TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ARI MARIN

TOIMINNANOHJAUSJÄRJESTELMÄN RAJAPINNAN PILKKOMINEN MIKROPALVELUIKSI

Diplomityö

Tarkastaja: Professori Kari Systä
Tarkastaja ja aihe hyväksytty
27. syyskuuta 2017

TIIVISTELMÄ

ARI MARIN: Toiminnanohjausjärjestelmän rajapinnan pilkkominen mikropalveluiksi
Tampereen teknillinen yliopisto
Diplomityö, 45 sivua, 0 liitesivua
Joulukuu 2017
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Kari Systä

Avainsanat: opinnäytetyö, erp, mikropalvelut, monoliitti

Tämän opinnäytetyön tavoitteena oli tutkia mikropalveluarkkitehtuuria ja sen sisällyttämistä nykyiseen monoliittiseen järjestelmään. Monoliitin ja mikropalveluiden vertailun jälkeen tarkasteltiin olemassa olevaa REST-rajapintaa sekä mahdollisuutta pilkkoa se mikropalveluiksi.

Monoliittisessa arkkitehtuurityylissä sovellus julkaistaan ja päivitetään yhtenä suurena kokonaisuutena. Mikropalvelut ovat pieniä ja itsenäisiä palveluita, jotka työskentelevät yhdessä toistensa kanssa. Mikropalvelut ovat kokonaisuutena ohjelmistoarkkitehtuurityyli, joka vaatii sovelluksen toiminnallisen jakamisen pieniin, mieluusti yhden asian tekeviin palveluihin. Jaetut palvelut koostavat yhdessä hajautetun järjestelmän. Järjestelmän palvelut voidaan ylläpitämisen ja jakelun helpottamiseksi laittaa esim. Docker-konteihin. Toisin kun monoliittisessa arkkitehtuurityylissä, mikropalveluissa yhden palvelun päivittäminen tai kaatuminen ei vaikuta olennaisesti koko muuhun järjestelmään.

Olemassa olevalle rajapinnalle tehtiin testi pilkkominen. Rajapinnasta eriytettiin yksi yksinkertainen ominaisuus omaksi palvelukseen ja samalla se poistettiin olemasta olevasta rajapinnasta. Tämän jälkeen Spring Cloud-kirjastojen avulla näistä luotiin omat palvelut. Lopputuloksena kaksi itsenäistä palvelua, jotka toimivat niin yhdessä kuin erikseen. Olemassa olevan monoliitin pilkkominen mikropalveluiksi on siis täysin mahdollista ja oikeilla kirjastoilla mahdollisesti jopa helppoa.

ABSTRACT

ARI MARIN: Breaking the ERP system's interface into microservices
Tampere University of Technology
Master of Science Thesis, 45 pages, 0 Appendix pages
December 2017
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Professor Kari Systä

Keywords: thesis, erp, microservices, monolith

This thesis' purpose was to study the microservice architecture and possibility to include this architecture into the current monolithic system. After the monolith and microservices comparison, an existing REST interface and the possibility of breaking it as a microservices, was examined.

A monolithic architecture application is published and updated as one large entity. Microservices are small and independent services that work together with other services. Microservices are the software architecture style that requires the application to be well-distributed to small services that follow single responsibility principle. These services create a distributed system. The services of the system can be placed on Docker Containers, for example, to facilitate maintenance and distribution. Unlike monoliths, in microservices updating or crashing of a single service does not affect the whole system.

Existing interface was subjected to a test splitting. One simple attribute was separated from the interface as an own service, while being removed from being an interface. After that, with the Spring Cloud libraries, these were made as their own services. The result is two independent services that work together as well as individually. Breaking the existing monolith into microservices is therefore perfectly possible, and with the right libraries, it may even be easy.

ALKUSANAT

”Alku aina hankalaa, lopussa kiitos seisoo”

Haluan kiittää professori Kari Systää hyvistä neuvoistaan sekä pitkäjänteisestä ohjauksestaan.

Lisäksi haluan kiittää Oscar Softwaren Mika Muurista diplomityön aiheesta, ainaisesta tsemppaamisesta sekä takapuolelle potkimisesta.

Tampereella, 5.12.2017

Ari Marin

SISÄLLYSLUETTELO

LYHENTEET JA MERKINNÄT

AcuXDBC	tiedonhallintajärjestelmä, joka on suunniteltu integroimaan ACU-COBOL-GT-datatiedostot relaatiotietokanta tyyliseen ympäristöön
AMQP	Advanced Message Queuing Protocol, avoimen standardin sovelluskerroksen protokolla sanomakeskeiseen väliohjelmistoon
Annotaatio	syntaktinen metatieto, joka voidaan lisätä Java-lähdekoodiin
Apache Tomcat	Apache Software Foundationin kehittämä avoimen lähdekoodin web-palvelin ja servlettijärjestelmä
API	engl. Application programming interface, ohjelmointirajapinta
CPU	engl. Central Processing Unit, prosessori
CRUD	Create, Read, Update, Delete, tietokantojen neljä perustoimintoa
DAO	engl. Data Access Object, olio joka tarjoaa abstraktin rajapinnan jonkinlaiseen tietokantaan / vastaavaan pysyvyys mekanismiin
DTO	engl. Data Transfer Object, datansiirto-olio, olio joka kuljettaa dataa prosessien välillä
ERP	engl. Enterprise Resource Planning, toiminnanohjausjärjestelmä eli tietojärjestelmä, jonka avulla voidaan integroida monia yrityksen toimintaan liittyviä toimintoja yhteen
Guava	joukko yleiskäyttöisiä avoimen lähdekoodin kirjastoja Javalle, joita ylläpitää pääasiallisesti Google
Hibernate	ORM-työkalu Javalle (kts. ORM)
HTTP	Hypertext Transfer Protocol, hypertekstin siirtoprotokolla, protokolla, jota selaimet ja WWW-palvelimet käyttävät tiedonsiirtoon
HTTPS	Hypertext Transfer Protocol Secure, viestintäprotokolla turvalliseen viestintään tietokoneverkossa, jota käytetään laajalti Internetissä. HTTPS yhdistää HTTP-protokollan sekä TLS/SSL-protokollan
IDE	engl. Integrated development environment, ohjelma tai joukko ohjelmia, jolla ohjelmoija suunnittelee ja toteuttaa ohjelmistoa
IPC	Inter-process communication, prosessien välinen kommunikaatio, kahden prosessin tai säikeen välinen informaation vaihto
JAR	eng. Java Archive, paketin tiedostomuoto, jota käytetään tyypillisesti monien Java-luokkatiedostojen ja niihin liittyvien metatietojen ja resurssien yhdistämiseksi yhdeksi tiedostoksi
JAX-RS	engl. Java API for RESTful Web Services, API-spesifikaatio, joka tukee Web-palveluiden luomista REST arkkitehtuurin mukaisesti
JDBC	engl. Java Database Connectivity, Java API, joka määrittelee tavan, miten asiakassovellus voi yhdistää ja käyttää tietokantaa
JDK	Java Development Kit
JEE/JavaEE	Java Enterprise Edition
Jersey	avoimen lähdekoodin sovelluskehys, joka on rakennettu JAX-RS:n päälle
JPA	engl. Java Persistence API, sovellusrajapinnan määrittely, joka kuvaa relaatiotietojen hallintaa sovelluksissa, joissa käytetään Javaa
JSON	JavaScript Object Notation, yksinkertainen avoimen standardin tiedostomuoto tiedonvälitykseen
JVM	engl. Java Virtual Machine, abstrakti laskentakone, jonka avulla tietokone voi suorittaa Java-ohjelmia

Koodipohja	engl. codebase, tietyn ohjelmistojärjestelmän, sovelluksen tai ohjelmistokomponentin koko lähdekoodi kokoelma
Load balancer	kuormantasaaja, tarjoaa sovellusta tasaisesti usealta palvelimelta
Merkintäkieli	järjestelmä dokumentin merkitsemiseksi tavalla, joka on syntaktisesti erotettavissa tekstistä
Metakieli	eng. metalanguage, kieli, jota käytetään silloin, kun kyseessä olevasta kielestä keskustellaan tai sitä tutkitaan. Esimerkkinä XML, joka on metakieli, jolla määritetään merkintäkieli
Node	Solmu, erilaisten tietorakenteiden perusosanen. Solmu voi sisältää tietoa ja linkkejä muihin solmuihin ja solmut muodostavat näin jonkinlaisen verkon
ORM	engl. Object-Relational Mapping, ohjelmointitekniikka datan muuntamiseksi yhteensopimattomien järjestelmien välillä käyttäen oliopohjaisia ohjelmointikieliä
PaaS	Platform as a Service, palvelualustan ulkoistaminen
REST	Representational State Transfer, arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen, joka pohjautuu HTTP-protokollaan
RabbitMQ	avoimenlähdekoodin sanomanvälittäjä (palvelu/sovellus)
Sarjallistaminen	datan sarjallistaminen on käsite, jossa jäsennelty data voidaan muuntaa sellaiseen muotoon, että se voidaan jakaa tai tallentaa siten, että sen alkuperäinen rakenne palautettavissa
Sarjallistamiskieli	kieli datan sarjallistamiseen. Käytetyimpiä kieliä ovat esim. JSON, XML ja YAML
Servletti	eng. servlet, Java-ohjelmointikielen luokka, jota käytetään laajentamaan sellaisten palvelimien ominaisuuksia, jotka isännöivät (host) sovelluksia, joihin pääsee pyyntö-vastausohjelmointimallin avulla
SOA	engl. Service-oriented architecture, ohjelmistosuunnittelun tyyli, jossa toiminnot ja prosessit suunnitellaan toimimaan itsenäisinä
STOMP	Streaming Text Oriented Message Protocol, yksinkertainen tekstipohjainen protokolla, joka on suunniteltu toimimaan sanomakeskeisen väliohjelmiston välityksellä
Spring	sovelluskehys, joka tarjoaa kattavan infrastruktuurituen Java-sovellusten kehittämiseen
TLS/SSL	salausprotokolla, jolla voidaan suojata tietoliikenne tietokoneverkossa
TTY	Tampereen teknillinen yliopisto
Toimialavetoinen suunnittelu	eng. Domain-driven design (DDD), ohjelmistokehityksen tyyli, jossa sidotaan toteutus jatkuvasti muuttuvaan malliin
UI	engl. User Interface, käyttöliittymä
URL	Uniform Resource Locator, standardoitu nimeämiskäytäntö internetin ja intranetin kautta saatavilla oleville tiedostoille / verkkosivun osoite
WAR-tiedosto	Web application Archive, Java-sovellusten pakkaamiseen käytettävä tiedostomuoto
XML	Extensible Markup Language, metakieli, jonka avulla määritetään merkintäkieliä, joilla on rakenne
YAML	ihmisen luettavissa oleva datan sarjallistamiskieli tai merkintäkieli. Sitä käytetään yleisesti määritystiedostoissa

1. JOHDANTO

Työn asiakkaana toimii Oscar Software Oy. Oscar Software on vuonna 2005 perustettu yritysten tietojärjestelmiin, ja etenkin toiminnanohjaukseen, erikoistunut yritys. Oscarilla on useita omia tuotteita ja ratkaisuja mutta yrityksen lippulaiva on Oscar Pro-toiminnanohjausjärjestelmä. Vaikka Oscar on vasta 12 vuotta vanha yrityksenä, löytyy taustalta yli 30 vuoden kokemus ja osaaminen. Yritys on viimeisen viiden vuoden aikana tuplannut niin liikevaihtonsa kuin myös henkilöstö määränsä. Henkilöstöä yrityksellä on noin 110.

Nykypäivänä tekniikka kehittyy todella nopeasti. Oscar Software on myös huomannut tämän ongelman; 1990-luvun huipputekniikka alkaa käydä liian kankeaksi, suppeaksi ja rahallisesti jopa kalliiksi. Vastauksena ongelmaan on yritys aloittanut uudensukupolven REST-rajapinnan kehityksen vuonna 2013.

Cloud on sovellus, joka toteuttaa ERP:n businesslogiikan ja tarjoaa tämän logiikan sovellusten käyttöön REST-rajapinnan avulla. Tästä businesslogiikka-rajapinta kombinaatiosta puhutaan yleisesti Cloud-rajapintana. Tämän rajapinnan tarkoituksena on antaa tulevaisuudessa mahdollisuus rakentaa käyttöliittymä lähes millä tahansa tekniikalla mutta edelleen hyödyntää nykyisiä taustajärjestelmiä, joissa jo monen asiakkaan data sijaitsee. Ajallisesti katsoen rajapinta on nuori ja on toistaiseksi keskittynyt suurimmilta osin ainoastaan Oscar Pro-toiminnanohjausjärjestelmään. Nykyisellään rajapinta kattaa vain pienen osan Oscar Pron ominaisuuksista. Nuoruudesta ja ominaisuuksien vähäisestä kattamisesta huolimatta on rajapinta kasvanut jo suureksi monoliitiksi, joka lopulta muuttuu hallittavuudeltaan mahdottomaksi, ellei asialle tehdä jotakin. Tämän ongelman ratkaisemiseksi on mietitty nykypäivänä paljon huomiota saanutta mikropalveluarkkitehtuuria.

Rajapintaa toteutetaan kerrosarkkitehtuurin periaatteiden mukaisesti ja business-logiikka kapseloidaan omaksi kokonaisuudeksi. Tämä kapseloitu business-logiikkakerros on se, jonka haasteita pyritään mikropalveluarkkitehtuurin avulla ratkaisemaan.

Tässä tutkimuksessa keskitytään siihen mikä on mikropalveluarkkitehtuuri ja siihen, kuinka tätä voidaan soveltaa olemassa olevassa projektissa. Tämä yleensä tarkoittaa olemassa olevan monoliittisen järjestelmän pilkkomista pienemmäksi. Tutkimuksen tavoitteena on saada tarkempi kokonaisvaltainen ymmärrys siitä mikä on mikropalveluarkkitehtuuri ja onko mikropalveluarkkitehtuuria mahdollista ottaa mukaan olemassa olevaan projektiin, joka on tehty perinteisemmällä monoliittisella arkkitehtuurilla.

Diplomityössä selvitetään aluksi yleisiä asioita ja avataan hieman teoriaa. Tämän jälkeen vertaillaan monoliittisen- ja mikropalveluarkkitehtuurin hyötyjä sekä haittoja. Syventävän teorian jälkeen perehdytään Oscar Softwaren Cloud-rajapintaan sekä sitä vasten tehtävään cERP-käyttöliittymään. Kappaleessa 6 käydään läpi esimerkin avulla, kuinka nykyisestä monoliittisesta ohjelmistosta voidaan eriyttää yksi palvelu omaksi mikropalveluksi. Tämän jälkeen tarkastellaan pilkkomisen onnistumista ja katsastetaan tulevaisuuden näkymiä. Lopuksi huomiot laitetaan yhteen Yhteenveto-kappaleen alle.

2. YLEISTÄ TEORIAA

Tässä kappaleessa käydään läpi teoriaa, jonka avulla on helpompi sisäistää myöhempien kappaleiden sisältö.

2.1 Ohjelmistoarkkitehtuurit yleisesti

Ohjelmistoarkkitehtuuri on järjestelmän suunnittelun, mallinnuksen sekä toteutuksen prosessi. Se määrittää kehykset sille, kuinka järjestelmää tulisi kehittää sekä ylläpitää [22]. Arkkitehtuuri siis jäsentää järjestelmän ja projektin, jonka avulla järjestelmä kehitettään. Koskimies & Mikkonen [22] ovat todennut "Voidaankin ajatella, että arkkitehtuuri on järjestelmän perustuslaki. Sitä on noudatettava järjestelmää rakennettaessa, ja sitä voidaan muuttaa vain erittäin painavilla perusteilla."

Arkkitehtuureja tarkasteltaessa voidaan käyttää eri näkökulmia. Tietystä näkökulmasta tehty arkkitehtuuri kuvaus on näkymä [22]. Näkymä on järjestelmäelementtien joukon ja niiden välisten suhteiden kuvaus. Nämä näkymät ovat arkkitehtuurin perusta ja näiden kautta arkkitehdin olisi hyvä lähteä suunnittelua tekemään [7]. Ongelmaksi saattaa kuitenkin ilmetä se, että ei ole tiedossa millaisia näkymiä on käytettävissä. Arkkitehdin tulisikin miettiä järjestelmää kolmella tavalla [7]:

1. kuinka järjestelmä on jäsennetty toteutusyksiköiden (koodiyksiköiden) joukoksi
2. kuinka ohjelmistorakenteet vastaavat järjestelmän ympäristöön liittyviä rakenteita
3. kuinka järjestelmä on jäsennetty elementtien joukoksi, jotka ovat ajonaikana dynaamisesti vuorovaikutuksessa keskenään

Kolme yllä mainittua ovat perus-näkymätyyppejä, joita näkymät vastaavat.

Ensimmäinen näkymätyyppi on nimeltään **moduuli näkymätyyppi**. Tämän näkymätyypin näkymät esittävät toteutusyksiköitä vastaavia elementtejä. Toisena näkymätyyppinä on **allokaatio näkymätyyppi**. Näkymät esittävät kuinka järjestelmä liittyy ympäristönsä ei-ohjelmallisiin rakenteisiin. Lopuksi löytyy **ajonaika (runtime) näkymätyyppi** (tunnetaan myös komponentti-ja-liitin näkymätyyppinä). Ajonaika näkymätyypin näkymät esittävät suoritettavia elementtejä ja niiden vuorovaikutuksia [7].

2.2 Prosessien välinen kommunikaatio

Prosessien välinen kommunikaatio (IPC) on kahden tai useamman prosessin välistä kommunikointia eli informaation välitystä. Näille IPC-menetelmille voidaan ajatella kaksi ulottuvuutta; ensimmäinen on se, onko yhteys yksi-yhteen (one-to-one) vai yksi-moneen (one-to-many) ja toinen, onko yhteys synkroninen vai asynkroninen [36]. Taulukossa 2.1. on esitetty erilaisia prosessien välisiä kommunikointimenetelmiä.

Taulukko 2.1 Esimerkkejä erilaisista IPC-menetelmistä [36]

	Yksi-yhteen	**Yksi-moneen**
Synkroninen	Pyyntö/vastaus	—
Asynkroninen	Ilmoitus	Julkaisu/merkintä
	Pyyntö/asynkroninen vastaus	Julkaisu/asynkroniset vastaukset

Taulukon 2.1. menetelmät ovat seuraavanlaisia:

- Pyyntö/vastaus (Request/response): asiakasohjelma lähettää pyynnön palvelulle ja odottaa vastausta.
- Ilmoitus (Notification): asiakasohjelma lähettää pyynnön palvelulle mutta ei odota vastausta (ja vastausta ei välttämättä edes lähetetä)
- Pyyntö/asynkroninen vastaus: asiakasohjelma lähettää pyynnön palvelulle, joka vastaa asynkroonisesti. Asiakasohjelma ei estä suoritusta odottaessaan ja usein toteutus suunnitellaan olettaen, että vastausta ei hetkeen saada
- Julkaisu/merkintä (Publish/Subscribe): asiakasohjelma julkaisee ilmoituksen, jonka nolla tai useampi palvelu nappaa, mikäli ko. palvelu on tästä kiinnostunut
- Julkaisu/asynkroniset vastaukset: asiakasohjelma julkaisee ilmoituksen ja odottaa tietyn aikaa vastauksien kiinnostuneilta palveluilta

2.3 Toiminnanohjausjärjestelmä

Toiminnanohjausjärjestelmä eli ERP-järjestelmä (Enterprise Resource Planning) on yrityksen käytössä oleva järjestelmä tai ohjelmisto, jonka tehtävänä on integroida yrityksen eri toiminnot yhteen, helposti hallittavaan kokonaisuuteen. Toiminnanohjausjärjestelmään voi sisältyä erilaisia toimintoja, kuten palkanlaskenta, kirjanpito, varastonhallinta, tuotannonohjaus. Toiminnanohjauksen avulla mahdollistetaan näiden eri toimintojen keskitetty hallinta ja seuranta. Informaatiota voidaan syöttää missä vaiheessa prosessia tahansa ja se on käytettävissä tämän jälkeen myös kaikkialla muualla yrityksen eri toiminnoissa. Toiminnanohjausjärjestelmän avulla on mahdollista helpottaa yrityksen jokapäiväistä toimintaa; toiminnanohjausjärjestelmän isoimmat hyödyt ja ominaisuudet yritykselle ovat parantunut tuotannon tehokkuus sekä apu tuotannon päätöksentekoon [1] [32].

