen palveluun aiheuttaa aina sen uudelleenkäynnistämisen. Ja riippuen ohjelmiston laajuudesta sekä sen monimutkaisuudesta uudelleenkäynnistys voi viedä kauankin aikaa. Oman ongelmansa aiheuttaa valittu tekniikka ja tapa ylläpitää ohjelmistoa. Edellä mainitut haasteet voivat myös aiheuttaa mahdollisesti palvelukatkoksia tai -häiriöitä käyttäjille. Tämä voi olla suurikin ongelma, mikäli ohjelmiston on tarkoitus olla jatkuvasti käyttökunnossa. (Syrjä, 2016)

4 MIKROPALVELU

Mikropalveluarkkitehtuurimallissa ohjelmisto on rakentunut useiden erillisten prosessien varaan, joiden pitää pystyä toimimaan itsenäisesti. Kukin näistä erillisesti toimivista prosesseista ei ole riippuvainen muista prosesseista ja omaa omat resurssit, joita ne käyttävät. Vastaavasti nämä mikropalvelut omaavat omat tietokannat sekä yhteydet näihin, joita ne eivät jaa muiden palveluiden kanssa. Bogard (2019), esittelee mikropalvelut palveluina, joidenka suunnittelun tarkoituksena on ollut saada kokoon mahdollisimman pieni autonominen kokonaisuus.

Mikropalveluarkkitehtuuria voi ajatella kokonaisuutena, joka koostuu useista itsenäisesti toimivista osista. Kukin näistä osista toimii itsenäisesti, hoitaen yhtä kokonaisuutta, joka sille on määrätty. Osa voi tarvita tehtävänsä suorittamiseen tietoa toiselta osalta, jonka se hakee rakennetun rajapinnan kautta, useimmiten http-pyyntönä. Tämän myötä saadaan kokonaisuus, jossa jokainen palvelu on selkeästi eroteltuna omiksi osioiksensa. Tätä on kuvattu kuviossa 2, jossa on esitelty neljä mikropalvelua omine tietokantayhteyksineen. Nämä palvelut kommunikoivat keskenään http-pyyntöjen kautta.



Kuvio 2. Mikropalveluarkkitehtuurin kuvaus

Hyötynä tästä saadaan hallittavan koodikannan pienuus. Tehdessä muutoksia palveluun, kehittäjän ei tarvitse huolehtia muusta kuin kyseisen palvelun logiikan ja koodikannan ymmärtämisestä, sekä niiden haltuun saamisesta. Kunhan tehdyt muutokset eivät aiheuta muutoksia palvelun rajapintojen tuottamien toiminnallisuuksiin ulkoapäin tai siitä saatuun tiedon muotoon. Tätä vasten on vaikea luoda

loogisia riippuvuuksia palveluiden välille, koska kukin palvelu voi toimia pelkästään muiden palveluiden rajapintojen luovuttamien tietojen varassa, eikä heillä ole pääsyä itse palvelun logiikkaan.

Vastaavasti palvelun päivittäminen uudempaan versioon on huomattavasti pienempi prosessi. Koska kehittäjän ei tarvitse välittää muista palveluista, tarvitsee hänen huolehtia pelkästään päivitettävän palvelun sisäisistä testeistä sekä palvelun uudelleenkäynnistämisestä. Koska nämä testit koskevat vain pientä koodikantaa, on niiden ajamiseen sekä mahdolliseen päivittämiseen vaadittu aika sekä työ huomattavasti pienempi. Myös päivitetyn ohjelmiston ajaminen tuotantoon on helpompaa, sillä yhden palvelun uudelleenkäynnistäminen muutosten viemiseksi vie vähemmän aikaa kuin koko järjestelmän, ja aiheuttaa mahdollisen katkoksen pelkästään yhdessä toiminnallisuudessa, eikä koko palvelussa. (Bogard, 2019)

Oman haasteen mikropalveluissa tuo tiedon laajalle leviäminen. Kun tarpeellinen tieto on useassa eri palvelussa, on sen ylläpitämisessä omat haasteensa. Siinä missä monoliittisessa palvelussa jonkin tiedon päivittäminen voi hoitua yhdellä tietokantakutsulla voi mikropalveluissa sama tieto olla hajautettuna useisiin eri palveluihin. Tällöin tiedon päivittäminen on pidempi prosessi, kun jokaisen palvelun rajapintaan pitää lähettää erillinen kutsu tiedon päivittämiseksi. Asian kääntöpuolena on päivitettävän tiedon määrän vähentyminen. Mikäli tietokantaobjekti on kasvanut suureksi, ja siitä päivitettäisiin vain yksi tietue, ei ole tarvetta koko objektin käsittelyyn. (Richardson, 2018)

Vastaavanlainen haaste aiheutuu logiikalle. Kun ohjelmistokehittäjällä ei ole välttämättä kunnollista tietoa muiden palveluiden toiminnasta ja logiikasta, on mahdollista, että bisneslogiikkaa kirjoitetaan useaan otteeseen. Tällöin vaikka toiminnallisuus muutettaisiin yhteen palveluun, voi toisessa palvelussa sama toiminnallisuus edelleen tehdä sen vanhalla, ei toivotulla tavalla.

Vastaavasti ketjuutuvia riippuvuuksia voi syntyä, joissa palvelut nojaavat toiminnassaan toistensa toteuttamaan logiikkaan. Tällaisessa tilanteessa pieni muutos voi aiheuttaa helposti ketjureaktion koko järjestelmän lävitse. Vastaava uhka on myös olemassa muissa arkkitehtuureissa, joten se ei ole mikropalveluille uniikki.

