Haaga-Helia
ammattikorkeakoulu Oy

Mikropalveluarkkitehtuuri hajautetun tietojärjestelmän toteutuksessa

Mika Ropponen

Opinnäytetyö
Tietojenkäsittely
2017

Tiivistelmä

Haaga-Helia ammattikorkeakoulu Oy

Tekijä
Mika Ropponen

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Raportin/Opinnäytetyön nimi
Mikropalveluarkkitehtuuri hajautetun tietojärjestelmän toteutuksessa

Sivu- ja liitesivumäärä
49

Opinnäytetyö selvittää mikropalveluarkkitehtuurin hyötyjä ja haasteita hajautetun tietojärjestelmän kehitystyössä. Teoriaosa esittelee yleisesti mikropalveluarkkitehtuuria sekä organisaation että kehitystyön näkökulmasta. Teoria pyrkii nostamaan esiin tärkeimpiä näkökulmia, mitä mikropalvelupohjaisen verkkosovelluksen suunnittelussa ja toteutuksessa tulisi ottaa huomioon. Lisäksi teoriassa käsitellään kehitys- ja tuotantoympäristöjen vaatimuksia mikropalvelupohjaiselle järjestelmäkehitykselle.

Mikropalvelujen suunnittelussa perehdytään sovelluksen modularisointiin, sisäiseen arkkitehtuuriin, tietoturvaan, vikasietoisuuteen ja erityisesti keskitettyyn hallintaan. Hallintamenetelmiin kuuluvat mikropalvelujen keskitetty konfigurointi, havaitseminen ja viestiliikenteen ohjaus. Mikropalvelujen välisen tiedonvaihdon osalta perehdytään sisäisiin ja ulkoisiin rajapintoihin, tietoliikenneprotokolliin ja viestiteknologioihin.

Mikropalvelujen ominaisuuksia tutkitaan käytännössä opinnäytetyön toteutusosassa, jossa kehitetään järjestelmäprosessi lääketietojen hakemiseksi suojatusta tietokantaresurssista. Toteutuksessa sovelletaan erityisesti Java-pohjaisia avoimen lähdekoodin ohjelmakirjastoja, joilla pyritään ratkomaan keskeisimmät hajautetun järjestelmän ongelmat. Järjestelmäprosessin kehitystyön tavoitteena on ennen kaikkea automatisoida tietojärjestelmän hallintamekanismit, jotta varsinaisten sovellusominaisuuksien kehitystyö olisi vapautettu hajautetun järjestelmän kompleksisesta kokonaistoiminnasta.

Selvitystyön ja käytännön toteutuksen tuloksena syntyy muutoksille joustava, vikasietoinen järjestelmä, jonka itsenäisiä mikropalveluja voidaan kehittää ilman vaikutusta muihin järjestelmän komponentteihin. Tulokset kuvaavat, miten hyvin mikropalvelupohjaisen järjestelmän hallintaominaisuudet ovat toteutettavissa ohjelmallisin menetelmin. Sovelletun teorian ja kehitystyön tuloksia käsitellään työn lopussa kehitystyön jälkiarvioinnin ja pohdinnan yhteydessä.

Asiasanat
Hajautetut järjestelmät, ohjelmistoarkkitehtuuri, ohjelmistokehitys, verkko-ohjelmointi

Sisällys

1 Johdanto

Monet internet-perustaisen liiketoimintamallin yritykset ovat hyödyntäneet mikropalveluarkkitehtuuria jo vuosia. Yksi kansainvälisesti tunnetuimmista verkkokirjakaupoista, Amazon.com, esitteli ensimmäisten joukossa uuden tapansa kehittää ohjelmistoja pilvipalvelualustalleen mikropalveluihin jaettuina. Samoin suurimpiin tilausvideopalveluihin lukeutuva Netflix toteutti avoimen lähdekoodin projekteina keskeisimmät mikropalveluarkkitehtuurin ominaisuudet kansainvälisen videojakelunsa tueksi. Mikropalvelujen käyttö rajoittui vuosia vain yrityksiin, jotka olivat pilvipalvelujen hyödyntämisen edelläkävijöitä. Laajemmin ohjelmistotuotannossa mikropalvelut ovat edelleen uusi teknologia, jonka hyötyjä liiketoiminnalle pohditaan tarkoin.

Hajautetuille tietojärjestelmille on tyypillistä, että yksittäisen tiedonhallintatapahtuman tai kokonaisen liiketoimintaprosessin suorittamiseen osallistuu useita palveluja, eli järjestelmäkomponentteja, jotka viestivät keskenään verkon yli. Palvelut voivat olla maantieteellisesti hajallaan eri palvelinkeskuksissa. Järjestelmän palvelut on usein toteutettu eri teknologioilla ja niitä suoritetaan erilaisilla palvelinalustoilla. Verkkopalveluohjelmistot ovat kehittyneet hajautettujen järjestelmien suuntaan, koska se mahdollistaa muiden organisaatioiden kehittämien ja ylläpitämien palvelujen hyödyntämisen osana järjestelmää. Hajautettu järjestelmä voidaan rakentaa pienemmistä itsenäisistä komponenteista, joiden hallinta on ketterää ja kustannustehokasta. Järjestelmän arkkitehtuuri voi kuitenkin kehittyä kompleksiseksi, minkä hallintaan mikropalvelut voivat tarjota tehokkaita ohjelmallisia ratkaisuja. Mikropalvelupohjaisten järjestelmien hallintaan on kehitetty myös valmiita pilvipalvelutuotteita. Valmiit kaupalliset tuotteet eivät aina kuitenkaan vastaa järjestelmäkehityksen synnyttämiin uusiin tarpeisiin. Lisäksi ne kasvattavat järjestelmäkehityksen kustannuksia. Ohjelmallisilla ratkaisuilla voidaan toteuttaa kustannustehokkaasti dynaaminen hajautetun järjestelmän toimintalogiikka pitkän tähtäimen järjestelmäkehitykseen. Hajautetun järjestelmän hallintamekanismien toteuttaminen ohjelmallisesti tuottaa ympäristöriippumattoman järjestelmän, jonka komponentit voidaan helposti siirtää palvelimelta toiselle. Ohjelmallisella hallintaratkaisulla järjestelmän toimintalogiikkaa voidaan kehittää kokonaisvaltaisesti ilman tarvetta palvelinteknologioiden syvempään hallintaan tai muiden erikoisasiantuntijoiden käyttöön.

Tässä opinnäytetyössä selvitetään mikropalveluarkkitehtuurin hyötyjä ja haasteita hajautetun tietojärjestelmän kehitystyössä. Teoria esittelee mikropalveluarkkitehtuuria ja sen suunnittelun näkökulmia, joita tarvitaan kestävän mikropalveluperustaisen järjestelmän toteuttamiseen. Koska teknologinen riippumattomuus on keskeisimpiä mikropalvelujen ominaisuuksia, käytännön toteutukseen valittuja teknologioita käsitellään teoriaosassa

arkkitehtuurin kokonaistoiminnan vaatimusten näkökulmasta teknisiin yksityiskohtiin menemättä. Toteutusosassa perehdytään tarkemmin Java-pohjaisiin avoimen lähdekoodin ohjelmistoteknologioihin, jotka on suunniteltu mikropalvelujen kehittämiseen. Toteutettava järjestelmäprosessi selvittää, miten pitkälle mikropalvelupohjaisen tietojärjestelmän hallintamekanismit voidaan tuottaa ohjelmallisesti ilman palvelinteknologioihin perustuvia ratkaisuja. Teoriaa soveltavan järjestelmäkehityksen tavoitteena on toteuttaa vakaan mikropalvelujärjestelmän ydintoiminnot, joita voidaan käyttää mallina muissa mikropalveluarkkitehtuuriin perustuvissa ohjelmistokehitysprojekteissa. Projekti suunnittelee ja toteuttaa järjestelmäprosessin, jossa mikropalvelun rajapintaan suoritetulla kyselyllä haetaan tietoja hajautetun järjestelmän suojatusta lääketietokannasta. Projekti selvittää, miten hakuprosessin vaiheet, kuten saatavilla olevien palvelujen havaitseminen, viestiliikenteen ohjaus, käyttöoikeuksien hallinta ja virhetilanteiden hallinta toimivat hajautetuissa mikropalveluissa.

Mikropalvelupohjaisten järjestelmien kehittämiseen ja ohjelmalliseen hallintaan on saatavilla avoimen lähdekoodin kirjastoja. Java-pohjaisiin ratkaisuihin vakiintuneen Spring-ohjelmistokehyksen alle on perustettu runsaasti projekteja pilvipalvelinympäristöön suunniteltujen mikropalvelusovellusten toteuttamiseen. Kirjastoissa on runsaasti erilaisia mikropalvelujen hallintaan suunniteltuja työkaluja, joiden ominaisuudet ovat ketterästi muunneltavissa vaatimusten mukaisiksi. Spring-kirjastot ovat hyvin testattuja ja niitä sovelletaan myös kaupallisessa ohjelmistotuotannossa. Tämän opinnäytetyön ohjelmistoprojekti keskittyy erityisesti Spring Cloud -kehysosan soveltamiseen mikropalvelujen hallintamekanismien toteuttamiseksi sekä mikropalvelujen välisen tiedonvaihdon toteuttamiseksi rajapintojen kautta. Näiden avulla saavutetaan vakaa, pitkälle automatisoitu järjestelmä, jossa varsinaisen käyttösovelluksen kehitystyö on vapautettu lähes kokonaan kompleksisesta ohjelmistoarkkitehtuurista sekä teknisestä infrastruktuurista. Tämän jälkeen ohjelmistokehittäjät voivat keskittyä varsinaisen käyttösovelluksen ominaisuuksien kehittämiseen. Hallintajärjestelmän kautta mikropalvelut – ja ohjelmistokehittäjät – saavat helposti käyttöönsä kaikkien järjestelmäkomponenttien tarjoamat palvelut tietämättä, miten ne on teknisesti toteutettu, ja missä ne fyysisesti sijaitsevat. Mikropalvelut keskustelevat toisilleen palvelujen nimillä. Kompleksinen tiedon prosessoinnin järjestelmä on piilotettu varsinaiselta sovelluslogiikalta. Sen toteuttaminen vaatii tarkkaa konfigurointi- ja ohjelmointityötä järjestelmän mikropalveluihin, mutta se tuottaa älykkään ja vakaan alustan hajautetun järjestelmän kehitystyölle.

Omassa työssäni ohjelmistokehittäjänä olen ollut mukana projekteissa, joissa mikropalveluarkkitehtuuria on sovellettu kehittämällä järjestelmää itsenäisinä komponentteina. Menetelmän myötä muutokset järjestelmän ominaisuuksiin ovat olleet nopeasti toteutettavissa

ja testattavissa. Järjestelmän kasvaessa on kuitenkin huomattu, että palvelujen välinen kokonaistoiminta, virhetilanteiden leviäminen sekä viestiliikenteen ohjaus käyvät hyvin työläiksi ymmärtää ja hallita. Hajautetun järjestelmän suunnittelussa on myös tärkeää selvittää sopivimmat menetelmät vahvan keskitetyn hallinnan, viestiliikenteen ohjauksen ja mikropalvelujen vikasietoisuuden toteuttamiseksi. Selvitystyö pyrkii hälventämään mielikuvaa, että mikropalveluarkkitehtuuri kasvattaisi merkittävästi kehitystyön monimutkaisuutta ja kustannuksia. Kun mikropalvelupohjaiselle projektille toteutetaan vahva arkkitehtuurinen ja tekninen perusta, varsinaisten arvoa tuottavien sovellusominaisuuksien kehittäminen on vaivatonta ja kustannuksia säästävää. Lisäksi järjestelmän kompleksisuus muuttuu kehittäjille näkymättömäksi. Selvitystyö on suunnattu erityisesti ohjelmistoalan asiantuntijoille ja opiskelijoille, jotka pohtivat mikropalveluarkkitehtuurin hyötyjä ja haasteita ohjelmistokehityksessä. Suuret hajautetut järjestelmät vaativat järeitä hallintamenetelmiä, mutta myös kevyitä ratkaisuja on hyvin saatavilla pienien ja keskisuurien järjestelmien hallintaan.

Toteutusosan luvut seuraavat tiedonhakuprosessin kehitystä. Prosessin vaiheita kuvataan yleisesti lukujen alkupuolella, jota seuraa tarkempi tekninen kuvaus käytetyistä menetelmistä. Seuraavalla sivulla oleva käsiteluettelo selittää tekstissä esiintyvien käsitteiden merkityksiä opinnäytetyön aihepiirin asiayhteydessä.

Käsiteluettelo:

ACID	Periaate, jonka avulla turvataan järjestelmän tietojen eheys kaikissa tilanteissa. Atomicity = Atomisuus, Consistency = Eheys, Isolation = Eristyneisyys, Durability = Pysyvyys.
Integrointi	Liittäminen osaksi järjestelmää.
Konvertointi	Toiseen muotoon muuntaminen.
Löyhä kytkentä	Järjestelmämalli, jossa komponentit hyödyntävät toisiaan tuntematta toistensa määrityksiä.
Migraatio	Tietojen siirto ohjelman eri versioiden välillä, esimerkiksi otettaessa käyttöön ohjelman uudempi versio.
Modularisointi	Tekeminen moduulirakenteiseksi, yksiköittäminen.
NoSQL	Tietokanta, joka poikkeaa perinteisestä relaatiomallista eikä noudata kiinteää taulukkoskeemaa. Soveltuu nopeisiin transaktioihin ja dokumenttien hallintaan.
Propagointi	Tiedon välitys tai levitys järjestelmässä.
Replikointi	Automatisoitu kopioituminen moneksi instanssiksi.
Sarjallistaminen	Prosessi, joka muuntaa tietorakenteen muotoon, jossa se on siirrettävissä verkon yli tai tietokoneen muistiin.
Skaalaus	Horisontaalinen skaalaus on saatavuuden kasvattamista kopioimalla palvelua moneksi rinnakkaisinstanssiksi. Vertikaalinen skaalaus on palvelun suoritustehon kasvattamista laitteiston ominaisuuksilla.
Transaktio	Tietokantajärjestelmässä suoritettava tiedon prosessointi, joka usein alkaa tietokantaan kohdistetusta kyselystä.

2 Mitä mikropalvelut ovat?

Modernin yhteiskunnan kaikki osa-alueet yhdistyvät internetin kautta. Paikallisilla markkinoilla toimineiden yritysten asiakaskunta ja kilpailu ovat laajentuneet globaaleiksi. Sovellukset, joissa on yksi kiinteä koodikanta keskustelemassa yhteen tietokantaan, ovat nopeasti jäämässä menneisyyteen. Nykyisin sovellukset rakentuvat useista hajautetuista palveluista ja tietokannoista, jotka viestivät internetin tai muun verkon yli. Sovellusten kompleksisuus on kasvanut merkittävästi, jolloin ohjelmistokehittäjien on omaksuttava uusi tapa ajatella sovelluskehitystä. Ohjelmistotuotannossa asiakkaat vaativat uusien ominaisuuksien ja versioiden nopeaa, irrallista toimitusta ilman koko tuotteen valmistumisen odottelua. Maailmanlaajuisessa käytössä sovelluksen käyttökuormaa ja transaktioiden määrää on mahdoton ennustaa. Siksi sovelluksen on pystyttävä dynaamisesti skaalautumaan useille palvelimille saatavuuden ja suorituskyvyn varmistamiseksi. Kun yksi kiinteä sovelluskokonaisuus puretaan pienemmiksi itsenäisiksi palveluiksi, voidaan rakentaa järjestelmää, joka on joustava palvelukohtaisille muutoksille. Järjestelmän virhetilanteet eivät kaada enää koko järjestelmään, vaan ne voidaan eristää yhteen palveluun. Yksittäisiä palveluja, eli järjestelmäkomponentteja, voidaan skaalata itsenäisesti useille palvelimille käyttökuorman kasvaessa. (Carnell 2017, 12-13.)

Mikropalvelut ovat vastakohta monoliittiselle sovellukselle. Monoliittinen sovellus voidaan julkaista ja ottaa käyttöön vain yhtenä suurena kokonaisuutena. Sitä vastoin, mikropalveluarkkitehtuurissa on kysymys modularisoinnista, suurien ohjelmistojen jakamisesta pienempiin itsenäisesti toimiviin yksiköihin, jotka yhdessä muodostavat organisaation toiminnalle merkityksellisen sovelluskokonaisuuden. Mikropalvelut ovat teknisesti riippumattomia muista mikropalveluista. Ne voidaan toteuttaa eri alustoilla sekä teknologioilla, jotka parhaiten tukevat kyseisen mikropalvelun suorittamaa yksilöllistä tehtävää. Mikropalvelut ovat itseään ylläpitäviä prosesseja pääsääntöisesti virtuaalipalvelinalustoilla, joilla on tehtävänsä suorittamiseen omat tukipalvelunsa ja tietovarastonsa. Mikropalvelut käyttävät keskinäiseen viestintään tiedonsiirtoprotokollia ja viestiteknologioita, jotka tukevat hajautettujen järjestelmien löyhää kytkentää (engl. loose coupling). (Eberhard 2017, 3-4.)

Tunnettuja mikropalveluarkkitehtuurin pioneereja ovat muun muassa verkkopalvelut Amazon, Netflix, The Guardian, UK Government Digital Service, RealEstate.com.au, Forward and CompareTheMarket.com. Lisäksi maailmalla toimii lukuisia organisaatioita, joiden kehitysmallit ovat luokiteltavissa mikropalveluiksi, vaikka ne eivät käytä tätä nimeä arkkitehtuuristaan. Näissä tapauksissa arkkitehtuuria usein nimitetään SOA:ksi (service-oriented architecture), jonka erot mikropalveluarkkitehtuuriin verrattuna ovat kiistanalaisia.

Tällä hetkellä ohjelmistoarkkitehtuurit näyttävät kehittyvän mikropalvelujen suuntaan. (Fowler 2014.)

2.1 Mikropalvelujen hyödyt

Mikropalvelut tarjoavat vahvan konseptin ohjelmistojen modularisointiin. Tiedämme kokemuksesta, että monoliittiset sovellukset kasvavat kehitystyön myötä hallitsemattomiksi ja niiden arkkitehtuuri hajoaa. Tällöin ohjelmiston sisäisten komponenttien väliset riippuvuudet tekevät kehitys- ja muutostyöstä haasteellista. Pienikin muutos yksittäiseen ohjelmaluokkaan tai funktioon voi aiheuttaa odottamattoman virhetilanteen järjestelmän toisessa osassa. Sitä vastoin itsenäisesti toimivien mikropalvelujen välille ei juurikaan pääse syntymään odottamattomia riippuvuuksia, sillä ne kommunikoivat keskenään eksplisiittisesti määriteltyjen rajapintojen kautta. (Eberhard 2017, 5.)

Mikropalveluarkkitehtuuri mahdollistaa vapauden valita palvelujen toteuttamiseen erilaisia teknologioita, jotka parhaiten soveltuvat tilanteeseen ja palvelun suorittamaan tehtävään. Mikropalvelujen viestintään käyttämä infrastruktuuri asettaa ainoat tekniset vaatimukset yksittäiselle mikropalvelulle (Eberhard 2017, 21). Mikropalvelun suurimpia etuja on myös sen helppo korvattavuus toisella, mikäli uusi palvelu toteuttaa korvattavan palvelun rajapinnan. Palvelun uusi versio voi perustua täysin eri koodikantaan ja teknologiaan, mikä on ollut vanhoissa järjestelmissä mahdotonta tai hyvin haasteellista toteuttaa. Uusia teknologioita voidaan helposti kokeilla itsenäisesti toimivissa mikropalveluissa, kun kokeilua varten valitaan matalimman riskin järjestelmäkomponentit. Monet organisaatiot ovat hyödyntäneet tätä mahdollisuutta ja ottaneet nopeasti käyttöön uusia innovatiivisia teknologioita olemassa olevaan järjestelmään, mikä ei ollut aiemmin mahdollista toteuttaa.
(Eberhard 2017, 5; Newman 2015, 4.)

