Jukka Tykkyläinen

Sisäisen rajapintaekosysteemin ja rajapinta ensin -mallin edut keskitettyyn ESB-integraatiomalliin verrattuna

Tietotekniikan kandidaatintutkielma

4. tammikuuta 2020

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Jukka Tykkyläinen

Yhteystiedot: jukka.tykkylainen@gmail.com

Ohjaaja: Sanna Mönkölä

Työn nimi: Sisäisen rajapintaekosysteemin ja rajapinta ensin -mallin edut keskitettyyn ESB-integraatiomalliin verrattuna

Title in English: The value of internal API ecosystem and API first over centralized ESB integration model

Työ: Kandidaatintutkielma

Sivumäärä: 21+0

Tiivistelmä: Organisaation sisäinen rajapintaekosysteemi (engl. *API ecosystem*) ja rajapinta ensin -toimintaperiaate (engl. *API first principle*) muodostavat modernin perustan laajan organisaation integraatioarkkitehtuurille ja usean tiimin joustavalle ja ketterälle kehitykselle mahdollisimman tehokkaasti.

Avainsanat: API, ohjelmistorajapinta, ekosysteemi, ESB, rajapinta ensin, integraatio, mikropalveluarkkitehtuuri, integraatioarkkitehtuuri, yrityksen palveluväylä

Abstract: The internal API ecosystem and API first principle create a modern foundation for corporate integration architecture and fluent and agile development in a multi-team environment.

Keywords: API, ecosystem, ESB, API first, integration, microservice architecture, integration architecture, enterprise service bus

Sisältö

1 Johdanto

Monen organisaation sisäinen integraatioarkkitehtuuri on perustunut viime vuosien aikana ESB-järjestelmien (yrityksen palveluväylä, engl. *enterprise service bus*) varaan, jotka sekä järjestelmän että yrityksen arkkitehtuurin osalta noudattavat palveluorientoitunutta arkkitehtuuria (engl. *service oriented architecture*, lyhennettynä SOA). Palveluorientoitunut arkkitehtuuri on vuosituhannen alusta asti aktiivikäytössä ollut laaja käsite, joka pitää sisällään monenlaisia hajautettuja järjestelmäarkkitehtuureja, mutta tämän kirjoituksen kontekstissa sillä viitataan malliin, jota sen toteuttamiseen on useimmiten perinteisesti käytetty eli SOAP-protokollaan (lyhenne engl. *Simple Object Access Protocol*) pohjautuviin verkkopalveluihin (engl. *web services*); usein orkestroituna ESB-järjestelmän kautta. Lisäksi uuden toimintatavan vertailukohtana käytetään lisäoletuksena mallia, jossa osaaminen integraatiotyökalun käytöstä on keskittynyt tietyille yksilöille, jotka on usein organisaatiossa keskitetty yhteen tai useampaan tiimiin.

Viime vuosina suosiotaan hajautetuissa arkkitehtuureissa on nostanut SOA:n tietty lähestymistapa, mikropalveluarkkitehtuuri Ketola 2019, luku 2, jossa yrityksen järjestelmäkokonaisuus muodostuu pienistä, tyypillisesti yksittäiseen vastuualueeseen keskittyvistä järjestelmistä, jotka kommunikoivat toistensa kanssa verkon yli. Mikropalveluarkkitehtuuri tuo mukanaan uusia vaatimuksia myös integraatioarkkitehtuurille ja erityisesti sen kyvylle tukea modernia joustavaa kehittämistä ja markkinoiden muutoksiin vastaamista. Pienten ja yksittäiseen tehtävään suunniteltujen palveluiden kommunikaatioon sovelletaan enenevissä määrin aiempaa SOAP-protokollaa kevyempiä muotoja, joista selkeästi suosituin ja laajimmiten käytetty on puhdas HTTP-protokolla (lyhenne engl. *Hypertext Transfer Protocol*), jossa viestisisältö on json-muotoista (engl. *JavaScript Object Notation*); monesti myös ammentaen suunnittelufilosofiassaan REST-arkkitehtuurityylistä (hyvin monesti tällaisiin rajapintoihin viitataankin yleistermillä REST-rajapinta, joka on lyhenne engl. *Representational state transfer* Schuchmann 2018, luku 3).

Tässä tekstissä kuvataan ensin luvaussa 2 korkealla tasolla ensin mainitun keskitetyn integraatioosaamisen ESB-mallin haasteet, joka luo pohjaa sille, miten näitä ongelmia lähestytään eri tavoin ehdotetussa mallissa. Tämän jälkeen luvaussa 3 kuvataan miten rajapintakehityksen

vastuujaottelun muuttaminen rajapinta ensin -malliseksi ratkaisee keskitetyn mallin haasteita. Lopuksi luvaussa 4 esitetään HTTP/json-protokollaa hyödyntävä yleinen referenssi-integraatioarkkitehtuuri, keskeisimmät siihen liittyvät työkalut sekä millä tavoin se vastaa järjestelmäkehityksen muuttuneisiin haasteisiin.

2 Keskitetyn integraatio-osaamisen ja ESB-mallin haasteet

2.1 ESB-malli

ESB-pohjaisissa integraatioarkkitehtuureissa itse ESB-järjestelmä on varsin monimutkainen ja jo itsessään kehitysalusta. Monimutkaisessa toimintaympäristössä toimivat suuryritykset sisältävät monesti kokonaisarkkitehtuurissa useita kymmeniä, satoja tai jopa tuhansia järjestelmiä, jotka tarjoavat integraatiorajapintoja hyvin vaihtelevilla protokollilla, eri tasoisilla rajapintakuvauksilla ja tähän liittyvällä tuella. Monessa perinteisissä organisaatiomalleissa tuoteomistajuus on sekä liiketoiminnallisessa että teknisessä mielessä erillisillä tiimeillä, jotka hallitsevat kyseisten ydinjärjestelmien vaatimukset ja ylläpidon (sekä kehitystarpeet, mikäli kyseessä on sisäisesti kehitetty järjestelmä).

