Tatu Puskala

Mikropalveluarkkitehtuuri toiminnanohjausjärjestelmän toteutuksessa

Tekniikan ja innovaatiojohtamisen yksikkö
Ohjelmistotekniikka, diplomityö

Vaasa 2020

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen yksikkö
Tekijä: Tatu Puskala
Tutkielman nimi: Mikropalveluarkkitehtuuri toiminnanohjausjärjestelmän toteutuksessa
Tutkinto: Diplomi-insinööri
Oppiaine: Ohjelmistotekniikka
Työn valvoja: Jouni Lampinen
Työn ohjaaja: Magnus Sundell
Valmistumisvuosi: 2020 **Sivumäärä:** 71

TIIVISTELMÄ:

Mikropalveluarkkitehtuuri on ohjelmistoarkkitehtuuri, jossa tietojärjestelmät koostuvat pienistä itsenäisesti toimivista osista, jotka yhdessä toteuttavat järjestelmän toiminnan. Tämän diplomityön aiheena on mikropalveluiden kokeilu pienen kehitystiimin ohjelmistoprojektissa. Työ suoritetaan vaasalaiselle ohjelmistoalan yritykselle. Työn tarkoituksena on kartoittaa arkkitehtuurin vaatimia teknologioita ja toimintatapoja sekä selvittää, kannattaako arkkitehtuuria hyödyntää yrityksen tämänhetkisissä ja tulevissa ohjelmistoprojekteissa.

Työ suoritettiin testaamalla mikropalveluarkkitehtuuria pilvipalveluna toteutettavan toiminnanohjausjärjestelmän toteutuksessa. Projekti aloitettiin perinteisellä monoliittisella arkkitehtuurilla, joka oli tarkoitus siirtää myöhemmin mikropalvelupohjaiseksi. Suunnitteluratkaisuissa pyrittiin myös huomioimaan tuleva siirtymä mikropalvelupohjaiseen arkkitehtuurin. Projektin edetessä mikropalveluarkkitehtuurista luovuttiin ja sovellus toteutettiin monoliittisena. Jatkokehitystä varten kartoitettiin suunnitelma järjestelmän jatkokehityksestä mikropalvelupohjaiseksi. Järjestelmästä toteutettiin demoversio, jossa yksi tietojärjestelmän osa eristettiin omaksi mikropalvelukseen.

Projektin kohteena ollut toiminnanohjausjärjestelmä siirtyi ensimmäisen asiakkaan käyttöön monoliittisena järjestelmänä. Lisäksi toteutettiin suunnitelma sovelluksen jatkokehittelystä mikropalvelupohjaiseksi sekä testiversio mikropalvelujärjestelmästä. Mikropalveluarkkitehtuuri on aikaa vievä menetelmä etenkin siihen tottumattomalle kehitystiimille. Arkkitehtuuriin siirtymisen alkuvaiheen kustannukset ovat korkeat. Mikropalveluiden hyödyt tulevat esiin pääasiassa vasta laajoissa ja pitkäaikaisissa ohjelmistoprojekteissa. Edellytyksenä mikropalveluarkkitehtuurin käyttämiselle voidaan pitää yrityksen näkökulmasta valmiutta korkeisiin alkuvaiheen kustannuksiin sekä riittävän laajan toimialan omaavaa sovellusta. Mikropalveluita ei suositella käytettäväksi yrityksen tämänhetkisessä tilanteessa.

AVAINSANAT: Ohjelmistoarkkitehtuuri, mikropalveluarkkitehtuuri, toiminnanohjausjärjestelmä

Sisällys

1 Johdanto 8

2 Taustaa 11

2.1 Monoliittinen arkkitehtuuri 12

3 Mikropalveluarkkitehtuuri 15

3.1 Mikropalveluarkkitehtuurin vahvuudet 16

3.2 Mikropalveluarkkitehtuurin heikkoudet 18

3.3 Milloin mikropalvelut 20

4 Käytännön toteutus 24

4.1 Monoliittisella arkkitehtuurilla aloittaminen 24

4.2 Suoraan mikropalveluilla aloittaminen 25

4.3 Palvelujen mallinnus 26

4.4 Monoliitin pilkkominen 29

4.5 Integraatio 31

4.6 Testaus 33

4.7 Julkaiseminen 34

4.8 Valvonta 37

5 Sovelluksen tämänhetkinen tilanne 39

5.1 Sovelluksen vaatimukset ja ominaisuudet 39

5.2 Sovelluksen rakenne 41

5.2.1 Organization 41

5.2.2 OrganizationProxy 42

5.2.3 Employment 43

5.2.4 TimeLog 44

5.2.5 TimeRule 45

5.2.6 DrivingLog 46

5.2.7 Identity 47

5.2.8 Permission 47

5.3 CQRS 48

5.4 Docker-kontit 50
5.5 Jatkuva integraation ja toimitus 51
5.6 Nykytilanne 52
6 Monoliitin pilkkominen ja demoversio 54
6.1 Rakenteen analyysi 54
6.2 Lähestymiskohdat 56
6.3 Suunnitelma 58
6.4 Demoversio 58
7 Tulokset 61
8 Johtopäätökset 65
Lähteet 68

Termit

ACID Atomicity, Consistency, Isolation, Durability, eli atomisuus eheys, eristyneisyys ja pysyvyys. Tietokantatransaktioiden ominaisuudet, joilla pyritään varmistamaan tietokannan eheys myös vikatilanteissa.

API Application Programming Interface, Ohjelmistorajapinta, määritelmä, jonka mukaan ohjelmat voivat kommunikoida keskenään.

Back end Palvelimen puolella toimiva osa sovellusta.

Callback-funktio Ohjelmointikielen funktio, joka annetaan parametriksi toiselle funktiolle.

CD Continuous Delivery, Jatkuva toimitus, menetelmä, joka mahdollistaa sovelluksen julkaisuemisen tuotantoon helposti.

CI Continuous Integration, Jatkuva integraatio, menetelmä, jossa jokainen versiohallinnan tietovarastojen säiltytyspaikkaan tehty muutos aiheuttaa testien ajamisen.

CQRS Command Query Responsibility Segregation, suunnitteluperiaate, jossa sovelluksen luku- ja kirjoitusmallit erotetaan toisistaan.

CRUD Create, Read, Update and Delete, relatatiotietokannan perusoperaatiot: tietueiden luominen, lukeminen, päivittäminen sekä poistaminen.

Docker Työkalu sovellusten luomiseen, julkaisemiseen ja ajamiseen ohjelmistokonteissa.

Docker-kontti Docker-työkalun käyttämä ohjelmistokontti.

Front end Sovelluksen käyttäjän puolella toimiva osa sovellusta.

HTTP Hypertext Transfer Protocol, selainten ja WWW-palveluiden käyttämä viestintäprotokolla.

Hypervisor Ohjelmisto, laiteohjelmisto, tai laitteisto, joka luo ja ajaa virtuaalikoneita.

Laravel Avoimen lähdekoodin web-ohjelmointiin tarkoitettu PHP-pohjainen sovelluskehys.

Lumen Laravel-pohjainen erityisesti mikropalveluiden toteuttamiseen tarkoitettu sovelluskehys.

Ohjelmistokontti, kontti Standardoitu ympäristö, jossa sovellus voidaan ajaa.

ORM Object Relational Mapping, ohjelmointitekniikka, joka mahdollistaa datan lukemisen ja kirjoittamisen tietokantaan olioperusteisen ohjelmointikielen olioiden kautta.

PHP PHP: Hypertext Preprocessor, erityisesti verkkosovellusten kehitykseen soveltuva komentokieli.

REST Representational State Transfer, arkkitehtuuri verkkopalveluiden rajapinnoille.

SaaS Software as a Service, keskitetyssä tuotantoympäristössä toimiva sovellus, jota tarjotaan asiakkaille palveluna.

Kuviot

Kuva 1. Esimerkki monoliittisen järjestelmän arkkitehtuurista komponentti- ja liitinnäkymästä. (Annett 2014) 12
Kuva 2. Tavanomainen monoliittisen ohjelman rakenne (Fowler, 2016) 13
Kuva 3. Mikropalvelupohjaisen ja monoliittisen arkkitehtuurin vaikutus tuottavuuteen (Fowler, 2015a). 20
Kuva 4. Saman sovelluksen kaksi mallia rajatuissa konteksteissaan (Fowler, 2014). 27
Kuva 5. Karkeajakoinen 'Warehouse' rajattu konteksti piilottaa sisäiset mikropalveluna ulkomaailmalle (Newman, 2015a) 28
Kuva 6. Virtuaalikoneiden ja konttien erot (Merkel, 2014). 36
Kuva 7. Mikropalveluiden valvontajärjestelmä (Richardson, 2018). 38
Kuva 8. Organization-moduulin rakenne. 41
Kuva 9. OrganizationProxy-moduulin rakenne. 42
Kuva 10. Employment-moduulin rakenne. 43
Kuva 11. TimeLog-moduulin yksinkertaistettu malli 44
Kuva 12. TimeRule-moduulin yksinkertaistettu malli. 45
Kuva 13. DrivingLog-moduulin domain-malli. 46
Kuva 14. Permission-moduulin rakenne. 47
Kuva 15. Sovelluksen kirjoitusmalli. 49
Kuva 16. Sovelluksen lukumalli. 50
Kuva 17. CI/CD-ketjun tuotantotyökulku. 52

1 Johdanto

Ohjelmistokehitys on nuori ja nopeasti muuttuva ala. Ohjelmistoille esitetään yhä kasvavia vaatimuksia. Teknologian kehitys auttaa omalta osaltaan vastaamaan kasvaviin haasteisiin. Skaalautuvien, hajautettujen järjestelmien toteuttaminen on nykyään huomattavasti helpompaa kuin aiemmin. Toisaalta jatkuvasti uudistuvan teknologian mahdollisuuksien tehokas hyödyntäminen vaatii erilaisia toimintatapoja myös ohjelmistokehityksessä.

Perinteisessä monoliittisessa arkkitehtuurissa sovellus ajetaan yhdessä prosessissa usein yhteen tietokantaan integroituna. Monoliittisen arkkitehtuurin ongelmat tulevat esiin pitkäaikaisissa, suurikokoisissa projekteissa. Hyviä suunnitteluperiaatteita noudattaenkin isojen projektien kehityskustannukset kasvavat ja toimintavarmuus heikkenee (Newman, 2015). Teknologian kehittyessä ja kehitystiimien eläessä vanhaan teknologiaan sidotut pitkäikäiset ohjelmistoprojektit paisuvat helposti vaikeasti muutettaviksi, rakenteeltaan epäkiinteiksi (*uncohesive*) ohjelmiksi, joiden ylläpidosta ja päivittämisestä tulee jatkuva taakka.

Mikropalveluarkkitehtuuri on ohjelmistoarkkitehtuurin suunta, joka pyrkii ratkaisemaan monoliittisen arkkitehtuurin ongelmia. Mikropalveluarkkitehtuurissa ohjelmat koostuvat mikropalveluista. Mikropalvelut ovat hienojakoisia, itsenäisesti julkaistavia ja omassa prosessissaan suoritettavia ohjelmia (Lewis, 2015). Mikropalvelut kommunikoivat standardoituja data- ja viestintäprotokollia sekä julkaistavia rajapintoja käyttäen. (Tyszberowics, 2018).

Mikropalveluajattelun ytimessä on skaalautuvuus sekä sovelluskehityksen ketteryys. Itsenäisten ja hienojakoisten mikropalvelujen skaalaus on tehokkaampaa kuin monoliittisen sovelluksen (Newman, 2015a). Eavesin (2014) mukaan hyvin toteutetun mikropalvelun täydelleen uusiksi kirjoittamiseen tarvittava aika mitataan viikoissa. Laajan mikropalvelupohjaisen ohjelmiston muuttaminen ja ylläpito on näin ollen huomattavasti halvempaa kuin monoliittisen.

Tämä työ tehdään toimeksiantona vaasalaiselle ohjelmistoalan yritykselle Black Label Bytes Oy:lle. Työn tekijä on työsuhteessa yritykseen. Yritys on toiminut alalla muutaman vuoden. Yrityksen sovelluskehitystiimi koostuu neljästä työntekijästä, joista kaksi jäsentä on kokeneempia ja toiset kaksi vähemmän kokeneita sovelluskehittäjiä. Tiimillä ei ole aikaisempaa kokemusta mikropalveluista. Työssä keskitytään mikropalveluarkkitehtuuriin toiminnanohjausjärjestelmän toteuttamisprojektissa. Järjestelmä toteutetaan modulaarisena, eri asiakkaiden tarpeisiin mukautettavana pilvipalveluna. Projektia käytetään mikropalveluarkkitehtuurin kokeilualustana.

Työn alkuperäisenä tavoitteena oli tuottaa ensimmäinen versio case-projektina käytetystä toiminnanohjausjärjestelmästä mikropalvelupohjaisena. Tavoitteesta kuitenkin luovuttiin aikataulupaineiden sekä mikropalveluarkkitehtuurin kannattavuuden epävarmuuden takia. Sen sijaan selvitettiin monoliittisena kehitetyn järjestelmän mikropalvelupohjaiseksi siirtämisen edellytykset ja kannattavuus. Selvityksen lisäksi kokonaisen mikropalvelupohjaisen järjestelmän kehittämisen sijaan tavoitteeksi otettiin pienimuotoisen demoversion toteuttaminen. Demoversiossa pieni osa monoliittisena toteutettua toiminnanohjausjärjestelmää erotetaan omaksi mikropalvelukseen.

Työn tutkimuskysymykset ovat:

1. Kannattaisiko yrityksen ottaa käyttöön mikropalveluarkkitehtuuri case-projektissa tai samankaltaisissa tulevissa projekteissa?

2. Mitä käytännön asioita on huomioitava mikropalvelupohjaista järjestelmää kehittäessä?

Työn toteutus voidaan jakaa kahteen vaiheeseen. Ensimmäisessä vaiheessa suunnitteilla oli vielä koko sovelluksen toteutus mikropalvelupohjaisena. Tässä vaiheessa toiminnanohjausjärjestelmää kehitettiin aluksi monoliittisena pyrkien kuitenkin siihen, että se olisi

helposti siirrettävissä mikropalvelupohjaiseksi. Mikropalveluarkkitehtuuria varten otettiin käyttöön myös useita sen edellyttämiä infrastruktuuriratkaisuja. Kun ajatuksesta toteuttaa järjestelmä tuotantoon mikropalvelupohjaisena luovuttiin, koodikantaa ei ollut enää tarvetta pitää helposti mikropalvelupohjaiseksi siirrettävänä.

Toinen vaihe tehtiin vajaa vuosi sen jälkeen, kun järjestelmä päätettiin toteuttaa monoliittisena. Mahdollisen siirtymän edellytykset kartoitetaan. Sovelluksesta toteutetaan myös demoversio, jossa yksi osa palvelua irrotetaan omaksi mikropalvelukseen. Demoversiossa havainnollistetaan ja kokeillaan arkkitehtuurin vaatimia teknisiä ratkaisuja.

Työn teoriaosuus käsittää luvut kaksi, kolme ja neljä. Luvussa kolme käsitellään yleistä teoriaa mikropalveluista, niiden hyvistä ja huonoista puolista sekä tilanteista, jolloin mikropalveluita kannattaa hyödyntää. Luvussa neljä käsitellään mikropalveluarkkitehtuurin käytännön toteutukseen ja vaatimuksiin liittyvää teoriaa. Luvussa viisi esitellään projektin ensimmäisessä vaiheessa tehty työ. Luvussa kuusi esitellään suunnitelma järjestelmän siirtämisestä mikropalveluksi sekä mikropalvelupohjaisen järjestelmän demoversio. Luvuissa seitsemän ja kahdeksan esitellään työn tulokset ja tärkeimmät johtopäätökset.

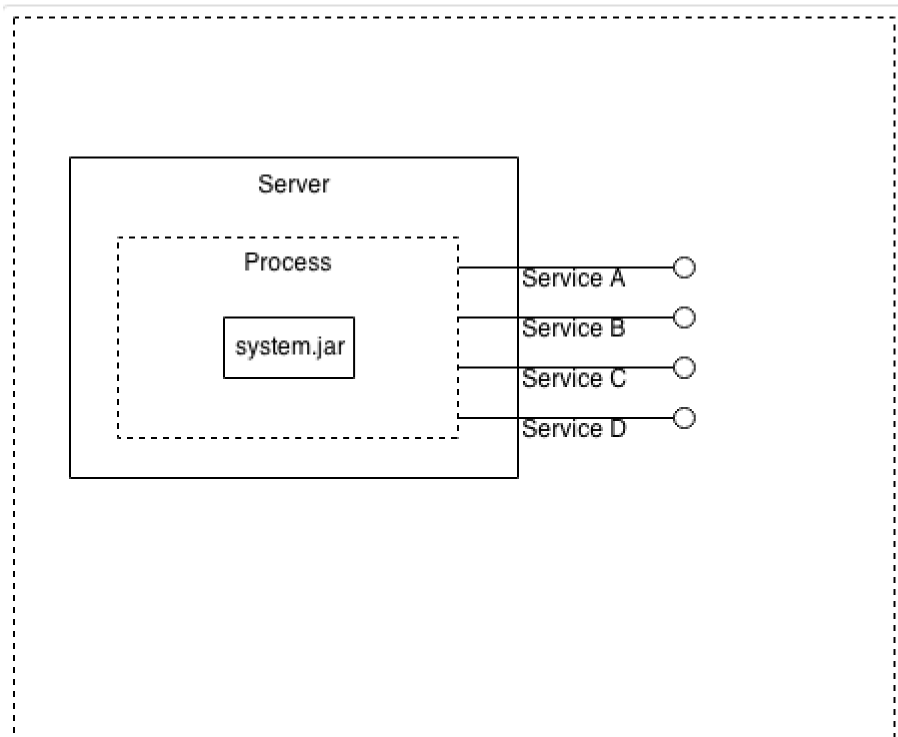
2 Taustaa

IEEE:n standardi määrittelee ohjelmistoarkkitehtuurin järjestelmän korkean tason organisaatioksi, joka koostuu järjestelmän osista, näiden osien välisistä suhteista ja järjestelmän suunnittelun ja evoluution periaatteista (IEEE 2000). Käytännössä termiä käytetään vaihtelevasti kontekstin mukaan. Koskimiehen ja Mikkosen (2005) mukaan arkkitehtuuri ei välttämättä suoraan määritä koodikannan rakennetta, vaan se voi olla myös olla hyvin korkean tason dokumentti järjestelmän suunnitteluperiaatteista. Fowler (2003) pitää ohjelmistoarkkitehtuuria sovelluskehittäjien jaettuna näkemyksenä järjestelmän suunnittelusta. Tähän näkemykseen kuuluu se, miten järjestelmä jaetaan komponentteihin ja miten komponentit ovat vuorovaikutuksessa keskenään. Arkkitehtuurin määrittävät komponentit koostuvat käytännössä usein pienemmistä komponenteista, mutta arkkitehtuuri ei määritä näitä.

Clement, Garlan, Bass ja Stafford (2010) määrittelvät ohjelmistoarkkitehtuurille abstraktiotason mukaan eri **näkymiä**. Näkymät koostuvat elementeistä ja näiden välisestä vuorovaikutuksesta. Koodikannan rakennetta määrittelevää näkymää kutsutaan moduulinäkymäksi.

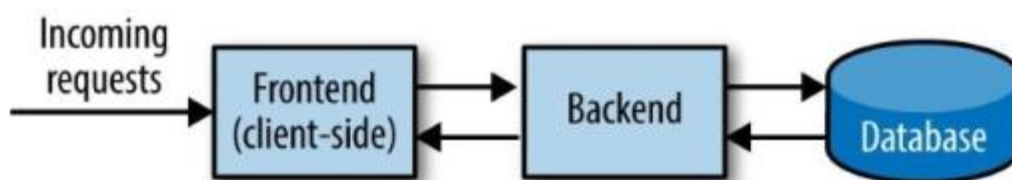
Diplomityössä keskitytään ohjelmistoarkkitehtuuriin pääasiassa komponentti ja liitin -tason näkymässä. Tässä näkymässä komponentit ovat järjestelmän ajonaikaisia prosesseja ja liittimet kuvaavat näiden komponenttien välistä kommunikaatiota (Clement et al., 2010). Kuvassa 1 on kuvattu monoliittisen sovelluksen back end -puoli komponentti ja liitin -tason näkymässä.

