Kirsi Rinnesalo

Mikropalveluarkkitehtuuri

Sovelluskohteena JYSOA-arkkitehtuuri

Tietotekniikan kandidaatintutkielma

13. joulukuuta 2019

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Kirsi Rinnesalo

Yhteystiedot: kirsi.rinnesalo@jyu.fi

Työn nimi: Mikropalveluarkkitehtuuri— Sovelluskohteena JYSOA-arkkitehtuuri

Title in English: Microservice Architecture— Sovelluskohteena JYSOA-arkkitehtuuri

Työ: Kandidaatintutkielma

Sivumäärä: 23+0

Tiivistelmä: Kirjallisuuskatsaus mikropalveluarkkitehtuuriin; mitä mikropalvelut ovat ja mihin niitä käytetään. Sovelluskohteena tarkastellaan Jyväskylän yliopiston digipalveluiden JYSOA-arkkitehtuuria.

Avainsanat: mikropalvelu, arkkitehtuuri, ohjelmistokehitys

Abstract: About microservice architecture; what microservices are and what they are used for. As a case study is used JYSOA architecture of Jyväskylä University Digital Services.

Keywords: microservice, architecture, software engineering

Kuvat

Sisältö

1 Johdanto

Mikropalvelut ovat varsin uusi kehityssuuntaus ohjelmistoarkkitehtuurissa. Ensimmäinen maininta onkin vasta 2014 Lewisin ja Fowlerin blogikirjoituksessa. Miksi puheet mikropalveluista sitten alkoivat?

Perinteisesti ohjelmistoja on rakennettu yhdeksi isoksi monoliitiksi. Monoliittinen ohjelmisto voi sisältää paljonkin erilaisia palveluja ja komponentteja, mutta se julkaistaan yhtenäisenä sovelluksena, jonka yksittäisiä osia ei voi suorittaa itsenäisesti. Järjestelmän palaset jakavat samat palvelinresurssit, muistialueet ja tietokannan (Dragoni ym. 2017). Monoliitin voi ja usein rakennetaankin modulaariseksi uudelleenkäytettävillä komponenteilla ja komponentit ovat itsenäisiä ja niistä voi muodostaa kirjastoja muidenkin sovellusten käyttöön, mutta julkaisuun niistä koostetaan yksi yhtenäinen iso sovellus (Martin 2018, s.176–179).

Richardsonin (2019, s. 4) mukaan monoliitin käytössä on paljon etuja. Sopivan kokoisena järjestelmänä sen sovelluskehitys on helppoa ja nopeaa, kun kaikki tarvittavat ovat yhdessä. Suorituskyky on hyvä, kun komponenttien välinen keskustelu hoituu metodikutsuilla. Yksinäinen ylläpitäjä tai pieni kehitystiimi tuntee sovelluksen ja koodin hyvin, ja järjestelmän modulaarisuus pysyy hyvin hallittavissa. Palvelun skaalauskin on helppoa; sovelluksesta vain kopioidaan rinnalle useampia instansseja ja kuormantasauksella saadaan käyttäjät jaettua instanssien kesken.

Koodimäärän kasvaessa sen hallittavuus vaikeutuu (Richardson 2019, s. 4-7). Laajempi koodipohja tarkoittaa usein myös isompaa kehitystiimiä. Yhden henkilön kapasiteetti ei riitä koko järjestelmän hallintaan, joten kehitystyötä on tehtävä yhdessä koko tiimillä, eikä koodin laaduntarkkailu ole enää yhtä helppoa. Modulaarisuuskin voi helposti rappeutua ja komponenttien vastuut kasvavat. Yksittäisen komponentin muutosten päivittäminen vaatii koko sovelluksen päivittämisen, mikä hankaloittaa jatkuvien julkaisujen tekemistä. Päivittämisestä aiheutuu väkisinkin käyttökatkoja ja siitä aiheutuu aina ongelmia käyttäjille.

Skaalautuvuuteen ainoana ratkaisuna on koko sovelluksen kopiointi, ja tämä taas tarvitsee resurssien moninkertaistamista rinnakkaisten sovellusten määrällä, sillä yksittäisten komponenttien resurssien lisääminen ei monoliittisessa sovelluksessa ole mahdollista, vaikka jokaisen komponentin resurssitarve onkin erilainen. Monoliittisen järjestelmän teknologia määrittelee ja asettaa rajoituksia yksittäisten komponenttien kehitykselle, eikä uusien sovelluskehysten käyttöönotto eri sovelluksen osissa ole mahdollista, tai ainakin useampien rinnakkaisten vaihtoehtojen ylläpitäminen on erittäin hankalaa.

Monoliittiselle arkkitehtuurille kaivattiin jotain kevyempää ratkaisua — jotain, joka helpottaa virheenkorjausta ja uusien ominaisuuksien julkaisemista. Seuraavaksi tutustutaan kirjallisuuskatsauksen kautta mikropalveluarkkitehtuuriin. Luodaan katsaus mikropalvelujen syntyyn, mitä ovat mikropalvelut ja mihin niitä käyttävä arkkitehtuuri soveltuu. Kirjallisuuskatsauksen jälkeen tarkastellaan esimerkkinä Jyväskylän yliopiston digipalveluiden JYSOA-arkkitehtuuria.