Vakaus on eräs mikropalvelujen vahvuuksista. Itsenäisinä komponentteina niille on ollut helppo kehittää vahvoja menetelmiä vikasietoisuuden kasvattamiseksi. Mikropalveluarkkitehtuurissa virhetilanteet on helppo eristää yhteen palveluun. Virheiden leviäminen saadaan tehokkaasti estettyä, ja muut mikropalvelut voivat jatkaa toimintaansa ilman havaittavia katkoksia. Palvelujen vakautta ja saatavuutta voidaan kasvattaa myös skaalaamalla. Mikäli tietyt palvelut vaativat käyttökuormansa takia tehokkaamman laiteympäristön, ne on helppo siirtää yksittäin suuremman suoritustehon palvelimelle. Tätä sanotaan vertikaaliseksi skaalaukseksi. Sen lisäksi, mikropalvelujen saatavuutta voidaan kasvattaa dynaamisesti horisontaalin skaalauksen avulla, jolloin palvelusta otetaan käyttöön rinnakkaisinstansseja käyttökuorman kasvaessa. Vastaavasti kuorman pienentyessä tarpeettomia instansseja poistetaan automaattisesti käytöstä. Vahva dynaamisuus ja järjestelmän mo-

nitorointi toteutetaan erilaisilla keskitetyn hallinnan menetelmillä. Mikropalveluarkkitehtuuri määrittää säännöt ja standardit, joiden avulla hajautetun järjestelmän keskitetty hallinta on mahdollista toteuttaa (Eberhard 2017, 22). Keskitetty hallinta on tärkeä osa mikropalveluarkkitehtuuria, koska itsenäisiä mikropalveluja voidaan hyödyntää tehokkaasti erilaisiin käyttötarkoituksiin. Verkkopalvelut mahdollistavat rajattomat mahdollisuudet, miten palveluja voidaan hyödyntää yhdessä. (Newman 2015, 5-7.)

Vanhojen järjestelmien jatkokehitykselle mikropalvelut tarjoavat täysin uudenlaisen ratkaisun. Vanhat järjestelmät voidaan laajentaa teknologiariippumattomiin mikropalveluihin. Tällöin säästytään suurilta muutoksilta vaikeasti ymmärrettävään, vanhaan koodikantaan. Järjestelmiä ei myöskään tarvitse kokonaan korvata uusilla ja modernien teknologioiden tarjoamat hyödyt saadaan valjastettua vanhojen järjestelmien kehitykseen.
(Eberhard 2017, 6.)

2.2 Mikropalvelujen haasteet

Mikropalveluarkkitehtuurille on ominaista hyvin korkea teknologinen monimutkaisuus. Eri palveluiden vaatimuksiin on sovellettu eri teknologioita, joista kehitystiimit ovat voineet päättää itsenäisesti. Ongelmatilanteissa kehitystiimit eivät välttämättä pysty auttamaan toisiaan osaamisen puuttuessa toisen tiimin käyttämästä ohjelmointikielestä tai muusta teknologiasta. (Eberhard 2017, 22.)

Mikropalveluarkkitehtuuri asettaa omat haasteensa ohjelmistokehitystyölle. Palvelujen välisiä suhteita voi olla vaikeaa ymmärtää. Kehittäjä identifioi mikropalvelut verkko-osoitteiden perusteella. Tästä johtuen ei ole itsestään selvää, mikä mikropalvelu kutsuu mitäkin palvelua. Kehittäjän on tunnettava mikropalvelun kutsumien verkko-osoitteiden päissä toimivat palvelut tai ainakin niiden rajapinnat. Hajautetuissa järjestelmissä (engl. distributed systems) palvelujen välinen tiedonvaihto on täysin riippuvainen verkon toimivuudesta. Näin ollen verkon ongelmat voivat hidastaa tai estää täysin mikropalvelujen välisen viestinnän. (Eberhard 2017, 8.)

Häiriöt mikropalvelujen välisessä viestinnässä ovat arkkitehtuurin suurimpia haasteita. Järjestelmän käytettävyys ei saa keskeytyä edes verkkohäiriöiden aikana. Mikropalvelujen vikasietoisuuden parantamiseksi on olemassa teknologioita, mutta ne eivät yksin ratkaise koko ongelmaa. Palvelujen määrityksissä on otettava huomioon, miten mikropalvelu toimii verkkoyhteyksien katketessa. Usein välimuistista palautettu vanha tieto on palvelun toiminnan kannalta riittävä eikä kehittyneempiä menetelmiä tarvita virhetilanteen hoitamiseksi. (Eberhard 2017, 21.)

2.3 Käyttötapaukset

Yksi tavallisimmista skenaarioista mikropalvelujen hyödyntämiseksi on vanhan verkkopalvelun modernisointi. Vanhan järjestelmän migraatio täysin uuteen on kallis, aikaa vievä ja riskialtis prosessi. Mikropalvelujen avulla vanha järjestelmä voidaan osittain hyödyntää ja laajentaa uusiin palveluihin ja teknologioihin. Verkkokauppa on tyypillinen palvelu, joka tarjoaa asiakkaan käyttöön lukuisia eri toimintoja käyttäjän rekisteröinnistä tilauksen käsittelyyn. Lisäksi se toimii yhdessä muiden järjestelmien, kuten kirjanpidon ja logistiikan kanssa. Yhdessä monoliittisessa koodikannassa tällainen järjestelmä kasvaa ajan mittaan mahdottomaksi kehittää, ylläpitää ja testata. Sitä vastoin, erillisinä mikropalveluina nämä ominaisuudet voidaan toteuttaa itsenäisten kehitystiimien toimesta ilman järjestelmän sisäisiä konflikteja. (Eberhard 2017, 11-19.)

Hajautetuissa viestijärjestelmissä on tyypillistä, että eri teknologioilla toteutetut laitteet ja järjestelmät lähettävät eri muotoista, prosessoimatonta tietoa. Tämä tieto on kerättävä hajautetuista lähteistä ja tallennettava jonnekin jatkoprosessointia ja analysointia varten. Liikenteenvalvontajärjestelmät mallintavat hyvin tällaista maantieteellisesti laajalle levittäytyvää järjestelmää, joka kerää signaaleja erilaisista laitteista ilman yhteistä viestintäprotokollaa. Erilliset mikropalvelut voidaan konfiguroida kunkin tilaa havainnoivan sensorin teknisten vaatimusten mukaisesti. Kerätty raakadata konvertoidaan mikropalveluissa yhteiseen muotoon ja välitetään edelleen prosessin seuraavan vaiheen palvelulle. Järjestelmään tulisi myös valita tietokantateknologioita, jotka optimaalisesti palvelevat kutakin tiedon prosessoinnin käyttötarkoitusta. Valvontajärjestelmissä uusin tieto pitää olla nopeasti saatavilla, mihin parhaiten soveltuvat modernit ei-relaatiotietokannat (NoSQL). Sitä vastoin perinteiset relaatiotietokannat tarjoavat paremmin ominaisuuksia tiedon syvällisempään analysointiin. Tällaisen järjestelmän eri prosesseilla on toisistaan poikkeavat teknologiapinot, joiden toteuttamiseen tarvitaan erilaisia asiantuntijoita. Mikropalvelujen kehitystiimit voivat koota sopivimmat asiantuntijat toteuttamaan yksilölliseen tehtävään adaptoituvia järjestelmäkomponentteja. (Eberhard 2017, 19-21.)

2.4 Mikropalvelut ja SOA

Mikropalveluarkkitehtuuri ja SOA (service-oriented architecture) ovat monilta ominaisuuksiltaan samankaltaisia. Molemmat pyrkivät modularisoimaan suuria järjestelmiä palveluiksi. SOA voi olla hyödyllinen näkökulma, kun vanhoja järjestelmiä siirretään mikropalveluihin. Se säilyttää vanhan järjestelmän monoliittisen rakenteen, mutta erottelee toiminnot omiksi palveluikseen rajapinnassa, joiden kautta laajentaminen mikropalveluihin on mah-

dollista. Molemmille arkkitehtuureille on myös ominaista, ettei kumpaakaan ole tarkasti määritelty. SOA-palvelut ovat yleensä mikropalveluja suurempia prosessiketjun loogisia osia. SOA ei kuitenkaan aseta vaatimuksia palvelun koolle. Mikropalvelussa koko on keskeinen määrittävä ominaisuus, joka määritetään sovelluksen sisäisen toiminnan perusteella. SOA-palvelulla ja mikropalvelulla on myös samoja ominaisuuksia, kuten

- toiminta itsenäisenä sovelluskomponenttina
- käytettävyys erillään muusta järjestelmästä
- saatavuus verkon kautta
- käyttötavan määrittävä rajapinta
- riippumattomuus teknologioista ja alustasta
- haku rekisteröityjen palvelujen hakemistosta

Organisaation laajuisen SOA:n kehitys ja hallinta ovat kuitenkin hyvin erilaisia verrattuna yhden sovellusarkkitehtuurin mikropalvelumalliin. Arkkitehtuurien eroja verrataan taulukossa 1. SOA:n palvelut toteuttavat usein organisaation sisäisiä prosesseja ja niitä käytetään hallintakäyttöliittymien eli portaalien kautta. Palveluja ja niiden rajapintoja kehitetään usein loppukäyttäjien uusien käyttötarpeiden ehdoilla, joten palvelun on tuettava myös vanhoja versioita. Rajapinnat kasvavat raskaiksi ja monimutkaisiksi. Lisäksi SOA:lle on ominaista palvelujen välisen viestinnän ja tietoturvan raskaat protokollapinot. Palvelut kommunikoivat keskenään integraatioalustan kautta, joka vastaa myös toiminnan kokonaisohjauksesta. Integraatiojärjestelmä on älykäs hallintajärjestelmä, joka sisältää mallit liiketoimintaprosesseista sekä koko liiketoimintalogiikan. Toiminnanohjaus- ja asiakashallintajärjestelmät ovat tyypillisiä SOA-järjestelmiä, joiden monimutkaisen arkkitehtuurin ominaisuuksia harvoin käytetään täydellä kapasiteetilla. Mikropalvelujen rajapinnat, protokollat ja integraatiokäytännöt ovat sitä vastoin kevyitä. Niiden viestintä ja toimintalogiikka on määritetty sisäisesti ilman tarvetta ulkopuolisille ohjausteknologioille. SOA kattaa koko organisaation järjestelmät sekä niiden välisen toiminnan. Sitä vastoin mikropalveluarkkitehtuuri on yhden järjestelmän tai sovellusprojektin arkkitehtuuri. (Eberhard 2017, 81-92.)

Taulukko 1. SOA:n ja mikropalvelujen erot (Eberhard 2017, 92)

	SOA	Mikropalvelut
Soveltamisala	Organisaation laajuinen arkkitehtuuri	Yhden sovellusprojektin arkkitehtuuri
Joustavuus	Saavutetaan palvelujen keskitetyllä hallinnalla	Saavutetaan palvelun nopealla kehityksellä ja käyttöönotolla
Organisaatio	Eri organisaatioyksiköt toteuttavat eri palveluja	Saman projektin kehitystiimit toteuttavat eri palveluja
Käyttöönotto	Useiden palvelujen monoliittinen käyttöönotto	Jokainen palvelu otetaan käyttöön itsenäisesti
Käyttöliittymä	Portaali, universaali käyttöliittymä kaikille palveluille	Palvelu voi sisältää käyttöliittymän

2.5 Jatkuva integraatio ja skaalaus

Mikropalveluarkkitehtuuri lyhentää merkittävästi sovelluksen toimitusaikoja tuotantoon. Palvelut voidaan ottaa käyttöön yksi kerrallaan. Mikropalvelut tukevat ketterien kehitysmenetelmien metodologiaa mahdollistaen suurien kehitystiimien jakamisen pieniksi, itsenäisiksi kehitystiimeiksi. Yksittäisten mikropalvelujen kehitystyöstä vastaavat tiimit voivat samanaikaisesti tuottaa sovelluksen ominaisuuksia ja integroida ne järjestelmään ilman aikaa kuluttavaa koordinointia muiden tiimien kanssa. (Eberhard 2017, 6-7.)

Mikropalveluarkkitehtuuri tukee jatkuvan toimituksen (engl. continuous delivery) tuotantoprosessia. Mikropalveluita voidaan skaalata, testata ja käyttöönottaa toisistaan riippumattomina. Kun jokainen mikropalvelu liitetään omaan tuotantoputkeen (engl. pipeline), on huomioitava, että ylläpidettäväksi tulee myös suuri määrä standardoituja tuotantoteknologioita (Eberhard 2017, 77). Yksittäisen mikropalvelun käyttöönottoon liittyy merkittävästi vähemmän riskejä kuin monoliittisen sovelluksen tapauksessa. Mikropalvelun eri versioita voidaan jopa käyttää rinnakkain sopivimman toteutustavan löytämiseksi. Jatkuva integraatio (engl. continuous integration), eli uusien komponenttien automatisoitu liittäminen osaksi tuotantojärjestelmää, on yksi tärkeimmistä syistä mikropalveluarkkitehtuurin esittelyyn uutena menetelmänä ohjelmistotuotannossa. (Eberhard 2017, 7-8.)

Yksittäinen mikropalvelu voidaan käyttöönottaa yhdelle tai skaalata usealle palvelimelle. Hajautetun mikropalvelun rinnakkaisinstanssit mahdollistavat käyttökuorman jakamisen usealle palvelimelle. Tällainen horisontaali skaalaus on kustannustehokasta verrattuna vertikaaliseen skaalaukseen, jossa resurssien määrä säilyy samana, mutta suoritustehoa kasvatetaan esimerkiksi paremman laitetekniikan avulla (Eberhard 2017, 148). Maailman-

laajuisessa järjestelmässä rinnakkaispalvelut voidaan näin ollen tuoda lähemmäksi käyttäjään suorituskyvyn parantamiseksi. Hajautetussa pilvipalvelinympäristössä ei ole enää merkitystä, mihin palvelinkeskuksiin järjestelmän komponentteja asennetaan. (Eberhard 2017, 61.)

2.6 Hyödyt organisaatiolle ja kehitystyölle

Mikropalvelujen teknologiariippumattomuus rohkaisee tuomaan uusia innovatiivisia ratkaisuja osaksi laajempaa järjestelmää. Mikropalveluarkkitehtuuri lisää merkittävästi tuotannossa toimivan järjestelmän näkyvyyttä sidosryhmille. Järjestelmän palveluita on helppo monitoroida ja kerätä liiketoimintaa ja sidosryhmiä kiinnostavaa reaaliaikaista tietoa, josta voidaan koostaa graafisia näkymiä ja tilastoja päätöksenteon tueksi (Nadareishvili, Mitra, McLarty & Amundsen 2016, 44). Järjestelmässä voidaan aina käyttää uusimpia teknologioita, mikä lisää kehittäjien motivaatiota ja kiinnostusta niin ohjelmistoprojekteja kuin työnantajaa kohtaan (Eberhard 2017, 62). Palvelut voidaan myös kehittää ja testata turvallisesti erillään muista järjestelmistä. Ongelmien ilmaantuessa mikropalvelu ja siihen liittyvä teknologiakokeilu voidaan helposti korvata tai jättää pois järjestelmästä. Helppo korvattavuus vähentää virheellisistä päätöksistä aiheutuvia kustannuksia. Tämä helpottaa merkittävästi teknologiavalintoihin liittyvää päätöksentekoa sekä vähentää potentiaalisia riskejä ja kustannuksia. (Eberhard 2017, 6-7.)

Muiden järjestelmien integroiminen on jatkuva vaatimus ohjelmistokehityksen asiakasprojekteissa. Järjestelmäkokonaisuuden osana on usein myös muiden yritysten ylläpitämiä laitteita ja ohjelmistoja. Hajautettujen järjestelmäkomponenttien helppo integroitavuus on yksi mikropalveluarkkitehtuurin perusominaisuuksista, mikä säästää merkittävästi kehitystyöhön tarvittavia resursseja. Integraatiokomponentit kuten myös muut mikropalvelut tai niiden yhdistelmät voidaan myydä erillisinä tuotteina. (Eberhard 2017, 21-22.)

Perintöjärjestelmien (engl. legacy systems) kehittäminen voi olla organisaatiolle hyvin haasteellista vanhentuneen ja vaikeasti ymmärrettävän koodikannan takia. Järjestelmän korvaaminen kokonaan uudella voi aiheuttaa myös liian suuret riskit toteutettavaksi. Vanhan järjestelmän jakaminen tai uudelleenkirjoittaminen mikropalveluiksi voi olla ongelmallista johtuen heikoista arkkitehtuurimäärityksistä, jotka selventäisivät komponenttien ja liiketoimintalogiikan välisistä suhteista (Eberhard 2017, 126). Jos vanha järjestelmä tukee kriittisiä liiketoimintaprosesseja, sen muuttaminen voi olla lähes mahdotonta. Laajentaminen mikropalveluihin, jotka tukevat komponenttien helppoa korvattavuutta, on kuitenkin varma investointi tulevaisuuteen. Vanhaan järjestelmään on tällöin kehitettävä rajapinta, joka mahdollistaa mikropalvelujen tiedonvaihdon vanhan järjestelmän kanssa. Mikropalve-

luita sovellettaessa kehittäjien ei myöskään tarvitse tuntea koko järjestelmää. Sopivat asiantuntijat voidaan koota yksittäisten mikropalvelujen ominaisuuksien perusteella kehitystyön toteuttamiseksi. (Eberhard 2017, 56-57.)

Uusien ohjelmistoversioiden julkaiseminen on tuotantojärjestelmissä jatkuvaa. On hyvin haasteellista määrittää, kuinka paljon virheiden korjauksia ja uusia ominaisuuksia pitäisi ottaa mukaan uuteen julkaisuversioon. Jokainen muutos kasvattaa merkittävästi riskiä, että se luo järjestelmässä uusia ongelmia. Mikropalveluihin perustuva suunnittelu ja kehitystyö rajaa muutosten vaikutuksia hyvin pienelle alueelle järjestelmän arkkitehtuurissa. Muutokset mikropalveluihin eivät lisää kustannuksia ja riskejä kuten muutokset laajan mittakaavan julkaisumalleissa. Ohjelmistojen kehityksen haasteena on aina niiden nopea vanheneminen. Arkkitehtuurin tulisi tukea järjestelmän komponenttien helppoa korvattavuutta sitä hetkeä varten, kun vanhaa sovellusta ei voida enää muuttaa vaatimuksia vastaavaksi. Ohjelmistojen kehittäminen mikropalveluina mahdollistaa pienien, riippumattomien muutosten toteuttamisen ja julkaisemisen usein, jolloin toiminnalliset viat saadaan nopeasti korjattua ja uudet ominaisuudet julkaistua tuotantoon ilman suuria riskejä. (Nadareishvili ym. 2016, 106-107.)

Aikanaan myös mikropalvelut vanhenevat, mutta ne voidaan helposti korvata tai modernisoida yksittäin uusimpien vaatimusten mukaisesti. Tämä tarkoittaa organisaatiolle kestävää pitkän tähtäimen järjestelmäkehitystä sekä korkeaa tuottavuutta. (Eberhard 2017, 57.)

3 Mikropalvelujen suunnittelu

Mikropalvelujen soveltaminen on paljon muutakin kuin pienien järjestelmäkomponenttien toteuttamista. Suunnittelutyön haasteena on se, että mikropalveluarkkitehtuurin määrittämiselle ei ole olemassa tiukkoja sääntöjä. Suunnittelussa on huomioitava hajautetun järjestelmän kaikki puolet, ja miten ne saadaan toimimaan yhdessä. Mikropalvelujärjestelmä käsittää organisaation kaikki osa-alueet, jotka liittyvät ohjelmistotuotantoon. Organisaation rakenne, palvelujen koordinointi, virhetilanteiden hallinta, tuotantokäytännöt, ja lukuisat muut näkökulmat, yhdistyvät toisiinsa mikropalvelupohjaisessa järjestelmässä. (Nadareishvili ym. 2016, 25-26.)