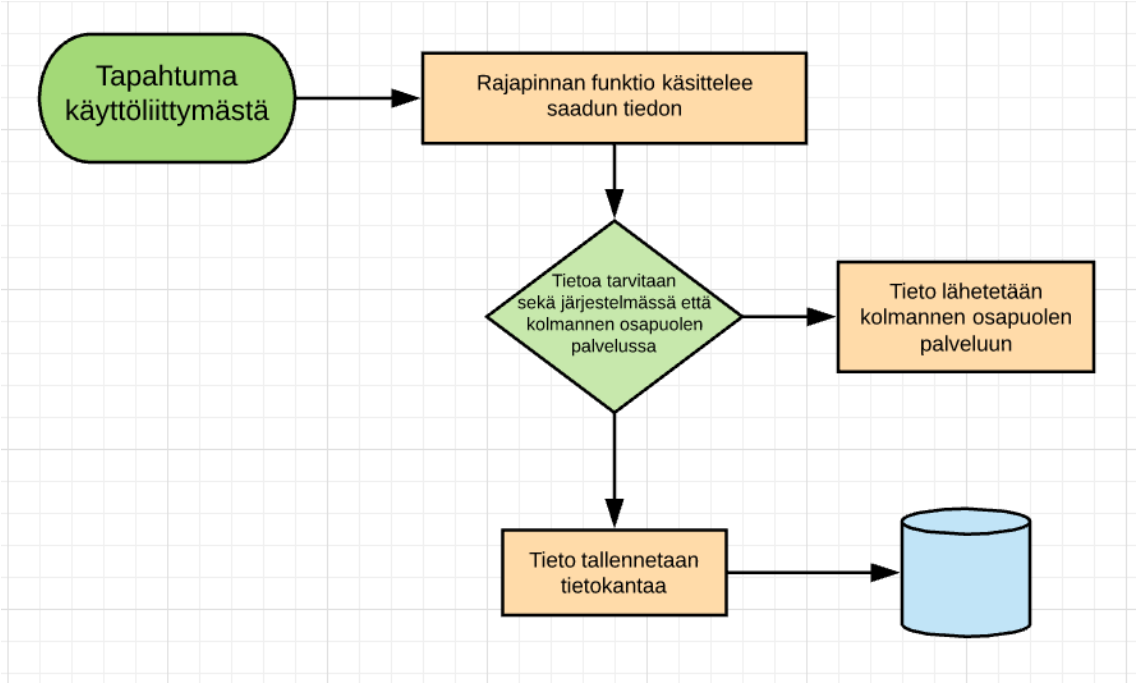
Molempia edellä mainittuja haasteita voidaan estää sekä vähentää parantamalla palvelusta tehtyä dokumentaatiota. Suunnittelemalla eri palveluiden vastuualueet selkeästi sekä kirjaamalla missä mikäkin toiminnallisuus tapahtuu, saadaan rajattua ohjelmiston kehittämistä halutulle alueelle. Myös uutta ominaisuutta lisätessä pitää käydä aina lävitse jo olemassa olevat palvelut ja harkita, minkä palvelun alaisuuteen toiminnallisuus kuuluu tai onko se jo olemassa jossain muodossa. (The InfoQ eMag, 2018)

5 PALVELUN HAJOITTAMINEN MIKROPALVELUIKSI

Kun muutetaan monoliittisella arkkitehtuurilla rakennettua järjestelmää mikropalveluiksi, pitää tässä paloittelussa ottaa huomioon useita eri asioita. Useasti monoliittinen järjestelmä on rakentunut vuosien aikana, useiden kehittäjien toimesta, vastaamaan ainakin osittain sen ajan yrityksen bisnesmallia sekä tapaa toimia.

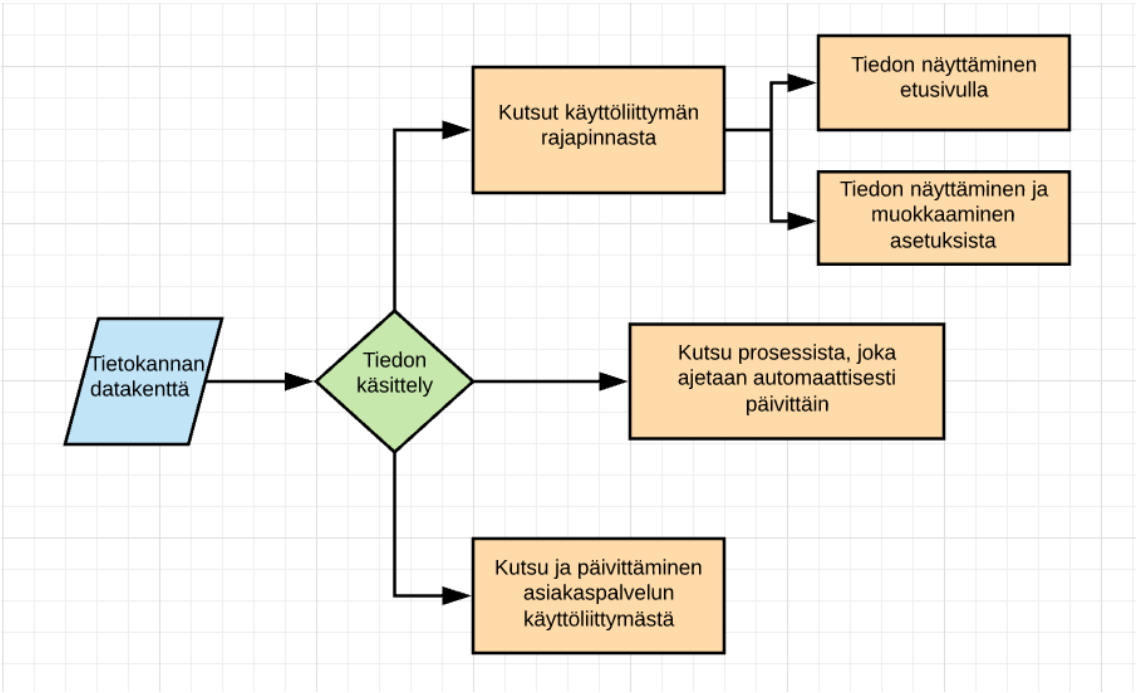
Kun tavoitellaan tällaisen palvelun uudelleenkirjoittamista mikropalveluina, pitää pystyä tunnistamaan tiedon kulku ohjelmiston sisällä sen toteuttamissa tehtävissä. Tämä tieto on tarpeellista suunnitellessa, kuinka samat prosessit tullaan suorittamaan uudella järjestelmällä, eikä mitään operaatioita tai muutoksia jää huomioimatta. (Evans, 2016)

Yksi tapa tunnistaa tämän tiedon kulku on katsoa toiminnallisuutta reunalta, käyttäjärajapinnalta, esimerkiksi graafisesta käyttöliittymästä Valitaan toiminnallisuus tai tapahtuma käyttöliittymästä ja seurataan sen käyttämän tiedon kulkua syvemmälle ohjelmiston syövereihin, kirjaten ylös minkä kaiken lävitse tieto kulkee ja mitä sille tehdään. Kuviossa 3 on esitetty esimerkki mahdollisesta tiedonkulusta seuraamalla sen kulkua käyttöliittymästä alkaen. Tämä tiedon kulku voidaan kirjata ylös eri tavoilla, esimerkiksi vuokaaviona. (Bogard, 2019).



KUVIO 3. Esimerkki tiedon kulusta seuratessa sitä käyttöliittymästä alkaen.

Toinen vaihtoehto on käydä ohjelmiston toimintaa lävitse päinvastaisesta suunnasta. Katsoa tietokanta taulu kerrallaan läpi, ja seurata missä kyseistä tietoa käytetään. Tarpeellista on myös kirjata koska, mikä ja kuinka sitä muokataan ja näin edelleen. Kuljetaan niin sanotusti pohjalta kohti reunaa, eli päinvastoin kuin edellisessä esimerkissä. Esimerkki mahdollisesta tiedon kulusta on kuvattu kuviossa 4. Käymällä tiedon käsittelyn polkuja tietokantatasolta lähtien saadaan helposti löydettyä ohjelmiston sisäiset automatisoidut järjestelmät. Nämä automatisoidut järjestelmät eivät usein näy päällepäin ja toimivat ilman erillistä käyttäjän antamaa käskyä. (Bogard, 2019)



KUVIO 4. Esimerkki tiedon kulusta järjestelmässä, seuraten sitä tietokantatasolta

Kun tutkitaan ohjelmiston toimintaa sekä tiedon kulkua siinä, käytetään usein molempia edellä mainittuja tapoja, jotta mitään toiminnallisuutta ei jäisi mahdollisesti pimentoon tai huomamaamatta. Näin kerätyn tiedon pohjalta on mahdollista alkaa suunnittelemaan ohjelmistoa tai sen osaa mikropalveluina. Kuten aikaisemmassa luvussa esitettiin, (mikro)palvelut ovat yksittäisessä prosessissa toimivia, itsenäisiä ohjelmistoja, jotka on suunniteltu pienin mahdollinen autonominen kokonaisuus mielessä, jonka palvelu toteuttaa. Tämä palvelun tehtävä tai tehtävät pitää määritellä jo tässä suunnitteluvaiheessa.