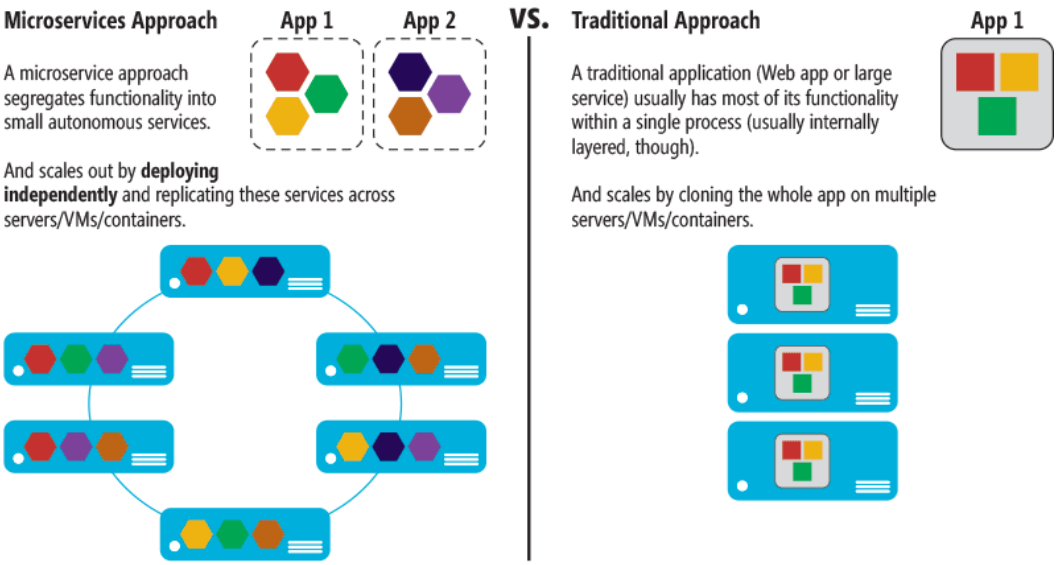
2 Mikropalveluarkkitehtuuri

Monoliittisen sovelluskehityksen ongelmiin ratkaisua tarjosi vuosituhannen vaihteessa palvelukeskeinen arkkitehtuuri SOA (*engl. Service Oriented Architecture*). Ehkä näkyvimmän lähtölaukauksen antoi Jeff Bezos vuonna 2002 lähettäessään Amazonin työntekijöille sähköpostin, jonka mukaan kaikkien kehitystiimien tulee julkaista palveluidensa data ja toiminnallisuudet palvelukuvauksina (API Evangelist 2012). Nämä palvelukuvaukset tarjoavat selkeät palvelurajapinnat hajautetuille itsenäisille sovelluksille tehden SOA:sta teknologiariippumattoman arkkitehtuurin.

Palvelukuvaukset ovat rakenteellisia ja järjestelmäriippumattomia, kuten WSDL (*engl. Web Service Definition Language*). Tosin vaikka www-sovelluspalvelu (*engl. Web Service*) onkin nykyään yleisin ratkaisumalli SOA-palvelulle, ei se ole ainoa, vaan muitakin tekniikoita on laajasti käytössä, kuten CORBA ja DCOM tai XML-RPC (MSDN 2016). SOA manifestin (2009) mukaan palvelukeskeisen arkkitehtuurin tärkein periaate on luoda organisaatiolle liiketoiminnallista etua ja se onnistuu tarkastelemalla organisaation palvelukokonaisuuksia yksittäisten sovellusten sijaan (Biske 2010, s. 29).

Newman (2015, luku 1) toteaa, että SOA:n hyvästä ideasta huolimatta sen toteutukset eivät olleetkaan kovin onnistuneita. Kun hajautuksella haettiin ratkaisua monoliittisten järjestelmien ongelmiin, saatiinkin tilalle entistä suurempi järjestelmä, jonka osien itsenäisyys ei enää ollutkaan itsestään selvää. Tämä ei tarkoita, että SOA itsessään olisi huono arkkitehtuuri, mutta sen toteutuksessa keskityttiin käytännön ongelmien sijaan rakentamaan keskitetyistä välityspalvelimista ja viestiväylistä kaikille sopivaa kokonaisuutta ja unohdettiin palvelukeskeisyys. Mikropalveluarkkitehtuuri sai alkunsa käytännön tarpeesta tehdä SOA oikein (Zimmermann 2017).

Kuvassa 1 on havainnollistettu, miten oikein toteutettuna mikropalveluarkkitehtuuri mahdollistaa ison skaalautuvan sovelluksen kehittämisen ja ketterän ja nopean päivittämisen.

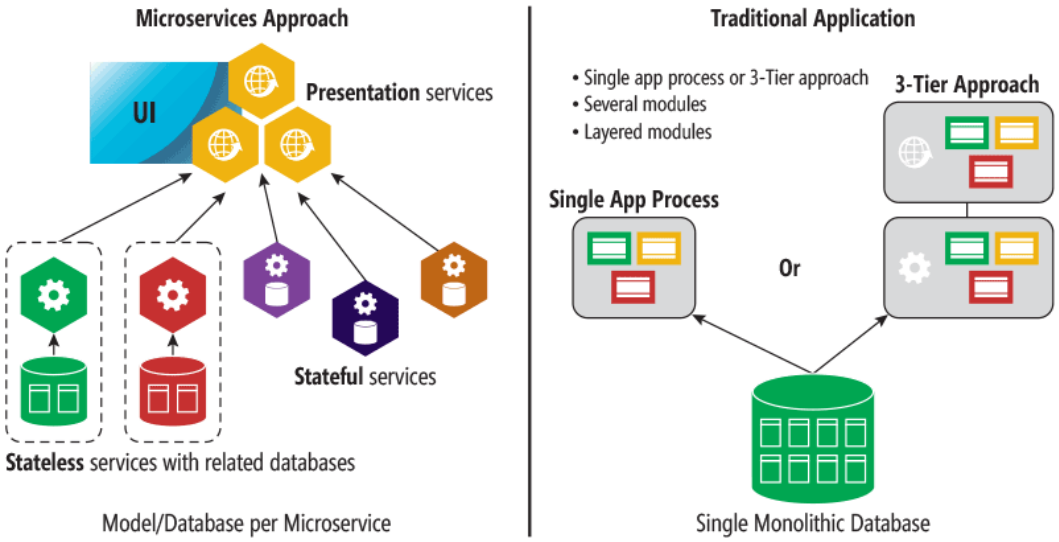


Kuva 1. Mikropalveluiden ja monoliittisen järjestelmän eroavaisuudet (Torre, Singh ja Turecek 2015)

2.1 Mitä mikropalvelut ovat?

Mikropalveluarkkitehtuurin määritelmä on varsin lavea ja kuten Saarelainen (2016) kertoo Hotarin todenneen, niin se on terminäkin hiukan harhaanjohtava. Todenmukaisempaa olisikin puhua omavaraisista järjestelmistä. Arkkitehtuurin tarkoituksena on pilkkoa laajempi järjestelmä pienempiin omavaraisiin palveluihin. Omavaraisuus tarkoittaa sitä, että jokaisella mikropalvelulla on oma sisäinen tietomallinsa, eikä sen tarvitse noudattaa organisaation laajuista tietomallia. Tärkeintä on, että rajapinnat ulospäin noudattavat sovittuja käytänteitä (Newman 2015, luku 1).