3.1 Kehitys- ja tuotantoympäristöt

Jokainen mikropalvelu versioidaan itsenäisesti versionhallintajärjestelmässä, jotta sovellus voidaan viedä itsenäisesti tuotantoon. Mikropalvelua varten on perustettava virtuaalipalvelinalusta, jolla sovellus voidaan suorittaa. Palvelinalustan lisäksi infrastruktuuriin voivat kuulua tietokanta- ja sovelluspalvelimet. Jokainen mikropalvelu tulisi olla kytkettynä automatisoituun ohjelmiston tuotantoprosessiin, joka huolehtii muutosten jälkeen sovelluksen kääntämisestä, testauksesta ja käyttöönotosta palvelimelle. (Eberhard 2017, 77.)

Jatkuvan integraation toiminnallisuudet pitävät huolen, että jokaisen sovellukseen toteutetun muutoksen jälkeen koko sovellus käännetään ja testataan uudelleen. Erityisesti, jos kääntäminen tai testaus epäonnistuu, kehitystiimi korjaa ongelman välittömästi. Jatkuva integraatio takaa, että sovellus säilyy aina toimivassa tilassa. Kehitystiimit, jotka soveltavat jatkuvan integraation käytäntöjä, pystyvät toimittamaan sovelluksensa virheettömämpinä sekä selvästi muita nopeammin. Jatkuva integraatio ei ole työkalu vaan joukko toimintaperiaatteita, joihin kehitystiimit sitoutuvat. Ominaisuuden valmistuttua sovellus on ensin käännettävä ja testattava paikallisesti. Mikäli ongelmia ei ilmene, muutokset voidaan siirtää versionhallintapalvelimella olevaan koodikantaan. Jatkuvan integraation ohjelmistot voivat lähtökohtaisesti prosessoida vain yhtä sovellusversiota kerrallaan, joten on varmistuttava, että CI-työkalut (engl. continuous integration tools) ovat vapaina käyttöön. Automaatio voidaan jättää kääntämään ja testaamaan versionhallinnan uutta sovellusversiota. Mikäli ongelmia ilmenee, korjaukset lähdekoodiin on suoritettava välittömästi. Kun sovellus läpäisee CI-prosessin, kehitystyössä voidaan siirtyä seuraavaan tehtävään. Sovellukseen liitettävät muutokset tulisi säilyttää pieninä, jotta ilmenevät ongelmat ovat nopeasti korjattavissa. (Humble & Farley 2011, 56-59.)

Teknologinen vapaus on yksi mikropalvelujen perusominaisuuksista. Kehitystiimien on valittava sopivimmat ohjelmointikielet ja teknologiat kunkin mikropalvelun suorittaman tehtävän perusteella. Myös saman teknologian käytöllä mikropalveluissa on paljon etuja. Tämä vähentää kompleksisuutta ja kehitystiimeillä on enemmän kokemusta tietyn teknologian soveltamisesta mikropalveluihin. Erityistilanteisiin voidaan joka tapauksessa valita muita teknologioita. (Eberhard 2017, 137.)

Hajautetun järjestelmän kasvaminen suureksi tuottaa usein odottamattomia riippuvuuksia komponenttien välille. Ilman arkkitehtuurin hallintaan suunniteltuja työkaluja komponenttien välisiä suhteita voi olla vaikea ymmärtää. Arkkitehtuurissa tulisi välttää kiertäviä riippuvuuksia mikropalvelujen välillä (engl. cyclic dependency). Tällaiset virhetilanteille altistavat riippuvuudet voidaan havaita graafisilla kehitystyökaluilla, joihin on määritettävissä arkkitehtuurille asetettuja sääntöjä. Tähän tarkoitukseen on apuohjelmistoja kuten Structure 101, Gephi ja jQAssistant. (Eberhard 2017, 104-107.)

Kontekstikartat (engl. context map) auttavat hahmottamaan, mihin tietomallin toiminnallisuuteen mikropalvelut on sidottu (engl. bound context). Tämä määrittää myös palvelujen välisen viestinnän sekä niiden keskinäiset suhteet. (Eberhard 2017, 108-109.)

3.2 Modularisointi ja tiedonvaihto

Mikropalvelut tulisi säilyttää pieninä, riippumattomina prosesseina. Järjestelmäkomponentteja ei pidä kuitenkaan perusteettomasti hajauttaa. Koska mikropalvelut viestivät verkon yli, palvelujen väliseen tiedon vaihtoon kuluu merkittävästi enemmän aikaa kuin kyselyn suorittamiseen saman prosessin sisällä. Verkkoviiveen lisäksi aikaa kuluu kyselyn parametrien ja vastauksen sarjallistamiseen (engl. serialization). Mitä pienemmiksi mikropalvelut on suunniteltu, sitä enemmän järjestelmässä on verkon yli tapahtuvaa viestintää. Tämä hidastaa samassa suhteessa järjestelmän kokonaistoimintaa. Kyselyn prosessointi verkon yli kuormittaa enemmän myös palvelimen prosessoria. Verkon ongelmat tai palvelujen kaatuminen voivat estää kyselyn suorittamisen kokonaan. Mikropalvelun on kuitenkin selvittävä järkevästi tilanteista, joissa kyselyn suorittaminen syystä tai toisesta ei onnistu. Mikropalvelujen käyttäytyminen virhetilanteissa tulisi olla selvillä jo määrittelyvaiheessa. (Eberhard 2017, 28-29.)

Mikropalveluarkkitehtuurissa on kriittistä, miten erilaiset virhetilanteet hoidetaan. Optimointia joudutaan suorittamaan sovelluksen lisäksi myös käyttöjärjestelmän tasolla. Palvelukyselyiden yhteydessä TCP/IP-verkkoyhteyksille on usein määritetty oletuksena useiden minuuttien aikakatkaisu. Mikropalvelun virhetilanne voi ruuhkauttaa tai jopa kaataa vas-

tausta odottavan järjestelmän, joka alkaa kuormittua virhetilanteessa aikakatkaisua odottavista kyselyistä. Siksi mikropalveluille tulisi konfiguroida lyhyet odotusajat, jotta epäonnistuneisiin kyselyihin voidaan reagoida nopeasti ja kohdistaa kysely uudestaan samaan tai rinnakkaiseen instanssiin. (Eberhard 2017, 62.)

REST (Representational State Transfer) on helposti käyttöönotettava protokolla mikropalvelujen välisen toiminnan toteuttamiseen verkon yli. Se identifioi verkossa sijaitsevat resurssit URI-osoitteina (Uniform Resource Identifier), jonka erikoistapaus URL (Uniform Resource Locator) on tunnettu erityisesti internet-sivustojen osoitteista. REST tarjoaa resurssien hallintaan joukon kiinteästi määritettyjä metodeja HTTP-siirtoprotokollaa käytettäessä (Hypertext Transfer Protocol). Tyypillisin formaatti REST-protokollan välittämälle tiedolle on JSON (JavaScript Object Notation), mutta kyselyä suorittava asiakassovellus voi pyytää HTTP-protokollan Accept-otsikkotiedoissa tietosisältöä myös muissa tunnetuissa tiedostomuodoissa, kuten XML- (Extensible Markup Language) tai HTML-muodoisssa (Hypertext Markup Language). HTTP- ja REST-protokollat toteuttavat lähtökohtaisesti tilatonta tiedonvaihtoa palvelimien välillä. Koska mitään tietoa aiemmista kyselyistä ei säilötä palvelimille, kyselyissä on oltava mukana kaikki sen käsittelemiseen tarvittava tieto. (Eberhard 2017, 175-176.)

Messaging Systems, eli viestijärjestelmät, ovat luotettavia hajautettujen järjestelmien viestintään. Viestit voidaan lähettää yhdelle tai usealle vastaanottajalle. Viestin lähetys kestää myös verkkohäiriöitä. Viestijärjestelmä säilöö viestit muistipuskuriin, jolloin ne ovat säilössä verkkoyhteyden katketessa myöhempää uudelleenlähetystä varten. Järjestelmä huolehtii viestien toimituksen lisäksi myös niiden onnistuneesta prosessoinnista. Mikäli viestin käsittelyssä tapahtuu virhe, viesti voidaan lähettää uudestaan lukuisia kertoja, kunnes prosessointi on onnistuneesti suoritettu loppuun tai viesti määritetään hylättäväksi. Viestijärjestelmässä lähettäjä siirtää viestin jonoon tuntematta viestin vastaanottajaa. Palvelimien väliset viestikanavat on määritelty eksplisiittisesti erillisessä rekisterissä. Viestijärjestelmät soveltuvat erityisesti tapahtumapohjaisiin arkkitehtuureihin (engl. event-driven architecture), joissa on tarve reagoida nopeasti tapahtumiin ja tilan muutoksiin. Viestijärjestelmän toteuttamiseen on saatavilla lukuisia eri ominaisuuksilla ja protokollilla viestiviä teknologioita, kuten JMS-koodikirjastoon (Java Messaging Service) perustuva ActiveMQ ja AMQP-standardiin (Advanced Message Queuing Protocol) perustuva RabbitMQ. Tällaisen tiedonvaihdon toteuttaminen vaatii useimmiten erillisten viestipalvelimien perustamista, mikä lisää merkittävästi järjestelmän infrastruktuurin monitoroinnin ja ylläpidon kuormaa. (Eberhard 2017, 180-183.)

Transaktioiden ACID-ominaisuudet, eli tiedon eheyden säilyttäminen kaikissa tilanteissa, on helppo toteuttaa mikropalvelun sisäisesti. Sitä vastoin monimutkaiset transaktiot läpi mikropalveluarkkitehtuurin vaativat kokonaisvaltaista koordinointia. Transaktion atomisuus (engl. atomicity) vaatii tiedonvaihdon onnistumista täydellisesti kaikkien siihen osallistuvien mikropalvelujen läpi aina tietokantaan asti. Virhetilanteessa jokaisen mikropalvelun tekemät muutokset on pystyttävä palauttamaan alkuperäiseen tilaansa. Tämä voi olla hyvin työlästä ja vaikeaa toteuttaa. Lähtökohtaisesti transaktion eheys (engl. consistency) voidaan taata vain, kun kaikki transaktion prosessoima tieto on saman mikropalvelun sisäistä. Verkon yli tapahtuva tiedonvaihto mikropalvelujen välillä on epäluotettavaa. Transaktion ei pitäisi kattaa useita mikropalveluja, mikäli tiedon eheyden säilyttäminen on kriittistä. Monen mikropalvelun transaktiot voivat olla myös viite liian hajautetusta järjestelmästä. (Eberhard 2017, 31-32.)

Mikäli mikropalveluarkkitehtuurin läpi suoritettu transaktio halutaan peruuttaa, järjestelmä voidaan palauttaa aikaisempaan tilaan kompensaatiotransaktion avulla. Verkkopalvelussa suoritettu matkavaraus voi kattaa useita järjestelmäpalveluita kuten hotellin, lentojen ja vuokra-auton varauksen. Nämä varaukset suoritetaan joko onnistuneesti yhdessä tai kaikki perutaan. Varaustapahtuma voidaan jakaa kolmeen mikropalveluun, koska jokainen suoritettava tehtävä on erilainen. Kyselyt hotellin, lentojen ja vuokra-auton saatavuudesta kohdistuvat eri järjestelmiin. Mikäli jokin varauksen kohteista ei ole varaushetkellä enää saatavilla, palvelut voivat viestiä toisilleen transaktion perumisesta tai suoritetun transaktion palauttamisesta (engl. rollback). Kompensaatio on siis tavallinen palvelukutsu. (Eberhard 2017, 32-33.)

Tietojärjestelmän kehityksessä on huolehdittava, ettei sen komponenttien välille muodostu epätoivottuja riippuvuuksia. Jotta kahden mikropalvelun välinen viestintä on mahdollinen, kehittäjän on määrätietoisesti rakennettava rajapinnat verkon yli tapahtuvaa viestintää varten. Tällainen yhteys ei synny vahingossa. Matkavarauksia tarjoava verkkopalvelu tarjoaa luonnollisesti myös hakupalvelun, joka auttaa käyttäjää löytämään meneillään olevaan varaukseen sopivia lisätuotteita. Hakupalvelun on saatava tietoja varauspalvelulta, mutta yhteys toiseen suuntaan ei ole mahdollinen. Tämä takaa, että hakupalvelua voidaan muuttaa ilman vaikutusta varausprosessiin. Epätoivottuja riippuvuuksia voidaan välttää myös arkkitehtuurinhallintatyökaluilla, joille annetaan malli suunnitelluista komponenttien välisistä suhteista. Tällainen apuohjelmisto havaitsee kehitystyön synnyttämät virheelliset riippuvuudet. Mikropalvelujen eristyneisyys pitää tietojärjestelmän arkkitehtuurin yleensä ehjänä ilman aputyökalujakin. (Eberhard 2017, 55-56.)

Mikropalvelut toteuttavat sekä järjestelmän sisäisiä että ulkoisia rajapintoja. Sisäisten rajapintojen kehittäminen on lähes yhtä riippumatonta kuin itse mikropalvelun kehitys. Sisäisten rajapintojen muutoksissa tarvitaan kehitystiimien välistä koordinointia, mikäli eri tiimit vastaavat keskenään kommunikoivien mikropalvelujen kehityksestä. Sitä vastoin ulkoisten rajapintojen ja versiokäytäntöjen suunnittelussa tarvitaan koordinointia ulkoisten käyttäjien kanssa. Ulkoiset rajapinnat mahdollistavat järjestelmän saatavuuden organisaation ulkopuolelta. Järjestelmä voi tarjota myös julkisen rajapinnan kaikille internetin käyttäjille. Tällöin aiempien rajapintaversioiden tukeminen laajalle ja mahdollisesti tuntemattomalle käyttäjäkunnalle on tärkeää. Ulkoisille rajapinnoille voidaan kuitenkin laatia sääntöjä, kuinka kauan tiettyä versiota tuetaan. Tämän jälkeen ulkoiset asiakassovellukset pakotetaan muutoksiin. Ulkoinen käyttäjä voi olla myös API (Application Programming Interface), ohjelmakirjasto, jonka kautta ulkoinen järjestelmä integroituu mikropalveluihin. (Eberhard 2017, 187-188.)

3.3 Sisäinen arkkitehtuuri

Mikropalvelujen yhteydessä puhutaan usein sovellusalakeskeisestä suunnittelusta (engl. domain-driven design). Mikropalvelut on jaettu konteksteihin, jotka edustavat sovelluksen erillisiä toimintoja (Eberhard 2017, 100). Yksinkertaistettuna tällä tarkoitetaan mikropalvelujen kehittämistä niin, että ne toteuttavat sovellusalan tietomallin ja liiketoimintalogiikan tarkoituksen mukaisia toimintoja sekä tukevat näiden muutoksia tulevaisuudessa. Toisin sanoen mikropalvelut toteuttavat yksiselitteisesti liiketoimintaprosessin tehtäviä. Tämän toteuttamiseksi liiketoimintalogiikka on jaettava osiin. Palvelut on nähtävä rakeisena kokonaisuutena, jonka jyviä mikropalvelut edustavat. Sovellus usein ratkaisee jonkinlaisen liiketoiminnan ongelman, jonka korkean tason kuvaus voidaan jakaa uudestaan ja uudestaan loogisiin osiin, kunnes päädytään matalan tason jakamattomiin toimintoihin. Nämä toimintalogiikan osaset edustavat potentiaalisia mikropalvelun tehtäviä.
(Carnell 2017, 38-39; Eberhard 2017, 44.)

Mikropalvelulla tulisi olla vahva sisäinen koheesio, eli yksi pysyvä, kiinteästi määritetty tehtävä. Mikäli mikropalvelun koheesio ei ole riittävän korkea, sen jakamista useiksi mikropalveluiksi tulisi harkita. Jakaminen takaa, että palvelut pysyvät pieninä sekä helposti ymmärrettävinä ja korvattavina. Mikropalvelujen kapseloinnilla tarkoitetaan sisäisen tietorakenteen piilottamista muilta mikropalveluilta. Palvelun käyttö tulee tapahtua ainoastaan rajapinnan kautta. Tällöin sisäistä rakennetta voidaan muuttaa ilman vaikutusta muuhun järjestelmään. (Eberhard 2017, 194.)

Helppoa korvattavuutta voidaan pitää mikropalvelun laadun mittarina. Tämä on ainoastaan mahdollista, kun mikropalvelun sisäinen kompleksisuus ja koko eivät kasva liian suuriksi (Eberhard 2017, 121). Skaalautuvat mikropalvelut tulisi aina säilyttää tilattomina (engl. stateless). Toisin sanoen, replikoitavien mikropalvelujen ei pidä säilöä sisäisesti käyttäjän istuntokohtaisia tietoja. Muutoin viestiliikenteen jakaminen rinnakkaisiin instansseihin johtaisi tilatiedon virheellisiin eroihin eri palvelimilla. Käyttäjän tila voidaan säilöä tietokantaan tai palvelun ulkopuoliseen muistivarastoon, johon kaikilla mikropalveluilla on pääsy. (Eberhard 2017, 149.)

3.4 Keskitetty hallinta

Mikropalveluarkkitehtuuriin yleensä toteutetaan keskitetty palvelujen konfigurointi, havaitseminen ja hallinta. Mikropalvelujen kuormittumista voidaan tasapainottaa replikoitujen palveluinstanssien avulla sekä automatisoidulla kuorman ohjauksella (engl. load balancing). (Eberhard 2017, 96.)

Mikropalvelupohjaisen järjestelmän konfigurointi voi olla vaativaa. Tyypillisesti mikropalveluja on yhdessä järjestelmässä runsaasti ja niille kaikille on määritettävä kelvolliset parametrit toimintojen käynnistämiseksi. Keskitetyllä konfiguraatiopalvelimella voidaan säilöä ja hallita mikropalvelujen suorituksenaikaisen konfiguraation ominaisuuksia, joiden kehitystyö on tuettu versionhallintajärjestelmällä (Rajesh 2016, 211). Tämän toteuttamiseen on API-kirjastoja kuten Spring Cloud Config ja Zookeeper. Nämä kirjastot on toteutettu Javalla ja ne ovat Apache-lisenssin alaista avointa lähdekoodia. Spring Cloud Config tukee myös konfigurointitietojen käyttämistä suoraan versionhallintapalvelimelta. Jaetun konfigurointipalvelun käyttöä on harkittava erikseen jokaisen mikropalvelun kohdalla ulkopuolisten muutosten rajoittamiseksi. Mikropalvelu voidaan konfiguroida myös paikallisesti omalla palvelinalustallaan, jolloin replikoitaessa palvelininstansseja rinnakkaispalveluilla on varmasti sama konfiguraatio. (Eberhard 2017, 139-140.)

Konfiguraatiotiedostoilla voidaan luoda profiileja, eli erilaisia konfiguraatiomäärityksiä eri käyttöympäristöjä varten. Mikropalvelu voi tarvita erilaiset suorituksenaikaiset asetukset esimerkiksi kehitys- ja tuotantoympäristöissä. Palvelun siirto toiselle palvelinalustalle voi vaatia myös ympäristökohtaisia muutoksia palvelun asetuksiin. Konfiguraatiotiedostoja voidaan käyttää paikallisesti tai ulkoisesta järjestelmästä. Paikallista tiedostoa on helppo muuttaa ja kehittää mikropalvelun mukana. Sitä vastoin, konfiguraation määrittäminen ulkoisissa tiedostojärjestelmissä on kallista ja joustamatonta. Tiedot eivät myöskään replikoidu palvelun mukana, mikä on mikropalvelun skaalautumista hankaloittava tekijä. Javapohjaisissa ratkaisuissa konfiguraation hallinta ulkoisissa tiedostojärjestelmissä on kuiten-

kin mahdollista JNDI:n (Java Naming and Directory Interface) kautta. Lisäksi Java-kirjastoissa on järjestelmänhallintaan suunniteltuja työkaluja, joilla voidaan toteuttaa myös palvelinympäristön konfiguraatioratkaisuja. Tällöin palvelukohtaiset määritykset saadaan myös replikoitua mikropalvelun mukana virtuaalipalvelimen skaalauksessa.
(Rajesh 2016, 211-212.)

