Tampereen yliopisto

Tero Mielikäinen

OLEMASSA OLEVAN INTEGRAATION SIIRTÄMINEN MODERNEILLE INTEGRAATIOALUSTOILLE JA -KEHYKSILLE

Diplomityö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Outi Sievi-Korte
Tarkastaja: Kari Systä
Syyskuu 2021

TIIVISTELMÄ

Tero Mielikäinen: ”OLEMASSA OLEVAN INTEGRAATION SIIRTÄMINEN MODERNEILLE INTEGRAATIOALUSTOILLE JA -KEHYKSILLE”
Diplomityö
Tampereen yliopisto
Tietotekniikka, DI
Syyskuu 2021

Integraatiot ovat olleet keskeisessä roolissa ohjelmistotuotannossa jo pitkään ja niiden merkitys kasvaa koko ajan. Sovellusten ja palveluiden laajentuessa kommunikointitarve ei ole enää yhden yrityksen sisällä, vaan järjestelmiin tuodaan myös ulkoisia palveluita. Integraatioilla on tarkoitus yhdistää nämä ulkoiset palvelut mahdollisimman saumattomasti ja niitä voidaan toteuttaa monella tapaa. Usein vanhemmat toteutukset ovat suoraan kahden järjestelmän välille luotuja räätälöityjä pisteestä-pisteeseen integraatioratkaisuja, jolloin usean integraation ylläpitäminen on usein kallista ja työlästä. Moderneilla integraatioalustoilla ja -kehyksillä voidaan keskeisen järjestelmän tai arkkitehtuurin avulla toteuttamista yhtenäistää, vähentää ylläpidollista työtä ja helpottaa uusien järjestelmien liittämistä osaksi integraatiojärjestelmää. Tämän tutkimuksen tarkoituksena on selvittää, kuinka olemassa oleva integraatiototeutus voidaan siirtää moderneille integraatioalustoille ja -kehyksille sekä minkälaisia hyötyjä ja kompromisseja siirtäminen tuo mukanaan.

Tutkimus toteutettiin tapaustutkimuksena, jossa toimeksi antavan yrityksen olemassa oleva Javaan perustuvan integraatiototeutuksen yksittäinen integraatiotapaus siirrettiin valituille alustoille ja kehyksille. Vaihtoehtoja integraatioalustoille oli paljon, mutta ne ovat periaatteiltaan samankaltaisia. Ratkaisevana tekijänä verrattavien alustojen valinnoissa oli yrityksen tarve saada integraatiototeutus kommunikoimaan sisäverkossa sijaitsevien resurssien kanssa. Vertailuun otettiin myös kaksi integraatiokehystä, jotka molemmat ovat JVM-pohjaisia. Vertailemalla alustoja pyrittiin selvittämään käytännössä mitä integraation siirtäminen vaatii ja minkälaisia kompromisseja ja parannuksia siirtyminen tuo aiempaan toteutukseen nähden. Alustojen välisiä eroja pyrittiin tuomaan esille erilaisien laatuominaisuuksien ja yrityksen esittämien vaatimusten perusteella.

Tutkimuksen perusteella selvisi, että olemassa olevaa ohjelmakoodia ei voida suoraan siirtää integraatioalustoille. Logiikka täytyy siirtäessä muuttaa ensin teknologiariippumattomaan muotoon ja toteuttaa integraatio sen perusteella alustan omilla työkaluilla. Vanhan toteutuksen pohjalta luodut uudet integraatiototeutukset tuovat kuitenkin merkittäviä parannuksia yrityksen nykyiseen tilanteeseen. Alustat tarjoavat helpon ja yhtenäisen tavan luoda integraatioita nopeasti ja tuovat vain vähän kompromisseja toteutusmielessä aiempaan toteutukseen verrattuna. Kehykset tuovat yhtenäistä rakennetta ohjelmoimalla tehtyyn toteutukseen, ilman ulkopuolisia järjestelmiä. Lopulta yrityksen tarpeisiin parhaat vaihtoehdot valittiin vertailemalla taulukoituja tuloksia laatuominaisuuksista, vaatimusten toteutumisesta ja alustoille suoritetun suorituskykytestin perusteella.

Avainsanat: integraatio, integraatiokehys, iPaaS, EiPaaS, integraatioalusta

ABSTRACT

Tero Mielikäinen: MIGRATING INTEGRATIONS TO MODERN INTEGRATION PLATFORMS AND FRAMEWORKS
Master's thesis
Tampere University
Information Technology, MSc
September 2021

Integrations have been critical part of the software industry for a long time, but the importance is only growing. As applications and services expand, the communication is no longer inside one specific domain and new services are brought externally from 3rd parties. Integrations serve the purpose of connecting these external services to work seamlessly with one other and there are many ways to implement integrations. In the past, most integrations were created as point-to-point connections which are specifically tailored for two systems integrating with each other. This makes maintenance costly and laborious. With modern integration platform or framework, implementations can be created more uniformly, they require less maintenance work and new systems can be connected more easily. The purpose of this thesis was to research how to migrate existing integration implementations to modern integration platforms and frameworks, and as well to evaluate the benefits and compromises that the migration brings.