2.1 Monoliittinen arkkitehtuuri



Kuva 1. Esimerkki monoliittisen järjestelmän arkkitehtuurista komponentti- ja liitinnäkymästä. (Annett 2014)

Monoliittisessa järjestelmässä (myöh. monoliitti) järjestelmän oma koodikanta toimii yhtenä kokonaisuutena (Fowler, 2016). Kuvassa 2 havainnollistetaan tavanomaisen monoliittisen sovelluksen rakennetta, joka on jaettu front end ja back end -puoleen sekä tietokantaan. Front end ja back end yhdistetään yleensä samaan koodikantaan. Front end -puoli lähettää pyyntöjä back end -puolelle, joka toteuttaa ohjelman varsinaisen logiikan. Erillinen tietokanta suoritetaan omassa prosessissaan. Mikäli järjestelmä ei tarvitse erillistä tietokantaa, voidaan myös tiedon varastointi suorittaa samassa prosessissa muistia käyttäen.



Kuva 2. Tavanomainen monoliittisen ohjelman rakenne (Fowler, 2016)

Monoliittinen arkkitehtuuri on perinteinen tapa toteuttaa ohjelmistoja. Kehitystyökalut on suunnattu monoliittisten ohjelmien toteuttamiseen (Richardson, 2013). Kehittäjät ovat myös tottuneet ohjelmoimaan monoliittisia ohjelmia. Muut monoliittisen arkkitehtuurin edut liittyvät sen yksinkertaisuuteen. Testaus on helppoa, sillä testausta varten tarvitsee suorittaa vain yksi ohjelma. Monoliitin julkaisu on yksinkertaista, suoritettava tiedosto tai kansio vain kopioidaan palvelimelle.

Ohjelmia voidaan skaalata vertikaalisesti tai horisontaalisesti. Horisontaalisessa skaalauksessa ohjelma ajetaan tehokkaammalla palvelimella (Fowler, 2016). Monoliitin horisontaalinen skaalaus toteutetaan ajamalla kopioita ohjelmasta eri palvelimilla. Ainoa tarvittava lisäosa on kuormantasaaja, joka jakaa pyynnöt eri palvelimille (Richardson, 2013).

Monoliitin ongelmat tulevat esille useimpien ohjelmien elinkaaren aikana. (Fowler, 2016). Muutosten tekeminen vaikeutuu koodikannan kasvaessa. Laajassa koodikannassa pienikin muutos voi aiheuttaa odottamattomia ongelmia. Toiminnan varmistaminen vaatii suurta määrää integraatiotestejä. Ongelmia pyritään ehkäisemään käyttämällä suunnitteluperiaatteita, jotka kasvattavat koodin kiinteyttä (*cohesion*) (Newman, 2015a). Kiinteään rakenteeseen pyritään abstraktioilla ja ryhmittämällä toisiinsa koodin liittyvät osat moduuleihin.

Rodgerin (2018) mukaan lähes kaikissa ohjelmistoprojekteissa joudutaan aika-ajoin joustamaan hyvistä suunnitteluperiaatteista, esimerkiksi aikataulupaineiden takia. Tästä aiheutuvaa eroa suunnitellusta ohjelmiston tasosta käytäntöön verrattuna kutsutaan tek-

niseksi velaksi. Velka olisi tarkoitus kuroa umpeen refaktoroimalla. Käytännössä pitkäaikaisissa monoliittisissa projekteissa tekninen velka kuitenkin kasautuu ja vaikeuttaa entisestään ohjelmiston jatkokehitystä.

Ohjelmiston kasvaessa kehittäjien on yhä vaikeampi ymmärtää laajaa, paljon riippuvuussuhteita sisältävää koodikantaa (Richardson, 2018). Uusilla kehittäjillä kestää kauan päästä projektiin sisään. Rakennetta heikosti ymmärtävät kehittäjät tekevät huonoja suunnitteluratkaisuja, jotka vaikeuttavat koodikannan ymmärtämistä entisestään. Tämä johtaa kierteeseen, joka tekee kehitystyöstä projektin edetessä aina vain hitaampaa.

Richardsonin (2018) mukaan monoliitin skaalaus on tehotonta. Ohjelman osien vaatimat resurssit vaihtelevat. Jotkin osat saattavat tarvita suurta muistimäärää, kun jotkut taas vaativat tehokkaampaa suoritinta. Vertikaalinen skaalaus joudutaan siis tekemään näiden eniten resursseja syövien ohjelman osien ehdoilla. Horisontaalisessa skaalauksessa koko ohjelma joudutaan kopioimaan muutaman enemmän resursseja vaativan osan takia.

Monoliittisen ohjelman julkaisussa koko ohjelma joudutaan julkaisemaan kerralla (Richardson, 2018). Suuret, paljon muutoksia sisältävät julkaisut ovat riskialttiita ja hitaita. Newmanin (2015a) mukaan tämä johtaa pidempiin julkaisuväleihin. Julkaisuvälien kasvaessa julkaisujen koot kasvavat entisestään, mikä taas kasvattaa riskejä entisestään.

Monoliittisen ohjelman ongelmat liittyvät siis kokoon ja monimutkaisuuteen. Ongelmat eivät tule esille pienissä ohjelmissa ja projektien alkuvaiheessa. Koodikannan kasvaessa kasvavat myös sekä ohjelman kehitys- että ylläpitokustannukset.

3 Mikropalveluarkkitehtuuri

Rodger (2018) määrittelee ajattelutavan ohjelmasta komponenteista koottavaksi kokonaisuudeksi mikropalveluajattelun takana olevaksi perusideaksi. Mikropalveluarkkitehtuurissa nämä komponentit ovat mikropalveluita. Jokainen näistä mikropalveluista voidaan julkaista erikseen, ja se toimii omassa prosessissaan (Newman, 2015a). Mikropalvelut kommunikoivat keskenään API-kutsuilla verkon yli.

Lewisin ja Fowlerin (2014) mukaan mikropalveluarkkitehtuurille ei ole yksityiskohtaista määritelmää. He kuitenkin määrittelevät ominaisuuksia, jotka yhdistävät useimpia mikropalvelupohjaisia arkkitehtuureja:

Palveluiden ja kehitystiimien jakaminen toimialan mukaan. Monissa organisaatioissa kehitystiimit jaetaan teknologioiden mukaan esimerkiksi front end-, back end- ja tietokantatiimeihin. Mikropalveluarkkitehtuurissa kehitystiimit vastaavat koko palvelustaan. Myös palvelut jaetaan mallinnettavien toimialan toiminnallisuuksien mukaan.

Ohjelmien ajattelu tuotteina eikä projekteina. Ohjelmistokehitystä ei ajatella projektina, jolla on alku ja selvä loppu. Kehittäjät ovat vastuussa ohjelmasta koko sen elinkaaren ajan.

Evolutiivinen suunnittelu. Ohjelmalle ei suunnitella etukäteen tiukkaa arkkitehtuuria, vaan ohjelmaa ajatellaan jatkuvasti kehittyvänä prosessina. Muutos nähdään olennaisena osana ohjelmistojen kehitystä. Ohjelmat kehitetään helposti muutettavaksi.

Hajautettu datan hallinta. Tavoitteena on, että jokainen mikropalvelu sisältää oman tietokantansa. Mikropalvelut voivat myös käyttää eri tietokantateknologioita.

Hajautettu hallinnointi. Pyrkimyksenä on, että palveluiden sisäistä toteutusta ei hallita. Mikropalveluiden kehittäjät voivat vapaasti valita teknologian, jolla palvelu toteutetaan.

Kevyet kommunikaatiomekanismit. Mikropalveluiden välinen kommunikaatio pyritään toteuttamaan yksinkertaisilla ja kevyillä mekanismeilla. Ohjelman ydinlogiikka toteutetaan palveluissa.

Infrastruktuurin automaatio. Suuren ja monimutkaisen mikropalveluiden verkon integroiminen helpottuu huomattavasti automatisoituja infrastruktuuriratkaisuja käyttämällä. Jatkuva integraatio (*CI*, continous integration) ja toimitus (*CD*, continous delivery) yksinkertaistavat mikropalveluiden julkaisemista ja testaamista. Arkkitehtuurin käyttäjät hyödyntävät automatisoituja ratkaisuja myös helpottamaan mikropalveluekosysteemin pyörittämistä tuotannossa.

Vikatilanteihin varautuminen. Ohjelman koostuessa erillisistä mikropalveluista tulee varmistaa, että yhden palvelun rikkoutuminen vaikuttaa mahdollisimman vähän muiden palveluiden toimintaan. Tämä edellyttää ylimääräistä testausta ja tuotannossa toimivien palveluiden valvontaa.

3.1 Mikropalveluarkkitehtuurin vahvuudet

Conwayn lain mukaan järjestelmän rakenne vastaa sen rakentajien organisaation rakennetta (Conway, 1968). Mikropalveluarkkitehtuurissa palvelujen pienikokoisuus mahdollistaa palvelujen kehityksen rajoittamisen yhteen kehitystiimiin (Newman, 2015a). Pienen kehitystiimin sisäisen kommunikaation vaivattomuus mahdollistaa jäsenten jatkuvan ja hienojakoisen vuorovaikutuksen. Tällä tavalla palveluiden sisäisestä ohjelmakoodista tulee kiinteää ja palveluista löyhästi toisiinsa kytkettyjä.

Ohjelman jakaminen pienikokoisiin mikropalveluihin auttaa pitämään koodikannan yksinkertaisena ja ymmärrettävänä (Richardson, 2018; Rodger, 2018). Mikropalvelut koteloivat (*encapsulate*) koodikantansa. Ainoa tapa kutsua mikropalvelua on verkon kautta.

Arkkitehtuuri estää modulaarisen rakenteen rikkomisen ja ehkäisee teknisen velan kertymistä. Mikropalvelut siis nopeuttavat kehitystyötä ja tätä kautta vähentävät kehitystyön kustannuksia laajoissa ohjelmistoprojekteissa.

Edellä mainitut tekijät tekevät myös mikropalvelupohjaisista järjestelmistä helposti muunneltavia. Kynnys pienikokoisten palveluiden uudelleenkirjoittamiseen tai poistamiseen on huomattavasti matalampi kuin monoliittisessa arkkitehtuurissa (Newman, 2015a). Arkkitehtuuri helpottaa ohjelmistojen muuttuviin vaatimuksiin vastaamista (Rodger, 2018).

Mikropalvelut mahdollistavat usean eri teknologian käyttämisen samassa ohjelmistossa (Newman, 2015a). Teknologinen heterogeenisuus antaa kehittäjille vapauden käyttää parhaiten osaamiaan ja parhaiten kuhunkin palveluun sopivia teknologioita. Esimerkiksi yksi palvelu voi käyttää dokumenttitietokantaa ja funktionaalista ohjelmointikieltä, toinen relaatiotietokantaa ja olioperusteista ohjelmointikieltä. Käytännössä monet yritykset ovat rajoittaneet tätä vapautta.

Mikropalvelut skaalautuvat tehokkaasti (Newman, 2015a). Hienojakoisista mikropalveluista koostuva järjestelmä voidaan skaalata kopioimalla vain eniten resursseja syövät palvelut. Toisaalta Fowler (2015c) ei pidä tätä hienojakoista skaalaamista erityisen hyödyllisenä. Newman käyttää esimerkkinä skaalauksesta Netflixiä ja Giltiä, jotka ovat erittäin suuria yrityksiä. Voidaan olettaa, että suurin osa skaalaamisen tehostamisen hyödyistä jää Netflixin kaltaisille erittäin suurten käyttäjämäärien ohjelmistojen kehittäjille.

Kuten edellä mainitaan, mikropalveluarkkitehtuuri edellyttää uudenlaisiin vikatilanteisiin varautumista. Kuitenkin mikropalvelut estävät katastrofaaliset vikatilanteet, joissa koko ohjelman toiminta pysähtyy yhden ohjelmointivirheen takia (Newman, 2015a). Vaikka yksi palvelu menisikin rikki, muut palvelut pystyvät enimmäkseen jatkamaan toimintaansa.

Richardsonin (2018) mukaan mikropalveluarkkitehtuurin suurin hyöty on isojen ja monimutkaisten ohjelmien jatkuvan julkaisun mahdollistaminen. Mikropalvelut pystytään julkaisemaan itsenäisesti. Pienikokoiset julkaisut nopeuttavat julkaisun yhteydessä ajettavia automaattisia testejä. Mikäli julkaisun jälkeen mikropalvelussa ilmenee ongelmia, pystytään palvelu palauttamaan nopeasti edelliseen versioon (Newman, 2015a). Nopeat ja vähemmän riskialttiit julkaisut nopeuttavat valmiitten toimintojen siirtämistä tuotantoon.

3.2 Mikropalveluarkkitehtuurin heikkoudet

Suuri osa mikropalveluarkkitehtuurin huonoista puolista liittyy hajautettujen järjestelmien mukanaan tuomaan monimutkaisuuteen (Newman, 2015a; Fowler, 2015c; Richardson, 2018). Hajautetuissa järjestelmissä on enemmän osia, jotka voivat mennä rikki.

Hajautetut järjestelmät joutuvat ottamaan huomioon verkon tuomat haasteet (Rotem-Gal-Oz, 2006). Verkko on epäluotettava: fyysisen laitteiston vikatilanteet voivat johtaa viestien katoamiseen tai viivästymiseen. Verkossa esiintyy viivettä ja verkon yli tapahtuva kommunikaatio tuo mukanaan turvallisuusongelmia.

Mikropalvelujen välisiin kutsuihin perustuva kommunikaatio on metodikutsuja hitaampaa ja edellyttää erillisen kommunikaatiomekanismin käyttöä (Lewis ja Fowler, 2014; Richardson, 2018). Lewisin ja Fowlerin mukaan verkon viiveen ongelmat tulevat vahvasti esiin juuri mikropalveluarkkitehtuurissa palveluiden suuren määrän takia. Hienojakoiset mikropalvelut tekevät paljon kutsuja toistensa välillä. Kommunikaation hitauden ongelmia pystytään vähentämään käyttämällä karkeajakoisimpia päätepisteitä, mikä vähentää verkon yli tehtävien kutsujen määrää. Tämä kuitenkin asettaa rajoitteita mikropalveluiden kehittäjille.

Koska data on varastoitu useaan tietokantaan, tietokantojen integriteettiä ei voida varmistaa transaktioilla (Fowler, 2015c). Hajautetusta tietomallista ja verkon epäluotettavuudesta johtuen kehittäjät joutuvat turvautumaan ennen pitkää saavutettavaan eheyteen (*eventual consistency).* Ennen pitkää saavutettavassa eheydessä taataan datan korkea saatavuus luopumalla tietokannan ACID-periaatteen tarjoamasta kovasta eheydestä (*strong consistency*). Kovasti eheässä tietokannassa tietokantakyselyt palauttavat aina uusimman version datasta, kun taas ennen pitkää saavutettavassa eheydessä tietokantaan tehdyt muutokset näkyvät ennemmin tai myöhemmin. Ennen pitkää saavutettava eheys voi johtaa viiveisiin järjestelmän toiminnassa, mikä joudutaan ottamaan huomioon järjestelmää kehitettäessä.

Toisistaan riippuvaisten mikropalveluiden julkaisu vaatii koordinointia (Richardson, 2018). Useita mikropalveluja käyttävien ominaisuuksien julkaisu edellyttää näiden palveluiden julkaisemista samalla kertaa. Mikropalveluita päivitettäessä tulee ottaa huomioon palveluiden vanhoista versioista riippuvaiset palvelut (Newman, 2015a). Päivitys voi aiheuttaa ongelmia vanhasta versiosta riippuvaisissa palveluissa. Näiden ongelmien vuoksi järjestelmän toiminnan varmistaminen julkaisujen yhteydessä edellyttää mikropalveluiden välistä integraatiotestausta ja versioinnin hallinnoimista.

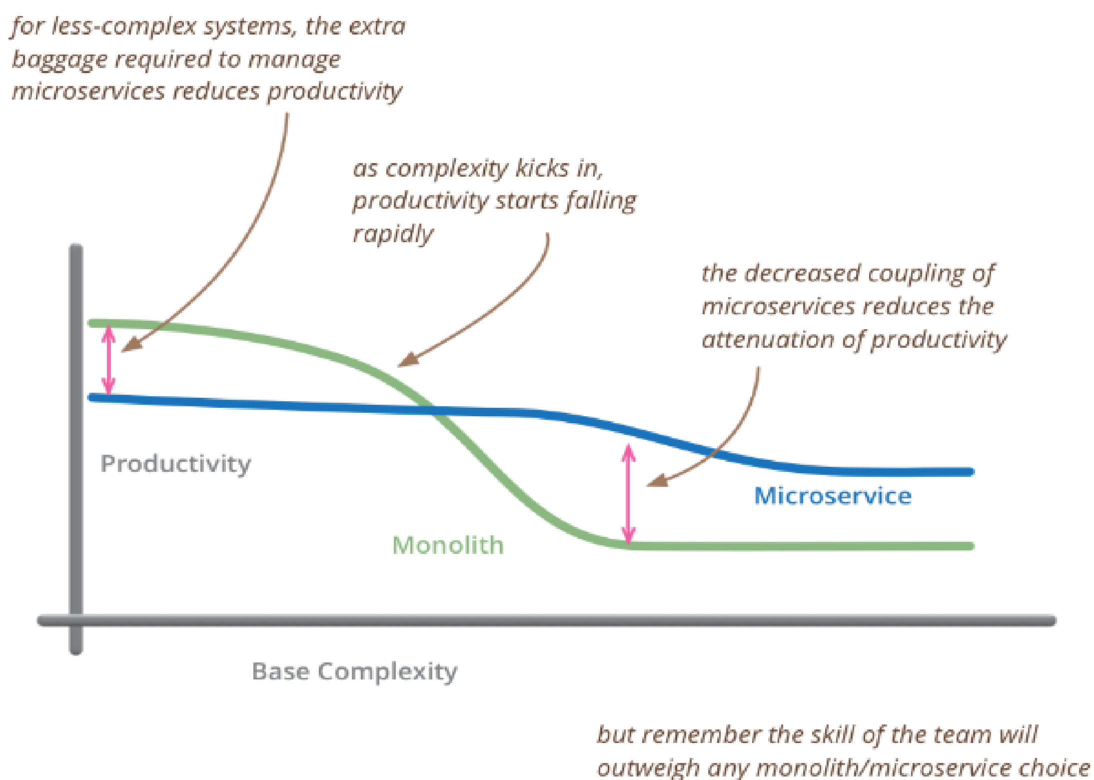
Operationaalinen kompleksisuus kasvaa suuren palvelumäärän julkaisemisen, ylläpidon ja hallinnoinnin myötä (Lewis ja Fowler, 2014). Mikropalveluarkkitehtuuri siirtää monimutkaisuuden itse mikropalveluista niiden välisiin yhteyksiin. Useita palveluita koskevien vikatilanteiden selvitys on hankalaa. Monimutkaisuutensa vuoksi mikropalveluiden hallinto ja ylläpito edellyttää automaatiota.

Järjestelmän pilkkominen mikropalveluiksi on haastavaa (Richardson, 2018). Väärin määritellyt mikropalveluiden rajat johtavat tiukasti toisiinsa kytkettyihin mikropalveluihin. Tällainen järjestelmä kärsii sekä monoliittisen että mikropalvelupohjaisen järjestelmän ongelmista.

3.3 Milloin mikropalvelut

Brooksin (1987) mukaan ohjelmistokehityksessä ei ole yhtä yksittäistä ratkaisua, joka kasvattaisi sovellusten tuottavuutta, luotettavuutta ja yksinkertaisuutta merkittävästi. Mikropalveluarkkitehtuuri ei ole poikkeus tästä säännöstä (Richardson, 2018). Mikropalveluarkkitehtuuria harkittaessa tulee punnita edellä mainittuja arkkitehtuurin hyviä ja huonoja puolia. Hyödyt ja haitat painottuvat eri tavoin yrityksen kehitystiimistä ja kehitettävästä ohjelmasta riippuen.

Mikropalveluarkkitehtuuri pyrkii ratkaisemaan monoliittisen arkkitehtuurin ongelmia. Nämä ongelmat eivät kuitenkaan esiinny kaikissa monoliittisissa järjestelmissä, joten on olennaista selvittää, millaiset järjestelmät hyötyvät mikropalvelupohjaisesta toteutuksesta.