Toiminnanohjausjärjestelmien kehittyminen on lähtenyt liikkeelle aina 1960-luvulta lähtien [1]. Ensimmäinen askel kohti ERP-järjestelmää oli 1960-luvulla syntynyt varastonhallinta, jonka tarkoitus on varaston toimenpiteiden, kuten tavaran siirtelyn, vastaanoton, hyllytyksen, keräilyn, pakkauksen sekä toimituksen hallinta. Tämä järjestelmä kehittyi entisestään seuraavien kymmenen vuoden aikana. 1980-luvulla järjestelmään tuli lisää ominaisuuksia, jonka johdosta siitä muotoutui varaston- ja tuotannonhallintajärjestelmä. Tarve monipuolisemmalle järjestelmälle oli suuri ja tähän vastattiin jälleen yksi vuosikymmen myöhemmin; 1990-luvulla sai alkunsa ensimmäinen toiminnanohjausjärjestelemä [1]. Kuvassa 2.1 on esitetty toiminnanohjausjärjestelmän historia pähkinänkuoressa sekä katsaus tulevaan.

Kuva 2.1 Toiminnanohjauksen historia aina varastonhallinnasta tulevaisuuden cERP-järjestelmään [1] (kuvaa lainattu Oscar Softwaren luvalla)

Valmiit toiminnanohjausjärjestelmät ovat erittäin laajoja kokonaisuuksia kohdistettuna suurelle joukolle yrityksiä. Järjestelmät voivat olla hyvin monimutkaisia ja hitaita. Tämä aiheuttaa sen, että järjestelmä ei välttämättä sovellu täysin kaikkien yritysten tarpeeseen.

Kuten mitä tahansa ohjelmistoa, voidaan tuominnanohjausjärjestelmää myös räätälöidä asiakkaiden tarpeiden mukaisesti. Tällaiset yrityksen tarpeiden mukaisesti toteutetut toiminnanohjausjärjestelmät antavat yrityksen johdolle ja muille vastuunkantajille selkeät ja ajantasaiset tiedot avuksi päätöksentekoihin. Täten johdolle jää enemmän aikaa niin sanottuun olennaiseen työhön, kun rutiinitoimet eivät sido heitä liikaa [32].

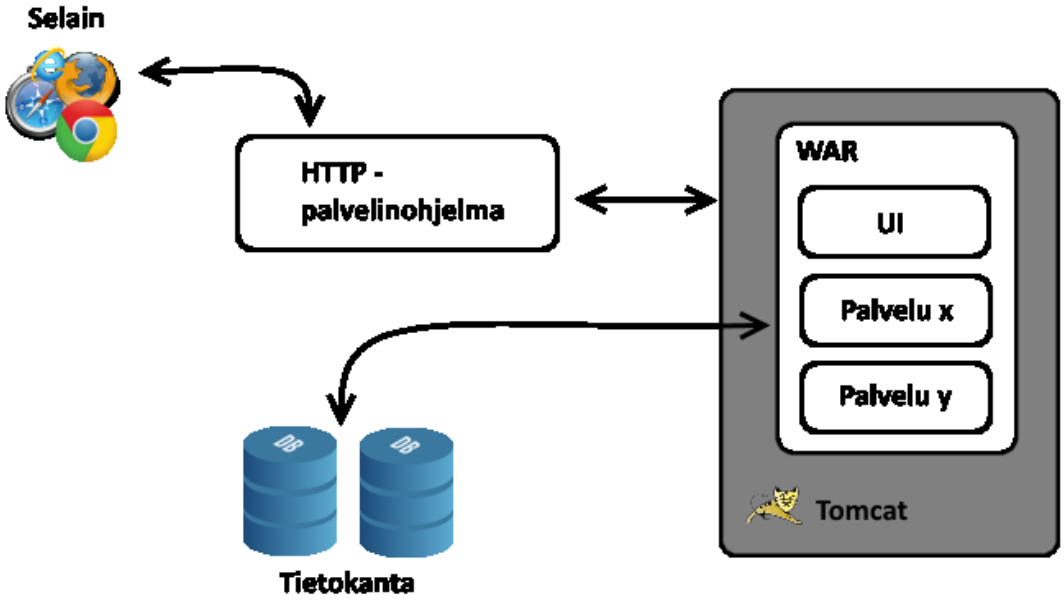
Taulukossa 2.2 on esitetty mahdollisia toiminnanohjausjärjestelmän ominaisuuksia. Mukaan lukeutuu mm. taloushallinto, asiakkuuden hallinta, verkkokauppa sekä pilvipalvelu.

Taulukko 2.2 Toiminnanohjausjärjestelmissä on usein todella kattavat ominaisuudet

Ominaisuus	
Asiakkuuden hallinta	Henkilöstöhallinta
Huolto- ja laitehallinta	Integraatiot ja rajapinnat
Johtamisen välineet	Materiaalihallinto
Taloushallinto	Tilaus- ja toimituskejun hallinta
Tuotannonohjaus ja projektinhallinta	Verkkokauppa

3. MONOLIITTINEN ARKKITEHTUURI

Monoliittinen arkkitehtuurityyli, nimensä mukaisesti, on tyyli, jossa sovellus julkaistaan ja päivitetään yhtenä suurena kokonaisuutena. Sovellus voi olla monoliittinen, vaikka sillä olisi looginen modulaarinen rakenne [34] [35]. Tällaista modulaarista monoliittia kuvastaa hyvin kuva 3.1. Kuvassa on mallinnettu perinteinen verkkokauppa, joka esimerkiksi ottaa vastaan tilauksia asiakkailta, tarkistaa inventaariot sekä käytettävissä olevat luotot ja toimittaa ne. Verkkokauppa koostuu useasta moduulista, kuten käyttöliittymästä sekä esimerkiksi taustapalveluista varasto- ja toimitustilausten ylläpitoon. Nämä moduulit on lopulta pakattu yhdeksi WAR-tiedostoksi, jonka avulla sovellus voidaan julkaista yhtenä isona kokonaisuutena.



Kuva 3.1 Monoliittinen verkkokauppa [35]

3.1 Monoliittisen arkkitehtuurin tyylejä

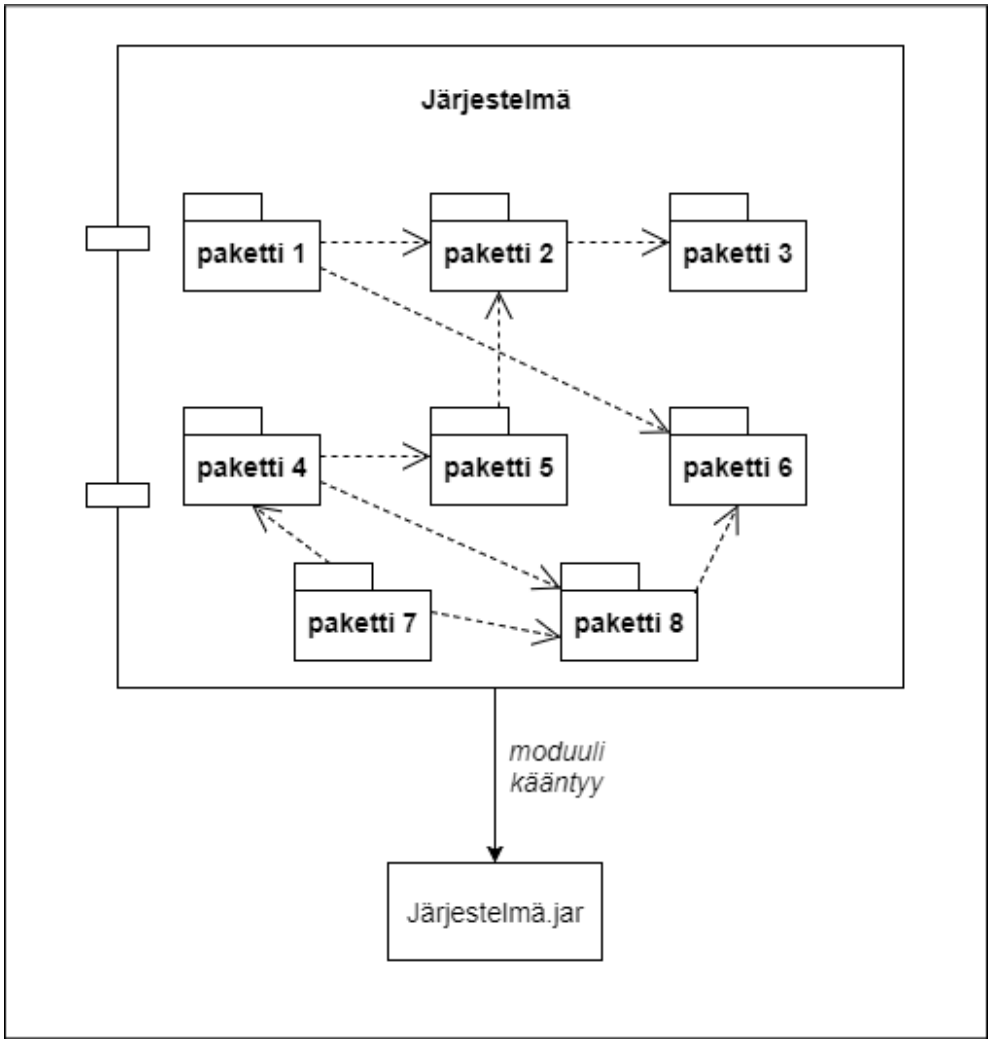
Annettin mukaan monoliittia voi pitää arkkitehtuurisena tyylinä tai ohjelmistokehityksen mallina, jotka voivat sopia erilaisiin näkymätyyppeihin [3]. Perus-näkymätyypit ovat esitetty alustavasti kappaleessa 2. Tässä lisäksi Annettin [3] määritelmät:

- Moduuli näkymätyyppi: koodiyksiköt ja näiden suhteet käännösaikana.
- Allokaatio näkymätyyppi: ohjelmiston liittäminen sen ympäristöön.
- Ajonaika näkymätyyppi: ohjelmistojärjestelmien staattinen rakenne ja niiden vuorovaikutus ajonaikana.

Määritelmiä verratessa aikaisemmin esitettyihin, ovat ne hyvin samankaltaiset. Erona huomataan kuitenkin se, että Annettin määritelmät ovat käytännönläheisempiä.

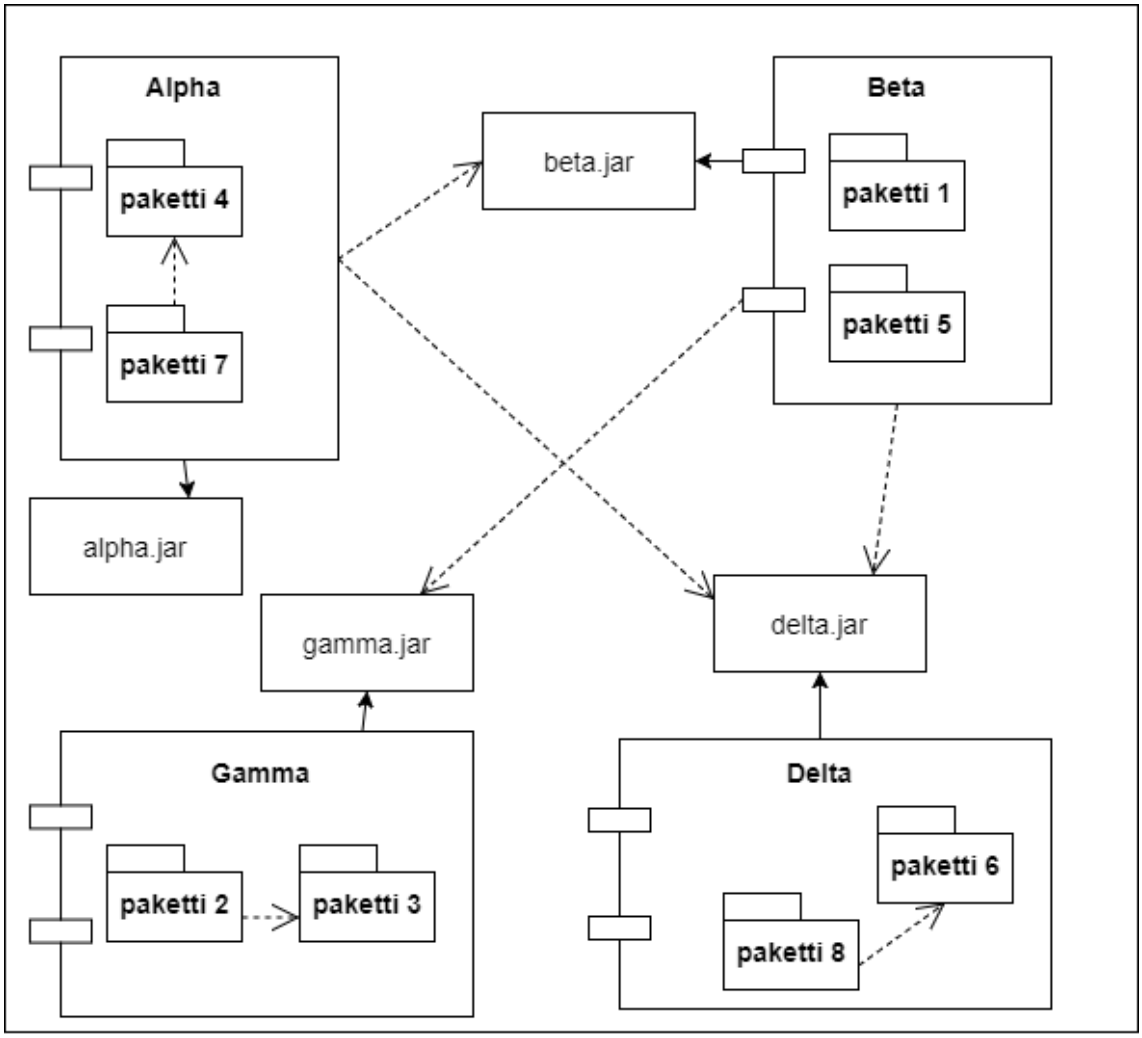
3.1.1 Monoliittinen moduuliarkkitehtuuri

Monoliittisessa moduuliarkkitehtuurissa koodipohja on toteutettu pakettikohtaisina moduuleina. Moduulit ovat sovelluksen itsenäisiä osia, joilla on omat syötteet, tulosteet sekä suoritettava tehtävä. Täten monoliittisessa moduuliarkkitehtuurissa koodi voi olla hyvin jäsenneltynä johdonmukaisiin ja irrallisiin luokkiin sekä paketteihin, mutta koko järjestelmän koodi sijaitsee yhdessä suuressa monoliittisessa kokonaisuudessa, josta valmis sovelluspaketti käännetään. Esimerkki monoliittisesta moduuli arkkitehtuurista on esitetty kuvassa 3.2. Kuvassa on järjestelmä, joka sisältää useita luokka-paketteja ja joka kokonaisuutena kääntyy lopulliseksi järjestelmä-artefaktiksi. Kuvan järjestelmä käyttää siis monoliittista moduuliarkkitehtuuria [3]. Kuvassa olevat katkoviivanuolet näyttävät sen, mistä paketista mikäkin paketti on riippuvainen.



Kuva 3.2 Yksi monoliittinen koodipohja, josta käännettään lopullinen artefakti [3]

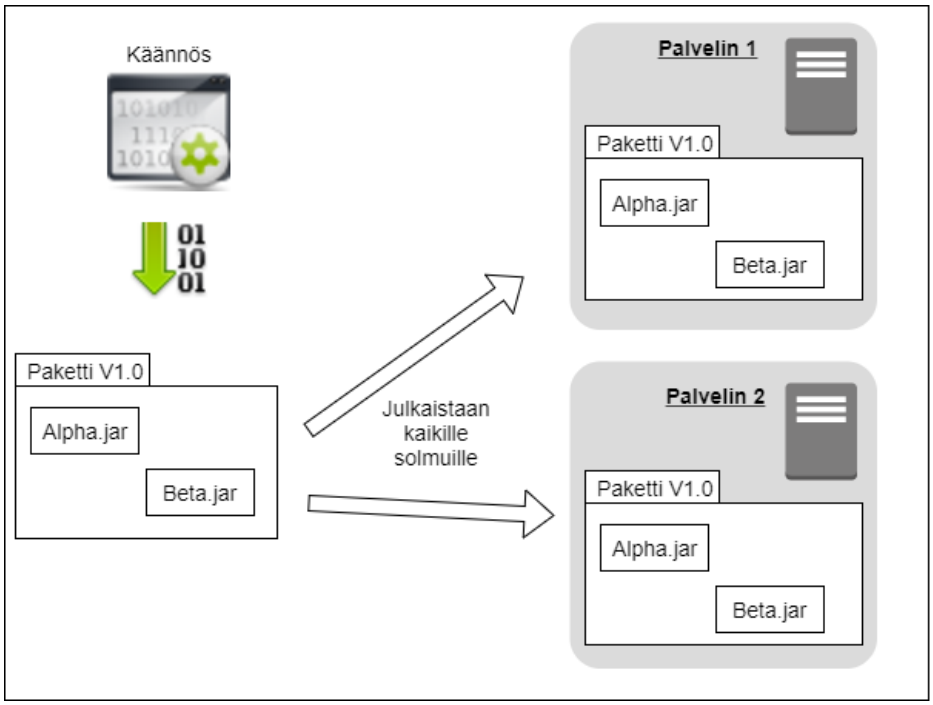
Kontrastina kuvalle 3.2 esittää kuva 3.3 samaisen järjestelmän, jonka koodipohja on jaettu useammaksi jakelupaketiksi [3]



Kuva 3.3 Useampi koodipohja useamman artefaktin kääntämiseen [3]

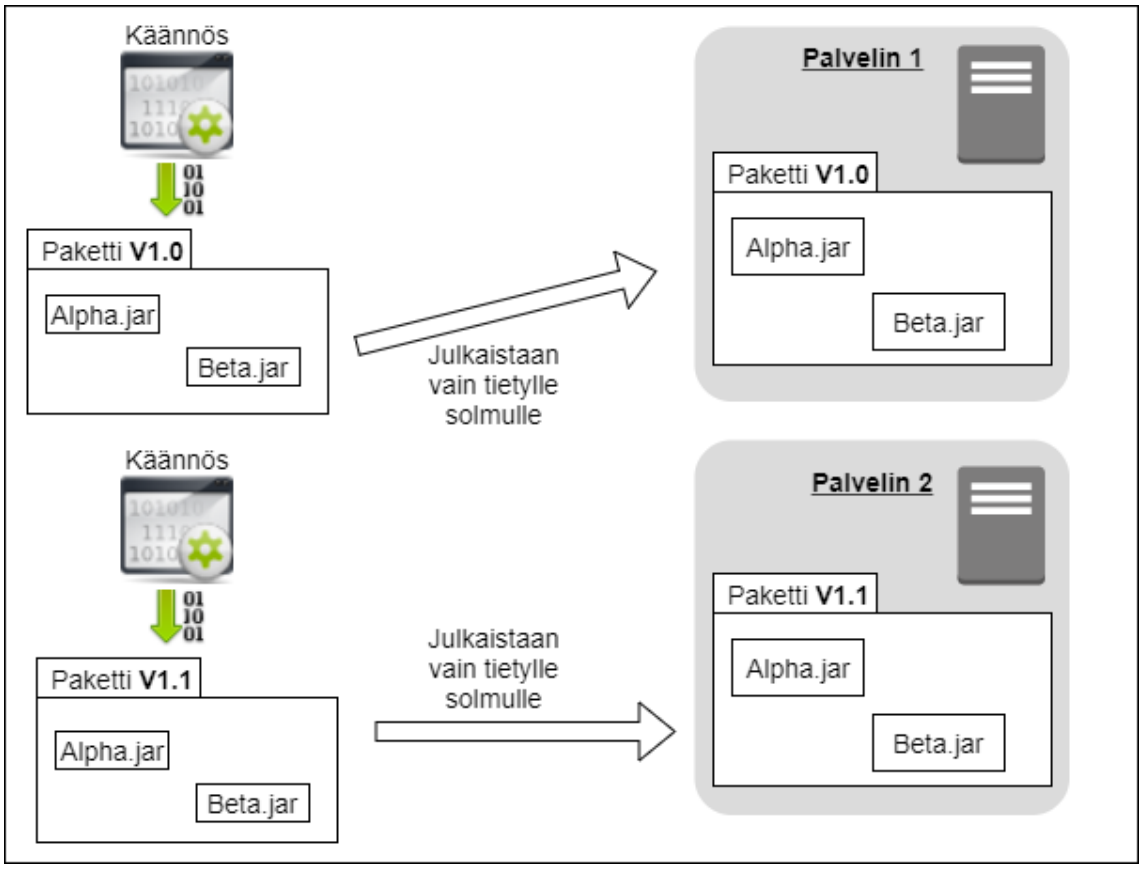
3.1.2 Monoliittinen allokaatioarkkitehtuuri

Monoliittisessa allokaatioarkkitehtuurissa järjestelmä julkaistaan kaikille solmuille yhtenä isona kokonaisuutena ja yhtenä samana versiona; kaikki komponentit ovat aina samassa versiossa, millä ajanhetkellä tahansa. Tämä on esitetty kuvassa 3.4. Kuvassa on järjestelmä, joka käyttää monoliittista allokaatio arkkitehtuuria [3].