Missä tahansa SOA-mallissa tyypillisesti pyritään järjestelmien ja rajapintojen uudelleenkäytettävyyteen, mikä protokollien ja rajapintatyylien määrän huomioiden tarkoittaa monesti merkittävää työmäärää integraatiotasolla, jossa mallin mukaisesti pyritään tarjoamaan rajapinnat jollain tavalla yhtenäisessä formaatissa (ESB-järjestelmien parissa suosittu valinta on ollut SOAP-protokolla). Koska tämä yhtenäistetty rajapintakerros julkistetaan esimerkkinä käytettävässä mallissa ESB:n kautta, jää monesti rajapintojen määrittelyvastuu myöskin ESB-kehittämiseen erikoistuneille integraatiokehittäjille (tämä vastuu voi myös olla organisoitu hajautemmin, mutta tämä skenaario jätetään tämän tekstin kattaman kokonaisuuden ulkopuolelle) (Fowler 2014).

Huomioiden tilanteen, jossa lähdejärjestelmissä on rajapintoja lukuisilla eri tavoilla, ESB-järjestelmään täytyy toteuttaa monesti merkittäviäkin määriä tietokenttien muutoslogiikkaa eli ETL-logiikkaa (pura, muuta ja lataa, engl. *extract, transform, load*), joka pitää tyypillisesti sisällään tietojen yhdistämistä monesta eri lähdejärjestelmästä, mahdollisesti ESB:n omasta tietokannasta sekä erilaisten muutoskaavojen soveltamista sisältöön. Muutoslogiikan toteuttaminen integraatio-operaatioihin johtaa useimmiten myös siihen, että järjestelmien liiketoiminnallisia sääntöjä alkaa päätyä myös ESB-järjestelmän toteutuksiin.

Huomaa, että vaikka tässä luvussa keskitytään tietosisällön muutoksiin, niin ESB:llä on myös merkittävä määrä muita rooleja liittyen pääsyhallintaan, tietoturvaan ja muihin seikkoihin kokonaisarkkitehtuurissa, mutta näitä ei käsitellä tekstissä merkittävissä määrin.

Kaikki tämä johtaa siihen, että ne osastot, joilla on paras tieto järjestelmien toiminnasta ja siihen liittyvästä syväosaamisesta ovat lähdejärjestelmätiimeissä, kun taas vastuu uudelleenkäytettävien rajapintojen suunnittelusta ja toteutuksesta on tiimillä, jolla on vain karkeamman tason tietoa kyseisistä aihekokonaisuuksista (monesti se, mitä virallisissa ja monesti hyvin epätäydellisissä sekä terminologialtaan epätarkoissa määritysdokumenteissa on viestitty).

Nämä uudelleenkäytettävät rajapinnat ovat monesti myös hyvin tiiviisti sidottuja taustalla olevaan lähdejärjestelmään, mikä tarkoittaa sitä, että kun lähdejärjestelmään on tarve viedä rajapintaa jollain tavalla muuttava päivitys, on myös vastaavien ESB-rajapintojen ja toteutusten päivitykset suoritettava käsi kädessä. Tämä eri tiimien ja järjestelmien kytkeytyminen (engl. *coupling*) on tietotekniikassa hyvin tunnettu ilmiö, jota lähes kaikissa moderneissa arkkitehtuurityyleissä pyritään välttämään. Eri ESB-järjestelmät tarjoavat ratkaisumalleja tämän kytkeytymisen välttämiseen tai sen haittojen vähentämiseen, mutta tästä huolimatta lähdejärjestelmien tiimeistä eriytetty integraatio-osaaminen (mikä tyypillisesti ESB-järjestelmien konfiguroinnin kompleksisuuden johdosta on välttämätöntä) muodostaa organisaatiotason ryhmien kytkeytymisen.

Muita haasteita ESB-mallissa ovat ongelmaselvityksen vaikeutuminen ja automaattisten toimitusputkien rakentamisen haastavuus, jos ESB-operaatioiden toteutusformaatti ei ole versiohallintaystävällinen. Kun ESB-operaatioihin kerääntyy liiketoiminnallista logiikkaa, kuten edellä on kuvattu, ilmenee siellä myös luonnollisesti toteutusvirheitä. Kattavalla testaamisella näistä virheistä voi löytää osan, mutta on suorastaan tietotekniikan laki, että erityisesti realistisen yritysympäristön resurssirajoitteet huomioiden ei täysin virheettömään lopputulokseen pääseminen ole mahdollista ja tämä johtaa virheiden löytymiseen ajoittain tuotantokäytössä. Näissä tapauksissa ajan ollessa kriittinen tekijä on tärkeää, että olisi nopeasti ilmiselvää missä järjestelmässä (mahdollisesti pitkässäkin) kutsuketjussa vika esiintyi. ESB-mallin toteutus muodostaa helposti epäjatkuvuuskohdan, jossa virheselvitys on vietävä aina ensin siitä vastaavalle integraatiotiimille, jonka jälkeen tarkempi analyysi ohjaa virheelliseen

lähdejärjestelmään tai mahdollisesti virheeseen ESB-muutosoperaatiossa itsessään.

Virhekäsittelyn epäjatkuvuuskohdat ja mahdollinen läpinäkyvyyden puute ilmenevät helposti myös kehitysprosessin aikana, jossa monen tiimin mallissa keskitetyistä logiikkaa sisältävistä osista voi muodostua syntipukkeja ja kokonaiskehitys hidastuu.

