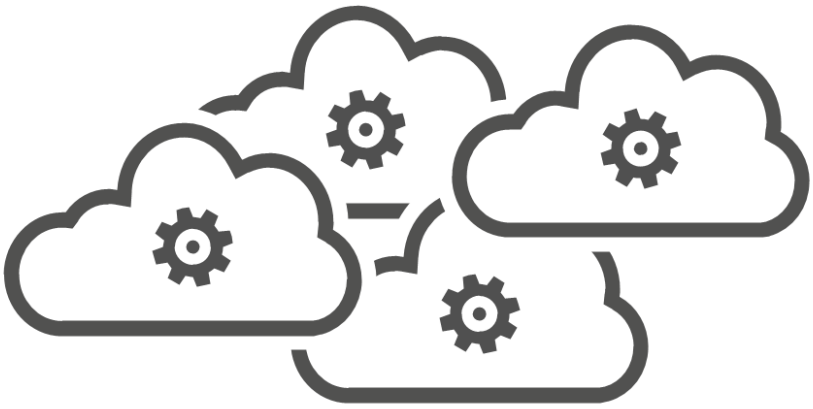
Jani Niemelä

Legacy-ohjelmistojen kontittaminen



Insinööri (AMK)

Tieto- ja viestintätekniikka

Syksy 2017

KAJAANIN
AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Tiivistelmä

Tekijä(t): Niemelä Jani

Työn nimi: Legacy-ohjelmistojen kontittaminen

Tutkintonimike: Insinööri (AMK), tietotekniikka

Asiasanat: Docker, Mikropalvelut, Legacy, LXC, Kontit

Eficode on Helsingissä toimiva ohjelmistoyritys, jolla on toimipisteitä Tampereella, Tukholmassa, Kööpenhaminassa ja Göteborgissa. Sen toimenkuvaan kuuluu asiakkaan bisnestarpeiden täyttämiseen ohjelmistokehitystä, konsultointia, ohjelmistotuotannon tehostamiseen liittyviä Devops-palveluita sekä digitaalisia asiakaskokemuksia palvelumuotoiluilla. Työ suoritettiin Eficoden tiloissa.

Tämän opinnäytetytön tavoitteena on kuvata, miten vanha legacy-järjestelmän monoliittinen arkkitehtuuri siirretään mikropalveluarkkitehtuuriin. Vaikka hajautetut järjestelmät eivät ole uusi asia tietotekniikassa, ovat mikropalvelut saaneet viiden vuoden aikana suuren suosion ja ovat tällä hetkellä pinnalla arkkitehtuurivalinnoissa.

Työssä tutkittiin miten kontitusteknologiat ovat kehittyneet 2000-luvulta nykyhetkeen. Tutkimuksessa käy ilmi vaiheet, miten monoliittinen arkkitehtuuri pilkotaan osiin ja millaisia muutoksia se tarvitsee toimiakseen itsenäisenä mikropalveluna. Nykypäivänä Docker tunnetaan käytetyimpänä alustana kontittamiseen ja sitä hyödynnetään työssä mikropalveluarkkitehtuurin luomiseen.

Työ aloitettiin jakamalla monoliittinen ohjelmisto omaan taustajärjestelmään, tietokantaan ja käyttöliittymäkerrokseen. Kehitystä varten luotiin automaattinen kehitysputki, joka testasi automaattisesti käyttöliittymän sekä tiedonhakukerroksen toimivuuden yksikkötesteillä. Ohjelmistoon tarvitsi tehdä uusina ominaisuuksina PDF-tiedoston luominen, toimintaloki ja palautusmekanismi, jolla pystyttiin palauttamaan tila entiteetin luomisesta lähtien. Näille ominaisuuksille luotiin omat palvelut, jotka yhdistettiin viestintäjonoon muiden palveluiden kanssa.

Työn tuloksena saatiin pienin toimiva tuote, joka pystyi erillisissä palveluissaan tallentamaan käyttäjätietoja ja tapahtumia, perumaan tapahtumia ja luomaan käyttäjätiedoista ja sen tapahtumista PDF-tiedostoja. Työssä saavutettiin sille annetut tavoitteet.

KAJAANIN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Abstract

Author(s): Niemelä Jani

Title of the Publication: Containerizing legacy applications

Degree Title: Bachelor of Engineering, Information Technology

Keywords: Docker, Containers, Microservices, LXC, Legacy

Eficode is software company located in Helsinki which has offices in Tampere, Stockholm, Copenhagen and Gothenburg. Its aim is to offer specialized software development, consulting and devops services to customers.

The aim of this Bachelor's thesis is to describe how monolithic legacy system can be modernized using microservice architecture. Even though distributed systems aren't a new thing in information technology, microservices have gained a lot of attention and have become the most used architecture in the past five years.

Containerization technologies from the 2000s and their development into their current state are examined how they started to progress towards what they are nowadays. The research shows the required steps to split monolithic architecture into own independent partitions called containers and the required changes. The microservice architecture in this work utilizes currently the most popular containerization platform called Docker.

As a result, the system is able to save user information and actions performed by users, rollback those events and create a PDF file using user information and its events in separate independent services which satisfies the given minimum viable product criteria.

Sisällys

Symboliluettelo

Alpine Linux = Itsenäinen, tietoturvallinen, resurssitehokas, ei kaupallinen ja pieni Linux-jakelu, suunniteltu tehokäyttäjille.

chroot = Unix-työkalu, jolla voidaan muuttaa suoritettavan prosessin ja sen lapsien juurihakemisto siten, että ne eivät voi nähdä tai päästä käsiksi kansion ulkopuolisiin tiedostoihin.

Devops = Ketterä ohjelmistokehitysfilosofia, jolla yhdistetään ohjelmistokehitys (Dev) ja tuotanto (Ops). Sen pääpiirteet keskittyvät automaatioon ja ohjelmistokehityksen vaiheiden seuraamiseen sovelluksen rakentamisesta integraatioon, testaukseen ja julkaisuun.

FreeBSD = Unixin kaltainen käyttöjärjestelmä.

IDE = Ohjelmointiympäristö, joka tarjoaa helpottavia työkaluja kehitykseen ja ohjelmointikielen syntaksin korostusta (syntax highlight).

JFrog Artifactory = Binäärivarasto, josta voidaan hakea ohjelmistopaketteja asennusta varten.

Legacy = Vanhalla teknologialla rakennettu ohjelmisto, josta puuttuvat testit ja joka on hankalasti muutettava.

LXC = Linux-käyttöjärjestelmän ydintä hyödyntävä eristämisteknologia.

Node.JS = Alustariippumaton palvelinpuolen JavaScript-ajoympäristö.

PostgeSQL = Avoimen lähdekoodin relaatiotietokantajärjestelmä.

RabbitMQ = Avoimen lähdekoodin viestien välittäjä, joka hyödyntää AMQ-protokollaa.

Rancher = Docker-konttien hallintaan suunniteltu työkalu.

REST = Arkkitehtuurimalli, joka voidaan toteuttaa HTTP-protokollan määrittelemien Method-kentän sallittujen arvojen avulla.

Robot Framework = Testiautomaatiotyökalu, jolla voidaan testata sovelluksen toiminnallisuutta.

Selenium = Web-ohjelmistojen testaukseen suunniteltu ohjelmistokehys, jolla voidaan testata esimerkiksi käyttöliittymän toimintaa automaattisesti ohjelmistokehitysputkessa.

1 Johdanto

Opinnäytetyö tehtiin projektin aikana Eficodella. Eficode on suomalaislähtöinen ohjelmistoalan yritys, joka tarjoaa asiakkaalleen räätälöityä ohjelmistokehitystä, konsultointia, digitaalisia asiakaskokemuksia palvelumuotoilulla sekä ohjelmistokehityksen tehostamista Devops-palveluilla. Työn tavoitteena oli saada vanha monoliittinen legacy-järjestelmä siirrettyä moderniin mikropalveluarkkitehtuurimalliin. Työn suorittaminen edellytti aktiivista itseopiskelua mikropalveluista, viestintäjonoista ja kehitysmalleista miten palveluiden välillä viestitään. Tavoitteena oli kuvata, miten legacy-järjestelmä siirretään mikropalveluksi ja toteuttaa kyseinen muutos, koska vanha järjestelmä korvataan ylläpidettävämmällä ja modernilla teknologialla. Koska yrityksen Devops-palveluiden tavoitteena on automatisoida tuotteiden paketointi-, laadunvarmistus- ja julkaisuprosessia, oli luontevaa käyttää kontitusteknologioita ja itsenäisiä mikropalveluja työn suorittamiseen.

Kautta aikojen ohjelmistoja on kehitetty asettamalla ohjelmiston vaatimat osat yhteen pinoon samalle palvelimelle, antamalla sille yhdeksän kuukauden julkaisuelinkaari, pieni sekoitus ja lopputuloksena on legacya - joka toimii tai sitten ei. Legacy-ohjelmistojen tyypillisiä tunnuspiirteitä ovat monoliittinen arkkitehtuuri, järjestelmän osien vahva riippuvuus toisistaan, testien puute sekä vanhenevat teknologiat. Yleistä on myös rajapintojen tai selkeästi erotetun käyttöliittymäkerroksen puuttuminen. Tällöin esimerkiksi sisällön ulkoasun muotoilu tapahtuu taustajärjestelmässä, vaikeuttaen edelleen palvelun ylläpitoa ja jatkokehitystä.

Nykyaikaiset ohjelmistokehitystrendit ratkaisevat näitä ongelmia. Mikropalvelut ja rajapinnat auttavat pilkkomaan ja erottamaan sovelluksen eri osia kontteihin (containers), joita voidaan tarpeen mukaan poistaa käytöstä, vaihtaa tai voidaan kasvattaa jo olemassa olevien instanssien määrää ajon aikana, jolloin monoliittisesta arkkitehtuurista päästään eroon ja päivittämisestä tulee helpompaa.

2 Legacy

Tietokoneohjelmien juuret ulottuvat 1800-luvulle, jolloin Ada Lovelace loi ensimmäisen algoritmin vuosina 1842–1843, jota myöskin tunnetaan maailman ensimmäisenä tietokoneohjelmana [1, s. 116–118]. Tästä vierähti kokonaiset sata vuotta siihen, että ensimmäiset sähkövirralla toimivat tietokoneet luotiin. Pian tämän jälkeen alkoi syntymään ohjelmointikieliä. 1950-luvulla syntyneet kielet, COBOL ja FORTRAN, ovat nykypäivänäkin vielä käytössä legacy-järjestelmien keskusyksiköissä.