Kuva 3.4 Monoliittinen allokointi ja julkaisu [3]

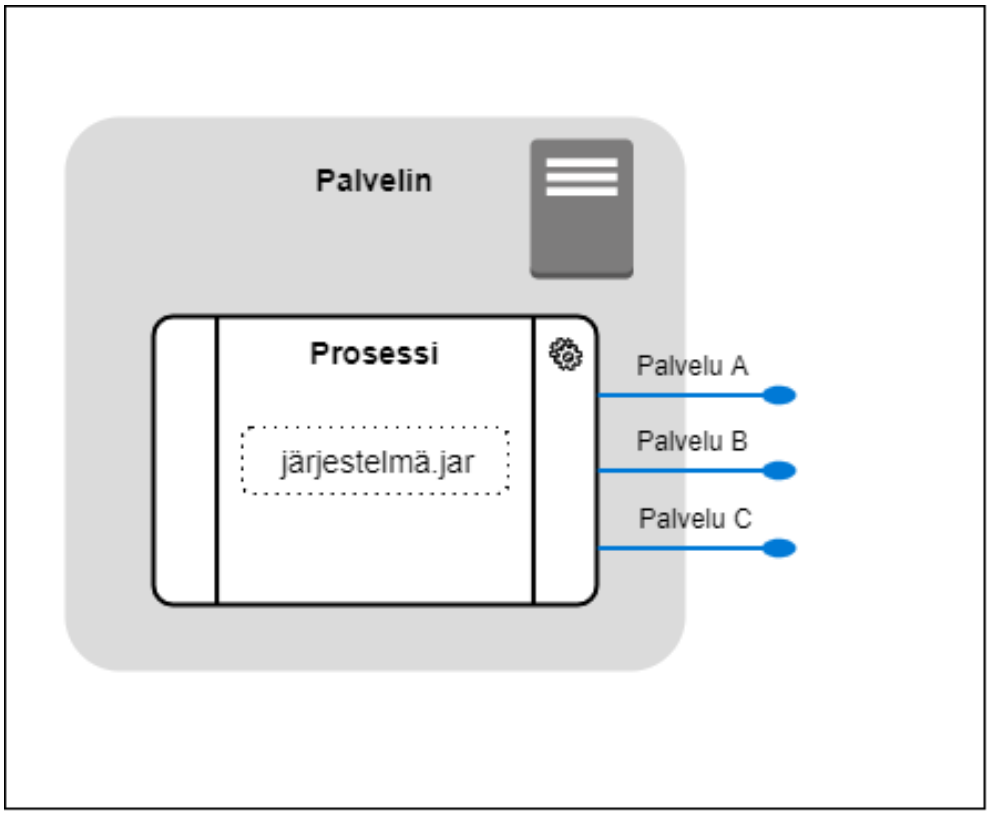
Kuvassa 3.5 on kuvattu edellisen järjestelmän toisenlainen monoliittinen toteutus, jossa järjestelmästä julkaistaan eri versio yksittäisille solmuille [3]. Tätä kutsutaan ei-monoliittiseksi allokoinniksi [3].



Kuva 3.5 Monoliittisten moduulien ei-monoliittinen allokointi [3]

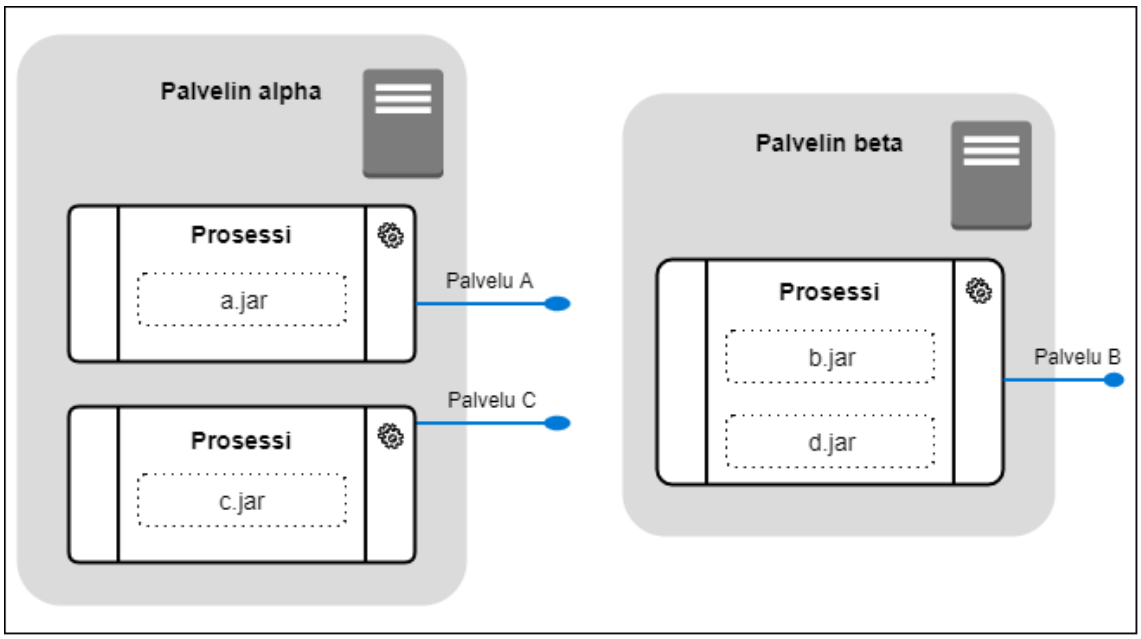
3.1.3 Monoliittinen ajonaika-arkkitehtuuri

Monoliittisessa ajonaika-arkkitehtuurissa järjestelmää ajetaan yhdessä prosessissa, vaikka järjestelmällä olisi, esimerkiksi, ulkoisia riippuvuuksia. Kuvassa 3.6. on esitetty ajonaika monoliitti. Kuvassa on yksittäinen prosessi, joka on vastuussa koko järjestelmästä [3].



Kuva 3.6 Monoliittinen ajonaika monoliittisella moduulilla sekä allokoinnilla [3]

Kuvassa 3.7 on kuvattu ei-monoliitti versio ylläolevasta järjestelmästä. Nyt jokaisella prosessilla on hoidettavanaan vain yksi palvelu.



Kuva 3.7 Yksi palvelu per prosessi, ei-monoliittiset moduulit sekä allokoinnit [3]

3.2 Monoliittisen arkkitehtuurin hyödyt

Monoliittisen arkkitehtuurin hyötyjä ovat:

- Useat kehitys ohjelmat, kuten integroidut kehitysympäristöt (IDE), tukevat tätä tyyliä [15] [35].
- Julkaiseminen on helppoa [1] [12] [15] [35]. Esimerkiksi WAR-tiedoston siirto julkaisu hakemistoon. Tämä myös helpottaa ohjelman asentamista ja testaamista kehittäjän omassa ympäristössä [12] [15].
- Helppo skaalata ajamalla useata kopiota sovelluksesta kuormantasaajan takana [1] [12] [35].

3.3 Monoliittisen arkkitehtuurin ongelmat

Monoliittisella arkkitehtuurilla on useita ongelmia, jotka yleensä nousevat esiin sovelluksen kasvaessa, järjestelmän vaatimusten lisääntyessä ja kehitystiimin kasvaessa tai vaihtuessa [15] [35]. Ketteryyden kärsiminen, tuottavuuden heikentyminen, heikko skaalautuvuus sekä vanhentuneet teknologiat ovat ongelmia, joita esiintyy monoliittisessa arkkitehtuurissa.

Pienen korjaustoimenpiteen tai uudistuksen tekeminen ja julkaiseminen voi viedä aikaa. On mahdollista, että muutoksen teko yhteen paikkaan saa aikaan sen, että tästä toiminnosta riippuva toisaalla oleva koodi tarvitsee myös muokkausta. Kun muokkaukset on tehty ja ne halutaan ottaa käyttöön kehitysympäristön lisäksi myös tuotannossa, täytyy koko sovellus julkaista uudelleen [15] [29] [34]. Tämä ilmenee hyvin varsinkin silloin, kun on pyrkimys julkaista uusia ominaisuuksia tai korjauksia muutamia kertoja päivän aikana; julkaisujen hidas tahti vaikuttaa suoraan ketteryyteen [1] [15] [34] [35].

Tiimin uuden kehittäjän saattaa olla vaikea ymmärtää ja sisäistää koko sovellus sen massiivisen koodipohjan (codebase) takia, varsinkin mikäli modulaarisuus puuttuu kokonaan. Ymmärrettävyys ja ylläpidettävyys heikkenevät entisestään sovelluksen kasvaessa [8] [29] [34] [35]. Sovelluksen suuri kasvu saattaa haitata myös työkaluja. Suuri koodi määrä voi hidastaa huomattavasti IDE:ia, sovelluksen kääntämistä ja julkaisua [1] [35].

Monoliittinen sovellus skaalautuu vain yhteen suuntaan [26]. Käytännössä tämä tarkoittaa sovelluksen kopiointia usealle palvelimelle ja käyttämällä kuormantasaajaa sovelluksen jakamiseen loppukäyttäjälle [1] [12] [35], tai käyttäjien yhteyksien jakamisen sijaan jaetaankin tietokantayhteydet. Nämä tavat eivät kuitenkaan huomioi yksittäisten komponenttien mahdollisia tarpeita; jokin komponentti voi tarvita enemmän prosessoria (CPU) ja toinen taas enemmän muistia [29]. Sovelluksen kompleksisuus myös säilyy, sillä tällä skaalaus tavalla sovellusta vain kopioidaan toisille palvelimille. Täten sovelluksessa oleva ongelma vaikuttaa jokaiseen ajossa olevaan versioon.

Projektissa käytettävä teknologia on usein päätetty projektin alussa, kun kehitystyötä on alettu tekemään. Teknologia kehittyy valtavaa vauhtia ja teknologia, joka valittiin projektin alussa ei välttämättä ole hyvä tai edes millään tasolla tehokas kyseiseen toimintaan nykypäivänä. Monoliitin kanssa teknologiaa on todella vaikea, ellei jopa mahdotonta vaihtaa kesken ilman, että kaikki koodi kirjoitetaan uudelleen [1] [15] [29] [34] [35].

4. MIKROPALVELUARKKITEHTUURI

Kuten mainittiin kappaleessa 3, monoliittisella arkkitehtuurilla on helppo aloittaa kehittäminen. Ajan myötä kuitenkin, kun järjestelmä kasvaa, alkaa tuottavuus sekä ketteryys kärsiä [30] [34]. Monoliitin hyödyt alkavat jäädä ongelmien alle. Ongelmista huolimatta järjestelmää tulisi edelleen pystyä päivittämään sekä ylläpitämään. Näitä ongelmia pystytään lähestymään monella eri tavalla, joista yksi on järjestelmän arkkitehtuurin muutos esimerkiksi mikropalveluarkkitehtuuriin (tai yksinkertaisesti mikropalveluiksi).

Mikropalveluille ei ole yhtä oikeaa määritelmää [12] mutta voidaan todeta, että mikropalvelut ovat pieniä ja itsenäisiä palveluita, jotka työskentelevät yhdessä toistensa kanssa [30]. Lisäksi mikropalvelut ovat kokonaisuutena ohjelmistoarkkitehtuurityyli, joka vaatii sovelluksen toiminnallisen jakamisen pieniin, mieluusti yhden asian tekeviin palveluihin [15]. Palveluiden itsenäisyys näkyy hyvin siinä, että jokainen palvelu on oma entiteettinsä, eli olionsa, jonka voi julkaista esimerkiksi eristettynä PaaS:nä tai omana järjestelmä prosessina [30]. Mikropalveluiden koko ei ole yksikäsitteinen asia [30] ja tästä lisää kappaleessa 4.2.3.

Vaikka mikropalveluilla ei ole yhtä määritelmää, on tällä arkkitehtuurityylillä tehdyillä palveluilla usein samankaltaisia ominaisuuksia [15]:

- Toimialavetoinen suunnittelu: Sovelluksen toiminallisuuden jakaminen. Jokainen tiimi vastaa koko toiminnallisuuden tuottamisesta kyseiselle toimialalle (domain) [15].
- Yhden vastuullisuuden periaate (Single Responsibility Principle): jokainen palvelu on vastuussa vain yhdestä toiminnallisuuden osasta ja tekee sen hyvin [15].
- Selvästi julkaistu rajapinta (API): jokainen palvelu julkaisee selvästi määritetyn rajapinnan [15].
- Itsenäinen julkaisu/päivitys/skaalaus/vaihto: kaikki palvelut ovat täysin itsenäisiä ja esimerkiksi yhden palvelun päivittämisen ei tulisi vaikuttaa muuhun järjestelmään [15].
- Mahdollisesti heterogeeninen: Se kuinka palvelu on toteutettu, ei tulisi kiinnostaa muita palveluita [15].
- Kevyt kommunikaatio: Palvelut kommunikoivat keskenään käyttäen kevyttä kommunikaatiomenetelmää, esimerkiksi REST.

4.1 Mikropalveluiden kommunikointitapoja

Palvelut voivat kommunikoida käyttäen erinäisiä IPC-menetelmiä, jotka käytiin perusteellisemmin läpi kappaleessa 2. Joillekin palveluille riittää näistä menetelmistä yksi, mutta osa saattaa käyttää useampaa menetelmää. Esitetyille menetelmille löytyy useita teknologioita, joiden avulla kyseistä menetelmää voidaan käyttää. Palvelut voivat esimerkiksi käyttää synkronista pyyntö/vastaus-pohjaista kommunikointitapaa, kuten Thrift,

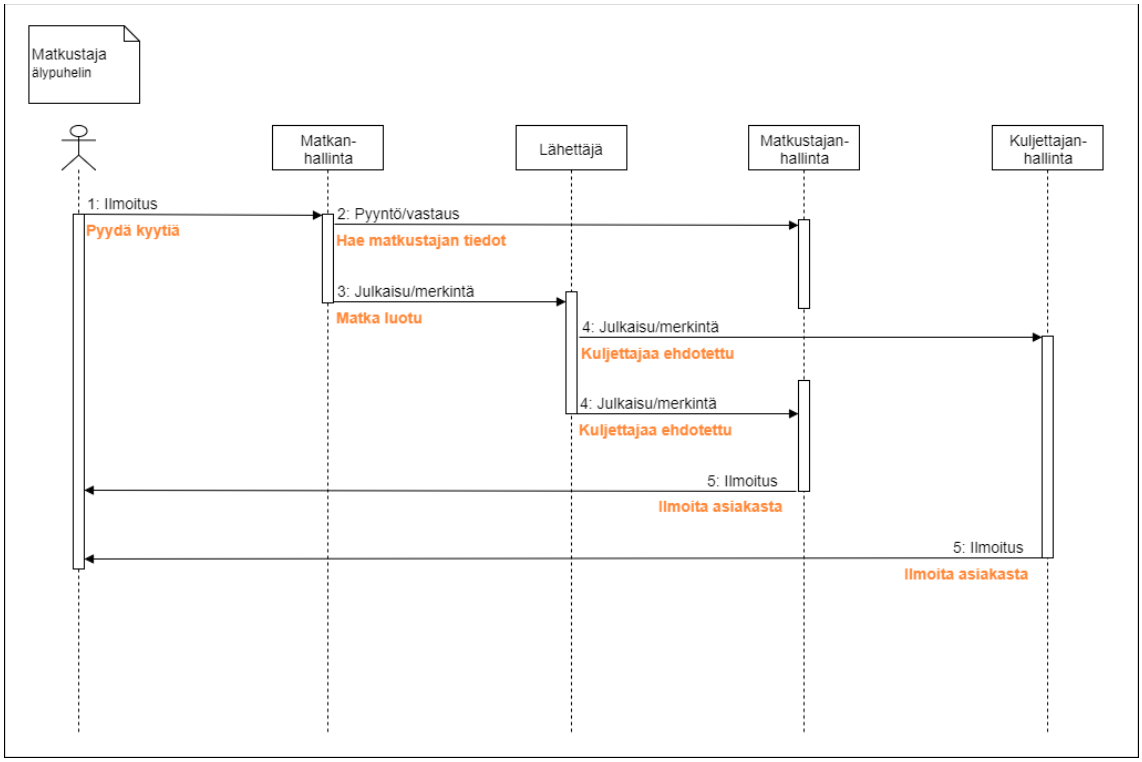
Apachen kehittämä sovelluskehys (framework) skaalautuvien koodikieliriippumattomien palveluiden kehittämiseen, tai REST. Palvelut voivat myös käyttää asynkronista sanomakeskeistä tapaa; yksinkertaista teksti-pohjaista protokollaa nimeltä Streaming Text Oriented Message Protocol (STOMP), joka on suunniteltu toimimaan sanomakeskeisen väliohjelmiston välityksellä tai avoimen standardin sovelluskerroksen protokollaa nimeltä Advanced Message Queuing Protocol (AMQP), joka on tarkoitettu sanomakeskeiseen väliohjelmistoon. Sanomilla on myös useita formaatteja, kuten esimerkiksi ihmisten luettavissa olevaa JSON sekä XML-muotoa tai erinäisiä binäärisiä-muotoja [36].