Kuva 3. Mikropalvelupohjaisen ja monoliittisen arkkitehtuurin vaikutus tuottavuuteen (Fowler, 2015a).

Fowlerin (2015a) mukaan mikropalveluarkkitehtuuria ei tule edes harkita, jos järjestelmä ei ole niin laaja, että sen hallitseminen monoliittisena ei tuota suuria vaikeuksia. Mikropalveluarkkitehtuurin tuomat vaihtoehtoiskustannukset ylittävät sen hyödyt suurimmassa osassa järjestelmiä. Kuvassa 3 verrataan monoliittisen ja mikropalveluarkkitehtuurin vaikutuksia tuottavuuteen sovelluksen kompleksisuuden mukaan: yksinkertaisemmissa sovelluksissa monoliitti on nopeampi, mutta monimutkaisuuden kasvaessa mikropalveluarkkitehtuuri on nopeampi.

Mikropalveluarkkitehtuuria harkittaessa tulisi toteutettavan järjestelmän toimiala tuntea hyvin (Newman, 2015a). Riski palvelurajojen väärin määrittelystä ennestään tuntemattomalla toimialalla on suuri.

Koska mikropalveluarkkitehtuurin skaalauksen tehokkuus perustuu palvelujen hienojakoisuuteen, saavutettava hyöty monoliittiin verrattuna riippuu ohjelman koosta sekä kuormituksen jakautumisesta. Mitä tasaisemmin järjestelmän kuormitus jakautuu eri osien kesken, sitä vähemmän mikropalvelut tehostavat skaalautumista. Pienikokoista monoliittia skaalattaessa ylimääräinen kopioitava osa jää pieneksi.

Mikropalveluarkkitehtuurin mahdollistama nopea julkaisu hyödyttää SaaS-pohjaisia järjestelmiä (Singleton, 2016). Päivitykset saadaan viiveettä tuotantojärjestelmään. SaaS-pohjaisessa järjestelmässä kehittäjät pystyvät kontrolloimaan tuotantoympäristöä ja julkaisuprosessia (Newman, 2015b). Tämä mahdollistaa järjestelmän asennuksen automatisoinnin toimintavarmasti. Mikäli järjestelmä toimitetaan ja ajetaan asiakkaiden omissa tuotantoympäristöissä, menetetään jatkuvan julkaisun edut. Mikropalvelupohjaisen järjestelmän asentaminen ja ylläpito eri asiakkaiden vaihtelevissa tuotantoympäristöissä on vaikeaa.

Singletonin (2016) mukaan arkkitehtuurin käyttäminen ei ole pääsääntöisesti kannattavaa alle 60 kehittäjän organisaatioissa. Mikäli kehitystiimi on niin pieni, että sitä ei kannata jakaa edelleen pienempiin tiimeihin, menetetään arkkitehtuurin organisaatiotason hyödyt. Lisäksi suurissa, pienempiin tiimeihin jaetuissa kehitystiimeissä koodin yhdistäminen julkaisua varten aiheuttaa usein lisätyötä. Yhdistämisen konflikteja ratkaistaessa on myös mahdollista, että jokin menee vikaan.

Kehitystiimin kannalta mikropalveluarkkitehtuurissa on jyrkkä oppimiskäyrä (Buchgeher, 2017).

Toisaalta verkkopalvelupohjaisuus ja suuri yrityskoko eivät ole elinehto mikropalveluarkkitehtuurin kannattavuudelle, mikäli muut tekijät puoltavat arkkitehtuuria. Buchgeher (2017) esittää tästä esimerkkinä AMS Engineering -yrityksen kehittämän laboratorioautomaatiojärjestelmän. Järjestelmä toimitettiin erikseen jokaiselle asiakkaalle, jotka vastasivat asennuksen jälkeisestä järjestelmän ylläpidosta. Järjestelmältä edellytetyt vaatimukset vaihtelivat suuresti asiakkaiden välillä. Mikropalveluarkkitehtuuriin päädyttiin sen tarjoaman suuren joustavuuden ja muokattavuuden takia.

AMS Engineeringin kehitystiimi koostui kahdeksasta sovelluskehittäjästä. Sovelluksesta luotiin pilottiversio, jossa otettiin käyttöön tarvittava infrastruktuuri ja luotiin muutama mikropalvelu. Pilottiversion toteutti kolmen kehittäjän tiimi. Kehittäjät olivat kokeneita, mutta heillä ei ollut aikaisempaa kokemusta mikropalveluarkkitehtuurista. Esimerkissä huomattavaa on arkkitehtuurin yritykselle aiheuttamat alkuvaiheen kustannukset. Yritys arvioi pelkästään vaaditun infrastruktuurin asentamisen vaatineen 2 500 työtuntia. Toisaalta tämän jälkeen ensimmäisen liiketoiminta-arvoa tuottaneen mikropalvelun kehittämiseen meni vain 100 tuntia.

Edellä mainitusta esimerkistä havaitaan, että suuri osa mikropalveluihin liittyvistä kustannuksista sijoittuu arkkitehtuurin käyttöönoton alkuvaiheeseen. Kun palveluiden julkaisun, valvonnan ja hallinnon automaatio on asennettu, sitä ei tarvitse tehdä enää uudestaan.

4 Käytännön toteutus

Mikropalvelupohjaisen järjestelmän kehittämiseen on kaksi lähestymistapaa (Newman, 2015a). Järjestelmän voi toteuttaa joko alusta alkaen mikropalvelupohjaisena tai aloittaa monoliittisella arkkitehtuurilla, joka siirretään myöhemmin mikropalvelupohjaiseksi. Mikropalveluarkkitehtuurin toteutus voidaan jakaa itse mikropalveluiden mallintamiseen sekä arkkitehtuurin tukena käytettävän infrastruktuurin käyttöönottoon. Mikropalveluiden mallintamiseen kuuluu palvelurajojen määrittäminen sekä mahdollinen palvelujen sisäisen mallinnus sekä mahdollinen palveluiden sisäisen toteutuksen hallinto. Infrastruktuuriin kuuluu palveluiden viestinvälitys, testaus, julkaisu sekä valvonta.

4.1 Monoliittisella arkkitehtuurilla aloittaminen

Fowlerin (2015b) mukaan monoliittisella arkkitehtuurilla aloittaminen on riskittömämpi toteutustapa. Monoliitin kehitys on alkuvaiheessa nopeaa ja monimutkaisuuden kasvaessa järjestelmä voidaan pilkkoa pala palalta mikropalveluihin.

Useita mikropalveluita kattava refaktoroiminen on vaikeampaa kuin monoliitin sisällä refaktoroiminen (Fowler, 2015b; Newman, 2015a). Sovelluksen kehityksen alussa vallitseva käsitys järjestelmän vaatimuksista ja niiden painopisteistä muuttuu usein kehittäjien tuntemuksen toimialasta kasvaessa. Aloittamalla monoliittisella arkkitehtuurilla järjestelmä on mikropalvelupohjaiseksi siirrettäessä kypsempi. Tämä auttaa määrittelemään mikropalveluiden rajat tarkemmin.

Koska monoliitilla aloitettaessa sama järjestelmä toteutetaan osittain kahteen kertaan, lähestymistapa vie pitkällä tähtäimellä enemmän aikaa kuin suoraan mikropalveluilla aloittaminen (Newman, 2015b).

Siirtymä mikropalveluihin voidaan toteuttaa vaiheittain tai kerralla (Fowler, 2015b). Jos monoliitti aiotaan jakaa kerralla mikropalveluihin, on monoliittia kehitettäessä erityisen

tärkeää huomioida modulaarinen rakenne sekä ohjelmointirajapinnoissa että tietokannoissa- Tämä tekee siirtymästä helpompaa ja vähemmän aikaa vievää. Vaiheittain siirtyminen jättää usein järjestelmän ytimen monoliittiseksi, vaikka uusi kehitys ja muutokset tapahtuvatkin mikropalveluissa.

Monoliitti voidaan myös hylätä kokonaan ja mikropalvelupohjainen järjestelmä toteuttaa puhtaalta pöydältä (Fowler, 2015b). Monoliitti voidaan kehittää nopeasti, kun tulevaa siirtymää ei tarvitse ottaa huomioon. Tämä lähestymistapa saattaa olla järkevä, mikäli järjestelmä on saatava nopeasti tuotantoon. Lähestymistapa vastaa Tilkovin (2015) seuraavassa luvussa esittämää tapaa.

Pienemmän riskin ja siirtymän vaiheittaisuuden vuoksi monoliitilla aloittamista voidaan suositella kehitystiimeille, joilla ei ole aikaisempaa kokemusta arkkitehtuurista. Newmanin (2015b) mukaan vaiheittainen siirtyminen mikropalveluihin antaa kuvan myös kehittäjien valmiudesta hallinnoida mikropalveluita. Jos jo muutaman mikropalvelun julkaisu ja ylläpito tuottaa vaikeuksia, täysin mikropalvelupohjainen ratkaisu tulee aiheuttamaan ongelmia.

4.2 Suoraan mikropalveluilla aloittaminen

Tilkov (2015) ei suosittele monoliitilla aloittamista. Hänen mukaansa monoliitin kehittämien johtaa väistämättä paljon moduulien välisiä riippuvuussuhteita sisältävään koodikantaan, jonka pilkkominen mikropalveluiksi on hyvin vaikeaa. Hän myös pitää mikropalveluarkkitehtuurin suurimpana hyötynä modulaarisen rakenteen rikkomisen vaikeuden. Tästä syystä hänen mukaansa on siirtyminen mikropalveluarkkitehtuuriin turhaa, mikäli kehittäjät pystyvät jo toteuttamaan kiinteän modulaarisen monoliitin. Tilkov on samaa mieltä Newmanin (2015a) kanssa sovelluksen toimialan tuntemuksen tärkeydestä ennen mikropalveluarkkitehtuurin käyttämistä. Hänen mukaansa ideaalitilanteessa mikropalvelupohjaista sovellusta edeltää saman sovelluksen monoliittinen versio.

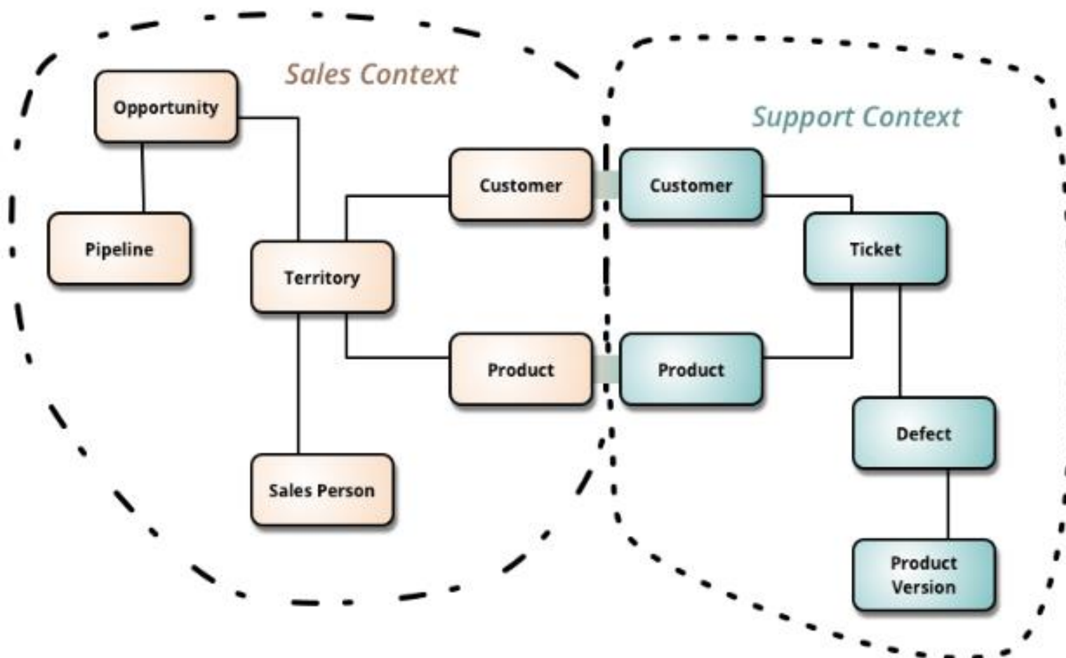
Kuten alaluvussa 4.1 todetaan, suoraan mikropalveluarkkitehtuurilla aloitettaessa palvelurajojen määritys on epävarmempaa. Jotta mikropalveluiden väliseltä refaktoroinnilta vältyttäisiin, kehityksen alkuvaiheessa on järkevää tehdä karkeajakoisempia mikropalveluita, jotka voidaan myöhemmin palvelurajojen vakiintuessa jakaa pienempiin mikropalveluihin (Newman, 2015a).

Korkean oppimiskäyrän ja käyttöönottokustannusten vuoksi sovelluksen kehittäminen on aluksi hidasta, jos kehitystiimillä ei ole aiempaa kokemusta mikropalveluista. Sitoutuminen kehityksen alusta asti entuudestaan tuntemattomaan arkkitehtuuriin on hyvin riskialtista.

Mikäli kehittäjillä on aikaisempaa kokemusta mikropalveluarkkitehtuurista ja tarvittava infrastruktuuri on jo otettu käyttöön aikaisemmissa järjestelmissä, suoraan mikropalveluilla aloittaminen voi olla järkevää. Aloittamalla suoraan mikropalveluarkkitehtuurilla vältetään monoliitin jakamisesta aiheutuva ylimääräinen työ.

4.3 Palvelujen mallinnus

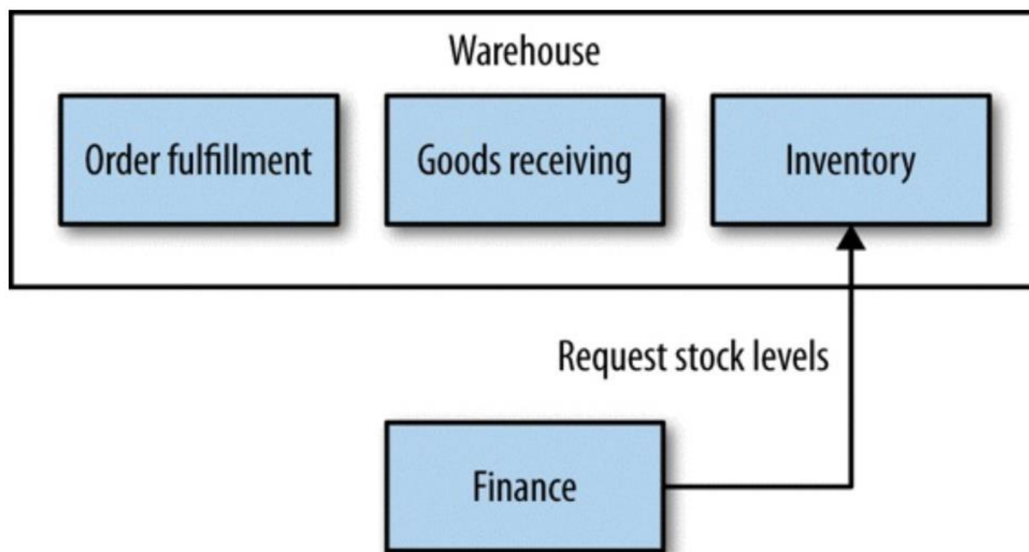
Palvelurajoja määrittäessä tavoitteena on pyrkiä tekemään palveluista mahdollisimman kiinteitä ja löyhästi kytkettyjä muihin palveluihin (Newman, 2015a). Mikropalvelujen mallintaminen suositellaan useimmiten tekemään toimialan toiminnallisuuksien mukaan (Newman, 2015a; Lewis ja Fowler, 2014).



Kuva 4. Saman sovelluksen kaksi mallia rajatuissa konteksteissaan (Fowler, 2014).

Evansin (2003) toimialakohtaisessa suunnittelussa (*domain driven design*) sovelluksesta luodaan malleja, jotka vastaavat taustalla olevan digitoitavan toimialan prosesseja. Sovellus kehitetään niin, että sen luokat ja metodit vastaavat mallia. Laajat sovellukset koostuvat useista malleista. Rajatut kontekstit (*bounded contexts*) määrittävät rajat, joissa mallit pätevät ja ne osat malleista, jotka pätevät mallin ulkopuolella. Mallit pystyvät vuorovaikuttamaan kontekstinsa ulkopuolella vain toisiin konteksteihin määriteltyjen rajapintojen kautta. Kuvassa 4 esitetään kaksi rajattua kontekstia ja niiden väliset rajapinnat.

Rajattujen kontekstien etsiminen auttaa mikropalvelujen rajojen määrittämisessä (Newman, 2015a; Lewis ja Fowler, 2014; Dehghani, 2018). Rajattuja konteksteja määritettäessä on hyvä jakaa sovellus aluksi isoihin ja karkeajakoisiin konteksteihin (Newman, 2015a). Saaduista konteksteista voidaan edelleen etsiä pienempiä rajattuja konteksteja, kunnes saavutettu rakeisuus (*granularity*) vastaa toivottavaa mikropalvelujen kokoa.



Kuva 5. Karkeajakoinen 'Warehouse' rajattu konteksti piilottaa sisäiset mikropalveluna ulkomaailmalle (Newman, 2015a)

Karkeajakoisemmat rajatut kontekstit, jotka on jaettu edelleen hienojakoisempiin mikropalveluihin, voivat toimia julkisivuna palveluille, joista se koostuu (Newman, 2015a). Kuvassa 5 on esitetty tällainen karkeajakoinen rajattu konteksti, joka sisältää useamman mikropalvelun. Karkeajakoiselle rajatulle kontekstille määritetään yksi rajapinta. Tällä tavalla ulkopuolisten mikropalvelujen ei tarvitse välittää kontekstien sisäisistä palveluista. Newmanin mukaan tämä lähestymistapa on järkevä, mikäli kehitystiimi vastaa koko karkeajakoisesta rajatusta kontekstista. Mikäli eri tiimit vastaavat kontekstin sisäisistä palveluista, kannattaa ne toteuttaa ylemmän tason palveluina. Tämä johtuu Conwayn lain mukaisesta kehittäjien organisoinnin vaikutuksesta ohjelman rakenteeseen.

Rajattuja konteksteja määritettäessä on tärkeää seurata taustalla olevan mallinnettavan toimialan toiminnallisuuksia eikä kontekstien välillä jaettavaa dataa (Newman, 2015a). Datan pohjalta mallintaminen johtaa suureen määrään aneemisia CRUD-pohjaisia palveluita (Newman, 2015a; Dehghani, 2018). Mallintamalla järjestelmä toimialan mukaan siitä tulee helpommin muunneltava, sillä muutokset rajoittuvat usein toimialojen sisälle.

Richardson (2018) ja Tyszberowicz, Heinrich, Liu ja Liu (2018) suosittelevat mikropalveluiden määrittelyn aloittamista käyttötapausten analysoimisella. Käyttötapauksista saadaan pääteltyä järjestelmän operaatiot ja tilan muuttujat, ts. toiminnan verbit ja substantiivit, joilla pystytään rakentamaan mallit ja niiden rajatut kontekstit.

Mikropalvelut voidaan myös mallintaa teknisen toiminnallisuuden mukaan (Newman, 2015a). Newman ei suosittele tätä lähestymistapaa.

Mikropalvelujen koolle ei ole tiukkaa määritelmää (Newman, 2015a; Richardson 2018; Lewis ja Fowler, 2014).

4.4 Monoliitin pilkkominen

Kuten alaluvussa 4.1 kerrotaan, voidaan monoliitti pilkkoa joko kerralla tai inkrementaalisesti. Fowler (2015b), Newman (2015a) ja Dehghani (2018) pitävät vaiheittaista lähestymistapaa parhaiten toimivana.

Mikäli monoliittia ei ole jaettu osiin toimialan mukaan tai mikäli periaatteista on lipsuttu ja teknistä velkaa on kerääntynyt liikaa, ensimmäinen vaihe monoliitin pilkkomisessa on rajattujen kontekstien etsiminen (Dehghani, 2018). Kun sovelluksen rajatut kontekstit on löydetty, monoliitti kannattaa refaktoroida toimialakohtaisesti jaetuksi modulaariseksi sovellukseksi. Vasta tämän jälkeen voidaan alkaa etsiä mikropalveluiden rajoja monoliitin karkeajakoisemmista rajatuista konteksteista. Jos monoliitti on jo jaettu toimialan mukaan moduuleihin eikä sovellukseen ole kertynyt paljoa teknistä velkaa, voidaan aloittaa suoraan tästä vaiheesta. Joskus pelkkä monoliitin refaktorointi kiinteämmäksi ratkaisee sovelluksen ongelmat niin, että siirto mikropalvelupohjaiseksi jää tarpeettomaksi.

