Jorma Syrjä

MIKROPALVELUARKKITEHTUURIT

Tekniikka
2016

VAASAN AMMATTIKORKEAKOULU
Tietotekniikka

TIIVISTELMÄ

Tekijä	Jorma Syrjä
Opinnäytetyön nimi	Mikropalveluarkkitehtuurit
Vuosi	2016
Kieli	suomi
Sivumäärä	56
Ohjaaja	Martti Mustonen

Tämän opinnäytetyön tavoitteena oli tutkia mikropalvelurakennetta vaihtoehtona ohjelmistosuunnittelussa. Vertailtavaksi valittiin kolme eri kehitysympäristöä, joilla toteutettiin yksinkertainen Internet of Things –sovellus. Lopuksi selvitettiin, mikä vaihtoehdoista olisi paras olemassaolevan sovelluksen uudelleenrakentamiseen mikropalveluiksi tai uuden sovelluksen rakentamiseen ohjelmistokehittäjän näkökulmasta.

Mikropalveluarkkitehtuuri on ohjelmiston suunnittelussa käytetty arkkitehtuurimalli hajautetusta sovelluksesta, jossa sovelluskokonaisuus on jaettu itsenäisiksi prosesseiksi, usein eri palvelimille tai pilvipalveluun. Mikropalvelu eroaa perinteisestä monoliittisesta arkkitehtuurimallista erityisesti modulaarisuudessaan, koska mikropalvelusovellus on jaettu pieniin osiin, sitä voi myös hallita pieninä osina. Esimerkiksi päivitys ja vikatilanteet kohdistuvat vain tiettyyn osaan sovellusta, esim. kirjautumiseen verkkopalvelussa, jolloin muut osat toimivat normaalisti muista osista riippumatta. Mikropalvelumalli on ideana vanha, mutta sen perusteella toteutettuja sovelluksia on alettu kehittämään vasta viime vuosien aikana.

Vertailun kohteena oli kolme eri kehitysympäristöä, Spring Cloud, Microsoft Azure Service Fabric ja Lightbend Lagom. Jokaisella ympäristöllä toteutettiin samankaltainen pieni Internet of Things –sovellus, jolloin kävi ilmi mm. eri ympäristöjen eroavaisuudet mikropalveluiden toteutuksissa ja niiden ohjelmoinnissa. Lopulta päädyttiin sellaiseen lopputulokseen, että jokainen kehitysympäristö on potentiaalinen vaihtoehto sekä uuden sovelluksen kehittämiseen että vanhan sovelluksen uudelleenrakentamiseen, joka ympäristöllä on omat vahvuudet ja heikkoudet.

Avainsanat mikropalvelu, Internet of Things, ohjelmisto, kehitysympäristö

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Jorma Syrjä
Title	Microservices Frameworks
Year	2016
Language	Finnish
Pages	56
Name of Supervisor	Martti Mustonen

The goal of this thesis was to investigate microservice architecture as a substitute for the traditional monolithic approach for software development. Three of the most prominent microservice frameworks were selected, and for comparison a simple Internet of Things application was implemented with each framework. After comparing the frameworks, one of them was selected to be the most potential choice for developing new software or rewriting existing applications with microservices.

Microservice is an architecture model for software development, where an application is split into small independent components. The biggest difference a microservice approach has with a monolithic approach is the modularity of the application: when the application is split into smaller components, updates and errors affect only one small part of the application, for example login in a web interface. Component independence also means that a microservice approach is good for distributed cloud applications. Microservice is an old concept, but only few applications that have this architecture exist to this day. Only lately it has been gaining interest with the release of frameworks such as Lightbend Lagom and Service Fabric.

The three frameworks which were compared are Spring Cloud, Microsoft Azure Service Fabric and Lightbend Lagom. A small Internet of Things application was implemented with each framework. During implementation the differences of each framework were notable, especially when implementing and programming new services. The conclusion was that each framework has potential with its own strengths and weaknesses and it is possible to create advanced applications with all of them.

Keywords microservice, Internet of Things, application, framework

SISÄLLYS

KUVIO- JA TAULUKKOLUETTELO

TERMIT JA LYHENTEET

Internet of Things	Esineiden internet
Java	Oliopohjainen ohjelmointikieli
Scala	Java-ohjelmointikielen murre
WAR	Web Archive –tiedosto
.NET	Visual Studiossa käytettävä ohjelmointiympäristö
JAR	Java Archive –tiedosto
CQRS	Command Query Responsibility Segregation, eli komentoviestien käsittelyn vastuun erottaminen
IDE	Integrated Development Environment eli integroitu kehitysympäristö
XML	eXtensive Markup Language, merkkikieli
DHCP	Dynamic Host Configuration Protocol, protokolla, jolla jaetaan IP-osoitteita verkon laitteille
HTTP	Hypertext Transfer Protocol, merkkikielen välitysprotokolla
DDoS	Distributed Denial of Service eli hajauttettu palvelunestohyökkäys

REST	Representative State Transfer, tilanvalitykseen perustuva rajapintamalli
Apache Tomcat	Web-sovellusten ajotyökalu
Apache Cassandra	Hajautettu tietokantasovellus
Maven	Java-projektin hallintatyökalu
C#	Oliopohjainen ohjelmointikieli
Visual Basic	Yleiskäyttöinen ohjelmointikieli
TCP	Transmission Control Protocol
URI	Unified Resource Identifier, standardoitu resurssin tunniste
SBT	Simple Build Tool, projektinhallintatyökalu
API	Application Programming Interface eli rajapinta

1 JOHDANTO

Perinteisessä vesiputousmallisessa ohjelmistoprojektissa on aina määrittely-, suunnittelu-, toteutus-, testaus- ja käyttöönottovaiheet. Määrittelyvaiheessa pohditaan, mitä valmiissa ohjelmistossa tulee olla ja miten sen tulee toteuttaa tietyt vaatimukset. Suunnitteluvaiheessa sovellukselle luodaan sekä arkkitehtuurinen että looginen malli. Testausvaiheessa varmistetaan, että sovellus toimii määrittelyvaiheessa kirjoitetun toiminnallisen määritelmän mukaisesti. Lopulta siirrytään käyttöönottovaiheeseen, jossa sovellus asennetaan ja laitetaan käyttövalmiiksi asiakkaalle.

Tutkitaan tarkemmin suunnitteluvaiheen tuloksia, erityisesti sovelluksen arkkitehtuurista mallia. Miten sovellus otetaan käyttöön? Miten jatkuva integraatio toteutetaan, eli miten saadaan muutokset nopeasti tuotantoon? Vastauksena on luultavasti sovellus, josta luodaan yksi ajettava tiedosto. Tämä ajettava tiedosto, esimerkiksi websovelluksissa WAR-tiedosto, sisältää sovelluksen kaiken logiikan ja toiminnallisuuden. Tällaistä sovellusta sanotaan monoliittiseksi sovellukseksi.

Tässä opinnäytetyössä tutkitaan vaihtoehtoista rakennetta monoliitille, mikropalvelut. Mikropalvelumalli on arkkitehtuurinen malli, jossa sovellus jaetaan pieniin itsenäisiin komponentteihin, jotka sisältävät vain osan sovelluksen koko toiminnallisuudesta. Jokaisesta komponentista luodaan oma ajettava tiedosto, joka voidaan sijoittaa vaikka eri palvelimille kuin muut komponentit. Päivitykset hoituvat sujuvammin, koska vain päivitetyt komponentit joudutaan käynnistämään uudelleen. Komponenteista käytetään nimitystä palvelu ja palvelut muodostavat yhdessä sovelluksen. Mikro-etuliite palvelulle tulee siitä, että yksi palvelu hoitaa vain tiettyä toimintoa, esimerkiksi tietokantaan kirjoittamista.

Vertailun vuoksi valittiin kolme kehitysympäristöä, joilla toteutettiin samankaltainen Internet of Things –sovellus, joka ottaa vastaan mittausarvoja, tallentaa ne tietokantaan ja tarvittaessa hakee niitä sieltä. Kehitysympäristöinä toimivat Javalla toteutettu **Spring Cloud**, Microsoftin pilvipalveluun Azureen .NETillä toteutettu **Service Fabric** sekä Javan ja Scalan sekoitus **Lightbend Lagom**.

2 MIKROPALVELUT

2.1 Mitä mikropalvelut ovat?

Mikropalvelu on käsite, joka viittaa tapaan suunnitella sovelluksen arkkitehtuuri. Sovelluksen arkkitehtuurilla tarkoitetaan sitä, miten lähdekoodi on jaettu ja miten toiminnalliset kerrokset, kuten käyttöliittymä ja data toteutetaan. Mikropalveluarkkitehtuurissa sovellus jaetaan toiminnallisiin osiin eli komponentteihin, jotka suorittavat yhtä ja vain yhtä toimintoa. Nämä osat ovat itsenäisiä ja niitä voidaan jakaa eri palvelimille. Tarpeen vaatiessa yhtä komponenttia voidaan käynnistää useampi prosessi taakan jakamiseksi ja järjestelmän vakauden turvaamiseksi.

Mikropalveluarkkitehtuurissa sovelluksen yhtä komponenttia sanotaan myös **palveluksi**. Yhdestä palvelusta voi olla samanaikaisesti useampi **instanssi** eli ilmentymä, ts. prosessi. Mikropalveluarkkitehtuuri pyörii laajalti näiden instanssien manipuloinnin ympärillä.

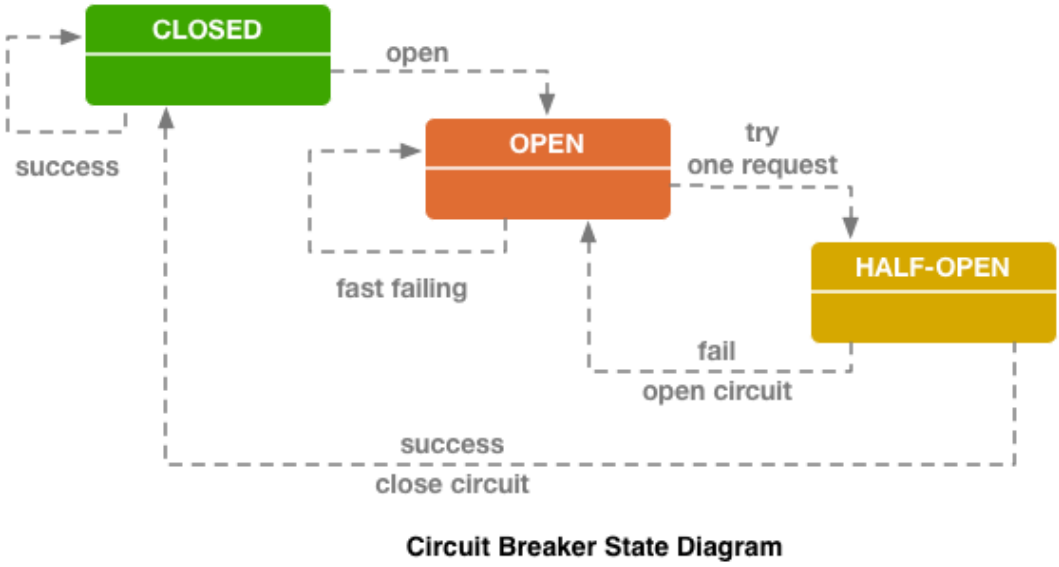
Vaikka mikropalveluarkkitehtuurille ei ole tarkkaa määritelmää, esimerkiksi komponenttien laajuuteen liittyen, on olemassa tiettyjä ominaisuuksia, jotka yhdistävät kaikkia mikropalveluarkkitehtuureja; varmuus, skaalautuvuus ja hajautettu ympäristö.

2.1.1 Varmuus

Varmuudella viitataan sovelluksen vakauteen. Mikropalvelut suunnitellaan varmoiksi, mikropalvelusovelluksella tulisi olla aina valvoja, joka pitää kirjaa eri komponenteista ja käynnistää lisää instansseja niistä, jotka vaikuttavat hitailta tai kaatuneilta. Mikropalvelusovelluksen palveluiden ja käyttäjien välissä on reunapalvelin, jossa liikennettä ohjataan katkaisimen (Circuit Breaker) kautta.

Katkaisin lisää vakautta mikropalveluarkkitehtuurissa. Se pysäyttää pyynnöt, jotka kohdistuvat toimimattomille palveluille ja palauttaa pyynnön lähettäjälle ennaltamääritetyn virheviestin. Katkaisimella on kolme tilaa, auki (Open), kiinni (Closed) ja puoliksi auki (Half-Open). Kun virheitä ei havaita, katkaisin on kiinni.