Itsenäisten mikropalveluiden eron perinteisen monoliitin keskitettyyn tiedonhallintaan voi nähdä kuvasta 2.



Kuva 2. Vertailu datan riippumattomuudesta mikropalveluissa ja monoliittisessa järjestelmässä (Torre, Singh ja Turecek 2015)

Krause (2015, luku 3) määrittelee mikropalveluiden perusfilosofiaksi liiketoiminnan monimutkaisuuden purkamista pienempiin tarkoin kohdistettuihin palveluihin. Filosofian perusajatus on keskittyä yksittäiseen kohteeseen sen sijaan, että yrittää tehdä useaa asiaa yhtä aikaa.

Mikropalveluiden merkittävimpinä ominaisuuksina pidetään skaalautuvuutta, ylläpidettävyyttä ja helppoa julkaisemista (Newman 2015, luku 1). Mikropalvelu on kapseloitu sovittujen käytänteiden mukasilla rajapinnoilla, mikä mahdollistaa palvelun kehittämisen tarpeeseen parhaiten sopivalla arkkitehtuurilla ja ohjelmointikielellä. Julkaistu rajapinta helpottaa palvelun uudelleenkäytettävyyttä useissa käyttöliittymissä tai toisissa palveluissa. Mikropalvelu on myös helppo korvata tai päivittää vapaasti, kun sen vaikutukset on rajattu rajapinnalla. Pienen palvelun laadunvarmistuskin on nopeampaa kuin muutokset isossa monoliitissa, jossa yhden osan vaikutukset voivat ulottua minne tahansa järjestelmässä. Jos mikropalvelun päivityksestä aiheutuu virheitä, on palaaminen vanhaan toimivaan versioon helppoa, kun vain se yksi pieni palvelu tarvitsee palauttaa ja näin muiden palveluiden toiminnallisuudet palautuvat automaattisesti. Jos huomataan jonkin yksittäisen palve-

lun vaativan enemmän resursseja, on sen skaalaaminen nopeaa vain monistamalla uusia instansseja tai kasvattamalla palvelun tarvitsemia resursseja. Vastavuoroisesti jos monitoroinnissa havaitaan jonkin palvelun käyttävän huomattavasti annettua vähemmän resursseja, on siltä helppo vähentää niitä.

2.2 Mihin mikropalveluarkkitehtuuri ei sovellu?

Mikropalveluja ei kannata ottaa käyttöön vain mikropalvelujen takia. Ne eivät tuo ratkaisua kaikkiin käyttökohteisiin, vaan arkkitehtuuri on luotu omaan tarpeeseensa (Fowler 2015). Kun järjestelmän tulee pyöriä pienessä tilassa, ei sen osien hajauttamisesta tule lisäarvoa. Toisaalta, kun keskustelun palvelujen välillä tulee olla saumatonta ja nopeaa, eivät mikropalvelut tuo silloinkaan toivottua ratkaisua. Mikropalvelujen keskustelu tapahtuu aina verkossa, joten viestinvälityksessä on aina mukana verkko-ongelmia; viestejä katoaa tai verkkoyhteydet katkeavat. Mikropalveluarkkitehtuurin ylläpitokaan ei välttämättä ole monoliittiä helpompaa. Vaikka yksittäinen palvelu on pieni ja sen päivittäminen helppoa ja riskit pieniä, on jokaista palvelua ylläpidettävät erikseen. Palvelujen määrän kasvaessa, kasvaa myös kompleksisuus.

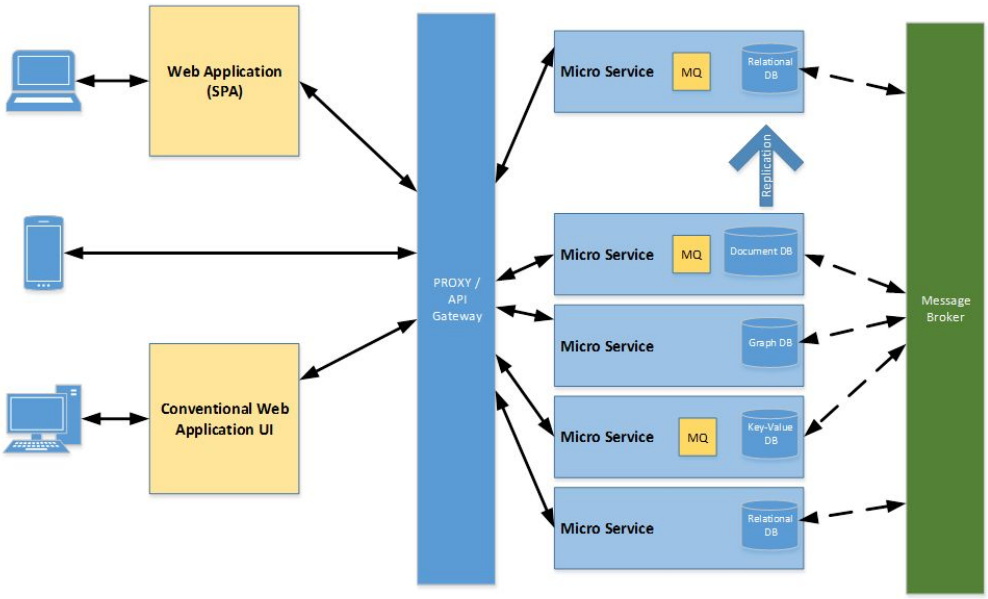
Newman (2015, luku 3) neuvoo aloittamaan varsinkin uusien järjestelmien suunnittelemisen monoliitista. Ennen kuin järjestelmän osien väliset riippuvuudet ja vastuut ovat tarkoin selvillä, ei järjestelmän palastelusta mikropalveluiksi ole hyötyä. Monoliitin pilkkominen pienemmiksi palveluiksi on helpompaa ja halvempaa kuin väärin määriteltyjen mikropalveluiden suhteiden korjaaminen jälkikäteen.