2.1 Mitä on legacy?

Michael Feathers [2, s. xv–xvii] kertoo legacy-koodin olevan koodia, josta puuttuvat testit. Se voi olla jonkun toisen kirjoittamaa, ulkopuolelta tullutta koodia. Ohjelmoijan näkökulmasta termi legacy tarkoittaa muutakin. Se voi tarkoittaa takkuista ja käsittämätöntä rakennetta omaavaa koodia, jota sinun pitää muuttaa, mutta et ymmärrä sitä. Tietotekniikan teollisuudessa legacysta on muodostunut slangisana koodille, joka on vaikeasti muutettavissa olevaa.

2.2 Miten legacya syntyy?

Ohjelmistot omistavat elinkaaren. Ohjelmistoja luodaan, käytetään, päivitetään ja lopulta poistetaan käytöstä. Ohjelmiston poistamiselta ei voida välttyä, vaan sitä yritetään siirtää. Ohjelmisto julistetaan kuolleeksi, kun järjestelmä, jolla ohjelmistoa suoritettiin, vanhentuu. Ohjelmiston elinkaaren aikana koodia muokataan ja siihen tehdään nopeita korjauksia, mikä yleensä heikentää ohjelmiston suunnitelmassa pysymistä sekä ylläpitämistä, mikä tekee hankalammaksi sen muuttamisen jatkossa. [3, s. 5–6.]

2.3 Monoliittisuus

Monoliittinen arkkitehtuuri on ohjelmistoarkkitehtuurin tyyli, joka tarkoittaa, että järjestelmä on paketoitu ja julkaistu yhdessä yksikössä. Yksiköllä tarkoitetaan suoritettavaa ohjelmaa tai hierarkkista kansiota lähdekoodia. Monoliittisessa arkkitehtuurissa yksi ohjelma tekee

kaiken. Se näyttää käyttöliittymän, hakee dataa, käsittelee käyttäjän syötteitä ja hoitaa bisneslogiikan suorittamisen [4, s. 3.]

Monoliittinen arkkitehtuuri ei suoraan tarkoita, että sovellus on legacya. On nopeampaa ja helpompaa kehittää monoliittista ohjelmistoa, koska ohjelmistokehitystyökalut (IDE) ovat keskittyneet rakentamaan yhtä sovellusta [4, s. 13]. Koska kaikki tarvittava on pakattu yhteen ohjelmaan, siitä löytyy muitakin hyviä puolia, kuten kompleksin kommunikoinnin puute verkossa, salauksen puuttuminen kommunikoinnissa eri osiin ohjelmistoa, sekä ohjelmiston julkaiseminen on suoraviivaista [5, s. 94].

Arkkitehtuurissa kuitenkin on suuria puutteita. Monoliittisuus vaatii, että ymmärretään, kuinka kaikki osat järjestelmästä sopivat yhteen projektin alusta asti. Osien tiukka yhdistyminen (coupling) toisiinsa tekee ongelmien korjaamisesta myöhemmin vaikeaa [5, s. 94]. Monoliittisessa käyttöjärjestelmän ytimessä (kernel) nämä ongelmat tulevat esille. Koska monoliittinen ydin ylläpitää muun muassa käyttöjärjestelmän muistinhallintaa, laitteistoajureita, ajastinta, levyjärjestelmää sekä verkkopinoa, yksi virhe leviää kaikkiin osiin, mikä aiheuttaa käyttöjärjestelmän kaatumisen [6, s. 9].

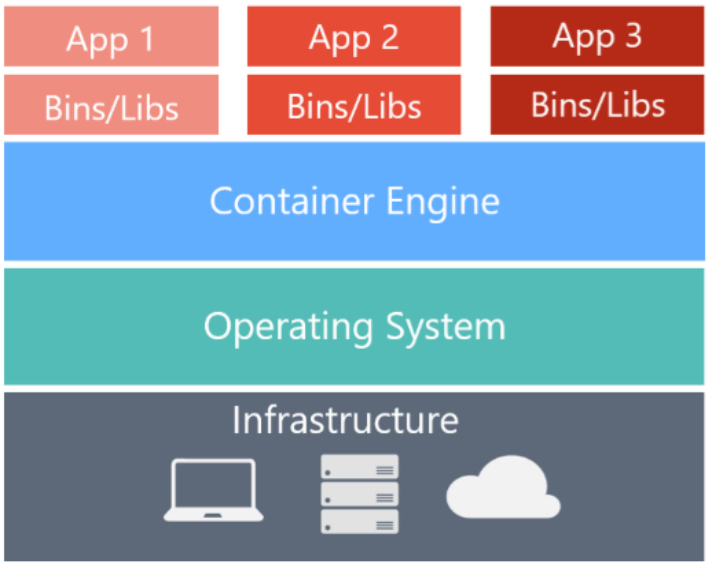
Monoliittisilla ohjelmistoilla on tapana kasvaa mammuttimaisiksi arkkitehtuureiksi. Joka kerta kun ohjelmistoon lisätään toimintoja, koodikanta kasvaa. Tietotekniikkayrityksissä työntekijät vaihtuvat ja kehitystiimit kasvavat, mikä aiheuttaa ylläpidon kustannuksien kasvun. Kehittämisestä tulee hidasta ja tuskaista, ketterästä kehityksestä on luovuttava, julkaiseminen on lähes mahdotonta sekä aikamääreet venyvät, kun kukaan ei enää ymmärrä sovellusta täysin ja lisäysten tekemisestä tulee aikaa vievää. [4, s. 4 – 6.]

Tästä aiheutuu syöksykierre. Jos koodikanta on jo valmiiksi vaikea ymmärtää, tulevat lisäykset tekevät siitä vieläkin hankalamman ymmärtää. Miljoonat rivit koodia riippuvuuksineen ylikuormittavat kehitystyökalut ja aiheuttavat käynnistymiseen, testaamiseen ja muokkaamiseen kuluvan ajan kasvun moninkertaiseksi. [4, s. 4.]

3 Kontittaminen

Kontittaminen ja eristäminen eivät ole uusia käsitteitä tietotekniikassa. Jotkut UNIX-johdanneiset ytimet ovat hyödyntäneet eristämistekniikkaa yli vuosikymmenien [7]. Vuonna 2008 Linux-käyttöjärjestelmän ytimen versioon 2.6.24 lisättiin kontitusteknologia LXC (LinuX Containers), jonka ansiosta kontitusteknologiat alkoivat yleistymään ja kehittymään [6, s. 11].

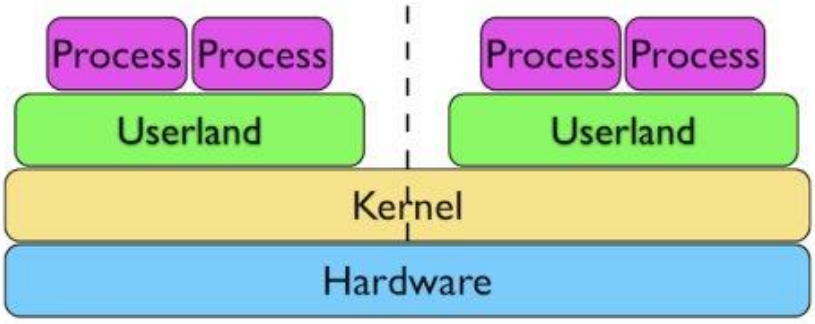
Ohjelmistojen kontittaminen on käyttöjärjestelmätason virtualisointitapa, jota käytetään hajautettujen ohjelmistojen suorittamiseen ja julkaisemiseen ilman virtuaalikoneen luomista jokaiselle sovellukselle. Tämä tarkoittaa sitä, että yhdellä isäntäkoneella voidaan suorittaa yhtä aikaa montaa sovellusta tai palvelua käyttäen samaa isäntäkoneen ydintä, mikä tekee konteista kevyempiä kuin virtuaalikoneet (kuva 1). Kontittamisessa paketoidaan ohjelmisto riippuvuuksineen ja sen ympäristön asetteet käyttöönottotiedostoina kontti-imageksi, josta voidaan myöhemmin luoda kontti [8, s. 2.]



Kuva 1. Kontittamisen perusperiaate [8]

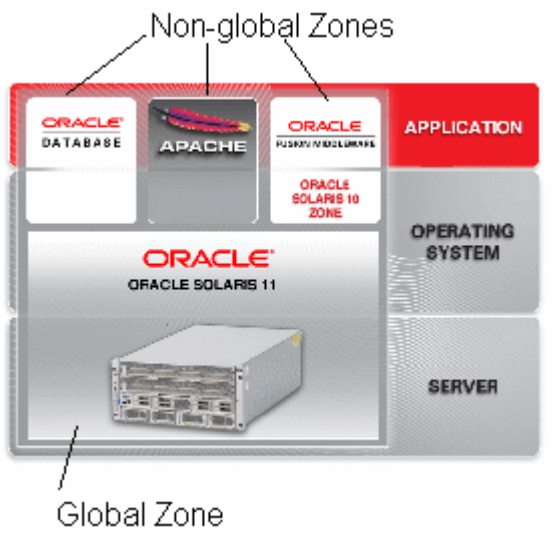
Vuonna 2000 jaettujen webhotellien tarjoaja R&D Associaties rahoitti FreeBSD:n kehitystä, jotta sen asiakkaat voisivat suorittaa eri versioita webohjelmistoista, tietokannoista sekä ohjelmointikielistä palvelimilla [14]. Tästä syntyivät FreeBSD-vankilat (jails), jotka ovat virtuaalisia ympäristöjä, jotka suoritetaan isäntäkoneessa [14]. Niillä mahdollistettiin ylläpitäjien pilkkoa järjestelmä useampaan itsenäiseen pienempään järjestelmään, joilla

pystyi olemaan oma IP-osoite per järjestelmä ja konfiguraatio [15]. Koska vankilat rakennettiin chroot-konseptin (change root directory) päälle, pystyivät prosessit vain menemään syvemmälle kansiopuuta asetetusta juurikansiosta [15]. Vankiloilla on omat käyttäjät sekä yksi pääkäyttäjä, jotka on rajoitettu vain vankilaympäristön sisälle, mikä tarkoittaa sitä, että käyttäjät eivät voi suorittaa toimintoja isäntäkoneessa (kuva 2). [15]