Se kuinka ohjelmiston toiminnallisuudet hajautetaan eri osiin ja mitä tehtäviä kukin palvelu tulee suorittamaan, on riippuvainen useista asioista. Mitä ohjelmiston hajautuksella ajetaan takaa. Onko tarkoituksena ajaa vanhaa ohjelmistoa sekä mikropalveluita rinnakkain, jolloin ne toimivat yhdessä. Vai siirretäänkö koko toiminnallisuus kerralla mikropalveluiden varaan. Mikäli edeltävä ohjelma on kirjoitettu modulaariseksi, voi usein vanha moduulin logiikka jo toimia mahdollisesti mikropalveluna, mikäli se on tarpeeksi pieni.

Luomalla niin kutsuttu pilottiversio mikropalveluina vanhan ohjelmiston ydintoiminnasta on yksi vaihtoehto lähtökohdaksi. Tämä mikropalvelu tai mikropalvelut tekevät vain ja ainoastaan ydintoiminnan tarvitsemat ominaisuudet, eivätkä mitään muuta. On myös mahdollista, että vanhaan ohjelmistoon on rakentunut useampia ydintoiminnaksi verrattavia toiminnallisuuksia, joita se suorittaa. Näiden tunnistamisessa on tärkeää tuntea yrityksen bisnesidea sekä löytää tiedonkulun polut kullekin toiminnalle ohjelmistosta. (Evans, 2016)

Kun nämä tiedonkulun polut on tunnistettu, pystytään niiden tarvitsemien logiikoiden pohjalta luomaan kuvaaja tarvittavista ominaisuuksista sekä suoritettavista tehtävistä. Saatujen tietojen sekä tehdyn kuvaajan pohjalta pystytään alkamaan määrittelemään kullekin mikropalvelulle sen vastuualueita sekä tehtäviä. Ja tämän kokonaisuuden pohjalta pystymme rakentamaan suunnitelman mikropalveluiden kehittämiselle. (Evans, 2016)

Mikäli vanhan ohjelmiston rakenne sen sallii, voidaan siitä irrottaa toiminnallisuus omaksi kokonaisuudekseen ja rakentaa se mikropalveluna tai mikropalveluina. Näitä palveluita voidaan sitten ajaa vanhan ohjelmiston rinnalla, ja ohjelmisto voi käyttää toiminnallisuutta rakennetun rajapinnan kautta. Tällöin ei ole tarpeellista rakentaa koko kokonaisuutta kerralla.

Edellä esitelty tapa siirtää ominaisuudet yksi kerrallaan mikropalveluiksi, pienentää tarvittavan kokonaisuuden hallintaa. Se myös vähentää aikaa, jonka resurssit ovat kiinni kerralla uuden ohjelmiston kehittämisessä. Mikropalvelunversioita ominaisuuksista voidaan suunnitella, rakentaa, testata ja ottaa käyttöön yksi kerrallaan. Ja vastaavasti ominaisuutta voidaan tarvittaessa jatkokehittää jo tässä

vaiheessa. Usein vanha palvelu on kuitenkin aktiivisesti käytössä muutosta tehdessä ja siihen tehdään jatkuvasti myös erisuuruisia kehitystöitä ylläpidon osalta. Näin ei ole tarvetta jäädyttää vanhan palvelun kehitystyötä muutostöiden ajaksi.

Kuitenkin edellä mainittu tapa vie usein kokonaisuutena enemmän resursseja. Rakentaessa ominaisuuksia yksi kerrallaan, usein on tarve muokata hieman vanhaa järjestelmää. Ainakin uusien rajapintojen rakentaminen on tarpeellista vanhan järjestelmän sekä uuden mikropalvelun tai mikropalveluiden toimimiselle yhdessä. Riippuen tavasta miten vanha ohjelmisto on rakennettu, tämä voi viedä paljonkin aikaa. Vastaavasti siirron yhteydessä saatetaan lisätä pelkän ominaisuuden siirron yhteyteen ylimääräistä kehitystyötä, joka tuo mukanansa omat ongelmansa.

6 CASE MAKSUPALVELU

6.1 Esittely

Yrityksen ohjelmisto on aluksi rakentunut hallitsemaan ajoneuvolataukseen liittyvää ohjauslogiikkaa. Kommunikaatio kentällä olevien latauslaitteiden kanssa tapahtuu Open Charge Alliancen määrittelemän OCPP-standardin mukaisesti. Open Charge Alliance on kansainvälinen, johtavien elektronisten ajoneuvojen latausjärjestelmien tarjoajien yhteenliittymä, joka pyrkii edistämään avoimia standardeja lataustekniikoille. Tällä pyritään varmistamaan, että auto sekä laturi pystyvät kommunikoimaan aina keskenään.

Ohjelmiston kautta voidaan seurata latureiden lähettämää viestiliikennettä. Tästä viestiliikenteestä voidaan kerätä ylös tietoja latauksista, kuten niiden aloitus ja lopetusajat sekä ladattu energia. Latauslaitteita pystytään myös tarvittaessa käskyttämään standardin mukaisesti etäkäskyillä. Niiltä voidaan myös vaatia latausluvan varmistusta ohjelmistolta ennen latauksen aloittamista.

Ajan myötä ohjelmistoon on lisätty bisneslogiikkaa, jolla on voitu kerätä lataustapahtumista niiden tietoja muistiin tietokantaan. Vastaavasti siihen on lisätty laitteiden sijaintitietoja ja nimeämisiä, helpottaen laturien tunnistamista ja löytämistä. Käyttäjien tiedot käsitellään kolmannen osapuolen palvelussa, joka on tähän erikoistunut ja yrityksen ohjelmistossa käsitellään ainoastaan palvelun antamia uniikkeja tunnisteita käyttäjistä. Näiden tunnisteiden avulla on kehitetty lupajärjestelmä latausoikeuksien sekä laturien hallinnoimista varten.

Ohjelmistoon kehitettiin mahdollisuus asettaa hinnasto latauslaitteille. Tämä hinnasto mahdollisti lataustapahtumille hinnan laskemisen. Jotta käyttäjää voidaan laskutta myös takautuvasti, alettiin keräämään käyttäjiltä heidän lataustapahtumien historiaa, jolloin käyttäjä pystyy maksamaan useamman maksamattoman lataustapahtuman kerralla.