Vaiheittainen monoliitin pilkkominen on järkevää aloittaa järjestelmän osista, jotka hyötyvät eniten koodikannan erottamisesta tai jotka on helppo erottaa (Newman, 2015a). Usein muuttuvat osat tai osat, joita tullaan muuttamaan lähitulevaisuudessa ovat hyviä

aloituskohtia. On myös hyvä aloittaa kokonaan yhden kehitystiimin vastuulla olleista osista. Erityistä tietoturvallisuutta edellyttävien osien irrottaminen mahdollistaa tiukemman turvallisuuden toteutuksen vaikuttamatta muuhun järjestelmän toimintaan.

Mikropalvelua irrotettaessa on tärkeää huomioida, kuinka tiukasti se on kytketty muuhun koodikantaan (Newman, 2015a). Tiukasti kytkettyjen järjestelmän osien irrottaminen on vaikeaa. Riippuvuussuhteiden kartoittamiseen voidaan käyttää erilaisia työkaluja. Useimmat IDE-työkalut mahdollistavat koodikannan riippuvuussuhteiden analysoinnin. Sosiaalinen koodianalyysi on vielä pidemmälle menevä analysointitekniikka, joka yhdistää koodin rakenteellisen analysoinnin kehittäjien toiminnan analysointiin versiohallintasovellusten kautta (Tornhill, 2019). Sosiaalisella koodianalyysilla pystytään etsimään koodikannan aktiivisimmin muuttuvat osat (Dehghani, 2018).

Koodista irrotettava osa voidaan joko kirjoittaa uudestaan tai siirtää suoraan mikropalveluun (Newman, 2018). Newmanin mukaan suora siirto on mahdollista, mikäli monoliitissa on kiinteä toimialan mukaan jaettu modulaarinen rakenne. Dehghani (2018) suosittelee kirjoittamaan palvelujen koodit uudestaan. Vanhassa koodissa on hänen mukaansa usein paljon monoliitin toteutukseen liittyvää boilerplate-koodia. Uudelleen kirjoittaminen myös mahdollistaa paremman, enemmän mikropalvelun toimialaa vastaavan toteutuksen. Siirrettäessä kannattaa aluksi vain kopioida toiminallisuus mikropalvelussa ja säilyttää monoliitti ennallaan (Newman, 2018). Toiminnallisuus voidaan myöhemmin poistaa monoliitista kokonaan, kun mikropalvelun toiminnasta on varmistuttu.

Pelkästään koodikannan siirtämisen jälkeen mikropalvelu on vielä kytketty monoliitin tietokantaan. Mikropalveluiden integraatio tietokantatasolla ei ole toivottavaa, mistä lisää Integraatio-luvussa. Tietokannan pilkkominen on haastavaa (Todkar, 2018; Newman, 2015a).

Todkar (2018) esittää vaiheittaisen strategian tietokannan pilkkomiselle. Palvelua mallinnettaessa selvitetään tarvittavat monoliitista irrotettavat osat. Mikropalvelu erotetaan

ensin loogisesti monoliitin sisällä, jolloin virheet palvelurajan määrittämisessä on helpompi korjata. Loogiseen erotteluun kuuluu abstrahointikerroksen luonti tietokantaan, jonka kautta järjestelmän erotettavat osat kutsuvat tietokantoja.

Loogisen erotuksen jälkeen tarvittavat uudet tietokantataulut toteutetaan monoliitin tietokantaan ja toteutetut abstrahointikerrokset kytketään uusiin tietokantatauluihin (Todkar, 2018). Kaikki viiteavainliitokset (*foreign key*) taulujen välillä täytyy poistaa, joten liitoslauseet (*join*) täytyy siirtää tietokantatasolta logiikkatasolle abstrahointikerrokseen. Tähän vaiheeseen kuuluu myös datan siirto erotettavista tauluista uusiin tauluihin. Tietokannan erottaminen ensin monoliitin sisällä mahdollistaa analyysin muutoksen vaikutuksesta suorituskykyyn. Liitoslauseiden toteuttaminen logiikkatasolla on raskaampaa kuin tietokantatasolla.

Seuraavaksi mikropalvelu erotetaan monoliitista niin, että se on vielä integroitu monoliitin tietokantaan (Todkar, 2018). Lopuksi uuden palvelun taulut ja data siirretään mikropalveluun ja ne poistetaan monoliitin tietokannasta.

4.5 Integraatio

Mikropalvelupohjaisen järjestelmän toiminta perustuu palveluiden väliseen kommunikaatioon (Richardson, 2018; Newman, 2015a). Metodikutsujen sijaan mikropalvelut joutuvat kommunikoimaan mahdollisesti hitaan verkon välityksellä ulkoista kommunikaatiomekanismia käyttäen. Oikeanlaisen kommunikaatiomekanismin valinta on siis ensiarvoisen tärkeää.

Nopeimmin käyttöön otettava integraatiomekanismi on jaettu tietokanta (Newman, 2015a). Integraatiota tietokantatasolla ei suositella, sillä se johtaa tiukasti toisiinsa kytkettyihin palveluihin, joita on vaikea muuttaa. Lisäksi se sitoo mikropalvelut yhteen tietokantateknologiaan.

Mikropalvelujen väliset viestit voidaan jakaa kahteen yläkategoriaan: synkronisisiin ja asynkronisiin (Rodger, 2018; Newman, 2015a.) Synkronisessa kommunikaatiossa mikropalvelut lähettävät kutsuja verkon yli ja pysäyttävät suorituksen vastauksen odottamisen ajaksi. Asynkronisessa kommunikaatiossa kutsun lähettäjä ei jää odottamaan vastausta.

Synkroninen kommunikaatio on yksinkertaisempaa, ja kutsujen tulokset nähdään heti (Rodger, 2018; Newman, 2015a). Synkroninen kommunikaatio sopii komentotyyppisiin (CRUD) toimintoihin ja monista järjestyksessä suoritettavista vaiheista koostuviin operaatioihin (Rodger, 2018). Asynkroninen kommunikaatio mahdollistaa käyttöliittymän responsiivisena pysymisen tilanteissa, joissa verkkoviiveestä tai muusta syystä vastaus kutsuun viivästyy.

Asynkroninen ja synkroninen kommunikaatio liittyvät kahteen tapaan toteuttaa mikropalvelujen välinen vuorovaikutus: **pyyntö/vastaus** ja **tapahtumapohjainen kommunikaatio** (Newman, 2015a). Pyyntö-/vastaus-pohjaisessa kommunikaatiossa asiakasohjelma lähettää pyynnön kohdeohjelmalle ja odottaa vastausta. Pyyntö-/vastaus-metodi on mahdollista toteuttaa sekä synkronisesti että asynkronisesti, esimerkiksi callback-funktion avulla. Tapahtumapohjaisessa kommunikaatiossa asiakasohjelmat julkaisevat tapahtumia ja kuuntelevat itselleen relevantteja muiden palveluiden julkaisemia tapahtumia. Tapahtumapohjainen kommunikaatio on luonteeltaan asynkronista.

Pyyntö/vastaus on yksinkertainen tapa toteuttaa prosessien välinen kommunikaatio (Williams, 2015). Yleisin tapa toteuttaa pyyntö-/vastaus-pohjainen kommunikaatio on käyttää ohjelmointirajapintojen toteutukseen REST-arkkitehtuuria ja viestinvälitykseen HTTP-protokollaa.

Eksplisiittinen pyyntöihin perustuva kommunikaatio johtaa Newmanin (2015a) mukaan herkästi järjestelmän toimintalogiikan pakkautumiseen muutamaan tärkeimpään mikropalveluun.

Tapahtumapohjainen kommunikaatio johtaa vähemmän tiukasti toisiinsa kytkettyihin mikropalveluihin (Newman, 2015a). Tapahtumia julkaisevat palvelut eivät ole kytkettyjä niitä kuunteleviin palveluihin, joten lähestymistapa johtaa löyhemmin kytkettyyn hajautetusti hallittuun järjestelmään. Uusien palvelujen liittäminen järjestelmään helpottuu, kun palvelut täytyy vain kytkeä tapahtumanjulkaisualustaan ja asettaa kuuntelemaan tarvitsemiaan tapahtumia. Tapahtumapohjainen kommunikaatio edellyttää jonkinlaisen viestinvälitysalustan käyttöönottoa tapahtumien julkaisua ja kuuntelemista varten. Yleisesti käytettyjä viestinvälitysalustoja ovat RabbitMQ ja ZeroMQ (Lewis ja Fowler, 2014).

4.6 Testaus

Mikropalveluarkkitehtuurissa testaus on haastavaa arkkitehtuurin hajautetun luonteen vuoksi (Richardson, 2018). Koska mikropalvelupohjainen järjestelmä perustuu palveluiden väliseen vuorovaikutukseen, kehittäjien tulee varmistaa palvelun toimivuus osana mikropalveluiden verkkoa.

Useita mikropalveluita kattavat testit ovat kuitenkin aikaa vieviä ja monimutkaisia toteuttaa (Richardson, 2018). Yksinkertainen mikropalvelun rajapintaa testaava testi (myöh. API-testi) saattaa vaatia useamman mikropalvelun ajamista ja monimutkaisen toimintalogiikan suorittamista. Asiakaspohjaiset sopimukset (*consumer driven contract*) ratkaisevat hitaan testauksen ongelman API-testeissä (Newman, 2015a; Richardson, 2018). Sopimukset määrittävät palveluiden asiakkaiden eli palveluita käyttävien mikropalveluiden odotukset palveluilta. Asiakaspalvelut kirjoittavat nämä odotukset koodimuodossa testeinä, jotka ajetaan odotusten toteuttajapalvelussa automaattisesti julkaisun yhteydessä. Kuluttajien toteuttamilla sopimustesteillä päästään eroon suuresta määrästä palvelujen välisiä integraatiotestejä.

Useita mikropalveluja kattavien testien hitaus johtuu suurimmaksi osaksi mikropalvelujen asentamisen viemästä ajasta. Richardson (2018) suosittelee tästä syystä tekemään

laajempia päästä päähän -testejä (*end-to-end test)*, jotka testaavat samassa testissä useita toiminnallisuuksia.

4.7 Julkaiseminen

Sovelluksen julkaisun automatisointi mahdollisimman pitkälle on mikropalveluarkkitehtuurissa tärkeää, sillä palvelumäärän kasvaessa toisistaan riippuvaisten mikropalveluiden julkaisusta tulee erittäin monimutkaista (Newman, 2015a). Jatkuvan integraation ja toimituksen periaatteet automatisoivat julkaisuprosessia ja takaavat julkaisun toimivuuden automatisoiduilla testeillä. Virtualisointi mahdollistaa fyysisen palvelimen laitteiston abstrahoinnin kautta mikropalvelukohtaisen ajonaikaisen ympäristön koteloinnin ja automatisoidun asennuksen.

Jatkuva integraatio (*CI, continuous integration*) on sovelluskehityskäytäntö, jossa kehittäjät integroivat tekemänsä muutokset tasaisin, lyhyin aikavälein yhteiseen versionhallintajärjestelmän tietovarastoon (*repository*) (Meyer, 2014). Tavallisesti pyrkimys on yhdistää jokaisen kehittäjän muutokset vähintään kerran päivässä. Integraation yhteydessä järjestelmän toimivuus muutosten jälkeen varmistetaan automatisoiduin testein. Jatkuva integraatio toteutetaan työkaluilla, jotka toimivat yhdessä versionhallintajärjestelmän ja integraatiopalvelimen kanssa. Integraatiotyökalu ajetaan aina, kun yhteiseen tietovarastoon siirretään muutoksia. Työkalu rakentaa integraatiopalvelimelle ajettavat artefaktit sovelluksesta muutosten kanssa ja ajaa testit näitä artefakteja vastaan.

Jatkuva toimitus (*CD, continuous delivery*) on jatkuvan integraation jatke. Jatkuvassa toimituksessa jatkuvan integraation prosessiin lisätään vaiheet, jotka tarvitaan sovelluksen julkaisuun tuotannossa. Jatkuvassa toimituksessa pyritään tilanteeseen, jossa sovelluksen uusimman version vienti tuotantoon on napinpainalluksen takana. Jatkuva toimitus eroaa jatkuvasta julkaisusta (*continuous deployment*) siinä, että jatkuvassa toimituksessa sovellusta ei julkaista automaattisesti tuotannossa.

Mikropalvelupohjaisessa järjestelmässä CI/CD-prosessin toteuttamiseen on useita tapoja (Newman, 2015a). Yksinkertaisin tapa on käyttää yhtä tietovarastoa, johon koko järjestelmän koodikanta yhdistetään. Tietovarastosta rakennetaan yksi koontiversio (*build*), jolla testit ajetaan integraatiopalvelimella. Yhden koontiversion lähestymistapa johtaa hitaaseen integraatioprosessiin, sillä muutoksiin liittyvän mikropalvelun testien lisäksi ajetaan myös muiden palveluiden testit. Integroitujen muutosten jälkeen on epäselvää, mitkä mikropalvelut pitää julkaista uudestaan muutosten takia.

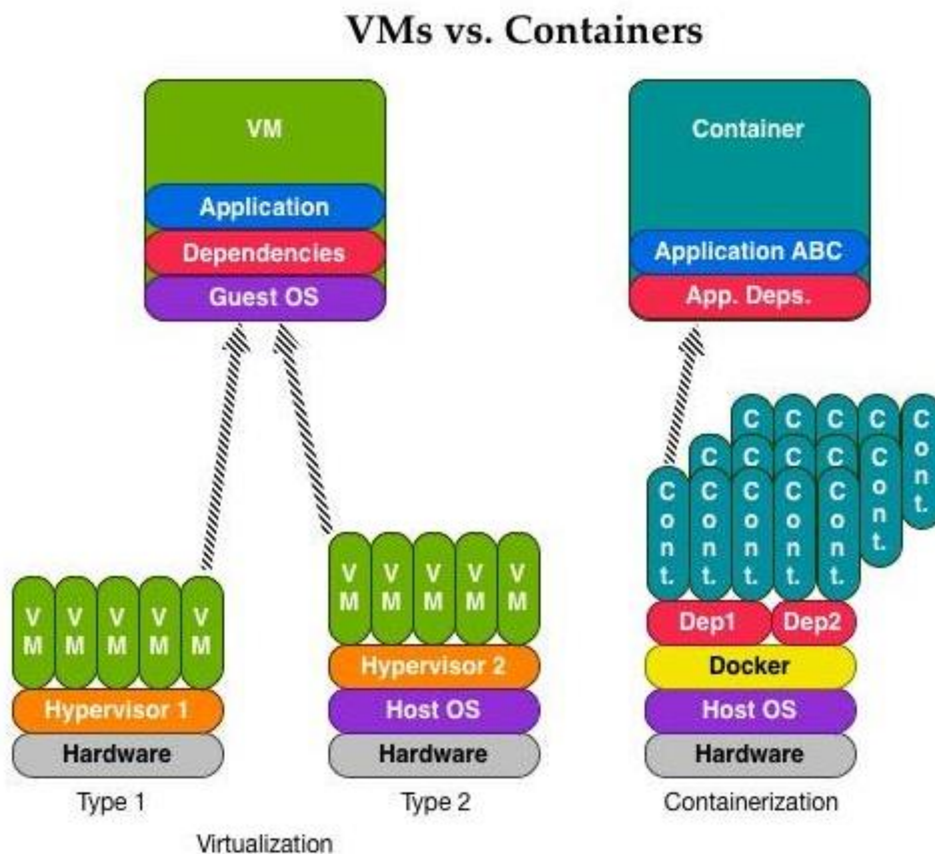
Newmanin (2015a) mukaan parempi lähestymistapa on säilyttää joka mikropalvelun koodia omassa tietovarastossaan, josta on oma koontiversio. Tällä tavalla mikropalvelusta vastuussa olevat kehittäjät kontrolloivat myös mikropalvelun CI/CD-prosessia. Testaus on nopeaa, sillä vain kyseisen palvelun omat testit ajetaan ja mikropalvelu on julkaistavissa itsenäisesti.

Virtualisointi on teknologia, joka mahdollistaa useamman itsenäisen ympäristön ajamisen samalla palvelimella. Laitteiston tarjoamat resurssit jaetaan useamman virtualisoidun ympäristön kesken (Merkel, 2014).

Mikropalvelujen tuotantoympäristöjen virtualisointi mahdollistaa ympäristön vaatimusten koteloinnin (Richardson, 2018). Teknologisesti heterogeenisillä mikropalveluilla voi olla paljon vaihtelevia ajon aikaisen ympäristön vaatimuksia. Virtualisoinnin avulla kehittäjät voivat määrittää automaattisesti julkaisuprosessissa asennettavan virtuaalisen ympäristön. Virtuaalisilla ympäristöillä voidaan myös kontrolloida kunkin mikropalvelun käytettävissä olevaa resurssimäärää.

Virtualisointi voidaan toteuttaa virtuaalikoneiden (esim. VMWare, VirtualBox) tai uudempien konttipohjaisten teknologioiden (esim. Docker) avulla (Merkel, 2014). Virtuaalikoneet virtualisoivat laitteistotasolla ottamalla käyttöön osan isäntäkoneen laitteiston suorituskyvystä ja luomalla kokonaisen itsenäisen tietokoneen käyttöönotetun laitteis-

ton päälle. Virtuaalikoneet toteutetaan hypervisor-osalla, joka luo ja ajaa koneita. Virtuaalikoneet voidaan jakaa kahteen tyyppiin. Ensimmäisen tyypin virtuaalikoneissa hypervisor on yhdistetty suoraan laitteistoon. Toisessa tyypissä hypervisor ajetaan isäntäkoneen käyttöjärjestelmän kautta. Kuvassa 6 kuvataan tyypin 1 ja 2 virtualisoinnit sekä konttipohjainen virtualisointi.



Kuva 6. Virtuaalikoneiden ja konttien erot (Merkel, 2014).

Ohjelmistokontit tarjoavat virtuaalikoneita kevyemmän, standardoidun virtualisointialustan (Merkel, 2014). Siinä missä virtuaalikoneet virtualisoivat laitteistotasolla, kontit virtualisoivat käyttöjärjestelmätasolla. Kontit ovat ikään kuin tiloja, jotka on eristetty toisista konteista ja tietyistä osista isäntäkäyttöjärjestelmää. Koska kontit käyttävät hyväksi isäntäkoneen käyttöjärjestelmää, niihin ei tarvitse sisällyttää omaa käyttöjärjestelmää.

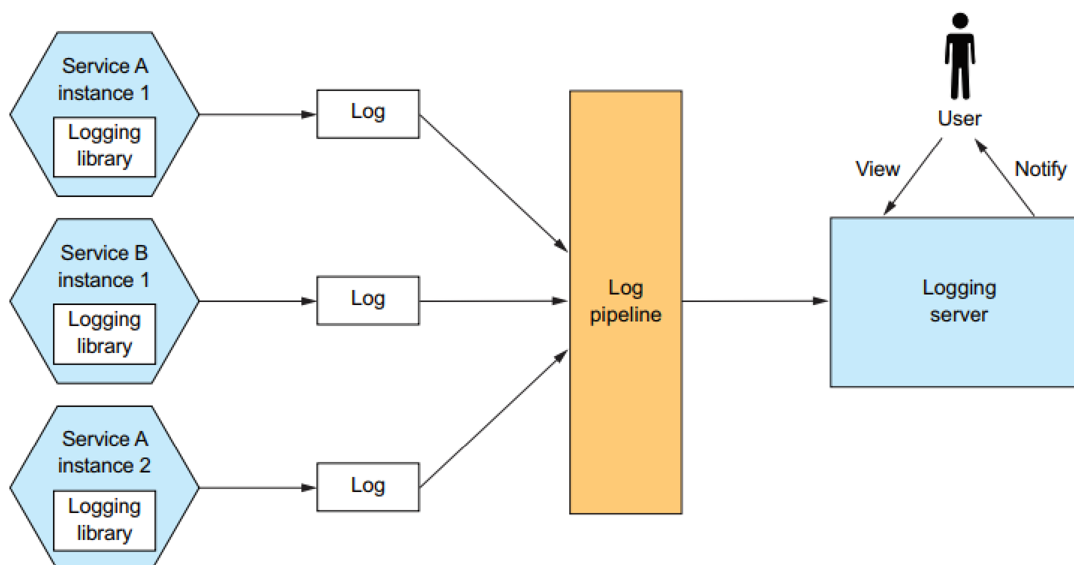
Konttien resurssienkäyttö on myös paljon virtuaalikoneita tehokkaampaa: jos kontti ei suorita mitään, se ei käytä isäntäkäyttöjärjestelmän resursseja.