Kuva 2. FreeBSD-vankiloiden periaate [16]

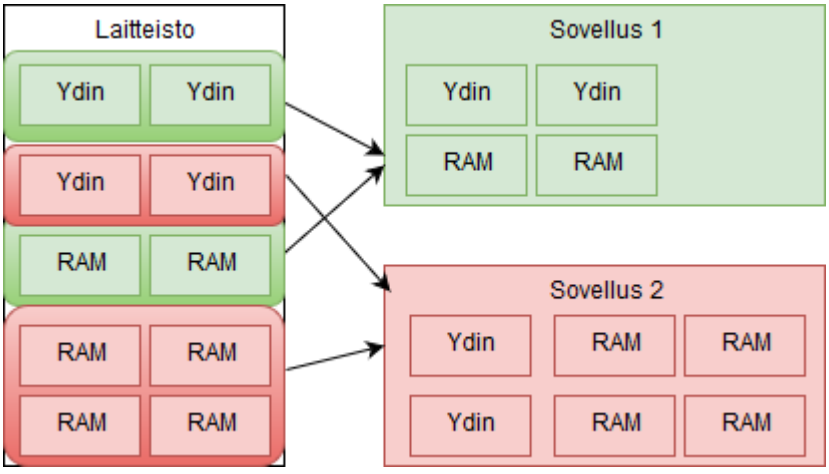
Vuonna 2004 Oracle kehitti Solaris Zonen, joka on järjestelmätason virtualisointi teknologian x86 ja SPARC järjestelmille. Ne tunnettiin alun perin Solaris Containersina. Alue (zone) on virtualisoitu käyttöjärjestelmäympäristö, joka on luotu Solaris käyttöjärjestelmästä. Alueen prosessit on eristetty isäntäkoneesta ja muista alueista siten, että ne eivät voi monitoroida tai vaikuttaa muihin alueiden prosesseihin edes pääkäyttäjän oikeuksilla. [17]. Yleisellä (global) alueella sijaitsevat Solariksen ydin, laitteistot sekä niiden ajurit, muistinhallintajärjestelmä, levyjärjestelmä sekä verkkopino. Jokainen alue hyödyntää yhtä yleistä aluetta, mutta niillä on omat levyjärjestelmät, prosessi-nimiavaruudet ja verkko-osoitteet (kuva 3) [18.]



Kuva 3. Solariksen alueet havainnollistettuna [18]

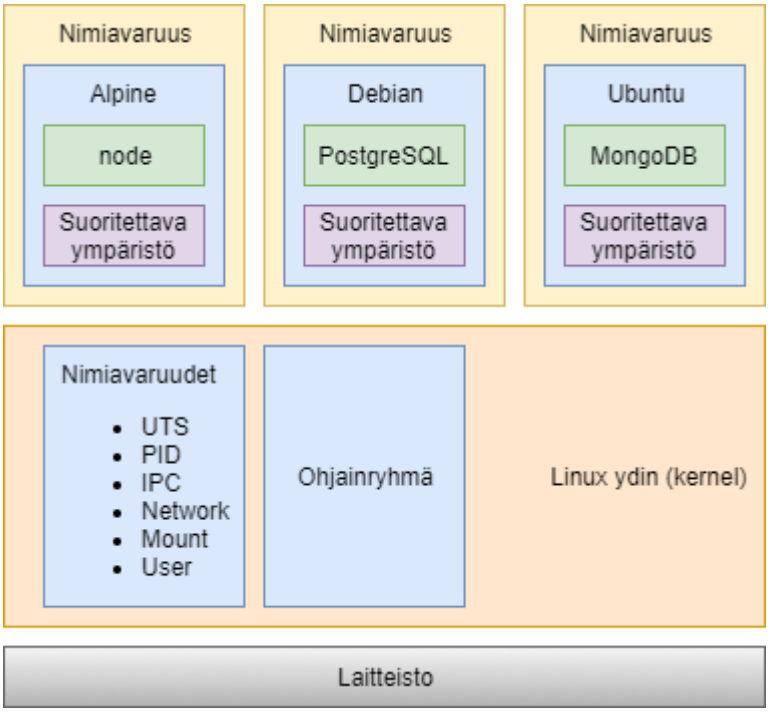
3.1 LXC

LXC hyödyntää Linuxista löytyvää ohjainryhmää (control group, cgroup), jolla se voi säädellä, kuinka paljon prosessiaikaa (CPU time) sekä sisään- ja ulostuloa (I/O) kontti saa käyttää. Ohjainryhmät ovat ytimen toimintoja, joilla voidaan hienojakoisesti säätää resurssien varaamista prosesseille (kuva 4). Ryhmät ovat hierarkkisia, joten ne perivät ominaisuuksia vanhemmiltaan. Prosessi tai prosessiryhmät ovat sidottuina samalla kriteerillä ja liitetty joukolla raja-arvoja ryhmiin. [6, s. 26.]



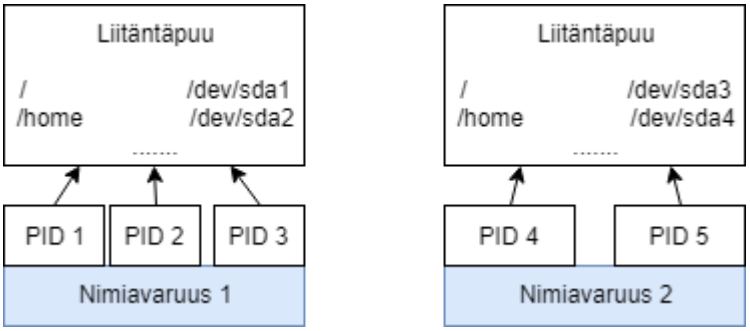
Kuva 4. Ohjainryhmillä resurssien jakaminen

Linuxin nimiavaruudet mahdollistavat kevyen prosessivirtualisoinnin antamalla prosessille ja sen lapsiprosesseille eri näkymät alla olevasta järjestelmästä. LXC hyödyntää kuutta eri nimiavaruutta. Niiden ansiosta prosessit voidaan erotella siten, että ne eivät näe toisiaan (kuva 5). [6, s. 11.]



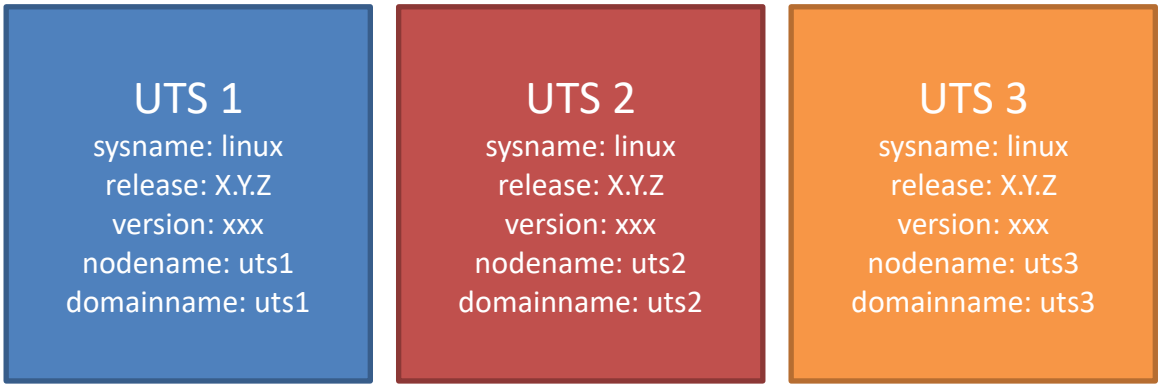
Kuva 5. LXC:n toiminta

Näistä nimiavaruuksista ensimmäinen on liitäntänimiavaruus (mount). Sillä mahdollistetaan prosessien käyttämän levyjärjestelmän liitoksien eroteltu näkymä. Koska uusi prosessi saa kopion kutsuvasta prosessista, voi se muuttaa liitäntäpuuta (mount tree) vapaasti vaikuttamatta vanhempaan prosessiin ja näin ollen mahdollistaa erilaisen levyjärjestelmän asettelun konttien sisällä (kuva 6). [6, s.11–13.]



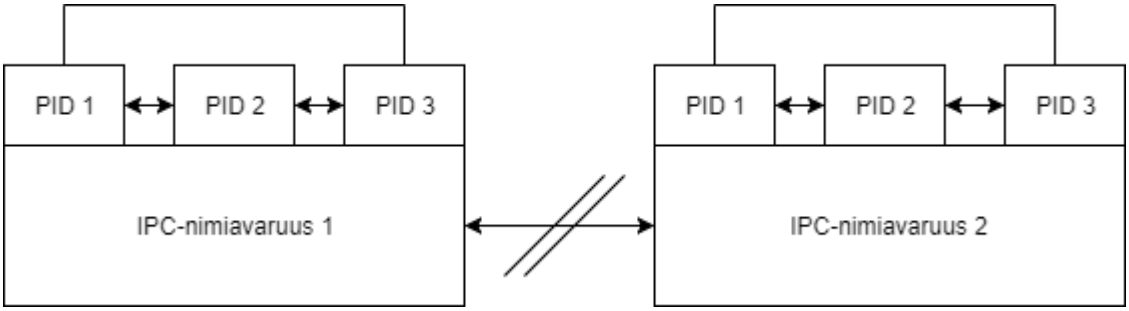
Kuva 6. Liitäntänimiavaruuden toiminta

Toinen nimiavaruus on Unix-ajanjakaja (Unix time-sharing, UTS), joka mahdollistaa isäntänimen ja domain-nimen eristämisen, jotta jokainen kontti saa oman tunnisteensa ja ovat mahdollista erottaa (kuva 7). UTS-nimiavaruudessa on mahdollista asettaa vain isäntä- ja domain-nimi, muihin tietoihin ei ole pääsyä. [6, s.13–16.]