Palvelujen havaitseminen (engl. service discovery) huolehtii, että mikropalvelut löytävät toisensa. Replikoidun palveluinstanssin verkko-osoitteet on piilotettava muilta mikropalveluilta niin, että on vain yksi piste, johon tietyn palvelun kyselyt kohdistetaan (Carnell 2017, 20). Yksinkertaisimmillaan tämä voidaan toteuttaa sovelluksen omalla konfiguraatiotiedostolla, joka määrittelee IP-osoitteet ja portit, joissa palvelut sijaitsevat. Tämä ei ole kuitenkaan kestävä ratkaisu tuotantojärjestelmille, joissa mikropalvelujen uusia instansseja otetaan jatkuvasti käyttöön kehitystyön ja skaalauksen myötä. Myös virhetilanteiden kaatamat instanssit on otettava huomioon. Palvelujen havaitsemisen pitää mukautua dynaamisesti järjestelmän muutoksiin. Tällaisen palvelun etuna on, että mikropalvelun kutsuja ei ole sidottu kiinteästi määritettyyn palveluinstanssiin vaan kyselyt voidaan ohjata kulloinkin parhaiten saatavilla olevaan palveluun. Palvelujen havaitseminen kokoaa rekisterin kaikista järjestelmän mikropalveluista, mitä voidaan hyödyntää myös palvelujen monitoroinnissa. Havaitsemispalvelua ei tarvita mikropalveluille, jotka kommunikoivat keskenään viestiteknologioilla kuten ActiveMQ. Tällaisilla palveluilla ei ole muuta väylää tiedonvaihtoon kuin kiinteästi määritetty kahdenvälinen viestikanava. (Eberhard 2017, 141.)

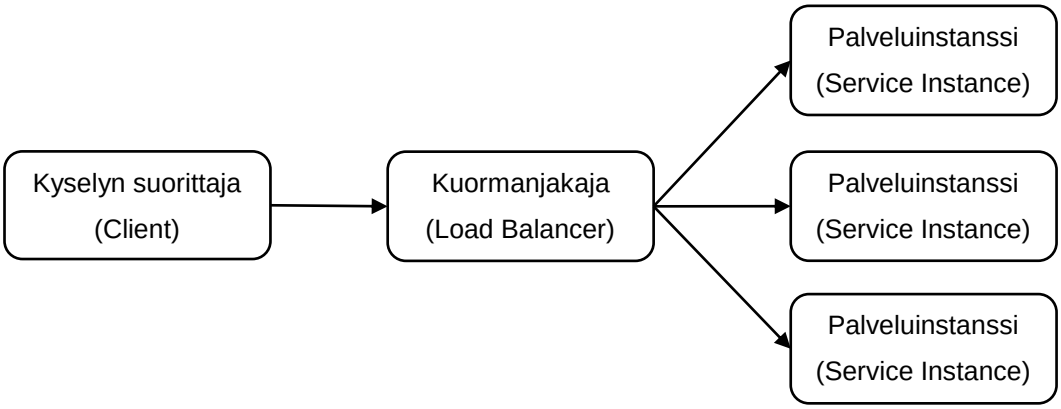
Carnell (2017, 100) määrittää havaitsemispalvelulle, eli agentille, neljä muita mikropalveluja tukevaa päätehtävää, jotka mahdollistavat

- rekisteröitymisen havaitsemispalveluun
- toisen mikropalvelun verkkosijainnin selvittämisen
- muiden mikropalvelujen tietojen kyselyn
- tilatietojen ilmoittamisen agentille

Kun palveluinstanssi käynnistyy, se rekisteröi havaitsemispalveluun fyysisen sijaintinsa, verkkopolun ja portin, joiden perusteella muut rekisteröityneet mikropalvelut pystyvät sen löytämään. Saman palvelun rinnakkaisinstanssit rekisteröivät yksilölliset verkko-osoitteensa saman palvelutunnuksen alle. Näin ollen, joukko palveluinstansseja toimii järjestelmässä saman palvelunimen takana. Jokainen rekisteröitynyt mikropalvelu lähettää jatkuvasti tilatietoja havaitsemispalvelulle saatavuutensa ja toimintakykynsä osoitukseksi. Agentti voi myös pyytää statuksen mikropalvelulta. Palvelu, joka ei lähetä tilatietoja, poistetaan automaattisesti saatavilla olevien palvelujen joukosta. (Carnell 2017, 101-102.)

Netflix Eureka on tehokas Java-kirjasto havaitsemispalvelun luontiin. Se suunniteltiin alun perin Amazon-pilvipalveluja varten. Eureka on Apache-lisenssin alaista avointa lähdekoodia. Palvelu tallentaa järjestelmän jokaisen mikropalvelun nimen alle palvelinosoitteen ja portin, jossa mikropalvelu on saatavilla. Eureka pystyy replikoimaan nämä tiedot monelle palvelimelle suorituskyvyn ja saatavuuden parantamiseksi. Tiedonvaihto Eureka-palvelun kanssa tapahtuu REST-protokollalla. Eurekaa voidaan käyttää käyttökuorman jakamiseen rekisteröimällä useita instansseja yhdeksi palveluksi. Jokaisen mikropalvelupohjaisen järjestelmän tulisi käyttää havaitsemisjärjestelmää luodakseen perustason keskitetyn hallinnan ja kuorman ohjauksen. (Eberhard 2017, 143.)

Mikropalvelujen automatisoitu skaalautuvuus useiksi instansseiksi on tärkeä kuorman jakamisen ominaisuus tuotantojärjestelmässä. REST- ja HTTP-protokollilla viestivien mikropalvelujen kuorman tasapainottamiseksi voidaan käyttää myös erillistä välityspalvelinta. Kuormanjakaja (engl. load balancer) toimii ulospäin yhtenä instanssina mutta välittää sisäänpäin viestejä useille saman palvelun instansseille. Palvelua voidaan hyödyntää myös uusien palveluversioiden käyttöönotossa. Uudelle mikropalvelulle voidaan konfiguroida aluksi vain vähän kuormaa. Mikäli ongelmia ei havaita, kuormaa voidaan asteittain kasvattaa aina täysimittaiseen käyttövalmiuteen saakka. Viestijärjestelmät voivat lähettää tietoa tarpeen mukaan joko yhdelle tai kaikille palvelun instansseille. Kuormanjakaja saa palvelininstansseilta tilatietoja poistaen toimimattomat palvelut tietoliikenteen ohjauksesta. Tällainen toteutustapa johtaa kuitenkin kaiken tietoliikenteen kulkemiseen kuormanjakajan kautta, joka tekee siitä pullonkaulan palvelujen väliselle viestinnälle. Kuormanjakaja voi myös ruuhkautua tai kaatua virhetilanteeseen, jolloin kaikki viestiliikenne katkeaa rekisteröidyille palveluille. Yhden kuormanjakajan tulisi tasapainottaa tietoliikennettä vain yhden mikropalvelun instansseille, jolloin toiminnan konfigurointi säilyy palvelukohtaisena ja selkeänä. Lisäksi tällaista välityspalvelua tulisi soveltaa vain tilattomille mikropalveluille, jotka eivät säilytä autentikoidun käyttäjän istunnon aikaista tietoa. Muutoin voi syntyä tarve välittää tietoa myös rinnakkaisten instanssien välillä, mikä rikkoo mikropalveluiden riippumattomuutta ja tekee toiminnasta kompleksista. Kuva 1 havainnollistaa tilannetta, jossa kuormanjakaja välittää kyselyn prosessoitavaksi vapaana olevaan palveluun.
(Eberhard 2017, 144-145.)



Kuva 1. Kuormaa jakava välityspalvelin (Eberhard 2017, 144)

Kuormaa jakava välityspalvelin voidaan teknisesti toteuttaa monella tavalla. Eräät sovelluspalvelimet, kuten Apache httpd server ja NginX web server, voidaan konfiguroida toimimaan tässä tehtävässä. Lisäksi eri pilvipalvelut tarjoavat tuotteita kuormanhallintaan. Esimerkiksi Amazon Web Services (AWS) tarjoaa palvelun Elastic Load Balancing, jossa kuormanhallinta voidaan yhdistää automatisoituun skaalaukseen. Tällöin palvelun korkea kuormittuminen käynnistää automaattisesti uusia rinnakkaisinstansseja.
(Eberhard 2017, 146.)

Kyselyjä lähettävä mikropalvelu voi myös itse sisältää kuormanjakajan, joka on toteutettu ohjelmallisesti osana mikropalvelun koodia tai samalla palvelinalustalla toimivan sovelluspalvelimen avulla. Ribbon on Java-kirjasto, jolla voi toteuttaa client-puolen kuormanohjauksen toiminnallisuudet. Ribbon toimii yhdessä järjestelmän palvelukomponentteja havaitsevan agentin, kuten Eureka-havaitsemispalvelun, kanssa. Kuormanohjaaja selvittää kaikki saatavilla olevat palveluinstanssit Eurekasta. Tämän jälkeen se pystyy itsenäisesti valitsemaan tiedonvaihtoon käytettävän palveluinstanssin. Tässä toteutusmallissa kuorman tasapainottamiseen liittyvät virhetilanteet on helpompi hallita. Lisäksi mikropalvelu pystyy jatkamaan palvelukutsujen suorittamista, vaikka Eureka-havaitsemispalvelu ei olisi tilapäisesti saatavilla. Menetelmä lisää merkittävästi mikropalvelun vakautta.
(Carnell 2017, 28-29.)

3.5 Tietoturva

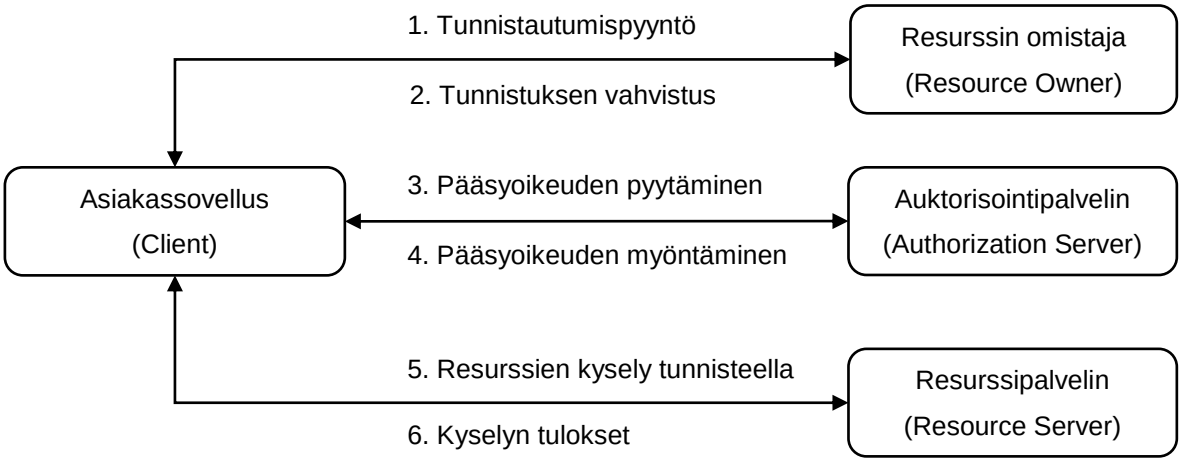
Autentikoinnin ja auktorisoinnin toteuttaminen on välttämätöntä kaikissa verkkosovelluksissa. OAuth on vahvaan asemaan noussut menetelmä edellä mainittujen toiminnallisuuksien toteuttamiseen. Se on ennen kaikkea valtuutusjärjestelmä, jolla käyttäjien tunnistus ja käyttöoikeuksien hallinta voidaan toteuttaa myös järjestelmän ulkopuolisilla palveluilla.

OAuth-standardia ja määrityksiä valvoo IETF (Internet Engineering Task Force). (Sharma 2016, 126.)

Jokaisen mikropalvelun on tiedettävä, kuka suorittaa palveluun kyselyjä. Siksi järjestelmässä on oltava tietoturva-arkkitehtuuri, joka takaa yhtenäiset käytännöt autentikointiin, eli käyttäjän identiteetin tunnistamiseen, sekä auktorisointiin, eli käyttöoikeuksien hallintaan. Yksittäisen mikropalvelun ei pidä suorittaa käyttäjän autentikointia kuten kirjautumistunnusten kelvollisuuden tarkistusta. Käyttäjien tunnistamiseen on toteutettava erillinen keskitetty palvelu, jota muut palvelut voivat tarvittaessa kutsua. Sitä vastoin, auktorisointiin tarvitaan menetelmästä riippuen eri palvelujen välistä tiedonvaihtoa. OAuth2 on laajasti verkkopalveluissa käytetty valtuutusprotokolla vahvan käyttöoikeuksien hallinnan toteuttamiseksi. Käyttöoikeuden pyytäminen ja pääsy jaettuun resurssiin, kuten tietokantaan, tapahtuu ketjutetussa tapahtumasarjassa seuraavasti:

- Asiakassovellus (client) pyytää suojatun resurssin omistajalta (resource owner), yleensä käyttäjältä, tunnistautumista käyttöoikeuden selvittämiseksi.
- Resurssin omistajan tunnistetietoja käytetään edelleen pääsyoikeuden selvittämiseen. Auktorisointipalvelin (authorization server) palauttaa käyttöoikeuden omaavia tunnistetietoja vastaan auktorisointikoodin, josta generoidaan pääsytunniste (access token) varsinaista resurssipalvelinta varten.
- Asiakassovellus suorittaa pääsytunnisteen avulla varsinaisen kyselyn resurssipalvelimelle, joka palauttaa kyselyn tulokset.

Käyttöoikeuspyyntöjen eri vaiheissa salattuja tunnistetietoja kuljetetaan HTTP-protokollan otsikkotiedoissa. OAuth2-protokollan mukaiset auktorisointivaiheet on esitetty kuvassa 2. Tavallisesti tunnistautumisvaihe ohjaa käyttäjän suoraan auktorisointipalvelimen näyttämälle verkkosivulle, jossa käyttäjä voi tarvittaessa antaa tunnistetietonsa ja vahvistaa pyyntönsä. Käyttäjän tunnistaminen on usein jo suoritettu toisaalla, jolloin asiakassovellus saa suoraan auktorisointipalvelimelta auktorisointikoodin, josta asiakassovellus generoi pääsytunnisteen tietokantakyselyn tai muun rajoitetun toimenpiteen suorittamiseksi. Mikropalvelut voivat välittää toisilleen käyttäjän pääsytunnisteen, jonka perusteella käyttäjän pääsyoikeudet eri resursseihin voidaan varmistaa suoraan. Spring Cloud Security -kirjasto tarjoaa hyvän perustan OAuth2-järjestelmän toteutukseen erityisesti Java-pohjaisille mikropalveluille. (Eberhard 2017, 152-156; Varanasi & Belinda 2015, 121-127.)



Kuva 2. OAuth2-protokolla (Eberhard 2017, 153)

OAuth2 ratkaisee tehokkaasti auktorisoinnin vaatimukset. Mikropalvelujen tietoturvaa voidaan tarvittaessa parantaa myös muilla menetelmillä. Palvelujen välinen tiedonvaihto voidaan salata SSL/TLS-sertifikaateilla, jolloin myös asiakassovellusten tunnistaminen digitaalisen allekirjoituksen perusteella on mahdollista. API-avaimien avulla järjestelmän ulkopuolisille sovelluksille voidaan antaa käyttöoikeuksia ja keinot tunnistautumiseen. OAuth2:ssa tämän ominaisuuden toteuttaa Client Credentials. Lisäksi palomuurien tehokkaalla käytöllä voidaan suojata mikropalvelujen välistä viestintää ja rajoittaa pääsyä järjestelmän eri osiin. Luvaton tunkeutuminen järjestelmään voidaan rajata näin jopa yhteen mikropalveluun. Lisäksi mikropalveluarkkitehtuuriin voidaan soveltaa periaatetta, että järjestelmä säilyttää vain kaikkein välttämättömimmän tiedon. Tiedonkeruun välttäminen tekee järjestelmästä epäkiinnostavan kohteen hyökkäyksille. (Eberhard 2017, 156-157.)

3.6 Vikasietoisuus

Mikropalvelujen sisäisen toiminnan tulisi olla vakaata ja vikasietoista. Yksittäisen mikropalvelun virhetilanteella tulisi olla minimaalinen vaikutus järjestelmän muihin mikropalveluihin. Mikropalvelujen vikasietoisuuden kasvattamiseksi voidaan soveltaa useita menetelmiä. Timeout, eli aikakatkaisu, reagoi tilanteeseen, jossa tiedonvaihdon kohdejärjestelmä ei ole syystä tai toisesta saatavilla. Odotusaika tulisi säätää pieneksi, jotta aikakatkaisun odotus ei estä palvelun muuta toimintaa. Circuit Breaker, eli piirikatkaisija, on sähkötekniikasta jäljitelty menetelmä palvelun automaattiseen sulkemiseen järjestelmästä vikatilanteessa. Tämä voidaan toteuttaa ohjelmallisesti tai tuoda järjestelmään valmiina tuotteena. Palvelun virhetilanteessa piirikatkaisija kytkeytyy päälle ja estää kaiken tietoliikenteen kaatuneeseen palveluun. Kyselyihin vastataan suoraan virheilmoituksella, jotta ne ohjataan toiseen toimivaan instanssiin. Piirikatkaisija voidaan konfiguroida toimimaan ai-

kaviiveellä, jolloin palvelulle annetaan aikaa toipua virhetilanteesta. Aikaviiveen jälkeen piirikatkaisija kytkeytyy pois päältä ja tietoliikenne ohjataan uudestaan palveluun. Hystrix on Apache-lisenssin alainen avoimen lähdekoodin Java-kirjasto, jolla tällainen toiminnallisuus on mahdollista toteuttaa. Hystrix on laajimmin käytetty Java-kirjasto mikropalvelujen vikasietoisuuden hallintaan. Se tarjoaa myös graafisen näkymän palvelujen ja piirikatkaisijoiden tilojen monitorointiin. Nopea virheilmoituksen palauttaminen voidaan ohjelmallisesti toteuttaa mikropalveluun myös ilman piirikatkaisija. Fail Fast -periaatetta noudattava palvelu pyrkii mahdollisimman nopeasti ilmoittamaan ulkopuolisille kutsujille, kun palvelun toiminta on virhetilanteen vuoksi estynyt. Tämä säästää myös aikakatkaisujen odottelulta lisäten merkittävästi järjestelmän toiminnan jouhevuutta ja vakautta.
(Eberhard 2017, 203-205.)

Laipiot (engl. bulkheads) ovat metallisia tukiseiniä, joilla laivan osastot saadaan eristettyä vedenpitäviksi. Kun laipiot on suljettu, vesi ei pääse enää osastolta toiselle. Bulkhead-eristäminen virhetilanteissa toteutetaan järjestelmään fyysisellä laitteistolla. Järjestelmä on osioitu usealle palvelimelle. Kun yhden palvelimen mikropalveluissa tapahtuu virhe, yhteydet tälle palvelimelle estetään molempiin suuntiin, jotta virhetilanteet eivät vaikuta muiden palvelimien mikropalveluihin. Tällaisiin palvelimien virhetilanteisiin voivat johtaa myös palvelimiin kohdistetut ylläpitotoimet. Palvelimet pitäisi saada pysyvään vakaaseen tilaan (engl. steady state), jotta ne pystyvät suorittamaan itsenäisesti mikropalveluja ilman ylläpitäjien puuttumista asiaan. Muutoinkin kaikenlaista palvelimien virittelyä tulisi välttää. Palvelinpohjaisiin vakausratkaisuihin voidaan soveltaa myös välimuistin käyttöä. Tällöin on tärkeää huomioida, ettei muistinvarainen väliaikaistiedon säilöminen kerrytä tietokuormaa loputtomasti. On tärkeää määrittää muistikapasiteetin rajat, jonka puitteissa välimuisti toimii. (Nygard 2012, 96-104.)