Synkronisessa pyyntö/vastaus-pohjaisessa kommunikointitavassa asiakasohjelma lähettää pyynnön palvelulle, palvelu prosessoi pyynnön ja palauttaa vastauksen. Usein säie, joka pyynnön suoritti, estää muun toiminnan suorittamisen sillä aikaa, kun paluuviestiä odotetaan. REST on yksi synkroninen pyyntö/vastaus-pohjainen kommunikointitapa ja Williamsin [43] mukaan se on suosituin tapa tehdä mikropalveluita. REST:in pääperiaatteet ovat resurssit, jotka usein kuvaavat oikeita asioita kuten asiakas tai tuote tai näiden kokoelmia. REST:issa resursseja käsitellään CRUD (Create, Read, Update, Delete)-metodien avulla, jotka käyttävät vastaavia HTTP:in verbejä (POST, GET, PUT, DELETE) ja tiettyä URL:ia. Tällaisella HTTP:a tai HTTPS:a käyttävällä tavalla on useita etuja mutta myös haittoja. Etuihin kuuluu esimerkiksi se, että HTTP on yksinkertainen ja tuttu, rajapintaa voidaan helposti testata käyttämällä selainta ja HTTP tukee suoraan pyyntö/vastaus-kommunikointitapaa. Haittana tällaisessa tavassa on se, että HTTP tukee suoraan vain pyyntö/vastaus-kommunikointitapaa, asiakassovelluksen ja palvelun molempien tulee olla koko kommunikoinnin ajan ajossa sekä asiakasohjelman tulee tietää mistä palvelu löytyy (esim. URL) [36].

Asynkronisessa sanomakeskeisessä-kommunikointitavassa asiakasohjelma tekee pyynnön palvelulle lähettämällä sanoman. Palvelun joko odotetaan tai ei odoteta vastaavan sanomaan. Mikäli palvelun odotetaan vastaavan, lähettää se erillisen viestin asiakasohjelmalle. Koska kommunikointi on asynkronista, ei muun toiminnan suorittaminen pysähdy, vaikka paluusanomaa ei vielä olisi saatukaan. Sanomia välitetään kanavissa. Yhteen kanavaan voi lähettää tietoa moni palvelu ja samaa kanavaa voi myös lukea moni palvelu. Kanavia on kahdenlaisia: pisteestä-pisteeseen (point-to-point) sekä julkaisu-merkintä. Pisteestä-pisteeseen ohjaa sanoman tasan yhdelle palvelulle, joka kuuntelee kanavaa. julkaisu-merkintä tapa ohjaa sanoman kaikille kanavaa kuunteleville palveluille. Sanoma järjestelmiä on useita; osa käyttää standardoituja AMQP tai STOMP-protokollia ja osa omia mutta dokumentoituja protokollia. Kaikki sanoma järjestelmät eivät ole kaupallisia vaan useita avoimen-lähdekoodin järjestelmiä löytyy myös, joista Humphreyn [18] mukaan suosituimmat ovat RabbitMQ ja Apache Kafka. Kuten synkronisella tavalla, myös asynkronisella kommunikointitavalla on omat vahvuudet ja heikkoudet. Vahvuuksiin lukeutuu esimerkiksi se, että asiakasohjelma irrotetaan palveluista; asiakasohjelma vain lähettää viestin kanavaan, sanomat jäävät jonoon, joten, toisin kuin synkronisessa tapauksessa, asiakasohjelman ja palvelun ei tarvitse olla ajossa samaan aikaan. Haittoina taasen

on operatiivinen kompleksisuus; sanomanvälittäjä (message broker) on uusi ylläpidettävä järjestelmän palanen, jonka saatavuuden tulisi olla todella korkea. Lisäksi pyyntö/vastaus-kommunikointitavan toteutus on huomattavasti monimutkaisempaa kuin synkronisessa vastineessa [36].

Kuvassa 4.1 on esitetty, kuinka taksi-palvelun viestintä voisi toimia. Kuvassa asiakas laittaa puhelimen sovelluksella ilmoituksen Matkanhallinta-palvelulle. Matkanhallinta-palvelu tarkastaa asiakkaan tilin tilan Matkustajanhallinta-palvelulta käyttäen pyyntö/vastaus-menetelmää. Matkanhallinta-palvelu luo matkan ja ilmoittaa sen Lähettäjä-palvelulle käyttäen julkaisu/merkintä-menetelmää. Lähettäjä-palvelu paikantaa lähimmän vapaana olevan kuljettajan. Sopivan löytyessä, kuljettaja saa tiedon matkustajasta ja matkustaja tiedon löytyneestä kuljettajasta.



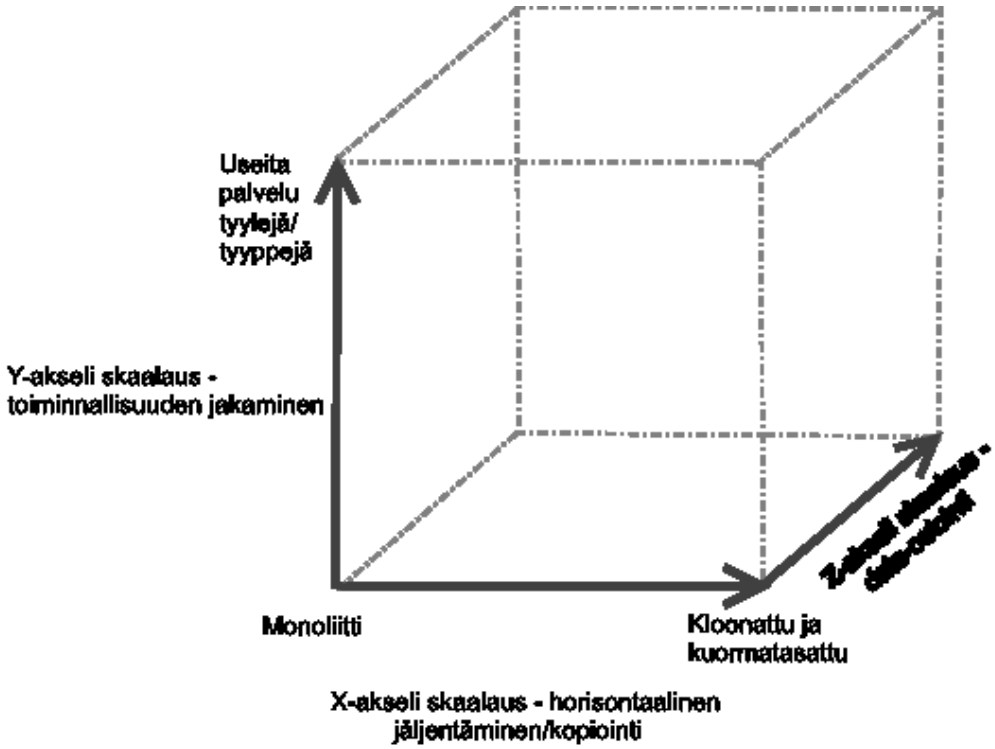
Kuva 4.1 Esimerkki IPC-menetelmistä taksi-palvelussa [36]

4.2 Sovelluksen pilkkominen mikropalveluiksi

Sovelluksen pilkkomisen mikropalveluiksi voi tehdä monella tapaa ja tyyli, tai tapa, yleensä määräytyy sen mukaan, millainen sovellus on kyseessä ja kuinka suuri kyseinen sovellus on. Pilkkomisen avuksi on kuitenkin löytynyt jo lähes vakiintunut aputyökalu; scale cube [14] [26] [34]. Lisäksi Newman on teoksessaan [30] maininnut jaetut kirjastot yhtenä mahdollisena aputyökaluna.

4.2.1 Scale cube

Scale cuben tarkoitus on helpottaa hahmottamaan, kuinka tulisi pilkkoa olemassa olevat datat, palvelut sekä transaktiot [14]. Scale cube on kuutio, jonka x, y ja z-akselit määrittävät sovelluksen skaalautuvuuden. Scale cube on esitetty kuvassa 4.2. Kuvassa vasen alanurkka, kohta jossa kaikki kolme akselia risteävät, on aloituspiste. Tässä kohtaa sovellus on monoliitti, jota voidaan skaalata erinäisten toimenpiteiden avulla joko yhdellä tai usealla akselilla kerrallaan.



Kuva 4.2 Scale cube [14]

Scale cuben x- akseli skaalausta sivuttiin aikaisemmin kappaleessa 3, kun puhuttiin monoliitin skaalauksesta. Sovelluksesta julkaistaan usea kopio ja nämä kopiot laitetaan kuormantasaajan taakse. Tämä on helppo tapa skaalata, mutta tämä tyyli ottaa kantaa ainoastaan pyyntöjen määrään kasvuun, mutta ei datan kasvuun [14] [26] [34].

Z-akselin skaalaus on hieman saman tyylinen kuin x-akselin; identtinen koodi pyörii jokaisella palvelimella. Erona x-akseli skaalaukseen on se, että nyt jokainen palvelin on vastuussa vain osasta dataa. Järjestelmässä on jokin osa tai komponentti, joka on täysin vastuussa pyyntöjen välittämisessä oikeille palvelimille. Reitityskriteerinä voi toimia esimerkiksi pyynnön attribuutti, kuten olion pääavain tai asiakkaan tyyppi. Z-akselin skaalauksella helpotetaan vikojen eristämistä sekä transaktioiden skaalautuvuutta [14] [26] [34].

Mikäli halutaan skaalata sovellusta y-akselin suuntaisesti, tulee sovelluksen toiminnallisuutta jakaa pienempiin kokonaisuuksiin. Jako voidaan tehdä toimenpiteille, datalle tai

molemmille. Y-akselin skaalauksen avulla saadaan aikaan se, että kaikki pyynnöt skaalautuvat ja ne käsitellään tarpeen mukaan eri tavalla. Tässä skaalautumisen tavassa logiikka ja data ovat eriytettynä, jonka johdosta kehittäjien ja kehitystiimien ketteryys sekä tuottavuus kasvavat. Ja koska toiminnallisuudet ovat omia pienempiä kokonaisuuksiaan, vika yhdessä komponentissa ei välttämättä heijastu enää koko sovellukseen. Tämä skaalaus tyyli kuitenkin usein on kalliimpi verrattuna kahteen muuhun skaalaukseen [14] [26] [34].

4.2.2 Jaetut kirjastot

Jaetut kirjastot, tai pelkästään kirjastot, ovat perinteinen tapa jakaa koodipohjaa pienempiin, uudelleen käytettäviin osiin, koodikielestä riippumatta. Kirjastot voivat olla itse tehtyjä tai kolmannen osapuolen tarjoamia ja niiden avulla voidaan helposti jakaa toiminnallisuuksia eri tiimeille ja palveluille. Jaetuilla kirjastoilla on kuitenkin omat ongelmansa. Kirjastojen lähes poikkeuksetta tulee olla kaikki samalla koodikielellä toteutettuja tai ainakin pyöriä samassa ympäristössä (Java Virtual Machine (JVM)-tyyppiset). Kirjastojen itsenäinen julkaisu ja skaalaus ei onnistu, ellei kirjasto ole dynaaminen, eikä muutoksia pysty tekemään ilman, että joutuisi julkaisemaan koko sovelluksen uudelleen [30].

4.2.3 Palveluiden koko

Kuten aikaisemmin mainittiin, palveluiden koon käsite ei ole yksikäsitteinen ja sille löytyy useita tulkintoja. Puhuttaessa palveluiden koosta, yleensä päädytään keskustelemaan palveluiden rakeisuudesta (granularity) tai siitä kuinka karkea- tai hienojakoinen palvelu on. Näiden kahden jakoisuuden erona on se, että karkeajakoiset palvelut koostuvat suuremmista komponenteista ja usein käärivät yhden tai useamman hienojakoisen palvelun yhteen karkeampaa toiminnallisuutta varten [16]. Täten hienojakoiset palvelut koostuvat pienemmistä komponenteista ja usein ovat alemman tason palveluita. Paremman mielikuvan saamiseksi voidaan ajatella hiekkapaperia; karkeajakoisessa hiekkapaperissa hiekka on suurempaa jolloin hiekanjyviä ei välttämättä ole hirveästi verrattuna hienojakoiseen hiekkapaperiin, jossa on enemmän mutta pienempiä hiekanjyviä.

Rakeisuuden, tai jakoisuuden, määrittäminen on myös vaikeaa, sillä sitä ei voida mitata absoluuttisten lukujen tai arvojen perusteella johtuen kyseisen rakeisuuden, tai jakoisuuden, mahdollisesti määrittävien käsitteiden subjektiivisuudesta. Esimerkiksi valitaan rakeisuuden määrittämiseksi palvelun tukemat toiminnot. Tällä ei saada yhtä ainoata arvoa, vaan lista toiminnoista, joista jokainen voi olla kokonainen liiketoimintaprosessi, toimijan (actor) toteuttama toimi tai tilan vaihto [16].

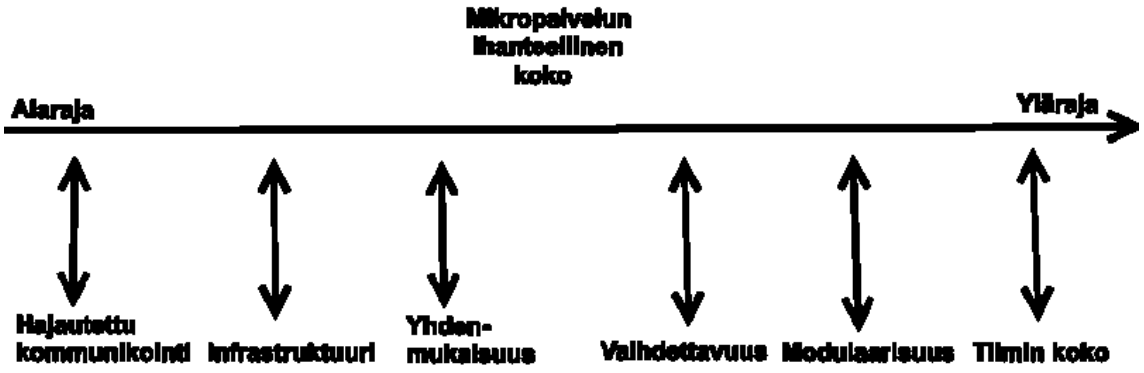
Wolff:n [44] mukaan palvelun kokoon vaikuttavat myös tiimin koko, modulaarisuus, vaihdettavuus, infrastruktuuri, hajautettu palveluiden välinen kommunikointi sekä yhdenmukaisuus. Näistä tiimi, modulaarisuus sekä vaihdettavuus määrittävät palvelun koon

yläpään ja infrastruktuuri, hajautettu palveluiden välinen kommunikointi sekä yhdenmukaisuus palvelun koon alapään.

Tiimin koko vaikuttaa palvelun koon yläpäähän; mikropalveluiden tulisi olla ainoastaan sen kokoisia, että sen parissa työskentely ei vaadi useata tiimiä tai yhtä todella suurta. Eri tiimien tulisi lisäksi pystyä julkaisemaan palveluita toisistaan huolimatta. Myös modulaarisuus rajoittaa kokoa. Mikropalvelun tulisi olla sellainen kooltaan, että yksi kehittäjä kykenee ymmärtämään se kokonaan sekä kehittämään sitä eteenpäin. Vaihdettavuus pienentää mikropalvelun yläpään kokoa. Mikäli palvelu halutaan vaihtaa, tulee kyseisen henkilön ymmärtää vaihdettava palvelu [44].

Tiedon yhdenmukaisuus sekä vuorovaikutukset vaikuttavat mikropalveluiden alapään kokoon. Palvelun ei tulisi olla niin pieni, että yhdenmukaisuus ja toimet taataan (ensure) useammassa palvelussa. Infrastruktuurin vaikutus on suhteellisen suuri; mikäli oikeanlaisen infrastruktuurin tarjoaminen mikropalvelulle on työlästä, tulee mikropalveluiden määrää pienentää tai pitää pienempänä, jolloin väkisinkin yksittäisen mikropalvelun koko kasvaa. Lisäksi palveluiden välisen kommunikoinnin taakka kasvaa sitä mukaan kuin mikropalveluiden määrä kasvaa, joten mikropalveluiden kokoa ei tule asettaa liian pieneksi [44].

Kuvassa 4.3. on esitetty Wolff:n [44] näkemys siitä, kuinka yllä mainitut toimet määräävät mikropalvelun koon.



Kuva 4.3 Mikropalvelun koon määrittely [44]

Mikä siis on mikropalvelun ihanteellinen koko? Newman [30] ajattelee ihanteellisen koon olevan ”riittävän pieni, eikä yhtään pienempi” ja Tilkov:n [41] mielestä mikropalveluista ei tulisi koittaa tehdä mahdollisimman pieniä. Myös Wolff [44] miettii ettei ihanteellista palvelun kokoa ei ole, vaan koko määräytyy käytetystä teknologiasta sekä mikropalvelun käyttötarkoituksesta.

4.3 Mikropalveluarkkitehtuurin hyödyt

Mikropalveluarkkitehtuurilla on useita hyötyjä. Mikropalvelut ovat verrattain pieniä, jonka ansiosta niitä on helpompi ymmärtää ja kehittää. Pienen koodipohjan ansiosta ei myöskään IDE mene ns. tukkoon. Tukkoon menemisellä tässä tarkoitetaan tilannetta, jossa koodipohja tai jopa yksittäiset moduulit olisivat niin isoja, että IDE:in tehdessä syntaksin tarkastusta, sekä muita koodin laatua parantavia toimenpiteitä, joutuu IDE käyttämään turhan paljon koneen resursseja, jolloin IDE:n käyttö muuttuu niin hitaaksi, että sitä on lähes mahdoton käyttää. Pienuutensa johdosta palvelut myös usein käynnistyvät nopeammin kuin monoliitti. Näiden seikkojen ansiosta kehittäjät ovat tuotteliaampia [4] [15] [34].

Mikropalvelut mahdollistavat jatkuvan julkaisemisen. Palveluita voidaan julkaista ja päivittää toisista palveluista riippumatta. Päivitys on mahdollista toisista riippumattomana, mikäli kyseinen päivitys ei koske millään tavalla muita palveluita [15] [34].

Kaikkia palveluita voidaan skaalata itsenäisesti (x-akseli ja z-akseli). Palveluita voidaan myös julkaista sellaiselle laitteistolle, joka vastaa parhaiten kyseisen tai kyseisten palveluiden resurssien tarvetta. Tämä eroaa huomattavasti monoliitista, jossa kaikki julkaistaan samaan paikkaan riippumatta resurssitarpeista [15] [34].

Mikropalvelu arkkitehtuurin myötä myös vikasietoisuus paranee. Esimerkiksi yhdellä palvelulla on muistivuotoa; tämä ei pistä koko sovellusta alas, vain osan sitä, sillä muistivuoto vaikuttaa vain kyseiseen palveluun. Monoliitissa koko sovellus menee alas [4] [15] [34].

Teknologia, jolla palvelua kehitettään, on jokaisen tiimin päätettävissä. Mikropalveluarkkitehtuurin ansiosta ei olla kiinni vuosia sitten valitussa teknologiassa. Myös uusien teknologioiden kokeilu on helppoa [4] [15] [34].

4.4 Mikropalveluarkkitehtuurin ongelmat

Mikropalveluarkkitehtuurilla on myös ongelmat ongelmansa, eikä arkkitehtuurityyliä kannata ottaa käyttöön liian kevein perustein. Hajautettujen järjestelmien kehittäminen on monimutkaista. Kehittäjien tulee toteuttaa IPC-menetelmä, jonka avulla palvelut kommunikoivat. Käyttötapausten, jotka kattavat useita palveluita ilman hajautettua vuorovaikutusta, toteuttaminen on hankalaa. IDE:t on usein suunniteltu monoliittien kehitykseen. Lisäksi automatisoitujen testien teko on vaikeaa varsinkin, jos kyseessä on useampi palvelu [4] [34].

Mikropalveluarkkitehtuuri tuo myös merkittävää toiminnallista monimutkaisuutta. Mukana on useita instansseja eri tyyppisistä palveluista, joita on pystyttävä hallitsemaan tuotannossa. Yleensä tämä vaatii korkeantason PaaS-tyylistä teknologiaa sekä automatisointia [34].