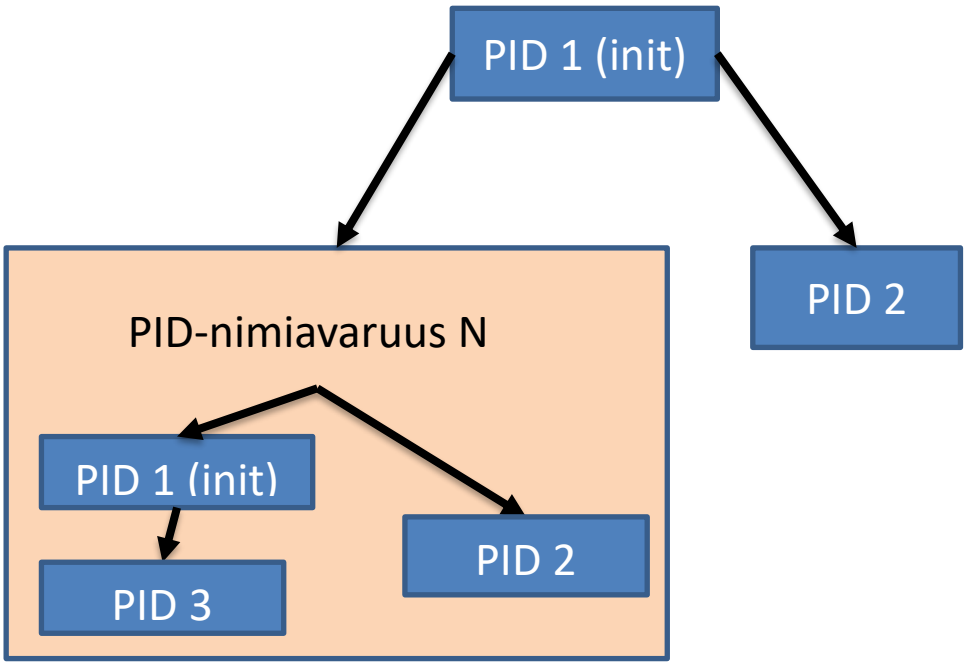
Kuva 7. UTS-nimiavaruuden toiminta

Kolmantena nimiavaruutena on prosessien välinen kommunikaatio (Inter-process communication, IPC), joka jakaa kahden prosessin tai säikeen välillä informaatiota, mahdollistaen prosessien erottamisen kontin sisällä (kuva 8). Tämä tarkoittaa sitä, että eri nimiavaruuksissa olevat prosessit omaavat oman jaetun muistin, semaforit ja viestintäjonot, eivätkä voi viestiä toisiin nimiavaruuksiin. [6, s.13–16.]



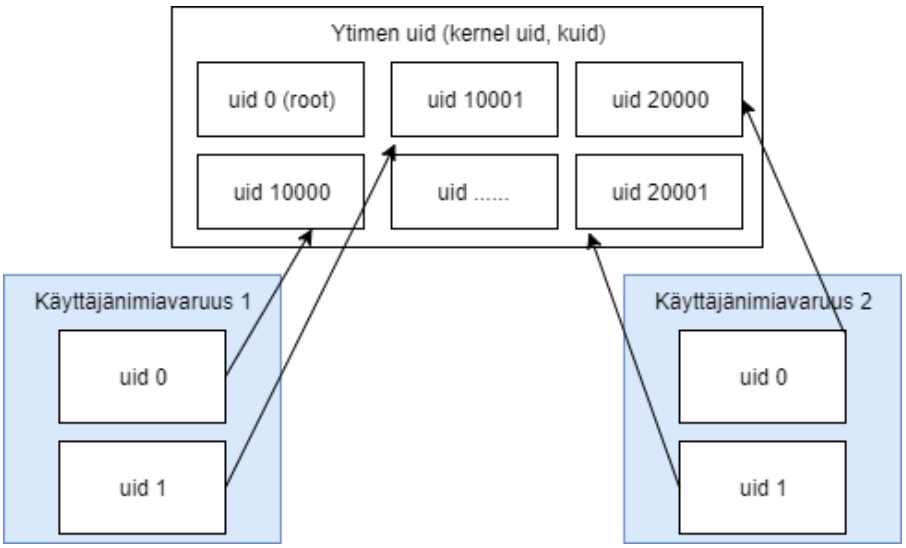
Kuva 8. IPC-nimiavaruuden toiminta

Neljäntenä nimiavaruutena toimii prosessin tunniste (PID), joka antaa kyvyn prosessille käyttää oletusnimiavaruudessa olevan prosessin tunnistetta. Tällä mahdollistetaan, että kontin sisällä voidaan suorittaa init-järjestelmä, jonka tehtävänä on käynnistää muut prosessit ja palvelut, aiheuttamatta ristiriitaa muiden prosessin tunnisteiden kanssa samassa käyttöjärjestelmässä (kuva 9). [6, s.13–16.]



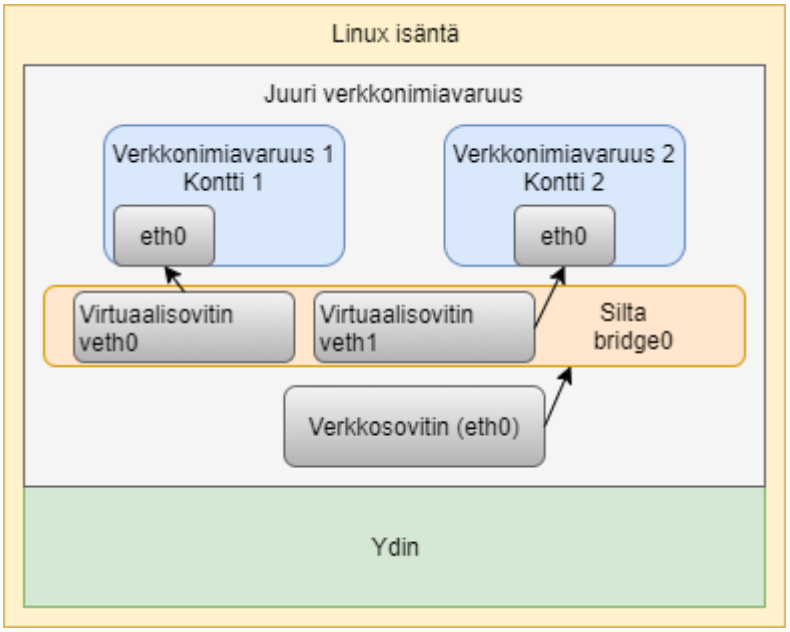
Kuva 9. PID nimiavaruuden toiminta

Viidennes nimiavaruus on käyttäjänimiavaruus (user), joka antaa prosessille eri käyttäjä- ja ryhmätunnisteen, mahdollistaen prosessin suorittamisen pääkäyttäjän oikeuksilla kontin sisällä, mutta ilman etuoikeuksia kontin ulkopuolella. Kontin sisällä oleva käyttäjätunniste yhdistetään ytimessä oleviin käyttäjätunnisteisiin (kuva 10). Tällä lisätään tietoturvaa, jos mahdollinen hyökkääjä pääsee pakenemaan eristetystä ympäristöstä, koska kontin ulkopuolisella käyttäjällä ei ole oikeuksia tehdä mitään. [6, s. 16–20.]



Kuva 10. Käyttäjänimiavaruuden toiminta

Kuudes ja viimeinen nimiavaruus on verkkonimiavaruus (network), joka tarjoaa verkkoresurssien, kuten laitteiden, osoitteiden, reittien ja palomuurisääntöjen erottamisen kopioimalla verkkopinon mahdollistaen prosesseille kyvyn kuunnella samaa porttia eri nimiavaruuksista (kuva 11). [6, s. 16–20.]



Kuva 11. Verkkonimiavaruuden toiminta

3.2 Docker

Docker on avointa lähdekoodia oleva eristämiseen suunniteltu moottori, joka automatisoi ohjelmistojen julkaisun kontteihin [9, s. 7]. Dockerin juuret ulottuvat vuoteen 2008, jolloin PaaS-yritys nimeltään dotCloud alkoi kehittää eristämisteknologiaan keskittynyttä moottoria nimeltä dotCloud. Tämä työkalu (dc) oli Python-pohjainen komentorivityökalu, joka on Dockerin esi-isä. Docker julkistettiin maailmalle ensimmäisen kerran vuonna 2013 PyConissa, kun sen kehittäjä, Solomon Hykes, piti lyhyen puheen ”the future of Linux containers”. Samana vuonna Dockerista tuli avointa lähdekoodia. [10.]

Dockeria kuvaillaan LXC:n jatkeena, koska se hyödyntää myöskin LXC:tä sekä Linuxin ohjainryhmiä ja ytimen nimiavaruuksia. Se alun perin aloitettiin projektina luomaan yhden sovelluksen LXC-kontteja, ja se esitteli merkittäviä uudistuksia LXC:hen tekemällä niistä siirrettäviä ja joustavampia. [11.]

Se määrittelee formaatin, jolla sovellus paketoidaan riippuvuuksineen yhdeksi objektiksi, joka voidaan siirtää mihin tahansa Dockerilla varustettuun laitteeseen. Tällä formaatilla voidaan määritellä ohjeet ja toiminnot kontin luomiseen. Tällä varmistetaan, että ohjelmiston ajoympäristö on sama millä tahansa laitteella. LXC:llä tällainen siirrettävyys ei ole mahdollista, koska se on sidottu järjestelmäkohtaisiin asetteisiin. [12.]

Docker on natiivisti saatavilla Windowsille ja Linuxille. Se on saatavilla myös MacOS:lle, mutta kontin suorittaminen ei tapahdu natiivisti, vaan virtualisointiohjelmiston sisällä [8, s. 3]. Windows Server Containers mahdollistaa Dockerin suorittaa Windows-kontteja LXC:n tapaisesti, eli jakamalla ytimen käytettäväksi konttien sisällä [13]. Docker käyttää Windowsin Hyper-V-virtualisointitekniikkaa, jotta se voi suorittaa ohjelmistoa natiivisti muilla alustoilla kuten Linuxilla virtuaalikoneessa [13].