Tämä kattaa merkittävimmät haasteet ESB-integrointimallissa, jossa integraatio-osaaminen on keskitetty ja luo pohjan seuraavalle kappaleelle, jossa esitetään toisenlaista lähestymistapaa, jonka avulla voidaan hajautettua kehitystä suorittaa välttäen useimmat esitetyistä haasteista.

2.2 Ohjelmistorajapinnat nyt ja aiemmin

Mikropalveluarkkitehtuureihin yhdistetään monesti hyvästä syystä tavoite, että palvelut keskustelevat hyvin määriteltyjen rajapintojen yli. Tällöin puhutaan järjestelmärajapinnoista eli API-rajapinnoista (engl. *application programming interface*). Kaikkeen verkko- ja järjestelmäliikenteeseen liittyy rajapinta, mutta se ei aina ole hyvin kuvattu. HTTP/json-protokollan noustessa sitä käytettiin aluksi paljon ilman hyviä rajapintakuvauksia, mutta viime vuosina myös tämän protokollan ympärille on kehittynyt useita eri kilpailevia rajapintojen kuvauskieliä, joista suosituin tällä hetkellä on OpenAPI (toiselta nimeltään Swagger), jonka uusin versio on 3.0. Ainoa maininnan arvioinen vaihtoehto OpenAPI:lle on RAML (lyhenne engl. *RESTful API Modeling Language*), mutta huomioiden sen merkittävästi suppeamman järjestelmätuen keskitytään tekstissä konkreettisiin rajapintakuvausformaatteihin liittyen viittaamaan OpenAPI-kuvauskieleen. Näihin kuvauksiin viitataan tässä tekstissä sekä termillä rajapintakuvaus että rajapintamäärittely.

OpenAPI-kuvauskieli on YAML-muotoista (jonka voi muotoilla olevan kevyempi vastine perinteisemmälle XML-formaatille, joka on lyhenne engl. *Extensible Markup Language*) ja se toteuttaa paljon samoja toimintoja kuin SOAP-protokollan vastine WSDL-skeema (engl. *web service definition language*). Erona WSDL-skeeman ja OpenAPI-kuvauksen välillä on muun muassa se, että OpenAPI pysyy lähtökohtaisesti yksinkertaisempana sisältämättä niin paljon XML:n ylitsepursuavien toiminnallisuuksien taakkaa, löyhempänä validointien suhteen sekä mahdollistaa tärkeänä seikkana esimerkkitiedon sisällyttämisen skeemakuvauk-

seen eri HTTP-paluukoodeilla.

Nämä luovat teknologisen pohjustuksen kappaleen pääaiheelle, mutta lisäksi on tärkeää pohjustaa myös kehitysmallin ja organisaation tausta, sillä modernin järjestelmäkehityksen maailmassa eri tiimien, toimintatapojen ja teknologioiden on tärkeää tukea toisiaan parhaiden tulosten saavuttamiseksi kovasti kilpaillussa ympäristössä. Järjestelmäkehityksen organisoinnissa on useimmissa yrityksissä siirrytty ketteriin malleihin, joihin monesti liittyy enenevissä määrin niin kutsuttu DevOps-kulttuuri (viitaten kehittämisen ja järjestelmien operoinnin roolien yhdistämiseen, engl. *development and operations*). Tämä tarkoittaa, että tiimit pyritään rakentamaan tarjottavien tuotteiden tai järjestelmäosaamisen ympärille sillä tavoin, että tiimistä löytyy kaikki tarvittava osaaminen hallinnoida kehityksen koko elinkaarta tarvemäärittelystä toteutukseen ja tuotantoon viemiseen sekä tuotannon ylläpitoon. Tämä toimintamalli poikkeaa perinteisestä erillisiin liiketoiminta-, kehitys- ja operointiorganisaatioihin eriytetystä organisaatiomallista ja on mahdollista viime vuosien järjestelmäkehityksen teknologioiden edistymisen sekä erityisesti pilvialustojen kyvykkyyden kasvun myötä (sekä julkipilvessä että pilvenomaisessa kehitystyylissä perinteisissä konesaleissa).

Keskitetyn integraatiotiimin kehitystyyli soveltui perinteiseen organisaatiojaotteluun oivasti, mutta myös devops-henkinen organisoituminen asettaa tavoitteeksi integraatioiden toteuttamisen osalta löytää uuden vaihtoehdon.

3 Rajapinta ensin -periaate ja sen vaikutus järjestelmäkehitykseen

3.1 Rajapinta ensin -periaate

Rajapinta ensin -periaatetta sovellettaessa (eli rajapinta ensin -lähestymistavalla) suurin muutos on siinä missä vaiheessa järjestelmän kehityskaarta uudelleenkäyttöön tähtäävä ohjelmistorajapinta laaditaan. Nimensä mukaisesti rajapinta ensin -periaatetta noudattavassa järjestelmäkehityksessä rajapinta suunnitellaan jo järjestelmän suunnittelun (tai ulkoa ostettavan järjestelmän käyttöön oton suunnittelun) yhteydessä. Tämä ehkä yksinkertaiselta ensikuulemalla vaikuttava muutos vaikuttaa todella fundamentaalisella tasolla useita kehitysprosessiin liittyviä vaiheita sekä muuttaa täysin miten vastuut jakautuvat eri tiimien välillä.