Ohjelmisto itsessään ei ole aikoinaan rakennettu tarkoituksella monoliitiksi, vaan se on siksi rakentunut ajan saatossa nopeasta pilotista. Se aikojen saatossa kir-

joitettiin kertaalleen uusiksi modulaarisemmaksi, pyrkien eroon alkutaipaleen nopeasta konseptiin pohjautuneesta ohjelmistosta. Kuitenkin tämä laajentunut koodikanta on alkanut muodostua esteeksi palvelun jatkokehittämiselle sekä ylläpidolle. Vastaavasti lukumäärältään kasvanut ominaisuuksien määrä ohjelmassa sekä niiden monipuolistuneet tarkoitukset asettavat erilaiset tarpeet ja vaatimukset saatavuudelle aikaisempaan verrattuna. Monoliittisen ohjelman huonona puolena, kun yksi ominaisuus on alhaalla, kaikki ovat alhaalla.

Johtuen käytettävien resurssien rajallisuudesta, sekä aktiivisesta kehitystyöstä joka vanhan ohjelmiston ominaisuuksiin kohdistuu, todettiin parhaimmaksi hajauttaa kukin ohjelman ominaisuus yhdeksi tai useammaksi mikropalveluksi. Tällöin vanhan ohjelmiston muiden toiminnallisuuksien kehitystyö ei kärsi tehtävästä muutostyöstä. Vastaavasti mikropalveluksi siirretty ominaisuus pystytään ottamaan käyttöön vanhan ohjelmiston rinnalla ja käytössä olleet resurssit voidaan siirtää tarvittaessa muihin kehitysprojekteihin.

6.2 Tekniikat

Vanha ohjelmisto on kirjoitettu JavaScript-koodilla, käyttäen Node.js:n runtime -ympäristöä. Node.Js on avoimella lähdekoodilla kirjoitettu, alustariippumaton ympäristö. Se pohjautuu Googlen Chrome V8 JavaScript-moottoriin. Tämä mahdollistaa JavaScript-koodin ajamisen palvelimella. Perinteisesti JavaScript-koodia on ajettu upotettuna verkkosivujen html-koodiin ja sen suorittamisesta on vastannut käyttäjän selaimen JavaScript-moottori. (Wandschneider, 2013)

Tietokantatekniikoista on valikoitunut käyttöön MongoDB. Se on MongoDB Inc:n kehittämä dokumenttitietokantaohjelmisto. Se poikkeaa perinteisestä relaatiomallisista tietokannoista ja kuuluu niin sanotusti NoSQL-tietokantoihin. MongoDB käyttää JSON-objektin kaltaista tietomuotoa. (Chodorow, 2013) Se oli myös valikoitunut käyttöön vanhassa ohjelmistossa, joten koimme turhaksi lähteä muuttamaan toimivaksi todettua ratkaisua.

Vanha ohjelmisto toimii kontitettuna Docker-kontissa palvelimella. Docker-kontti on Docker Inc. -yrityksen tuote. Se tarjoaa mahdollisuuden standardoituun, eris-

tettyyn ajoympäristöön, jossa palvelua voidaan ajaa. Se pitää sovelluksen tarvitsema minimiajoympäristö yhdessä paketissa, joka on perinteisesti käytettyä virtuaalista palvelinta kevyempi ajaa. Docker myös mahdollistaa sovelluksien helpomman siirtämisen tarvittaessa, kun koko sovellus ja sen tarvitsemat tiedostot kulkevat mukana yhtenä tiedostona. (Mouat, 2016)

Kirjoitettavat mikropalvelut tullaan toteuttamaan samoilla tekniikoilla. Nämä tekniikkavalinnat auttavat kehittäjien helpompaa liikkumista eri projektien välillä. Kun yrityksen asiakkaiden käyttämä verkkosivu sekä yrityksen oman ohjelmiston palvelut on kehitetty samalla JavaScript-kielellä, helpottaa se työstä toiseen siirtymistä. Yrityksessä käytettävät resurssit ohjelmiston ylläpitämiseen ja jatkokehitykseen ovat rajalliset, joten tarjoutuva mahdollisuus liikkua eri kehitysroolien välillä on tärkeää.

6.3 Lähtökohdat

Aloittaessa maksujärjestelmän siirtämistä omiksi, erillisiksi mikropalveluiksensa, on osa vanhan ohjelmiston ominaisuuksista jo siirretty erillisiksi mikropalveluiksensa. Vanhaan ohjelmistoon on kirjoitettu REST-rajapinta, jonka kautta vanha ohjelmisto sekä uudet palvelut kommunikoivat keskenään. Vastaavanlainen rajapinta tultaisiin rakentamaan myös vanhan ohjelmiston sekä uusien maksupalveluiden välille. Myös erillinen ohjelma on rakennettu hoitamaan tiedon liikkumista oikeassa muodossa loppukäyttäjälle näytettäviin näkymiin.

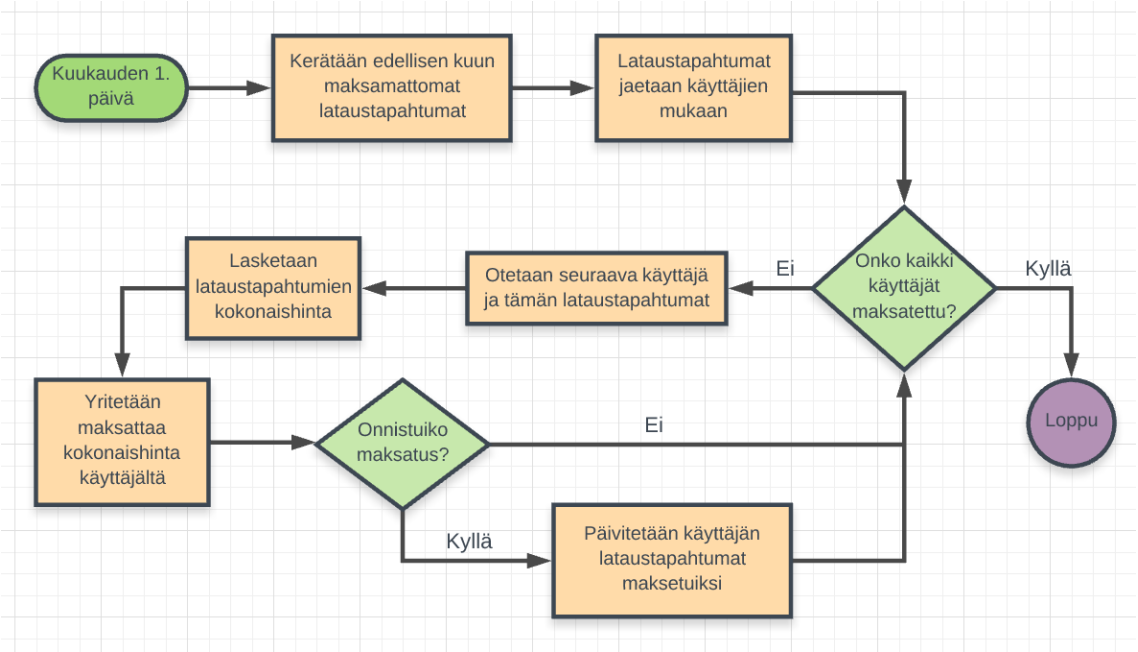
Tarkoituksena muutostyötä tehdessä ei ole tehdä jatkokehitystä ominaisuuksiin. Tarkoituksena on eriyttää maksujärjestelmän bisneslogiikka omaksi kokonaisuudekseen, mahdollisesti erillisiksi mikropalveluiksi. Ominaisuuden erittelyn tarkoituksena olisi jakaa resurssien käyttöä sekä pyrkiä saamaan aikaiseksi paremmin hallittava kokonaisuus kehittäjille. Tarkoituksena olisi myös jakaa palvelinresurssien tarvetta sekä käyttöä useampien eri ohjelmien välille.