Docker Hub on pilvessä (cloud) toimiva rekisteripalvelu, joka koostuu useammasta säilytyspaikasta (repository), minne voidaan tallettaa eri versiota samasta ohjelmistosta tai palvelusta Docker imagena. Se antaa kehittäjälle tavan julkaista omia Docker-imagejaan muille käytettäväksi ja ladata muiden tekemiä imageja käytettäväksi. Se voidaan integroida versiohallintaan, jolloin jokaisesta muutoksesta koodikantaan voidaan rakentaa automaattisesti image, joka on valmis julkaistavaksi. Web-kiinnittimien (hook) avulla julkaiseminen voidaan tehdä automaattiseksi. [19.]

Docker Hubin säilytyspaikoissa jokainen image on versioitu. Dockeriin on luotu merkitsemistyökalu, jolla eri versiot imageista voidaan erottaa. Koska vanhemmat versiot ovat saatavilla, on mahdollista käyttää eri versioita palveluista tai palauttaa vanha versio, jos päivittäminen rikkoi ohjelmiston. Viralliset Docker Hubin säilytyspaikat tarjoavat pohjaimageja käyttöjärjestelmistä sekä valmiiksi rakennettuja imageja suosituista palvelinohjelmistoista, ohjelmointikielistä ja tietovarastoista nopeaa kehittämistä varten. Lisäksi se tarjoaa yksityisen säilytyspaikan imageille, joka vaatii kirjautumisen. [19.]

Docker on suunniteltu suorittamaan vain yhtä prosessia per kontti, vaikkakin mahdollistaa useamman prosessin käynnistämisen. Sen ideologiana on ajaa jokainen prosessi eri kontissa, kun taas LXC suorittaa init-järjestelmän, jolla voi suorittaa montaa prosessia. Se tarjoaa helpomman tavan päivittää suoritettavia ohjelmia, koska vain päivitettävä kontti sammuu ja muut jatkavat suoritustaan. Tätä ideologiaa käytetään myös mikropalveluarkkitehtuurissa [13].

3.3 Mikropalveluarkkitehtuuri

Mikropalveluarkkitehtuuri on tapa, jota käytetään kehityksessä luomaan itsenäisesti julkaistavia, pieniä ja modulaarisia palveluita, missä jokainen palvelu suorittaa uniikkia prosessia ja kommunikoi hyvin määritellyn rajapinnan kautta toistensa kanssa. Adrian Cockcroft [4, s. 6] määrittelee mikropalveluarkkitehtuurin palvelupainotteiseksi (service oriented) arkkitehtuuriksi, jotka koostuvat löysästi kytketyistä elementeistä, joilla on rajoitettu asiayhteys. Chris Richardson [4, s. 7] lisää määritelmään sen, että palveluita voidaan skaalata (scale) ja jokaisella palvelulla on oma tietovarasto.

Mikropalveluista löytyy monia hyötyjä verrattuna monoliittiseen arkkitehtuuriin. Jokainen palvelu on pienehkö, joten kehittäjillä on helpompaa ymmärtää se kokonaisuudessaan. Palveluiden käynnistyminen ei ole raskas prosessi ja yleensä ne käynnistyvät nopeammin kuin monoliittiset. Koska palvelut ovat itsenäisiä, niitä voidaan päivittää ja korvata ajon aikana. [4, s.11–12.]

Mikropalvelut mahdollistavat sen, että kehittäjien ei tarvitse sitoutua vain yhteen teknologiapinoon (technology stack) koko sovelluksen elinkaaren ajan, vaan uusiin ominaisuuksiin voidaan ottaa mikä tahansa ohjelmointikieli tai ohjelmistokehys. Lisäksi koska teknologia kehittyy, on mahdollista kirjoittaa sovelluksen osa uudestaan paremmilla kielillä tai korvata paremmalla teknologialla. Vaikka teknologia ei toimisi, on mahdollista korvata se kehityksen aikana, koska se on vain yksi osa-alue kokonaisesta sovelluksesta. [4, s. 12.]

Monoliittisessa arkkitehtuurissa oli ongelmia viansietokyvyn (fault tolerance) kanssa, koska yksi virhe levisi muihin järjestelmiin. Mikropalveluissa tämä ongelma ratkeaa implisiittisesti, koska yhden palvelun hajoaminen ei vie koko järjestelmää alas, vaan ainoastaan kyseisen palvelun, jolloin muut palvelut silti jatkavat toimintaansa. [4, s.12.]

Arkkitehtuuri ei ole suora ratkaisu kaikkiin IT-ongelmiin, koska huonosti kirjoitettua ja suunniteltua koodia ei voida korjata eristämisellä. Ei ole olemassa konkreettista tapaa tai algoritmia hajottaa järjestelmää eri palveluihin, vaan kaikki riippuu käyttötarkoituksesta. Tekemällä kaikesta itsenäisen palvelun ajautuu ongelmiin, missä arkkitehtuuri on sekavampi ja hankala ymmärtää, ellei jopa hajautettu monoliitti, jossa palvelut riippuvat toisistaan ja niitä ei voida ottaa käyttöön yksitellen. Viestinnästä tulee hankalampaa, koska tarvitaan prosessien välistä viestintää (inter-process communication, IPC), mutta palvelut eivät yleensä sijaitse samalla palvelimella. [4, s. 13.]

4 Ohjelmiston muuttaminen mikropalveluiksi

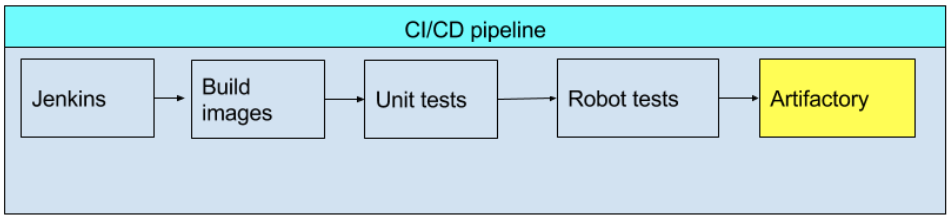
Tässä työssä käsitellään legacymaista monoliittista verkkosovellusta, joka korvataan uudella, nykyaikaisemmalla, nykypäivän ohjelmistosuunnittelutrendejä noudattavalla sovelluksella. Sovelluksessa hyödynnetään mikropalveluarkkitehtuuria, rajapintoja REST-arkkitehtuurimallilla, orkestrointia Rancherilla ja DevOps-filosofiaa.

Aikaisempi ohjelmisto ei hyödyntänyt kontitusteknologioita eikä automaattista jatkuvaa integraatiota tai julkaisua. Yksikkö- sekä käyttöliittymätestien puute teki kehityksestä vaikeaa, sillä ei ollut varmuutta, toimiko sovellus halutulla tavalla tai rikkoiko uusi ominaisuus jo olemassa olevia ominaisuuksia. Nämä ongelmat ratkaistiin heti projektin alkaessa.

Uusi ohjelmisto hyödyntää Dockeria kontitukseen. Sen jokainen eri osa on pilkottu erilliseen konttiin. Käyttöliittymäkerros kommunikoi taustajärjestelmän kanssa REST-rajapinnan (API) kautta. Taustajärjestelmällä on PostgreSQL-tietokanta käytössä erillisessä kontissa, jota se käyttää tiedon varastointiin.

4.1 Uuden sovelluksen ohjelmistokehitysputki

Projektissa hyödynnetään jatkuvaa integraatiota sekä julkaisua. Joka kerta kun paikalliseen versiohallintaan lisätään muutos (commit), jatkuvan integraation työ (job) suoritetaan. Tämä työ hakee muutokset ulkopuolelta tulevasta tietosäiliöstä (remote repository) ja alkaa suorittamaan ohjelmistokehitysputkea (pipeline), joka on määritelty projektille. Kuvassa 12 nähdään projektille määritelty kehitysputki.



Kuva 12. Projektin jatkuvan integraation ja julkaisun ohjelmistoputki

Ohjelmistokehitysputken tavoite on pitää ohjelmistokoodi toimivana ohjelmiston kehityksen ajan sekä automatisoida toistuvat työtehtävät niin, että kehittäjän ei tarvitse tehdä niitä manuaalisesti, säästäen aikaa ja vaivaa. Putkessa rakennetaan kontti-imaget, jotta ne

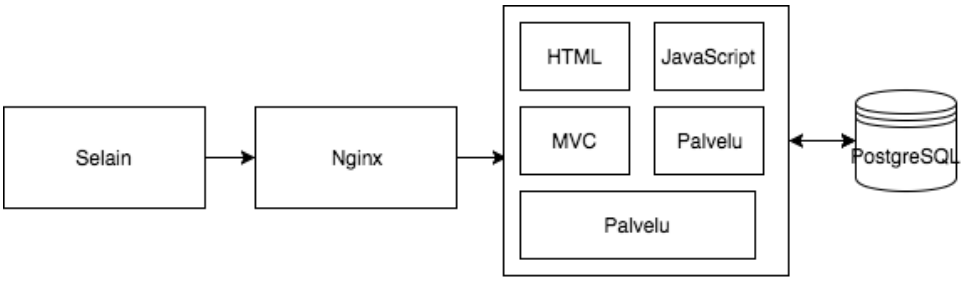
ovat käytettävissä ja siirrettävissä. Rakennuksen jälkeen infrastruktuuri on valmis suoritettavaksi. Konttien suorituksen aikana ne menevät testausprosessin läpi.

Jokaista konttia vasten suoritetaan yksikkötestit. Niillä varmistetaan, että muutokset eivät rikkoneet jo olemassa olevia toimintoja, varmistetaan rajapintojen päätepisteiden toimivuus sekä analysoidaan ohjelmistokoodin laatu. Jos yksikkötestit suoriutuvat, siirrytään automatisoituun käyttöliittymätestaukseen, jossa erillinen kontti, joka on suunniteltu pelkästään käyttöliittymätestaukseen, suorittaa automaattiset testit Firefox-selaimella käyttämällä Robot Frameworkia ja Seleniumia.

Kun yksikkö- ja käyttöliittymätestit on suoritettu onnistuneesti, on varmistettu, että ne ovat julkaisukelpoisia imageja. Valmiiksi rakennetut imaget merkitään uusimmiksi ja julkaistaan JFrog Artifactory-binäärivarastoon. Tästä varastosta orkestrointisovellus Rancher hakee uusimmat kontti-imaget ja suorittaa ne uusiksi konteiksi korvaten vanhat suorituksessa olevat kontit.