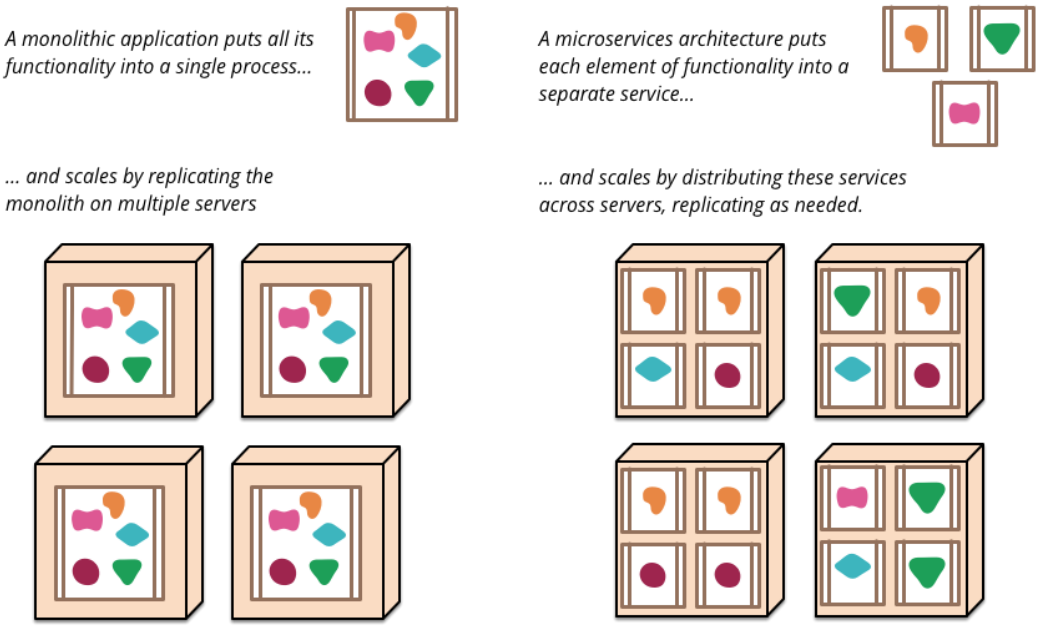
Virheiden kasautuessa katkaisin avataan ja tietyn ajan jälkeen muutetaan puoliksi aukinaiseen tilaan, jossa yksi pyyntö päästetään palvelulle. Jos pyyntö onnistuu, katkaisin suljetaan ja pyynnöt ohjautuvat jälleen normaalisti (**Kuvio 1**.). /8/



Kuvio 1. Katkaisimen toimintaperiaate.

2.1.2 Skaalautuvuus

Skaalautuvuudella viitataan siihen miten sovelluksen resursseja voidaan kasvattaa ja vähentää tarpeen mukaan. Mikropalveluiden skaalaus on helppoa, selvitetään, mikä toiminnallisuus vaatii lisää resursseja ja käynnistetään siitä vastuussa olevalle komponentille uusi prosessi. Tämä on tehokkaampaa kuin monoliittisessa sovelluksessa, jossa ainut vaihtoehto on kopioida koko sovellus (**Kuvio 2.**). /2/



Kuvio 2. Monoliitti- ja mikropalvelusovelluksen skaalaus. /2/

2.1.3 Hajautettu ympäristö

Hajautetulla ympäristöllä viitataan pilvipalveluun tai monen palvelimen verkkoon. Mikropalvelut on luotu hajautettuja sovelluksia varten, sillä komponentteja voidaan jakaa mielivaltaisesti eri palvelimille tai pilvipalveluun. Näiden palveluiden yhdistämisestä keskenään pitää huolen instanssipalvelin, johon jokainen konponentti rekisteröi itsensä käynnistyessään. Kun yksi komponentti keskustelee toisen komponentin kanssa, ne löytävät toisensa instanssipalvelimen kautta. Tämän ansiosta instanssipalvelin voi suorittaa suuren osan kuorman jakamisesta.

Sovelluksen käyttäjät ja sen tarjoamat palvelut yhdistää reunapalvelin, joka toimii samalla periaatteella kuin reititin. Yleisin tapa selvittää käyttäjän haluama palvelu on tarkastella pyynnön polkua, esimerkiksi /api/devices/. Reunapalvelin yhdistää tämän polun tiettyyn palveluun ja ohjaa käyttäjän pyynnön eteenpäin.

2.2 Mikropalvelut vs. monoliitti

Mikropalveluarkkitehtuurin vastakohta on monoliitti, jossa sovellus rakennetaan yhden prosessin ympärille. Tuloksena syntyy yksi paketti, esimerkiksi JAR- tai

WAR-tiedosto, joka sisältää sovelluksen kaiken logiikan. Tämä paketti on itsenäinen kokonaisuus ja valmiina käyttöönottoa varten.

Monoliittinen arkkitehtuuri on perinteinen tapa suunnitella sovellus. Vaikka kaikki logiikka ja toiminnallisuus on lopullisessa tuotteessa samassa paketissa, kehitysvaiheessa sovellus voidaan jakaa useampaan kerrokseen, kuten Model-View-Controller –mallin mukaisesti. Näiden kerrosten sisällä voidaan tehdä jakoja pienempiin osiin, jolloin kehittäjien on helpompi jakaa työ keskenään osaamisen perusteella. Jos kerran monoliittisen sovelluksen jakaminen osiin on helppoa ja mahdollista, miksi pitäisi edes harkita mikropalvelumallia?

Monoliittisen sovelluksen tuomat ongelmat kasvavat samaa tahtia kuin sovellus itse. Suuri määrä koodirivejä paisuttaa ajettavaa ohjelmaa ja mahdollisten virheiden todennäköisyys kasvaa rivi riviltä. Päivitysvaiheessa koko ohjelma kaadetaan, vaihdetaan uuteen ja käynnistetään uudelleen. Etenkin suurten monoliittisten sovellusten käyttäjille tämä voi tarkoittaa pitkää toimimattomuuden ajanjaksoa.

Ylläpidon lisäksi monoliittisten sovellusten toinen suuri ongelma ilmenee skaalauksessa. Mitä tehdään, jos asiakkaita on paljon ja kuormitus kasvaa? Koko sovellus kopioidaan toiselle palvelimelle ja mahdolliset palomuurien asetukset ja reititykset kloonataan uutta palvelinta varten. Pahimmassa tapauksessa myös tietokanta joudutaan kloonaamaan. Joka tapauksessa monoliittisen sovelluksen skaalaus tarkoittaa aina sitä, että joudutaan lisäämän resursseja toista tai useampaa samanlaista sovellusta varten.

Kahden edellisen kappaleen mainitsemat ongelmat ylläpidosta ja skaalauksesta ovat mikropalvelumalliselle sovellukselle kuin ilmaa, sillä mikropalvelut on luotu näiden ongelmien ratkaisemiseksi. Päivitysten suorittaminen tarkoittaa ainoastaan yhden komponentin, ts. palvelun, alasajamista ja uudelleenkäynnistämistä, koska mikropalvelusovellus koostuu komponenteista. Komponentti on itsenäinen prosessi, joka suorittaa vain tiettyä tehtävää. Komponentit ovat suora vastaus myös skaalausongelmalle itsenäisyytensä ansiosta, sillä tarvittavia komponentteja voidaan tarvittaessa käynnistää lisää ja ajaa alas kuormituksen vaihdellessa.

Esimerkiksi YouTuben kaltainen videopalvelu voisi lisätä videon välityspalveluita sen mukaan, kuinka paljon käyttäjiä on milläkin hetkellä kirjautuneena. /1;2/

2.3 Palveluidenväinen kommunikointi

Mikropalvelut eivät ole kuitenkaan täydellisiä – hajauttamisesta aiheutuu myös ongelmia.

Palveluiden välisen kommunikoinnin toteuttaminen voi olla haastavaa. Siinä käytetyt viestit ovat häiriölle alttiita, koska ne joudutaan usein kuljettamaan verkon yli. Viestit saattavat kadota tai turmeltua matkan aikana, verkkoyhteydessä saattaa ilmetä katkoksia tai topologia voi muuttua kesken viestittelyn. Pahimmassa tapauksessa hidas komponentti voi aiheuttaa kutsujalleen aikakatkaisun. /3/

2.3.1 Sovelluksen tilan yhtenäisyys

Koska komponentit ovat itsenäisiä, tietoja sovelluksen tilasta on vaikeaa pitää yhtenäisenä komponenttien kesken. Tämä johtaa siihen, että sovelluksen eri komponenteissa tapahtuvat tilamuutokset eivät näy muille komponenteille. Esimerkiksi yhden komponentin päivityksen aikana toiset komponentit, jotka mahdollisesti kutsuvat sitä, eivät ymmärrä päivitystilaa ja tulkitsevat sen virhetilanteeksi. /3/

2.3.2 Ylläpito

Mikropalvelusovelluksen ylläpidon vaikeus on varsinkin kokeneille ohjelmistokehittäjille pääsyy monoliittissa pysymiseen. Suuressa mikropalvelusovelluksessa palveluita on paljon ja niitä kaikkia ylläpidetään erikseen. Päivitykset ja monitorointi tapahtuu yhden palvelun joka instanssille, mikä voi tarkoittaa sitä, että yksi pieni muutos koodissa johtaa 100:aan eri prosessin uudelleenkäynnistämiseen eri palvelimilla. /3/

2.4 Mikropalvelut vs. monoliitti – yhteenveto

Arkkitehtuurin valinta ei ole itsestäänselvä asia, sillä kumpikaan näistä kahdesta vaihtoehdoista ei sulje pois toista. Monoliitti on perinteinen tapa kehittää sovellus,

mikä on todettu toimivaksi ties kuinka monta kertaa toisin kuin mikropalvelut. Toisaalta pilvipohjaisten sovellusten yleistyminen johtaa siihen, että mikropalvelumalli saattaa herättää kiinnostusta sellaisissa kehittäjissä, jotka ovat aikeissa tehdä ensimmäisen sovelluksensa. Uusien sovellusten kehittäjien kiinnostus saattaa olla ainut mahdollisuus mikropalveluarkkitehtuurille loistaa, sillä vanhojen monoliittisten sovellusten uudelleenkirjoittaminen mikropalveluiksi on kallista, aikaavievää ja riskialtista, koska monet kehitysympäristöt ovat vielä uusia ja epäkypsiä.

3 TESTISOVELLUKSEN MÄÄRITTELY

Jotta kehitysympäristöjen vertailussa olisi jotain järkeä, toteutetaan jokaisella niistä sovellus saman määritelmän pohjalta.

3.1 Vaatimukset

Testisovelluksen tulee täyttää vaatimukset prioriteetilla 1 ja jos kehitysympäristö tarjoaa omalaatuisen toteutustavan vaatimukselle, tulee testisovelluksen täyttää vaatimukset prioriteetilla 2 (**Taulukko 1.**).

Taulukko 1. Testisovelluksen vaatimukset.

Vaatimus	Prioriteetti
Sovellus vastaanottaa mittaustietoja	1
Sovellus vastaanottaa useita kutsuja eri käyttäjiltä samanaikaisesti	1
Sovellus vastaanottaa mittaustietojen kyselyitä ja hakee mittaustietoja tietokannasta	1
Sovelluksen tietokanta on toteutettu hajautetulla ja pilvipalveluvalmiilla ratkaisulla	1
Sovelluksen tietokantatoteutuksissa käytetään CQRS-rakennetta	1
Käyttäjä voi rekisteröidä laitteen ja hakea laitteen mittausarvoja sekä niiden keskiarvoja	2

3.1.1 CQRS-rakenne

CQRS tulee sanoista Command and Query Responsibility Segregation. Se on ohjelmistotuotannossa suunnittelumalli, jossa sovelluksen luku- ja kirjoitustoimintojen toteutukset erotetaan. Testisovelluksien kannalta se tarkoittaa sitä, että mikropalveluarkkitehtuurin mukaisesti luku- ja kirjoitustoimenpiteet tulee erottaa eri palveluiksi. /22/

3.2 Käytetyt tekniikat

IDE:nä Java-sovelluksia varten käytettiin Eclipse Mars.2 Release (4.5.2) ja .NET-sovellusta varten käytettiin Visual Studio 2015 Community.

Jokaisen testisovelluksen toteuttamiseen käytettiin ainoastaan kehitysympäristön tarjoamia kirjastoja.

4 SPRING, SPRING BOOT JA SPRING CLOUD

4.1 Spring

Spring on suosittu kehys Java-sovelluskehittäjien parissa. Se on kevyt ja helppokäyttöinen hallintatyökalu, jolla kehittäjät voivat hallita ohjelmistojensa rajapintoja ja toteutuksia. Kehysmäisyys takaa sen, että sen lisääminen ja poistaminen sovelluksesta on kehittäjälle helppoa, sillä Spring on sovelluksen ympärillä sen sijaan, että sovellus olisi rakennettu Springin ympärille. Tätä kutsutaan Inversion of Control –periaatteeksi.

4.1.1 Dependency Injection

Dependency injection on ohjelmistotekniikassa suunnittelumalli, jossa tarvittava riippuvuus (dependency) syötetään (inject, injection) sitä tarvitsevalle palvelulle eli objektille. Olio-ohjelmoinnissa tämä tarkoittaa sitä, että olio alustetaan käynnistyksen yhteydessä sille määritetyillä parametreilla.