3 Mikropalvelujen tekniikkaa

Standardoidun ajoympäristön mikropalveluille mahdollistaa konttiympäristö. Palveluverkkoarkkitehtuurista taas haetaan helpotusta palvelukokonaisuuden integrointiin, monitorointiin ja virhetilanteiden selvittelyyn.

Koska hajautetut järjestelmät ovat monimutkaisempia kuin yksittäiset sovellukset, tulee kiinnittää erityisesti huomiota niin kontekstin ja palveluiden kuin palveluiden keskinäiseenkin kommunikointiin. Palvelujen keskinäinen kommunikointi on hidasta, koska on oletettava, että kaikki kommunikaatio tapahtuu verkon yli, vaikka palvelut olisivatkin vierivierekkäin samalla prosessorilla (Martin 2018, s.177–179).

Viestinvälitysarkkitehtuuri tarjoaa ratkaisukeinoja viestintään palveluiden välillä ja niiden kanssa järjestelmän ulkopuolelta, kuten Krause (2015, luku 5) on kuvassa 3 havainnollistanut. Mikropalveluiden keskinäiseen kommunikointiin arkkitehtuuri ehdottaa ratkaisuksi tapahtumapohjaista viestiväylää. Käyttöliittymät ja ulkopuoliset käyttäjät kutsuvat mikropalveluiden toiminnallisuuksia rajapintayhdyskäytävän yhtenäisen rajapinnan kautta.



Kuva 3. Mikropalveluarkkitehtuuri (Krause 2015, kuva 21)

Mikropalveluideologiassa tulee hyvin esille ketterien menetelmien perusajatukset iteratiivisuudesta, mukautuvuudesta ja nopean julkaisun peraatteesta. DevOps-toimintamalli tuo ketteryyden mikropalveluiden kehittämiseen ja ylläpitoon.

3.1 Konttiteknologia

Konttiteknologia tarjoaa mikropalvelulle standardoidun ympäristön, joka on helppo siirtää kehityskoneelta testiympäristöön ja edelleen tuotantopalvelimelle (Newman 2015, luku 6). Kontteja on helppo siirtää myös palvelimelta toiselle tai vaikkapa konesalista pilvipalveluun.

Fyysiset palvelimet ja virtuaalipalvelimet tarjoavat perinteisesti sovellukselle toimintaympäristön, mutta palvelimella on paljon muutakin kuin varsinaiset sovelluksen tarvitsemat kirjastot ja asetukset. Palvelin koostuu fyysisen laitteen lisäksi käyttöjärjestelmästä ja virtualisointiympäristöstä. Jokainen virtuaalipalvelin sisältää lisäksi oman täysverisen käyttöjärjestelmänsä kaikkine kirjastoineen. Niin fyysisiä kuin virtuaalipalvelimiakin tulee päivittää ja ylläpitää ja näillä on aina vaikutuksensa palvelimessa ajettavaan sovellukseenkin.

Kontti koostuu hyvin minimalistisesta ajoympäristöstä sisältäen ainoastaan tarvitsemansa riippuvuudet. Näin kontin koko jää vain murto-osaan virtuaalipalvelimen koosta. Palvelimelta säästyy levytilaa, kun virtualisointiympäristön sijaan siellä ajetaankin konttien ajoympäristöä. Samalla minimoidaan käyttöjärjestelmäkomponenttien vaikutukset kontissa ajettavaan sovellukseen. Yhdessä virtuaalipalvelimessa voidaan ajaa useita kontteja toisistaan riippumatta ja alustapalvelimen päivittämiseksi riittää siirtää kontit uudelle palvelimelle, eikä se vie kuin sekunteja. Yleisimmäksi konttien ajoympäristöksi on noussut Docker (Krause 2015, luku 9).

Useampien konttien ajamiseen tarvitaan konttien hallintajärjestelmä, jolla kontit jaetaan useille palvelimille. Hallintajärjestelmällä hallitaan konfiguraatiotiedostoja, tietokantojen sijaintia ja konttien kuormantasausta.

3.2 Palveluverkko

Luonteensa vuoksi hajautettu järjestelmä on altis virheille ja Newman (2015, luku 11) kehottaakin ottamaan jo suunnittelussa huomioon, ettei virheitä voi välttää, vaan keskittyä siihen, miten virheistä palaudutaan mahdollisimman nopeasti.

Palveluverkko (*engl. Service Mesh*) on usein sivuvaunumallilla (*engl. Sidecar Design Pattern*) toteutettu arkkitehtuurikerros hajautetuille järjestelmille. Palveluverkon avulla standardoidaan esimerkiksi mikropalveluiden välistä kommunikointia ja lokitusta (Calcote 2018, s.7–9). Monitoroinnilla tarkkaillaan mikropalveluiden tilaa ja niiden tuottamaa lokitietoa, jotka sisältävät muun muassa tietoa mikropalveluiden kuormasta tai virhetilanteista.

Richardsonin mukaan (2019, s. 380–382) palveluverkkojen avulla hajautettuihin järjestelmiin saadaan näkyvyyttä ja joustavuutta sekä liikenteen ja turvallisuuden hallintaa. Itse mikropalveluihin ei tarvitse lisätä mitään, eikä sen kehittäjän tarvitse välittää monimutkaisista liiketoimintalogiikan ulkopuolisista tehtävistä. Näin palvelun kokokin pysyy pienenä, kun se voi keskittyä vain ja ainoastaan omaan tehtäväänsä.

3.3 Viestiväylä

Mikropalveluiden keskinäiseen viestintään Krause (2015, luku 5) suosittelee käytettävän viestiväylää (*engl. Event Bus*) tai -jonoa (*engl. Message Queue*). Koskimiehen ja Mikkosen (2005, s. 139–140) mukaan viestiväylän avulla voidaan rakentaa järjestelmä, jossa palvelujen ei tarvitse tietää toisistaan. Viesti abstrahoi ja kapseloi palvelupyynnön ja vastauksen sekä niiden sisältämän informaation ja käytetyn protokollan. Välittäjä (*engl. Message Broker*) huolehtii viestinvälityksestä siitä kiinnostuneille palveluille. Viestinvälitys on kriittinen osa mikropalveluarkkitehtuuria, joten sen luotettavuuteen on kiinnitettävä erityistä huomiota (Dragoni ym. 2017). Viestiväylä tarjoaa apua myös yhtäaikaisuuden hallintaan ja kuormantasaukseen; viestin tallentaminen jonoon on kevyempää kuin sen käsittely, joten ruuhkatilanteessa on mahdollista lisätä viestejä käsittelevien instanssien määrää.