4.2 Alkutoimenpiteet

Sovelluksen käyttöliittymäkerros ja taustajärjestelmä oli rakennettu samaan pakettiin, joten sivuston renderöinti tapahtui palvelinpuolella. Tietokantayhteys otettiin taustajärjestelmässä, joten bisneslogiikka hoidettiin ilman rajapintoja saman järjestelmän sisällä ilman ulkoisia palveluita (kuva 13). Vanhan sovelluksen tietokantaa pystyttiin uusiksi käyttämään suoraan.



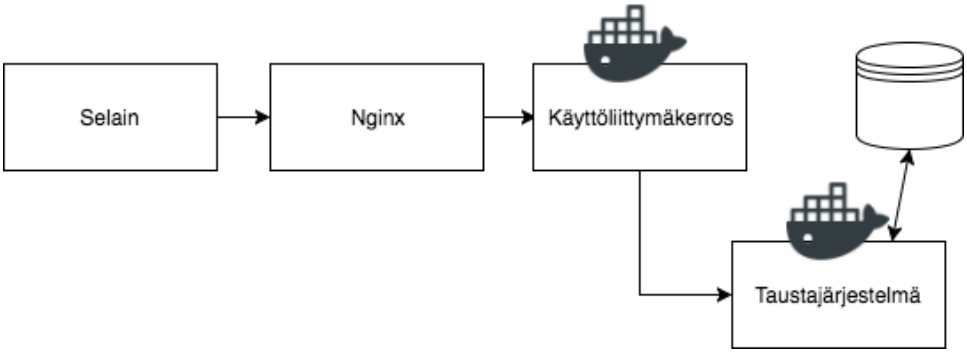
Kuva 13. Vanhan sovelluksen monoliittinen arkkitehtuuri

Vanhassa sovelluksessa käyttäjän syötteiden käsittely hoidettiin ohjainten kautta. Jos ohjain tarvitsi pääsyä tietokantaan asiakkaan puolelta, se lähetti AJAX-pyynnön samalla palvelimella sijaitsevaan skriptitiedostoon, josta tietokannasta tuleva tieto oli saatavina

JSON-muodossa (JavaScript Object Notation). Kaikki interaktiivisuus käsiteltiin JavaScriptin avulla, jolloin myös välillä AJAX-pyynnöt renderöivät sivua asiakkaan puolella.

4.3 Kontittaminen

Koska vanha sovellus oli rakennettu monoliittisesti, oli luontevaa ensiksi pilkkoa MVC-rakenne. Vanhan monoliitin osat jaettiin omiin kontteihinsa. Erotuksesta syntyi kontti käyttöliittymäkerrokselle, taustajärjestelmälle sekä tietokannalle (kuva 14).



Kuva 14. Monoliittinen arkkitehtuuri eroteltuna mikropalveluiksi

Ohjelmistokoodia tarvitsi muokata, jotta se saatiin toimimaan uudella arkkitehtuurilla. Olemassa olevien moduulien pilkkominen omiksi palveluiksi monoliitista ei onnistu suoraan. Oli tärkeää saada ohjelmisto toimimaan mikropalveluina samalla lailla kuin se toimi monoliittisena, ennen uusien ominaisuuksien lisäämistä.

Jokaiselle palvelulle rakennettiin oma kontti. Ohjelmistokoodia suorittavat imaget rakennettiin käyttäen virallisia Dockerhubin imageja pohjana, mutta niihin lisättiin omat asetteet. Palvelinohjelmistot käyttävät muuttamattomia imageja, ja niille annetaan vain parametrejä, joilla ne toimivat halutulla lailla.

Ohjelmisto rakennettiin Node.JS-ohjelmistokirjaston ympärille, joten oli luontevaa käyttää valmiita Node.JS:n kontti-imageja lähdekoodin suorittamiseen, samalla säästäen vaivaa ja aikaa, kun imagea ei tarvitse rakentaa alusta alkaen (kuva 15). Halusin, että imaget ovat mahdollisimman pieniä, joten käytimme kaikissa konteissa Alpine Linux -pohjaisia kontti-imageja.

```
FROM node:alpine

# Create app directory
RUN mkdir -p /usr/src/app
WORKDIR /usr/src/app

# Install app dependencies
COPY package.json /usr/src/app/
RUN npm install

# Bundle app source
COPY . /usr/src/app

EXPOSE 9001

CMD [ "npm", "start" ]
```

Kuva 15. Ohjelmistokoodia suorittavan kontin Dockerfile

Koska sovellus on pilkottu moneen konttiin, on työlästä käynnistää kaikki palvelut yksitellen. Tämän ongelman ratkaisua varten Docker tarjoaa työkalua nimeltään docker-compose, joka ennen käynnistymistään rakentaa ensiksi kontti-imaget, jonka jälkeen luo kontit, joihin imaget liitetään. Tämä jälkeen se käynnistää kaikki palvelut ja yhdistää riippuvuutena olevat kontit tunnistettavaksi isäntänimillä. Docker-composelle voidaan antaa useampi konfigurointitiedosto, jotka se automaattisesti yhdistää. Oletuksena se lataa docker-compose.yml-tiedoston, jonne määriteltiin vain ohjelmiston tuotannossa suorittamiseen tarvittavat palvelut. Kaikki testaukseen liittyvät kontit määriteltiin eri tiedostoon, jotka suoritetaan vain jatkuvan integraation ohjelmistokehitysputkessa.

Docker-composen konfiguraatiotiedostolla voidaan asettaa ympäristömuuttujia konteille, avata ja ohjata isännän ja kontin portteja, liittää levyjaot isännältä kontille ja yhdistää ne samaan verkkoon, jotta ne voivat kommunikoida keskenään. Ympäristömuuttujilla tehtiin ohjelmiston konfigurointi helpoksi koodissa, koska isäntänimeksi voitiin antaa kontin nimi, jonka kautta selvitettiin kontin oikea IP-osoite (kuva 16). Tämän ansiosta kontti-imagea ei tarvitse rakentaa uudestaan, kun asetteet tulevat ympäristömuuttujissa.

```yaml
version: '2'

services:
  db:
    environment:
      POSTGRES_DB: demo
      POSTGRES_PASSWORD: demo
      POSTGRES_USER: demo
    image: postgres:9.4
    volumes:
      - dbdata:/var/lib/postgresql/data

  backend:
    build: backend
    command: ./wait-for db:5432 -- npm run dev
    environment:
      DATABASE_URL: postgres://demo:demo@db/demo
      GOOGLE_OAUTH_ID: ${GOOGLE_OAUTH_ID}
      GOOGLE_OAUTH_SECRET: ${GOOGLE_OAUTH_SECRET}
      SECRET: ${SECRET}
      RABBITMQ_URL: amqp://test:password@rabbitmq
    volumes:
      - ./backend/src:/usr/src/app/src
      - media_files:/usr/src/app/media_files
    ports:
      - 9000:9000
    depends_on:
      - db
      - rabbitmq
```

Kuva 16. Docker-compose tiedosto

Docker-ympäristöllä varmistettiin sovelluksen liikkuvuus. Isäntäkone tarvitsi vain Docker-asiakasohjelman, jolla koko infrastruktuuri voitiin pystyttää. Ei tarvittu enää manuaalista työtä tiedostojen siirtämiseen esimerkiksi virtuaalikoneeseen, järjestelmän asetteiden muuttamista tai ohjelmiston riippuvuuksien asentamista järjestelmään. Kehitysympäristössä Docker-kontit saavat levyjakona lähdekoodikansion isäntäkoneelta, jolloin kaikki muutokset, jotka tehdään isäntäkoneella siirtyvät automaattisesti kontin sisälle, jolloin kehittäjien on mahdollista käyttää mitä tahansa työkalua kehitykseen.

Jatkuvasti muuttuviin kontteihin luotiin tapa nähdä ovatko tiedostot muuttuneet. Kun kontti huomasi, että tiedostot muuttuivat, käynnistettiin sovellus uudestaan uusimmilla muutoksilla (kuva 17). Tällä tavoin pystyttiin välttämään kaikki manuaalinen työ muutoksien yhteydessä ja keskittymään vain kehittämiseen.

```
dev_1       | webpack: Compiled successfully.
backend_1   | [nodemon] restarting due to changes...
backend_1   | [nodemon] starting `node server.js`
backend_1   | Fri, 13 Oct 2017 10:46:34 GMT sequelize
b/sequelize.js:236:13
backend_1   | App listening on port 9000
rabbitmq    |
```

Kuva 17. Sovelluksen automaattinen käynnistyminen uudestaan muutoksen yhteydessä

4.4 Viestintä konttien välillä

Vanhan sovelluksen rajapintojen puuttuminen arkkitehtuurisen syyn vuoksi oli seuraava käsiteltävä asia muutosprosessissa. Käyttöliittymäkerros tarvitsi pääsyn tietokannassa olevaan dataan. Koska selaimella suoritettava JavaScript ei pysty suorittamaan tietokantakyselyitä, luotiin taustajärjestelmä, joka pystyi suorittamaan tietokantakyselyitä ja lähettämään datan takaisin käyttöliittymäkerrokselle.