Spring tarjoaa kehittäjälle mahdollisuuden määritellä objektit ja niille kuuluvat riippuvuudet XML-kartassa. Karttaan merkitään objektien nimet ja niiden alustusparametrit. Spring-sovelluksen käynnistyessä kartta ladataan muistiin ja siinä mainitut objektit alustetaan.

Springin tuoma hyöty on juuri kyseisessä XML-kartassa, johon tehdyt muutokset tulevat voimaan kaikkialla sovelluksessa, eikä kehittäjän tarvitse manuaalisesti käydä koodia läpi alustusparametreja vaihtaen. /4;5/

4.2 Spring Boot

Spring Boot on valinnainen lisäkomponentti Spring-sovellusten kehittäjille. Sen avulla kehittäjä voi paketoida sovelluksensa helposti itsenäiseksi kokonaisuudeksi, esimerkiksi JAR-paketiksi. Luotu paketti sisältää sovelluksen kaiken logiikan ja on valmis ajettavaksi.

Spring Bootin voi lisätä Spring-projektiinsa suoraan Mavenista tai komentoriviltä. /5/

4.3 Spring Cloud

Spring Cloud on kokoelma kirjastoja, joiden avulla on mahdollista luoda hajautettuja sovelluksia. Siinä on mukana muun muassa instanssi- ja reitityspalvelimet, kuorman jakaja ja hajautettu viestijärjestelmä. Spring Cloudia varten kehittäjän täytyy tehdä projektistansa Spring Boot –projekti.

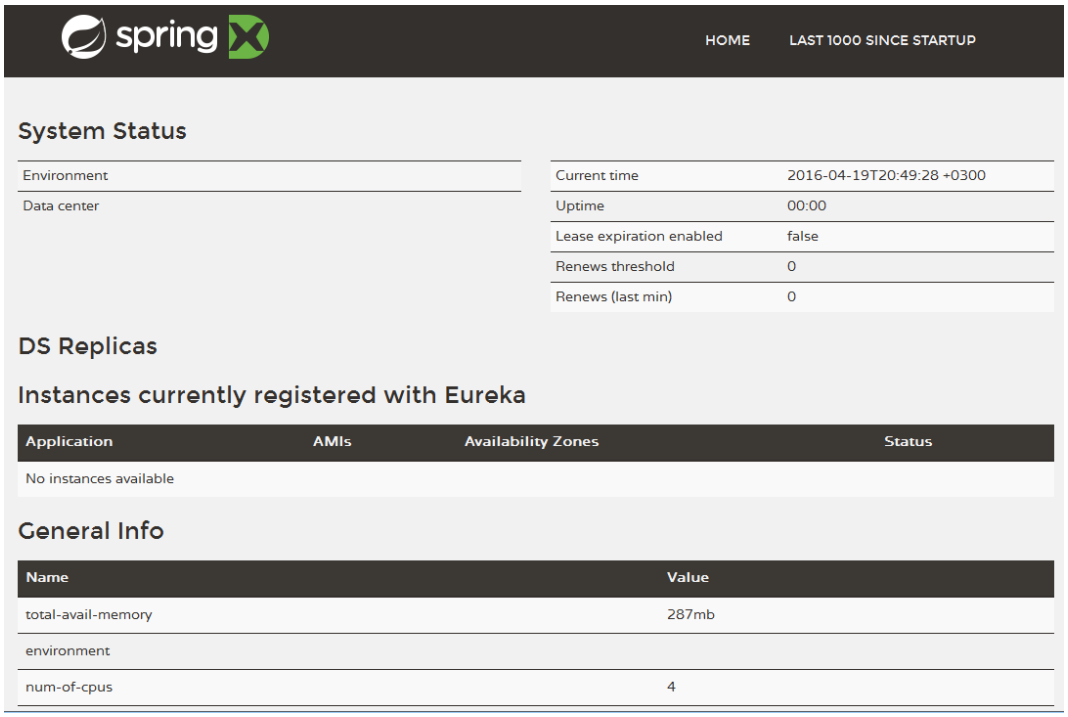
4.4 Mikropalvelut Springillä

Mikropalvelusovelluksen tekemiseen Springillä tarvitaan kaikkia edellämainittuja komponentteja. Spring alustaa Spring Cloudin tarjoamat kirjastot ja Spring Bootilla luodaan palvelu, jota voidaan ajaa itsenäisenä prosessina, eli toisinsanoen yhtenä mikropalvelun komponenttina. /9/

4.4.1 Netflix Stack

Netflix on yksi suurimmista Spring-mikropalveluiden käyttäjistä ja sen kehittäjät ovat julkaisseet omia sovelluksiaan vapaana lähdekoodina. Näitä sovelluksia kutsutaan Netflix Stackiksi ja ne hoitavat mikropalveluarkkitehtuurin hallinnolliset tehtävät, minkä ansiosta uuden kehittäjän täytyy toteuttaa ainoastaan oman sovelluksensa palvelut.

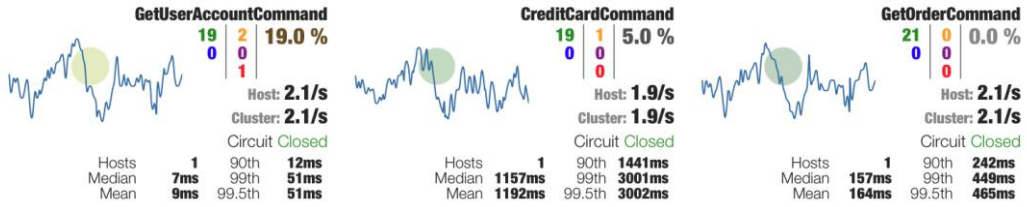
Eureka on tärkein sovellus Netflix Stackissä. Se on itsenäinen instanssipalvelin, joka pitää kirjaa kaikista siihen rekisteröityneistä instansseista. Eurekan avulla ylläpitäjät voivat pitää silmällä eri palveluja ja niiden instanssien lukumääriä (**Kuvio 3.)**. Eurekan toiminta instanssipalvelina on aika pitkälti samanlaista kuin DHCP-palvelimen toiminta, palvelun instanssi rekisteröi itsensä Eurekalle ja saa määräajan voimassa olevan suhteen. Suhteen loputtua instanssin tulee rekisteröidä itsensä uudelleen. Jos uudelleenrekisteröintiä ei tapahdu, Eureka poistaa instanssin saatavilla olevien listalta.



Kuvio 3. Eureka Dashboard.

Zuul on sovelluksen toiminnallisuuden kannalta valinnainen mutta tärkeä sovellus, joka toimii reitittimenä käyttäjien ja sovelluksen välissä. Zuul on reunapalvelin, joka pystyy ohjaamaan liikennettä oikeisiin palveluihin käyttämällä Eurekalta saatua instanssirekisteriä. Zuulin ansiosta sovellus tarvitsee ainoastaan yhden avoimen reitin palomuurissa ulkopäin tulevaa liikennettä varten sen sijaan, että jokaista ulkoapäin tulevaa liikennettä vastaanottavalle palvelulle avattaisiin oma reitti. Zuul pystyy myös suodattamaan liikennettä käyttämällä ohjelmoituja sääntöjä. Esimerkkinä voisi olla sääntö, joka estää liian suuren HTTP-pyynnön käsittelyn rajoittaen DDoS-hyökkäysten tehokkuutta.

Hystrix on palveluiden tarkkailuun ja virheensietoon kehitetty komponentti, jonka lisääminen palveluun tuottaa tietovirran ja katkaisijan toiminnallisuuden. Tietovirta voidaan ohjata Hystrix Dashboardiin, jossa näkyy palvelun tarjoamat metodit ja niiden kuormitus- ja virhetilastot (**Kuvio 4.)**.Useiden Hystrix-tietovirtojen yhdistämiseen on olemassa **Turbine**, joka tuottaa yhden Turbine-tietovirran. Tämä Turbine-tietovirta voi pitää sisällään esimerkiksi mikropalvelusovelluksen kaikkien palveluiden kuormitus- ja virhetilastot. /8/

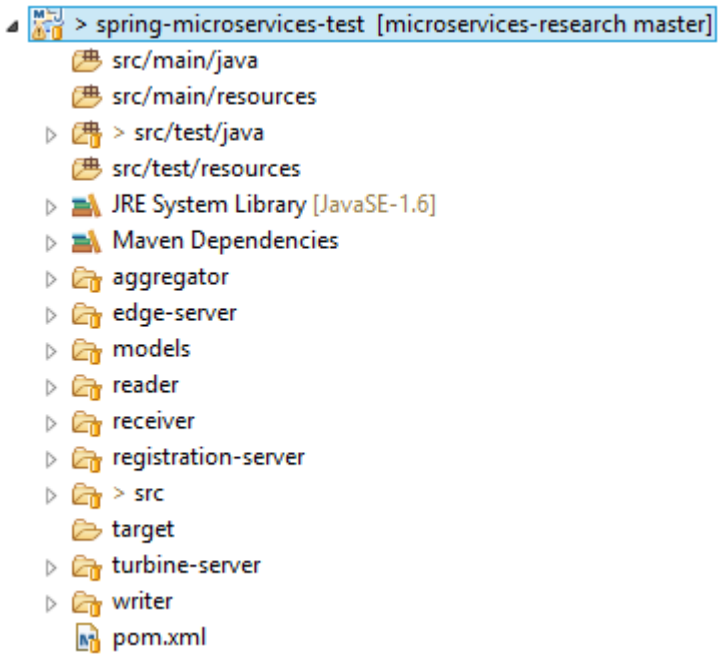


Kuvio 4. Hystrix Dashboard, kuormitus- ja virhetilastot. /7/

4.5 Testisovelluksen toteutus

Mikropalveluiden toteuttaminen Springillä tapahtuu paljolti samalla tavalla kuin monoliittisen sovelluksen eri toiminnallisuuksien toteuttaminen. Sen sijaan, että projekti jaettaisiin useampaan pakettiin, kuten Model-Viev-Controller –mallissa, jokaiselle toiminnallisuudelle, kuten esimerkiksi REST-rajapinnalle tai tietokannan lukijalle, luodaan oma projekti.

Hyvä käytäntö on luoda Maven-projekteja, joista yksi on ns. emoprojekti, joka pitää sisällään kaikki mikropalvelusovelluksen palvelut (**Kuvio 5.)**. Tämä testisovellus kehitettiin käyttämällä Eclipseä.



Kuvio 5. Spring-testisovelluksen Maven-emoprojekti, jolla on lapsiprojekteja.

Emoprojektin pom.xml –tiedostossa palvelut ovat emoprojektin moduuleja (**Kuvio 6.)**. Tämä tarkoittaa sitä, että kun emoprojekti rakennetaan, moduulit rakennetaan samanaikaisesti. Tämä helpottaa esimerkiksi sovelluksen julkaisua.

```
<modules>
    <module>reader</module>
    <module>writer</module>
    <module>receiver</module>

    <module>models</module>
    <module>registration-server</module>
    <module>edge-server</module>

    <module>composite-reader</module>
    <module>turbine-server</module>
</modules>
```

Kuvio 6. Spring-testisovelluksen palvelut moduuleina.

4.5.1 Uuden palvelun lisääminen

Palvelun lisääminen tapahtuu luomalla uusi moduuli emoprojektille. Palvelu koostuu usein kolmesta Java-luokasta, jotka ovat konfiguraatio- tai palvelinluokka, ohjaajaluokka ja palveluluokka.

Konfiguraatio- tai palvelinluokka on palvelun lähtöpiste. Minimissään se voi koostua pelkästä main-metodista, joka käynnistää palvelun Spring Boot –sovelluksena. Tähän luokkaan syötetään Springin avulla Spring Cloud –riippuvuudet. Syöttö tapahtuu lisäämällä annotaatioita luokan nimen yläpuolelle (**Kuvio 7.)**.

```
@EnableDiscoveryClient
@SpringBootApplication
@ComponentScan(useDefaultFilters = false)
@EnableHystrix
/**
 * Wrapper class for data receiver service.
 * @author jormasyrja
 *
 */
public class ReceiverServer {
```

Kuvio 7. Spring-testisovelluksen Receiver-palvelun konfiguraatioluokka.

@EnableDiscoveryClient –annotaatio rekisteröi palvelun Eurekan instanssirekisteriin.

@SpringBootApplication –annotaatio merkitsee luokan käynnistettäväksi Spring Boot –sovelluksena.

@ComponentScan –annotaatio estää automaattisen Spring Cloud –jakelun komponenttien skannauksen ja lisäämisen muualta kuin tästä luokasta.

@EnableHystrix –annotaatio lisää palveluun Hystrixin kuormitus- ja virhetilastot sisältävän tietovirran sekä mahdollisuuden käyttää katkaisijatoimintoja.