Julkaistaessa uutta ominaisuutta, joka kattaa useita palveluita, vaaditaan tiimeiltä tiimien välistä kanssakäymistä. Tällöin tulee suunnitella tarpeeksi tarkasti, kuinka ominaisuus ajetaan ja minne kaikkialle se vaikuttaa, esimerkiksi olemassa olevien palveluiden riippuvuuksiin [4] [34].

Yksi haaste on valita missä vaiheessa järjestelmän elinkaarta mikropalveluarkkitehtuuria käytetään. Kehityksen alkuvaiheessa ongelmia ei juuri esiinny, eikä yleensä sellaisia, joihin lääke olisi mikropalvelut, joiden käyttö alusta alkaen voisi aluksi hidastaa kehitystä [34].

Mikropalvelut poikkeuksetta tarkoittaa hajautettua järjestelmää. Tyyli ja toimet, jolla hajautus tehdään, määrittävät kuitenkin sen onko kyseessä oikeasti mikropalvelut vai vain hajautettu monoliitti. Hughsonin [17] mukaan järjestelmä on hajautettu monoliitti, mikäli

- kaikki palvelut täytyy julkaista samaan aikaan, jotta järjestelmä toimisi,
- palveluiden kommunikointi on synkronista sekä vikasiedotonta tai
- järjestelmällä on vain yksi ainoa tietovarasto.

Lisäksi Christensen [6] on pitänyt esitelmän, jossa hän esittää jaettujen kirjastojen vaikutusta mikropalveluihin sekä hajautettuun monoliittiin. Hänen esityksessään jaetut kirjastot ovat niitä kirjastoja, joita tarvitaan palveluiden suorittamiseen, esimerkiksi Spring sekä Guava. Lopulta järjestelmän toiminta voi riippua sadoista kirjastoista, joita kaikkia tarvitaan koko järjestelmän suorittamiseen. Christensenin mielestä, jos palvelu ei pysty suoriutumaan järjestelmässä ilman että kaikki nämä kirjastot ovat saatavilla, on kyseessä hajautettu monoliitti [6]. Tällöin ollaan vain hajautettu monoliitti ympäri verkkoa ja kärsitty hajautetun järjestelmän tuomat ongelmat, mutta samalla menetetty paljon mikropalveluiden tuomista hyödyistä [6].

5. NYKYINEN RAJAPINTA SEKÄ YMPÄRISTÖ

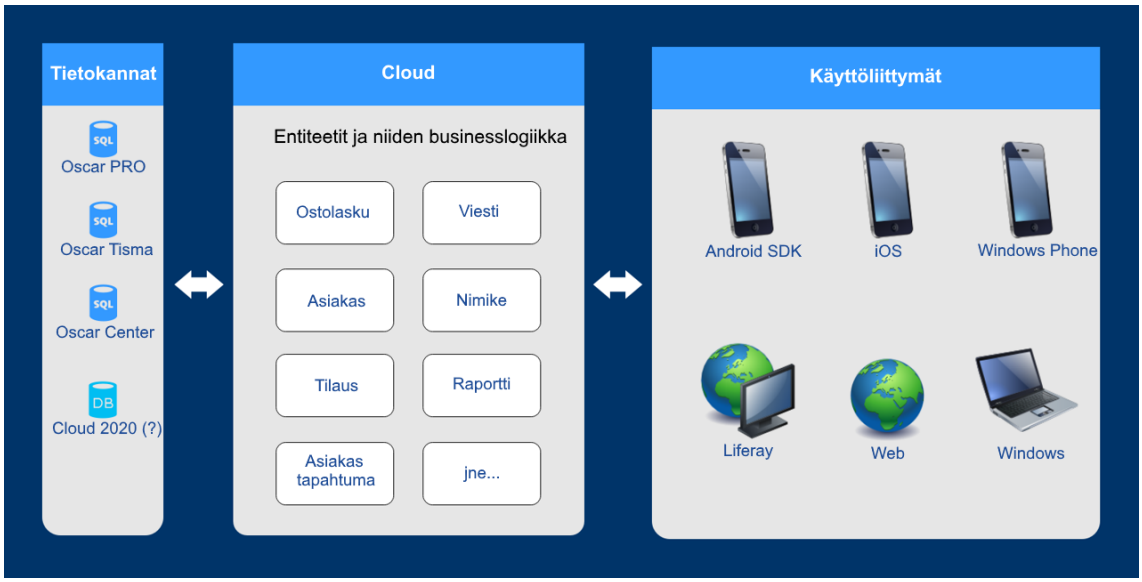
Kuten johdannossa mainittiin, nykyistä, Cloud-nimellä kulkevaa rajapintaa on kehitetty Oscar Softwarella vuodesta 2013 lähtien. Vuosien varrella rajapintaa on kehittänyt useampi tiimi ja koodipohja on laajentunut. Vaikka projektin struktuuri on pysynyt modulaarisena, on aikaansaannos muovautunut monoliitiksi. Monoliitti on kirjoitushetkellä edelleen hallittavissa mutta tietoa siitä, kuinka kauan, on vaikea määrittää. Koska kyseessä on, tällä hetkellä, rajapinta ERP-järjestelmille, tarjoaa se oivan mahdollisuuden miettiä voisiko nykyiselle monoliitille tehdä jotain, kuten pilkkoa se pienempii autonomisiin mikropalveluihin.

Tässä kappaleessa käydään läpi rajapinnan nykyinen tilanne sekä toiminta ympäristö. Lisäksi mietitään rajapinnan pilkkomisen syitä sekä tavoitteita.

5.1 Cloud-rajapinta

Cloud-rajapinta on Javalla toteutettu ohjelmisto, oikeammin businesslogiikkakerros, joka tarjoaa REST-rajapinnan. Cloud-rajapintaa voidaan tarjota asiakkaille pilvipalveluna. Rajapinta koostuu muutamista itse tehdyistä kirjastoista sekä kolmannen osapuolen kirjastoista. Lisäksi löytyy kehittäjiä varten luotu asiakaskirjasto (client), jolla helpotetaan kutsujen tekemistä rajapintaa vasten. Rajapinnan tarkoituksena on toimia yhtenä rajapintana kaikkien Oscar Softwaren tuotteiden välillä; ratkaisut on pyritty tekemään siten, että oli taustajärjestelmänä tai taustalla sitten Firebird tai Oracle tietokanta, sen ei tulisi vaikuttaa kuin maksimissaan tietokanta-konfiguraatioon.

Yleinen arkkitehtuuri tietokantojen, Cloudin sekä käyttöliittymien välillä on kuvattu kuvassa 5.1. Kuvassa vasemmalla näkyy järjestelmäriippumattomat tietokannat, jotka ovat yhteydessä Cloud-sovellukseen. Cloud toimii logiikkakerroksena käyttöliittymille.



Kuva 5.1 Cloud-sovelluksen korkean tason arkkitehtuuri

Cloud on toteutettu käyttäen Javaa ja tarkemmin sanottuna Java Enterprise Edition-versiota (JEE/JavaEE). Java kehitystä tehdään Java Development Kit:n (JDK) avulla. Projektissa käytettävä JDK versio on 7, joka kirjoitushetkellä on toisiksi uusin, JDK 8 ollessa uusin. Cloud-rajapinta on yksi WAR-paketti, jonka voidaan julkaista ajoon esimerkiksi Apache Tomcatiin.

Rajapinnan keskeisimmät teknologiat ovat:

- Java Persistence API (JPA) 2.1
- Hibernate
- Spring
- Java API for RESTful Web Services (JAX-RS) / Jersey

Businesslogiikkakerroksessa käytetään Java Persistence API:a, joka tarjoaa mahdollisuuden luoda luokkia, jotka voidaan tallentaa suoraan tietokantaan. JPA:n avulla toteutetaan olio-relaatiokuvaus (ORM) data access object (DAO)-kerroksella sekä pohja REST-metodeille. Olio-relaatiokuvaus on kuvaus siitä, kuinka luokka kuvautuu tietokannassa olevaan tauluun. ORM toteuttajana toimii Hibernate.

Spring sovelluskehyksellä toteutetaan taustajärjestelmä-instanssikohtaisten sovelluskontekstien luonti ja hallinta sekä suoritetaan transaktiot Service-kerroksella. Taustajärjestelmät ovat kuvan 5.1 mukaisia erilaisten järjestelmien tietokantoja. Esimerkiksi Oscar Softwaren kohdalla kyseessä voisi olla Oscar Pro:n Oracle-tietokanta tai Oscar Tisman COBOL datatiedostoista AcuXDBC-tiedonhallintajärjestelmän avulla luotu relaatiotietokantaa mallintava tiedosto. Sovelluskonteksti on Spring sovelluskehyksen määrittämä ApplicationContext. Lyhyesti sanottuna ApplicationContext on Spring sovelluksien keskusrajapinta, jonka tarkoitus on tarjota määrittelytietoja. Ajon aikana ApplicationContext

on ainoastaan luettavissa, mutta mikäli sovellus sitä tukee, voidaan konteksti ladata uudelleen [33].

JAX-RS on Java API sekä spesifikaatio, joka tarjoaa tuen Web-palveluiden luomiseen REST-arkkitehtuurin mukaisesti. JAX-RS käyttää Java SE 5 mukana tulleita annotaatioita Java-pohjaisten verkkopalvelujen luomisen ja käyttöönoton yksinkertaistamiseksi. Sen avulla pystytään myös toteuttamaan asiakassovelluksia REST-palveluun [42]. Jersey on avoimen lähdekoodin sovelluskehys Java-pohjaisten verkkopalvelujen kehittämiseen, joka on rakennettu JAX-RS:n päälle ja toimii JAX-RS-viitetoteutuksena. Lisäksi Jersey laajentaa JAX-RS:iä, helpottaen palveluiden toteutusta entisestään [19]. JAX-RS:in ja Jerseyn avulla julkaistaan palvelintoteutuksen metodit käyttäen kirjastojen Java REST-annotaatioita sekä siirretään datansiirto-olioita (DTO) asiakaskirjaston ja palvelintoteutuksen välillä. Ohjelmassa 1 on esitetty muutamia Java annotaatioita, joita käytetään REST metodeissa. Annotaatio *@GET* määrittää, että kyseessä olevaa getCurrencies-metodia kutsutaan silloin kun HTTP Get-kutsu tulee *@Path*-annotaation määrittämään URI:in. *@Produces*-annotaatio määrittää sen, mitä tyyppiä palautuva data on. Ohjelmassa metodi palauttaa datan JSON-muodossa. *@PathParam* sekä *@QueryParam*-annotaatiot määrittävät mitä ja minkä tyylisiä parametreja kutsu ottaa vastaansa.

```
@GET
@ApiOperation(value = "Lists currencies by criteria", notes = "Lists currencies by criteria")
@ApiResponses(value = {
    @ApiResponse(code = 200, message = "Successful retrieval of currencies"),
    @ApiResponse(code = 404, message = "Currencies not found"),
    @ApiResponse(code = 500, message = "Internal server error")
})
@Produces(MediaType.APPLICATION_JSON)
@Path("{oscarBackendName}/currencies")
public List<CurrencyDTO> getCurrencies(@PathParam("oscarBackendName") String oscarBackendName, @QueryParam("parameterPath") String parameterPath, @QueryParam("orderPath") String orderPath,
@QueryParam("pageNumber") Integer pageNumber, @QueryParam("pageSize") Integer pageSize) throws Exception {

    AnnotationConfigApplicationContext context = getContext(oscarBackendName);
    currencyService = context.getBean(CurrencyService.class);
```

Ohjelma 1. *Erinäisiä Java REST-annotaatioita*

Rajapintaa voidaan ajatella eri kerroksina. Näistä muutama mainittiin edellä olevassa teknologia osiossa. Järjestelmän kerroksia ovat:

1. Entiteetit
2. Data Access Object (DAO)
3. Service
4. REST-palvelintoteutus
5. Data Transfer Object (DTO)

6. REST-asiakaskirjasto

Entiteetit ovat luokkia, joiden tehtävänä on kuvata taustajärjestelmän tauluja ja olioita. Koska tausatajärjestelmiä voi olla useita, on entiteetti-luokat pyritty rakentamaan siten, että ne sisältävät vain attribuutteja, jotka pystytään ORM-kuvaamaan taustajärjestelmästä riippumatta. Entiteettien suunnittelussa on koitettu huomioida mahdollisuus sisäiseen periyttämiseen entiteettien välillä. Entiteetit on nimetty englanniksi ja nimeämisessä on pyritty käyttämään englanninkielistä termiä; esimerkiksi ostolasku on nimetty PurchaseInvoice.

Tällä hetkellä entiteettejä on 112 kappaletta. Kaikki 112 eivät kuitenkaan kuvaudu suoraan tietokannan tauluihin, vaan osa toimii pääentiteettinä, joista voidaan periyttää uusia, spesifisempiä entiteettejä. Esimerkkeinä tällaisista entiteeteistä ovat geneerinen entiteetti GenericEntity, josta on periytetty mm. tiedosto (File)-entiteetti, laskurivi (InvoiceRow)-entiteetti ja tilausrivi (OrderRow)-entiteetti, sekä tilaus-entiteetti Order, josta on periytetty myynti- ja ostotilaus-entiteetit (SalesOrder ja PurchaseOrder). Muita entiteettejä on esimerkiksi luokittelu (Classification), valuutta (Currency), työvuoro (Shift) ja toimittaja (Supplier). Taulukossa 5.1 on listattu kaikki nykyiset entiteetit aakkosjärjestyksessä.

Taulukko 5.1 Cloud-sovelluksen käyttämät entiteetit aakkosittain

Entiteetti						
Account	Activity-Contact-Person	Activity-Type	Batch	BatchE-vent	Board	BoardCo-lumn
Booking	Boo-kingCode	Boo-kingRea-son	Business-Partner	Business-Partner-Link	Classifi-cation	Classifi-cation-Descrip-tion
Classifi-cation-Number-ListValue	Classifi-cati-onString-ListValue	Configura-tionData	CostCen-ter	Country	Currency	Customer
Customer-Classifi-cation	Customer-Contact-Person	Customer-RelatedAc-tivity	Custo-merStatus	Custom-Target	Depart-ment	Distributi-onList
Distributi-onList-Member	Employee	Employee-TimeT-rackingDe-tails	EntityMap	File	Generi-cEntity	Group
Industry	Invoice	InvoiceCir-culation-Comment	InvoiceCir-cula-tionItem	Invoi-ceRow	Invoi-ceRowTar-get	Invoi-ceRowTar-getTrans-lation
Item	ItemClas-sification	ItemStock-Balance	Load-Group	LoginUser	LoginUs-erCreden-tials	Message

Numbering-Sequence	Order	OrderRow	PaymentTerms	Person	Price	ProfitCenter
Project	PurchaseInvoice	PurchaseInvoiceRow	PurchaseOrder	PurchaseOrderRow	PurchasePaymentTerms	PurchaseShipmentMethod
PurchaseShipmentTerms	QualityDeviation	QualityDeviationStatus	ReportGenerationData	SalaryDayState	SalesInvoice	SalesInvoiceRow
SalesOrder	SalesOrderRow	SalesOrderRowText	SalesOrderText	SalesPaymentTerms	SalesPerson	SalesShipmentMethod
SalesShipmentTerms	SalesTarget	SalesTargetLogEntry	SerialNumber	ServiceContract	Shift	ShiftLimit
ShipmentMethod	ShipmentTerms	StatisticsGroup	StepwisePurchasePrice	Stock	StockAction	StockShelfAddress
StockShelfItemBalance	Supplier	SupplierBankAccount	SupplierItemPrice	Tag	TargetLink	Task
TaskLoggingData	TaskRow	Team	TravelExpenseSpecification	TravelExpenseSpecificationRow	TypeOfCost	Unit
UserStatusDescription	VacationCalendar	VacationSpecification	ValueAddedTax	WorkHourEntry	WorkHourType	Wrapper

Data Access Objectit eli DAO:it ovat olioita, jotka tarjoavat abstraktin rajapinnan sovelluksen ja tietokannan välille. Cloudissa tietokantataulu- ja käsitekohtaiset DAO-kantaluokat toteuttavat yleiskäyttöisen DAO-rajapinnan, joka määrittelee yhteiset metodit (create, read, update, delete, räätälöidyt parametrikyselyt). Käsitekohtainen DAO-rajapinta periyttää yleiskäyttöisen DAO-rajapinnan ja lisää tarvittaessa käsitekohtaisia toimintoja. Esimerkkinä asiakas DAO, joka perinteisten CRUD-metodien lisäksi määrittää muutaman uuden haun, joissa käytetään hakukriteerinä asiakastunnusta. Jokaista käsitekohtaista DAO:ia varten on lisäksi tehty erillinen konkreettinen taustajärjestelmäkohtainen sovitinluokka. Nämä sovitinluokat periyttävät itseään vastaavan entiteetin, jolloin mahdollistetaan entiteetin attribuuttien kuvaaminen tietokantataulun kenttiin. Lisäksi sovitinluokat määrittävät käytettävän tietokantataulun nimen, taustajärjestelmäprofiilin sekä tarvittaessa hoitavat esimerkiksi taustajärjestelmän, eli tietokannan, NOT NULL-kenttiin kuvattujen attribuuttien alustukset.

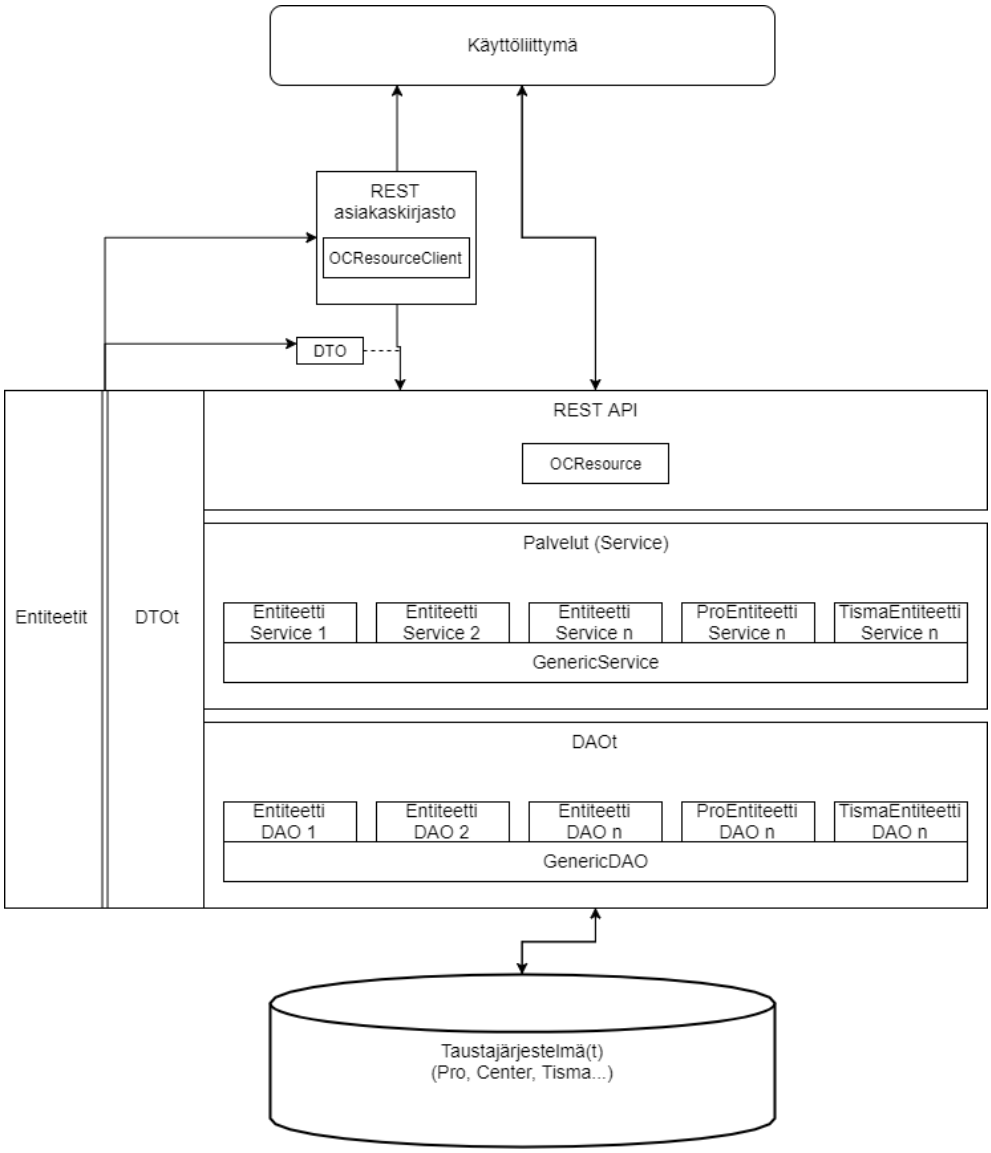
Service-kerroksella on yleiskäyttöinen Service-rajapinta ja abstrakti Service-kantaluokka sekä entiteettikohtaiset toteutukset aivan kuten DAO-kerroksella. Service-kerros on taustajärjestelmäkohtaisista yksityiskohdista riippumaton kevyt kerros DAO-kerroksen päällä ja sitä tarvitaan transaktioiden aikaansaamiseksi.