Ensimmäinen vaikutus periaatteesta on tiimivastuutukseen. Siinä kun aiemmin kuvatussa keskitetyn integroinnin mallissa uudelleenkäytettävän rajapinnan suunnitteluvastuu oli erillisellä tiimillä, jolla ei lähtökohtaisesti ole tietoa lähdejärjestelmien tarjoamasta tai sen loppuasiakkaiden vaatimuksista, niin rajapinta ensin -lähestymistavassa uudelleenkäytettävän rajapinnan suunnitteluvastuu on lähdejärjestelmästä vastaavilla henkilöillä. Lähdejärjestelmätiimille tulee tätä kautta aiempaan suhteutettuna lisävastuita, mutta toisaalta he myös parhaiten tietävät järjestelmän toiminnalliset yksityiskohdat sekä ymmärtävät mitä asiakastarvetta (sisäistä tai ulkoista) järjestelmä palvelee, jolloin heillä voi olettaa olevan myös parhaat lähtökohdat laatia oikeasti uudelleenkäytettävä rajapinta.

Parhaassa tapauksessa järjestelmän toimintalogiikasta vastaava liiketoimintavastuullinen näkee rajapinnan osana sisäistä tuotetarjoamaa ja kohtelee sitä samalla tavoin kuin mitä tahansa tarjottavan tuotteen tietoja. Vastaavat tekniset henkilöt laativat määrittelyn yhdyessä rajapintakuvauksen - mielellään ennen kuin varsinaisen järjestelmän tai palvelun toteutus edes alkaa kooditasolla. Kuvauksessa noudatetaan organisaation yhteisiä käytänteitä, jotka on ennen tämän mallin laajempaa käyttöönottoa tärkeää määritellä, jotta mahdollistetaan tiimien välinen hyvä tekninen kommunikaatio ilman, että esimerkiksi eri tiimien järjestelmiin tarvitsee tunnistautua tai hoitaa luvitus toisistaan aivan poikkeavilla tavoilla.

Näillä vastuumuutoksilla varmistetaan se, että rajapintakuvaus ei ensinnäkään jää vain jälkiajatukseksi, vaan siitä rakentuu kokonaisarkkitehtuurissa ensimmäisen luokan kansalainen. Toisekseen rajapinta on myös paremmin linjassa liiketoiminnallisten tavoitteiden kanssa, mikä heijastuu myös muiden sisäisten tiimien liiketoimintaohjautuvuuteen ja ketteryyteen positiivisessa mielessä. Tämä helpottaa myös mahdollisesti eri tiimeissä sijaitsevien liiketoiminnan ja kehitysvastuullisten välillä, sillä rajapintakuvauksen toiminnallisen tason versiokin on jo tyyliltään teknisluontoisempaa kuin järjestelmämääritykset monesti. Mallissa, jossa myös liiketoiminnallinen ja asiakastarpeiden osaaminen on kehittäjien kanssa samassa tiimissä, ei tämä liene merkittävä lisähyöty.

Rajapintakuvaus on alkuvaiheessa mahdollista laatia joko yllä kuvatulla tavalla (lähtien kuvauksen laatimisesta joko käsin tai erilaisia työkaluja hyödyntämällä) tai sitten muodostamalla se automatisoidusti lähdekoodista (toteutuskielestä riippuen avainmääritteet tulevat annotaatioiden, attribuuttien tai konfiguraation perusteella). Lähdekoodipohjainen rajapintakuvausten muodostustapa ei kuitenkaan ole suositeltava, sillä siihen liittyy muutamia haittapuolia: rikkovien rajapintamuutosten tekeminen vahingossa muuttuu todennäköisemmäksi ja tällä tavoin myöskin esimerkkivastausten sisällyttäminen rajapintakuvaukseen muodostuu haastavaksi, vaikka kehityskielen konfiguraatiomalli tukisikin sellaista.

Rikkovat rajapintamuutokset tarkoittavat sellaisia muutoksia, jotka muutokset yhteydessä pakottavat myös palvelua rajapinnan kautta käyttävät järjestelmät päivittämään omaan toiminnallisuuttaan. Esimerkkejä tällaisista muutoksista ovat aiemmin käytössä olleen kentän nimen tai tyypin muuttaminen niin, että se ei ole enää yhteensopiva vanhan versio kanssa. Rikkovia muutoksia eivät taas toisaalta ole tapaukset, joissa aiempaan versioon verrattuna on vain lisätty uusi kenttä, sillä HTTP/json-maailmassa rajapintakuvaukseen kuulumattomat yllättävät kentät tyypillisesti vain jätetään huomioimatta (toisin kuin SOAP-mallissa, jossa ne oletuksena rikkovat rajapinnan toiminnan).

Rikkovat muutokset ovat siis jotain, jota kehitysprosessissa tulisi välttää vahvasti - erityisesti, kun palvelu on jo tuotannossa. Välillä väistämättä kuitenkin ilmenee tarpeita tehdä myös rikkovia muutoksia ja tässä suhteessa problematiikka on sama kuin aiemmin kuvatussa ESB-mallissa, jossa joudutaan tekemään päivitykset synkronoidusti ESB:n ja lähdejärjestelmien välillä. Rajapinta ensin -lähestymistavassa onkin suositeltavaa tiedostaa tämä tarve jo etukä-

teen ja määritellä yleinen malli, jolla rikkovat muutokset käsitellään. Valitettava vaihtoehto voi riippua organisaatiosta tai organisaation sisällä voi olla tähän useita eri lähestymistapoja käytössä.

Yksi tyypillinen tapa rikkovien tai muuten vain laajojen muutosten käsittelyyn on (erityisesti jo tuotantokäytössä oleville järjestelmille) toteuttaa uusi versio rajapintakuvauksesta, joka on rinta rinnan käytettävissä vanhan version kanssa. Tämä voi tosin asettaa taustajärjestelmälle vaatimuksia, jotka eivät ole mahdollisia kaikissa tapauksissa.