Viestintä hyödyntää REST-rajapintoja. Taustajärjestelmään luotiin päätepisteitä (endpoints), joista jokainen teki erinäisiä toimenpiteitä. Päätepisteisiin voidaan lähettää parametrejä, kuten käyttäjän tai osaston tunniste (kuva 18). REST on arkkitehtuurimalli, joka voidaan toteuttaa HTTP-protokollan määrittelemien Method-kentän sallittujen arvojen avulla. Kaikki HTTP-protokollan GET-pyynnöt lukevat tietokannasta dataa, POST-pyynnöt luovat uusia entiteettejä, PUT-pyynnöt päivittävät olemassa olevaa entiteettiä ja DELETE-pyynnöt poistavat entiteetin.

```
privateRouter.get('/users/:userId/projects', projects.getAll);
privateRouter.post('/users/:id/picture', users.addPicture);
privateRouter.get('/users/:userId/skills', skills.getAll);

privateRouter.post('/skills_users', skillsUsers.create);
privateRouter.post('/skills_users/remove', skillsUsers.remove);
privateRouter.delete('/skills_users/:id', skillsUsers.remove);

privateRouter.post('/project_users', projectUsers.create);
privateRouter.post('/project_users/remove', projectUsers.remove);
privateRouter.delete('/project_users/:id', projectUsers.remove);

privateRouter.get('/departments', departments.getAll);
privateRouter.get('/departments/:departmentId/users', users.getAll);
```

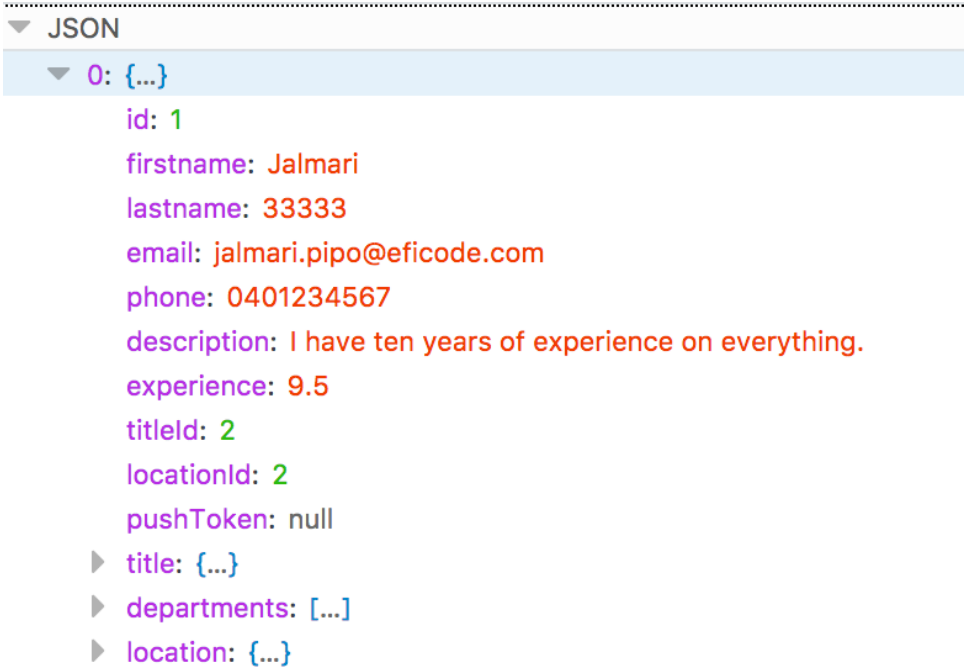
Kuva 18. Päätepisteiden ja niiden toimintojen määritteleminen

Näiden pyyntöjen taakse luotiin tietokantakyselyitä, jotka eroteltiin sekä ryhmiteltiin päätepisteiden nimillä. Uudessa sovelluksessa hyödynnetään ohjaimia kuten aikaisemmassakin sovelluksessa. Käyttäjän syöte lähetetään reitittimelle HTTP-pyyntönä, josta reititin ohjaa pyynnön oikealle ohjaimelle. Ohjain tietää, mitä sen tulee tehdä saadessaan tapahtuman. Se voi tarkistaa, onko tieto oikeaa, onko tarvittavat parametrit asetettu ja muokata vastausta parametrien mukaan (kuva 19).

```
exports.getAll = async (ctx) => {
  let options = {};

  if (ctx.params.userId) {
    options = {
      where: { '$users.id$': ctx.params.userId },
      include: [{
        model: database.User,
        as: 'users',
        attributes: ['id'],
      }, {
        model: database.SkillGroup,
        attributes: ['name', 'id'],
      }],
    };
  }

  let skills = await database.Skill.findAll(options);

  if (ctx.errors) {
    ctx.throw(400, 'VALIDATION_ERROR');
  }

  ctx.body = skills.map((skill) => { return skill.toJSON(); });
  ctx.status = 200;
};
```

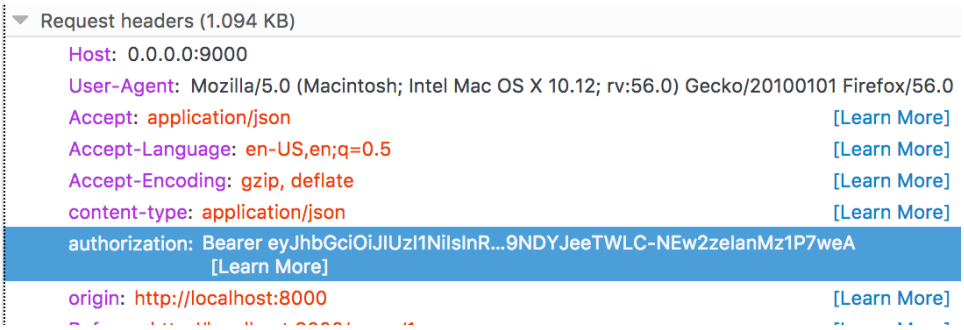
Kuva 19. Esimerkki parametrien vaikutuksesta vastaukseen

Ryhmityksen tarkoituksena oli pitää rajapinta selvänä ja samalla tuoda kehittäjälle ilmi, minkälaisen datan kanssa hän on tekemisissä. Esimerkiksi /users-haaran päätepisteet kertovat, että kehittäjä työskentelee käyttäjä-entiteetin kanssa. Päätepisteille myös lisättiin toimintoja, jotka olivat sidoksissa ryhmän nimeen, esimerkiksi /departments/<id>/users-haara, joka kertoo osaston tunnisteella kaikki työntekijät, jotka kuuluvat kyseiseen osastoon (kuva 20).



Kuva 20. /departmens/<id>/users-haaran vastaus

Pyynnön oikeuksien tarkistusta ei tarvittu vanhassa järjestelmässä, koska siinä ei ollut ulkoisia palveluita. Uudessa järjestelmässä oli tarpeen luoda tunnistautuminen, jolla varmistettiin, oliko pyynnön lähettävä käyttäjä tunnistautunut. Koska HTTP-protokolla on tilaton, ei vanhan järjestelmän istuntopohjainen tunnistautuminen ollut vaihtoehto. Tunnistautuminen toteutettiin JSON Web Tokeneilla (JWT), jolloin käyttäjän identiteetti saatiin siirrettyä http-otsikoissa palvelusta toiselle samalla varmistaen, että token on aito (kuva 21).



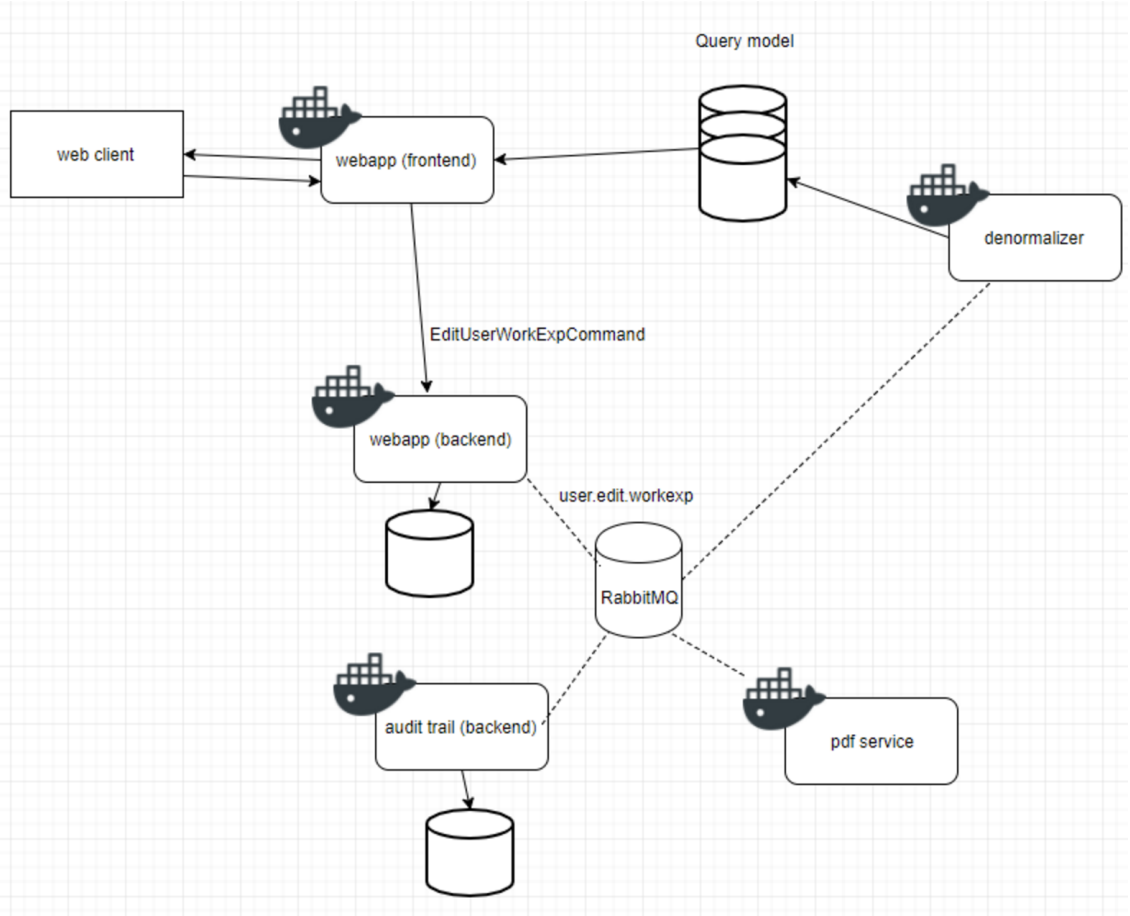
Kuva 21. JSON Web Tokenin siirtäminen HTTP-otsikoissa

4.5 Uudet ominaisuudet

Uuteen sovellukseen luotiin lisää ominaisuuksia, joita ei löytynyt vanhasta. Ominaisuuksia varten rakennettiin omat palvelut, joilla vältetään taustajärjestelmän monoliittiseksi muodostuminen. Arkkitehtuuri koki muodonmuutoksen mikropalveluiksi, mikä auttoi kehitystiimiä keskittymään omiin aihealueisiinsa nopeuttaen kehitysprosessia ja tuomalla oman erikoisosaamisen tiettyyn osaan ohjelmistoa.

Lisäominaisuuksia varten arkkitehtuuriin lisättiin dokumenttitietokanta omalle palvelulle. Tietokannasta tehtiin Docker-kontti ja sille lisättiin oma levyjako pitääkseen tietokannan pysyvästi. Uuden palvelun tarkoitus oli pitää toimintalokia tapahtumapohjaisesti tallessa.