Ohjaajaluokka hoitaa viestien vastaanottamisen. Se voi olla esimerkiksi REST-rajapinta, joka vastaanottaa HTTP-pyyntöjä käyttäjiltä (**Kuvio 8.)**.

```
@HystrixCommand(fallbackMethod = "getRelayFallback")
@RequestMapping(value = "/receive", consumes = {
        MediaType.APPLICATION_JSON_VALUE }, method = RequestMethod.POST)
public ResponseEntity<?> receive(@RequestBody APIMessage msg){
    if(msg == null || msg.getName() == null || msg.getValue() == null)
        return new ResponseEntity<Object>(HttpStatus.BAD_REQUEST);

    log.info("APIMessage received: " + msg.toString());
    return recSvc.receive(msg);
}

public ResponseEntity<?> getRelayFallback(@RequestBody APIMessage msg){
    return new ResponseEntity<Object>(HttpStatus.SERVICE_UNAVAILABLE);
}
```

Kuvio 8. REST-metodi Spring-testisovelluksen Receiver-palvelun ohjausluokassa.

Palveluluokassa toteutetaan palvelun toiminto. Toiminto voi olla esimerkiksi mittausarvon vastaanottaminen tai sen tallentaminen tietokantaan.

Palvelulla on oltava myös asetukset Spring Bootia varten. Asetuksissa määritetään muun muassa palvelun nimi, instanssitunniste sekä Eureka-palvelimen osoite (**Kuvio 9.)**.

```
spring:
  application:
     name: receiver-service

# Discovery Server Access
eureka:
  instance:
    leaseRenewalIntervalInSeconds: 10
    metadataMap:
      instanceId: ${vcap.application.instance_id:${spring.a
      cluster: RECEIVER-SERVICE
  client:
    serviceUrl:
      defaultZone: http://localhost:1111/eureka/

# HTTP Server
server:
  port: 0   # HTTP (Tomcat) port
```

Kuvio 9. Spring-testisovelluksen Receiver-palvelun Spring Boot -asetukset.

4.5.2 Palveluidenvälinen viestintä

Spring Cloud tarjoaa valmiita toteutuksia viestien lähettämiseen muun muassa REST-rajapintaa tai viestijonoa käyttäen. Testisovelluksessa käytetään REST-rajapintoja RestTemplate-luokan kautta. RestTemplatelle syötetään halutun palvelun nimi ja REST-kutsun polku. RestTemplate hakee instanssin kohdepalvelusta Eurekan instanssirekisteristä kuorman jaon kautta.

4.5.3 Eureka, Zuul ja Turbine testisovelluksessa

Eureka, Zuul ja Turbine pystyvät toimimaan itsenäisinä palvelimina. Niitä varten on mahdollista tehdä oma moduuli emoprojektin alle, mutta myös valmiiksi ajettavat WAR-paketit ovat saatavilla. WAR-paketin ajaminen onnistuu esimerkiksi Apache Tomcatissa.

4.6 Testisovelluksen yhteenveto

Testisovelluksessa on 4 palvelua (**Taulukko 2.**) sekä erilliset palvelimet Eurekalle, Zuulille ja Turbinelle (**Kuvio 10)**. Tietokantana toimii Apache Cassandra.

Receiver-palvelu vastaanottaa mittausarvoja REST-rajapinnan kautta ja välittää ne kirjoitettavaksi tietokantaan *Data-write* –palvelulle.

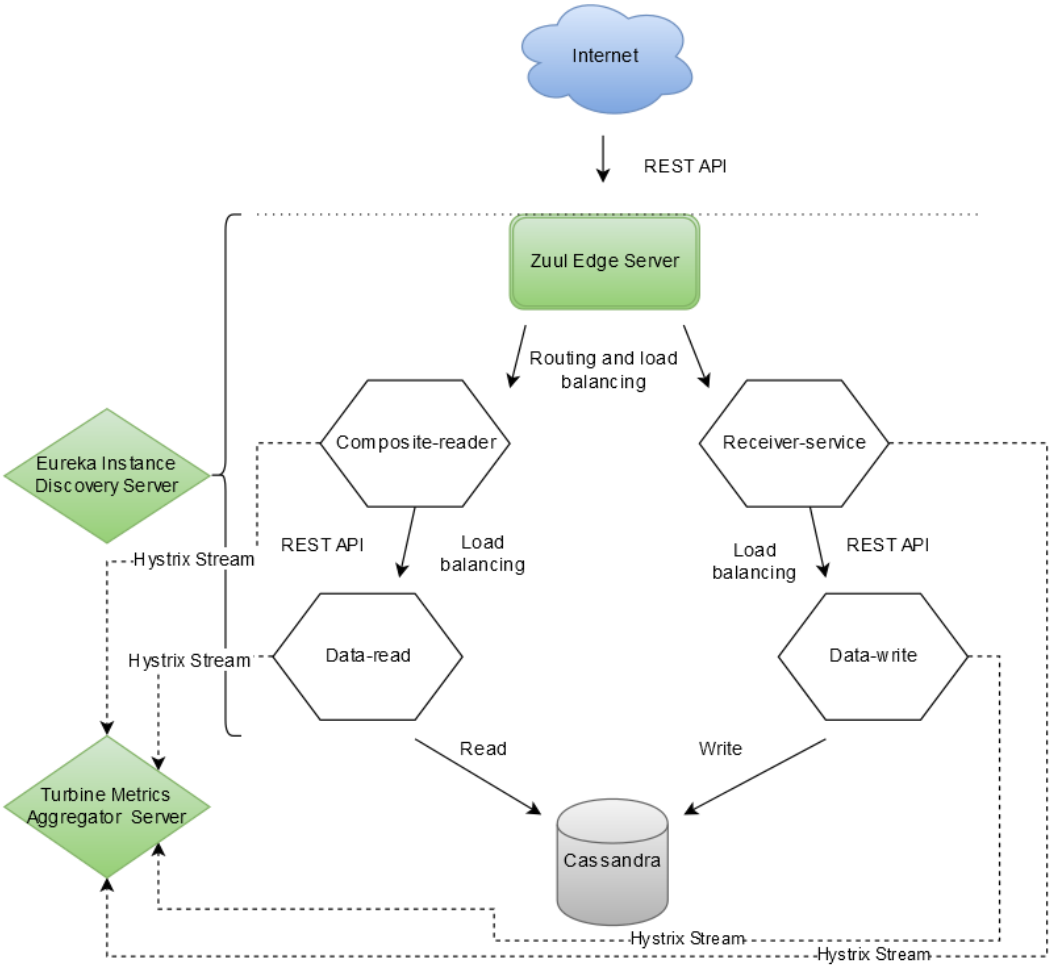
Composite-reader –palvelu vastaanottaa pyyntöjä mittausarvojen lukemiseen tietokannasta REST-rajapinnan kautta ja välittää pyynnöt *Data-read* –palvelulle ja palauttaa sen vastauksen pyynnön lähettäjälle.

Jokaiselle palvelulle on lisätty Hystrix-toiminnallisuus. Turbine-palvelin käyttää Eureka-palvelimen instanssirekisteriä löytääkseen kaikkien palveluiden Hystrix-tietovirrat.

Sekä reunapalvelimen (Zuul) että ensimmäisen tason palveluiden (Receiver ja Composite-read) välissä tapahtuu kuormituksen jako (Load balancing) (**Kuvio 10.**). Tämä tarkoittaa sitä, että kaikkien palveluiden instanssien lisääminen kasvattaa koko sovelluksen suorituskykyä.

Taulukko 2. Spring-testisovelluksen palveluiden yhteenveto.

Palvelun nimi	**Tehtävä**
Receiver-service	Vastaanottaa mittausarvoja REST-rajapinnan kautta.
Composite-reader	Vastaanottaa mittausarvojen kyselyitä REST-rajapinnan kautta.
Data-write	Kirjoittaa mittausarvot tietokantaan.
Data-read	Lukee mittausarvoja tietokannasta.



Kuvio 10. Spring-testisovelluksen arkkitehtuurinen diagrammi.

4.7 Käyttöönotto

Spring-testisovelluksen käyttöönotto tapahtuu käynnistämällä ensiksi Eureka- ja Zuul-palvelimet, jonka jälkeen palvelut voidaan käynnistää missä järjestyksessä tahansa.

Projektin rakentaminen Maven build-komennolla tuottaa jokaiselle ajettavalle kokonaisuudelle JAR-tiedoston, joka on valmis ajettaavaksi. Nämä JAR-tiedostot voidaan hajauttaa eri palvelimille tai käynnistää paikallisesti.

4.8 Havainnot

Mikropalvelusovelluksen toteuttaminen Springillä on yllättävän helppoa ja tuntuu tutulta. Yhden palvelun toteutuksen voi tehdä samalla periaattella kuin minkä tahansa sovelluksen. Spring Cloud- ja Spring Boot –ominaisuudet saadaan lisäämällä muutama annotaatio, jolloin sovellus muuttuu mikropalveluksi. Instanssipalvelimen Eurekan ja reunapalvelimen Zuulin saa helposti otettua käyttöön, mikä säästää aikaa ja vaivaa.

Eureka-palvelin pitää kirjaa mainiosti jokaisen palvelun instansseista. Zuul-palvelin ja sovelluksen palvelut löytävät toisensa nopeasti Eurekan kautta. Eurekan ainut vika vaikuttaa olevan kaatuneiden instanssien poistaminen instanssirekisteristä. Kun instanssi kaatui, Eurekalla kesti jopa yli minuutti poistaa se rekisteristä. Tämä johti lumipallomaiseen virheiden kasaantumiseen, kun Eureka tarjosi kaatunutta palvelua Zuulille ja sovelluksen toisille palveluille.

Reunapalvelin Zuul toimii myös hyvin silloin, kun kaikki palvelut ovat normaalisti saatavilla. Jos jonkin palvelun kaikki instanssit olivat kaatuneet, pyynnön saapuessa Zuul yrittää epätoivoisesti muodostaa yhteyden johonkin niistä ja siinä epäonnistuttua aiheuttaa pitkän listan virheilmoituksia. Pyynnön lähettäjälle taas palautetaan virhekoodi, joka viittaa palvelinvirheeseen.

Hystrixin kuormitus- ja virhetilastot sekä katkaisintoiminnot sai otettua helposti käyttöön ja ne toimivat kuin pitää. Varsinaisten tilastojen yhdistäminen ja näyttäminen Hystrix Dashboardissa oli hieman vaikeampaa, mutta lopulta kaikkien palveluiden tilastot tulivat näkyviin yhdelle sivulle. Ainut huono puoli tilastoissa on se, että Hystrix laskee kuorman ja virheet per palvelu, ei per palvelun instanssi. Jos haluaa tietää kuinka paljon liikennettä menee millekin instanssille, se ei ole Hystrixin avulla mahdollista.

Kun käyttää Netflixin tarjoamia työkaluja sovelluksen toteuttamiseen ja ylläpitoon, kynnys on tosi matala. Mikropalveluista tietämätönkin kehittäjä voi helposti luoda monimutkaisen ja toimivan sovelluksen.

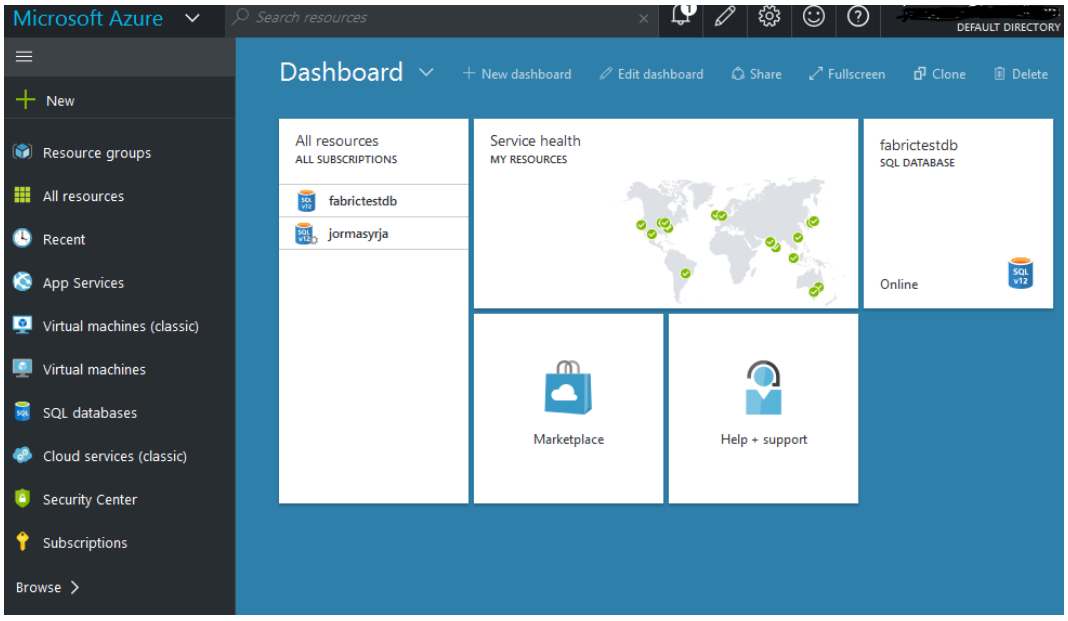
5 MICROSOFT AZURE SERVICE FABRIC

Service Fabric on Microsoftin kehittämä mikropalvelukehitysympäristö, joka on suunniteltu ajettavaksi Azure-pilvipalvelussa. Service Fabric on toteutettu .NET-kielillä ja sovellusten kehittäjiä varten on olemassa liitännäinen, jonka voi asentaa Visual Studioon.