Myös monoliittisen arkkitehtuurin aiheuttama uhka järjestelmän kaatumisesta vakavan virheen vuoksi tiedostettiin. Tällainen vakava virhe saattaisi pysäyttää suoritettavia operaatioita, joista virhe ei olisi ollut lähtöisin. Tämänkaltaisten kaatumisten seuraamusten selvittäminen sekä korjaaminen aiheuttaisivat ylimääräistä

työtä sekä päänvaivaa ohjelmistokehittäjille, unohtamatta tästä syntyviä huonoja asiakaskokemuksia.

Suunnittelun alkutilanteessa on tiedossa käyttötapaukset, jonka pohjalta uuden ominaisuuden tulisi toimia. Myös vanhan ohjelman maksatuksen pitäisi toimia näiden käyttötapausten mukaisesti, mikäli se ei ole muuttunut ajan varrella. Kunkin käyttäjän lataustapahtumat kerätään talteen, sekä niille mahdollisesti laskettu hinta.

Kuviossa 5 on esitelty kuukausimaksatuksen kulkua palvelussa. Uuden kuukauden alkaessa, kerätään käyttäjän edeltävän kuun aikana loppuneet maksulliset lataustapahtumat. Näiden lataustapahtumien hinta lasketaan yhteen ja se laskutetaan käyttäjän kortilta, joka on tallennettu kolmannen osapuolen palveluun. Palvelu ei tässä tilanteessa vielä luo automaattisesti maksutapahtumasta kuittia, jossa on eriteltynä eri lataustapahtumat. Käyttäjälle toimitettava kuitti on automaattisesti luotu kolmannen osapuolen palvelun kautta.



KUVIO 5. Kuukausimaksatuksen käyttötapauksen kulku

6.4 Suunnittelu

Varmistaaksemme ettemme riko mahdollisia riippuvuuksia, joissa ominaisuuden tietoja on käytetty, käymme palvelun lävitse. Koska ominaisuudessa on kyseessä

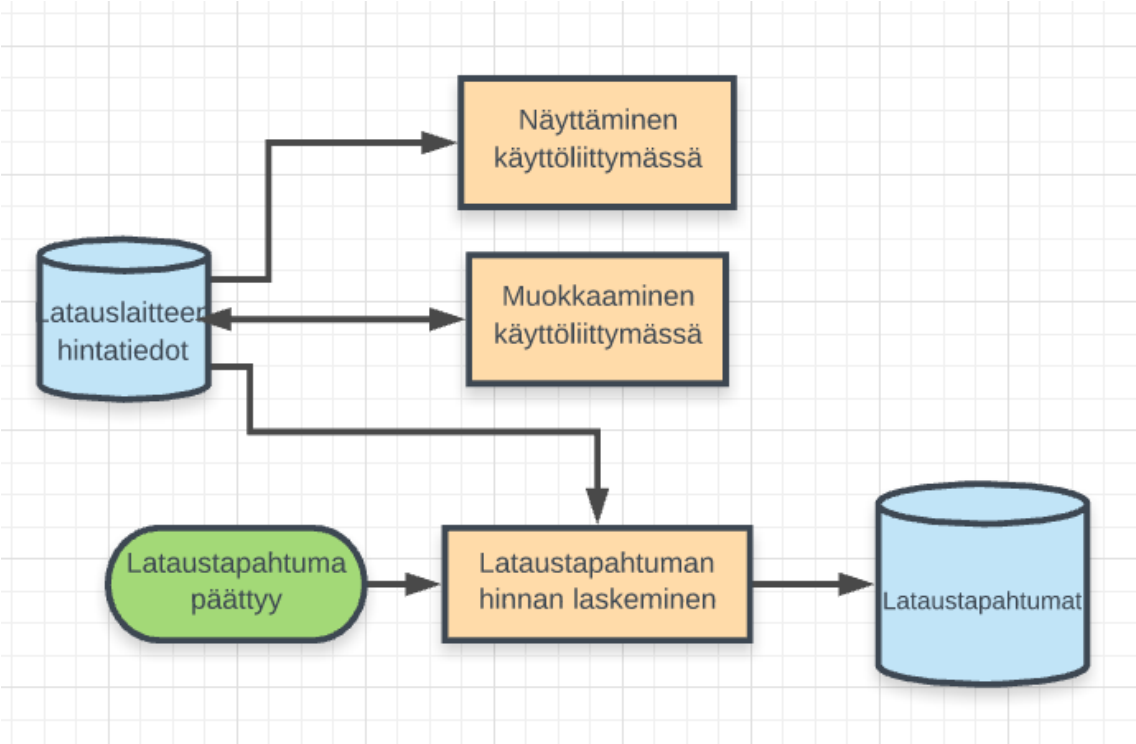
automatisoitu järjestelmä, ei sillä ole käyttöliittymätasolla näkyvyyttä. Eli käymme lävitse ominaisuuden toiminnan sekä sen riippuvuudet vanhasta ohjelmistosta aloittaen tietokantatasolta ja kulkien tietämme ylöspäin. Kirjaamme löytyneet riippuvuudet sekä ominaisuudet ylös, jotta mikään ei unohtuisi ja voimme tarvittaessa palata näihin.

Edellisen luvussa esitellyn bisneslogiikan kuvauksesta tiedämme, että tarvitsemme lataustapahtuman hinnan muodostamiseksi tietoomme latauslaitteiden hintatiedot, lataustapahtumien keston sekä energiankulutukset ja latauksen suorittaneen asiakkaan tunnisteen. Vastaavasti lataustapahtumien maksattamiseen asiakkaalta, tarvitsemme hänen kootut maksamattomat lataustapahtumat, näiden yhteenlasketun hinnan sekä asiakkaan tunnisteen kolmannen osapuolen maksupalvelua varten.

6.4.1 Latauslaite

Aloitamme ominaisuuksien läpikäynnin tietokannan kokoelmista, joista etsimme ensimmäisenä latauslaitteiden kokoelman. Tutkimalla latauslaitteen objektia kokoelmassa näemme, että objektin osana, yhtenä parametrinä on tiedossa laitteen hintatiedot aloitukselle, energiankulutukselle sekä ajalle lataustapahtumille. Etsiessämme vanhan ohjelmiston koodista funktioita, jotka käyttävät latauslaitteen tietokantaobjekteja sekä tarkalleen näiden hintatietoja.