3.4 Rajapintayhdyskäytävä

Kun hajautetussa arkkitehtuurissa palveluiden vuorovaikutus tapahtuu aina verkon kautta, ja toisaalta mikropalveluiden itsenäisyyden vuoksi palveluiden rajapinnat voivat olla toteutettu millä tahansa protokollalla, tarvitaan keino hallita mikropalveluiden toiminnallisuuksia yhtenäisen rajapintapalvelun kautta (Dragoni ym. 2017).

Rajapintayhdyskäytävä (*engl. API Gateway*) toimii rajapintakerroksena taustajärjestelmien ja käyttäjille näkyvien palveluiden välissä. Sen tehtävänä on helpottaa uusien sovellusten rakentamista ja käyttäjille näkyvien toiminnallisuuksien kehittämistä. Rajapintayhdyskäytävän kautta palveluita voidaan myös helposti tarjota ulospäin, jolloin kuka tahansa voi rakentaa tarvitsemiaan sovelluksia tarjotuista tiedoista. Rajapintayhdyskäytävä tarjoaa myös mahdollisuuden rakentaa asiakaskohtaisia rajapintoja sen mukaan, mitä tietoja rajapinnasta asiakas tarvitsee. Rajapintayhdyskäytävä toimii välityspalvelimena asiakkaalta mikropalveluihin, mutta sillä voi hoitaa myös kuormantasauksen, tunnistautumisen ja virheenkäsittelyn (Richardson 2019, s. 259–268).

Selvästi rajapintayhdyskäytävän hyötynä on, että se piilottaa mikropalvelut käyttäjiltä selkeän rajapintakerroksen taakse yksinkertaistaen samalla asiakasohjelmistojen koodia. Rajapintayhdyskäytävä ei kuitenkaan tuo pelkkää autuutta, vaan sillä on myös oma taakkansa. Onhan siinä taas yksi ohjelmisto lisää kehitettävänä ja ylläpidettävänä, mutta siitä voi helposti muodostua myös pullonkaula.

3.5 DevOps-toimintamalli

Perinteisesti ohjelmistoyrityksessä kehitystiimit (*engl. development*) ja ylläpito (*engl. operations*) ovat erillään toisistaan, mutta DevOpsin tarkoituksena on yhdistää kehitys ja ylläpito tuoden nopeutta ja joustavuutta kehitykseen ja julkaisuun sekä vähentää erillisten tiimien välistä kommunikaatiokuilua (Ebert ym. 2016).

DevOps-toimintamallilla pyritään automatisoimaan ohjelmistokehityksen vaiheita (Zimmermann 2017). Perusperiaatteena DevOpsissa on lean-henkinen turhien työ-

vaiheiden karsiminen ja toistuvien operaatioiden automatisointi. Automaatio tehostaa jatkuvaa julkaisua ja monitorointia sekä nopeuttaa virheistä palautumista.

Ebertin ym. (2016) mukaan mikropalvelut ovat riippuvaisia DevOpsista, joten he suosittelevatkin palveluiden kehittämisen aloitettavan DevOps-strategian luomisella. Näin siitä saadaan hyöty heti alusta lähtien palveluiden integroimisessa ja lopulta myös julkaisussa. Tavoitteena on toimintamalli, jossa kehitettävä mikropalvelu on automaattisesti testattu ja milloin tahansa julkaistavissa. Automatisointi vähentää rutiinityön virheiden riskiä sekä karsii manuaalisia ja toistuvia työvaiheita. Jatkuva integrointi pitää huolen, että viallinen koodi huomataan nopeasti parantaen näin ohjelmiston laatua.

4 Digipalveluiden JYSOA-arkkitehtuuri

Jyväskylän yliopiston digipalveluissa on kohta vuosikymmenen ajan kehitetty JYSOA-arkkitehtuuria. Vuosien saatossa arkkitehtuuri on etsinyt muotoaan palvelukeskeisestä arkkitehtuurista päätyen lopulta kohti mikropalveluja.

Sovelluskirjo yliopistolla on laaja ja vielä 2000-luvun alkupuoliskolla yhteiskäytössä olevia sovelluksia hallittiin milloin missäkin yksikössä ympäri yliopistoa. Vuonna 2007 tietohallintokeskuksen perustamisen yhteydessä järjestelmäkehitystä pyrittiin keskittämään ja vaikkei se aivan täysimääräisenä toteutunutkaan, niin iso osa tiedekuntien ja yksiköiden ylläpitämistä sekalaisista sovelluksista on nykyisin digipalveluiden ylläpidossa, kuten esimerkiksi opintotietojärjestelmät ja opintorekisterit, oppimisympäristöt sekä yliopiston www-sivut. Keskitetty ylläpito ja kehitystoiminta on mahdollistanut yhtenäisemmän arkkitehtuurin kehityksen. Yhtenäinen arkkitehtuuri mahdollistaa myös tietojen jakamisen yksiköiden omille sovelluksille selkeästi määritellyin rajapinnoin.

4.1 Käyttötarpeet

Koulutuksen tietojärjestelmien osalta yliopistolla on menossa tällä hetkellä melkoinen mullistus, kun omatekoisista järjestelmistä ollaan siirtymässä kansalliseen yhteistyöhön. Opintotietojärjestelmä Korppi on tarjonnut kaikkien yliopistolaisten, niin perustutkinto-opiskelijoiden, tohtoriopiskelijoiden, opettajien, hallintohenkilöstön kuin avoimen yliopistonkin käyttöön monipuolisia työkaluja. Korppi on sisältänyt kurssihallinnan lisäksi tilavarauspalvelut, ryhmä- ja oikeushallinnan, työkaluja kyselyjen ja opinnäytteiden hallintaan, postilistat sekä kalenterin. Korppi on myös integroitu oppimisympäristöihin Moodleen, Optimaan ja Koppaan, kurssipalautejärjestelmä Webropoliin ja opintorekistereihin Rotiin ja ARKOon sekä yliopiston verkkomaksupalveluun Paymentsiin. Korpin lisäksi sisällönhallintajärjestelmä Plonen päälle on rakennettu monia palveluita, kuten yliopiston ja sen yksiköiden www-sivut, Koppa, Payments, opetusohjelma, liikuntamaksu sekä työnkulut.