5.1 Azure

Azure (**Kuvio 11.**) on Microsoftin pilvipalvelu, jossa käyttäjät voivat ajaa sovelluksiansa ja säilöä dataa. Azure tukee useita eri ohjelmointikieliä ja ajoympäristöjä, kuten Linuxia ja Javaa.

Eräs mielenkiintoinen maksutapa on käytön mukainen laskutus. Asiakas voi muuttaa resurssiensa määrää tarpeen mukaan. Yhdistämällä tämän maksutavan mikropalveluiden skaalautuvuuteen saadaan aikaan tehokas sovellus, joka ei juurikaan tuhlaa palvelinkustannuksia. /10/

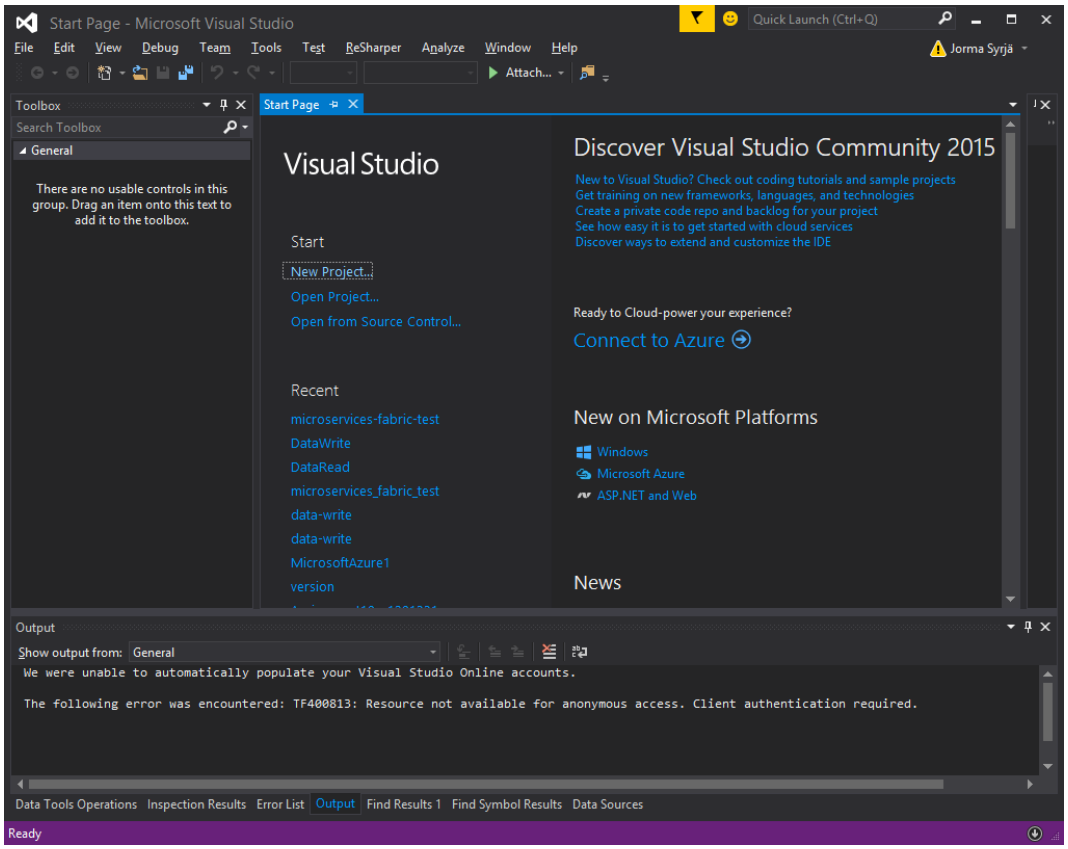


Kuvio 11. Azure-työpöytä.

5.2 Visual Studio ja .NET

Visual Studio on ilmainen kehitystyökalu Microsoftin .NET-ympäristöä varten. .NET-ympäristöön kuuluu erä luokkakirjastoja, joita voidaan ajaa usealla kielellä. .NET-kieliin kuuluu muun muassa Visual Basic ja C#. /11/

Visual Studiosta on saatavilla useita eri versioita. Tämän testisovelluksen tekoon käytettiin ilmaista Visual Studio 2015 Communitya (**Kuvio 12.**).

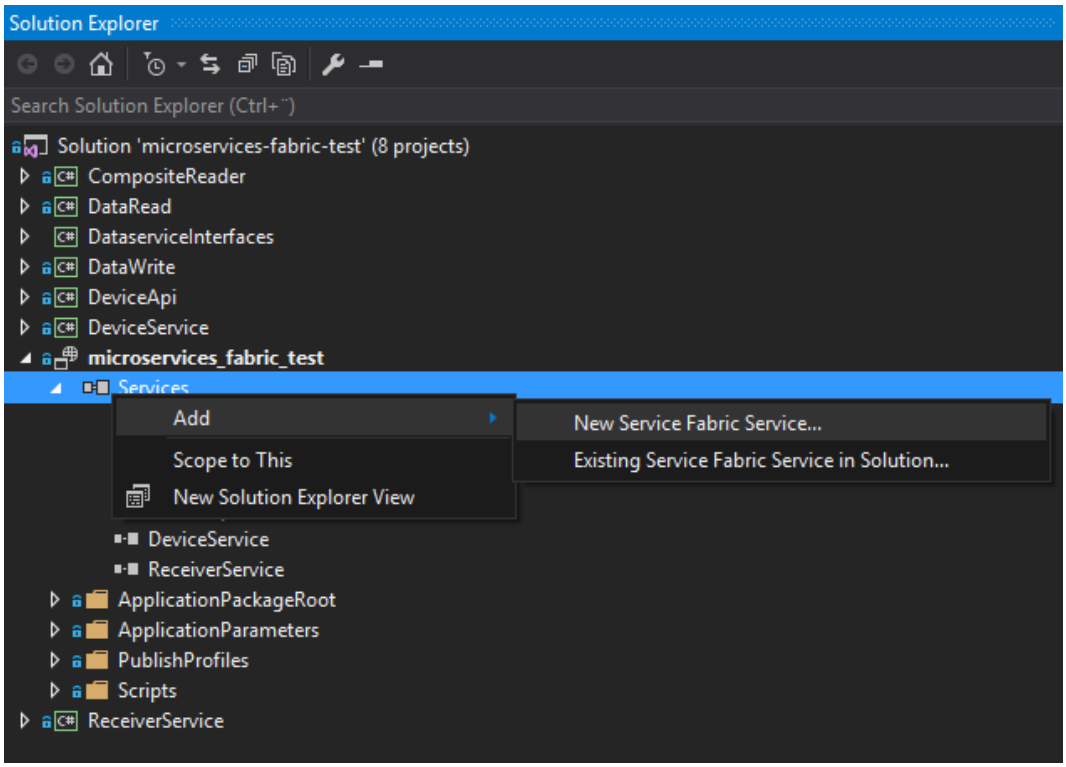


Kuvio 12. Visual Studion aloitusnäkymä.

5.3 Service Fabric

Service Fabric tarjoaa kehittäjille mahdollisuuden tuottaa pilvivalmiita sovelluksia, jotka ovat luotettavia ja kasvuvaraisia. Sovellukseen voi lisätä haluamansa määrän palveluita kolmesta eri palveluluokasta. Kaikki palvelut suorittavat yhtä tehtävää mikropalvelumallin mukaisesti, mutta niissä on hieman eroja.

Uuden palvelun lisääminen Service Fabric –projektiin käy helposti klikkaamalla emoprojektia ja valitsemmalla Add Service (**Kuvio 13.)**.



Kuvio 13. Uuden palvelun lisääminen Service Fabric -projektiin.

5.3.1 Tilaton (Stateless) palvelu

Tilaton palvelu on yksinkertaisten ja kertaluontoisten tehtävien suorittamista varten. Tilaton palvelu voi hoitaa esimerkiksi viestin vastaanottamisen tai tallentamisen tietokantaan. Tilaton palvelu ei pidä sisällään mitään tietoa. /12/

5.3.2 Tilallinen (Stateful) palvelu

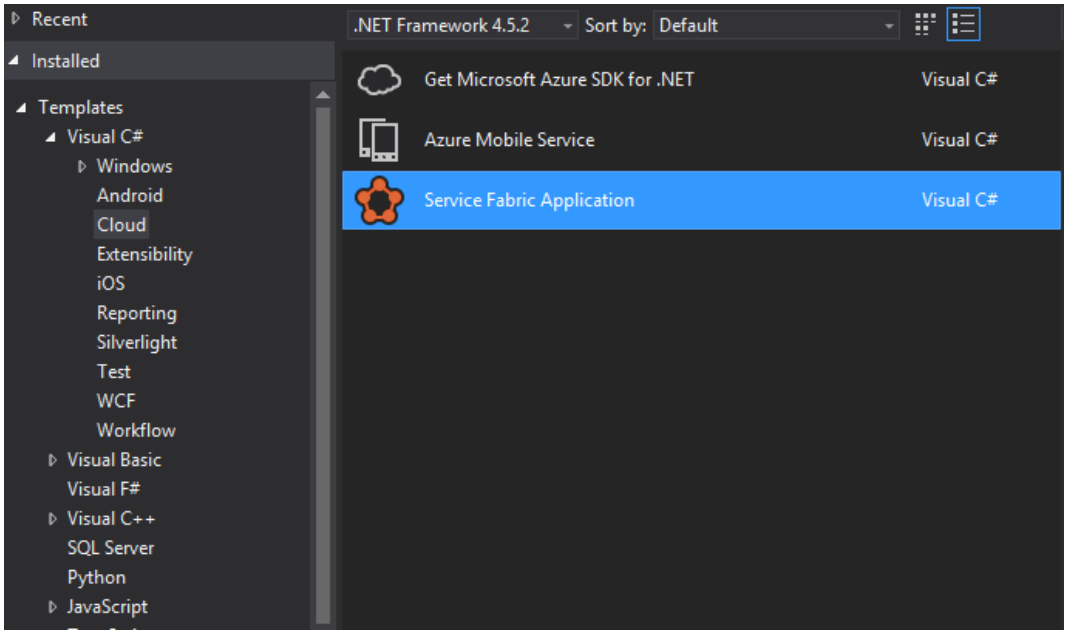
Tilallinen palvelu pitää sisällään jonkinlaisen tilan. Tila voi olla esimerkiksi laitteen mittausarvojen laskettu keskiarvo tai toiminnanohjauksen valinta. Tilallisesta palvelusta otetaan tietyin väliajoin kopio, joka tallennetaan tietokantaan. Palvelu lataa viimeisimmän kopion uudelleenkäynnistyksen yhteydessä. /12/

5.3.3 Actor, ts. ”työläinen”

”Työläinen” ajaa yksinkertaista logiikkaa ja se sisältää tiedon yksinkertaisesta tilasta, esimerkiksi päällä/pois. Työläinen on pieni ja kevyt ja niitä voidaan käyttää esimerkiksi Internet of Things –sovelluksessa, jossa yksi työläinen vastaa yhtä sovellukseen mittausarvoja lähettävää tietokonetta. /13/

5.4 Testisovelluksen toteutus

Esimerkkisovelluksen Visual Studio –projektin luomista varten täytyy asentaa Service Fabric –liitännäinen. Liitännäisen avulla on mahdollista luoda Service Fabric –projekti (**Kuvio 14.)**, joka pitää valmiiksi sisällään kaiken mikropalveluarkkitehtuurin paitsi palvelut.



Kuvio 14. Uuden projektin luominen templatesta Visual Studiossa..

5.4.1 Uuden palvelun lisääminen

Kun uuden palvelun on lisännyt emoprojektiin (**Kuvio 13.**), jäljellä on vain toteutus. Halutun palvelutyppin runko lisätään automaattisesti palvelun alle.

Toteutuksessa on suositeltavaa käyttää asynkronisia metodeja. Tämä tapahtuu C#:ssa käytämällä metodin esittelylausekkeessa muuttuujaa **async**, käyttämällä

paluuarvotyyppinä Task ja lisäämällä **await**-avainsanalla varustetun metodikutsun metodin toteutukseen (**Kuvio 15.**).

```
public async Task<HttpResponseMessage> Put([FromUri] Guid deviceId, [FromBody] Datanode dn)
{
    if (Datanode.validate(dn))
    {
        IReceiver receiver = ServiceProxy.Create<IReceiver>(new Uri("fabric:/microservices_fa
        InternalMessageResponse resp = await receiver.Receive(deviceId, dn);
        switch (resp)
        {
```

Kuvio 15. Osittainen esimerkki asynkronisesta .NET REST-metodista.