Jaramillo, Nguyen ja Smart (2016) suosittelevat mikropalvelujen julkaisemista Docker-konteissa. Edellä mainittujen virtualisaation ja ohjelmistokonttien etujen lisäksi Docker-kontit toimivat useilla eri alustoilla kuten Linux-distribuutioilla ja pilvipalveluissa. Docker-kontit luodaan automatisoidusti kuhunkin konttiin määritellyn komentosarjan (*Dockerfile)* mukaan.

Sovelluksista rakennetaan docker-muistinkuvia (*docker-image*) (Docker Foundation, 2019). Docker-muistinkuva on itsenäisesti suoritettava pakkaus, joka sisältää koko sovelluksen koodikannan, ajonajan ja kaiken muun suorittamiseen tarvittavan. Docker-muistinkuvat rakennetaan dockerfilen perusteella. Docker-kontit ovat käynnistettyjä docker-muistinkuvia. Muistinkuvien rakentamisesta ja konttien käynnistämisestä ja ajamisesta vastaa Docker engine -sovellus.

4.8 Valvonta

Mikropalveluarkkitehtuuri kasvattaa järjestelmän tilanteen valvonnan monimutkaisuutta (Newman, 2015a). Toisin kuin monoliittisessa järjestelmässä mikropalveluarkkitehtuurissa ongelman aiheuttajan etsimiseen ei ole välttämättä selvää alkukohtaa. Toisista mikropalveluista riippuvaisten mikropalvelujen ketjussa vikatilanteen ilmenemispaikka saattaa olla kaukana aiheuttajasta. Sama mikropalvelu voi olla myös skaalattu kloonaamalla.



Kuva 7. Mikropalveluiden valvontajärjestelmä (Richardson, 2018).

Mikropalvelupohjaisen järjestelmän valvonta voidaan toteuttaa kokoamalla yhteen mikropalvelutasolla tuotettua valvontadataa, kuten lokitiedostoja (Richardson, 2018). Kuvassa 7 kuvataan lokitietoja aggregoiva järjestelmä: mikropalvelutasolla lokitietoja voidaan tuottaa ohjelmakirjastojen avulla. Mikropalvelut siirtävät lokitietonsa keskitetylle lokitietojen keräyspalvelimelle, jonka kautta kehittäjät voivat valvoa järjestelmän toimintaa. Valvontajärjestelmään tarvitaan lokitietojen keräyspalvelin, mikropalvelutasolla lokien tuottaja sekä prosessi, joka siirtää lokitiedot mikropalveluista lokitietojen keräyspalvelimelle.

5 Sovelluksen tämänhetkinen tilanne

Yrityksessä kiinnostuttiin mikropalveluarkkitehtuurista lähinnä arkkitehtuurin herättämän innostuksen ja sen lupaamien mahdollisuuksien takia. Arkkitehtuurin kokeilualustaksi otettiin tuolloin suunnitteluvaiheessa ollut toiminnanohjausjärjestelmä, jota käytetään tämän diplomityön case-projektina.

Sovellusta lähdettiin kehittämään ensin monoliittisena huomioiden tuleva siirtymä mikropalveluarkkitehtuuriin. Koodikannasta pyrittiin saaman mahdollisimman helposti mikropalveluiksi pilkottava pyrkimällä kiinteään modulaariseen rakenteeseen. Sovellusta varten otettiin käyttöön myös automatisoitu jatkuva julkaisu ja valmius jatkuvaan toimitukseen.

Ajatuksesta siirtää sovellus mikropalvelupohjaiseksi luovuttiin muutama kuukausi kehitystyön aloittamisen jälkeen. Sovellusta kehitettäessä käytetyt siirtoa helpottavat periaatteet hidastivat kehitystyötä. Myös mikropalveluarkkitehtuurin hyödyt paljastuivat kyseenalaiseksi.

Toiminnanohjausjärjestelmä on ollut kehityksessä vuoden 2018 lopusta lähtien. Järjestelmää kehitettiin yhteistyössä ensimmäisen asiakkaan kanssa, jonka käyttöön järjestelmä siirtyi kesällä 2019.

5.1 Sovelluksen vaatimukset ja ominaisuudet

Sovellus on pilvipalveluna toimiva toiminnanohjausjärjestelmä ensisijaisesti palvelualan yrityksille. Sovellus tarjotaan palveluna, eli kaikki sovellusta käyttävät yrityksen käyttävät samaa sovellusta, mutta jokainen omaa dataansa. Alustavat ominaisuudet määriteltiin yhdessä ensimmäisten käyttäjien kanssa. Keskeisimmät vaatimukset liittyvät työajan seurantaan ja työn organisointiin.

Työntekijä voi merkitä sovellukseen työtunnit, poissaolot ja työmatkat. Työtunteja voidaan merkitä kellokorttimaisesti leimaamalla sisään/ulos tai vaihtoehtoisesti merkitsemällä suoraan tehtyjä työjaksoja. Tehtyjä työtunteja voidaan kohdistaa eri tavoilla eri yritysten tarpeiden mukaan. Mahdollisia kohdistuskohteita ovat alustavasti asiakkaat, toimipisteet, joissa työ on tehty, projektit, tehtävälajit ja tehtävälajien alitehtävälajit. Työntekijät voivat kirjata aikaa joko mobiilisovelluksen tai verkkosivun kautta.

Tehdyistä työtunneista, poissaoloista ja työmatkoista voidaan luoda erilaisia raportteja muun muassa laskutusta ja palkanmaksua varten. Tiedot tehdyistä työtunneista, poissaoloista ynnä muista vastaavista voidaan myös siirtää suoraan järjestelmää käyttävälle tilitoimistolle.

Järjestelmää voidaan käyttää työvuorojen suunnitteluun. Alustavasti työvuoroille voidaan määritellä toimipiste, jossa vuoro suoritetaan, ja suoritettava tehtävä (tehtävälaji) sekä työntekijät, joille vuoro osoitetaan. Työntekijöille voidaan myös määrittää pätevyyksiä eri toimipisteisiin ja tehtävälajeihin, jolloin työvuorojen suunnittelija pystyy näkemään, kuka on pätevä mihinkin työtehtävään.

Työvuoroille rajataan aikaikkuna, jonka aikana vuoro on suoritettava, ja maksimiaika työvuorolle. Työntekijälle on myös mahdollista lähettää automaattisesti sähköpostiin tai mobiilisovellukseen erilaisia työvuoroihin liittyviä muistutuksia. Muistutuksia voidaan lähettää, jos työntekijä ei ole aloittanut määrättyä työvuoroa ajoissa tai mikäli maksimiaika on lähestymässä. Muistutukset voidaan myös kytkeä pois päältä.

Järjestelmä on konfiguroitavissa eri organisaatioiden tarpeiden mukaan. Edellä mainitut aikaleimaus- ja työajan kohdistamistavat ovat organisaatiotason asetuksia. Järjestelmästä on myös tarkoitus tehdä tilauspohjainen, jolloin yritykset valitsevat itselleen tarpeellisia järjestelmän eri osia ja maksavat vain käyttämistään osista.

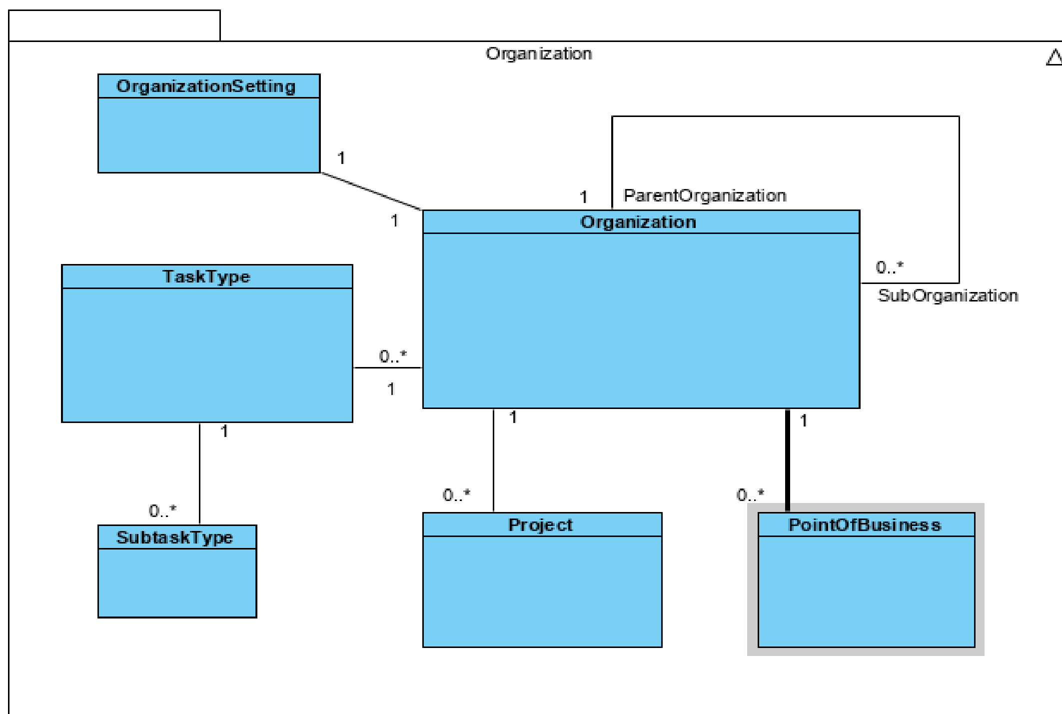
Järjestelmän tukee eritasoisia käyttäjiä. Yritykset voivat määritellä omat käyttäjätasonsa työntekijöilleen ja käyttäjätasoille pääsyt sovelluksen eri ominaisuuksiin.

Sovelluksen back end toteutettiin PHP-kielellä käyttäen Laravel -sovelluskehystä. Tietokantana toimii MySQL.

5.2 Sovelluksen rakenne

Sovellus jaettiin karkeajakoisiin moduuleihin toimialan mukaan. Moduuleita oli tarkoitus käyttää lähtökohtana mikropalveluiden rajojen määrittämiseen. Moduulit toteutettiin PHP:n nimiavaruuksina.

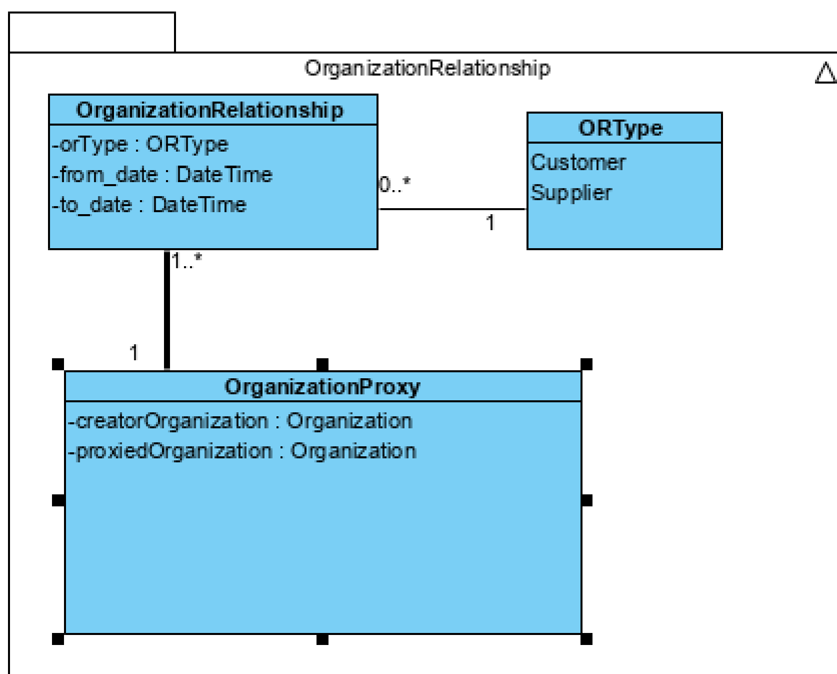
5.2.1 Organization



Kuva 8. Organization-moduulin rakenne.

Organisaatiomoduuli sisältää yritysten organisaation rakenteen ja muita työn organisointiin liittyviä asioita. Organisaation rakenteeseen kuuluvat itse yritys ja mahdolliset aliorganisaatiot. Työn organisointiin liittyvät asiat ovat organisaatiotason kohteita, johon tehtyjä työtunteja ja suunniteltuja työvuoroja voidaan liittää. Näitä kohteita ovat toimipisteet (**PointOfBusiness**), projektit (**Project**), sekä tehtävälajit ja alitehtävälajit (**TaskType, SubtaskType**). Moduuli sisältää myös organisaatio- ja aliorganisaatiokohtaisia asetuksia portaalin toiminnasta (**OrganizationSetting**).

5.2.2 OrganizationProxy

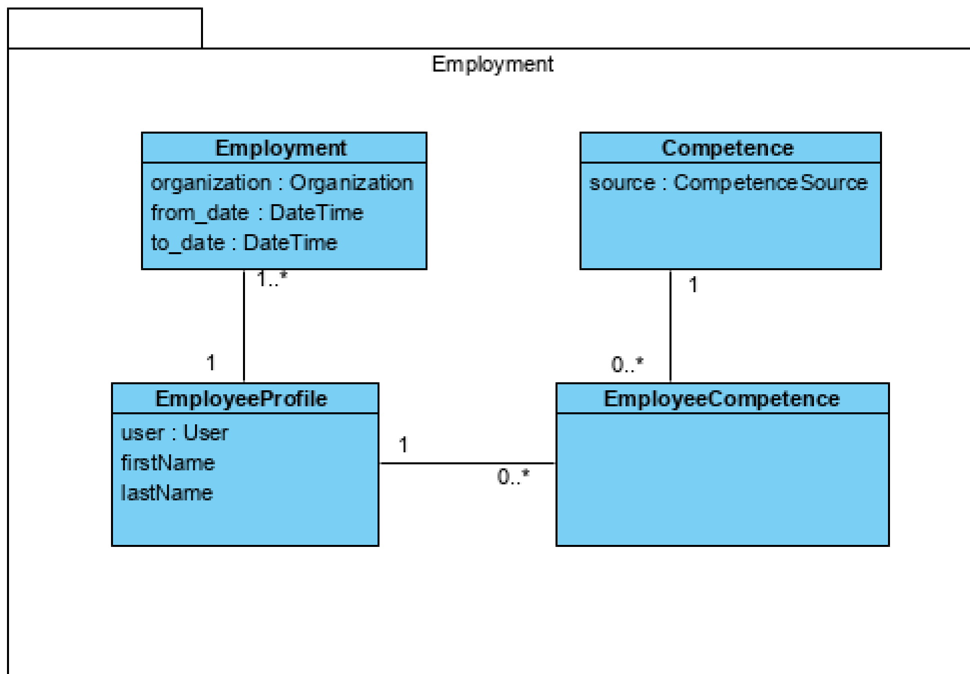


Kuva 9. OrganizationProxy-moduulin rakenne.

OrganisationProxy-moduuli on tarkoitettu organisaatioiden asiakkaiden ja toimittajien hallinnointiin. Asiakkaat ja toimittajat (**OrganizationProxy**) voivat olla järjestelmän muita käyttäjäorganisaatioita (**-proxiedOrganization**) tai ulkoisia organisaatioita. Työntekijät voivat osoittaa tekemänsä työtunnit asiakkaalle

Järjestelmässä olevien organisaatioiden välisillä toimittaja-/asiakassuhteilla on tarkoitus tulevaisuudessa integroida esimerkiksi palkanmaksu tilitoimiston ja asiakasyrityksen välillä tai laskutus toimittajien/asiakkaiden välillä.

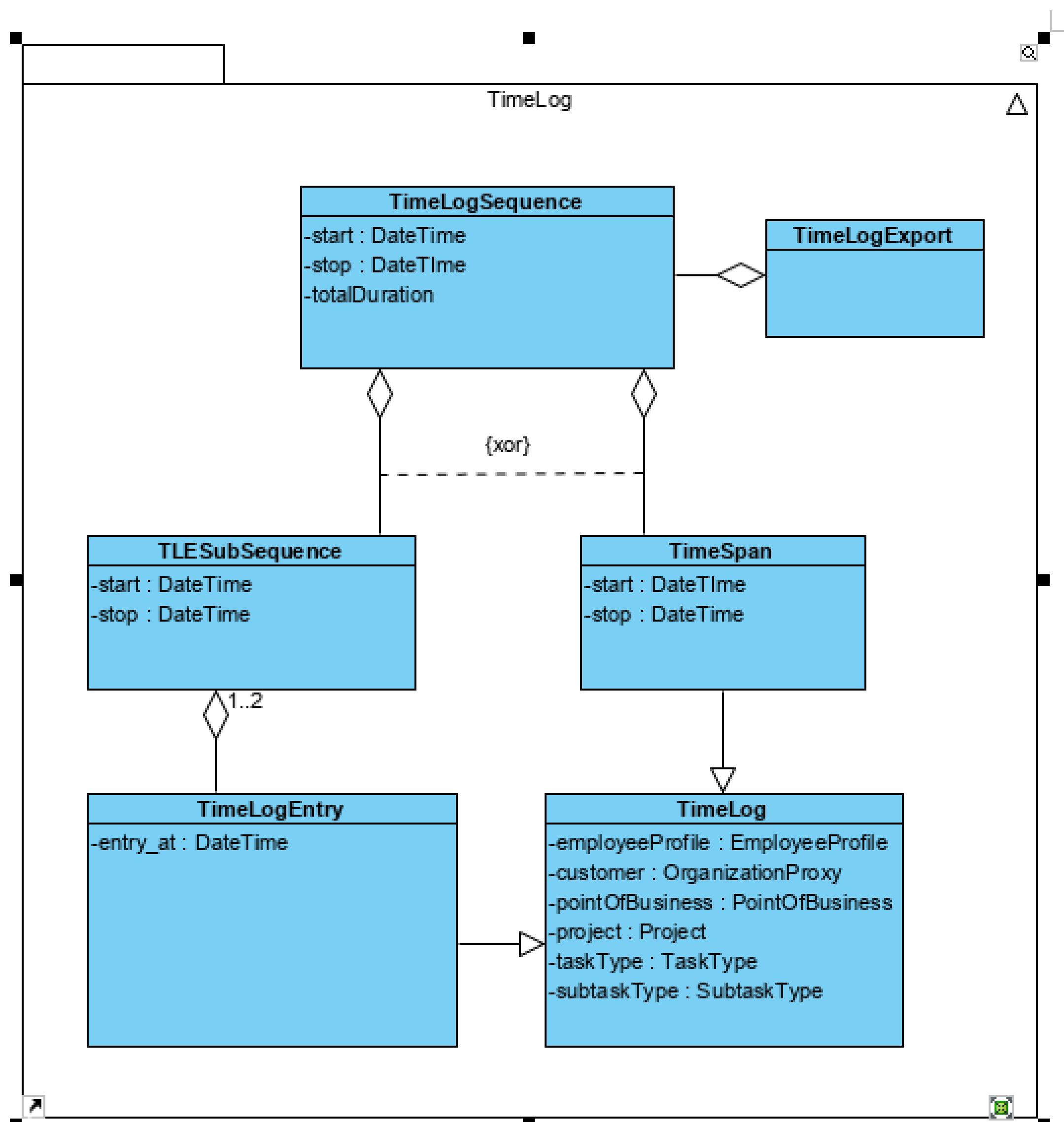
5.2.3 Employment



Kuva 10. Employment-moduulin rakenne.