Näiden monen mainitun palvelun yhteistoiminta vaatii luotettavaa ja jatkuvaa kommunikointia järjestelmien välillä ja tähän tarpeeseen digipalveluissa on yritetty vastata JYSOA-arkkitehtuurilla. Arkkitehtuurin kehittäminen lähti liikkeelle vuonna 2011 avoimen yliopiston ilmoittautumismaksujen siirtämisestä Korpista Paymentsiin. Hajautetun arkkitehtuurin perustaksi tuli AMQP-viestiväylä. Maksupalvelujen jälkeen seuraava iso kehityskohde digipalveluissa oli vanhan Jore-opintorekisterin uudistaminen. Roti-opintorekisterin kehittäminen alkoi vuonna 2012 ja Joren integraatiot huomioiden Roti-kehityksessä pyritiin rakentamaan JYSOA:n päälle itsenäisiä palveluita, joita Roti itsekin hyödyntää. Rekisterikin jaettiin kahtia Rehti-opiskelijarekisteriin ja Roti-opintorekisteriin. Rehti käsittelee kaiken henkilötiedon, jolloin Rotissa voidaan keskittyä opintotietojen — opiskeluoikeuksien ja suoritusten — käsittelyyn. Opintotiedot koostuvat erilaisista koodistotiedoista, kuten tutkinnot, opintorakenteet, organisaatiot ja arvolauseet, ja näitä tarvitaan monissa muissakin järjestelmissä. Koodistot irrotettiin omaksi erilliseksi palvelukseen.

Vuosikymmenen puolivälissä opintohallinnossa tuli tarve lukuvuosisuunnitteluun tarkoitetulle järjestelmälle. Tämän tarpeen täytti Peppi-konsortion kehittämä Peppi, josta yliopistolle otettiin käyttöön vain koulutuksen ja vuosisuunnittelun osio ja siksi järjestelmää kutsutaan Pepin sijaan KOVSiksi. Vuosisuunnitteluun kiinteänä osana kuuluvat tilavaraukset, joten oli loogista ottaa myös KOVSin resurssivarausosio käyttöön. Samoihin aikoihin yliopistolla alkoi olla liikehdintää kansalliseen yhteistyöhön Funidatan kehittämään Sisu-järjestelmään. Sisu otetaan käyttöön vaiheittain sen eri osioiden valmistumisen myötä. 2018 ensimmäiset opiskelijat pääsivät suunnittelemaan opintojaan siellä ja 2019 syksystä alkaen kaikki perustutkinto-opiskelijat suunnittelevat opintonsa ja ilmoittautuvat niihin Sisussa. Rekisterikäyttöliittymien valmistumisen jälkeen Rotinkin toiminnallisuudet siirretään Sisuun. Myöhemmin käytön on tarkoitus laajentua myös avoimen yliopiston käyttöön.

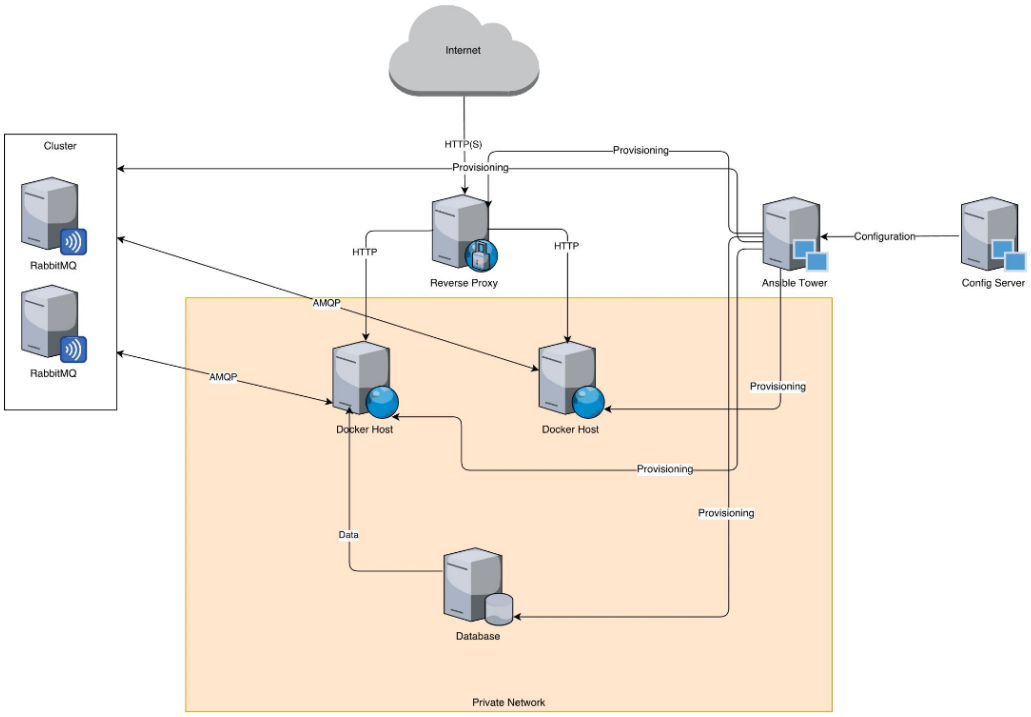
Nykypäivän langaton ja liikkuva elämäntyyli on antanut aihetta kehittää toimintoja mobiileiksi ja tästä poiki ensimmäinen versio MyJYU-sovelluksesta (2019) (kuva 7), joka koostaa käyttäjän kalenterit useammasta järjestelmästä ja sisältää mobiilikäyttöliittymän Navi-kampuskarttapalveluun.