Asynkronisuuden määrittäminen on tärkeää, jos palvelu joutuu käsittelemään useita viestejä samanaikaisesti. Toinen syy on estää lukkiutuminen pitkän tehtävän suorittamisen ajaksi.

5.4.2 Palveluidenvälinen viestintä

Service Fabric tarjoaa omalaatuisen tavan saman sovelluksen sisällä olevien palvelujen väliseen viestintään. Sovelluksen kehittäjä voi valita viestin kuljetustavaksi muun muassa HTTP:n, TCP:n tai objektipohjaisen viestijonon.

Viestien lähettämistä varten täytyy luoda rajapintaluokka (**Kuvio 16.**).

```
public interface IReceiver : IService
{
    Task<InternalMessageResponse> Receive(Guid deviceId, Datanode dn);
}
```

Kuvio 16. Rajapintaluokka palveluidenvälistä viestintää varten.

Lähettäjä luo instanssin rajapintaluokasta ja kutsuu sen tarjoamaa metodia antaen samalla vastaanottajapalvelun Service Fabric URI:n (**Kuvio 17.**). Service Fabric URI on palvelut yksilöllistävä tunniste, joka koostuu etuliitteestä fabric, sovelluksen nimestä (tässä microservices_fabric_test) sekä kohdepalvelun nimestä.

```
IReceiver receiver;
receiver = ServiceProxy.Create
    <IReceiver>(new Uri("fabric:/microservices_fabric_test/DataWrite"));

InternalMessageResponse resp = await receiver.Receive(deviceId, dn);
```

Kuvio 17. Palveluidenvälisen viestin lähettäminen rajapintaluokan avulla.

Vastaanottajan tulee toteuttaa rajapinta ja luoda käynnistyksen yhteydessä kuuntelijaolio (**Kuvio 18.**).

```
internal sealed class DataWrite : StatelessService, IReceiver
{
    public DataWrite(StatelessServiceContext context)
        : base(context)
    { }

    private WriteDao dao;

    public async Task<InternalMessageResponse> Receive(Guid deviceid, Datanode dn)
    {
        ServiceEventSource.Current.ServiceMessage(this,"Received: {0}",dn.ToString());
        IDevices dvc_check =
            ServiceProxy.Create
            <IDevices>(new Uri("fabric:/microservices_fabric_test/DeviceService"));
        bool x = await dvc_check.CheckExistence(deviceid);

        if (!x)
        {
            return InternalMessageResponse.NO_CONTENT;
        }

        if (dao == null)
        {
            return InternalMessageResponse.SERVICE_UNAVAILABLE;
        }

        bool success = await dao.WriteNode(deviceid,dn);

        if(success)
            return InternalMessageResponse.OK;

        return InternalMessageResponse.INTERNAL_ERROR;
    }
```

Kuvio 18. Tilaton palvelu, joka toteuttaa palveluidenvälisen viestinnän rajapinnan.

5.5 Testisovelluksen yhteenveto

Testisovellus sisältää 6 eri palvelua (**Taulukko 3.**). Tietokantana toimi Azuressa ajettava Microsoft SQL Server.

Receiver vastaanottaa mittausarvoja REST-rajapinnan kautta ja välittää ne DataWrite-palvelulle. Ennen välitystä mittausarvo tarkistetaan oikeaksi.

DataWrite vastaanottaa kirjoitettavia mittausarvoja ja laitteen ID:n, jolle mittausarvo kuuluu. Laitteen olemassaolo tarkastetaan DeviceService-palvelulta. Jos laitetta ei ole olemassa, kirjoitusta ei suoriteta.

DeviceService vastaanottaa uusien laitteiden rekisteröitymispyyntöjä ja kyselyitä laitteiden olemassaolosta. Sekä rekisteröinti- että olemassaolopyynnöissä on mukana ID, jota käytetään pyynnön käsittelyssä.

DeviceApi vastaanottaa laitteiden rekisteröitymispyyntöjä REST-rajapinnan kautta ja välittää ne DeviceService-palvelulle.

CompositeReader vastaanottaa mittausarvojen lukupyyntöjä REST-rajapinnan kautta ja välittää ne DataRead-palvelulle. Ennen välitystä tarkastetaan, onko vastaanotetussa pyynnössä mukana ID.

DataRead vastaanottaa mittausarvojen lukupyyntöjä ja lähettää kyselyn tietokantaan.

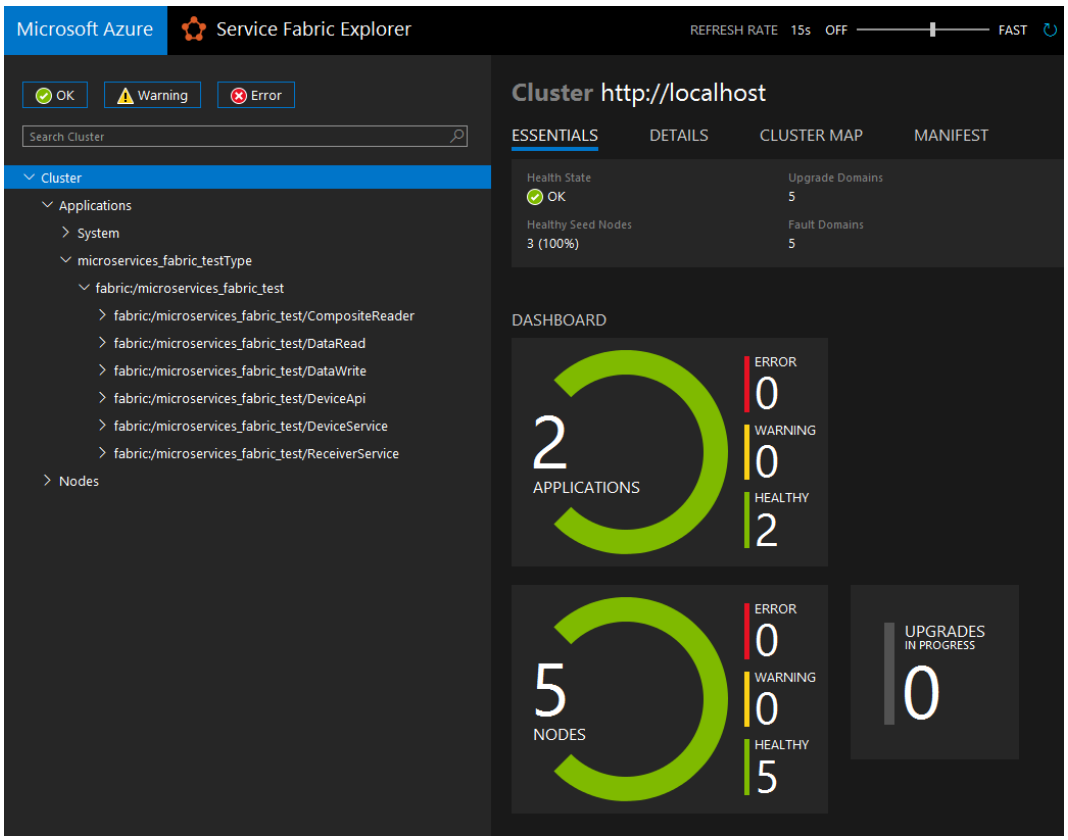
Taulukko 3. Service Fabric –testisovelluksen palveluiden yhteenveto.

Palvelun nimi	Tehtävä
Receiver	Vastaanottaa mittausarvoja REST-rajapinnan kauttta.
DeviceApi	Vastaanottaa laitteiden rekisteröintikyselyitä REST-rajapinnan kautta.
CompositeReader	Vastaanottaa mittausarvojen lukupyyntöjä REST-rajapinnan kautta.
DataWrite	Kirjoittaa mittausarvot tietokantaan.
DataRead	Lukee mittausarvoja tietokannasta.
DeviceService	Rekisteröi uuden laitteen tietokantaan ja vastaa laitteiden olemassaoloon koskeviin kyselyihin.

5.6 Käyttöönotto

Visual Studio voi käynnistää Service Fabric –projektin suoraan paikalliselle koneelle virtuaalipalvelimelle. Testiajoa varten Visual Studio täytyy käynnistää järjestelmänvalvojan oikeuksilla.

Oletuksena sovelluksesta luodaan viiden palvelimen klusteri. Palvelut on jaettu satunnaisesti klusterin eri palvelimille. Kun klusteri on käynnistynyt, käyttäjä voi avata Fabric Dashboardin (**Kuvio 19.**), joka toimii ylläpitö- ja tilastotyöpöytänä sovellukselle.



Kuvio 19. Fabric Dashboard sovelluksen ajon aikana.

Fabric Dashboard toimii selaimessa ja se näyttää kaikki sovellukset, palvelimet ja palvelimilla suoritettavat palvelut. Etusivulla näkyy ajossa olevat sovellusten ja palvelimien lukumäärä sekä näiden tilat.

5.7 Havainnot

Visual Studiolla toteutettavat Service Fabric –mikropalvelusovellukset on helppo toteuttaa ja ajaa. Uuden projektin luominen onnistuu helposti ja palveluiden lisääminen on yksinkertaista. Palveluiden toteutuksen lisäksi mitään muita toimenpiteitä ei vaadita kehittäjältä, sillä palvelut lisätään automaattisesti sovellukseen. Sekä instanssipalvelin että monitorointi ovat sisäänrakennettuja.

Käyttöönotto rajoittuu paikalliselle koneelle ilman maksua. Sovelluksen ajaminen paikallisena on erittäin raskasta ja käynnistys kestää useita minuutteja. Tämä voi johtua siitä, että oletuksena luodaan viisi palvelinta.

Maksua vastaan Service Fabric –sovelluksen voi ottaa käyttöön helposti Azureen tai haluamalleen palvelimelle. Azuressa ajettava Fabric-sovellus voi hyödyntää automaattisesti skaalautuvia resursseja. /14;15/ Skaalaus on mahdollista suorittaa ainoastaan automaattisesti.

Service Fabricilla toteutettu mikropalvelusovellus on hyvä vaihtoehto suurille yrityksille, jotka eivät halua itse hoitaa palvelimia ja resursseja. Microsoftin tuotteisiin suuntautuvat kehittäjät saavat hyödynnettyä uusimpia .NET-kirjastoja ja Visual Studion työkaluja.

6 LIGHTBEND LAGOM

Lightbend (ent. Typesafe) on ohjelmistoyritys, joka tarjoaa pilvisovelluksia. Heidän tuotteisiinsa kuuluu muun muassa ConductR, Lagom, Akka ja Play. Activator on Lightbendin työkalu edellämainittujen tuotteiden asentamista ja ajamista varten.

6.1 ConductR

ConductR on käyttöönottotyökalu, jonka avulla muista Lightbendin tuotteista voi helposti tehdä hajautettuja sovelluksia. Se toimii sekä instanssi-, reuna- että monitorointipalvelimena, johon esimerkiksi Lagom-sovellus voi rekisteröityä. ConductR tarkkailee rekisteröityneitä sovelluksia ja tarjoaa kuormituksen jakamista ja automaattista virheen korjausta. /16/

6.2 Akka

Akka on kirjasto suorituskykyisten Java-, Scala- ja .NET-sovellusten kehitykseen. Akkan tarjoamat toteutukset suuntautuvat tehtävien samanaikaisuuteen (concurrency). Virheiden korjaus perustuu ”let it crash” –periaatteeseen, jossa virheen annetaan kaataa sovellus. Juurisyytutkimuksen avulla sovellusta yritetään automaattisesti korjata. Akka on yksi Lagomin perustuksista. /17/

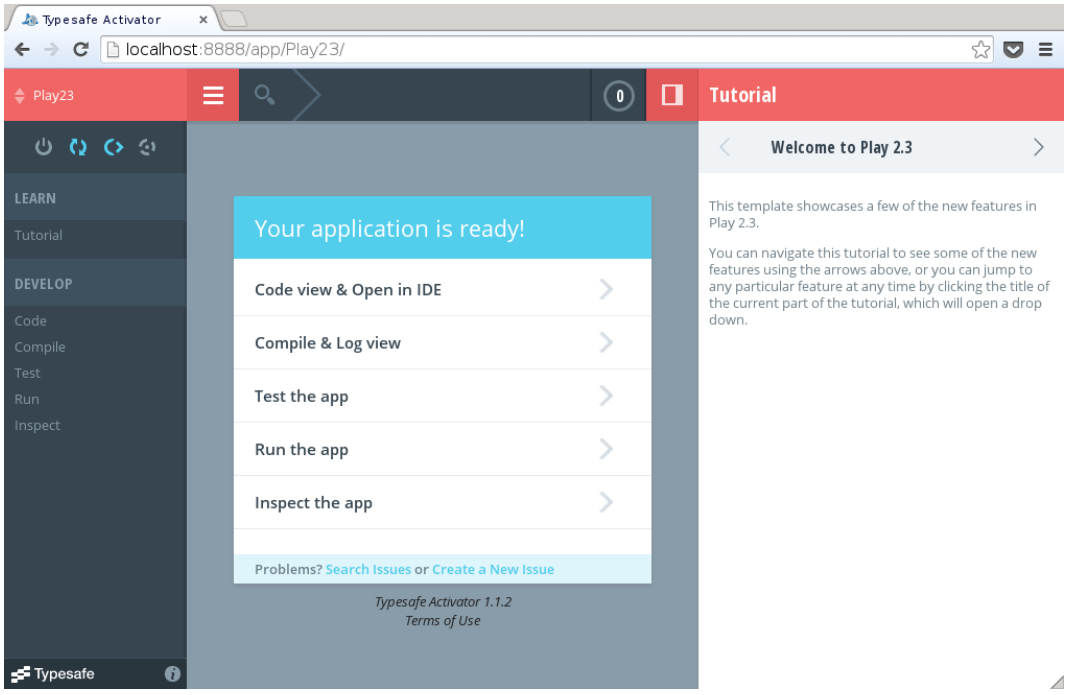
6.3 Play

Play on kehitysympäristö suorituskykyisten ja kevyiden Java- ja Scala –websovellusten kehitykseen. Play on rakennettu Akkan päälle, joten sen tarjoamat kehitystyökalut ovat myös suuntautuneet sovelluksen tehtävien samanaikaisuuteen. Play on myös yksi Lagomin perustuksista. /18/