Kaikki tämä tarkoittaa lähdejärjestelmästä vastaavalle tiimille lisävastuuta, mutta rikkovien muutosten tekemisen tiimillä kasvavassa työmäärässä on myös se hyöty, että tämä luonnostaan motivoi suunnittelemaan kunkin rajapintaversion tarkemmin. Toisaalta kokonaisuutta tarkastellessa mallilla tyypillisesti säästetään siinä, että useimmiten tarve erillisen integraatiotiimin toteutusprojekteille poistuu kokonaan madaltaen merkittävästi kommunikaatiosta ja projektihallinnasta koituvaa lisäkuormaa.

Toinen lähdejärjestelmätiimille tuleva lisävastuu on rajapinnan toteuttaminen. Mikäli aiempi toimintamalli on ollut tarjota mitä mahdollisesti ulkoa hankittu järjestelmä on tarjonnut sellaisenaan, niin tämä on kehitystiimille merkittävämpi kokonaisuus ja organisaation keskitettyjen toimintojen on hyödyllistä tarjota apua ja työkaluja toteutukseen - erityisesti tilanteissa, joissa valittu yleiskäyttöinen protokolla on eri kuin lähdejärjestelmän luonnostaan tukema.

Kannattaa huomata, että ESB-järjestelmien ja myöhemmin mainittavien sisäisen rajapintaekosysteemin mahdollistamien järjestelmien välillä on päällekkäisyyttä ja hajautettua ESB-mallia soveltavat organisaatio voivat jo toimia ESB:n kanssakin näiden periaatteiden mukaan.

Rajapinta ensin -lähestymistapa liittyy rajapintakuvauksen lisäksi myös ylläpidollinen puoli. Rajapintakuvaus on käytännössä organisaation sisäinen myös jonkinlaisen palvelutasolupauksen sisältävä sopimus, jonka tiimi lupaa täyttää ja lähestymistavan voidaan todeta olevan yksi esimerkki sopimusvetoisesta kehityksestä (engl. *contract driven development*). Muut palvelua hyödyntävät kehittäjät eivät ole kiinnostuneita toteutuksen yksityiskohdista, kunhan vain voivat luottaa, että palvelu taustalla vastaa luvatuilla vasteajoilla ja suorittaa luvatut toimenpiteet. Tämä ajattelutapa ja lupauksen pitäminen luo luottamusta organisaation

sisällä sekä selkeyttää rooleja ja kommunikaatiota. Myös suhtautuminen muihin tiimeihin muuttuu - toiset sisäiset tiimit nähdään tässä mallissa luonnostaan asiakkaina ja mieluiten kommunikaatiossa ja lupausten pitämisessä pyritään myös vastaavaan tasoon kuin yrityksen asiakkaille kommunikoidessa.

Rajapinnan ensin määrittelemällä on lisähyötyjä järjestelmän automatisoidun testaamisen osalta, sillä realististen palvelutason testitapausten määrittely on aivan käytännöllisestikin mahdollista aloittaa, kun rajapintakuvauksen ensimmäinen tekninen versio (esim. OpenAPI-formaatissa) on laadittu. Manuaalisesti laadittavien automatisoitujen testitapausten lisäksi nykyisin on tarjolla myös tapoja, joilla rajapintakuvaukseen pohjautuen voidaan laatia testejä esimerkiksi evolutionäärisiä algoritmejä hyödyntämällä (Arcuri 2019) tai vastaavasti kuin sopimusvetoisessa kehityksessä perinteisesti kooditasolla (Leitner ym. 2007).

Riippumatta millä tavoin organisaatiossa päätetään määrittää rajapintakuvaus ja toteuttaa rajapintojen takana tapahtuvat muunnosoperaatiot, niin on keskeistä, että järjestelmien väliseen liikenteeseen liittyvät yleiset seikat sekä rajapintojen elinkaaren hallinta on määritetty selkeästi organisaatiotason prosessitason hallintakehikossa ja rajapintojen suunnitteluohjeistuksessa. On tärkeää ottaa kantaa siihen, miten aiemmin mainittu versiointi suoritetaan ja miten vanhojen versioiden käytöstä poistaminen kommunikoidaan palvelun käyttäjille, miten yhtenäinen käyttäjien kirjautuminen sekä mahdollinen ulkoisten käyttäjien tunnistaminen mahdollistetaan ja hoidetaan.

Näissä seikoissa keskeisessä asemassa on sisäisen rajapintaekosysteemin yleiset osat sekä työkalut, joita käsitellään seuraavassa luvussa.

4 Sisäisen rajapintaekosysteemin yleiset osat ja työkalut

Rajapinta ensin -periaate vaatii toimiakseen tiimeille kohdistetun ohjeistuksen lisäksi myös työkaluja ja kokonaisarkkitehtuurin yhteisiä komponentteja. Tämän luvun kappaleissa kuvataan keskeisimpiä näistä komponenteista, niiden yleisen tason rooleja viitearkkitehtuurissa sekä minkälaisia eri realistisia vaihtoehtoja arkkitehtuuritason toteutusmalleihin liittyy. Viitearkkitehtuurin ja sitä kautta eri sen komponenttien roolikuvauksissa on jo otettu näkemystä, minkä taustalla on organisaatioille tyypillisesti tärkeitä kriteerejä, kuten esimerkiksi yksittäiseen tuotetarjoajaan lukittautumisen välttäminen niin paljon kuin realistisesti ja kustannustehokkaasti on mahdollista siltikin mahdollistaen ja tukien ketterää hajautettua kehitystä.