Tutkiessamme löydämme tiedon reitit, joita pitkin latauslaitteen hintatiedot haetaan näytettäväksi loppukäyttäjälle. Myös hintatietojen päivittämiseen tarvittavat reitit kirjataan ylös kaavioon. Näin pystymme luomaan selkeän näkymän ominaisuuksien riippuvuuksista. Löydämme myös tehtävän tietokantahaun, joka suoritetaan lataustapahtuman loppuessa. Automatiikka laskee tällöin lataustapahtumalle hintatiedot perustuen latauslaitteeseen merkittyihin hintatietoihin. Näitä löydettyjä tiedon reittejä on kuvattu kuviossa 6.



KUVIO 6. Latauslaitteen hintatietojen tiedonkulku

Tämän tiedon pohjalta näemme muiden ominaisuuksien tarpeet latauslaitteen hintatiedoille. Tavoitteena on erottaa latauslaitteen tekniset tiedot sekä sen hintatiedot erillisiksi dokumenteiksi, eri tietokantoihin, joten tulemme tarvitsemaan rajapinnan vanhaan palveluun, jotta se osaa hakea hintatiedot niitä tarvitessaan uudesta palvelusta. Vastaavasti meidän pitää muistaa merkata kukin latauslaitteen hintatietojen tietokantadokumentti sen latauslaitteen uniikilla tunnisteella, jolle se kuuluu. Lataustapahtuman hintatietojen dokumentointi käydään lävitse tarkemmin seuraavassa luvussa.

6.4.2 Lataustapahtuma

Tarkastellessamme lataustapahtuman tietokantatiedostoa, löydämme automatisoidut tapahtumat, joiden mukaisesti se luodaan. Latauksen alkaessa, saatuamme latauslaitteelta viestin, luomme lataustapahtumalle dokumentin. Tällöin dokumenttiin merkataan sille uniikki tunnistetieto, latauksen aloittaneen käyttäjän tiedot, latauslaite jolta lataus suoritetaan sekä alkamisaika ja muuta teknistä tietoa.

Lataustapahtuman loputtua, saamme latauslaitteelta viestin, jonka pohjalta lataustapahtumalle lasketaan ja merkataan muun muassa sen loppumisaika, kulutettu energia sekä muuta teknistä tietoa. Myös lataustapahtuman mahdollinen hinta lasketaan tässä tilanteessa. Latauslaitteen tiedot tarkastetaan lataustapahtumaan merkityltä latauslaitteelta sen uniikin tunnuksen pohjalta. Mikäli latauslaitteen tiedoista löytyy hintatiedot, lasketaan käytetylle energialle sekä ajalle hinta, ja ne lisätään yhdessä mahdollisen aloitusmaksun mukana osaksi lataustapahtuman dokumenttia. Myös lataustapahtuman kokonaishinta lasketaan ja merkataan dokumenttiin.

Mikäli hintatietoja ei ole olemassa tai ne on merkattu latauslaitteen hintatiedoissa nollaksi, merkataan lataustapahtuman hinnaksi nolla euroa. Jokaiselta onnistuneelta ja suoritetulta lataustapahtumalta kuitenkin löytyy hintatiedot sen dokumentista, vaikka lataus olisi ollut ilmainen.

Tämän tiedon pohjalta näemme mitä lataustapahtuman hintatietojen muodostaminen vaatii. Ja mitä meidän pitää ottaa huomioon erottaessamme lataustapahtuman hintatiedot itse lataustapahtuman dokumentista. Tarvitsemme mahdollisuuden luoda ja laskea lataustapahtumalle hintatiedot lataustapahtuman loppuessa. Lataustapahtuman hintatieto pitää myös olla liitetty lataustapahtumaan uniikilla tunnisteella. Uusi palvelu tarvitsee rajapinnan, jonka kautta latauksen hintatietoja voi hakea niin latauksen hintatietojen uniikilla tunnisteella kuin myös lataustapahtuman uniikilla tunnisteella. Näin vanha palvelu voi hakea loppukäyttäjälle toteutuneen lataustapahtuman hintatiedot.

6.4.3 Automaattinen maksatus

Automaattinen maksatus laukaistaan ajastetulla http-pyynnöllä palvelun rajapintaan, joka tulee palvelun ulkopuolelta kuun ensimmäisenä päivänä. Maksatusta suorittaessaan se hakee ensiksi kokoamalla lataustapahtumien tietokannasta kaikki tapahtumat, jotka on merkitty loppuneen ennen kuluvan kuukauden alkua, ja joiden hintatietoihin ei ole merkattu maksua suoritetuksi. Tämä koonnin tulos on jaettu perustuen niihin merkattuihin käyttäjien tunnisteisiin. Näin kullakin käyttäjän tunnisteella on sille merkattu lista lataustapahtumista.

Dokumenttien koonnin pohjalta voidaan käydä kunkin käyttäjän maksamattomat lataustapahtumat yksi kerrallaan lävitse. Lataustapahtumista lasketaan käyttäjälle kokonaissumma, joka pyritään laskuttamaan käyttäjältä kolmannen osapuolen palvelun kautta välittömästi. Mikäli maksatus onnistuu, merkataan kunkin lataustapahtuman maksaminen onnistuneeksi. Vastaavasti maksutapahtuman epäonnistuessa, annetaan lataustapahtumien olla ja siirrytään seuraavaan käyttäjään.

Automaattinen maksatus tarvitsee itsellensä kootusti lataustapahtumien hintatiedot, sekä latauksen suorittaneen loppukäyttäjän tunnisteen. Tällä loppukäyttäjän tunnisteella voidaan maksattaa loppukäyttäjältä lataustapahtumat. Automaattinen maksatus ei tarvitse kovinkaan monimuotoisesti eri tietoja, mutta tarvittavien tietojen volyymi on suuri. Vastaavasti se nostaa tarpeen palveluiden rajapinnoille, jotta se voi hakea kootusti tiedot lataustapahtumien hinnoista.

6.5 Toteutus

6.5.1 Hintatiedot

Toteutus aloitetaan eri hintatietojen keräämisellä omaan palveluun. Kirjoitamme uuden palvelun, jonka tarkoituksena on ylläpitää hintatietoja lataustapahtumille sekä latauslaitteille. Kukin hintatieto tulee omaamaan uniikin tunnisteen, sekä tiedon mille lataustapahtumalle tai latauslaitteelle se kuuluu. Palvelulle tullaan rakentamaan REST-rajapinta, jonka kautta sen tietoja voidaan hakea sekä päivittää käyttäen http-kutsuja. Palvelu rakennetaan NodeJs-palveluna, kuten edeltäjänsäkin.

Aloitamme rakentamalla tietokantadokumentit latauslaitteen sekä lataustapahtumien hintatiedoille. Latauslaitteen tietokantadokumentti rakennetaan sisältämään latauslaitteen hintatiedot ajalle, energialle, aloitusmaksulle. Näiden lisäksi sille tehdään tietue latauslaitteen uniikille tunnisteelle. Näin palvelusta on mahdollista tunnistaa mille latauslaitteelle kukin hintatieto kuuluu.

Vastaavasti tehdessämme lataustapahtuman tietokantadokumenttiin merkataan lataustapahtumalle laskettu kokonaishinta, sen kuluttaman energian hinta sekä

sen aikaperusteinen hinta. Dokumenttiin merkataan myös lataustapahtuman uniikki tunniste, latauksen suorittaneen käyttäjän uniikki tunniste sekä latauslaitteen uniikki tunniste, jossa lataus on suoritettu.