6.4 Activator

Activator on työkalu, jolla voi asentaa, luoda ja ajaa Lightbendin tuotteiden projekteja. Esimerkiksi Activatorin kautta voi luoda uuden Lagom-projektin, jossa on valmiina riippuvuudet ja esimerkkikoodeja. Activator toimii sekä komentoriviltä että graafisena käyttöliittymänä (**Kuvio 20.**). Activator käyttää Simple Build Toolia

(SBT) projektien kääntämiseen, ajamiseen ja riippuvuuksien hakemiseen. /20/



Kuvio 20. Activator UI. /19/

6.5 Lagom

Lagom on ruotsia ja tarkoittaa vapaasti suomennettuna ”juuri oikea määrä”. Nimellä on erikoismerkitys, sillä Lightbend kehitti Lagomin mikropalveluarkkitehtuurien malliksi, jossa yksi mikropalvelu suunniteltaisiin juuri oikean kokoiseksi, ei liian pieneksi tai suureksi.

Suuri osa Lagomin perustuksista on kirjoitettu Scalalla, mutta itse sovellusten toteutuksen voi tehdä myös Javalla. Scala-pohjaisuudesta kuitenkin juurtaa rajoitus käännöstyökalun käytön suhteen, sillä Lagom-sovellukset voidaan kääntää ja ajaa ainoastaan käyttämällä Lightbendin Activatoria ja SBT:tä.

Lagom on jaettu neljään osaan:

1. Service API, joka rajapinta toteutuksille, joita mikropalvelut käyttävät
2. Persistence API, joka on rajapinta tietokanta- ja tallennustoiminnoille
3. Täysi kehitysympäristö sovelluksen ajamista ja koodin päivittämistä varten
4. Tuotantoympäristö, jossa on sisäänrakennettu tuki ConductR:lle. /20/

Lagom-sovelluksissa voi olla sekä tilattomia että tilallisia palveluita.

6.5.1 Tilaton palvelu

Tilaton palvelu on oletustyyppi palveluille, joiden tehtävä on suorittaa yhtä tehtävää. Palvelu ei pidä sisällään minkäänlaista tietoa.

6.5.2 Tilallinen palvelu

Tilallinen palvelu on vaihtoehtoinen palvelutyyppi, jolla on mahdollisuus käsitellä säilyviä olioita. Säilyvällä oliolla on tunniste ja tila ja sille voidaan lähettää komentoja. Komennot saattavat muuttaa tilaa ja tilan muutos saattaa tuottaa tapahtuman, joka tallennetaan tietokantaan. Uudelleenkäynnistyksen jälkeen säilyvän olion tila palautetaan käyttämällä tapahtumahistoriaa. Toisin sanoen palvelu ei itsessään pidä sisällään mitään tilaa, vaan tilallisia olioita.

6.6 Testisovelluksen toteutus

Lagom rajoittaa toteutuksia siten, että Javalla kirjoitettu koodi tulee kääntää versiolla 8+. Lagomin tarjoamat rajapinnat palveluiden toteutuksiin käyttävät paljon Java versio 8:ssa tulleita uusia ominaisuuksia, kuten tehtävien samanaikaisuuteen perustuvia luokkia ja lambda-lausekkeita (**Kuvio 21**.).

```
@Override
public ServiceCall<String, Datanode, String> receive() {

    return(id,request) ->{
        log.info("Received: " + request == null ? "null" : request.toString());
        if(request.getTimestamp() == null)
            request.setTimestamp(new Date());
        CompletionStage<Done> response = writer.addNode()
                .invoke(NotUsed.getInstance(), request)
                .exceptionally(ex -> showException(ex));
        dService.update().invoke(id,request).exceptionally(ex -> showException(ex));
        return response.thenApply(x -> "OK: " + response);
    };
}
```

Kuvio 21. Metodi, jossa käytetään lambda-lausetta ja CompletionStage-luokkaa.

6.6.1 Lambda-rakenne

Java versio 8:ssa lisättiin lambda-lausekkeet, joilla on mahdollista yksinkertaistaa yhden metodin rajapintaluokkien toteutuksia. Lambda-lauseke muodostuu vastaanotetusta parametrista, nuolesta ja koodista, johon parametri syötetään ja josta mahdollinen paluuarvo saadaan.

Esimerkiksi lambda-lauseke

```
return (int x, int y) -> x + y;
```

Vastaanottaa kaksi kokonaislukua x ja y, sijoittaa ne toiminnallisuuteensa (x+y) ja palauttaa laskutoimituksen vastauksen. Tämä yhden rivin koodipätkä pitää sisällään saman toiminnallisuuden kuin seuraava esimerkki.

```
public int metodi(int x, int y){
            return x+y; }
```

Käytännössä lambda-lauseke on metodi, joka sijoitetaan toisen metodin parametriksi tai paluuarvoon.

6.6.2 CompletionStage

CompletionStage-luokka on yksi Java versio 8:ssa lisätyistä samanaikaisuuteen perustuvista toteutuksista. CompletionStage esittää tehtävän etenemistä ja se pitää sisällään suoritettavan tehtävän paluuarvon. Eräs käyttötapa on kutsua metodia, jonka suorittaminen kestää ja joka tuottaa CompletionStage-olion. CompletionStage-oliolla on metodi, joka voidaan suorittaa silloin, kun tehtävä on suoritettu loppuun. Tämä metodi voidaan yhdistää esimerkiksi REST-rajapinnan lähettämään vastaukseen, kuten kuviossa 20.

6.6.3 Uuden palvelun lisääminen

Lagom-sovelluksen palveluita varten täytyy luoda 2 projektia per palvelu, API-projekti ja toteutusprojekti.

API-projektissa määritellään palvelun tarjoamat rajapinnat ja niiden kutsut (**Kuvio 22**). Tämä projekti tulee lisätä riippuvuudeksi niiden palveluiden toteutuksille, jotka lähettävät viestejä kyseiselle palvelulle.

```
public interface ReceiverService extends Service {

    ServiceCall<String,Datanode,String> receive();

    @Override
      default Descriptor descriptor() {
        return named("receiverservice").with(
            restCall(Method.PUT, "/api/receive/:id", receive())
          ).withAutoAcl(true);
      }
}
```

Kuvio 22. Määrittelyluokka palvelun API-projektissa.

Toteutusprojektiin tehdään palvelun toteutukset palvelun rajapinnoille ja määritetään palvelun asetukset. API-projektissa määriteltyjen rajapintojen toteutuksessa on pakko käyttää lambda-rakenteita.

6.6.4 Palveluidenvälinen viestintä

Palveluiden välistä viestintää varten tulee lähettäjäpalvelun toteutusprojektiin lisätä riippuvuus vastaanottajapalvelun API-projektiin. Tämän lisäksi lähettäjäpalvelun pitää rekisteröidä itsensä vastaanottajan asiakkaaksi ja luoda instanssi vastaanottajasta omassa konstruktorissaan (**Kuvio 23.**). Lähettäjäpalvelun käynnistyessä ConductR syöttää vastaanottajapalvelusta instanssin lähettäjälle.

Vastaanottajapalvelun instanssin alustuksen jälkeen lähettäjä voi kutsua kaikkia vastaanottajan API-projektissa määriteltyjä rajapintoja.

```
public class ReceiverImpl implements ReceiverService {

    private final Logger log = LoggerFactory.getLogger(this.getClass());

    DatawriteService writer;
    DeviceEntityService dService;

    @Inject
    public ReceiverImpl(DatawriteService writer, DeviceEntityService dService) {
        this.writer = writer;
        this.dService = dService;
    }
```

Kuvio 23. Viestirajapintojen alustus palveluidenvälistä viestintää varten lähettäjäpalvelussa.

6.6.5 Tilallinen palvelu ja säilyvät oliot

Säilyvien olioiden hallintaan tarvitaan seuraavat toteutukset:

- Tilallinen palvelu, joka vastaanottaa säilyvien olioiden komennot ja ohjaa ne oikeisiin osoitteisiin.
- Säilyvän olion tilan kuvausluokka.
- Säilyvän olion tapahtuman kuvausluokka.
- Säilyvän olion komentojen kuvausluokka.

Säilyvän olion toteutukset laitetaan sitä hallitsevan palvelun toteutusprojektiin. Hallitsevan palvelun käynnistyksessä sen tulee alustaa itsellensä instanssi PersistentEntityRegistry-luokasta (**Kuvio 24.**), jonka kautta palvelu voi kutsua säilyviin olioihin liittyviä metodeja. PersistentEntityRegistry-instanssiin tulee rekisteröidä palvelun käyttämät säilyvät oliot.

```
public class DeviceEntityImpl implements DeviceEntityService{

    private final PersistentEntityRegistry registry;

    @Inject
    public DeviceEntityImpl(PersistentEntityRegistry registry) {
        this.registry = registry;
        registry.register(Device.class);
    }
```

Kuvio 24. Säilyviä olioita käyttävän palvelun alustaminen.

PersistentEntityRegistryn kautta palvelu voi kutsua säilyviä olioita luokan nimen ja tunnisteen avulla. Komennot suoritetaan lähettämällä instanssi komentoluokasta (**Kuvio 25.**), jotka määritetään säilyvän olion komentojen kuvausluokassa. (**Kuvio 26.**).

```
@Override
public ServiceCall<String, Datanode, Done> update() {
    return(id,request) -> {
        PersistentEntityRef<DeviceCommand> ref =
                registry.refFor(Device.class, id);
        return ref.ask(new Increment(
                Double.parseDouble(request.getValue())
                ,new Date()))
                .thenApply(ok -> ok);
    };
}
```

Kuvio 25. Komennon lähettäminen säilyvälle oliolle.

```
public interface DeviceCommand extends Jsonable{

    @SuppressWarnings("serial")
    public final class Increment implements DeviceCommand,
    CompressedJsonable, PersistentEntity.ReplyType<Done>{
        public final Double value;
        public final Date timestamp;

        @JsonCreator
        public Increment(Double value,Date timestamp) {
            this.value = value;
            this.timestamp = timestamp;
        }
```

Kuvio 26. Esimerkki säilyvän olion komentojen kuvausluokasta.

6.7 Testisovelluksen yhteenveto

Testisovellus sisältää 5 eri palvelua (**Taulukko 4.**). Tietokantana toimi sisäänrakennettu Apache Cassandra.

Receiver vastaanottaa mittausarvoja REST-rajapinnan kautta ja välittää ne DataWrite-palvelulle.

DataWrite kirjoittaa mittausarvoja tietokantaan ja välittää ne DeviceEntity-palvelulle.

DeviceEntity vastaanottaa mittausarvoja ja kirjoittaa ne tunnisteen perusteella säilyviin olioihin. Sillä on REST-rajapinta, jonka kautta voi kysellä laitteen mittausarvojen keskiarvoa tunnisteen perusteella.

CompositeReader vastaanottaa mittausarvojen lukukyselyitä REST-rajapinnan kautta ja välittää kyselyt DataRead-palvelulle.

DataRead lukee mittausarvoja tietokannasta.

Taulukko 4. Lagom-testisovelluksen palveluiden yhteenveto.

Palvelun nimi	**Tehtävä**
Receiver	Vastaanottaa mittausarvoja REST-rajapinnan kautta.
CompositeReader	Vastaanottaa mittausarvojen lukukyselyitä REST-rajapinnan kautta.
DataWrite	Kirjoittaa mittausarvoja tietokantaan.
DataRead	Lukee mittausarvoja tietokannasta.
DeviceEntity	Pitää kirjaa laitteiden mittausarvojen keskiarvoista ja tallentaa keskiarvot tietokantaan.

6.8 Käyttöönotto

Lagom-testisovelluksen käynnistäminen tapahtuu ajamalla Activatorin komentoriviltä projektin juurihakemistossa, jossa sijaitsee build.sbt –tiedosto. Activator lataa projektin muistiin ja sen jälkeen koko sovelluksen voi käynnistää komennolla runAll.