Rajapintaekosysteemillä viitataan monesti organisaation kolmansille osapuolille tarjoamiin rajapintoihin ja niistä muodostuvaan ekosysteemiin, mutta tässä luvussa keskitytään pääasiallisesti organisaation sisäisiin järjestelmärajapintoihin ja niistä muodostuvaan ekosysteemiin eli puhutaan aiemmin kuvatun keskitetyn integraatiotiimin ESB-mallin korvaavasta mallista.

4.1 Sisäinen kehittäjäportaali

Kehittäjäportaali on sekä ulkoisessa että sisäisessä käytössä integraatiokokonaisuuden näkyvin osa. Sinne rekisteröidään (mieluiten automatisoitujen toimitusputkien kautta) kaikki organisaation palveluiden välillä käytössä olevat sisäiset rajapinnat. Huomaa, että on täysin mahdollista, että sekä sisäinen että ulkoinen kehittäjäportaali (sekä viitearkkitehtuurin muutkin osat) on toteutettu samalla tuotteella - tai jopa samassa instanssissa riippuen organisaation tarpeista ja käyttötapauksista.

Kehittäjäportaalin merkittävin arvo muodostuu sen tarjoaman näkyvyyden sekä hallinnan keskittämisen kautta. Yhdistettynä aiemmin kuvattuihin suosituksiin siitä, miten ohjelmistorajapinta kuvataan ja että siihen sisällytetään esimerkkitietoa, kehittäjäportaali voi hyvien hakutoimintojen kautta tarjota täyden näkevyyden kehittäjille ja määrittelijöille palveluihin, jotka organisaatiolla on tarjolla sekä mahdollistaa kehityksen aloittamisen näitä palveluita vasten todella nopeasti.

Käytännön tasolla tämä toteutuu, mikäli kehittäjät voivat itse rekisteröityä kehittäjäportaaliin, etsiä eri kriteerein tarvitsemansa palvelut, kirjautua näiden rajapintakuvausten postilistoille, ladata itse rajapintakuvaukset ja aloittaen kehittämisen niitä vasten. Kun rajapintakuvaukseen on sisällytetty mukaan esimerkkitietoa eri HTTP-vastauskoodeille, niin myöhemmin kuvattavilla työkaluilla voi kehittäjän omalle koneelle nostaa paikalliset niin kutsutut huijariversiot palveluista (engl. *mock service*), joita varten kehitettävän järjestelmän kutsuja voi testata ilman pelkoa, että sekoittaisi jaettuja ympäristöjä.

Kehittäjäportaali voi myös rajapintakuvauksen esimerkkitietoon pohjautuen mahdollistaa vastaavien kutsujen suorittamisen jo portaalin kautta luoden myös jaetun n.s. hiekkalaatikkoympäristön (engl. *sandbox environment*) ilman, että sellaista tarvitsee kehittäjien toteuttaa erikseen.

Muita vastuita kehittäjäportaalilla on toimia tärkeänä tiedotuskanavana postilistakirjautumisten perusteella kehittäjille rajapintakuvauksen elinkaareen liittyvistä muutoksista, kuten vaikkapa uusista versioista ja tärkeimpänä vanhojen käytössä olevien versioiden vanhenemisesta ja käytöstä poistumisesta.

Kehittäjäportaalin oikeushallintarakenne on myös tärkeää suunnitella hyvin. Koska kyseessä on luontaisesti organisaatiotasolla keskitetty komponentti, niin virheellisellä oikeusrakenteella on täysin mahdollista päätyä vastaavaan ylikeskitettyyn tiimien toimintaa hidastavaan tilanteeseen kuin keskitetyssä ESB-mallissa. On siis suositeltavaa, että oikeusmallinnuksessa kehitystiimeille mahdollistetaan täysin itsenäinen heidän omien palvelujensa hallinnointi, uusien rajapintamääritysten lisääminen sekä niihin liittyvät versiopäivitykset.

Kehittäjäportaali tarjoaa myös tiedon osoitteista, joihin varsinaiset kutsut tulee kussakin ympäristössä kohdistaa (olettaen, että organisaatiolla on eritellyt tuotanto- ja testausympäristöt). Nämä osoitetiedot liittyvät vahvasti siihen minkä rajapintaporttikäytävän (engl. *API gateway*) takana kyseinen palvelu sijaitsee.

4.2 Rajapintaporttikäytävät

Rajapintaporttikäytävät ovat osittain jaettuja komponentteja integraatioarkkitehtuurissa, jotka toimivat keskitettyinä polkuina, joiden kautta lähes kaikki kutsut reititetään varsinaisille rajapinnat toteuttaville taustapalveluille. Rajapintaporttikäytävillä on useita eri tärkeitä rooleja, jotka ovat melko samankaltaisia ympäristöstä riippumatta, mutta porttikäytävien tarkka määrä, jaottelu ja organisaatioon spesifiin arkkitehtuuriin liittyvät tekijät riippuvat hyvin vahvasti käyttötapauksesta, joten siihen ei oteta merkittävästi kantaa tässä kappaleessa. Mainittakoon, että maantieteellisesti hajautetuiissa kokonaisarkkitehtuureissa on varmaankin suotavaa, että jokaisella etäisyyksien osalta merkittävällä maantieteellisellä alueella on ainakin oma porttikäytävä ihan puhtaasti vain liikenteen latenssin optimoimiseksi (erityisesti, mikäli porttikäytäviä käytetään myös kyseisen alustan sisäisen liikenteen välittämiseen).

Rajapintaporttikäytävät voivat olla käytössä joko ulkoisen tai sisäisen liikenteen vastaanottamiseen - tai sekä että, mikäli kokonaisarkkitehtuuri on rakennettu sitä tukemaan. Sisäiseen liikenteeseen nämä rajapintapalvelut toimivat erityisen hyvin mikropalveluarkkitehtuureissa Ketola 2019, luku 2.3.1.