3.7 Tekniset haasteet

Järjestelmän purkaminen mikropalveluiksi tekee järjestelmän kokonaistoiminnasta monimutkaista sekä aiheuttaa haasteita sekä teknisellä että arkkitehtuurisella tasolla. Verkkoviiveet ja mikropalvelujen kaatumiset voivat aiheuttaa odottamattomia ongelmia. Kehitystyössä myös tietyn toiminnallisuuden siirtäminen mikropalvelusta toiseen voi olla hankalaa. Mikropalveluiden tekniset ja arkkitehtuuriset haasteet on hyvä ottaa huomioon jo suunnitteluvaiheessa. Verkkoviiveen aiheuttaman haitan vähentämiseksi tulisi harkita, milloin aktiivisesti keskenään viestivät mikropalvelut kannattaa yhdistää ja milloin pitää erillään. Keskinäisen riippumattomuuden saavuttamiseksi mikropalvelujen ei myöskään tulisi jakaa omia resurssejaan muiden palvelujen kanssa. Riippuvuuksia jaettuihin koodikirjastoihin tulisi myös välttää. Muutokset jaettuihin resursseihin vaativat koordinoituja

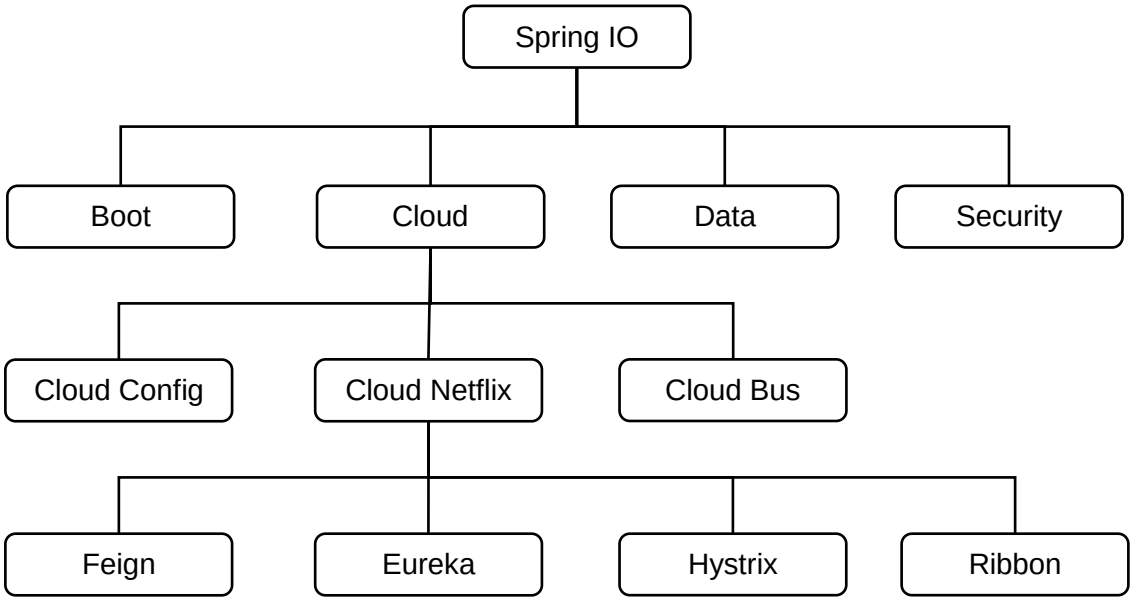
muutoksia läpi arkkitehtuurin. Lisäksi niiden toteuttamiseen tarvitaan tiivistä yhteistyötä kehitystiimien välillä, mikä rikkoo mikropalveluarkkitehtuurin periaatteita. (Eberhard 2017, 70-72.)

Verkon yli tapahtuva viestintä on epäluotettavaa eikä kyselyn suorittaminen aina onnistu sallitun viiveajan puitteissa. Lisäksi yksittäisen mikropalvelun toiminta voi pysähtyä virhetilanteeseen. Muiden mikropalveluiden on kompensoitava nämä virhetilanteet ja huolehdittava järjestelmän toiminnan jatkumisesta. Tällaisen toiminnallisuuden toteuttamiseen on olemassa erilaisia teknologioita. Vaihtoehtoisesti tieto voidaan korvata virhetilanteissa oletusarvoilla tai välimuistista palautetuilla arvoilla. Joissain tilanteissa riittää pelkkä saatavilla olevien palvelujen rajoittaminen. Tämän toteuttamiseksi voi olla välttämätöntä tinkiä palvelun laadusta. (Eberhard 2017, 73.)

Teknologisen riippumattomuuden toteuttamiseksi mikropalvelut julkaistaan usein kukin omalla virtuaalipalvelimellaan. Kun järjestelmässä on satoja mikropalveluja, ylläpidettävänä on myös satoja virtuaalikoneita. Virtualisoinnista huolimatta suuren palvelinmäärän hallinta on haasteellista. Palvelimien keskitettyyn hallintaan tarvitaan infrastruktuuria ja automatisoitua hallintajärjestelmää, jotka kykenevät luomaan suuren määrän virtuaalikoneita ja valvomaan niiden tilaa. (Eberhard 2017, 76.)

4 Tietojärjestelmän toteutus mikropalveluina

Java-pohjaisessa mikropalvelujen toteutuksessa kehittyneimmät avoimen lähdekoodin kirjastot perustuvat Spring-ohjelmistokehykseen. Tässä toteutuksessa sovellettiin erityisesti Spring-ohjelmistokehyksen aliprojekteja Spring Boot ja Spring Cloud. Nämä tarjoavat vahvat, nopeasti sovellettavat koodikirjastot pilvipalveluympäristöön kehitettävän mikropalveluarkkitehtuurin toiminnallisuuksien toteuttamiseen. Ohjelmistokehyksen hyödyntäminen vähentää merkittävästi työlästä matalan tason Java EE -ohjelmointia (Java Enterprise Edition) sekä tarjoaa perusteellisesti testatut koodikirjastot sovellusten kehittämiseen. Spring Cloud -kehys kattaa edelleen aliprojekteja pilvi- ja mikropalvelupohjaisia ratkaisuja varten. Keskeisimmät mikropalvelujen kehittämiseen sovelletut Spring-projektit on esitetty kuvassa 3. Spring Cloud Netflix -projektin alla on mikropalvelujen keskitettyyn hallintaan ja viestiliikenteen ohjaukseen soveltuvia kirjastoja, kuten Feign, Eureka, Hystrix ja Ribbon.



Kuva 3. Spring IO -ohjelmistokehys

Kehitysprojekti toteutettiin ketterien kehitysmenetelmien metodologiaa mukaillen. Projektin tavoitteena oli toteuttaa mikropalvelupohjainen tiedonhakuprosessin prototyyppi, joka mallintaa teoriaosan esittämää kestävän kehityksen mikropalveluarkkitehtuuria. Sovellusta voidaan käyttää mallina muissa mikropalvelupohjaisissa ohjelmistoprojekteissa. Lisäksi selvitettiin, miten hyvin mikropalveluarkkitehtuurin keskeisimmät ominaisuudet voidaan toteuttaa ohjelmallisesti mikropalveluihin suunniteltujen avoimen lähdekoodin API-kirjastojen avulla. Toteutuksen ulkopuolelle rajattiin kaupalliset pilvipalvelutuotteet sekä

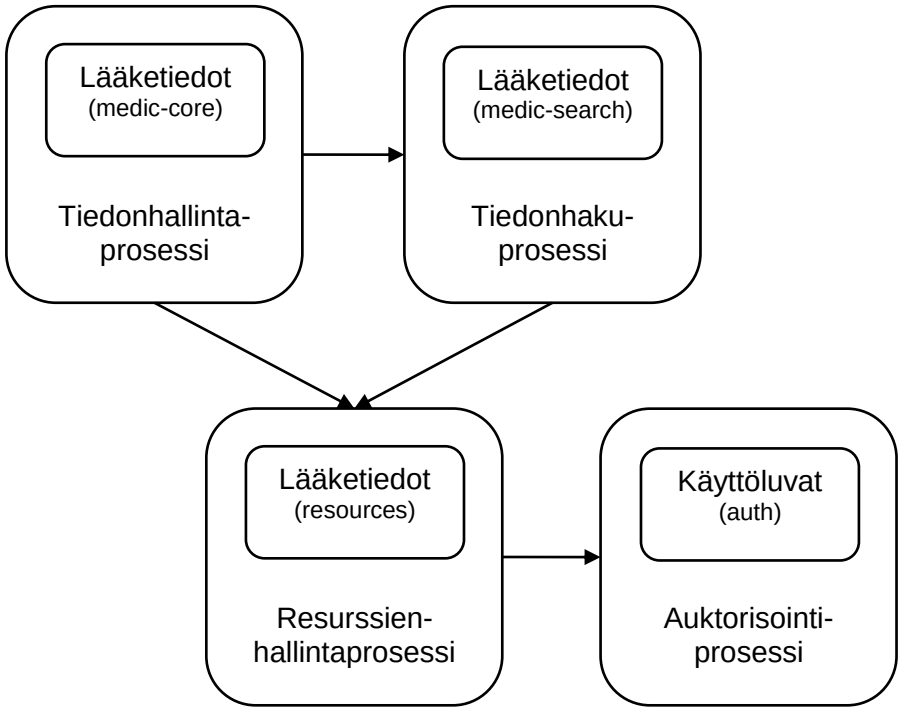
tuotantoteknologiat, joiden käyttöönotto edellyttää palvelinteknologioiden erikoisasiantuntemusta. Ohjelmistoprojekti keskittyi spesifisti järjestelmän palvelinohjelmiston (backend) kehittämiseen sekä mikropalvelujen välisen tiedonvaihdon mekanismeihin. Käyttäjärajapinnan asiakassovellukset ja käyttöliittymät (frontend) rajattiin pois tästä projektissa, koska niihin liittyvät teknologiset ratkaisut eivät ole osa mikropalveluarkkitehtuuria. Lisäksi tiedonhakuprosessin prototyyppiin ei toteutettu autentikointijärjestelmää käyttäjien tunnistamiseen. Prosessi on täysin järjestelmän sisäinen. Sitä vastoin, järjestelmän omia tietokantaresursseja suojataan käyttöoikeuksien hallintapalvelulla, joka myöntää muille mikropalveluille valtuutuksia suorittaa kyselyjä jaettuihin tietokantaresursseihin resurssipalvelun kautta.

4.1 Tietomalli ja arkkitehtuuri

Projektin sovellusalaksi valittiin tietomallin suunnittelemiseksi lääketiedon hallinta. Lääketiedolle on ominaista, että yhteen lääkeaineeseen tai -tuotteeseen liittyy runsaasti informaatiota. Lisäksi tämä informaatio sisältää tietoja, jotka linkittyvät muihin lääkeaineisiin, tieteellisiin tutkimuksiin, viranomaismääräyksiin ja valmistajan tietolähteisiin. Tällaista järjestelmää perustettaessa on haasteellista, jollei mahdotonta, ennakoida tulevaisuuden vaatimuksia järjestelmän toiminnalle. Tällöin mikropalveluarkkitehtuuri tarjoaa kestävän pohjan pitkän tähtäimen järjestelmäkehitykselle. Järjestelmän perustusvaiheessa huolehdittiin kriittisimpien mikropalvelujen korkeasta koheesiosta, kuten Eberhard painotti sisäisen arkkitehtuurin suunnittelussa. Toisin sanoen, järjestelmä jaettiin toimintalogiikan perusteella osiin, jossa jokaisella mikropalvelulla on yksiselitteinen tehtävä. Näin ollen toimintoja voitiin kehittää ilman vaikutusta muiden mikropalvelujen toimintaan. Erityisesti lääketiedon hakupalvelu (medic-search) voi ajan myötä kehittyä kompleksiseksi ja se voi integroitua muihin tietojärjestelmiin, joten se pidettiin erillään muista perustoiminnoista (medic-core). Arkkitehtuurin suunnittelussa käyttäjää palveleva lääketiedon prosessointi jaettiin tietomallin määrittämän toimintalogiikan perusteella siis kahdeksi erilliseksi mikropalveluksi – lääketiedon ydin- ja hakupalveluksi. Muut järjestelmän hallintapalvelut moduloituivat tehtävänsä perusteella erillisiksi konfiguraatio-, havaitsemis-, resurssi- ja auktorisointipalveluiksi.

Järjestelmästä luotiin tietomalli, jotta mikropalvelujen tehtävät voitiin määrittää suhteessa toimintalogiikkaan. Järjestelmän prosessoiman tiedon aihepiiri, lääketieto, määritti myös projektin sovellusalan. Järjestelmä suunniteltiin domain-driven design -periaatteilla, jolloin komponenttien tehtävät muodostuivat sovellusalan tietosisällön perusteella. Tämä on kuvattu tietomallissa niin sanotulla kontekstikartalla kuvassa 4. Lisäksi laadittiin koko arkkitehtuurin kuvaus mikropalvelujen keskinäisten suhteiden määrittämiseksi. Arkkitehtuuriku-

vauksessa ovat mukana myös järjestelmän hallintaan liittyvät mikropalvelut, jotka ovat näkymättömiä varsinaiselle sovelluslogiikalle.

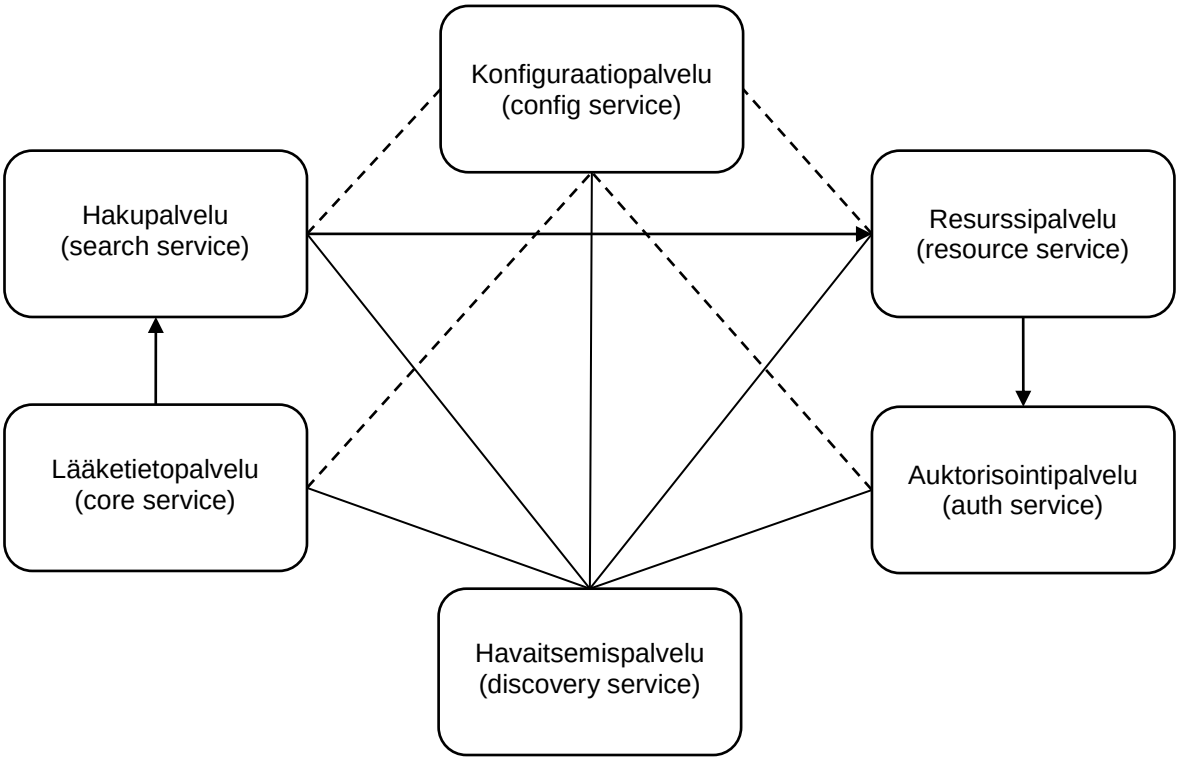


Kuva 4. Lääketietopalvelun tietomalli ja toimintalogiikka

Konfiguraatio- ja havaitsemispalvelut toteuttavat mikropalveluarkkitehtuurin keskitetyn hallinnan tärkeimmät toiminnallisuudet. Konfiguraatiopalvelu huolehtii keskitetysti kaikkien mikropalvelujen käynnistyksen ja suorituksenaikaisten vakioitujen ominaisuuksien määrityksestä. Mikäli palvelujen asetuksiin tulee muutoksia, se lähettää viestijonojärjestelmän kautta käskyn mikropalveluille ladata asetukset uudestaan konfiguraatiopalvelusta. Havaitsemispalvelu, sen sijaan, huolehtii palvelujen havaitsemisesta välittäen mikropalveluille tiedot muiden palvelujen sijainnista ja tilasta verkossa. Havaitsemispalvelu on myös osa älykästä tiedonvaihdon mekanismia, jossa mikropalvelu voi suorittaa kyselyn toiseen mikropalveluun pelkän palvelun nimen perusteella. Havaitsemispalvelu ohjaa kyselyn parhaiten saatavilla olevaan palveluinstanssiin ja hoitaa näin myös käyttökuorman tasapainotusta rinnakkaisinstanssien välillä.

Auktorisointipalvelu huolehtii ensisijaisesti käyttöoikeuksien keskitetystä hallinnasta. Resurssipalvelu huolehtii yhdessä auktorisointipalvelun kanssa pääsystä jaettuihin tietokantaresursseihin. Auktorisointipalvelu ylläpitää tietokantaa käyttöoikeuksista. Lisäksi valtuutusprosessiin liittyy tunnistesäilö (token repository), josta voimassaolevat käyttöoikeustunnisteet ovat nopeasti tarkistettavissa. Mikäli kyselyn suorittajalla ei ole voimassaolevia

käyttöoikeustunnisteita, niitä on pyydettävä auktorisointipalvelulta. Järjestelmän käsittelemät lääketiedot säilötään omaan lääketietokantaan. Kaikki transaktiot lääketietokantaan kulkevat resurssipalvelun kautta. Resurssipalvelu huolehtii yhdessä auktorisointipalvelun kanssa pääsyoikeuksista suojattuihin tietokantaresursseihin. Järjestelmän arkkitehtuuri on esitetty kuvassa 5. Yhdysnuolet osoittavat tiedonhakuprosessin suuntaa. Yhdysviivat osoittavat mikropalvelujen viestintää havaitsemispalvelun kanssa, ja katkoviivat konfiguraatiopalvelun kanssa. Järjestelmän tiedonvaihdossa on pystytty säilyttämään järjestelmän toimintaa selkiyttävä rakenne, jossa jokaisessa kahden palvelun välisessä tiedonvaihdossa toinen on yksiselitteisesti client, eli kyselyn suorittaja, ja toinen on server, eli kyselyn prosessoija sekä vastauksen palauttaja.



Kuva 5. Lääketietopalvelun arkkitehtuuri

Palvelujen konfigurointia varten määriteltiin toteutettavien mikropalvelujen nimet, joilla ne järjestelmässä tunnistetaan. Lisäksi päätettiin verkkosijainnit, eli paikallispalvelimen portit, joissa niitä tullaan kehitysympäristössä suorittamaan. Projektissa kehitettiin mikropalvelupohjaisen järjestelmänprosessin prototyyppiä, joten tuotantoympäristön määritystä ei tarvittu. Jokaisen mikropalvelun kohdalla harkittiin erikseen, rekisteröityykö palvelu konfiguraatiopalveluun ja havaitsemispalveluun. Näitä mikropalveluarkkitehtuurin hallintapalveluja esitellään seuraavissa luvuissa. Taulukko 2 koostaa toteutettujen mikropalvelujen määritykset arkkitehtuurissa.