Employment-moduuli on tarkoitettu työntekijöiden hallinnointiin. **EmployeeProfile** kuvaa yhtä työntekijää ja sisältää tietoja tästä. **EmployeeProfile** liitetään **Identity**-moduulin **User**-käyttäjämalliin. **Employment**-malli kuvaa työntekijän työsuhdetta yritykseen **(Organization)**. Työntekijöille on mahdollista määrittää pätevyyksiä (**Competence**). Pätevyys voidaan linkittää erilaisiin pätevyyksien lähteisiin (**source**). Tällä hetkellä lähteenä käytetään ainoastaan toimipistettä (**PointOfBusiness**). Toimipisteen pätevyyttä käytetään tällä hetkellä työvuorosuunnittelussa määrittämään, keillä työntekijöillä on pätevyys tehdä työvuoro missäkin toimipisteessä.

5.2.4 TimeLog



Kuva 11. TimeLog-moduulin yksinkertaistettu malli

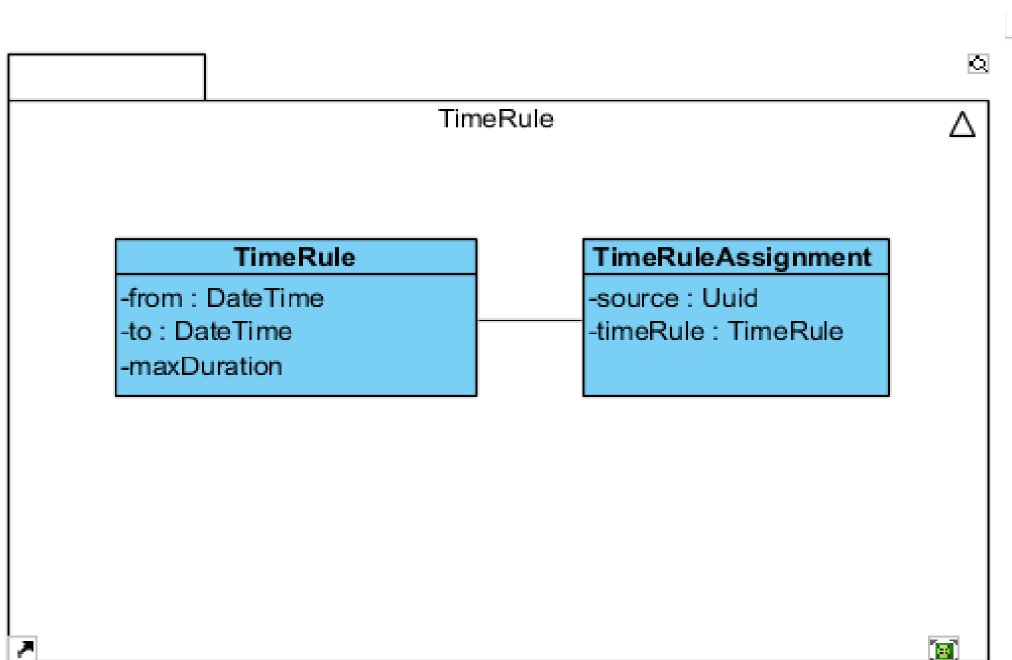
TimeLog-moduuli sisältää työajanseurannan ja työvuorojen suunnittelun sekä näihin liittyviä oheistoimintoja. Tehtyä työaikaa voidaan mitata joko aikaleimauksilla (**TimeLogEntry**) tai suoraan aikajaksoilla (**TimeSpan**) organisaation asetuksista riippuen. Aikaleimaukset ovat yksinkertaistettuna kellokorttimaisia sisään/ulos leimauksia. Työntekijät syöttävät aikaleimaukset tai -jaksot verkkosivun tai mobiilisovelluksen kautta. Sovellus

muodostaa lähetetyistä aikaleimauksista reaaliajassa aikajaksoja vastaavia alisekvenssejä (**TLESubSequence**).

Työaikasekvenssi (**TimeLogSequence**) on tehdystä työajasta luotu aggregaatti. Alisekvensseistä ja aikajaksoista ryhmitetään työaikasekvenssejä määriteltyjen sääntöjen mukaan. Määritellyt säännöt riippuvat organisaation asetuksista. Jos yritys käyttää esimerkiksi työvuorojen suunnittelua, ryhmitys voidaan toteuttaa yhdistämällä aikaleimaukset suunniteltuihin työvuoroihin.

Työaikasekvenssejä voidaan eksportoida Excel-muodossa (**TimeLogExport**). Raportteja voidaan tehdä muun muassa laskutukseen ja palkanmaksuun. Eri raportit sisältävät erilaista business-logiikkaa.

5.2.5 TimeRule

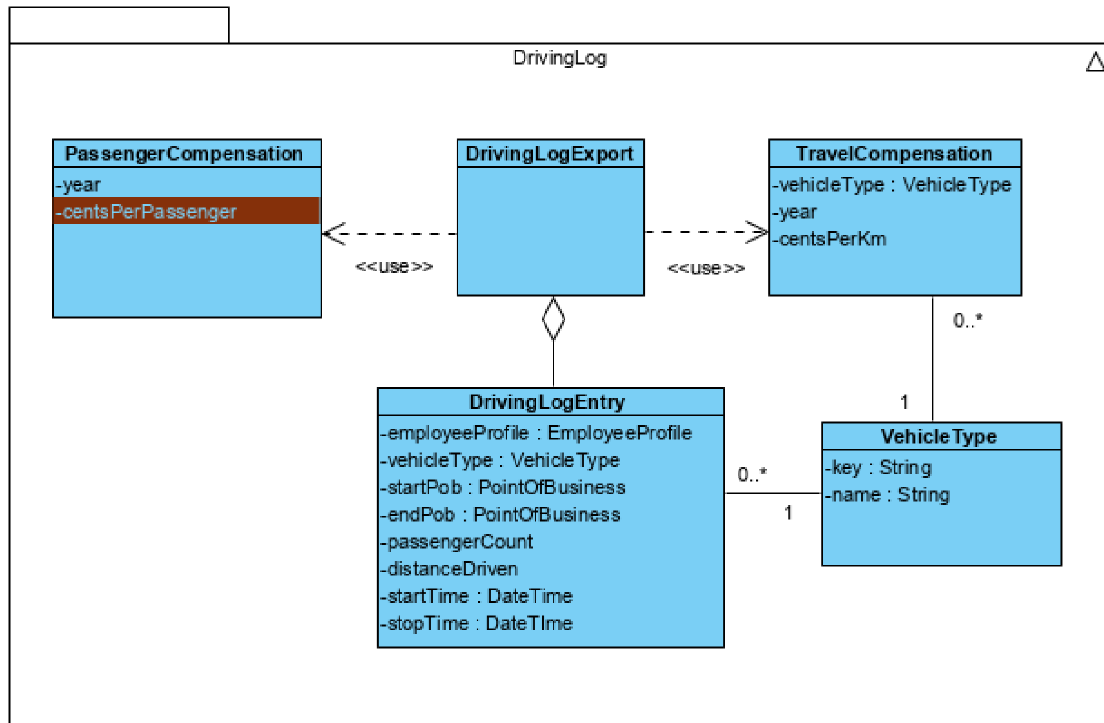


Kuva 12. TimeRule-moduulin yksinkertaistettu malli.

Työvuoronsuunnitteluun käytetty moduuli. Moduulissa voidaan määritellä aikasääntöjä (**TimeRule**), joille voidaan määritellä maksimiaika ja intervalli. Aikasäännöt voidaan

osoittaa eri kohteisiin (**TimeRuleAssignment**). Osoitus ja aikasääntö muodostavat yhdessä työvuoron.

5.2.6 DrivingLog



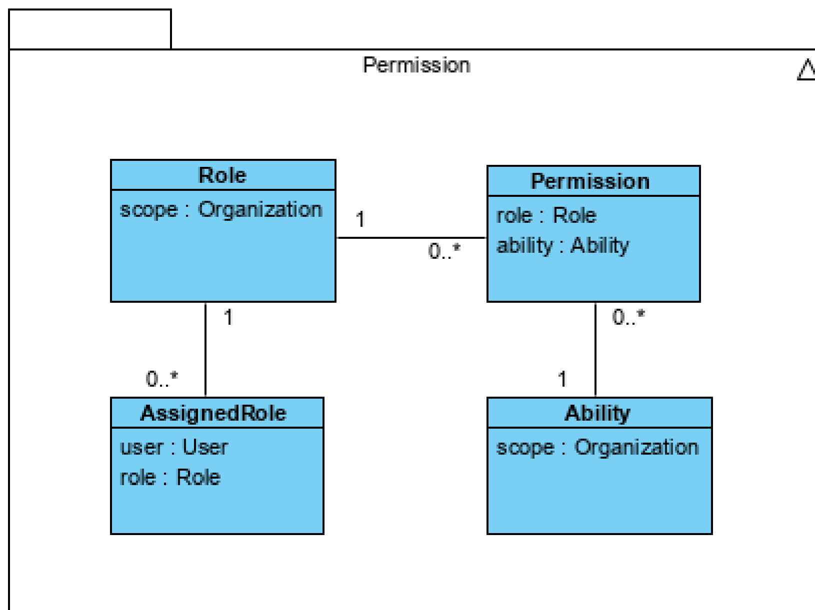
Kuva 13. DrivingLog-moduulin domain-malli.

DrivinLog-moduuli sisältää työmatkojen lokituksen. Moduulia käytetään työmatkakorvausten laskemiseen ja eksportointiin (**DrivingLogExport**). Työmatkakorvaukset lasketaan työmatkojen (**DrivingLogEntry**) ajoneuvotyypin (**VehicleType**) kilometrikorvauksen (**TravelCompensation**) ja matkustajakohtaisen kilometrikorvauksen (**PassengerCompensation**) perusteella.

5.2.7 Identity

Identity on käyttäjien tilien (**User**-luokka) hallitsemiseen käyetty moduuli. Se sisältää käyttäjätilien rekisteröinnin, deaktivoinnin ja autentikaation. Verkkosivun autentikaatio suoritetaan tavanomaisesti istunnoilla ja evästeillä. API-autentikaatioon käytetään oauth2-palvelinta. Tilien hallinnan toiminnot ovat enemmänkin infrastruktuurin kuin business-logiikan toimintoja. Koska tilien hallintaan liittyy kuitenkin enemmän vaatimuksia tietoturvallisuuden kannalta, Identity-moduuli erotettiin omaksi moduulikseen.

5.2.8 Permission



Kuva 14. Permission-moduulin rakenne.

Permission -moduulissa hallitaan käyttäjien auktorisointia. Auktorisointi toteutetaan roolien (**Role**) ja kykyjen (**Ability**) kautta. Kyvyt ovat auktorisoinnin perusta. Niillä ohjataan käyttäjien pääsyä järjestelmän eri osiin ja eri osien näkyvyyttä käyttöliittymissä. Roolit eivät sisällä omaa logiikkaansa, vaan ne ovat organisaatioiden määrittämiä karkeajakoisempia kykyjen yhdistelmiä, joita osoitetaan eri käyttäjille.

Tulevaisuudessa kyvyillä on tarkoitus ohjata myös organisaatioiden pääsyä järjestelmän eri osiin. Kyvyt voidaan osoittaa **scope** -attribuutin kautta osia käyttäville organisaatioille. Tämä mahdollistaa erilaisten tilaustyyppien käyttämisen.

5.3 CQRS

CQSR, **Command Query Responsilibity Segregation**, tarkoittaa sovelluksen tietokantarajapinnan luku- ja kirjoitusmallien erottamista toisistaan (Richardon, 2018). Tietokannan lukuoperaatiot toteutetaan lukumallin kyselyjen (*query)* kautta, kun taas muutosoperaatiot tehdään kirjoitusmallin komentojen (*command*) kautta. Kyselyt palauttavat dataa, komennot eivät.

Luku- ja kirjoitusmallien erottaminen johtaa selkeämpään rakenteeseen monimutkaisissa sovelluksissa, missä luku- ja kirjoitusoperaatiot eroavat toisistaan (Richardson 2018). Koska data ei kulje samaa reittiä, CQRS mahdollistaa asynkroniset kirjoitusoperaatiot ja synkroniset lukuoperaatiot. CQRS-periaatteen toteutus monoliittisessa sovelluksessa kasvattaa olennaisesti sovelluksen kompleksisuutta.

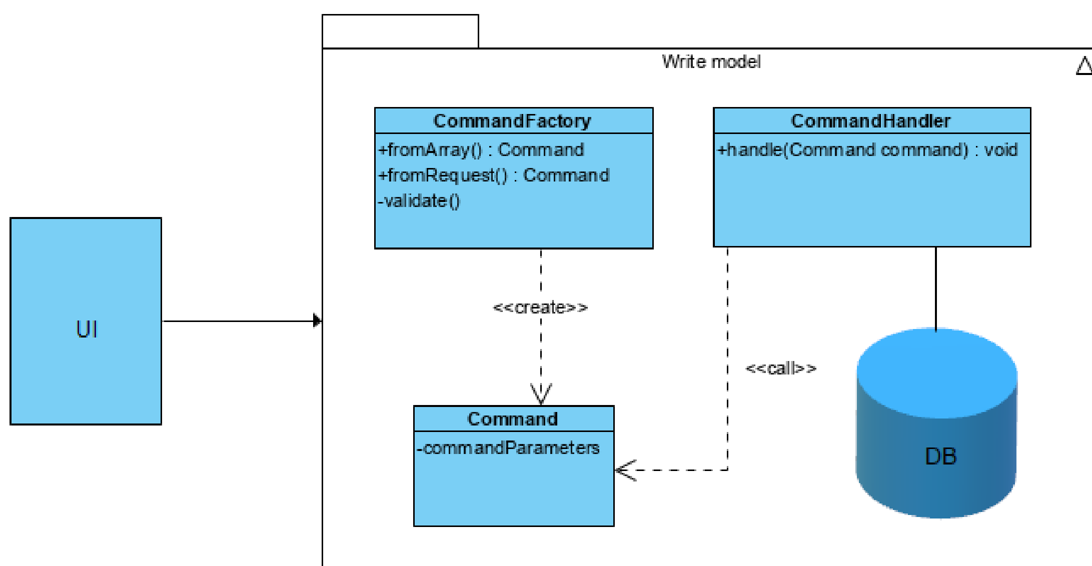
Mikropalveluarkkitehtuurissa luku- ja kirjoitusmallit ovat mikropalveluiden rajapinnoissa implisiittisesti eroteltuja (Richardson, 2018). Kukin rajapinnan päätepiste vastaa yhtä kyselyä tai komentoa, joten data kulkee luonnostaan eri reittiä. Asynkroniset komennot ja synkroniset kyselyt liittyvät palvelujen väliseen viestinreitityksen konfiguraatioon.

Sovelluksessa käytettiin CQRS-periaatetta. Käyttämällä CQRS-periaatetta sovelluksen monoliittisessa versiossa pyrittiin helpottamaan siirtymää mikropalveluihin. Monoliitissa mikropalveluja vastaavissa moduuleissa toteutetuista komennoista ja kyselyistä käyvät ilmi erotettavan mikropalvelun rajapinnan tarvittavat päätepisteet.

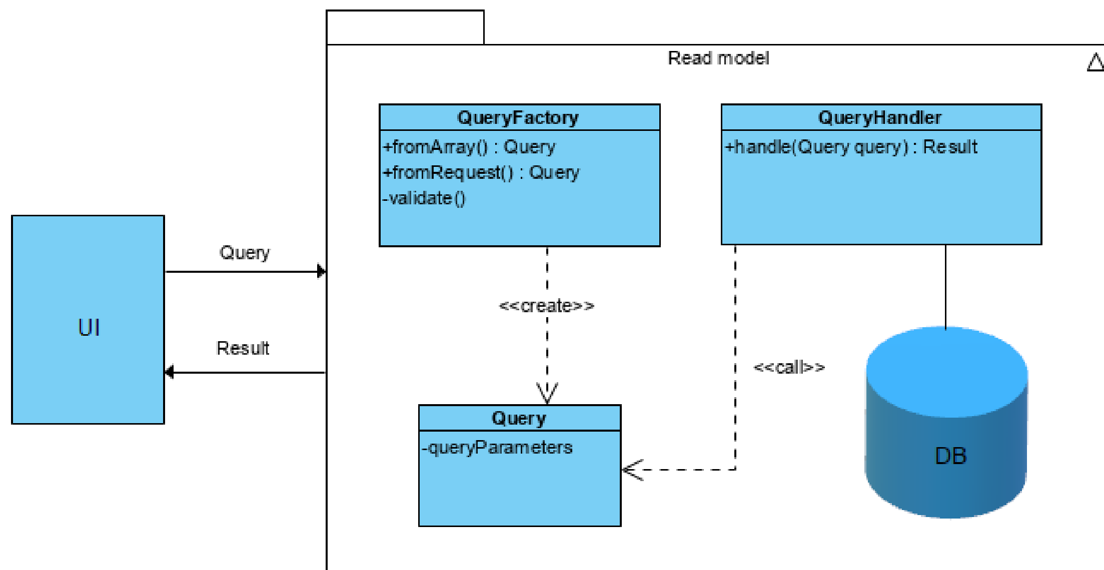
Keskittämällä myös sovelluksen sisäiset kirjoitus- ja lukuoperaatiot moduulien komentoihin ja kyselyihin saadaan moduulien välinen kommunikaatio vastaaman mikropalvelupohjaisen järjestelmän kommunikaatiota.

Kuvassa 15 mallinnettu kirjoitusmalli toteutettiin **CommandFactory-**, **Command-** ja **CommandHandler**-luokilla. CommandFactory-luokka ottaa vastaan parametreja mallin käyttäjältä, validoi ne, ja luo niistä Command-olioita. Command-oliot sisältävät kirjoitettavat datat. CommandHandler-luokka vastaanottaa Command-olioita ja tallentaa olioiden datat tietokantaan.

Kuvassa 16 mallinnettu lukumalli koostuu **QueryFactory-, Query-** ja **QueryHandler**-luokista. QueryFactory-luokka vastaa kirjoitusmallin Factory-luokkaa. Se validoi käyttäjän parametrit ja rakentaa niistä Query-olioita. QueryHandler-luokka taas vastaanottaa Query-olion, muodostaa siitä tietokantakyselyn ja palauttaa tuloksen. Koska lukuoperaatiot eivät muuta järjestelmän tilaa, on validointi vähemmän tiukkaa. Lukuoperaatioiden validoinnilla voidaan antaa palautetta käyttäjälle, mikäli tämä esimerkiksi käyttää kyselyssä väärän tyyppistä parametria.



Kuva 15. Sovelluksen kirjoitusmalli.



Kuva 16. Sovelluksen lukumalli.

Jokaiselle eri luku- ja kirjoitusoperaatiolle määritettiin oma nimiavaruus, joka sisältää omat Factory-, Command/Query- ja Handler-luokkansa. Esimerkiksi Employment-moduuli sisältää StoreEmployeeProfile-kirjoitusoperaation ja GetEmployeeProfiles-lukuoperaation.

5.4 Docker-kontit

Mikropalveluja varten päätettiin siirtyä käyttämään konttiteknologiaa. Teknologiaksi valittiin Docker sen vakiintuneen aseman vuoksi. Kontteja siirryttiin käyttämään sekä kehitys- että tuotantoympäristöissä.

Docker valittiin konttiteknologiaksi, sillä se on vakiintunut alan standardiksi. Julkisissa Docker-hakemistoissa on saatavilla runsaasti valmiita docker-muistinkuvia, mikä mahdollistaa konttiteknologian nopean käyttöönoton.

Yhdenmukaistamalla kehitys- ja tuotantoympäristöt suoraviivaistetaan testaus- ja julkaisuprosessia ja vältytään ympäristöjen eroista johtuvista ohjelmointivirheistä. Docker havaittiin myös kevyemmäksi kuin aiemmin kehitysympäristössä käytetty VirtualBox-virtuaalikone.

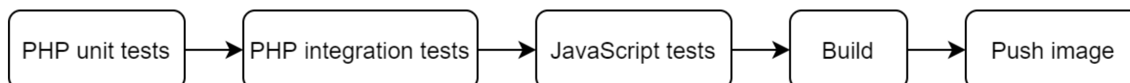
5.5 Jatkuva integraation ja toimitus

Mikropalvelupohjaisen järjestelmän kehittämiseen valmistauduttiin ottamalla käyttöön jatkuvan integraation ja toimituksen ketju. Ketju on toteutettu Docker-konttien, Google Cloud Container Registry -pilvipalvelun, BitBucket -versionhallinnan tietovarastojen säilytyspalvelun ja CircleCI-työkalun avulla.