Rakennamme palvelulle HTTP-pyynnöt hintatietojen luomiselle sekä päivittämiselle. Näin voimme myöhemmin rakentaa vanhaan ohjelmistoon ohjauksen logiikalle hintatietojen hakemiselle sekä päivittämiselle. Kummallekin tietokantadokumentille rakennamme normaalit CRUD-pyynnöt rajapintaan. Eli mahdollisuuden luoda (Create), lukea (Read), päivittää (Update) ja poistaa (Delete). Luomme myös käytettävissä olevan pyynnön, jonka kautta pystymme hakemaan kaikki tietokannassa olevat dokumentit, tietyin rajoituksin. Voimme esimerkiksi rajata hakutulokseksi tietyn latauslaitteen tai käyttäjän lataustapahtumat. Rakennamme myös pyynnön, joka toimittaa listan kaikista käyttäjistä, joilla on maksamattomia lataustapahtumia ennen kuluvan kuun alkua.

6.5.2 Maksatus

Kuten esittelimme luvussa 6.4.3 Automaattinen maksatus, palvelu tarvitsee toimiakseen muiden palveluiden tietoja. Näitä tietoja se tulee hakemaan hyödyntämällä vanhaan ohjelmistoon sekä uuteen hintapalveluun rakennettuja http-rajapintoja. Itse maksatus käynnistetään ajastetulla http-pyynnöllä maksatuspalveluun.

Jotta asiakkaalta voidaan maksetutta lataustapahtumien kokonaishinta, tehdään se kolmannen osapuolen palvelun kautta. Kolmannen osapuolen palvelun käyttämisellä on tarkoitus varmistaa käyttäjän tärkeiden tietojen, kuten tässä tapauksessa maksukortin tietojen, pysyminen tallessa ja erillisenä bisneslogiikasta. Kukin käyttäjä käy lisäämässä itse luottokorttinsa kolmannen osapuolen palveluun heidän portaalinsa kautta. Tämä portaali tarjotaan asiakkaalle yrityksen ohjelmiston kautta.

Tämä kolmannen osapuolen palvelu palauttaa yritykselle satunnaisesta merkkisarjasta luodun koodin. Tämä koodi on uniikki kullekin käyttäjälle sekä maksukortille. Sillä voidaan myös maksetutta ainoastaan yhdessä yritykselle annetun oman uniikin koodin kanssa, jota ei ole tallennettu tietokantoihin. Mikäli yrityksen

oma koodin on virheellinen, ei maksamista sallita. Tällä voidaan varmistaa, että mahdollisen tietomurron tapahtuessa järjestelmään olisivat käyttäjien maksutiedot turvassa. Vaikka murtautuja kopioisi tietokannat, olisivat asiakkaiden tallennetut koodit hyödyttömiä yksissään.

Maksatuksen logiikka hakee ensiksi kaikkien käyttäjien tunnisteet hintapalvelusta, joilla on maksamattomia lataustapahtumia ennen alkanutta kuukautta. Tämä pyyntö toteutetaan sitä varten tehdyllä pyynnöllä rajapinnassa. Tämän rajapinnan rakentaminen on kuvattu edellisessä luvussa. Tämän jälkeen se käy kunkin käyttäjän yksitellen lävitse, hakemalla näiden käyttäjien maksamattomat lataustapahtumat hintapalvelun rajapinnasta. Laskutettava summa lasketaan saatujen lataustapahtumien kokonaishinnoista. Palvelu hakee vanhasta järjestelmästä koodin, jonka kanssa käyttäjältä voidaan maksattaa lataustapahtuma kolmannen osapuolen palvelun kautta.

Kun saamme kolmannen osapuolen maksatuspalvelulta viestin maksun onnistumisesta, päivitämme kunkin lataustapahtuman maksetuksi rajapinnan kautta. Mikäli maksutapahtuma epäonnistuu, haemme vanhasta palvelusta käyttäjän tiedot ja kirjaamme epäonnistuneen maksatuksen tiedot sekä annetun syyn raportointiin, ennen kuin siirrymme seuraavaan käyttäjään. Vastaavasti jos maksatuksessa tapahtuu jokin muu virhetilanne, maksatus keskeytetään kyseiseltä käyttäjältä ja siirrymme seuraavaan käyttäjään. Näin pystymme käymään kaikki käyttäjät lävitse yksi kerrallaan.

6.5.3 Vanha palvelu

Palaamme nyt vanhaan palveluun ja ohjaamme sen logiikan pyytämään uusilta palveluilta latauslaitteiden ja lataustapahtumien hintatietoja tarvittaessa. Vastaavasti ohjaamme pyynnöt latauslaitteiden hintatietojen päivittämisestä uuteen palveluun. Lataustapahtuman alkaessa lisäämme sille pyynnön luoda latauksellе hintaobjektin uuteen palveluun, kun lataustapahtuma on luotu vanhaan järjestelmään. Vastaavasti lisäämme pyynnön hinnan laskemiseksi ja päivittämiseksi, kun lataustapahtuman lopetusviesti tulee ja lataus merkitään suoritetuksi.

Olemassa olevien tietojen siirtäminen uusiin palveluihin tapahtuu manuaalisesti. Kirjoitamme pienet, ajettavat koodinpätkät latauslaitteiden ja lataustapahtumien hintatiedoille, jotka käyvät vanhan tietokannan dokumentit lävitse ja siirtävät niistä tarvittavat tiedot uuden palvelun dokumenttitietokantaan. Tällä saamme varmistettua tietokantojen yhteneväisyyden tietojen osalta.

Koska maksatuspalvelu tarvitsee toimiakseen käyttäjän tietoja, luomme vanhan järjestelmän rajapintaan mahdollisuuden noutaa käyttäjän uniikilla tunnisteella tämän maksatustiedot. Kullekin lataustapahtumalle oli merkitty sen tehneen käyttäjän tunniste, josta maksatuspalvelu sen saa.

6.6 Uusien palveluiden käyttöönotto

Saatuamme uudet palvelut testattua ja näiden toiminnan varmistettua, valmistaudumme näiden käyttöönottoon. Itse käyttöönoton prosessi on harjoiteltu ja varmistettu kehitysympäristössä, joka vastaa toiminnaltaan tuotannon toimintaa, mutta on rajattu pelkästään kehitystiimin käyttöön. Tässä ympäristössä pystymme varmistamaan rajapintojen toimimisen halutulla tavalla ajamalla lataustapahtumia ja muutoksia palveluihin.

Käyttöönotto aloitetaan lukitsemalla latauslaitteiden tiedot, jotta niitä ei voi muokata kesken käyttöönoton. Tällä estetään olemassa olevan datan muuttuminen vanhentuneeksi kesken käyttöönoton. Vastaavasti jo olemassa olevien, jo päättyneiden latausten muokkaaminen estetään, vaikkakaan tätä ei tehdä kuin poikkeustapauksissa normaalisti.