Taulukko 2. Mikropalvelujen kehitysympäristön verkkomääritykset

Microservice	Host	Port	Config Client	Discovery Client
service-config	localhost	8888	FALSE	TRUE
service-discovery	localhost	8000	FALSE	FALSE
service-medic-core	localhost	8001	TRUE	TRUE
service-medic-search	localhost	8002	TRUE	TRUE
service-resources	localhost	8003	TRUE	TRUE
service-auth	localhost	8004	TRUE	TRUE

4.2 Palvelusovellusten konfigurointi

Tietojärjestelmään toteutettiin erillinen mikropalvelu hoitamaan keskitettyä konfiguraationhallintaa. Toteutuksessa käytettiin Spring Cloud Config -kirjastoa. Konfiguraatioon määritettiin ympäristötietojen lisäksi myös merkityksettömiä parametrejä tiedon välittymisen testaamiseksi. Konfiguraatiopalvelu, samoin kuin muut mikropalvelut, luotiin Spring Boot -projektina, jonka resources-hakemistoon lisättiin application.yml-tiedosto konfiguraatiopalvelun määrittämistä varten. Tiedostomuoto yml (tai yaml) viittaa YAML-merkintäkieleen, jonka syntaksissa puurakenne määritetään sisennyksillä. YAML on yleisesti käytetty konfiguraatiotiedostoissa helpomman luettavuuden vuoksi. Konfiguraatiot voidaan määrittää myös properties-tiedostossa, kuten application.properties, jossa puurakenteen sijaan jokainen ominaisuus on määritelty omalla rivillään. Konfiguraatiopalvelun tekniset toiminnallisuudet tuotiin lisäämällä Maven-ympäristönhallintaan kirjastoriippuvuudet spring-cloud-dependencies ja spring-cloud-config-server. Eberhard ja Rajesh painottivat versionhallinnan tärkeyttä keskitetyn konfiguraatioratkaisun toteutuksessa, jotta mikropalvelujen määritykset eivät hajoaisi sovelluksen laajentuessa. Näin ollen varsinaiset mikropalvelujen konfiguraatiotiedostot vietiin Git-versionhallintapalvelimelle käyttäen GitHub-verkkopalvelua. Konfiguraatiopalvelu määritettiin hakemaan asiakassovellusten, eli tässä tapauksessa muiden mikropalvelujen, pyytämiä konfiguraatiotietoja Git-palvelimelta. Lisäksi määriteltiin paikallinen varahakemisto, offline-repository, jonka avulla mikropalvelut saavat vähintään palvelun käynnistymiseen tarvittavat tiedot konfiguraatiopalvelusta. Tällä turvattiin konfiguraatiopalvelun toimivuus myös silloin kuin tietojen haku verkosta ei syystä tai toisesta onnistu, johon varautumista myös Eberhard korosti mikropalvelun sisäisen arkkitehtuurin suunnittelussa luvussa 3.3.

Ohjelmakoodin puolella konfiguraatiopalvelun ominaisuudet otettiin käyttöön lisäämällä annotaatio @EnableConfigServer sovelluksen pääohjelmaluokkaan ServiceConfig-Application. Annotaatiot ovat metatietoa, joiden avulla voidaan lisätä toiminnallisuuksia lähdekoodiin ja ohjata myös sovelluksen suorituksenaikaista toimintaa. Kehitysympäristössä kaikki palvelut luotiin moduuleina yhden pääprojektin sisään. Tällöin kaikki mikropalvelut toimivat kehitystyön aikana samalla paikallispalvelimella. Jokainen palvelu on siis määritettävä eri sovellusporttiin konfliktien välttämiseksi. Konfiguraatiopalvelu määriteltiin vakiintuneen käytännön mukaisesti porttiin 8888.

Konfiguraatiopalvelua (server) käyttävät mikropalvelut konfiguroitiin vastaavasti palvelun asiakassovelluksiksi (client). Keskitetyn konfiguraation client-ominaisuudet voidaan toteuttaa Spring Cloud -kirjastolla. Projektin alkuvaiheessa luotiin yksi client-palvelu, service-medic-core, joka tulisi myöhemmin sisältämään sovelluksen toimintalogiikan perustoiminnot. Medic-palvelun resources-hakemistoon luotiin käynnistystiedosto bootstrap.yml, jonne määritettiin palvelun oma sijainti verkossa (localhost:8001), konfiguraatiopalvelun sijainti (localhost:8888), palvelun nimi (service-medic-core) sekä oletuksena käytettävän konfiguraatioprofiilin nimi (default). Näiden tietojen avulla Spring Boot -sovellus hakee automaattisesti sovelluksen käynnistyessä tiedot konfiguraatiopalvelusta eikä toiminnallisuus vaadi muita toimia varsinaisen ohjelmakoodin puolella. Palvelun nimimääritys on tärkeä oikeiden konfiguraatiotiedostojen tunnistamiseksi. Versionhallintapalvelimella sijaitsevat konfiguraatiotiedostot nimetään ja haetaan palvelun nimen ja konfiguraatioprofiilin nimen perusteella. Oletustiedosto palvelulle service-medic-core on siis service-medic-core.yml tai service-medic-core-default.yml, jossa default on profiilin nimi. Kehitysympäristön määritykset voitaisi nimetä service-medic-core-development.yml ja tuotantoympäristön määritykset service-medic-core-production.yml. Konfiguraatiomäärityksille tyypillisesti yaml-tiedostojen rakentaminen vaati äärimmäistä tarkkuutta ja virheettömyyttä. Lääketietopalveluun ohjelmoitiin kevyt REST-rajapinta, josta tiedon välittyminen Git-palvelimelta konfiguraatiopalvelun kautta voitiin testata. Kun määritykset saatiin oikein yaml-tiedostoihin, palvelut toimivat käynnistyksen jälkeen vakaasti ja lääketietopalvelun määritystietoja voitiin hakea myös verkkoselaimella REST-rajapinnan kautta. Kuvassa 6 on esitetty server- ja client-sovelluksen perusmääritykset keskitetyn konfiguraatiopalvelun toteuttamiseksi.

SERVER - application.yml

```
---
spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/micaro/ont-project
          searchPaths: config
        native:
          searchLocations: classpath:offline-repository/
server:
  port: 8888
```

CLIENT - bootstrap.yml

```
---
spring:
  profiles:
    active: default
  application:
    name: service-medic
  cloud:
    config:
      uri: http://localhost:8888
server:
  port: 8001
```

Kuva 6. Konfiguraatiopalvelun määritys

Tämä on nopea ja suoraviivainen tapa pienen tai keskisuuren järjestelmän konfigurointiin. Ratkaisu ei ole kuitenkaan sellaisenaan dynaaminen sovelluksen tai konfiguraation muutostilanteissa, koska se vaatii sovelluksen kääntämisen ja uudelleenkäynnistämisen. Toteutustavan hyötynä on kuitenkin konfiguraatiopalvelun riippumattomuus palvelinalustasta, joten palvelua on helppo replikoida erilaisille virtuaalipalvelinalustoille palvelun saatavuuden parantamiseksi. Konfiguraatiopalvelun on huolehdittava, että kaikilla saman palvelun rinnakkaisinstansseilla on aina sama konfiguraatio. Siksi tarvitaan lisämenetelmiä muutoksien lataamiseksi kaikkiin mikropalveluihin. Keskitetty konfiguraatio voidaan toteuttaa myös palvelinympäristön menetelmillä, kuten ympäristömuuttujilla. Tällöin sovellus ei ole tosin enää siirrettävissä tuotantoympäristöstään, johon se on suunniteltu. Konfigurointipalvelua kehitettäessä oivallettiin, että versionhallintapalvelimella oleviin konfigurointitiedostoihin voidaan sisällyttää lisäksi myös muita vakioituja parametrejä palvelujen toiminnan ohjaamiseksi.

Mikropalvelut hakevat konfiguraatiotiedot keskitetystä palvelusta ilman apumenetelmiä vain käynnistyksen yhteydessä, joten muutokset vaativat palvelujen uudelleenkäynnistyksen. Spring Cloud Bus -kirjasto tarjoaa mekanismin konfiguraatiotietojen muutosten välittämiseksi mikropalveluille niiden lukumäärästä ja sijainnista riippumatta. Näiden tietojen muuttaminen voi vaatia ohjelmakoodin puolella annotaation @RefreshScope käyttöä, koska vakioita ei voi lähtökohtaisesti muuttaa ohjelman suorituksen aikana. Mikropalvelu-

jen asetustietojen muutostapahtuma toteutettiin projektissa viestiteknologialla, joka käyttää AMQP-protokollaa (Advanced Message Queuing Protocol). Toiminnallisuus otettiin käyttöön lisäämällä kaikkiin konfiguraation muutostapahtumia seuraaviin mikropalveluihin kirjastoriippuvuus spring-cloud-starter-bus-amqp. Sama kirjasto toteutti myös tarvittavat toiminnallisuudet itse konfiguraatiopalveluun, joka johtaa järjestelmän muutostapahtumaa. Cloud Bus tunnistaa automaattisesti, onko mikropalvelu server vai client. Lisäksi viestijärjestelmän toteuttamiseen tarvitaan varsinainen viestinvälitysteknologia. Suosituimmista viestiteknologioista RabbitMQ on nopein asentaa käyttövalmiiksi kehitysympäristöön. RabbitMQ Serverin (versio 3.6.14) asentaminen vaati myös Erlang-ohjelmointikielen (versio 9.1) suoritusympäristön asennuksen. Muutostapahtuma voitiin tämän jälkeen laukaista lähettämällä tyhjä palvelupyyntö POST-metodilla konfiguraatiopalvelun REST-rajapinnan päätepisteeseen /bus/refresh - kehityksen aikana localhost:8888/bus/refresh. Toiminnon testaamisessa törmättiin kyselyn suorittamisen käyttöoikeusongelmiin, jonka takia kysely palautti virhestatuksen ”401 Unauthorized”. Tämä voitiin kiertää kehityksen ajaksi lisäämällä application.yml-tiedostoon määritys ”management.security.enabled: false”, joka poisti Spring Boot -sovelluksen käyttöoikeuksien tarkistuksen käytöstä. Tämän jälkeen kysely palautti statuksen ”200 OK” ja todettiin, että Git-palvelimelle tehdyt asetusmuutokset välittyivät onnistuneesti kaikkiin mikropalveluihin. On kuitenkin huolehdittava, että tuotantojärjestelmässä mikropalveluiden asetusten uudelleenlataaminen voi tapahtua vain määritetyn käyttöoikeuden omistajan käynnistämänä. Konfiguraatiopalvelun tilaa voidaan monitoroida suorittamalla kutsu palvelun päätepisteeseen /health. Tämä palautti testissä JSON-muotoisen tilatiedon {”status”: ”UP”}. Projektin edetessä mikropalvelun tilatiedot täydentyivät myös muiden hallintateknologioiden lisäämillä tiedoilla.

RabbitMQ:n käyttöönotto aiheutti konfliktin palvelujen verkko-osoitteisiin. Viestijärjestelmän Erlang-suoritusympäristö käynnistyi oletuksena porttiin 8080. Tämä portti oli jo määritelty järjestelmään Eureka-havaitsemispalvelun käyttöön, jota käsitellään seuraavassa luvussa. Käytännöllisin tapa ratkaista tämä konflikti oli vaihtaa Eureka-palvelu toiseen porttiin. Koska kaikki muut järjestelmän mikropalvelut oli määritetty rekisteröitymään portissa 8080 sijaitsevaan Eureka-palveluun, tarvittiin muutos kaikkien mikropalvelujen konfiguraatioihin. Koska palvelut saavat Eureka-palvelun tiedot keskitetystä konfiguraatiopalvelusta, verkko-osoitteen muutos oli nopea toteuttaa kaikille järjestelmän palveluille. Havaitsemispalvelu määritettiin porttiin 8000, jonka jälkeen lähetettiin muutospyyntö kohteeseen /bus/refresh. Mikropalvelut saivat viestijärjestelmästä käskyn ladata asetukset uudestaan konfiguraatiopalvelusta, ja tiedonvaihto käynnistyi uudestaan Eurekan uuteen verkko-osoitteeseen. Keskitetty konfigurointipalvelu todisti vahvuutensa heti projektin alkupuolella. Keskitetty konfigurointipalvelu on ehdoton hallintakomponentti hajautetussa mikropalvelujärjestelmässä.

4.3 Palvelujen rekisteröinti ja havaitseminen

Mikropalvelujen havaitsemiseen ja rekisteröimiseen (engl. service discovery) käytettiin Spring Cloud Netflix Eureka -kirjastoa. Havaitsemispalvelun toteuttamiseksi Maven-riippuvuuksiin lisättiin spring-cloud-starter-eureka-server. Eberhard kehotti välttämään kiertäviä riippuvuuksia mikropalvelujen välillä, mikä tuottaa odottamattomia virhetilanteita sekä kehityksen aikana, että tuotannossa suorituksen aikana. Tähän ongelmaan törmättiin ensimmäisen kerran havaitsemispalvelua perustettaessa. Myös havaitsemispalvelun pitäisi olla konfiguraatiopalvelun client, jotta sitä voidaan skaalata ja sen konfiguraatiota hallita keskitetysti konfiguraatiopalvelun kautta. Toisaalta konfiguraatiopalvelun on oltava havaitsemispalvelun client, jotta se saa tietoja muista järjestelmän palveluista ja on itse saatavilla muille palveluille. Koska molemmat ovat riippuvaisia toisistaan, ne eivät pysty käynnistymään ja rekisteröitymään järjestelmään, mikäli toinen palvelu ei ole käynnissä ja saatavilla. Tässä tilanteessa ensin käynnistetty palvelusovellus kaatuu siis aina virheeseen. Hallintamenetelmät sitovat dedikoituneet mikropalvelut tiukasti yhteen, joten riippumattomuutta ei pysty toteuttamaan kuten tavallisissa sovelluspalveluissa. Jokin järjestelmän hallintapalveluista on siis oltava riippumaton (engl. stand-alone), jotta järjestelmän käynnistämisellä on jokin aloituspiste. Tässä tapauksessa ensin käynnistettäväksi, riippumattomaksi järjestelmäkomponentiksi valittiin havaitsemispalvelu, koska se voidaan konfiguroida myös paikallisesti ilman konfiguraatiopalvelun käyttöä. Sen sijaan, konfiguraatiopalvelun rekisteröityminen havaitsemispalveluun on tärkeää, jotta palvelut löytävät toisensa molempiin suuntiin. Tässä projektissa Eureka-palvelua ei replikoitu moneksi instanssiksi, joten ainoaan instanssiin konfiguroitiin application.yml-tiedostoon kiinteä verkko-osoite (REST-päätepiste) palvelemaan muiden mikropalvelujen rekisteröitymistä järjestelmään. Mikäli palveluja olisi enemmän, myös Eureka-palvelu itse rekisteröityisi client-roolissa muihin havaitsemispalveluihin. Alla oleva konfiguraatio määritti yhden riippumattoman Eureka-palvelun toiminnan.

```
---
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
    serviceUrl:
      defaultZone: http://127.0.0.1:8000/eureka/
server:
  port: 8080
```

Ohjelmakoodin puolella Eureka-palvelu toteutetaan pääohjelmaluokassa annotaatiolla @EnableEurekaServer.

```
@SpringBootApplication
@EnableEurekaServer
public class ServiceDiscoveryApplication {

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

Vastaavasti kaikki muut palvelut, jotka liittyvät Eurekan kautta järjestelmään, määritettiin client-rooliin. Ensimmäisenä havaitsemispalvelun asiakkaaksi määritettiin konfiguraatiopalvelu. Kirjastoriippuvuus on sama kuin server-puolella. Client-puolen palvelujen konfiguraatioon on määritettävä mikropalvelusovelluksen nimi, jolla se identifioidaan järjestelmässä. Tämä tulee lisätä tiedostoon bootstrap.yml, jonka määritykset suoritetaan ensimmäisenä Spring Boot -sovelluksen käynnistyessä. Tämän lisäksi asetustiedostoihin lisättiin Eureka-palvelun sijainti verkossa.

```
---
spring:
  application:
    name: service-config
eureka:
  client:
    serviceUrl:
      defaultZone: http://127.0.0.1:8000/eureka/
```

Ohjelmakoodin puolella client-ominaisuudet saatiin käyttöön annotaatiolla @EnableDiscoveryClient, jonka jälkeen palvelu rekisteröityi käynnistyessään automaattisesti Eureka-palveluun ja sai haettua tietoja muista järjestelmään rekisteröityneistä palveluista.

```
@SpringBootApplication
@RestController
@EnableConfigServer
@EnableDiscoveryClient
public class ServiceConfigApplication {

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

Rekisteröityneiden mikropalvelujen päätepisteistä /service-instances/service-name voitiin hakea toisen mikropalvelun tietoja. Tämä kysely tuotti kattavan JSON-listauksen palvelun ominaisuuksista. Alla on esitetty osa kyselyn tuloksen sisällöstä.

```
"host": "192.168.1.102",
"port": 8001,
"uri": "http://192.168.1.102:8001",
"metadata": {},
"serviceId": "SERVICE-MEDIC-CORE",
"instanceInfo": {
"instanceId": "192.168.1.102:service-medic-core:8001",
"app": "SERVICE-MEDIC-CORE",
"appGroupName": null,
"ipAddr": "192.168.1.102",
"sid": "na",
"homePageUrl": "http://192.168.1.102:8001/",
"statusPageUrl": "http://192.168.1.102:8001/info",
"healthCheckUrl": "http://192.168.1.102:8001/health",
```


Näillä tiedoilla mikropalvelut saavat yhteyden toisiinsa ja pystyvät kohdistamaan kyselyjä haluttuun instanssiin kompleksisessa järjestelmässä. Mikäli tarvittava palveluinstanssi kuormittuu liikaa tai kaatuu virhetilanteeseen, Eureka-palvelu antaisi tiedot palvelun toiseen instanssiin, jonka URI (Uniform Resource Identifier) voi olla esimerkiksi seuraava IP-osoite http://192.168.1.103:8001. Tehokkaiden valvontaominaisuuksien lisäksi tämä on mikropalvelujen välisen tiedonvaihdon sydän. Havaitsemisjärjestelmä on edistyksellinen menetelmä dynaamisen järjestelmän tietoliikenteen hallintaan. Varsinainen elonmerkki lähetetään kuitenkin Eurekaan rekisteröityneiltä client-palveluilta. Ne lähettävät säädettävällä sekvenssillä signaalia (heartbeat) Eureka-palveluun toimintakykynsä osoitukseksi. Mikäli syke lakkaa asetetun aikakehyksen ajaksi, palveluinstanssi poistetaan automaattisesti järjestelmästä ja tietoliikenne ohjataan muihin instansseihin. Palvelu voi toivuttuaan rekisteröityä uudestaan järjestelmään. Eurekan toimintaa voidaan muuttaa sekä ohjelmallisesti että asetustiedostoilla. Oletusasetuksillakin havaitsemispalvelu toimi virheettömästi ja tehokkaasti koko kehitystyön ajan. Samoin client-palvelun sammuttaminen ohjasi kyselyt toiminnassa olevaan rinnakkaisinstanssiin ilman havaittavaa viivettä. Aikaviiveiden määrittämisessä on kuitenkin huomioitava, että kehitysympäristössä verkkoviive on minimaalinen, koska kaikki palvelut ovat samalla paikallispalvelimella. Tuotannossa mikropalvelujen välinen viestintä verkon yli on selvästi hitaampaa teknisen analytiikan asteikoilla. Eureka-palvelulta saadaan kyseltyä kattavasti järjestelmän palvelujen tilatietoja ulkoiseen valvontajärjestelmään, joka koostaa tiedot mielekkäästi graafisen näkymään. Projektissa havaittiin pelkkien lokitietojen olevan hyvä tapa seurata palvelujen tilaa. Oheinen näyte Eurekan lokitiedoista sisältää mikropalvelujen tilatietoja, joista rekisteröityneet mikropalvelut ovat myös nopeasti tarkistettavissa. Tiedoista nähdään palvelun nimi, tila ja verkko-osoite. Lisäksi nähdään, onko palvelu replikoitu järjestelmässä moneksi instanssiksi.

```
Registered instance SERVICE-CONFIG/192.168.1.102:service-config:8888 with
status UP (replication=false)

Registered instance SERVICE-MEDIC-CORE/192.168.1.102:service-medic-core:8001
with status UP (replication=true)

Registered instance SERVICE-MEDIC-SEARCH/192.168.1.102:service-medic-
search:8002 with status UP (replication=true)
```