REST-palvelintoteutus toteuttaa kullekin entiteetille CRUD-metodeja vastaavat toiminnot, parametrikyselyt ja räätälöidyt metodit sekä julkaisee metodit annotaatioiden avulla. Parametrikyselyt ovat perus CRUD-metodien lisänä olevia kyselyitä, joiden avulla voidaan esimerkiksi hakea kaikki kyseiset entiteetit rajaten tulosjoukkoa annettujen parametrien mukaan. Räätälöidyt metodit ovat nimensä mukaisesti hyvin spesifejä ja toteutettu entiteetille, koska toimintoa ei saada tehtyä järkevästi normaalien metodien avulla. Täten Räätälöidyt metodit ovat harvinaisia. Kun palvelintoteutus julkaistaan tai uudelleen käynnistetään, ensimmäinen kysely laittaa ajoon kontekstienluontisäikeen, joka luo kaikki saatavilla olevat kontekstit peräkkäin. Mahdolliset muut alustuksen aikana tulevat kyselyt odottavat kontekstienluontisäikeen valmistumista. Säikeen suorituksen valmistuttua myöhemmät kyselyt käyttävät jo luotuja konteksteja. Palvelintoteutusta voidaan käyttää myös suoraan selaimella, jolloin vastaus selaimeen tulee JSON-muodossa. REST-palvelintoteutuksessa, esimerkiksi kyselyä tehtäessä, käyttäjä autentikoidaan ja annetaan pääsy käyttäjän Tomcat-käyttäjätunnukseen liitettyjen roolien mukaan. Autentikointi-metodina toimii HTTP:n basic authentication.

DTO:it on kevyitä, persistenssikontekstista irrotettuja olioita, jotka periyttävät vastaavat entiteetit mutta ei tuo lisää ominaisuuksia. DTO:ja voidaan siirtää REST-palvelintoteutuksen ja –asiakkaan välillä sekä käyttää suoraan Cloud-sovelluksissa.

REST-asiakaskirjasto keskustelee REST-palvelintoteutuksen kanssa HTTP-protokollalla. Asiakaskirjasto muuntaa parametrikyselyiden nimet ja arvot HTTP-parametreina kuljetettavaan merkkijonomuotoon. Lisäksi kirjasto käsittelee DTO:ja, mikä kannustaa niiden käyttöön myös Cloud-sovelluksessa.

Kuvassa 5.2 on kuvattu Cloud-rajapinnan arkkitehtuuri kerroksittain. Käyttöliittymä keskustelee REST-rajapinnan kanssa joko suorana tai asiakaskirjaston välityksellä. REST-rajapinta on yhteydessä palvelu-kerrokseen, joka vuorostaan on yhteydessä DTO-kerrokseen. DTO-kerros huolehtii taustajärjestelmien kanssa keskustelun. Kuvassa on esitetty myös se, että entiteettikohtaisesta Service- ja DAO-rajapinnasta voi olla myös pääjärjestelmäkohtaisia toteutusluokkia. Kaikki kerrokset käyttävät hyödykseen entiteettejä.



Kuva 5.2 Cloud-rajapinnan kerrosarkkitehtuuri

Cloud-sovelluksen toimiminen, varsinkin asiakaskohtaisesti, vaatii erinäisiä määrittelyitä. Näihin määrittelyihin lukeutuu muun muassa:

- käyttäjienhallinta
- käyttäjien oikeuksienhallinta
- välimuistinhallinta
- tietokantayhteydenhallinta

Käyttäjienhallinta rajapinnassa on toteutettu käyttäen Tomcatin käyttäjienhallinta määrittelytiedostoa tomcat-users.xml. Tässä tiedostossa listataan kaikki erilaiset roolit sekä tunnukset ja näiden salasanat. Roolit on nimetty asiakkaiden mukaan, kuten yleensä myös tunnukset ja salasanat ovat generoituja sattumanvaraisia merkkijonoja. Luotujen tunnuksien, käyttäjien, oikeuksienhallinta tapahtuu web.xml-tiedostossa. Web.xml-tiedoston alussa evätään kaikki oikeudet kaikilta tunnuksilta. Tämän jälkeen annetaan jokaiselle

tomcat-users.xml-tiedostossa määritellylle käyttäjälle erillinen oikeus URL:in, jossa on mukana ko. käyttäjätunnus. Samalla asetetaan pakotus autentikoinnille.

Tietokantayhteydet määritellään erillisissä *asiakas_tietokanta*.properties-tiedostoissa. Tiedostossa määritellään taustajärjestelmän tyyppi, taustajärjestelmän nimi, tietokantayhteyden tiedot JDBC-määreiden avulla; määritetään käytettävä tietokanta ohjain (driver), tietokanta URL sekä tietokannan tunnukset. Lisäksi tiedostossa määritetään yhteyden käyttämän välimuistin määrittelytiedosto. Tietokannan välimuistinhallintaan käytettävät määrittelytiedostot on nimetty Tomcat-käyttäjien mukaan, *käyttäjä*.ehcache.xml. Tiedosto sisältää muun muassa välimuistin nimen, väliaikaistiedostojen tallennuspolun sekä lähdekoodissa merkittyjen ja käyttöön haluttujen välimuistien määritykset. Nämä määritykset sisältävät esimerkiksi käyttöönotettujen välimuistien nimet, ko. välimuistien maksimi tulosmäärät, joita pidetään yllä sekä kauanko kestää, että välimuisti vanhenee. Välimuistin toteutukseen käytetään Ehcachea. Ehcache on avoimen lähdekoodin hajautettu välimuistitoteutus, joka on tarkoitettu yleiskäyttöön sekä Java EE-sovelluksiin.

Cloud- rajapinta on projektin näkökulmasta modulaarinen mutta silti monoliitti. Entiteetit on eriytetty aivan omaksi projektikseen. DAO:it sekä Servicet ovat omissa paketeissaan REST-palvelintoteutusprojektin alla. Kun projekti käännetään, syntyy siitä yksi iso WAR-tiedosto, joka sisältää sekä REST- että entiteettiprojektin. Taulukossa 5.2 on esitetty pakettitasolla REST-palvelintoteutusprojekti sekä entiteettiprojekti.

Taulukko 5.2 Projektit ovat modulaarisia mutta päätyvät lopulta suureksi monoliitiksi

REST-palvelintoteutusprojekti	Entiteettiprojekti
fi.oscar.cloud.clients	fi.oscar.cloud.entity.
fi.oscar.cloud.dao	fi.oscar.cloud.entity.dto
fi.oscar.cloud.dao.center	fi.oscar.cloud.entity.enumtypes
fi.oscar.cloud.dao.mapping	fi.oscar.cloud.entity.mapping
fi.oscar.cloud.dao.pro	fi.oscar.cloud.entity.util
fi.oscar.cloud.dao.pro.converter	fi.oscar.cloud.report
fi.oscar.cloud.dao.tisma	fi.oscar.cloud.report.dto
fi.oscar.cloud.dao.ventus	
fi.oscar.cloud.dao.ventus.converter	
fi.oscar.cloud.rest	
fi.oscar.cloud.rest.errorhandling	
fi.oscar.cloud.service	
fi.oscar.cloud.service.pro	
fi.oscar.cloud.service.tisma	
fi.oscar.cloud.service.vetus	
15 kpl	7 kpl

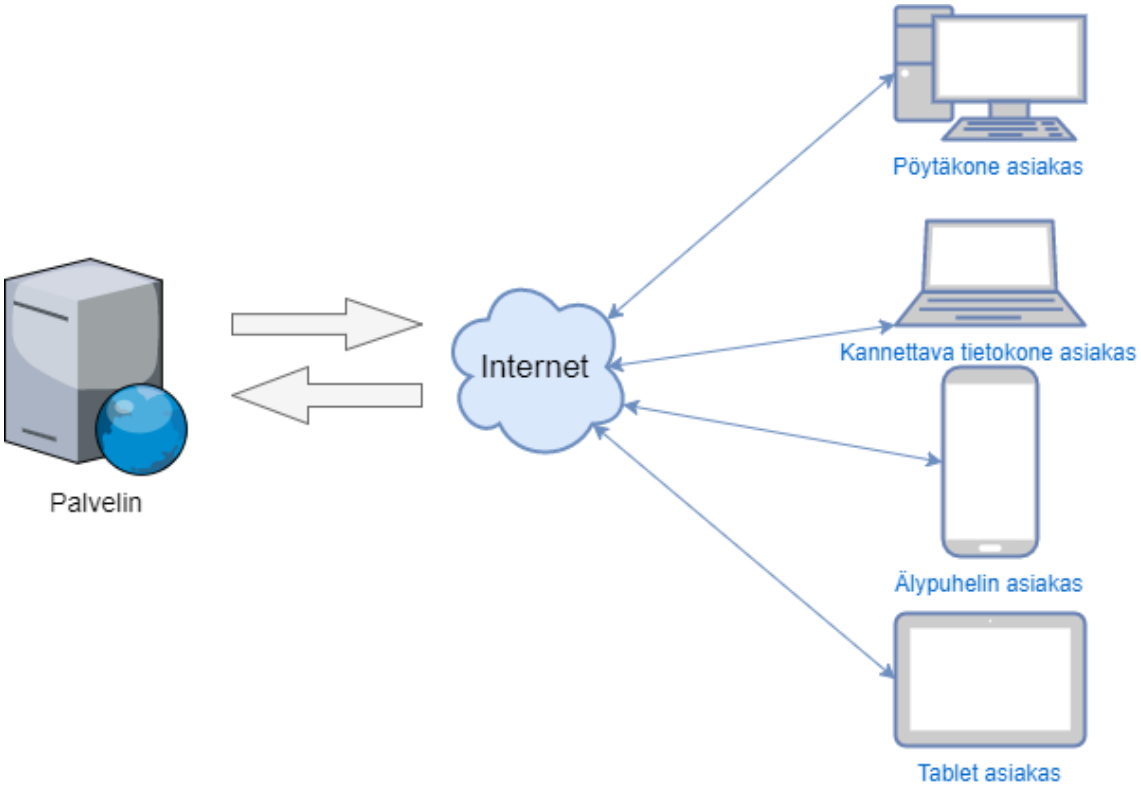
REST-palvelintoteutuksen keskeisin osa eli CRUD-metodit ovat kaikki kuitenkin samassa paketissa (fi.oscar.cloud.rest) ja tiedostossa. Koodiriveinä tämä yksi toteutettu luokka on jo yli 20000 riviä pitkä.

5.2 cERP-käyttöliittymä

Tällä hetkellä Cloud-rajapinnan lisäksi tehdään myös erillistä web-käyttöliittymää, joka hyödyntää rajapintaa. Tämä käyttöliittymä kulkee nimellä cERP. cERP:n keskeisimmät teknologiat ovat Liferay Portal, Bootsrap 3, jQuery, Vue.js ja Knockout.js sekä Gson.

cERP:n alustana toimii Liferay Portal versio 6.2. Liferay on Javalla toteutettu web-alusta, joka sisältää tarvittavat ominaisuudet verkkosivustojen ja portaalien kehittämiseen.

Web-alusta käyttää asiakas-palvelinarkkitehtuuria. Asiakas-palvelinarkkitehtuuri esittää, kuinka palvelin tarjoaa resursseja ja palveluita asiakkaille. Palvelimet tarjoavat resursseja asiakkaan laitteisiin, kuten pöytätietokoneisiin, kannettaviin tietokoneisiin, tabletteihin ja älypuhelimiin. Yleensä yksittäinen palvelin voi tarjota resursseja useille asiakkaille kerralla. Asiakas-palvelinarkkitehtuurissa asiakas ei jaa resurssejansa laisinkaan vaan kutsuu palvelimen palveluita [28]. Kuvassa 5.3 on esitetty normaali asiakas-palvelinarkkitehtuuri. Erinäiset asiakaslaitteet ottavat yhteyden palvelimeen internetin välityksellä.



Kuva 5.3 Asiakas-palvelinarkkitehtuuri

Liferay sisältää sisäänrakennetun verkkosisällönhallintajärjestelmän, jonka avulla käyttäjät voivat luoda verkkosivustoja ja portaaleja kokoamalla yhteen teemoja, sivuja, portletteja sekä yhteisiä navigointeja. Liferaysta on saatavilla avoimen lähdekoodin Community Edition (CE) versio sekä maksullinen Digital Experience Platform (DXP). cERP on kehitetty Liferay CE-version päälle. Liferay tarvitsee toimiakseen tietokannan. Liferay tarjoaa oman HSQL kannan, joka ei kuitenkaan virallisten Liferay dokumentaatioiden [24] mukaan ole tuotantokäyttöön tarkoitettu. Liferay tukee tunnetuimpia tietokantoja kuten DB2, MySQL, Oracle, PostgreSQL, SQL Server sekä Sybase [24].

Näistä tietokannoista cERP:ssä on käytössä MySQL. MySQL on valikoitunut käyttöön osaltaan sen takia, että kyseisestä tietokannasta löytyy kokemusta yrityksen sisältä ja osaltaan siksi, että Liferayn mukana tulee ajuri MySQL kantaa varten.

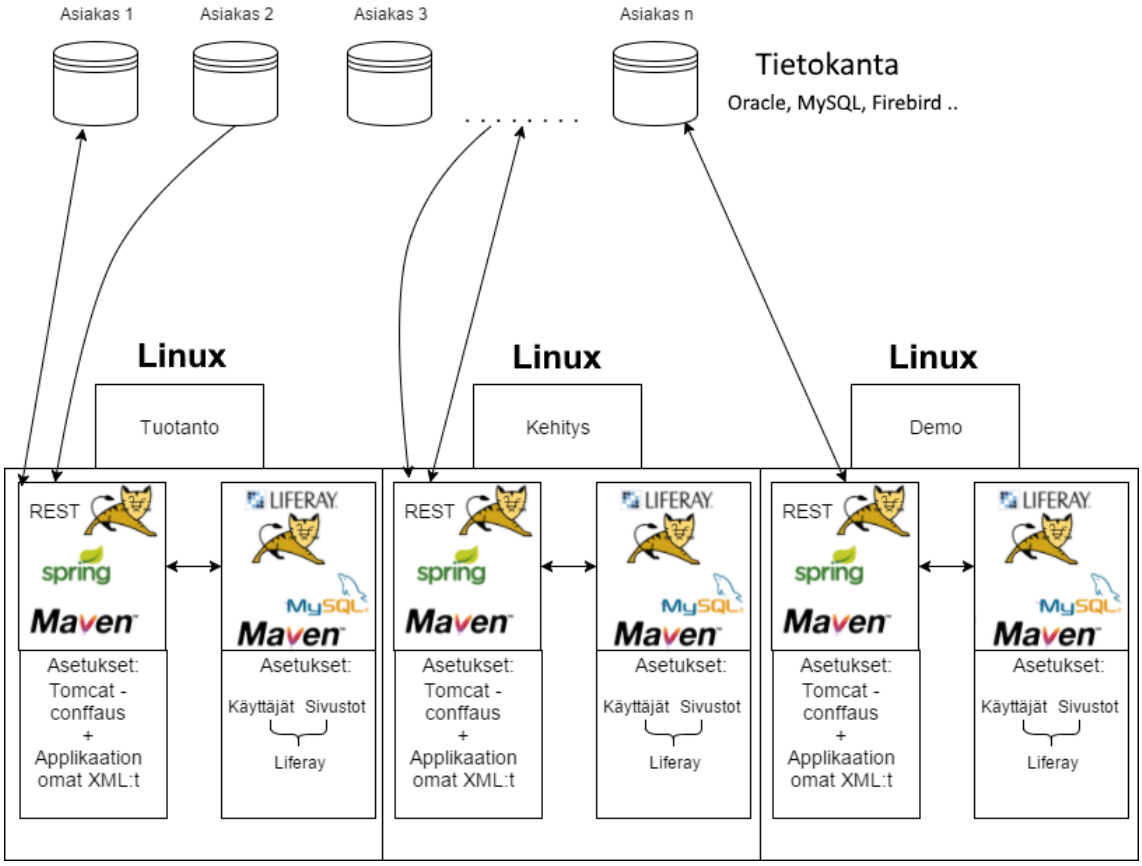
Bootstrap on, omien kotisivujensa mukaan, suosituin HTML, CSS ja JS sovelluskehys responsiivisten, mobiili-ensin-projektien kehittämiseen. Bootstrap sisältää valmiita määrityksiä CSS-elementeille sekä lisäksi myös runsaasti JavaScript-laajennoksia ulkoasuun ja responsiivisuuteen liittyen. Bootstrapin JavaScript-toiminnot tarvitsevat toimiakeen jQuery-kirjaston. cERP käyttää Bootstrap-sovelluskehystä ulkoasun pohjakirjastona.

jQuery on kevyt JavaScript-kirjasto, jonka tarkoituksena on helpottaa JavaScriptin käyttöä web-sivuilla. jQueryn iskulause kuvaa tätä yritystä: ”kirjoita vähemmän, tee enemmän”.

Vue.js on JavaScript sovelluskehys käyttöliittymien rakentamiseksi ja knockout.js on JavaScript-kirjasto, jonka avulla voidaan luoda monipuolisia ja reagoivia käyttöliittymiä. cERP käyttää näitä kahta käyttöliittymä- ja template-kirjastoina. Knockout.js pyritään korvaamaan tulevaisuudessa kokonaan Vue.js-kirjastolla.