Sovellus käynnistyy kehitystilaan ja lokimerkinnät tulostuvat samaan ikkunaan, josta runAll ajettiin.

6.9 Havainnot

Lagom-sovelluksen kehityksen aloittaminen oli hankalaa. Activator ei toiminut ja SBT ei onnistunut kääntämään edes testiprojektia. UI oli täysin rikki ja eräs kehittäjistä Lightbendillä kehotti minua käyttämään Activatoria pelkästään komentorivin kautta. Tämä ratkaisu toimi mainiosti, kunnes loin omia projekteja omille palveluilleni.

Jokaisen palvelun API- ja toteutusprojektit tulee määritellä projektin build.sbt –tiedostossa sijaintineen ja riippuvuuksineen. Joidenkin Lagomin toimintojen riippuvuuksien lisääminen oli vaikeaa, koska en tiennyt niiden nimiä. Edes Lightbendin omassa Getting Started –aloitusoppaassa ei lukenut joidenkin riippuvuuksien nimiä.

Säätämiseen meni useiden tuntien työpanos. Kun lopulta sain kaiken toimimaan ja projektit kääntymään, jouduin opettelemaan käytännöt, joilla palvelut toteutetaan Lagom-sovelluksessa. Onneksi lambda-rakenteiden ja CompletionStagen käyttö ei ollut vaikeaa minulle. Eräs vähän vanhempi ja kokeneempi työkaverini totesi, että nämä uudet Java-maailman jutut eivät oikein uppoa vanhoihin konkareihin, joten se saattaa olla este Lagomin käyttämiselle joillekin kehittäjille.

Loppujen lopuksi sain todeta, että Lagom on erittäin kehittynyt kehitysympäristö, jolla mikropalvelusovellusten kehittäminen on hyvin rutiininomaista ja helppoa. Sen avulla syntyy erittäin suorituskykyisiä ja virheenkestäviä palveluja. Instanssi- ja reunapalvelimet ovat esiasennettuja jokaiseen sovellukseen ja ne toimivat moitteettomasti.

7 KEHITYSYMPÄRISTÖJEN VERTAILU

Kehitysympäristöt ovat keskenään hyvin erilaisia ja mikropalvelusovelluksen toteutuksen kannalta tärkeät piirteet erottuvat jokaisessa hyvin.

7.1 Spring

Spring-sovelluksen toteuttaminen oli kaikkein helpoin. Se ei juurikaan vaadi uusien toteutustapojen ja kirjastojen opettelua, jos käyttää Spring Cloud ja Spring Web –ympäristöjen tarjoamia kirjastoja. Lopputuloksena syntyy kehittäjän osaamisen tasoinen sovellus, joka on valmis mikropalvelu.

Vianmäärityksen tarkkuuden ja virheiden sietävyyden toteutuksesta vastuussa on kehittäjä. Mitään automaattista virheiden korjausta ei ole saatavilla, ainoastaan tilastoja.

7.2 Service Fabric

Service Fabric –sovelluksen toteutus on helppoa, jos hallitsee .NETin ja sen tarjoamat kirjastot. Kehittäjän kädet ovat tiukasti sidottu sovellusten välisen kommunikoinnin kanssa, mutta tarjolla olevat vaihtoehdot ovat hyviä. .NETin REST-rajapinnan toteuttaminen on tuskallista, sillä .NETin tarjoama Web API –kirjasto on vielä epäkypsä.

Service Fabric –sovellus ei juurikaan näytä tilastoja sovelluksen kautta kulkevasta liikenteestä, vaan kaikki nojautuu mahdollisen Azure-käyttöönoton varaan. On selvää, että Microsoft haluaa Service Fabric –sovellukset ajettavan Azuressa.

7.3 Lagom

Lagom on uusi, mielenkiintoinen ja kunnianhimoinen projekti mikropalveluiden suosion kasvattamiseksi. Pienen lämmittelyn jälkeen palveluiden ja muiden toteutusten kanssa ei paljoa kulu aikaa ja tuloksena syntyy hyvin suorituskykyinen ja virheitäsietävä sovellus. Lagom tarjoaa muihin kehitysympäristöihin verrattuna yllättävän paljon työkaluja ja jos Lightbendiin voi uskoa, Lagom tulee kasvamaan suorituskykyisemmäksi ja monipuolisemmaksi ympäristöksi.

Lagomin heikot puolet sikiävät enemmän henkilökohtaisista mieltymyksistä. Lagom-sovelluksen toteuttamiseen ja ajamiseen vaaditaan sekä Activator että SBT. Projektin hallintatyökaluna SBT on erittäin huono ja sen syntaksi on sekavaa. Activator saattaa sovelluksen käynnistyessä ahmaista niin paljon muistia, että JVM kaatuu. Tämä tosin voi johtua sisäänrakennetusta Apache Cassandra –tietokannasta, joka itsessään on yksi Lagomin kehityksen eduista.

7.4 Yhteenveto vertailusta

Jokaisella testatulla kehitysympäristöllä oli erilainen tapa määritellä palvelut ajettava sovellus ja siihen kuuluvat palvelut. Vaikka kahdessa ympäristössä ohjelmointikieli oli sama, toteutustavat olivat täysin erilaiset. Taulukossa 5 on listattu yhteenveto kehitysympäristöistä ja asioista, jotka ovat tärkeitä mikropalvelusovellukselle.

Taulukko 5. Kehitysympäristöjen yhteenveto.

Ominaisuus	Spring	Service Fabric	Lagom
Omaksumisen vaikeus	Helppo	Kohtalainen	Vaikea
Ohjelmointikieli	Java	.NET –kielet, esim. C# ja Visual Basic	Java, Scala
Instanssipalvelin	Eureka	Naming Service (Sisäänrakennettu **palvelu**)	Sisäänrakennettu
Reunapalvelin	Zuul	Ei saatavilla	Sisäänrakennettu
Sovelluksen monitorointi	Hystrix – virheet ja kuormitus per palvelu	Palvelinten lukumäärä, sovelluksen virheet	Ei saatavilla
Virheenkorjaus	Kehittäjän vastuulla	Tuki Azuren automaattista uudelleenkäynnistystä varten	ConductR, tarvittaessa uudelleenkäynnistys ja uusien instanssien kickstart

Palveluiden instanssien hallinta	Käyttöönott ajan vastuulla	Automaattinen	Automaattinen

8 JOHTOPÄÄTÖKSET

Aloitin tämän työn tekemisen tietämättä mitään mikropalveluista. Ensimmäisen kehitysympäristön, Springin valinta ensimmäiseksi oli sattuma, johon törmäsin, kun etsin tietoa mikropalveluista. Idea Service Fabricin ja Lagomin tutkimisesta tulivat työpaikaltani. Monet suuret yhtiöt ovat sitoutuneet Microsoftiin ja sen tuotteisiin, joten Service Fabricin huomioon ottaminen eri mikropalveluarkkitehtuurien vertailussa on erittäin viisasta. Viimeinen vaihtoehto, Lagom, herätti aluksi vain kysymyksiä ja epäilyksiä pyrkyrimäisestä ohjelmistoyrityksestä, joka häikäilemättä tyrkyttää omia tuotteitaan.

Alkukankeudesta huolimatta **Lagom** oli se kehitysympäristö, joka herätti minussa eniten intoa uusien sovellusten kehityksestä. Sen opettelu on raskasta ja vaatii avointa mieltä. Eräs kokeneempi työkaverini sanoi, että Lagomin toteutuksien opettelu ei luonnistu hänenkaltaiseltaan tapojensa orjalta. Hän myös sanoi, että nuorempi sukupolvi, joka on juuri nyt siirtymässä työelämään, on enemmän kuin kykenevä suurten ja mahtavien Lagom-sovellusten tekoon, kunhan motivaatiota löytyy.

Lagom ei tietenkään ole ainut ympäristö, jolla on mahdollista tehdä toimivia ja kestäviä sovelluksia. Microsoftin tuotteisiin sitoutuneelle ja .NETin osaavalle kehitystiimille Service Fabric on erittäin hyvä valinta, kunhan sovellukset ajetaan Azuressa. Myös Springillä voi tehdä toimivia mikropalveluita ja Netflix on elävä esimerkki onnistuneesta Spring-mikropalvelutoteutuksesta. Syy, miksi en suosittele Springiä suurten sovellusten kehitykseen on se, että se on täysin avoin. Ei ole tahoa, joka pitää huolen sen toimivuudesta, kuten Lagomin kehittäjät Lightbendillä ja Service Fabricin kehittäjät Microsoftilla. Spring on täysin riippuvainen avoimen lähdekoodin yhteisön panoksesta.

Nuoren ohjelmistokehittäjäsukupolven potentiaalin perusteella väitän, että Lagom ja mikropalveluarkkitehtuurit ovat suuri ja merkittävä osa ohjelmistokehityksen tulevaisuutta. Perinteistä monoliittia tuskin unohdetaan koskaan, mutta väistämätöntä on se, että monet sovellukset siirretään tai kehitetään pilvipalvelimille. Pilvessä ajettavat sovellukset voivat ainoastaan hyötyä

mikropalveluista ja sovellusten koon ja käyttäjäkunnan kasvaessa entisestään, mikropalveluarkkitehtuuriratkaisu saattaa olla ainut tapa sovelluksen käytettävyyden säilyttämiseksi.

LÄHTEET

1. Hoff, T. The Great Microservices vs Monolithic Apps Twitter Melee
   Viitattu 18.4.2016
   http://highscalability.com/blog/2014/7/28/the-great-microservices-vs-monolithic-apps-twitter-melee.html
2. Fowler, M. Microservices
   Viitattu 18.4.2016
   http://martinfowler.com/articles/microservices.html
3. Fowler, M. Microservice Trade-Offs
   Viitattu 18.4.2016
   http://martinfowler.com/articles/microservice-trade-offs.html
4. Nene, D. A beginners guide to Dependency Injection
   Viitattu 19.4.2016
   http://www.theserverside.com/news/1321158/A-beginners-guide-to-Dependency-Injection
5. Webb, P. ja Syer, D. Spring Boot – Simplifying Spring for Everyone
   Viitattu 19.3.2016
   https://spring.io/blog/2013/08/06/spring-boot-simplifying-spring-for-everyone
6. Stack Overflow: What exactly is Spring?
   Viitattu 19.3.2016
   http://stackoverflow.com/questions/1061717/what-exactly-is-spring-for/1064562#1064562
7. Resilience with Hystrix
   Viitattu 19.4.2016
   http://labs.enonic.com/articles/resilience-with-hystrix
8. Gopal, S. Application Resiliency using Netflix Hystrix
   Viitattu 19.4.2016
   http://www.ebaytechblog.com/2015/09/08/application-resiliency-using-netflix-hystrix/

9. Chapman, P. Microservices with Spring
   Viitattu 19.4.2016
   https://spring.io/blog/2015/07/14/microservices-with-spring
10. What is Azure
   Viitattu 20.4.2016
   https://azure.microsoft.com/en-gb/overview/what-is-azure/
11. .NET Framework
   Viitattu 20.4.2016
   https://en.wikipedia.org/wiki/.NET_Framework
12. Snider, M. Reliable Services overview
   Viitattu 20.4.2016
   https://azure.microsoft.com/en-us/documentation/articles/service-fabric-reliable-services-introduction/
13. Turecek, V. Introduction to Service Fabric Reliable Actors
   Viitattu 20.4.2016
   https://azure.microsoft.com/en-us/documentation/articles/service-fabric-reliable-actors-introduction/
14. Daniel, C. Create a Service Fabric cluster from the Azure portal
   Viitattu 20.4.2016
   https://azure.microsoft.com/en-us/documentation/articles/service-fabric-cluster-creation-via-portal/
15. Daniel, C. Service Fabric cluster capacity planning considerations
   Viitattu 20.4.2016
   https://azure.microsoft.com/en-us/documentation/articles/service-fabric-cluster-capacity/
16. ConductR
   Viitattu 21.4.2016
   http://www.lightbend.com/products/conductr
17. What is Akka
   Viitattu 21.4.2016
   http://doc.akka.io/docs/akka/2.4.4/intro/what-is-akka.html

18. Built for asynchronous programming
Viitattu 21.4.2016
https://www.playframework.com/documentation/2.5.x/Philosophy
19. Stocker, M. Play 2.3 Released: Modularization, Java 8 and WebJars
Viitattu 21.4.2016
http://www.infoq.com/news/2014/05/play-23
20. Providing a high productivity work environment
Viitattu 21.4.2016
https://www.lightbend.com/community/core-tools/activator-and-sbt
21. What is Lagom?
Viitattu 21.4.2016
http://www.lagomframework.com/documentation/1.0.x/WhatIsLagom.html
22. Command and Query Responsibility Segregation (CQRS) Pattern
Viitattu 17.3.2016
https://msdn.microsoft.com/en-us/library/dn568103.aspx