4.4 Tiedonvaihto ja tietokannat

Mikropalveluiden välinen viestintä toteutettiin REST-rajapintojen kautta. Transaktioissa läpi arkkitehtuurin mikropalvelut kutsuvat seuraavan mikropalvelun rajapinnassa olevaa päätepistettä ja voivat välittää kyselyn kuormana tietosisältöä eteenpäin. Lisäksi tietoa voidaan välittää HTTP-protokollan otsikkotiedoissa. Jokainen mikropalvelu voidaan nähdä järjestelmäprosessin vaiheena, jossa tietoa voidaan tarvittaessa muokata mikropalvelun suorittaman tehtävän mukaisesti. Kun tiedonvaihdon kutsuketju oli toteutettu lääketiedon ydinpalvelusta tiedonhakupalvelun ja resurssipalvelun kautta tietokantaan, järjestelmäprosessin kahdensuuntainen tiedonsiirtokanava on periaatteessa jo olemassa. Ongelmaksi kuitenkin muodostuu se, ettei kussakin vaiheessa kutsua suorittava mikropalvelu tiedä ilman apukeinoja, missä kutsun kohteena oleva mikropalvelu sijaitsee verkossa. Sen lisäksi, mikropalvelun skaalaus ja viestiliikenteen ohjaus voi muuttaa kutsun kohteena olevaa palveluinstanssia. Aiemmin konfiguroitu havaitsemispalvelu ylläpitää rekisteriä saatavilla olevista palveluista ja tietää niiden verkkosijainnit. Kehittäjän olisi siis ensin suoritettava kysely esimerkiksi Eureka-havaitsemispalveluun. Kun halutun palvelun verkko-osoite ja saatavuus ovat selvillä, voidaan varsinainen palvelukutsu suorittaa seuraavan prosessivaiheen mikropalveluun. Tämän toteuttaminen vaatisi jonkin verran ohjelmointityötä, joka voidaan kuitenkin välttää automatisoimalla kohdepalvelun selvittäminen ennen jokaista REST-kyselyä. Tähän tarkoitukseen sovellettiin Feign-kirjastoa, jolla voidaan luoda sovelluskerroksen rajapinta toisen mikropalvelun rajapinnan toimintojen kutsumiseksi palvelunimen kautta. Feign toimii yhdessä Eurekan kanssa välittäen kutsun automaattisesti parhaiten saatavilla olevan palveluinstanssin REST-päätepisteeseen. Toteutus on kuvattu tarkemmin jäljempänä. Näillä menetelmillä saatiin ketterästi aikaan kahdensuuntainen viestiyhteys transaktioon osallistuvien mikropalvelujen välille.

Koska järjestelmään voidaan tulevaisuudessa ottaa käyttöön erilaisia tietokantapalvelimia, siihen varauduttiin toteutuksen aikana hyödyntämällä mikropalveluihin upotettavia H2-muistitietokantoja. Tällöin säästyttiin erillisten tietokantapalvelimien asennuksilta sekä sovellusten teknologiakohtaiselta konfiguroinnilta tietokantojen käyttöön. Muistitietokannan säilömät tiedot häviävät sovelluksen sammutuksen yhteydessä, joten tietokannan kehityksenaikaiset alustustiedot määritettiin sovelluksen sisällä. Usein sovelluksen konfi-

guraatiohakemistoon luodaan SQL-skriptit tietokannan skeeman, eli tietorakenteen, sekä kehityksessä tarvittavan testitiedon määrittämiseksi. Projektissa tiedonhallintaan sovellettiin ORM-menetelmiä (Object Relational Mapping), jolloin tietokannan skeema määritellään Java-luokissa JPA-annotaatioiden avulla (Java Persistence API). Lääketietokanta upotettiin resurssipalveluun, joka valvoo myös tietokannan käyttöoikeuksia yhdessä auktorisointipalvelun kanssa. Muistitietokantojen käyttö kehitystyössä on nopea ja edistyksellinen tapa korvata tuotannossa käytettävää tietokantapalvelinta. Muistitietokantoja voidaan käyttää myös mikropalveluissa, jotka eivät säilö pysyvästi tietoa. H2-muistitietokannan vähimmäismääritykseen riittää application.yml-tiedostossa alla esitetystä konfiguraatiosta pelkkä spring.datasource.url ja jpa.hibernate.ddl-auto: create-drop, jolloin tietokanta luodaan automaattisesti sovelluksen käynnistyessä ja hävitetään sammutettaessa.

```
spring:
  datasource:
    url: jdbc:h2:mem:test;LOCK_MODE=3
  h2:
    console:
      enabled: false
  jpa:
    hibernate:
      ddl-auto: create-drop
    show-sql: false
```

REST-protokollaa toteuttavan rajapinnan ominaisuudet ja edellä mainitut ORM-toiminnallisuudet tuotiin Spring Boot -sovelluksiin kirjastoriippuvuuksilla spring-boot-starter-data-rest ja spring-boot-starter-data-jpa. Feign-kirjastolla automatisoitiin saatavilla olevan palveluinstanssin verkkosijainnin haku Eureka-havaitsemispalvelusta. Maven-kirjastoriippuvuuksiin lisättiin vielä spring-cloud-starter-feign. Sovellusten RestController-luokkiin määriteltiin Feign-rajapinnat, jotta toisen palvelun REST-päätepisteitä voidaan kutsua pelkän palvelunimen kautta. Alla esitetyssä koodinäytteessä toteutettiin lääketiedon ydinpalveluun (service-medic-core) Feign-rajapinta, jotta kaikkien lääketietojen pyytäminen lääketiedon hakupalvelulta (service-medic-search) olisi ketterää toteuttaa ohjelmallisesti. Feign vapauttaa mikropalvelun täysin verkkosijaintien selvittämiseltä. Ohjelmakoodi määrittää, että kutsuttaessa palvelun päätepistettä /medicines, palvelu suorittaa edelleen kyselyn toiseen palveluun nimeltä service-medic-search, jonka päätepisteestä /medicines noudetaan kaikki lääketiedot. Feign-kirjaston toiminnallisuudet siis piilottavat kokonaan tapahtuman, jossa Eureka-havaitsemispalvelulta pyydetään kohdepalvelun vapaan instanssin verkko-osoite. REST-kyselyt toimivat hyvin Feign-rajapinnan kautta. Tämän jälkeen kehitystyössä voitiin käyttää muita mikropalveluja kuin ne olisivat olleet saman palvelusovelluksen sisällä.

```
@RestController
public class MedicCoreController {

   private ServiceMedicSearch serviceMedicSearch;

   @Component
   @FeignClient("service-medic-search")
   public interface ServiceMedicSearch {

      @GetMapping("/search/{term}")
      List<Medicine> search(@PathVariable("term") String term);
   }

   @Autowired
   public MedicCoreController(ServiceMedicSearch serviceMedicSearch) {
      this.serviceMedicSearch = serviceMedicSearch;
   }

   @GetMapping("/search/{term}")
   public List<Medicine> search(@PathVariable("term") String term);
}
```

Resurssipalveluun luotiin sovelluskerroksen rajapinnan avulla CrudRepository, jolla kyselyjen suorittaminen Java-luokilla määritettyyn muistitietokantaan oli ketterää. Alla olevasta ohjelmakoodinäytteestä nähdään, että rajapinta säilöö Medicine-olioita. Repository tarjoaa valmiina kattavasti valmiita metodeja tiedonhallintaan. Lisäksi määritettiin mukautetut metodit findAll() kaikkien lääketietojen hakuun ja findByMedicineNameIgnoreCase-Containing(String medicineName) lääketietojen etsimiseen lääkeaineen nimen tai nimen osan perusteella. Näiden metodien kutsuminen oli siis projektissa toteutettavan tiedonhakuprosessiin lopussa. Tietokannasta palautuvat hakutulokset lähdetään näiden jälkeen kuljettamaan takaisin kutsujalle samaa mikropalveluketjua pitkin.

```
@Repository
public interface MedicineRepository extends CrudRepository<Medicine, Long> {

   List<Medicine> findAll();

   List<Medicine> findByMedicineNameIgnoreCaseContaining(
      String medicineName);

}
```

4.5 Käyttöoikeuksien hallinta

Järjestelmään toteutettiin erillinen auktorisointipalvelu, joka valtuuttaa muut mikropalvelut suorittamaan kyselyjä suojattuihin resursseihin. Tässä projektissa lääketietokanta toimi muille mikropalveluille jaettuna mutta suojattuna resurssina. Resurssipalvelu selvittää auktorisointipalvelusta, että kyselyä suorittavilla mikropalveluilla on riittävät käyttöoikeudet pyytämänsä toimenpiteen suorittamiseksi lääketietokantaan. Valtuutuskäytännön toteut-

tamiseen sovellettiin tietoturvaa käsittelevän luvun 3.5 menetelmää. OAuth2-konteksti on hyvin laaja. Käyttöoikeuksien myöntäminen voidaan toteuttaa ohjelmallisesti lukuisilla eri tavoilla. Tässä kuvataan resurssi- ja auktorisointipalvelun välisen tiedonvaihdon pääkohdat, joilla tietokantaan pääsyä suojataan.

Auktorisointipalvelun toteuttamiseen tarvittiin kirjastoriippuvuudet spring-cloud-security ja spring-security-oauth2. Konfiguraatioon määritettiin REST-päätepiste /auth, johon käyttöoikeuspyynnöt kohdistetaan. Ohjelmallisesti määritettiin tavat, joilla käyttöoikeus voidaan myöntää. AuthorizedGrantTypes-määritykseen lisättiin siis valtuutuskäytäntöjä, kuten client_credentials ja refresh_token. Ensimmäinen tarkoittaa muiden mikropalvelujen tunnistautumista käyttäjätunnuksella ja salasanalla. Jäljempi tarkoittaa tunnistetta, jolla client-palvelu voi pyytää pääsyoikeutta uudestaan, kun aiempi valtuutus on vanhentunut. Scope-määrityksen arvo webclient määritti valtuutuskäytännön kattavan kaikki verkkosovellusten suorittamat kyselyt.

Kun mikropalvelu kohdistaa kyselyn lääketietokantaan resurssipalvelun kautta, kyselyn mukana kuljetetaan HTTP-protokollan parametrit, joilla käyttöoikeuksien tarkistaminen on mahdollista. Kyselyssä on oltava vähintään parametrit grant_type arvolla client_credentials ja scope arvolla webclient, sekä username ja password. Projektissa mikropalvelujen tunnistamiseen käytettiin käyttäjätunnuksena system ja salasanana myös system. Nämä tunnukset lisättiin auktorisointipalvelun muistinvaraiseen autentikointijärjestelmään. OAuth2-kirjaston tarjoamat toiminnallisuudet luovat automaattisesti auktorisointipalveluun REST-päätepisteitä valtuutusmekanismin toteuttamiseksi.

Kyselyä suorittavan mikropalvelun tunnistetiedot lähetetään resurssipalvelusta auktorisointipalvelun päätepisteeseen /auth/oauth/token. Mikäli auktorisointipalvelu valtuuttaa tunnistetietoja vastaan kyselyn suorittamisen, se palauttaa resurssipalvelulle tunnisteet kyselyn suorittamiseksi tietokantaan. Alla on esitetty vastauksena saadut tunnistetiedot, joilla kyselyn suorittaminen suojattuun tietokantaresurssiin oli mahdollista.

```
{
    "access_token": "abe3f3c3-7010-470e-93be-4803154435b8",
    "token_type": "bearer",
    "refresh_token": "694f25f4-5f99-4f0a-8944-30230bdf589d",
    "expires_in": 43199,
    "scope": "webclient"
}
```

Pääsytunnisteella (access_token) resurssipalvelu suorittaa kyselyn suojattuun tietokantaresurssiin. Kenttä expires_in määrittää vanhenemisajan pääsytunnisteelle. Tämän jälkeen tunnisteen uusiminen onnistuu nopeasti virkistystunnisteen (refresh_token) avulla. Tunnis-

teiden nopea vanheneminen suojelee tehokkaasti verkkouhkilta, kuten hakkeroinnilta. Tunnistautumisen jälkeen toimenpiteet on suoritettava nopeasti loppuun tai tunnistautuminen on uusittava. Tällöin kyselyn suorittamiseen tarvittavat tunnisteet vaihtuvat. Verkossa tapahtuvien väärinkäytösten toteuttamiseen ei jää siis riittävästi aikaa. Toteutettu auktorisointi on kevennetty versio OAuth2-valtuutuskäytännöstä, jolloin pääsytunnisteen myöntämisessä on vähemmän vaiheita.

Järjestelmäprosessin prototyyppiin toteutettiin auktorisointipalvelu valtuuttamaan ainoastaan järjestelmätunnuksilla suoritettavia tietokantakyselyjä. Ratkaisu mallintaa paikkaa prosessiketjussa ja arkkitehtuurissa, jossa auktorisointia suoritetaan. Tuotantojärjestelmää varten olisi myös myöhemmin kehitettävä menetelmä, jolla auktorisoinnin vaatima tunnistetietojen propagointi suoritetaan palveluketjun läpi - käyttäjältä aina resurssipalvelulle asti. Spring Cloud Netflix Zuul -kirjastolla on mahdollista toteuttaa viestiliikenteen reititys mikropalvelujen välille, joka huolehtii myös HTTP-protokollan mukana kulkevien otsikkotietojen ja parametrien välittymisestä. Tällöin jokaiseen mikropalveluun ei tarvitse toteuttaa erillistä menetelmää propagointia varten. Menetelmän toteuttamisen katsottiin olevan kuitenkin prosessisuunnittelun ulkopuolella, joten se rajattiin pois prototyypin toteutuksesta.

4.6 Palvelusovellusten vikasietoisuus

Mikropalvelun kestävyydellä (engl. resilience) tarkoitetaan erityisesti virhetilanteiden hallintaa. Vikasietoisuuden vahvistamiseksi on lukuisia menetelmiä, joista Eberhardin kuvaama piirikatkaisija luvussa 3.6 on oivallinen ja ketterästi toteutettava mikropalvelun sisäinen ratkaisu. Toiminnallisuus on toteutettavissa useilla eri API-kirjastoilla, joista valittiin Hystrix nopean käyttöönottoratkaisun perusteella. Hystrix otettiin käyttöön kaikilla järjestelmän mikropalveluilla. Lääketietopalvelujen Maven-kirjastoriippuvuuksiin lisättiin spring-cloud-starter-hystrix ja pääohjelmaluokkaan annotaatio @EnableHystrix. Tämän jälkeen Hystrix toimi oletusasetuksilla virhetilanteiden hallitsemiseksi. Ilman mukautettuja määrityksiä Hystrix reagoi palvelun virheisiin, kun sovelluksessa tapahtui 20 virhettä (exception) 5 sekunnin sisällä. Toimivuus testattiin ohjelmoimalla REST-rajapintaan virhetapahtuma jokaisen palveluun kohdistuvan kyselyn yhteydessä. Kun palveluun kohdistettiin runsaasti kyselyjä verkkoselaimella tai Postman-apuohjelmalla Hystrix-piirikatkaisia kytkeytyi päälle estäen uusien kyselyjen pääsyn sovellukseen. Tässä tilanteessa Eureka-havaitsemispalvelu ohjaa kyselyt palvelun toiseen instanssiin. Piirikatkaisija antaa sovellukselle oletuksena myös 5 sekuntia aikaa toipua virhetilanteesta, jonka jälkeen piirikatkaisia kytkeytyy pois päältä palauttaen palvelun muiden mikropalvelujen saataville. Menetelmällä voidaan toteuttaa ohjelmallisesti muitakin toimenpiteitä palvelun katkaisun lisäksi. Annotaatio @HystrixCommand määrittää metodit, jotka suoritetaan piirikatkaisijan kytkeytyessä pääl-

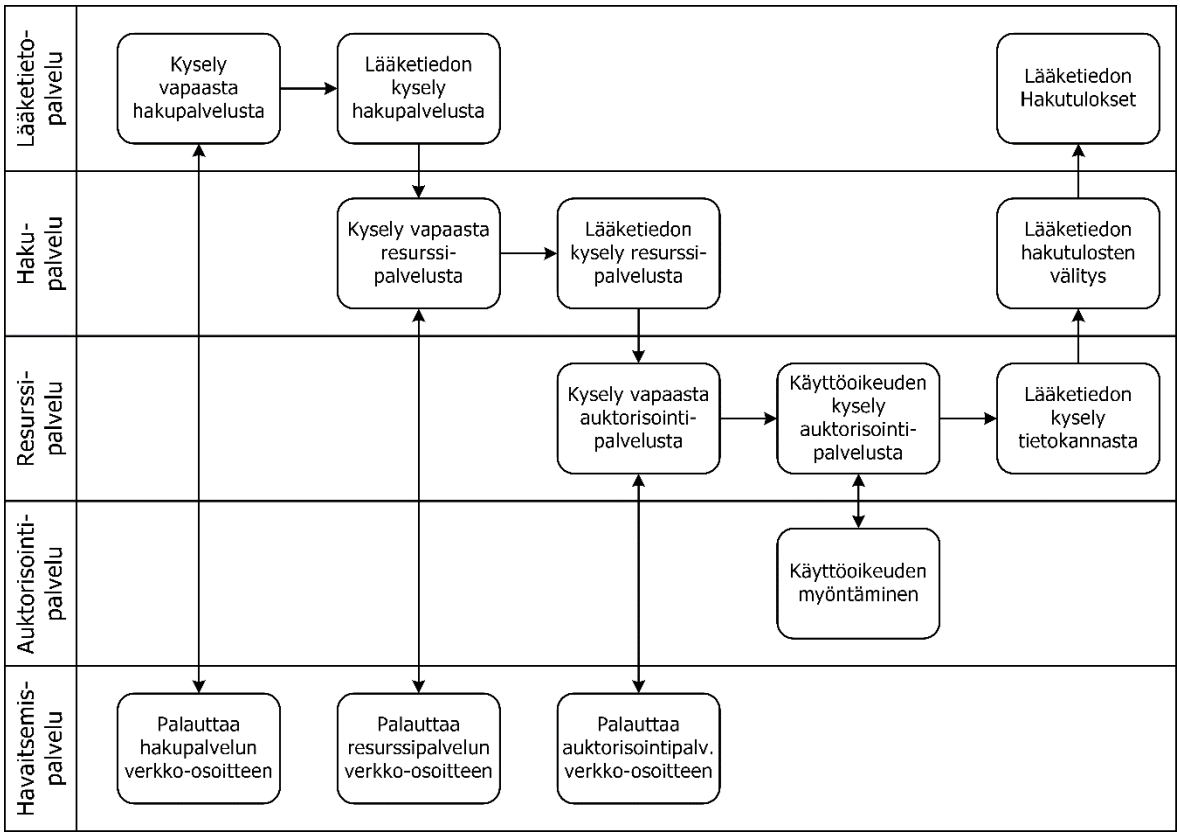
le. Lisäksi voidaan määrittää erilaiset ehdot, joiden toteutuessa kyseinen metodi suoritetaan. Alla on kuvattu metodin määritys, joka suoritetaan, kun yli 20 % palveluun kohdistuneista kyselyistä johtaa virheeseen 10 sekunnin tarkastelujaksolla. Piirikatkaisijan kytkeydyttyä päälle palvelu yritetään palauttaa takaisin käyttöön 1000 millisekunnin kuluttua.

```
@HystrixCommand(
   fallBackMethod=”defaultMethod”,
   commandProperties={
   @HystrixProperty(circuitBreaker.errorThresholdPercentage”, value=”20”),
   @HystrixProperty(circuitBreaker.sleepWindowInMilliseconds”, value=”1000”)
   }
)
public Object method(...) {}
```

Hystrix-kirjaston toimintovalikoima on laaja ja sen dokumentaatio on löydettävissä GitHub-palvelusta. Menetelmä toteutti tehokkaasti mikropalvelun vikasietoisuuden ja vakauden. Edelleen on mahdollista, että palvelu suorittaa virheen, josta se ei toivu. Käytännössä mikropalveluun kohdistetut REST-kyselyt eivät kuitenkaan aiheuta virhettä joka pysäyttäisi koko sovelluksen toiminnan.