Aiemmin mainittiin, että lähes kaikki rajapintakuvauksia vastaavat kutsut menisivät porttikäytävien kautta, mutta tämä riippuu myöskin organisaation alustojen arkkitehtuurista. Kaikkein keskeisintä viestit on reitittää tällä tavoin täysistään eri tavoin käsiteltyjen ja toiminnallisesti eroteltujen kontekstien välillä (joista esimerkkejä voisivat olla vaikkapa täysin eri tavoin palvelintasolla ylläpidetyt ja toisiinsa toiminnallisesti liittymättömät järjestelmät, joiden palvelimet sijaitsevat kuitenkin vaikkapa samassa konesalissa).

Esimerkki tapauksesta, jossa liikennettä ei ole todennäköisesti mielekästä reitittää näin on Kubernetes-klusterin sisäisten palvelujen välinen liikenne, joka monessa tapauksessa kuitenkin pohjautuu vastaavanlaisiin rajapintakuvauksiin. Tällaisessa tapauksessa rajapintakutsut, jotka tulevat klusterin ulkopuolelta palvelulle, joka on klusterin sisällä, pitäisi mitä todennäköisimmin olla reititetty porttikäytävän kautta.

Porttikäytävien tuotevalinnat voivat olla kehittäjäportaalista itsenäiset ja erillään, mutta monesti tämä aiheuttaa kehitysorganisaatiolle merkittävää lisätyötä. Porttikäytävät ja kehittäjäportaalit yhtenä pakettina tarjoavat tuotetarjoajat panostavat näiden yhteiseen toimintaan

ja mahdollistavat usein esimerkiksi aiemmin mainitun yhdenmukaisen tunnistautumismallin (nykyisin käytössä useimmiten OAuth2 tai OIDC -protokollien mukainen luvitus- ja tunnistustapa tilanteesta riippuen).

Yksi hyvin keskeinen ja kokonaisarkkitehtuurin luonnetta muovaava seikka on minkälaisia sisältöön liittyviä toiminnallisia vastuita porttikäytävälle annetaan. Yksi keskitetyn integraatiotiimin ESB-mallin keskeisimmistä ongelmista oli sinne helposti päätyvä liiketoimintalogiikka ja siitä aiheutuvat moninaiset ongelmat. Nämä haasteet vältetään yhdistämällä aiemmin kuvattu rajapinta ensin -lähestymistapa siihen, että porttikäytävissä kielletään tekemästä minkäänlaista viestien sisällön muotoilua eli älykkäät palvelut ja tyhmät putket -periaate (engl. *Smart endpoints and dumb pipes*) (Fowler 2014). Tämä pitää järjstelmien vastuut yhdenmukaisina ja selkeinä, mikä tukee myös tarvittaessa tuotantoympäristössä nopeaa ongelmanselvitystä.

Tämän myötä lopuksi voidaan listata vastuut, jotka porttikäytävällä tyypillisesti on:

- Viestimäärän rajoittaminen (sisäisten sekä ulkoisten palveluestohyökkäysten ehkäisemiseen) sekä tähän mahdollisesti liittyvä katkaisija (engl. *circuit breaker*)
- Tunnistus- ja luvitusvarmistukset määritysten mukaisesti - onhan kyseisellä sisäisellä tai loppukäyttäjällä pääsyoikeus hyödynnettävään palveluun (ja hyödynnettävällä tasolla huomioiden tuotanto- ja testiympäristöjen eri oikeutukset)
- Tarjoaa kutsuja suorittaville järjestelmille muuttumattoman osoitteen, johon kutsut kohdistaa, vaikka palvelut taustalla skaalattaisiin dynaamisesti
- Viestien reitittäminen kohdepalvelulle

4.3 Rajapintakuvausten suunnittelu- ja julkistustyökalut

Modernissa DevOps-hengessä on suositeltavaa pyrkiä kaikki koodina -malliin ja täysin automatisoituihin toimitusputkiin (engl. *CI/CD pipelines*). Rajapintakuvausten osalta suositeltava malli on pitää kuvausdokumenttia versiohallinnassa erillään itse palvelutoteutuksesta sekä hallinnoida siihen liittyviä muutoksia vaikkapa tyypillisten Git-pohjaisten yhdistämispyyntökäytänteiden (engl. *pull request*) ja niihin liittyvien koodikatselmointien kautta.

Rajapintakuvausten vieminen kehittäjäportaaliin on hyödyllistä automatisoida täysin aivan alkuvaiheessa, mikä helpottaa järjestelmään siirtymistä, pienentää pitkällä tähtäimellä kehitystiimien työmääriä (sillä kyseessä lienee useimmiten hyvin standardiprosessi eikä siis ensimmäisten kertojen jälkeen vaadi merkittävästi lisätyötä), vähentää inhimillisten virheiden määrää sekä muodostaa kattavan lokin muutoksista.

Valituista työkaluista riippuen mahdollisesti koko rajapintamäärityksen elinkaari voi olla toteutettavissa automatisoidusti.

Automatisoidun toimitusputken osalta tässä ei oteta kantaa yksittäisiin työkaluihin, sillä on tyypillisesti tuottavinta hyödyntää niitä, jotka organisaation kehitysosastoilla on jo valmiiksi käytössä.

Rajapintamääritysten julkaisemisen lisäksi työkalutukea tarvitaan myös määritysten laatimisessa. Vaikka esimerkiksi OpenAPI-kuvauskieli on ihmisluettavaa, niin sen käsin kirjoittaminen on silti virhealtista ja työlästä - puhumattakaan esimerkkitiedon täydentämisestä. OpenAPI-kielen ympärille on muodostunut tässä suhteessa jo hyvin kattava avoimen lähdekoodin ekosysteemi ja tiedot kaikista tarjolla olevista työkaluista voi löytää internetistä OpenAPI-formaatin sivuilta (osoitteesta https://openapi.tools/).