Koska uutta maksupalvelua sekä hintapalvelua kumpaakaan ei käytä muut palvelut kuin vanha ohjelmisto, voidaan ne käynnistää tuotannossa valmiiksi. Kun palvelu on todettu toimivaksi, voidaan suorittaa tietokantojen tietojen siirto niihin. Tietojen siirron jälkeen pystymme varmistamaan palveluiden rajapintojen toimimisen. Kun uudet palvelut on todettu toimiviksi, ajamme päivitetyn version vanhasta palvelusta, jossa liikenne on ohjattu uusiin palveluihin.

Johtuen latausliikenteen jatkuvasta elämisestä, emme pysty jäädyttämään lataustapahtumien tietokantaa täysin. Tämä aiheuttaa haasteen käyttöönotossa,

jolloin jokaiselle lataustapahtumalle ei ole olemassa hintatietoja uudessa palvelussa. Tämä tapahtuu, jos uusi lataustapahtuma alkaa tai loppuu tietokantojen kopioimisen ja päivitetyn version käyttöönoton välillä. Totesimme tämän haasteen olevan kuitenkin sen verran pieni, että korjaamme sen mahdollisesti aiheuttamat virheet käymällä manuaalisesti tämän aikavälin lataustapahtumat ja varmistamme että jokaiselle löytyy vastine.

7 POHDINTA

Opinnäytetyön lähtökohtana oli esitellä tietoja monoliittisesta ohjelmistoarkkitehtuurimallista sekä mikropalvelun ohjelmistoarkkitehtuurimallista. Teoriaa varten kerättiin tietoa alan luennoista sekä kirjallisuudesta. Johtuen alan luonteesta tämä tieto ei kuitenkaan usein ollut akateemisesti tutkittua ja näin akateemisesti vahvistettua.

Pääosin ohjelmiston yleiset toimintamallit, kuten erilaiset arkkitehtuurimallit, eivät ole standardoituja vaan ne ovat muodostuneet kollektiivisesti useiden eri henkilöiden jaetuista mielipiteistä ja kokemuksista työelämässä sekä heidän toteuttamissaan tutkimuksissa. Näin kerätyssä ja esitetyssä tiedossa voi olla ristiriitaisuuksia aiheesta löytyvän tiedon kanssa. Kuitenkin nämä ristiriitaisuudet ovat pieniä ja eivät vaikuta kokonaiskuvaan merkittävästi.

Teoriaa kerätessä onnistuttiin keräämään niin ajan tasaisesta kirjallisuudesta kuin esitetyistä luennoista. Tällä kerätyllä teorialla saadaan pohjustettua lukijalle tarvittavat tiedot ohjelmistoarkkitehtuurien ymmärtämiseen sekä siihen miksi ja kuinka prosessit toteutettiin. Aiheen rajausta olisi voitu laajentaa suuremmaksi, ja näin laajentaa kerättyä aineistoa. Vaihtoehtoisesti rajauksen puitteissa olisi voinut hakea aineistoa, jonka avulla tietoa olisi ollut mahdollista käydä syvemmin lävitse. Näin päädytään kuitenkin helposti tilanteeseen, jossa kerätty aineisto ei enää palvele opinnäytetyön tarkoitusta.

Kiinnostus hajautettuja järjestelmiä sekä mikropalveluja kohtaan on ollut noussussa niiden vaatiman kehitysympäristön yleistyttyä. Eritoten ohjelmiston kontittamisen vaatiman tekniikoiden kehityttyä helpommaksi ja nopeammaksi käyttää on niiden suosio kasvanut suuresti. Myös ohjelmoijien kesken lisääntynyt tietoisuus niiden antamista eduista on edesauttanut näiden yleistymistä.

Esimerkkinä käytetty yrityksen ohjelmistosta ominaisuuden hajauttaminen omiksi mikropalveluiksensa on toimiva. Se ei ole oppikirjaesimerkki prosessista ja teknologioista, vaan siinä sovelletaan esitettyä teoriaa toimivan kokonaisuuden saavuttamiseksi. Opinnäytetyössä esitetyssä toteutuksessa rakennetut palvelut eivät täysin noudata esitettyjä mikropalveluiden arkkitehtuurin rajoja vaan sen toteutus

on lähempänä hajautetun järjestelmän arkkitehtuuria. Tämä on askel oikeaan suuntaan järjestelmän ylläpidon sekä jatkokehityksen suhteen, eikä tämä estä saamasta osaa mikropalveluiden tarjoamista hyödyistä.

Kokonaisuutena opinnäytetyö on onnistunut. Se jättää avoimeksi kysymyksiä osasta muista arkkitehtuurimalleista sekä osan teoriasta. Vastaavasti kuvaus käytännöstä olisi voinut olla monipuolisempi sekä tarkemmin esitetyn asian piirissä. Kuitenkin edellä mainituista ongelmista huolimatta antaa se hyvän kokonaiskuvan mikroarkkitehtuurin käytöstä ja monoliittisen ohjelmiston hajauttamisesta. Aiheesta tiedon lisäämiseksi ohjaisin lukijan tutkimaan tarkemmin muiden yritysten kohtaamia haasteita ja kuinka he ovat näitä ratkoneet omalla tahollansa.

LÄHTEET

Bogard, J. Gettting from Monolith to Microservices, Luento, Dev Up. Julkaistu: 24.2.2019. Luettu 12.5.2020. https://www.infoq.com/presentations/monolith-microservices-refactoring-analysis-tools/

Chodorow, K. 2013. MongoDB: The Definitive Guide, Second Edition. USA: O'Reilly Media Inc.

D'Souza, D.F. & Wills, A. 1998: Objects, Components, and Frameworks with UML: The Catalysis Approach. Addi-son Wesley.

Evans, J. Mastering Chaos – A Netflix Guide to Microservices. Luento, Qcon, Julkaistu: 7.12.2016. Luettu: 15.4.2020. https://www.infoq.com/presentations/netflix-chaos-microservices/

Hall, R & Pauls, K. McCulloch, S. & Savage, D. 2011. OSGI in Action, Creating Modular Applications in Java. Wiley India Pvt. Limited.

IEEE 1471-2000 Recommended Practice for Architectural Description for Software-Intensive Systems. Luettu 20.4.2020. https://standards.ieee.org/standard/1471-2000.html

Koskimies, K. & Mikkonen, T. 2005. Ohjelmistoarkkitehtuurit. Suomi: Talentum Oyj.

Mouat, A. 2016. Using Docker. USA: O'Reilly Media Inc.

Dragoni, N., Giallorenzo, S., Lluch Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L. 2017. Microservices: Yesterday, Today, and Tomorrow. Springer International Publishing.

Richardson, C. 2018. Microservices Patterns: With Examples in Java. USA: Manning Publications.

Syrjä, J. 2016. Mikropalveluarkkitehtuurit. Vaasan Ammattikorkeakoulu. Opinnäytetyö.

The InfoQ eMag: Microservices - Patterns and Practises, Online-lehti. Julkaistu: 23.3.2018. Luettu: 12.6.2020. https://www.infoq.com/minibooks/microservices-patterns-practices/

Wandschneider, M. 2013. Learning Node.js A Hands-On Guide to Building Web Applications on JavaScript. USA: Addison-Wesley Professional