Thesis was done as a case study, where implementation of single integration case from company's Java-based solution was selected to be the target case of the migration research. There were multiple options for integration platforms to choose from, but they were very similar on how they work. Three platforms, which are different enough, were chosen for comparison group. To make comparison more complete, two JVM-based integration frameworks were chosen as reference. Migration was done for all platforms and frameworks selected to comparison group. With the migration the purpose was to recognize what it requires in practise and if there's any compromises and benefits compared to previous implementation.

As a result, it was found out that existing code cannot be migrated directly to any of the researched platforms. Instead, the process flow of the integration case needs to be transformed into technology independent form, that can be used as a base for implementing integration solutions in the integration platforms. Compared to current solution, the implementations made during this study gave significant improvements. Integration platforms provided easy and fast way to create integration flows with very few compromises. Integration frameworks provide unified structure for programming-based solution without introducing separate third-party systems. Finally, best options for the assigning company were selected by comparing tabulated results of quality properties, pre-defined requirements, and performance tests.

Keywords: integration, integration framework, iPaaS, EiPaaS, integration platform

ALKUSANAT

Tämä diplomityö on tehty Tampereella toimivalle ohjelmistoyritys Oscar Software Oy:lle. Integraatioalustojen ja -kehysten tutkiminen oli heillä ollut asialistalla jo pidemmän aikaa, mutta resurssien takia jäänyt tekemättä. Haluan kiittää yrityksen puolelta Tommi Ritolaa mielenkiintoisesta aiheesta ja tuesta työn aikana. Suuret kiitokset myös työn ohjaajana sekä tarkastajana toimineelle Outi Sievi-Kortteelle ja toiselle tarkastajalle Kari Syställe.

Erityinen kiitos kuuluu kuitenkin perheelleni ja ystävilleni, jotka ovat auttaneet vaikeina hetkinä ja tuoneet vastapainoa työn tekemiseen. Isä ja äiti ovat olleet henkisen tuen lisäksi myös tarvittaessa rahallisena tukena, jolloin on voinut aina keskittyä täysillä opiskeluun. Tämä ei ole itsestäänselvyys, joten kiitos siitä.

Tampereella 18.09.2021

Tero Mielikäinen

SISÄLLYSLUETTELO

1. JOHDANTO 1
2. INTEGRAATIOT 4
2.1 Yleisesti 4
2.2 Integraatiotavat 4
2.3 Ratkaisumallit 7
2.3.1 Point-to-point – pisteestä pisteeseen 8
2.3.2 Hub-and-Spoke – keskitetty malli 10
2.3.3 Enterprise Service Bus - palveluväylä 13
3. INTEGRAATIOALUSTAT JA -KEHYKSET 15
3.1 Yleistä 15
3.2 Tutkittavat alustat 16
3.2.1 Boomi AtomSphere 17
3.2.2 Anypoint Platform ja Mule 18
3.2.3 Azure Logic Apps 19
3.3 Verrattavat integraatiokehykset 20
4. LÄHTÖTILANNE JA SUUNNITTELU 22
4.1 Integrointitapaus 22
4.2 Yrityksen järjestelmät ja integraatioratkaisu 23
4.3 Ongelman kuvaus ja motivaatio 24
4.4 Toteutuksen vaatimukset 25
4.5 Teknisen toteutuksen suunnittelu 26
5. INTEGRAATION SIIRTÄMINEN 28
5.1 Alustat 28
5.1.1 Boomi 28
5.1.2 Anypoint 34
5.1.3 Azure 39
5.2 Kehykset 45
5.2.1 Spring Integration 46
5.2.2 Apache Camel 49
5.3 Yhteenveto 51
6. TULOKSET 52
6.1 Yleistä tuloksista 52
6.2 Vaatimusten toteutuminen 55
6.3 Parannukset edelliseen toteutukseen verrattuna 58
6.4 Kompromissit 59
7. LOPPUPÄÄTELMÄT 61
LÄHTEET 66