Kolmas kokonaisuus, jossa työkalut ovat hyvin tärkeässä asemassa, on koodin muodostaminen rajapintamäärityksistä sekä rajapinnan toteuttavan palvelun puolella että erityisesti sitä hyödyntävissä järjestelmissä. Toteuttavassa palvelussa muodostamisella on ilmiselvän duplikoidun työn vähentämisen lisäksi myös se lisähyöty, että palvelun sisäisen toteutuksen logiikka tulee eristettyä rajapinnan tietomallista lähes itsestään. Tämä liittyy myös aiemmin mainittuun perusteeseen sille, miksi rajapintamääritystä ei kannata muodostaa palvelun lähdekoodista vaan juuri toisin päin - lähdekoodista muodostamisessa myös sidos palvelun sisäiseen malliin jäisi helposti aiheuttaen entistä todennäköisimmin tahattomia rajapinnan rikkovia muutoksia.

Rajapinnan kutsujan näkökulmasta koodin muodostaminen rajapintamäärityksestä on keskeistä pääasiassa lisätyön välttämisen näkökulmasta, mutta olettaen, että rajapintakuvauksessa kenttien tyypit on määritelty hyvin tuo kuvauksesta muodostettu tietomalli erityisesti vahvasti tyypitettyjen ohjelmointikielten puolella jo itsessään lisää turvallisuutta kääntäjän

ja kehitysympäristön tyypillisten työkalujen kautta.

4.4 Kirjastoriippuvuuden haittapuolet

Erikseen mainittakoon vielä yksi tietty lähestymistapa rajapinnan käytön helpottamiseen ja siihen liittyviin haittapuoliin: rajapinnan jaetun kutsukirjaston käyttö eli kutsuvan järjestelmän toteutukseen käyttöön tuleva ohjelmistokirjasto, jonka kautta kutsun suorittaminen tehdään. Kutsukirjaston tarjoavat tyypillisesti samat kehittäjät, jotka toteuttavat taustapalvelun ja vaikka se soveltuu tiettyihin ympäristöihin, liittyy siihen useimmiten merkittäviä haittatekijöitä.

Kutsukirjaston käyttäminen kytkee järjestelmiä yhteen binääri- tai kooditasolla, mikä sotii suoraan vastoin mikropalveluarkkitehtuurien tavoitetta vähentää eri järjestelmien vahvoja kytköksiä.

Kirjastolähestyminen aiheuttaa myös sen, että monia eri ohjelmointikieliä käyttävässä organisaatiossa jokaiselle eri kielelle pitäisi tehdä oma versionsa. Tiimin koodia päätyy myös tätä kautta muiden kehitystiimien palveluihin, mikä hankaloittaa järjestelmän omistajuuden yksiselitteistä määrittelyä ja aiheuttaa täten haasteita vastuutuksessa.

Korkealla tasolla tämä lähestymistapa tekee epäselvempää siitä, missä selkeät rajat menevät, joihin pystyttää arkkitehtuuriset rajat ja tarkistuspisteet kokonaisjärjestelmän joustavan kestävyyden varmistamiseksi (Newman 2015, ss. 9).

Palvelukohtaisen rajapinnan kutsukirjastoa ei siis kannata useimmissa tilanteissa hyödyntää ilman hyvin keskeisiä erikoistarpeita.

5 Yhteenveto

Sisäinen rajapintaekosysteemi ja rajapinta ensin -periaate tarjoavat luontevan ja kattavan lähestymistavan integraatioille moderneissa mikropalveluarkkitehtuureissa, mutta eivät suoraan vastaa kaikkiin kysymyksiin, joihin perinteisempi ESB-malli vastasi. Näihin avoimiin kysymyksiin liittyvät muun muassa prosessitason hallintakehys, joka on räätälöitävä vahvasti yrityksen tarpeita vastaamaan sekä suuri määrä teknisiä yksityiskohtia liittyen erityisesti työkaluihin ETL-logiikan tai siis rajapintojen muotomuutosoperaatioiden ympärillä (jotka rajapinta ensin -mallissa ovat lähdejärjestelmistä vastaavien tiimien vastuulla).

Kokonaisuudessaan rajapinta ensin -lähestymistapa tukee hyvin myös DevOps-ajattelua ja ketterää kehitystä, joten on todennäköistä, että yhä useampi organisaatio omaksuu sen lähitulevaisuudessa.

Lähteet

Arcuri, Andrea. 2019. “RESTful API Automated Test Case Generation”. *arXiv* (Westerdals Oslo ACT, Oslo, Norway). doi:10.1109/QRS.2017.11. https://arxiv.org/abs/1901.01538.

Fowler, Martin. 2014. *Microservices.* Saatavilla WWW-muodossa, https://martinfowler.com/articles/microservices.html, viitattu 4.1.2020.

Ketola, Olli. 2019. “Mikropalveluarkkitehtuuriin perustuvien järjestelmien kehittämistä tukevat tekijät”. Tutkielma, Jyväskylän yliopisto.

Leitner, Andreas, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer ja Arno Fiva. 2007. “Contract driven development = test driven development - writing test cases”. Teoksessa *ESEC-FSE ’07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering,* 425–434. ACM. doi:10.1145/1287624.1287685. https://doi-org.ezproxy.jyu.fi/10.1145/1287624.1287685.

Newman, Sam. 2015. *Building Microservices.* United States of America: O’Reilly.

Schuchmann, Marcel. 2018. “Designing acloud architecture for an application with many users”. Tutkielma, University of Jyväskylä.