Google Cloud Container Registry on Google Cloud Platformiin kuuluva palvelu, johon voidaan varastoida docker-muistinkuvia (Google, 2020). Palveluun varastoidaan testeissä käytettävän sovelluksen docker-muistinkuva ja CÌ/CD-ketjun tuottamien julkaisuversioiden docker-muistinkuvat. Tämä antaa valmiuden jatkuvaan toimitukseen, toisin sanoen uusi tuotantoversio voidaan julkaista napin painalluksella.

CircleCI on työkalu CI/CD-ketjun toteuttamiseen (CircleCI Documentation, 2020). CircleCI:hin määritellään työnkulkuja (*workflow)*, jotka voidaan asettaa versionhallinnan haaroihin (branch) Työnkulut koostuvat ajettavista tehtävistä (*job*) jotka koostuvat eri askeleista (*step*).

Tehtäville määritellään tehtävän toteuttamisympäristö ja suoritettavat komennot. Ympäristö koostuu suorittajasta (*executor*), hakemistosta ja konfiguraatiosta. Suorittaja voi olla virtuaalikone tai docker-muistinkuva. CircleCI tarjoaa Linux-, macOS- ja Windows-virtuaalikoneet sekä valmiita docker-muistinkuvia. Hakemistoksi määritellään jokin hakemisto työnkulkuun liitetyn versionhallinnan haaran sisältä. Hakemisto sisällytetään suorittajan virtuaaliympäristöön. Tehtävässä voidaan siis ajaa komentoja määritellyssä virtuaaliympäristössä, joka sisältää sovelluksen koodikannan.



Kuva 17. CI/CD-ketjun tuotantotyökulku.

CircleCI:hin määritettiin kehitys- ja tuotantotyönkulut. Kehitystyönkulku on käytössä kaikilla muilla paitsi julkaisu-versiohaaroilla (*release branch*). Tuotantotyönkulku ajaa ensin testit nopeimmista hitaimpiin. rakentaa sitten tuotannossa käytettävän docker-muistinkuvan sovelluksesta ja lataa sen Googlen pilvipalvelun konttirekisteriin. Tuotantotyönkulun vaiheet on kuvattu kuvassa 17.

Kehitystyönkulussa docker-muistinkuva rakennetaan kehitysympäristön riippuvuuksilla eli tuotannossa käytettyjen pakkausten lisäksi sellaisilla pakkauksilla, jotka on määritetty käytettäväksi pelkästään sovelluksen kehitysympäristössä. Kehitystyönkulkua ei myöskään ajeta automaattisesti joka kerta, kun sitä käyttävään versiohaaraan lisätään muutoksia. Sen sijaan työnkulun vaiheet ajetaan manuaalisesti CircleCI:n verkkosovelluksen kautta. Tämä mahdollistaa pelkkien testien ajamisen. Tästä on kehitystyössä hyötyä, sillä Googlen pilvipalvelu ajaa testit lokaalista kehitysympäristöä nopeammin.

5.6 Nykytilanne

Sovellusta kehitettäessä ilmeni, että järjestelmän siirtäminen mikropalvelukohtaiseksi aiheuttaisi ongelmia pysyä aikataulussa. Kehittäjien aiemman kokemuksen puute mikropalveluarkkitehtuurista kasvatti osaltaan epävarmuutta ja riskiä. Myös arkkitehtuurin hyödyt kyseenalaistettiin. Kokonsa puolesta sovelluksen skaalautumisessa ei tulisi ongelmia pitkään aikaan, ja pienen kokonsa takia kehitystiimi ei myöskään hyötyisi arkkitehtuurin organisaatiotason hyödyistä. Näitten syitten takia mikropalveluarkkitehtuurista luovuttiin ja kehitystä jatkettiin monoliittisella arkkitehtuurilla.

Joistakin mikropalveluarkkitehtuuria varten käytetyistä suunnitteluperiaatteista luovuttiin. Toimialan mukaan moduuleihin jaetusta rakenteesta on kuitenkin pidetty kiinni. Moduulit on pyritty pitämään kiinteinä, mutta riippuvuuksia ei enää rajoiteta Query- ja Command -luokkien tarjoamiin rajapintoihin.

Query-lukumallilla piilotettiin alla oleva tietokantatason integraatio moduulien välillä. Koska monoliitin tietokantaa ei tulla pilkkomaan, ylimääräisestä tietokannan abstrahoinnista moduulitasolla ei ole monoliittisessa sovelluksessa hyötyä. Ennalta määriteltyjen Query-luokkien käyttö oli monissa tilanteissa kankeaa, ja se johti ylimääräiseen koodiin ja tietokannan lukuoperaatioihin. Näitten syitten takia moduulien välisten lukuoperaatioiden rajaamisesta moduulien Query-luokkiin luovuttiin. Kirjoitusoperaatiot toteutetaan edelleen Command-luokkien kautta.

6 Monoliitin pilkkominen ja demoversio

Mikropalveluarkkitehtuurista luopumisen jälkeen päätettiin kuitenkin tutkia mahdollisuutta toteuttaa siirto arkkitehtuuriin tulevaisuudessa. Siirtymisen edellytyksistä, kannattavuudesta ja toteutuksesta toteutettiin selvitys. Sovelluksesta myös toteutettiin pienimuotoinen mikropalvelupohjainen demoversio, jossa demonstroitiin käytännön ratkaisuja.

Selvityksessä kartoitettiin sovelluksen koodikannan riippuvuussuhteet. Selvittämällä kuinka tiukasti moduulit ovat kytkettyjä toisiinsa saadaan arvioitua työmäärää, jonka kunkin moduulin erottaminen mikropalvelluksi edellyttäisi. Kannattavuutta tutkittiin arvioimalla sovelluksen tämänhetkistä tilannetta sekä koodikannan ja skaalaamisen vaatimuksia tulevaisuudessa. Siirtymän toteutuksesta laadittiin suunnitelma, jossa selvitettiin miltä osin ja mitä priorisoiden siirtymä kannattaisi toteuttaa.

Demoversiossa pieni osa monoliittisesta sovelluksesta erotetaan omaksi mikropalvelukseen. Toteuttamalla käytännössä toimiva demoversio pyritään selvittämään mikropalvelupohjaisen järjestelmän minimivaatimuksia.

6.1 Rakenteen analyysi

Moduulien väliset riippuvuudet kartoitettiin etsimällä moduulien koodikannasta moduulin ulkopuolisia luokkia käyttävää koodia. Etsittävät luokat rajoitettiin toisten moduulien luokkiin, eli siinä ei huomioitu yleiskäyttöön tarkoitettuja Common -nimiavaruuteen tehtyjä luokkia eikä sovelluskehyksen apuluokkia. Myös alun perin mikropalvelujen rajapintoja vastaamaan luodut Command- ja Query- luokat jätettiin pois tuloksista. Taulukossa 1 esitetään kartoitetut moduulien väliset riippuvuudet.

Riippuvuussuhde ->	Organization	TimeLog	OrganizationRelationship	Employment
Organization		9	5	7
TimeLog	50		8	16
OrganizationRelationship	4	0		0
Employment	2	2	0	
Permission	0	0	0	0
DrivingLog	5	0	0	2
Identity	1	0	0	1
TimeRule	2	5	0	0

Riippuvuussuhde ->	Permission	DrivingLog	Identity	TimeRule
Organization	2	0	1	3
TimeLog	1	0	2	21
OrganizationRelationship	0	0	0	0
Employment	0	1	1	0
Permission		0	0	0
DrivingLog	0		0	0
Identity	0	0		0
TimeRule	0	0	0	

Taulukko 1. Moduulien välisten riippuvuuksien määrä. Riippuvuuksien suunta on pystyrivin moduuleista vaakarivin moduuleihin.

Huomattiin, että TimeLog- ja Organization-moduulit ovat selvästi tiukimmin muihin moduuleihin kytkettyjä. TimeLog-moduulissa on 98 ja Organization-moduulissa 27 riippuvuutta. Tulos ei ole yllättävä, sillä suurin osa kehitystyöstä mikropalveluarkkitehtuurista luopumisen jälkeen on tapahtunut näissä moduuleissa.

Suurin osa riippuvuuksista koostui Laravelin Eloquent-nimisen ORM-kirjaston (*Object Relational Mapping*) kautta suoritetuista toisen moduulin tietokantatauluihin koostuvista tietokantakyselyistä. Eloquent kuvaa tietokannan taulut erityisiin Model-luokkiin eli malleihin. Mallien kautta pystytään vaivattomasti hakemaan alla olevan tietokantataulun relaatioiden mallit. Tätä ominaisuutta on käytetty laajasti sen helppokäyttöisyyden vuoksi. Esimerkiksi seuraava metodi TimeLogEntry-mallissa määrittelee relaation Organization-mallin kautta time_log_entries -taulun ja organizations-taulun välille, jossa vierasavain sijaitsee time_log_entries -taulussa:

```
public function organization()
	{
	return $this->belongsTo(\Organization\Organization::class);
	}
```

Suoranaisten logiikkaa sisältävien riippuvuuksien määrä rajoittui muutamaan TimeLog-moduulin käyttämään luokkaan. Moduulit ovat siis tiukasti kytkettyjä lähinnä tietokantatasolla. Koska kaikki järjestelmän tietokantakyselyt on tehty Eloquent-mallien kautta, luokkakohtaisten riippuvuuksien kartoitus paljastaa myös tietokantakyselyjen riippuvuudet.

Luokkariippuvuuksien määrä on vain suuntaa antava tieto moduulien mikropalveluiksi erottamiseen vaaditusta työmäärästä. Järjestelmän toiminta perustuu vahvasti Eloquent-mallien kautta tehtyihin tietokantakyselyihin. Pelkkä luokkariippuvuuksien kartoitus paljastaa vain Eloquent-malleihin määritellyt relaatiot, ei siis paikkoja, joissa kyseistä relaatiota on käytetty.

Työmäärä on näin ollen tosiasiassa huomattavasti suurempi, kuin luokkariippuvuuksien määrä antaisi olettaa. Eloquent-malleihin määriteltyjä relaatioita toisten moduulien tauluihin ei ole myöskään järkevää korvata kutsuilla toisiin mikropalveluihin. Käytännössä tämä vaatisi lähes koko Eloquent-järjestelmän uudelleen kirjoittamista. Toisiin mikropalveluihin tehtyjen verkkokutsujen piilottaminen samaan tietokantakyselyjä abstrahoivaan järjestelmään johtaisi myös helposti suureen määrään hitaita palvelujen välisiä kutsuja.

6.2 Lähestymiskohdat

Toiminnanohjausjärjestelmien potentiaalinen toimiala on hyvin laaja. Koska case-sovelluksesta on tarkoitus tehdä useamman yrityksen tarpeisiin vastaava tuote, on vääjäämätöntä, että sovelluksesta tulee laaja ja monimutkainen kokonaisuus. On siis ajan kysymys, milloin sovelluksen koko alkaa haitata kehitystyötä.

Buchgeherin (2017) esimerkkiprojektissa 2 500 tunnin alkuvaiheen kustannukset tarkoittaisivat, että siirtymä mikropalveluarkkitehtuuriin vaatisi yli puoli vuotta yrityksen back end -tiimiltä. Kustannusvaatimus ei ole suoraan verrannollinen, sillä sovellukset eroavat

toisistaan. Esimerkkiprojektiin verrattuna siirtymää ei tarvitsisi aloittaa aivan alusta, sillä joitakin mikropalveluarkkitehtuurin edellytyksiä on yrityksessä jo ehditty ottaa käyttöön. On kuitenkin ilmeistä, että siirtymä veisi paljon aikaa, eikä pienen yrityksen ole järkevää pysäyttää muuta kehitystyötä vaikeasti ennustettavissa olevaksi ajaksi. Inkrementaalinen lähestymistapa monoliitin pilkkomiseen on siis järkevin tapa toteuttaa siirtymä.

Seuraava kysymys on, kannattaisiko koko järjestelmä siirtää mikropalvelupohjaiseksi vai toteuttaa siirtymä vain osittain. Monoliittinen sovellus ei ole vielä kasvanut niin suureksi, että kehitystyö olisi hidastunut. Osittainen siirtymä keskittyen ongelmakohtiin voisi olla siis järkevä lähestymistapa.

Newman (2015a) suosittelee monoliitin pilkkomisen aloittamista kohdista, jotka hyötyvät eniten koodikannan erottamisesta tai jotka ovat helppoja erottaa. Monoliitissa ei ole havaittu kehitystä hidastavia kipukohtia. Suorituskyvyn kannalta on kuitenkin havaittu, että TimeLog-moduulin erilaisten raporttien luonti on ylivoimaisesti raskaimpia operaatioita järjestelmässä. Hyvin suurien raporttien luomista varten on jouduttu kasvattamaan palvelimelle asetettua skriptien maksimisuoritusaikaa. Raportointijärjestelmä voisi olla yksi järkevä kohta, joka voisi hyötyä mikropalveluarkkitehtuurin tehokkaammasta skaalautumisesta. Raportointijärjestelmä on riippuvainen suuresta määrästä sekä TimeLog-moduulin että muiden moduulien dataa, joten suorituskyvyn kannalta jatkuva suurien datamäärien kysely verkkokutsujen kautta voisi aiheuttaa ongelmia.

Helpoimmin erotettavia osia ovat moduulit, jotka ovat vähiten kytkettyjä muhin järjestelmän moduuleihin. Vähiten riippuvuussuhteita sisältävät moduulit olivat Identity (6 luokkariippuvuutta moodulista tai moduuliin), Permission (3) ja DrivingLog (9). Nämä moduulit on luotu kehitystyön alkuvaiheessa, kun monoliittia kehitettäessä pyrittiin vielä huomioimaan siirtymä mikropalveluarkkitehtuuriin, joten ne on pyritty pitämään monoliitista helposti eroteltavina.

6.3 Suunnitelma

Koska monoliitti ei ole vielä kasvanut niin suureksi, että sovellus olisi kärsinyt monoliitin ongelmista, sovelluksen siirtämisestä ei olisi suurta hyötyä. Koodikannan jakaminen mikropalveluiksi tekisi todennäköisesti yksittäisten mikropalveluiden koodin jonkin verran helpommin ymmärrettäväksi, sillä pienemmissä koodikannoissa on vähemmän sisäistettävää. Mikropalveluiden pakottama kova kotelointi myös estäisi teknisen velan kertymistä järjestelmään.

Sen sijaan, että järjestelmä siirrettäisiin mikropalvelupohjaisiksi, siihen kehitettävät uudet ominaisuudet voitaisiin kehittää mikropalveluina. Toteuttamalla uudet ominaisuudet mikropalveluina vältyttäisiin kasvavan monoliitin ongelmista ja toisaalta vältyttäisiin olemassa olevan monoliittisen sovelluksen siirtämiseltä. Monoliitissa mahdollisesti ilmeneviä kipukohtia, kuten raportointijärjestelmää, voitaisiin myös erottaa mikropalveluiksi.

Toinen strategia voisi olla toiminnanohjausjärjestelmän jakaminen useisiin laajempiin sovelluksiin suoranaisen mikropalveluarkkitehtuurin sijasta. Tällä tavalla vältyttäisiin suuren mikropalvelumäärän hallinnoinnin haasteilta sekä suurikokoisen monoliitin ongelmilta. Strategiaa puoltaa se, että yritys ei pienen kokonsa vuoksi suoranaisesti hyödy mikropalveluarkkitehtuurin organisaatiotason hyödyistä. On myös oletettavaa, että sovelluksen käyttäjämäärät eivät kasva niin paljon, että mikropalveluarkkitehtuurin paremmasta skaalautumisesta olisi merkittävästi hyötyä.

6.4 Demoversio

Demoversiossa yksi osa monoliittia erotettiin omaksi mikropalvelukseen. Tavoitteena oli demonstroida mikropalveluarkkitehtuuria käytännössä eikä niinkään toteuttaa tuotannossa käytettävää sovellusta. Demoversiolla pyrittiin myös tutkimaan käytännön toteutuksessa mahdollisesti ilmeneviä haasteita.

Irrotettavaksi osaksi valittiin DrivingLog-moduuli. Moduuli valittiin irrotettavaksi, koska se koettiin helpoimmaksi irrottaa. Muut löyhästi kytketyt moduulit, Permission ja Identity, vastaavat käyttäjien autentikoinnista ja auktorisoinnista. Mikropalvelupohjaisissa järjestelmissä nämä toiminnot toteutetaan tavallisesti ainakin osittain erityisellä API-yhdyskäytävällä, joka autentikoi ja auktorisoi käyttäjät ennen pyyntöjen ohjaamista palveluille. Näiden moduulien siirto mikropalveluiksi olisi siis monimutkaisempaa kuin puhtaasti business-logiikkaa sisältävän DrivingLog- moduulin.

Moduuli irrotettiin vaiheittain. Ensin moduuli irrotettiin monoliitin sisällä loogisesti koodikannasta niin, että ainoat riippuvuudet olivat tulevan mikropalvelun rajapintaa vastaavat Command- ja Query-luokat. Riippuvuuksien vähäisen määrän vuoksi tämä oli vaivatonta. Testejä päivitettiin sisältämään uudet Command- ja Query-luokat, ja erottelun onnistuneisuudesta varmistuttiin ajamalla testit.

Seuraavaksi moduuli irrotettiin monoliitista omaksi mikropalvelukseen. Moduulin koodikanta siirrettiin omaan docker-konttiinsa. Jokaisesta Command- ja Query-operaatiosta tehtiin päätepiste mikropalvelun rajapintaan.

Tässä vaiheessa uusi mikropalvelu oli vielä kytketty monoliitin tietokantaan. Tietokannassa olevat DrivingLog-moduulin taulut olivat pysyneet selvästi erossa muista tauluista, joten taulujen siirtäminen omaan tietokantaan oli yksinkertaista. Monoliitin tietokannasta poistettiin kaikki vierasavaimet mikropalvelun tauluihin ja taulut siirrettiin mikropalvelun omaan tietokantaan.

Koska monoliitti vastaa autentikoinnista ja auktorisoinnista, kutsut mikropalveluun ohjataan monoliitin kautta. Kutsuja monoliitista mikropalveluun ei siis tarvitse erikseen autentikoida. Kutsut mikropalvelusta monoliittiin autentikoidaan monoliitin sisältämän oauth-serverin kautta.

Mikropalvelussa käytettiin Lumen-sovelluskehystä. Lumen on Laravel-sovelluskehykseen perustuva keveyteen ja nopeuteen pyrkivä minimalistinen sovelluskehys, joka on suunniteltu mikropalveluja varten. Lumen valittiin, koska se on laajalti yhteensopiva monoliitissa käytetyn Laravel-pohjaisen koodin kanssa. Rajapinta toteutettiin REST-arkkitehtuurimallia käyttäen, ja kommunikaatio tapahtuu http-protokollaa käyttäen. REST ja http valittiin, sillä monoliitin nykyinen rajapinta oli toteutettu näitä teknologioita käyttäen. Siirtyminen tapahtumapohjaiseen kommunikaatioon olisi vaatinut järjestelmään laajempia muutoksia.

7 Tulokset

Ensimmäisessä vaiheessa mikropalveluarkkitehtuurin testialustana käytetty sovellus päädyttiin toteuttamaan monoliittisena. Ensimmäisessä vaiheessa otettiin kuitenkin käyttöön mikropalveluarkkitehtuuria tukeva jatkuvan integraation ja toimituksen prosessiketju sekä sovelluksen ajaminen docker-konteissa.

Toisessa vaiheessa monoliittisena toteutetun toiminnanohjausjärjestelmän siirtämisestä mikropalvelupohjaiseksi tehtiin selvitys. Selvityksessä tutkittiin, kannattaisiko sovellus siirtää mikropalvelupohjaiseksi.

Sovelluksesta luotiin pienimuotoinen demoversio, jossa yksi osa järjestelmää irrotettiin monoliitista omaksi mikropalvelukseen. Irrotettava osa oli työmatkojen kirjaamiseen käytetty DrivingLog-moduuli. Moduuli oli kehitetty ennen mikropalveluarkkitehtuurista luopumista, joten se oli löyhästi kytketty muuhun monoliittiin. Kiinteän, löyhästi kytketyn monoliitin osan irrottaminen mikropalveluksi oli yksinkertainen toimenpide.