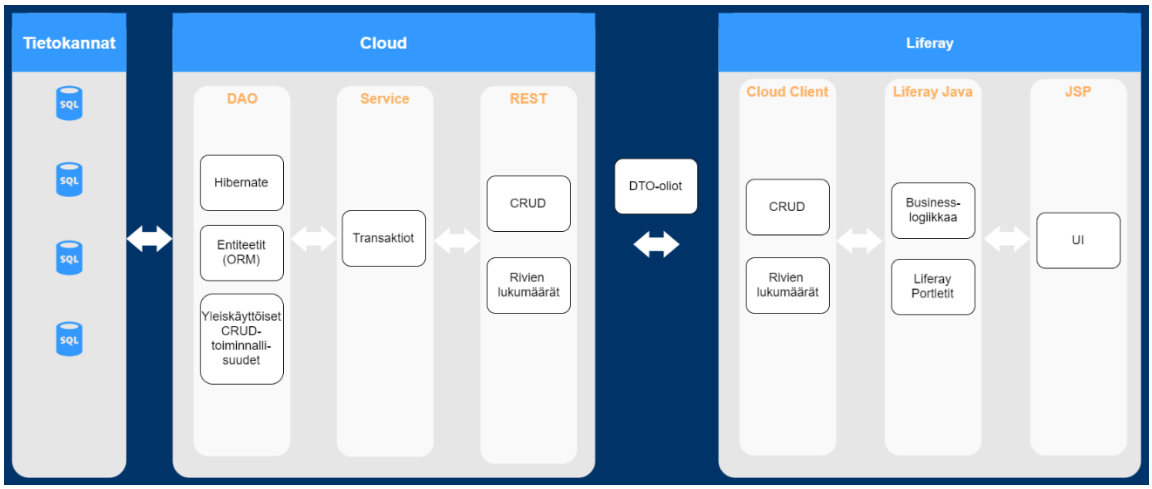
Gson on Java-kirjasto, jonka avulla voidaan muuntaa Java-objekti JSON-muotoon. Kirjaston avulla voidaan myös muuntaa JSON-merkkijono vastaavaksi Java-objektiksi. cERP-projektissa tätä kirjastoa käytetetään REST JSON-datan välitykseen.

cERP ja Cloud on ajossa erillisillä virtuaalikoneilla. Palvelimia on kolmea eri tyylistä: tuotanto, kehitys ja demo. Yhdessä palvelimessa voi olla ajossa monia instansseja molemmista eli esimerkiksi kehityspalvelimella voi olla ajossa kolme eri kehitysversiota. Cloud ja cERP eivät jaa samaa Tomcatia, vaikka keskustelevatkin keskenään. Jokaisella eri versiolla Cloudista ja cERPistä on siis oma Tomcatinsa. Palvelinympäristö kuvausta on esitetty kuvassa 5.4 Kuvassa ylhäällä on asiakkaiden, niin oikeiden kuin kehitys, tietokantoja. Näihin kantoihin tehdään toimenpiteitä REST:n kautta (Cloud), mutta alkuperäiset kutsut tulevat cERP-Liferay ympäristöstä.



Kuva 5.4 cERP ja Cloud palvelinympäristöjen kuvaus

Kuvassa 5.5 on kuvattu tarkemmin cERP-Liferay käyttöliittymän ja Cloud-sovelluksen yhtenäisyys. cERP tekee kutstuja Cloudiin. Data välitetään DTO-olioiden avulla. Cloud prosessoi pyynnön ja hoitaa tarvittavat toimenpiteet kantoihin. Vastaus välitetään takaisin käyttäen DTO-olioita.



Kuva 5.5 cERP:n ja Cloud-sovelluksen yhteistoiminta

5.3 Cloud-rajapinnan pilkkomisen syyt ja tavoitteet

Nykyinen Cloud-rajapintasovellus on vuosien saatossa kasvanut yhä suuremmaksi ja suuremmaksi. Hanketta aloitettaessa arkkitehtuuriratkaisuksi lopulta muodostui monoliittinen arkkitehtuuri. Näin ollen nykyinen sovellus on massiivinen monoliittinen kokonaisuus. Kuten kappaleessa 3.3 on käyty läpi, monoliittisissa sovelluksissa on useita ongelmia kuten päivittämisen työläys ja hankaluus sekä sovelluksen skaalautuvuus.

Oscar Software tarjoaa Cloud-rajapintaa sekä cERP-käyttöliittymää pilvipalveluna. Pilvipalvelu aspektin myötä on monoliittisten sovellusten ongelmat nousseet entistä pahemmin esille. Päivitystä tai korjaustoimenpidettä ei välttämättä saada asiakkaalle ilman järjestelmän uudelleen käynnistämistä ja mikä pahinta, näitä toimenpiteitä ei voida tehdä vaikuttamatta muihin samassa versiossa oleviin asiakkaisiin. Tästä johtuen päivitykset sekä kiireettömät korjaustoimenpiteet ajoitetaan toimistoaikojen ulkopuolelle, jolloin liikenne olisi mahdollisimman pientä. Tämä usein pitkittää kehittäjien päivää entisestään ja täten pitkällä aikavälillä vaikuttaa negatiivisesti tuottavuuteen kehittäjien väsyessä sekä stressaantuessa.

Päivityksen tai korjaustoimenpiteen ollessa valmis tulee järjestelmä käynnistää jälleen. Käynnistyksen yhteydessä ladataan kaikkien samassa Tomcat-palvelimessa olevien asiakkaiden tietokantayhteydet, sekä näiden yhteyksien välimuistiasetukset ja alustetaan. Riippuen asiakkaiden määrästä alustustoimenpide voi pahimmillaan kestää useita minuutteja.

Edellä mainittujen hankaluuksien johdosta, Oscar Softwarella on mietitty tapaa, jonka avulla nämä ongelmat saataisiin poistettua tai ainakin niiden haittavaikutusta lieventää. Yhtenä vaihtoehtona esille nousi mikropalvelut ja tätä myötä monoliittisen arkkitehtuurin pilkkominen.

Pilkkomisen pääasialliset tavoitteet ovat saada järjestelmä jaettua pieniin hallittaviin kokonaisuuksiin, koodipohjan versioinnin parantaminen, ongelmattomampi päivitys- sekä korjausprosessi ja projektin teknologian nykyaikaistaminen.

6. RAJAPINNAN PILKKOMINEN MIKROPALVELUIKSI

Mahdollisesti helpoin tapa ottaa mikropalvelut mukaan olemassa olevaan projektiin on siten, että lopettaa uusien ominaisuuksien teon suoraan olemassa olevaan koodipohjaan ja aloittaa ominaisuuksien tuottamisen itsenäisinä mikropalveluina. Nämä uudet mikropalvelut sitten laitetaan keskustelemaan olemassa olevan koodipohjan kanssa. Tällöin ei enää kasvateta monoliitin massiivista kokoa entisestään ja lisäksi, refaktoroinnin ansiosta, tulevaisuudessa monoliitin pilkkominen on entistäkin helpompaa sekä nopeampaa. Vaikka kyseessä on suhteellisen helppo tapa ottaa mikropalvelut käyttöön, vaatii tämä lähestyminen myös työtä; mikropalveluiden ja olemassa olevan koodipohjan tulee kuitenkin pystyä keskustelemaan keskenään.

Olemassa olevan koodipohjan pilkkominen on myös aivan varteenotettava mahdollisuus. Pilkkomisessa on kuitenkin huomioitava sellainen seikka, että se tulee viemään enemmän aikaa kuin se, että ruvetaan tekemään uusia palveluita mikropalveluina. Vanhan sovelluksen pilkkominen ei täten välttämättä ole paras tai oikea lähestymistapa resurssi näkökulmasta katsottuna.

Pilkkomisessa on myös syytä käyttää hieman aikaa palveluiden koon miettimiseen. Ongelmaa voidaan lähestyä usealla tapaa ja tässä on esitetty kaksi hieman erilaista mahdollisuutta.

Ensimmäinen tapa on miettiä olemassa olevien ominaisuuksien käyttötapauksia sekä ongelmakohtia, joita ne pyrkivät ratkaisemaan. Esimerkiksi myyntilaskut, ostolaskut sekä matkalaskut ovat kaikki laskuja, joten ne voisivat kuulua laskut-palveluun. Samoin esimerkiksi työajanseuranta-palvelun voisi koostaa sisään-ulos leimaus, jonoleimaus yms.

Toinen tapa lähestyä palveluiden kokoa on miettiä olemassa olevat ominaisuudet myytävinä tuotteina. Ne ominaisuudet, jotka mielletään lisäominaisuuksiksi tai palveluiksi, voisivat olla omia mikropalveluitaan. Tällöin esimerkiksi myyntilaskut ja ostolaskut olisivat laskut-palvelussa mutta matkalaskut olisi aivan oma mikropalvelunsa.

Riippumatta siitä miten asiaa ajattelee, tärkeintä aluksi kuitenkin on se, että olemassa olevaa monoliittia pilkotaan ja rajataan tavalla tai toisella pienemmäksi. Aluksi palveluiden koko voi olla mieluummin liian iso kuin liian pieni. Tällöin ollaan jo otettu ensimmäiset askeleet kohti uutta arkkitehtuuria sekä teknologiaa ja mikäli tulevaisuudessa joku palveluista näyttää liian isolta, voidaan se helpommin pilkkoa pienemmäksi kuin kasvattaa kokoa.

Kuten aikaisemmin on mainittu, on nykyinen rajapinta toteutettu käyttäen Javaa sekä Spring-sovelluskehystä ja kommunikointi menetelmänä toimii REST. Kappaleessa 5.1 käytiin läpi, että osa rajapinnan määrittelytiedostoista on Tomcat riippuvaisia. Tähän Tomcat riippuvuuteen ollaan saamassa muutos käynnissä olevan kehityshankkeen kautta. Määrittelytiedostot ollaan muuttamassa sellaisiksi, että niitä voidaan muokata ja lisätä ajonaikaisesti, ilman järjestelmän uudelleen käynnistystä. Tämä pyritään saamaan aikaan ottamalla projektiin mukaan Spring Cloud-kirjastot. Spring Cloud-kirjastot auttavat monessa osa-alueessa mutta etenkin pilvipohjaisissa ratkaisuissa, mukaan luettuna mikropalvelut [23].

Richardson on tuonut artikkelissaan [37] esille kolme eri tyyppistä strategiaa sille, kuinka monoliittia voidaan lähteä pilkkomaan ja muokkaaman. Ensimmäinen tyyli on sama, joka on mainittu kappaleen 6 alussa: lopetetaan monoliitin kasvattaminen ja tehdään uudet ominaisuudet mikropalveluina. Tämä tyyli vaatii monoliitin ja uusien palveluiden lisäksi pyyntöjen reitityksen sekä liimakoodin, jolla mahdollistetaan monoliitin ja uusien palveluiden välinen vuorovaikutus. Toinen Richardsonin esiin tuoma tyyli on perinteinen käyttöliittymän ja businesslogiikan erottaminen. Kolmantena tyylinä Richardson esittää olemassa olevien palveluiden eriyttämisen ja muokkaamisen mikropalveluiksi. Tällä tyylillä monoliitin koko pienenee vähitellen [37].

Yllä mainituista tyyleistä vain kahta voidaan käyttää Cloud-rajapintaan, sillä kuten kappaleessa 5 on käyty läpi, on käyttöliittymä eriytetty Cloud-projektista jo valmiiksi. Koska tarkoituksena on saada Cloud-rajapinnan koodipohjaa hallittavammaksi, esiintyy Richardsonin kolmas tyyli edukseen. Tämä tyyli tarvitsee toimiakseen jotakin, jonka avulla eriytettävä palvelu saadaan toimimaan omana palvelunaan, tavan välittää kutsut oikeille palveluille sekä eriytettävä palvelu. Eriytettävän palvelun valinta voi olla aluksi hankalaa ja tämän johdosta Richardson neuvoo valitsemaan alkuun helppoja palveluita, jolloin saadaan koodipohjaa pienemmäksi ja opitaan tekemään eriyttämisiä tehokkaasti [37].

Esimerkki pilkkominen toteutetaan käyttäen Richardsonin mainitsemaa [37] eriyttämistyyliä. Tekninen osa toteutetaan käyttäen Spring Cloud-kirjastoja ja varsinkin sen sisältämiä Spring Boot, Netflix Eureka, Netflx Ribbon ja Netflix Zuul komponentteja:

- **Spring Boot:** Spring Boot pyrkii helpottamaan Spring sovellusten tekoa, etenkin itsenäisiä (standalone) sovelluksia. Spring Bootissa on Tomcat sisäänrakennettuna. Tämän kirjaston avulla saadaan pienellä refaktoroinnilla tehtyä monoliitin sisällä olevasta palvelusta oma itsenäinen palvelu.
- **Netflix Eureka:** Eureka on REST-pohjainen palvelu, jota käytetään ensisijaisesti pilvessä palveluiden paikallistamiseen kuormituksen tasapainottamiseksi ja keskitason palvelimien vikasiirroksi. Eurekan avulla saadaan pidettyä yllä tietoa siitä, missä palvelut sijaitsevat.

- **Netflix Ribbon**: Ribbon on IPC-kirjasto, jossa on sisäänrakennettu ohjelmisto kuormantasaaja. Ribbonin tarjoamat ominaisuudet ovat kuormantasaus, vikasietoisuus, moniprotokolla tuki asynkronisessa ja reaktiivisessa mallissa sekä välimuisti ja niputus (caching and batching).
- **Netflix Zuul**: Zuul on reunapalvelu, joka tarjoaa dynaamisen reitityksen, seurannan, joustavuuden ja turvallisuuden. Kirjaston avulla pystytään välittämään kutsut oikeille palveluille.

Spring Cloud-kirjastojen valintaan vaikuttaa sekä sen sisältämät kirjastot sekä se, että kyseinen teknologia on tulossa mukaan projektiin myöhemmässä vaiheessa, kun määrittelytiedostojen kehityshanke on valmis.

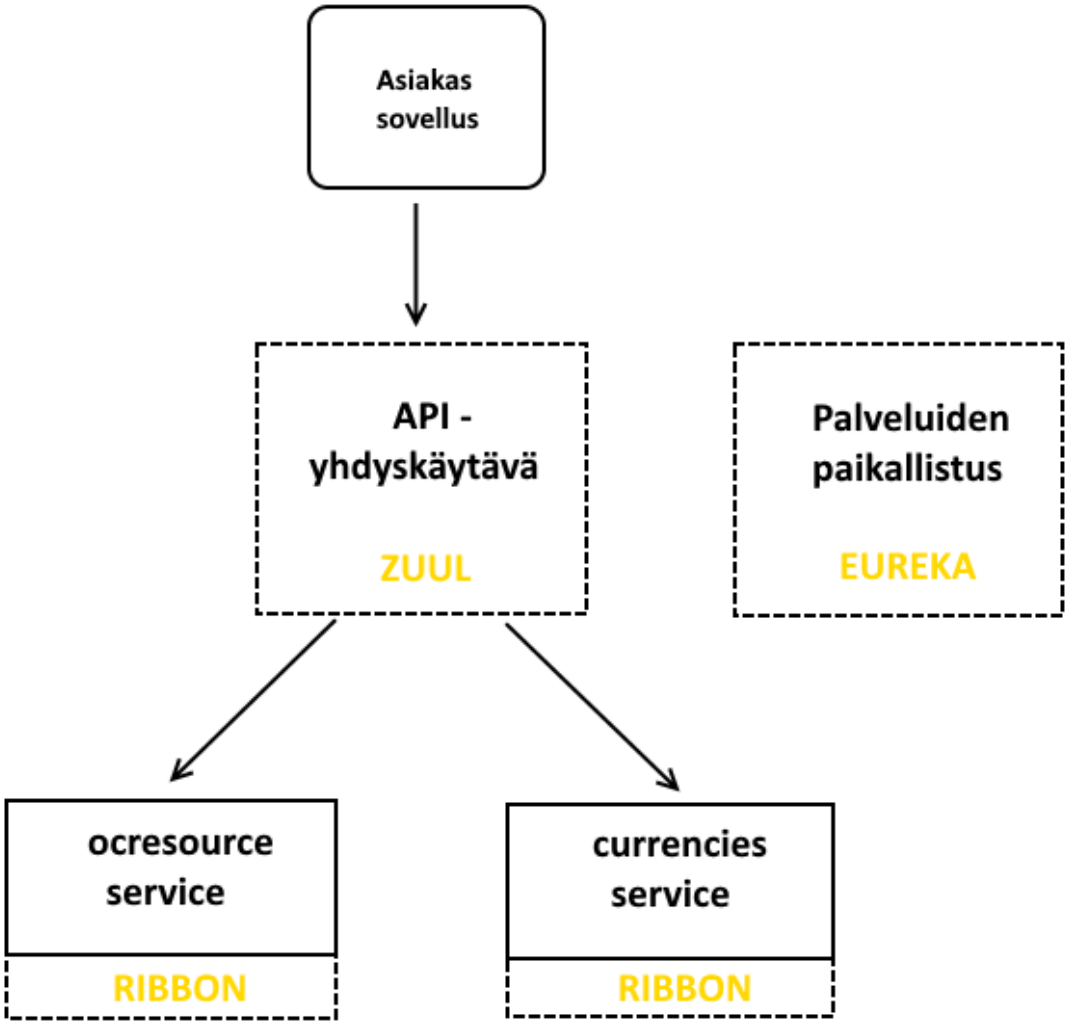
Kommunikaatiomenetelmäksi valitaan REST sen ollessa jo tärkeä osa projektia. Vaihtoehtoisesti menetelmäksi olisi voitu valita viestinvälitys käyttäen RabbitMQ, joka on osa määrittelytiedostojen muutos projektia. Tämä jäi valitsematta kuitenkin siitä syystä, että toteutus oli nopeampi tehdä käyttäen REST:ia.

Esimerkkitapauksessa Spring Boot määrittelytiedosto-muodoksi valitaan YAML. YAML-määrittelytiedostomuoto valikoitui, koska tätä muotoa käytettiin lähes poikkeuksetta kaikissa manuaalin esimerkeissä sekä muualla internetissä olevissa esimerkeissä ja kysymyksissä. Edellä mainituista seikoista johtuen, voidaan päätellä kyseisen määrittelytiedostomuodon olevan eräänlainen *de facto*, syrjäyttäen oman kokemukseni mukaan usein Java-sovelluksissa käytetyn XML-tiedostomuodon. Toinen syy, minkä takia YAML valikoitui, on sen yksinkertaisuuden ansiosta.

Esimerkkitapauksessa olemassa olevasta rajapinnasta erotetaan yksi yksinkertainen palvelu: valuutta-palvelu (currencies-service). Valuuta-palvelulla voidaan palauttaa taustajärjestelmään tallennetut valuuttakurssit, lisätä valuuttakursseja sekä päivittää niitä. Palvelun irrottaminen muusta järjestelmästä toteutetaan ottamalla kopio olemassa olevasta rajapintasovelluksesta ja poistamalla sieltä tarpeettomat luokat, kuten geneeriset luokat sekä valuutta entiteetin DAO ja service-kerroksien toteutukset. Lisäksi alkuperäisestä (ocresource-service) poistetaan kaikki valuutta-palveluun liittyvät tiedot.

Näiden kahden palvelun lisäksi toteutetaan discovery-service (Eureka) johon eri palvelut voivat rekisteröityä sekä gateway-service (Zuul), joka hoitaa asiakas sovellukselta tulevien pyyntöjen jakamisen.

Kuvassa 6.1 on esitetty esimerkki pilkkomisen arkkitehtuurista. Asiakasohjelmalta tulee pyyntö, joka menee Zuul reunapalvelulle. Zuul ohjaa pyynnön oikealle palvelulle riippuen pyynnöstä. Tämän toteuttamiseksi ocresource-service ja currencies-service ovat rekisteröityneet YAML-määrittelytiedostojen mukaisesti Eurekaan, josta Zuul saa tiedon palveluiden sijainnista.



Kuva 6.1 Toteutetun pilkkomisen arkkitehtuuri

Valuutta-palvelun eriyttämisen jälkeen sekä currencies-service että ocresource-service muunnetaan Spring Boot-sovellukseksi. Tämä tapahtuu yksinkertaisesti lisäämällä yksi uusi Application-luokka sekä annotaatio, jolla määritetään luokan olevan Spring Boot-sovellus. Samalla annotaatioiden avulla käynnistetään sovelluksen rekisteröinti mahdollisuus, jolloin Eureka pystyy saamaan tiedon sovelluksen tilasta. Tämä kaikki on esitetty ohjelmassa 2.