4.2 Arkkitehtuuri

Alunperin yliopiston järjestelmät olivat erillisiä monoliittejä:

- käyttäjähallinta: Aman, AD, IDM
- oppimisympäristöt: Koppa, Optima, Moodle
- sisällönhallinta: Plone
- opintorekisterit ja opintotietojärjestelmät: Jore, Roti, ARKO, Korppi, Sisu

Kun järjestelmien välille tarvittiin tiedonsiirtoja, olivat ne ajastettuja integraatioita shell-skripteillä. Sittemmin mukaan tuli reaaliaikaisempia REST-rajapintoja. Lopulta kehitettiin yhteinen viestiväylä AMQP-protokollan toteuttavalla RabbitMQ-viestijonojärjestelmällä. Viestijonot mahdollistivat SOA-palveluiden käyttämisen, jotka pikkuhiljaa ovat pienentyneet yhä pienemmiksi mikropalveluiksi.



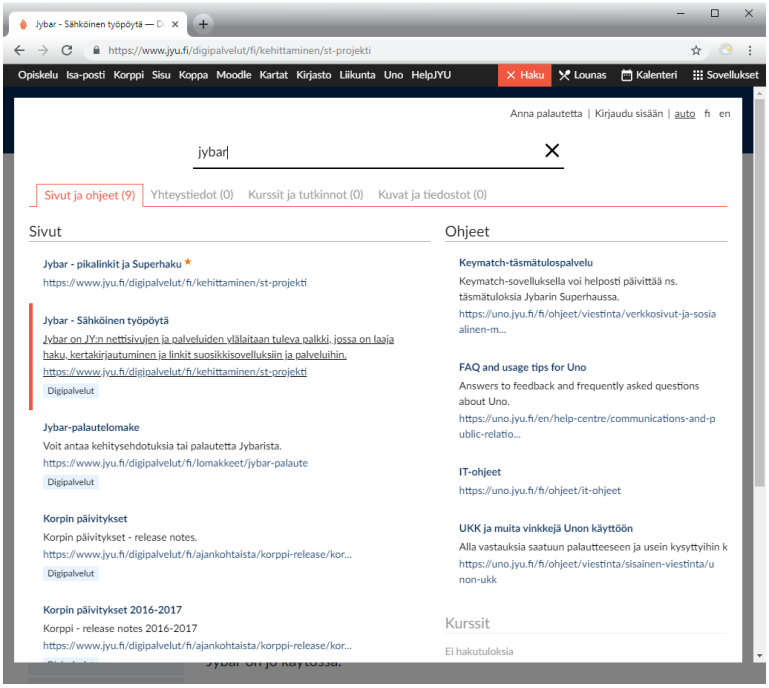
Kuva 4. JYSOA-arkkitehtuuri (Talaskivi 2017)

Kuvassa 4 on havainnollistettu JYSOA-arkkitehtuuria. Konttiympäristöksi on valittu Docker, jonka konttien hallinta tehdään Ansible Towerilla. Ansiblea varten Docker-

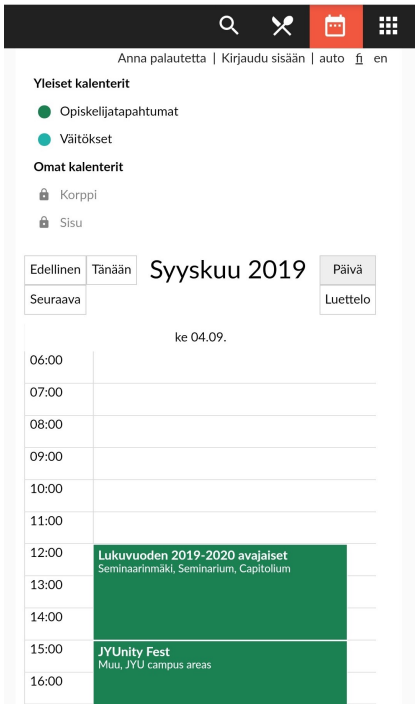
konttien konfiguraatiotiedostot on tallennettu Substance D -sisällönhallintajärjestelmään. JYSOA-palveluja ei ole aiemmin tarjottu digipalveluiden ulkopuolelle, mutta Sisun käyttöönoton myötä rajapintayhdyskäytävä mahdollistaa esimerkiksi kurssisuoritusten Sisuun viemisen JYSOA:n ulkopuolisista järjestelmistä.

JYSOA sisältää paljon eri kokoisia palveluja. Isompia palveluita ovat esimerkiksi aiemmin mainitut koodistopalvelu ja opiskelijatietopalvelu. Pienemmistä palveluista saa muun muassa halutun henkilön Korppi-kalenterin tai vaikkapa käyttäjätunnuksen perusteella henkilön Roti-id:n. Yksi palvelu osaa lähettää viestejä kurssipalautejärjestelmään Webropoliin. Tätä palvelua käytetään, kun omat palvelunsa selvittävät päättyvät kurssit ja toiset kokoavat Webropolin tarvitsemat kurssitiedot Korpista ja Sisusta. Tällä hetkellä kriittisimpiä palveluita on Sisussa tapahtuvista muutoksista ilmoittava palvelu, sillä Sisun ympärillä on jatkuvassa käytössä paljon muita järjestelmiä, kuten juuri mainittu kurssipalautejärjestelmä, oppimisympäristöt tai Korppi-postilistat.