4.7 Tiedonhakuprosessi

Kyselyn suorittaminen ketjutettiin Feign-kirjaston avulla mikropalvelun rajapinnasta toiseen. Lääketiedon ydinpalvelu (service-medic-core) aloitti tapahtumaketjun, jossa kysely eteni hakupalvelun (service-medic-search) ja resurssipalvelun (service-resources) kautta lääketietokantaan. Kyselyn tulokset palautuivat samaa reittiä takaisin. Tiedonhakuprosessin jokaisessa vaiheessa selvitettiin Eureka-havaitsemispalvelusta seuraavan mikropalvelun saatavuus ja verkkosijainti. Resurssipalvelu suoritti lisäksi käyttöoikeuksien tarkistuksen auktorisointipalveluun (service-auth), jotta transaktion suorittamiseen saatiin valtuutus. Tiedonhakuprosessi on esitetty kokonaisuudessaan prosessikaaviossa kuvassa 7.



Kuva 7. Lääketiedon hakuprosessi

Lääketietokantaan lisättiin toimintojen testausta varten satunnaisia lääketuotetietoja. Tiedonhakuprosessi käynnistettiin lähettämällä Postman-apuohjelmalla GET-kysely Lääketietopalvelun REST-rajapinnan päätepisteeseen /search/{term}, jossa term oli tietojen etsimiseen käytettävä hakusana polkuparametrina välitettynä. Testikyselyn kohdeosoite oli kehitysympäristössä localhost:8001/search/burana. Lääketietokannasta haettiin siis lääkeaineet tai -tuotteet, joiden nimessä esiintyy ”burana”. Kyselyn mukaan määriteltiin myös auktorisointipalvelun tarvitsemat tunnistetiedot. Kysely eteni vaiheittain mikropalvelujen läpi kuvassa 7 esitetyn tiedonhakuprosessin mukaisesti. Vastauksena saatiin status ”200 OK” sekä odotusten mukaiset hakutulokset JSON-muodossa.


```
[
   {
      "medicineID": 1,
      "medicineName": "BURANA",
      "concentration": "400 mg",
      "productSize": "30 tabl.",
      "manufacturer": "Orion Pharma"
   },
   {
      "medicineID": 2,
      "medicineName": "BURANA-CAPS",
      "concentration": "400 mg",
      "productSize": "20 tabl.",
      "manufacturer": "Orion Pharma"
   }
]
```

Mikropalvelupohjaisessa tiedonhakuprosessissa kysely eteni vaiheittain useiden mikropalvelujen ja älykkäiden hallintamenetelmien avulla aina lääketietokantaan asti. Kyselyn suoritus oli kuitenkin nopea. Virhetilanteessa Hystrix-piirikatkaisia poistaa toimimattoman palveluinstanssin saatavilta ja kysely ohjataan rinnakkaisinstanssin kautta eteenpäin. Kehitetyn tiedonhakuprosessin mikropalvelut voitaisi ottaa tuotannossa käyttöön myös hajautetusti eri palvelimille. Tällöin virtuaalipalvelimia voidaan mikropalveluineen skaalata useiksi instansseiksi, ja hakuprosessin toimintavarmuus olisi korkeampi kuin paikallisessa kehitysympäristössä. Verkkoviivettä kertyisi hakuprosessin läpiviennissä kuitenkin enemmän. Lääketietopalvelun jatkokehitys on tämän jälkeen ketterää. Palveluun voidaan lisätä muita toiminnallisuuksia ilman tarvetta hallintajärjestelmän muutoksiin. Mikropalvelut saavat toistensa ominaisuuksia käyttöönsä rajapintojen kautta pelkän palvelun nimen perusteella. Mikäli mikropalvelujen suorituksenaikaisiin ympäristömuuttujiin ja määrityksiin tarvitaan muutoksia, ne voidaan tehdä keskitetysti konfiguraatiopalvelun kautta.

4.8 Kehitystyön jälkiarviointi

Hajautetun mikropalvelujärjestelmän kehittäminen oli vahvan ohjelmistokehyksen ja toimivien ohjelmakirjastojen ansiosta selkeää ja suoraviivaista. Järjestelmän hallinnan ja vakauden toteuttavat ominaisuudet pystyttiin ottamaan käyttöön yksi kerrallaan ohjelmakoodin ja konfiguraation määrityksillä. Ongelmia toimintojen toteutuksessa oli vähän, ja ne pääsääntöisesti liittyivät ohjelmakirjastojen versioiden välisiin konflikteihin. Kun ominaisuus oli toteutettu, sitä ei tarvinnut enää muutella muiden ominaisuuksien takia. Haasteellista, sitä vastoin, oli tarvittavan dokumentaation löytäminen API-kirjastojen soveltamiseksi. Toiset kirjastot oli kattavasti dokumentoitu GitHub- ja Spring.io-sivustoille, toiset vaativat tiedonjyvästen etsimistä internetistä. Projektissa opittiin, että dokumentaatio on välttämätön resurssi valmiiden API-kirjastojen soveltamisessa. Työ on aloitettava dokumentaation pääkohtien ja vaatimusten tarkistamisella, jotta ohjelmakirjaston integroiminen järjestelmään olisi mahdollista. Projektin ohessa kertyi runsaasti tietoa muiden kehittäjien kohtaamista ongelmista yleisesti ohjelmakirjastojen käyttöönotossa. Usein ongelmat liittyivät välttämättömien määritysten puuttumiseen konfiguraatiossa. Tässä projektissa ominaisuuksien yhteensopivuusongelmat olivat vähäisiä, koska Spring-ohjelmistokehyksen alaiset projektit ovat jo valmiiksi integraatiotestattuja. Ominaisuuksien käyttöönotto annotaatioilla ja rajapinnoilla edisti merkittävästi kehitystyötä ja vähensi matalan tason ohjelmointityötä. Kirjastojen soveltaminen vaati kuitenkin Spring-ohjelmistokehyksen ja sen lainalaisuuksien hyvää hallintaa, jotta kirjastojen tarjoamat toiminnallisuudet määritettiin oikein Spring Boot -sovelluksen sisällä.

Ohjelmakirjastojen versiopäivitykset on automatisoitu Maven-ympäristönhallinnan kautta. Näin ollen, mikropalvelujen päivityspaketin kääntäminen ja julkaiseminen ovat nopea toimenpide kehitysympäristöstä käsin. Järjestelmän kasvaessa on harkittava tuotantoautomaatioiden käyttöönottoa, jotka huolehtivat sovellusten testaamisesta ja kääntämisestä tiukkojen käytäntöjen mukaisesti. Tässä projektissa tuotetulla järjestelmällä on hyvin vähän tarvetta yksikkötestaukselle. Toiminnallisuudet on toteutettu enimmäkseen annotaatioilla ja rajapinnoilla, joten matalan tason ohjelmakoodia on vähän. Sen sijaan, mikropalvelujen välistä toimintaa testattiin integraatiotestauksen avulla. Integraatiotestejä, samoin kuin yksikkötestejä, voidaan kirjoittaa Spring Boot -sovellusten testiympäristöön ketterästi JUnit- ja Mockito-kirjastojen avulla. Projektin mikropalvelujen perustason integraatiotestaus toteutettiin sovelluksen kontekstilatauksella. Testauksen käynnistyessä rakentuu palvelun oikea suorituksenaikainen ympäristö, joka luo yhteydet määritettyihin riippuvuuksiin. Mikäli palvelu on määritetty esimerkiksi konfiguraatiopalvelun client-rooliin, sen on testauksen käynnistyessä saatava toimiva yhteys kyseiseen palveluun. Muutoin testaus tekee virheen ja suoritus päättyy. Projektin kaikki mikropalvelut suoriutuivat virheettömästi testauksen kontekstilatauksesta, joten palvelujen suorituksenaikainen ympäristö on vakaa ja yhteydet muihin palveluihin toimivat ongelmitta.

Projekti tuotti sovellettujen ohjelmakirjastojen ominaisuuksilla vahvan, automatisoidun järjestelmän, jonka riippumattomien sovelluspalvelujen jatkokehitys on vaivatonta. Keskitetyn hallinnan palveluihin on mahdollisuus toteuttaa lukemattomia edistyneitä lisäominaisuuksia, jotka lisäävät palvelun vakautta ja varmistavat prosessoitavan tiedon eheyttä. Sovellusten toimintamallien kehittäminen erilaisiin virhetilanteisiin on mikropalveluarkkitehtuurin jatkokehityksessä erittäin tärkeää. Kun järjestelmän käyttötarkoitus ja erityisesti käyttötapaukset tarkentuvat kehitystyön myötä, tulisi varmistaa, että kaikki mikropalvelut selviävät virhetilanteista. Järjestelmän mikropalvelujen pitää reagoida virhetilanteisiin nopeasti ja estää virheen leviäminen muihin palveluihin. Tässä projektissa käytetty Hystrix-piirikatkaisija on vahva menetelmä mikropalvelun virhetilanteiden hallintaan. Hystrix sisältää kattavan koodikirjaston ohjelmallisesti hallittavien virhetilanteiden määrittämiseen. Saatavilla on myös lukuisia muita kirjastoja ja menetelmiä mikropalveluiden vikasietoisuuden parantamiseksi.

Keskitetyn hallinnan palvelut ovat mikropalvelupohjaisen järjestelmän tietoliikenteen pullonkauloja, joiden kuormittumista on vältettävä niin kuormanjakajien kuin skaalauksen avulla. Kaikkien mikropalvelujen tiedonvaihdon menetelmiä tulisi vahvistaa välimuistin tehokkaalla käytöllä. Tämä parantaa niin palvelujen toimintaa kuin tiedon eheyden säilymistä virhetilanteissa. On tärkeää muistaa, että projektissa toteutettu Eureka-havaitsemispalvelu ei suorita tietoliikenteen reititystä. Se ainoastaan välittää tietoja järjes-

telmän saatavilla olevista palveluista client-palveluille. Yksittäisten palvelujen kuormittumista voidaan helpottaa erilaisilla reititys- ja kuormanohjausmekanismeilla. Ribbon-kirjastolla voidaan toteuttaa client-puolen kuormanohjausta. Tällöin mikropalvelu kysyy havaitsemispalvelulta tietyn nimisen palvelun kaikkien instanssien tiedot ja päättää itse mihin palveluun kyselynsä kohdistaa. Varsinainen tietoliikenteen reitityspalvelu on myös mahdollista toteuttaa ohjelmallisesti ilman palvelinteknologioita. Zuul-kirjastolla voidaan toteuttaa niin sanottu API Gateway, joka ohjaa asiakassovellukselta tulevaa tietoliikennettä palveluinstansseihin. Perinteisesti näitä toteutetaan palvelemaan erilaisia järjestelmän ulkopuolisia client-sovelluksia, jotta käyttäjän laitteen tekniset ominaisuudet voidaan ottaa paremmin huomioon. Esimerkiksi mobiililaitteilta ja verkkoselaimilta tulevat kyselyt voidaan ohjata täysin eri palveluihin. Menetelmää voidaan käyttää hyvin myös mikropalvelujen välisen tiedonvaihdon ohjaukseen, kuten tietoturvaluvussa 4.5 Zuul-kirjastoa pohdittiin ratkaisuksi HTTP-protokollan kuljettamien tietojen välittämiseksi mikropalvelulta toiselle. Tässä projektissa tulisi parantaa reititystä ja kuormanjakoa erityisesti resurssi-, konfiguraatio- ja havaitsemispalveluihin, joihin kohdistuu kaikkien palvelujen kyselyjä. Järjestelmän kasvaessa palveluryppäille on perustettava omat hallintapalvelunsa. Esimerkiksi yksi havaitsemispalvelu voi palvella tehokkaasti vain rajattua mikropalvelujoukkoa.

Skaalaus, samoin kuin tuotantoautomaatio, vaativat vahvaa palvelin- ja tuotantoteknologioiden hallintaa. Jotta skaalautuvan järjestelmän tekninen infrastruktuuri olisi luotettava, on sen toteutus suunniteltava ohjelmistokehitystiimien ulkopuolella järjestelmäasiantuntijoiden toimesta. Mikropalveluarkkitehtuuri tukee nykyaikaisten DevOps-menetelmien (development and operations) soveltamista, jossa kehitystyötä toteutetaan yhdessä automatisoitujen tuotanto- ja valvontajärjestelmien tukemana. Tässä projektissa toteutetut järjestelmän hallintaominaisuudet voidaan toteuttaa myös palvelinpohjaisilla ratkaisuilla. Aina järjestelmäkehityksessä ei ole molempien osa-alueiden asiantuntemusta saatavilla tai se lisäisi merkittävästi kustannuksia. Siksi ohjelmistopohjaiset, joustavat ratkaisut ovat suositeltavampi vaihtoehto projektin alkuvaiheessa. Mikäli myöhemmin tehdään päätös järjestelmän vertikaaliseen skaalaukseen, eli suoritustehon kasvattamiseen, ohjelmistopohjaiseen toteutukseen voidaan ottaa mukaan myös palvelinpohjaisia ratkaisuja. Toisin päin siirtyminen tarkoittaa yleensä ohjelmistokehitystyön aloittamista alusta, koska palvelinympäristöstä ei ole juuri mitään siirrettävissä ohjelmatoteutukseen.

5 Pohdinta ja yhteenveto

Mikropalveluarkkitehtuuri on edistyksellinen kehitysmalli hajautettujen järjestelmien kehitykseen ja hallintaan. Sovelluksen jakaminen mikropalveluiksi helpottaa merkittävästi toimintojen kehittämistä erillään muusta järjestelmästä. Käytettävät teknologiat voidaan aina valita mikropalvelun suorittaman tehtävän perusteella. Mikropalveluarkkitehtuuri rohkaisee kokeilemaan uusia innovatiivisia menetelmiä palvelujen kehittämiseksi, koska kaikki kehitystyön riskit on rajattu mikropalvelun sisälle.

Selvitystyö ja toteutettu järjestelmäprosessi osoittivat, että toiminnallisten osien eristäminen omiksi mikropalveluiksi voi olla hyödyllistä sekä kehitystyön että hallittavuuden kannalta. Lisäksi mikropalveluarkkitehtuuri edellyttää huolellisesti suunniteltua hallintajärjestelmää. Kun hajautetulla järjestelmällä on vahvat mekanismit keskitettyyn konfigurointiin, palvelujen havaitsemiseen, viestiliikenteen ohjaukseen ja virhetilanteiden hallintaan, sovellusominaisuuksien kehitystyö on vapautettu arkkitehtuurin kompleksisesta toiminnasta. Hallintajärjestelmän toteuttamiseen on saatavilla runsaasti niin kaupallisia kuin avoimen lähdekoodin menetelmiä. Järjestelmää voidaan kehittää sekä palvelinpohjaisilla että ohjelmistopohjaisilla ratkaisuilla.

Mikropalvelupohjaisen järjestelmän hallintaan on vaivatonta löytää erilaisia tilanteisiin sopivia toteutustapoja. Järjestelmän vakautta parantavat mekanismit, kuten piirikatkaisija (circuit breaker), kuormanohjaaja (load balancer), havaitsemispalvelu (discovery service) ja lukuisat muut, ovat tunnettuja käsitteitä järjestelmäkehityksessä. Niiden hakeminen internetin hakukoneista tuottaa runsaasti tuloksia. Uusia menetelmiä voi rohkeasti esitellä projektissa hyödynnettäviksi, koska niiden käyttöönotto vähimmäiskonfiguraatiolla on usein nopeaa yhdelle mikropalvelulle. Teknologinen vapaus tarkoittaa kuitenkin myös suurta vastuuta. Koska mikropalveluarkkitehtuuri ei aseta juuri rajoitteita toteutustavoille, kehitystiimien on itse määriteltävä yhteiset säännöt kehitettävälle järjestelmälle.

Mikropalvelujen soveltaminen edellyttää laaja-alaisempaa teknologioiden hallintaa ja liiketoiminnan ymmärrystä kuin monoliittinen kehitystyö, koska tuotantojärjestelmät voivat koostua eri organisaatioiden kehittämistä ja ylläpitämistä palveluista. Kehittäjien on ymmärrettävä ohjelmistoteknologioiden lisäksi palvelin- ja verkkoteknologioita, tietoliikenneprotokollia sekä hajautetun järjestelmän komponenttien kontekstuaalista suhdetta sovellusalaan ja toimintalogiikkaan. Toisaalta mikropalveluarkkitehtuurissa yhdistyvät kaikki ohjelmistokehityksen ja teknologioiden vahvuudet, joilla saavutetaan vakaa pohja pitkän tähtäimen järjestelmäkehitykselle. Lisäksi mikropalvelut pysyvät vahvasti mukana teknologioiden evoluutiossa.

Opinnäytetyön teoria- ja toteutusosa keräsivät yhdessä suuren määrän hyödyllistä tietoa ja kokemusta sovellettavaksi työelämän ohjelmistoprojekteissa. Järjestelmäprosessiin toteutetut menetelmät ovat sovellettavissa myös muihin kuin mikropalvelupohjaisiin ratkaisuihin. Keskeisintä on tunnistaa, milloin toimintalogiikan osa kannattaa pitää itsenäisenä komponenttina ja milloin hallittavuus paranee toimintoja yhdistämällä. Opinnäytetyön tulokset motivoivat soveltamaan sekä mikropalveluarkkitehtuuria että yksittäisiä menetelmiä ohjelmistokehityksessä. Lukemattomat työtunnit lähdekirjallisuuden ja ohjelmakoodin parissa eivät menneet hukkaan. Kerätty kokemus on tuottanut runsaasti ideoita, miten esimerkiksi hakupalvelua voi laajentaa ulkoisiin palveluihin. Aihepiirin opiskelu ja kehitystyö jatkuvat tulevaisuudessakin. Tällä kokemuksella pystyn jo tuomaan esille hyödyllisiä näkökulmia, joilla ohjelmistokehityksen ongelmia ratkotaan mikropalveluilla.

Lähteet

Carnell, J. 2017. Spring Microservices in Action. 1. painos. Manning Publications Co.

Eberhard, W. 2017. Microservice: Flexible Software Architecture. 1. painos. Addison-Wesley.

Fowler, M. 2014. Microservices. Luettavissa: https://martinfowler.com/articles/microservices.html. Luettu: 19.9.2017.

Humble, J. & Farley, D. 2011. Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley.

Nadareishvili, I., Mitra, R., McLarty, M. & Amundsen, M. 2016. Microservice Architecture: Aligning Principles, Practices, and Culture. 1. uudistettu painos. O'Reilly Media, Inc.

Newman, S. 2015. Building Microservices: Designing Fine-Grained Systems. 1. uudistettu painos. O'Reilly Media, Inc.

Nygard, M. T. 2012. Release It! Design and Deploy Production-Ready Software. 1. uudistettu painos. Pragmatic Bookshelf.

Rajesh, RV. 2016. Spring Microservices: Build Scalable Microservices with Spring, Docker and Mesos. 1. painos. Packt Publishing Ltd.

Sharma, S. 2016. Mastering Microservices with Java. 1. painos. Packt Publishing Ltd.

Varanasi, B. & Belida, S. 2015. Spring REST. 1. painos. Apress.