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application extends SpringBootServletInitializer {

    @Override
    protected     SpringApplicationBuilder     configure(SpringApplica-
tionBuilder application){
        return application.sources(Application.class);
    }

    public static void main (String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Ohjelma 2. *Spring Boot-luokka, joka on rekisteröitävissä Eurekaan*

Tämän jälkeen luodaan gateway-service sekä discovery-service. Molemmat palvelut ovat Spring Boot-sovelluksia aivan kuten rajapinnat. Molemmat sisältävät vain Application-luokan sekä application.yml-määrittelytiedoston. Application luokissa annotaatioilla **@EnableEurekaServer** määritetään Eureka palvelin ja **@EnableZuulProxy** määritetään Zuul. Muutoin Application-luokat ovat identtisiä kuvassa 6.1. nähtävän luokan kanssa.

Määrittelytiedostoissa on asetettuna palvelun nimi, sovelluksen portti sekä muuta tärkeää tietoa kuten sen, mistä löytyy Eureka palvelu. Määrittelytiedot vaihtuvat palvelun mukaan. Esimerkki käytetystä määrittelytiedostosta on esitetty ohjelmassa 3, jossa on ocresource-service-palvelun määrittelytiedot.

```
server:
  port: ${PORT:8181}
spring:
  application:
    name: ocresource-service
  jmx:
    default-domain: ocresource-service
  logging:
    level:
      org.springframework: WARN
      org.hibernate: WARN

eureka:
  client:
    serviceUrl:
      defaultZone: ${DISCOVERY_URL:http://localhost:8761}/eureka/
  instance:
    leaseRenewalIntervalInSeconds: 30
    leaseExpirationDurationInSeconds: 20

ribbon:
  eureka:
    enabled: true
```

Ohjelma 3. *ocresource-service-palvelun määrittelytiedosto*

Lopulta lisätään palveluille tarvittavat kirjastoriippuvuudet. Kuten aikaisemmin on todettu, riippuvuuksia hallinnoidaan käyttäen Apache Mavenia. Ohjelmassa 4 on listattu tärkeimmät lisätyt kirjastoriippuvuudet palvelukohtaisesti.

```
<!-- Currencies & ocresource -->
<dependency>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<!-- Zuul -->
<dependency>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-starter-eureka</artifactId>
</dependency>
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter</artifactId>
</dependency>
<dependency>
	<groupId>org.springframework</groupId>
	<artifactId>spring-web</artifactId>
</dependency>
<dependency>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-starter-zuul</artifactId>
</dependency>
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<!-- Eureka -->
<dependency>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>

<!-- Kaikille plugin -->
<plugin>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-maven-plugin</artifactId>
	<configuration>
		<addResources>true</addResources>
	</configuration>
</plugin>
```

Ohjelma 4. *Lisätyt kirjastoriippuvuudet*

Palveluiden ollessa valmiita, pystytään ne kääntämään JAR ja WAR-paketeiksi. JAR-paketit saadaan käyntiin komentoriviltä esimerkiksi käskyllä *java-jar discovery-service.jar* ja WAR-tiedostot voidaan julkaista normaalisti Tomcatissa. Näin on esimerkki ympäristö pystyssä ja mikäli määrittelytiedostot ovat oikein määritelty, rajapinnasta haettavat valuutat ohjataan currencies-service-palvelulle ja kaikki muut ocresource-service-palvelulle.

7. PILKKOMISEN ONNISTUMINEN SEKÄ AJATUKSIA JA TULEVAISUUDEN NÄKYMIÄ

Mikropalveluiden toteutus käyttäen Spring Cloud-kirjastoja on suhteellisen yksinkertaista ja suoraviivaista. Hyvin pienillä muutoksilla saadaan olemassa olevasta monoliitista tehtyä mikropalvelu tai mikropalveluita. Oletettavasti tulevaisuudessa tämä muuttuu vain entistä helpommaksi tekniikoiden ja teknologioiden uusiutuessa ja parantuessa.

Täysin ongelmitta pilkkominen ei kuitenkaan luultavasti onnistu; esimerkkitapauksessa, esimerkiksi JAR-paketiksi paketoitu palvelu ei kykene löytämään ajonaikana haettavia tiedostopolkuja. Tämä tosin on ymmärrettävää, sillä usein ohjelmointikielten tiedostoihin ja näiden polkuihin liittyvät toimenpiteet hoidetaan ajavan käyttöjärjestelmän näkökulmasta, jolloin JAR-paketin sisällä oleviin polkuihin ei kyseisillä toiminoilla päästä käsiksi. Pienellä funktion muutoksella tämäkin olisi kyllä onnistunut, mutta jokainen tällainen tuo lisää muokattavaa ja vie täten enemmän aikaa muuntaa.

Esimerkkitapauksessa on huomioitava esimerkin yksinkertaisuus. Esimerkissä ei toteutettu mikropalveluita, jotka olisivat kutsujen lisäksi toistensa tai kolmannen palvelun tiedoista riippuvaisia. Tämä tuo oletettavasti huomattavasti lisää monimutkaisuutta palveluiden kommunikaation suhteen. Lisäksi on hyvä miettiä sitä, tarvitaanko kommunikoinnissa mahdollisesti jonkinlaista viestijonoa.

Esimerkissä olisi voitu kommunikointi tapana käyttää REST sijaan esimerkiksi AMQP:tä, jolloin olisi tarvittu lisäksi sanomanvälitys palvelu, kuten RabbitMQ. Aikaisemminkin on mainittu, että RabbitMQ on mukana Cloud-rajapinnan määrittelytiedostojen muutos projektissa. Tästäkin syystä RabbitMQ olisi ollut varteen otettava teknologia valinta. Toistaiseksi REST on kuitenkin riittävä, mutta on syytä pitää mielessä muitakin vaihtoehtoja, mikäli REST osoittaa esimerkiksi suorituskykyongelmia järjestelmän kasvaessa. Mikäli tulevaisuudessa näyttää siltä, että REST on korvattava, on syytä pitää mielessä, että tämä muutos mahdollisesti vaikuttaa myös järjestelmän koodipohjassa aikoinaan tehtyihin ratkaisuihin, joten edessä voi olla isomman luokan refaktorointi.

On myös syytä huomioida sellainen seikka, että esimerkistä puuttui kokonaan jakelua helpottavat osat kuten esimerkiksi Docker-kontit. Docker-kontit ovat jo muutamassa Oscar Softwaren projektissa vähintään testauksessa mukana mutta jäivät tämän tutkimuksen rajojen ulkopuolelle. Tästä huolimatta on mietittävä, olisiko ensimmäinen askel ennen mikropalveluarkkitehtuuriin siirtymistä vain sellainen, että jokaisen asiakkaan Cloud-rajapinta ja mahdollisesti myös cERP-käyttöliittymä laitettaisiin Docker-konttiin. Tällä tavalla asiakkaat saadaan eriytettyä toisistaan, eikä yhdelle tehtävät toimenpiteet enää vaikuta muihin asiakkaisiin. Muihin ongelmakohtiin tämä ei suoranaisesti kylläkään vaikuta.

Jatko- tai sivututkimuksena voitaisiin tutkia sitä, kuinka Docker-kontit saataisiin oikea-oppisesti lisättyä projektiin ja kuinka tämä vaikuttaisi nykyiseen. Mikäli projekti valjastaa mikropalvelut uudeksi arkkitehtuuriksi, voisi olla myös silloin syytä katsoa, kuinka Docker-kontit auttaisivat.

Tekniikka kehittyy todella nopeasti nykyaikana. Onkin syytä pitää mielessä, että mikä on tällä hetkellä oikea ja hyvä tapa tehdä hajautettuja pilvipalveluita ei välttämättä huomenna enää ole. Tulee kuitenkin pitää järki päässä eikä ottaa kaikkea uutta teknologiaa käyttöön vain, koska se on uutta. Martin Fowler on antanut ensisijaiseksi ohjeeksi koskien mikropalveluarkkitehtuurin ottamista nykyiseen projektiin: ”**don't even consider microservices unless you have a system that's too complex to manage as a monolith.** The majority of software systems should be built as a single monolithic application.”. Tämä lainaus suomeksi voisi olla jotakuinkin seuraavanlainen: ”älä edes harkitse mikropalveluja, ellei sinulla ole järjestelmää, joka on liian monimutkainen hallittava monoliittina. Suurin osa ohjelmistoista tulisi rakentaa yhtenä monoliittisena sovelluksena. Kiinnitä huomiota hyvään modulaarisuuteen kyseisen monoliitin sisällä, mutta älä yritä jakaa sitä erillisiin palveluihin.” [13].

8. YHTEENVETO

Monoliittinen arkkitehtuuri on normaali tapa toteuttaa järjestelmiä. Monoliittinen arkkitehtuuri saattaa tuntua kankealta uudempiin arkkitehtuurityyleihin verrattuna mutta on edelleen pätevä tapa toteuttaa järjestelmiä. Monet aputyökalut kuten IDE:it on luotu monoliittista arkkitehtuuria silmällä pitäen. Riippuen toteutetusta järjestelmästä, monoliitin kankeus saattaa kuitenkin tulla todelliseksi ongelmaksi, etenkin pilvipalveluissa.

Mikropalveluarkkitehtuuri on oiva tapa toteuttaa vikasietoisia, skaalautuvia ja hajautettuja järjestelmiä, jotka ovat nykyaikana pilvipalveluiden kulmakivi. Mikropalveluarkkitehtuuri ei kuitenkaan ole hopealuoti, joten ennen sen käyttöönottamista on syytä punnita tarkkaan hyödyt ja haitat.

Taulukossa 8.1 on koottu yhteen sekä monoliittien että mikropalveluiden hyödyt sekä haitat.

Taulukko 8.1 Monoliittisen arkkitehtuurin sekä mikropalveluarkkitehtuurin hyödyt ja haitat

	Monoliittinen arkkitehtuuri	**Mikropalveluarkkitehtuuri**
Hyödyt	• Useiden kehitysohjelmien tuki • Julkaiseminen on helppoa • Helppo skaalata ajamalla useata kopiota sovelluksesta kuormantasaajan takana	• Pieniä, joten helppo ymmärtää ja kehittää-> parantunut tuottavuus • Palveluita voidaan julkaista ja päivittää toisista palveluista riippumatta • Palveluita voidaan skaalata itsenäisesti z-akselin ja x-akselin suunnassa • Parantunut vikasietoisuus • Käytettävä teknologia tiimin päätettävissä
Haitat	• Kärsinyt ketteryys • Heikentynyt tuottavuus • Heikko skaalautuvuus: ainoastaan x-askelin suuntaan eli monistus • Vanhentuneet teknologiat	• Hajautettujen järjestelmien kehittäminen on monimutkaista • Palveluiden välinen kommunikointi tulee toteuttaa • Käyttötapaukset helposti monimutkaisia ja niiden toteutus hankalaa • Automatisoitujen testien teko vaikeaa • Toiminnallisuus monimutkaistuu • Projektin tiimien välinen kanssakäyminen tulee tärkeämmäksi

Olemassa olevan monoliittisen rajapinnan pilkkominen mikropalveluiksi on täysin mahdollista ja Oscar Softwaren Cloud-rajapinnan kohdalla myös helppoa, mikäli käytetään

Spring Cloud-kirjastoja, eikä palveluiden välisiä kommunikointeja tarvitse miettiä. Nykyaikaisten kirjastojen avulla saadaan mikropalvelut, ainakin yksinkertaiset palvelut, mukaan projektiin todella vähäisellä koodipohjan refaktoroinnilla.

Vaikka mikropalvelujen mukaan ottaminen on helppoa, voi olla paikallaan miettiä, onko sille todellakin tarvetta vai pääsisikö, ainakin lähes, samaan lopputulokseen jotakin toista kautta. Arkkitehtuurin vaihto vaatii kuitenkin aina työtä sekä aikaa.

LÄHTEET

[1] R. Aamuvuori, M. Valtee, Toiminnanohjauksen onnela ja tuonela, Oscar Software Oy, Nov 2017, s. 6-9

[2] S. Abram, Microservices, Oct 2014, Saatavissa: http://www.javacodegeeks.com/2014/10/microservices.html

[3] R. Annett, What is a Monolith?, Nov 2014, Saatavissa: http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html

[4] V. Badola, Microservices is a way of breaking large software projects into loosely coupled modules, which communicate with each other through simple APIs, Nov 2015, Saatavissa: https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/

[5] B. Butzin, F. Golatowski and D. Timmermann, "Microservices approach for the internet of things," 2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA), Berlin, 2016, pp. 1-6. Saatavissa: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7733707&isnumber=7733490

[6] B. Christensen, Don't Build a Distributed Monolith, Jan 2016, Saatavissa: https://www.microservices.com/talks/dont-build-a-distributed-monolith/

[7] P. Clements, D. Garlan, R. Little, R. Nord, J. Stafford, "Documenting software architectures: views and beyond", 2003, IEEE Computer Society, pp. 740.

[8] D. Escobar, D. Cárdenas, R. Amarillo, E. Castro, K. Garcés, C. Parra, R. Casallas, Towards the understanding and evolution of monolithic applications as microservices, 2016 XLII Latin American Computing Conference (CLEI), Saatavissa: http://ieeexplore.ieee.org/document/7833410/

[9] C. Esposito, A. Castiglione and K. K. R. Choo, "Challenges in Delivering Software in the Cloud as Microservices," in IEEE Cloud Computing, vol. 3, no. 5, pp. 10-14, Sept.-Oct. 2016. Saatavissa: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7742281&isnumber=7742214

[10] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen and M. Villari, "Open Issues in Scheduling Microservices in the Cloud," in IEEE Cloud Computing, vol. 3, no. 5, pp. 81-88, Sept.-Oct. 2016. Saatavissa: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7742215&isnumber=7742214

[11] C. Fetzer, Building Critical Applications Using Microservices, IEEE Security & Privacy, Year: 2016, Volume: 14, Issue: 6, Pages: 86- 89, Saatavissa: http://ieeexplore.ieee.org/document/7782696/

[12] M. Fowler, J. Lewis, Microservices, Mar 2014, Saatavissa: http://martinfowler.com/articles/microservices.html

[13] M. Fowler, Microservice Premium, May 2015, Saatavissa: http://martinfowler.com/articles/microservices.html

[14] M. T. Fisher and M. L. Abbott. The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, Addison-Wesley Professional, 2009. s. 325-355

[15] A. Gupta, Microservices, Monoliths, and NoOps, Mar 2015, Saatavissa: http://blog.arungupta.me/microservices-monoliths-noops/

[16] R. Haesen, M. Snoeck, W. Lemahieu, S. Poelmans, (2008) On the Definition of Service Granularity and Its Architectural Impact. In: Z. Bellahsène, M. Léonard (eds) Advanced Information Systems Engineering. CAiSE 2008. Lecture Notes in Computer Science, vol 5074. Springer, Berlin, Heidelberg

[17] G. Hughson, What Makes a Monolith Monolithic?, Jul 2017, Saatavissa: https://genehughson.wordpress.com/2017/07/26/what-makes-a-monolith-monolithic/

[18] P. Humphrey, Understanding When to use RabbitMQ or Apache Kafka, Apr 2017, Saatavissa: https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka

[19] Jersey, RESTful Web Services in Java. 2017, Saatavissa: https://jersey.github.io/

[20] H. Khazaei; Cornel Barna; Nasim Beigi-Mohammadi; Marin Litoiu, Efficiency Analysis of Provisioning Microservices, 2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Year: 2016, Saatavissa: http://ieeexplore.ieee.org/document/7830692/

[21] P. Kookarinrat and Y. Temtanapat, "Design and implementation of a decentralized message bus for microservices," 2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE), Khon Kaen, Thailand, 2016, pp. 1-6. Saatavissa: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7748869&isnumber=7748833

[22] K. Koskimies, T. Mikkonen, Ohjelmistoarkkitehtuurit, Helsinki: Talentum, 2005. (Valikko-sarja), s.18-28

[23] R. Laddad, Introducing Spring Cloud, Jun 2014, Saatavissa: https://spring.io/blog/2014/06/03/introducing-spring-cloud

[24] Liferay Portal, Using Liferay's setup wizard, 2014, Saatavissa: https://dev.liferay.com/discover/deployment/-/knowledge_base/6-2/using-liferays-setup-wizard

[25] D. S. Linthicum, "Practical Use of Microservices in Moving Workloads to the Cloud," in IEEE Cloud Computing, vol. 3, no. 5, pp. 6-9, Sept.-Oct. 2016. Saatavissa: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7742277&isnumber=7742214

[26] L. MacVittie, The Art of Scale: Microservices, The Scale Cube and Load Balancing, Nov 2014 (updated Sep 2016), Saatavissa: https://devcentral.f5.com/articles/the-art-of-scale-microservices-the-scale-cube-and-load-balancing

[27] R. C. Martin, The Single Responsibility Principle, Luettu heinäkuussa 2017, Saatavissa: http://programmer.97things.oreilly.com/wiki/index.php/The_Single_Responsibility_Principle

[28] B. Mitchell, Introduction to Client Server Networks, Mar 2017, Saatavissa: https://www.lifewire.com/introduction-to-client-server-networks-817420

[29] D. Namiot, M. Sneps-Sneppe, On Micro-services Architecture, International Journal of Open Information Technologies, vol. 2, no. 9, 2014, Saatavissa: http://injoit.org/index.php/j1/article/view/139/104

[30] S. Newman, Building Microservices, O'Reilly Media, 2015, s. 2-11

[31] R. O'Connor, P. Elger, P. M. Clarke, Exploring the Impact of Situational Context — A Case Study of a Software Development Process for a Microservices Architecture, 2016 IEEE/ACM International Conference on Software and System Processes (ICSSP), Saatavissa: http://ieeexplore.ieee.org/document/7831529/

[32] Oscar Software, Toiminnanohjausjärjestelmä helpottaa yrityksen arkea, 2017, Saatavissa: https://www.oscar.fi/erp-jarjestelma-toiminnanohjaus

[33] Pivotal, Understanding Application Context, 2017, Saatavissa: https://spring.io/understanding/application-context

[34] C. Richardson, Microservices: Decomposing Applications for Deployability and Scalability, May 2014, Saatavissa: http://www.infoq.com/articles/microservices-intro

[35] C. Richardson, Pattern: Monolithic Architecture, 2014. Saatavissa: http://microservices.io/patterns/monolithic.html

[36] C. Richardson, Building Microservices: Inter-Process Communication in a Microservices Architecture, nginx, Jul 2015, Saatavissa: https://www.nginx.com/blog/building-microservices-inter-process-communication/

[37] C. Richardson, Refactoring a Monolith into Microservices, nginx, Mar 2016, Saatavissa: https://www.nginx.com/blog/refactoring-a-monolith-into-microservices/

[38] A. Sill, "The Design and Architecture of Microservices," in IEEE Cloud Computing, vol. 3, no. 5, pp. 76-80, Sept.-Oct. 2016. Saatavissa: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7742259&isnumber=7742214

[39] A. Singleton, "The Economics of Microservices," in IEEE Cloud Computing, vol. 3, no. 5, pp. 16-20, Sept.-Oct. 2016. Saatavissa: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7742218&isnumber=7742214

[40] G. Sousa, W. Rudametkin, L. Duchien, Automated Setup of Multi-cloud Environments for Microservices Applications, 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), Saatavissa: http://ieeexplore.ieee.org/document/7820288/

[41] S. Tilkov, How small should your microservice be?, Nov 2014, Saatavissa: https://www.innoq.com/blog/st/2014/11/how-small-should-your-microservice-be/

[42] Tutorialspoint, RESTful Web Services- Java (JAX-RS), 2017, Saatavissa: https://www.tutorialspoint.com/restful/restful_jax_rs.htm

[43] C. Williams, Is REST Best In A Microservices Architecture?, Dec 2015, Saatavissa: https://capgemini.github.io/architecture/is-rest-best-microservices/

[44] E. Wolff, What Are Microservies?, Nov 2016, Saatavissa: http://www.informit.com/articles/article.aspx?p=2738465