Työn ensimmäinen tutkimuskysymys oli, *kannattaisiko yrityksen ottaa käyttöön mikropalveluarkkitehtuuri case-projektissa tai samankaltaisissa projekteissa*? Vastaus tähän kysymykseen on: ei yrityksen nykyisessä tilanteessa.

Sovellusta kehitettiin aluksi ajatuksena toteuttaa se mikropalvelupohjaisena. Mikropalveluarkkitehtuurista luovuttiin kahdesta syystä. Ensinnäkin pelkästään mikropalvelusuuntaisen monoliitinkin kehittäminen oli hitaampaa kuin puhtaasti monoliittisen ja toiseksi mikropalveluarkkitehtuurin hyödyt yrityksen näkökulmasta kyseenalaistettiin.

Monoliittisen sovelluksen kehittäminen on alkuvaiheessa nopeampaa kuin mikropalvelupohjaisen (Fowler, 2015b). Arkkitehtuurin alkuvaiheen hitaus havaittiin jo mikropalvelusuuntaista monoliittia kehittäessä. Tietokantatason riippuvuuksien välttäminen johti joustamattomaan tapaan tehdä tietokantakyselyitä. Myös jatkuva moduulien välisten riippuvuuksien välttäminen hankaloitti kehitystyötä. Jälkikäteen voidaan myös havaita,

että palvelurajat olisivat vaatineet uudelleen määrittelyä, mikäli sovellus olisi toteutettu mikropalvelupohjaisena: esimerkiksi TimeLog- ja TimeRule-moduulit ovat tiukasti kytkettyjä toisiinsa.

Newman (2019) ei suosittele mikropalveluarkkitehtuuria tästä syystä startup-yrityksille. Hänen mukaansa mikropalveluarkkitehtuuri ratkaisee enemmänkin asemansa vakiinnuttaneen yrityksen ongelmia, kun sovelluksia tulee skaalata. Mikropalveluarkkitehtuuria käyttävät yritykset ovat pääasiassa suuria (Netflix, Guardian, Spotify). Case-sovellus ei tule oletettavasti pitkään aikaan kärsimään monoliittisen arkkitehtuurin ongelmista. Pitkällä tähtäimellä on kuitenkin oletettavaa, että case-sovellus kasvaa suuren toimialansa vuoksi riittävän suureksi hyötyäkseen mikropalveluarkkitehtuurista. Vaikka arkkitehtuurin käyttö olisikin pitkällä aikajänteellä kannattavaa, yritykselle on nuoren ikänsä ja pienen kokonsa vuoksi tärkeää saada nopeasti toteutettua toimivia järjestelmiä.

Tulevia projekteja ajatellen huomioitavaa on kehitettävä sovellus ja yrityksen tilanne. Parhaiten mikropalveluarkkitehtuurista hyötyvät sellaiset sovellukset, joiden toimiala on riittävän laaja, niin että monoliitin ongelmat tulevat esille. Myös tuotteena tehdyt sovellukset sopivat paremmin mikropalveluarkkitehtuuriin, koska yrityksellä on täysi kontrolli sovelluksesta. Tämä mahdollistaa pitkäjänteisen kehityksen. Yrityksen tilanteen kannalta tulee huomioida monoliittista sovellusta hitaampi ja sitä kautta kalliimpi alkuvaiheen kehitys.

Toinen tutkimuskysymys oli, *mitä käytännön asioita on huomioitava mikropalvelupohjaista järjestelmää kehittäessä*? Huomioitavat asiat voidaan jakaa kahteen osaan: itse sovellukseen liittyviin ja sovelluksen ympärillä toimivaan hallinnointiin liittyviin. Sovellukseen liittyvistä asioista saatiin hyvä kuva case-sovelluksen ja demoversion kautta. Hallinnoinnin haasteiden selvittämiseen olisi tarvittu kokonainen tuotannossa toimiva mikropalvelupohjainen järjestelmä, joten niiden selvittämisessä jouduttiin nojautumaan kirjallisuuskatsaukseen.

Demoversion kehitys onnistui nopeasti ja ongelmattomasti. Helppous johtui siitä, että erotetun monoliitin osan toteutuksessa oli pyritty siihen, että se olisi helposti erotettavissa. Mikäli mikropalveluarkkitehtuuria kehitetään monoliitti ensin -strategialla, on olennaista toteuttaa sovellus ’mikropalvelusuuntaisena’. Mikropalvelusuuntaisessa toteutuksessa koodikanta ryhmitellään toimialan mukaan moduuleihin. Toimialojen mukaan jaetut moduulit pyritään pitämään mahdollisimman kiinteinä ja löyhästi kytkettyinä muihin moduuleihin, ja moduulien välinen kommunikointi rajoitetaan tiettyihin rajapintoihin.

Puhtaasti monoliittisena toteutetun järjestelmän mikropalvelupohjaisena toteuttamisen vaatima työmäärä riippuu siis järjestelmän rakenteesta. Mikropalveluarkkitehtuurista luopumisen jälkeen eniten kehityksen alla olleet osat olivat olennaisesti vahvemmin kytkettyjä muuhun sovellukseen. Esimerkiksi TimeLog-moduulin irrottaminen monoliitista olisi ollut huomattavasti suuritöisempi suuren riippuvuusmäärän takia. Vahvasti kytkettyä moduulia ei ole myöskään järkevää irrottaa suoraan ilman muutoksia, sillä se johtaisi hajautettuun monoliittiin, josta Brown (2015) varoittaa. Hajautetussa monoliitissa mikropalvelut ovat tiukasti kytkettyjä toisiinsa, mikä johtaa suureen määrään verkkokutsuja sekä vaikeasti muutettaviin mikropalveluihin. Monoliitin rakenteesta riippuen on siis varauduttava kirjoittamaan suuriakin määriä koodikannasta uudelleen.

Pienimuotoisella demoversiolla ei saada kuvaa laajan, tuotannossa toimivan mikropalvelupohjaisen järjestelmän hallinnoinnin haasteista. Lewis ja Fowler (2014) sekä Newman (2015a) painottavat kuitenkin mikropalvelupohjaisen järjestelmän ylläpidon ja hallinnoinnin haasteita ja monimutkaisuutta. Newman (2015a) pitää mikropalvelupohjaisen järjestelmän ylläpidon edellytyksenä automatisoitua jatkuvaa integrointia ja toimitusta sekä lokitietojen keräysjärjestelmän käyttöönottoa.

Käyttöön otetut docker-kontit ja jatkuvan integraation ja toimituksen ketju tukevat hyvin myös monoliittisen sovelluksen kehitystä. Kehitysympäristössä docker-kontit ovat aiem-

min käytettyjä virtuaalikoneita kevyempiä. Käyttämällä samoja docker-kontteja sekä tuotanto- että kehitysympäristöissä vältytään eroavista ympäristöistä johtuvista virheistä. Toiminnanohjausjärjestelmän kaikkien testien ajaminen CircleCI:n kautta pilvipalvelussa on useita minuutteja nopeampaa kuin testien ajaminen paikallisesti. Ketju myös parantaa sovelluksen toimintavarmuutta edellyttäen, että testit eivät löydä ohjelmistovirheitä ennen kuin sovellus voidaan julkaista.

8 Johtopäätökset

Työn päätarkoituksena oli selvittää, kannattaisiko yrityksen käyttää mikropalveluarkkitehtuuria case-sovelluksessa tai tulevissa samankaltaisissa ohjelmistoprojekteissa. Toisena tavoitteena oli selvittää mikropalvelupohjaisen sovelluksen käytännön edellytyksiä.

Työssä päädyttiin siihen, että mikropalveluarkkitehtuuri ei sovellu yritykselle sen tämänhetkisessä tilanteessa. Mikropalveluarkkitehtuuri on pitkän aikavälin sijoitus, joka alussa hidastaa kehitystyötä merkittävästi mutta nopeuttaa kehitystyötä sovelluksen kasvaessa riittävän monimutkaiseksi. Alkuvaiheen kehityksen hitaus havaittiin case-projektin kehitystyössä. Buchgeherin (2017) esittelemän AMS Engineering -yrityksen mikropalvelupohjaisen projektin 2 500 tunnin alkuvaiheen kustannukset toimivat suuntaa antavana arviona siitä, kuinka paljon case-sovelluksen toteutus mikropalvelupohjaisena olisi loppuun vietynä maksanut. Toteutettu mikropalvelupohjaisen järjestelmän demoversio on liian yksinkertainen, että kehitystyön nopeutta monimutkaisessa sovelluksessa voitaisiin analysoida sen kautta, mutta kirjallisuuskatsauksessa havaittiin yleinen konsensus, että mikropalveluarkkitehtuuri nopeuttaa kehitystyötä sovelluksen kasvaessa riittävän monimutkaiseksi (esim. Fowler 2015; Newman 2015a). Työn toimeksiantajayritykselle on nuoren ikänsä ja pienen kokonsa vuoksi tärkeämpää saada kehitettyä sovelluksia nopeasti.

Case-sovelluksen havaittiin kuitenkin soveltuvan hyvin mikropalvelupohjaiseksi. Case-sovelluksena käytetyn toiminnanohjausjärjestelmän kaltaiset pilvipalveluna toteutettavat sovellukset ovat hyvin mikropalvelupohjaiseksi soveltuvia ohjelmistoja. Kuten case-sovellusta kehitettäessä huomattiin, toiminnanohjausjärjestelmien toimiala on tavanomaisesti riittävän laaja, että järjestelmät hyötyvät kompleksisuutensa takia mikropalveluarkkitehtuurista. Kirjallisuuskatsauksessa havaittiin, että mikropalvelupohjainen järjestelmä hyötyy pilvipalvelupohjaisuudesta, sillä se mahdollistaa helpomman skaalauksen ja hallinnoinnin (Newman, 2015a). Case-sovelluksen järjestelmä myös toteutettiin tuotteena, mikä mahdollistaa kehityksen pitkällä aikavälillä. Mikäli ohjelmistoprojekti tuotetaan tilaustyönä asiakkaalle, voidaan mikropalveluiden käyttämisen edellytyksenä pitää

asiakkaan sitoutumista projektin pitkäaikaiseen kehitykseen, sillä lyhyissä projekteissa mikropalveluarkkitehtuurin hyödyt jäävät saavuttamatta.

Käytännön edellytyksiä pyrittiin tutkimaan toteuttamalla case-sovellus mikropalvelupohjaisena. Tavoitteesta jouduttiin luopumaan aikataulusyitten takia, mikä sinällään vahvistaa käsityksen arkkitehtuurin korkeasta lähtökustannuksesta. Sen sijaan toteutettiin pienimuotoinen demoversio, jossa yksi osa sovellusta irrotettiin mikropalveluksi. Yhden ohjelmiston moduulin erottaminen mikropalveluksi onnistui nopeasti. Yksinkertaisuutensa takia voidaan kuitenkin katsoa, että demoversio antaa vajavaisen kuvan arkkitehtuurin todellisista haasteista. Edellytyksiä selvitettäessä nojauduttiinkin siksi vahvasti kirjallisuuskatsaukseen. Demoversio kuitenkin havainnollistaa sen, että monoliitin mikropalveluiksi pilkkomisen työmäärä riippuu suuresti monoliitin rakenteesta. Valittu irrotettava osa oli kiinteä ja löyhästi kytketty muuhun sovellukseen ja siten helposti irrotettavissa.

Tutkimuksen keskeisin tulos on yleistettävissä muihinkin yrityksiin, joilla ei ole varaa korkeisiin alkuvaiheen kustannuksiin ja uuden teknologian käyttöönottoon liittyvään riskiin. Useat pk-yritykset ovat tällaisia yrityksiä. Etenkin startup-yrityksillä on tavallisesti rajalliset resurssit ja tämän vuoksi kiire saada kassavirtaa tuottavia sovelluksia tuotantoon. Yleisesti ottaen mikropalveluarkkitehtuuria harkitsevien yritysten tulee ottaa huomioon korkeat alkuvaiheen kustannukset ja hajautettujen mikropalvelujen hallinnoinnin monimutkaisuus. Koska mikropalveluarkkitehtuurin hyödyt tulevat esille vasta laajoissa ja monimutkaisissa sovelluksissa, mikropalveluarkkitehtuuri ei ole ratkaisu, jota kannattaisi käyttää joka sovelluksessa ja yrityksessä.

Sovellusten kannalta mikropalveluarkkitehtuuri soveltuu toiminnanohjausjärjestelmien lisäksi muihinkin laajoihin ja pitkäaikaisiin ohjelmistoprojekteihin. Olennaista on, että arkkitehtuurin hyödyt tulevat esille vasta sovelluksen kasvaessa riittävän suureksi. Mikropalveluarkkitehtuuria käytetään ratkaisuna tilanteisiin, jossa järjestelmän hallitsemi-

nen monoliittisena olisi liian vaativaa (Fowler, 2015a). Tiettyä määritelmää riittävän suurelle koolle ei ole, vaan se riippuu jo olemassa olevan järjestelmän siirtoa mikropalvelupohjaiseksi harkittaessa järjestelmän rakenteesta ja teknisen velan määrästä.

Case-projektin alussa käytetty lähestymistapa oli aloittaa projekti monoliittisella sovelluksella ja siirtää se tarpeen vaatiessa mikropalvelupohjaiseksi. Lähestymistapa havaittiin toimivaksi ensimmäistä mikropalvelupohjaista järjestelmäänsä kehittävälle yritykselle. Lähestymistavalla saatiin minimoitua arkkitehtuurin riskejä, sillä näin toteutettuna projekti ei ole tiukasti sidottu mikropalveluarkkitehtuuriin ennen lopullista siirtymää. Kun mikropalveluarkkitehtuuri todettiin kannattamattomaksi työn toimeksiantajayritykselle, voitiin jatkaa suoraan monoliittisen sovelluksen kehitystä. Työtunteja meni hukkaan huomattavasti vähemmän, kuin jos case-sovellus olisi toteutettu alusta asti mikropalvelupohjaisena. Monoliittisena toteutettu ensiversio sovelluksesta saatiin myös nopeammin tuotantoon. Toisaalta tämän lähestymistavan haittana ovat siirtymän aiheuttamat lisäkustannukset, kuten siirtymisen suunnitelmaa kartoitettaessa huomattiin.

Tässä työssä toteutettu mikropalvelupohjaisen järjestelmän demoversio ei suppeutensa vuoksi anna kovin laajaa kuvaa mikropalveluarkkitehtuurin käytännön haasteista. Jatkotutkimuksena voitaisiin tutkia laajaa, jo tuotannossa toimivaa tai tuotantoon siirtymässä olevaa mikropalvelupohjaista järjestelmää. Kuten tässäkin työssä, jatkotutkimuksissa voitaisiin keskittyä mikropalveluarkkitehtuuriin eritysesti pk-yritysten näkökulmasta.

Tutkittavia asioita voisivat olla täysipainoisen mikropalvelupohjaisen järjestelmän käyttöönoton käytännön toteutus ja siinä huomioitavat asiat tai järjestelmän hallinnointi tuotannossa. Myös järjestelmän vaatimia panostuksia ja vaikutuksia kustannuksiin sekä lyhyellä että pitkällä aikavälillä voitaisiin tutkia. Tarkempia teknisiä näkökulmia tutkimukselle voisivat olla esimerkiksi hajautettu tiedon varastointi tai mikropalveluiden väliset kommunikaatiomenetelmät.

Lähteet

Annett, R. (2014). What is a Monolith? Noudettu 15.12.2019. http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html

Brandon, B. (2013). Enterprise integration using rest. Noudettu 28.11.2019. https://martinfowler.com/articles/enterpriseREST.html

Brooks, F. (1987). No silver bullet - Essence and accidents of software engineering. *IEEE computer*, *20*(4), s. 10–19.

Brown, S. (2014). Big balls of mud. Noudettu 13.1.2020. http://www.codingthearchitecture.com/2014/07/06/distributed_big_balls_of_mud.html

Buchgeher, G., Winterer, M., Weinreich, R., Luger, J., Wingelhofer, R., & Aistleitner, M. (2017, September). Microservices in a Small Development Organization. In *European Conference on Software Architecture* (s. 208–215). Springer, Cham.

Conway, M. E. (1968). How do committees invent. *Datamation*, *14*(4), s. 28–31.

Daigneau, R. (2012). *Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services*. Addison-Wesley.

Dehghani, Z. (2018). How to break a Monolith into Microservices. Noudettu 16.1.2020. https://martinfowler.com/articles/break-monolith-into-microservices.html

Docker Foundation. (2019). Docker Overview. Noudettu 5.3.2020. https://docs.docker.com/engine/docker-overview/

Eaves, J. (2014) The Odyssey: from monoliths to microservices at realestate.com.au.

Noudettu 4.12.2019. https://yowconference.com/talks/jon-eaves/yow-2014-sydney/the-odyssey-from-monoliths-to-microservices-at-realestatecomau-11079/

Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

Fowler, M. (2003). Who needs an architect? *IEEE Software*, (5), 11–13.

Fowler, M. (2006). Continuous integration. Noudettu 24.1.2020. https://martinfowler.com/articles/continuousIntegration.html.

Fowler, M. (2014). Bounded Context. Noudettu 8.1.2020. https://martinfowler.com/bliki/BoundexContext.html.

Fowler, M. (2015a). Microservices Premium. Noudettu 10.1.2020. https://martinfowler.com/bliki/MicroservicePremium.html.

Fowler, M. (2015b). Monolith First. Noudettu 15.1.2020. https://martinfowler.com/bliki/MonolithFirst.html.

Fowler, M. (2015c). Microservice Trade-Offs. Noudettu 20.1.2020. https://martinfowler.com/bliki/MonolithFirst.html.

Fowler, S. J. (2016). Production-ready microservices: Building standardized systems across an engineering organization. " O'Reilly Media, Inc.".

Google (2020). Google cloud platform documentation. Noudettu 24.2.2020. https://cloud.google.com/docs.

IEEE Architecture Working Group. (2000). IEEE recommended practice for architectural description of software-intensive systems. *IEEE std*, *1471*.

Jaramillo, D., Nguyen, D. V., & Smart, R. (2016, March). Leveraging microservices architecture by using Docker technology. In *SoutheastCon 2016* (s. 1–5). IEEE.

Koskimies, K. & Mikkonen, T. (2005): *Ohjelmistoarkkitehtuurit*. Talentum.

Lewis, J. & Fowler, M. (2014). Microservices: a definition of this new architectural term. Noudettu 8.12.2019. https://martinfowler.com/articles/dont-start-monolith.html.

Martin, R. C. (2002). *Agile software development: principles, patterns, and practices*. Prentice Hall.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, *2014*(239), 2.

Meyer, M. (2014). Continuous integration and its tools. *IEEE software*, *31*(3), s. 14–16.

Newman, S. (2015a). *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc.

Newman S. (2015b). Microservices For Greenfield? Noudettu 14.1.2020. https://samnewman.io/blog/2015/04/07/microservices-for-greenfield/.

Clement, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J. & Little, R. (2010). *Documenting software architectures: views and beyond*. Pearson Education.

Richardson, C. (2014). Microservices: Decomposing Applications for Deployability and Scalability. Noudettu 15.1.2020. http://www.infoq.com/articles/microservices-intro

Richardson, C. (2018). *Microservices patterns*. Shelter Island: Manning Publications.

Rodger, R. J. (2018*). The tao of microservices*. Manning Publications Company.

Rotem-Gal-Oz, A. (2006). Fallacies of distributed computing explained. Noudettu 5.2.2020. http://www. rgoarchitects. com/Files/fallacies. pdf, 20.

Tilkov, S. (2015). Don’t start with a monolith. Noudettu 1.12.2019. https://martinfowler.com/articles/microservices.html.

Todkar, P. (2018) How to extract a data-rich service from a monolith. Noudettu 21.1.2020. https://martinfowler.com/articles/extract-data-rich-service.html#Respectx201catomicStepOfArchitectureEvolutionx201dPrinciple.

Tornhill, A. (2019). Behavioral Code Analysis In Practice: CodeScene Use Cases and Roles. Noudettu 21.1.2020. https://www.empear.com/docs/CodeSceneUseCasesAndRoles.pdf/

Tyszberowicz, S., Heinrich, R., Liu, B., & Liu, Z. (Syyskuu 2018). Identifying Microservices Using Functional Decomposition. In *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications* (s. 50–65). Springer, Cham.