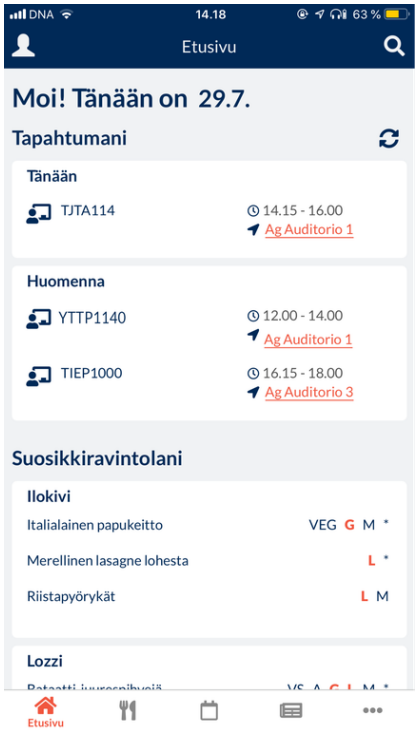
Kaikille hyvin näkyvä esimerkki JYSOA-palveluita käyttävästä sovelluksesta on sähköinen työpöytä Jybar (2018), joka kokoaa useammasta palvelusta käyttöliittymän (kuvat 5 ja 6), johon on helppo päästä monilta kaikkien käytössä olevilta yliopiston sivustoilta. Samoja palveluja hyödynnetään myös MyJYU-mobiilisovelluksessa (2019) (kuva 7).



Kuva 5. Jybar tietokoneen selaimessa



Kuva 6. Jybar mobiilissa



Kuva 7. MyJYU-mobiilisovellus (2019)

5 Yhteenveto

Perinteinen monoliittinen tapa rakentaa ohjelmistoja on edelleenkin järkevä tapa aloittaa ohjelmiston kehitys. Mikropalveluarkkitehtuurin rakentaminen tyhjästä on kallista, mutta monoliitin kasvaessa sen pilkkominen pienempiin osiin antaa sille pidemmän eliniän. Kun mikropalvelujen kehittäminen on organisaatiossa saatu käyntiin ja DevOps on juurtunut organisaation yleiseksi toimintamalliksi, on edullisempaa kehittää uudetkin toiminnot mikropalveluina, jolloin niiden ylläpito ja korvaaminen uudella helpottuu.

Jyväskylän yliopiston digipalveluissa on pikkuhiljaa siirrytty vanhojen isojen monoliittien jatkuvan refaktoroinnin sijaan palvelukeskeiseen arkkitehtuuriin ja siitä eteenpäin mikropalveluarkkitehtuuriin. Kaikki kolme elävät sulassa sovussa, mutta uudet integraatiot ja palvelut rakennetaan mikropalveluarkkitehtuuria noudattaen.

Kirjallisuutta

API Evangelist. 2012. *The Secret to Amazons Success Internal APIs.* https://apievangelist.com/2012/01/12/the-secret-to-amazons-success-internal-apis/. Viitattu 18. 10. 2019.

Biske, Todd. 2010. *SOA Governance : The Key to Successful SOA Adoption in Your Organization.* Packt Publishing. ISBN: 978-1-847-19586-9.

Calcote, Lee. 2018. *The Enterprise Path to Service Mesh Architectures.* Sebastopol: O'Reilly Media. ISBN: 978-1-492-04176-4.

Digipalvelut, Jyväskylän yliopisto. 2018. *Jybar - Sähköinen työpöytä.* https://www.jyu.fi/digipalvelut/fi/kehittaminen/st-projekti. Viitattu 18. 10. 2019.

———. 2019. *MyJYU-mobiilisovellus.* https://www.jyu.fi/digipalvelut/fi/kehittaminen/myjyu-mobiilisovellus. Viitattu 18. 10. 2019.

Dragoni, Nicola, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin ja Larisa Safina. 2017. "Microservices: Yesterday, Today, and Tomorrow". Teoksessa *Present and Ulterior Software Engineering*, toimittanut Manuel Mazzara ja Bertrand Meyer, 195–216. Cham: Springer International Publishing. ISBN: 978-3-319-67425-4. doi:10.1007/978-3-319-67425-4_12.

Ebert, Christof, Gorka Gallardo, Josune Hernantes ja Nicolas Serrano. 2016. "DevOps". *IEEE Software* 33 (3): 94–100. ISSN: 1937-4194. doi:10.1109/MS.2016.68.

Erl, Thomas, Ali Arsanjani, Grady Booch, Toufic Boubez, Paul C. Brown, David Chappell, John deVadoss ym. 2009. *SOA Manifesto.* http://www.soa-manifesto.org. Viitattu 18. 10. 2019.

Fowler, Martin. 2015. *Microservice Trade-Offs.* Saatavilla www-muodossa, https://martinfowler.com/articles/microservice-trade-offs.html. Viitattu 18. 10. 2019.

Koskimies, Kai, ja Tommi Mikkonen. 2005. *Ohjelmistoarkkitehtuurit.* Helsinki: Talentum.

Krause, Lucas. 2015. *Microservices: Patterns and Applications: Designing fine-grained services by applying patterns.* Kindle edition, version 1.5.

Lewis, James, ja Martin Fowler. 2014. *Microservices.* https://martinfowler.com/articles/microservices.html. Viitattu 18. 10. 2019.

Martin, Robert C. 2018. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Pearson Education. ISBN: 978-0-13-449416-6.

MSDN. 2016. *Chapter 1: Service Oriented Architecture (SOA).* https://web.archive.org/web/20160206132542/https://msdn.microsoft.com/en-us/library/bb833022.aspx. Viitattu 18. 10. 2019.

Newman, Sam. 2015. *Building Microservices.* Sebastopol: O'Reilly Media. ISBN: 978-1-49-195035-7.

Richardson, Chris. 2019. *Microservices Patterns.* Shelter Island: Manning Publications Co. ISBN: 978-1-617-29454-9.

Saarelainen, Ari. 2016. "Mikropalvelut korvaavat it-möhkäleet". *Tivi* 2/2016:18–25. https://www.tivi.fi/uutiset/mikropalvelut-korvaavat-it-mohkaleet/cd467d49-6606-30ad-84e4-1e196a00b97c.

Talaskivi, Jussi. 2017. *JYSOA-järjestelmäkuva.* Digipalveluiden sisäinen dokumentti.

Torre, Cesar de la, Kunal Deep Singh ja Vaclav Turecek. 2015. "Azure Service Fabric and the Microservices Architecture". *MSDN Magazine* December 2015:22–26. https://msdn.microsoft.com/magazine/mt595752.

Zimmermann, Olaf. 2017. "Microservices tenets". *Computer Science - Research and Development* 32, numero 3 (heinäkuu): 301–310. ISSN: 1865-2042. doi:10.1007/s00450-016-0337-0.https://doi.org/10.1007/s00450-016-0337-0.