Palvelupohjaisen arkkitehtuurin vuoksi järjestelmässä ei ollut enää yhtä niin sanottua taustajärjestelmää vaan useampi palvelu jotka käytännössä toimivat taustajärjestelminä (kuva 22). Jokaisella taustajärjestelmillä on käytettävissä oma tietokanta, jotka eivät riipu toisistaan.



Kuva 22. Mikropalveluarkkitehtuuri uusien ominaisuuksien jälkeen

4.6 Viestintä laajemmassa skaalassa

Rajapintojen lisäksi palvelut viestivät keskenään AMQP:n (Advanced Messaging Queuing Protocol) avulla. Kaikki palvelut ovat kiinnittyneet RabbitMQ-viestintäjonoon, jonka kautta rajapintareititin lähettää komentoja epäsynkronisesti. Näillä komennoilla on ryhmätunniste, joilla erotetaan, mitä tyyppiä komento on (kuva 23).

```
bus.publish('user.edit.personal', Object.assign({}, user.toJSON()));
```

Kuva 23. Komennon lähettäminen taustajärjestelmästä

Palvelut merkitsevät, mitä komentoja haluavat käsitellä antamalla komennon ryhmätunnisteen lausekkeeksi, joka voi sisältää jokerimerkkejä ja sääntöjä, kuten että tunnisteen täytyy alkaa tai loppua kirjainyhdistelmään (kuva 24). Jos komennon ryhmätunnistelauseke täsmää kuunneltavaan lausekkeeseen, se otetaan käsittelyyn ja poistetaan jonosta. Palvelu voi myös hylätä viestin, jolloin se myös merkitään käsitellyksi.

```
bus.subscribe('user.edit.*', (event) => {
  repo.get(event.id, (err, user) => {
    if (err) throw err;

    user.update(event);

    repo.commit(user, err => {
      if (err) throw err;
    })
  });
});
```

Kuva 24. Komennon vastaanottaminen eri palvelusta

CAP-teoreema (Consistency, Availability, Partition Tolerance) sanoo, että voit saavuttaa vain kaksi seuraavista kolmesta: yhtenäisyys, saatavuus ja osituksen sietokyky. Tämä arkkitehtuurimalli hyödyntää saatavuutta ja osituksen sietokykyä, mikä tarkoittaa sitä, että ohjelmisto voi jatkaa toimintaan, vaikka tarvittavaa tietoa ei ole saatavilla heti. Tämä malli perustuu CQRS-ohjelmistokehitysmalliin (Command Query Responsibility Segregation).

Tarkoituksena ei ole, että tieto olisi yhtenäistä heti toimintojen jälkeen, vaan että se olisi lopulta yhtenäistä.

CQRS-mallin tarkoituksena on erottaa muutosten tekeminen haettavasta tiedosta. Sovelluksessa CQRS-mallia hyödynnetään siten, että jokaisen palvelun tietokannat yhdistetään normalisoituun tietokantaan, joka toimii kyselymallina käyttöliittymäkerrokselle. Tässä tietokannassa on kerätty kaikkien palvelujen tarvittava tieto yhdeksi, helposti käsiteltävästi olevaksi tiedoksi. Kun komennot lähetetään AMQ-protokollan kautta, palvelut voivat kirjoittaa omaan tietokantaansa, jonka jälkeen ne lähettävät tapahtuman viestijonoon. Komennot kirjataan preesensissä, koska se viestii siitä, että tapahtumaa ollaan tekemässä. Tapahtumat kirjataan imperfektistä, jotta tiedetään komennon suoriutuneen ja odotetaan jälkitoimintoja. Esimerkiksi user.edited.education viestii siitä, että käyttäjä on muokannut omaa koulutustaan onnistuneesti. Näin kaikki palvelut tietävät, että tapahtuma on kirjattu ylös ja voivat tehdä omassa palvelussaan tarvittavat asiat. Jonoa pitkin tieto välittyy lopulta normalisoituun tietokantaan, jolloin tieto on lopulta yhtenäistä.

5 Pohdinta

Kalle Sirkesalon [21] kommentti ”ei se huono koodi korjaannu kontittamisella” pitää paikkansa. Ei voida olettaa, että kontittamalla huonoa koodia syntyisi taianomaisesti parempi sovellus, koska noudatettiin nykypäivän ohjelmistotrendejä. Nykypäivänä mikropalveluilla yritetään korjata ongelmia, ennen kuin niitä edes on. Lopputuloksena kuitenkin päädytään tilanteeseen, missä mikropalvelut eivät korjaa ongelmaa vaan kontittaminen teki sovelluksesta yhtä sekavan.

Tässä työssä mikropalvelujen käyttöönotto sujui suunnittelun vuoksi hyvin. Oli totta, että mikropalveluarkkitehtuuri tekee arkkitehtuurista monimutkaisemman kuin monoliittinen. Vaati paljon opiskelua, miten palveluiden tulisi viestiä keskenään, miten palvelut erotetaan keskenään ja mikä on kunkin palvelun rooli.

6 Yhteenveto

Opinnäytetyön tavoitteena oli kuvata ja muuttaa monoliittinen arkkitehtuurimalli mikropalveluiksi. Tavoitteessa onnistuttiin siten, että alkuperäinen ohjelmisto pilkottiin erotetuiksi taustajärjestelmäksi ja käyttöliittymäkerrokseksi. Palvelut eivät olleet enää rajoittuneita yhdelle palvelimelle, vaan niitä pystyttiin suorittamaan eri palvelimilla itsenäisesti ja sammuttamaan tai vaihtamaan vapaasti, koska ne olivat tilattomia. Kun palveluita lisättiin työn edetessä, alkoi arkkitehtuurin vaativuus olemaan huomattava. Viestintään ei enää riittänyt pelkkä REST-rajapinta, joka alun perin rakennettiin käyttöliittymäkerroksen ja taustajärjestelmän erottamiseen.

Useampi palvelu yhdistettiin viestintäjonoon, jonka kautta viestintä helpottui palveluiden välillä. Taustajärjestelmät vastaanottivat HTTP-pyyntöjä, joita ne lähettivät viestijonoon toimintona muille taustajärjestelmille käsiteltäväksi. Jokaisella palvelulla on oma REST-rajapintansa, jota vasten voidaan lähettää HTTP-pyyntöjä. Automaattisella ohjelmistokehitysputkella ohjelmisto oli testattavana jokaisen muutoksen jälkeen, jos se suoriutui yksikkö- ja käyttöliittymätesteistä onnistuneesti. Käyttöönotto onnistui automaattisesti, mikä olisi ollut hankalaa ja työlästä, koska palveluita oli monta.

Työ oli monipuolinen ja haastava siinä mielessä, että suurimman osan ajasta ei kahdeksan tuntia pitkä työpäivä riittänyt vaan tutkimustyötä piti tehdä vapaa-aikanakin useammasta lähteestä. Työn aikana sain paljon tietoa, miten eristämisteknologiat toimivat ja kuinka ne ovat kehittyneet ajan myötä, miten mikropalvelut toimivat ja mitä niiden suunnittelussa tulee ottaa huomioon, kun viestintäkerrosta rakennetaan.

Lähteet

[1] J. Gleick, The Information: A History, a Theory, a Flood, Fourth Estate, 2011.

[2] M. Feathers., Working Effectively with Legacy Code, Prentice Hall, 2004.

[3] D. Bernstein, Beyond Legacy Code: Nine Practices to Extend the Life (and Value) of Your Software, Pragmatic Bookshelf, 2015.

[4] C. Richardson., Microservice Patterns, Manning Publications Co., 2017

[5] R. Stephens., Beginning software engineering, Wrox, 2015

[6] K. Ivanov., Containerization with LXC, Packt Publishing, 2017

[7] J. Ellingwood., The Docker Ecosystem: An Overview of Containerzation, DigitalOcean, 2015; Saatavilla https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-overview-of-containerization. Viitattu 27/9/2017

[8] C. Torre, Containerized Docker Application Lifecycle with Microsoft Platform and Tools, Microsoft Press, 2017

[9] J. Turnbull., The Docker Book: Containerization is the new virtualization, James Turnbull, 2014

[10] J. Petazzonni, From dotCloud to Docker, 2017; Saatavilla https://jpetazzo.github.io/2017/02/24/from-dotcloud-to-docker/ . Viitattu 30/9/2017

[11] Kirjoittaja tuntematon, Docker vs LXC, 2017; Saatavilla https://www.upguard.com/articles/docker-vs-lxc Viitattu 4/11/2017

[12] Docker Inc, Docker frequently asked questions, 2017; Saatavilla https://docs.docker.com/engine/faq/ Viitattu 1/10/2017

[13] Scott M. Fulton, Finally, Linux Container Could Run On Windows with Docker's LinuxKit, 2017; Saatavilla https://thenewstack.io/finally-linux-containers-really-will-run-windows-linuxkit/ Viitattu 1/10/2017

[14] P-H. Kamp, Jails – High value but shitty Virtualization, 2016; Saatavilla http://phk.freebsd.dk/sagas/jails.html Viitattu 22/10/2017

[15] M. Riondato, Jails; Saatavilla https://www.freebsd.org/doc/handbook/jails.html Viitattu 22/10/2017

[16] W. Shields, Day 14 – FreeBSD Jails, SysAdvent, 2010; Saatavilla http://sysadvent.blogspot.fi/2010/12/day-14-freebsd-jails.html Viitattu 22/10/2017

[17] Oracle Co., Introduction to Solaris Zones, 2010; Saatavilla https://docs.oracle.com/cd/E19044-01/sol.containers/817-1592/zones.intro-1/index.html Viitattu 4/11/2017

[18] D. Drewanz, L. Grimmer., The Role of Oracle Solaris Zones and Linux Containers in a Virtualization Strategy, 2012; Saatavilla http://www.oracle.com/technetwork/articles/servers-storage-admin/zones-containers-virtualization-1880908.html Viitattu 4/11/2017

[19] Docker Inc, Overview of Docker Hub; Saatavilla https://docs.docker.com/docker-hub/ Viitattu 4/11/2017

[20] K. Sirkesalo, Slack keskustelu 20.10.2017, suullinen tiedonanto, 2017