KUVALUETTELO

Kuva 1 Yleinen kulkukaavio viestipohjaisesta integraatiosta [4] *7*
Kuva 2 Synkronisen pisteestä pisteeseen -integraation kulku [10] *8*
Kuva 3 Asynkroninen pisteestä pisteeseen -integraatio jonoilla toteutettuna [10] *9*
Kuva 4 Yksinkertainen pisteestä pisteeseen -verkko [8] *9*
Kuva 5 Monen järjestelmän pisteestä pisteeseen -integraatioverkko [8] *10*
Kuva 6 Yksinkertaistettu keskitetyn mallin integraatioverkko [10] *11*
Kuva 7 Yksinkertainen Publish-and-Subscribe viestintämalli hubilla toteutettuna [9] *12*
Kuva 8 Palveluväylän yleiskuva [14] (ch. 1.7.2, muokattu) *14*
Kuva 9 Unifaun Onlinesta tulostetut lähetystarrat *23*
Kuva 10 Tiedon muokkaukseen käytettävä Map-elementti *29*
Kuva 11 Rahtikirjanumeron asettaminen muuttujaan Set Properties-elementissä *30*
Kuva 12 Lopullinen Boomi-integraatioprosessi *33*
Kuva 13 Automaattisesti luotu Swagger dokumentaatiosivu *34*
Kuva 14 Muuttujan asettaminen Mulen Transform-elementissä *35*
Kuva 15 Unifauniin käytetyn HTTP-kutsuelementin asetuksia *37*
Kuva 16 Lopullinen Mule-integraatioprosessi *38*
Kuva 17 Mulen tiedon muunnoksessa tullut virhe lokitiedostossa *39*
Kuva 18 Logic Apps-yhdyskäytävän yleisnäkymä *41*
Kuva 19 Lopullinen integraatioprosessi Logic App designer -näkymässä *43*
Kuva 20 Kauan kestänyt kutsu Logic App -rajapintaan Postmanissa *44*

TAULUKKOLUETTELO

Taulukko 1 Azuren integraatiopalvelut käyttötapauksineen [40] *19*
Taulukko 2 Alustojen tarkasteltavia ominaisuuksia *52*
Taulukko 3 Suorituskykyvertailut ja niistä kerätyt tilastoarvot *54*
Taulukko 4 Alustojen vaatimuksien toteutuminen *56*

LYHENTEET JA MERKINNÄT

CSV	Comma Separated Values, tiedostomuoto, jossa sisältö erotellaan pilkuilla, tai jossain tapauksissa puolipisteellä.
XML	Extensible Markup Language. Laajennettu merkintäkieli, tiedostomuoto, jolla voidaan esittää rakenteellista tietoa.
iPaaS	engl. Integration Platform as a Service, integraatioalustan tarjoaminen palveluna
WMS	engl. Warehouse Management System, varastonhallintajärjestelmä
API	engl. Application Programming Interface, sovellusohjelmointirajapinta.
URL	engl. Uniform Resource Locator, verkkosivun osoite
SOA	engl. Service-Oriented Architecture, palvelupohjainen arkkitehtuuri
SOAP	engl. Simple Object Access Protocol, sovellusten välinen viestipohjainen tietoliikenneprotokolla.
ESB	engl. Enterprise Service Bus, integraatiomalli, jossa yksi osa toimii keskeisenä palveluväylänä.
EIP	engl. Enterprise Integration Patterns, integraatiototeutuspohjien kokoelma
EAI	engl. Enterprise Application Integration, yrityssovellusten integrointiin käytetty malli
JVM	Java Virtual Machine, käännettyjä Java ohjelmia suorittava abstrakti virtuaalikone.
SQL	engl. Strucutred Query Language, relaatiotietokannoissa käytettävä kyselykieli
HTTP	engl. Hypertext Transfer Protocol, selainten ja verkkopalvelimien väliseen tiedonsiirtoon käytettävä protokolla

.

1. JOHDANTO

Integraatiot eivät ajatustasolla ole mitenkään uusi asia, eivätkä rajoitu pelkästään ohjelmistoihin tai mihinkään teknologiaan. Integraatiolla tarkoitetaan kahden toisistaan riippumattoman kokonaisuuden yhdistämistä ja vuorovaikutuksen mahdollistamista. Käytännönläheisenä esimerkkinä voidaan ajatella siltaa. Sillan tehtävä on yhdistää, eli integroida ihmisiä ja palveluita vaikeakulkuisten paikkojen, kuten virtaavan joen yli. [3]

Informaatiotieteiden nopean kehityksen ja integraatiotarpeiden lisääntyessä yritysten integraatioille on muodostunut omat markkinat muun ohjelmistokehityksen oheen. Integraatioita tarjoavat alustat ja kehykset kehittyvät nopeasti, kuten kaikki muukin ohjelmistoalalla. Mukautetut integraatiototeutukset vanhenevat nopeasti, ellei niitä pidetä ajan tasalla. [6]

Tutkimuksen aiheena on, kuinka olemassa olevan integraation siirtäminen integraatioalustalle (iPaaS, Integration Platform as a Service) tapahtuu käytännössä. Nämä alustat ovat pilvipohjaisia integraatiokokonaisuuksia, joilla voidaan hallita useampia integraatioprosesseja ja rajapintoja niiden aktivointiin. Alustojen lisäksi tutkitaan kahta integraatiokehystä (engl. integration framework), jotka laajentavat ohjelmointikieltä tarjoten siihen integraatioihin tarkoitettuja toiminnallisuuksia. Alustojen valinnassa pääasiallisena vaikuttavana tekijänä on yrityksen sisäverkossa toimiminen, joiden lisäksi valittaville alustoille määritellään myös erillisiä toteutukseen liittyviä vaatimuksia. Useampaa alustaa ja kehystä vertaillen saadaan mahdollisimman kattavat vaihtoehdot yrityksen tarpeisiin.

Lopputuloksena pyritään saamaan vastauksia seuraaviin tutkimuskysymyksiin: Kuinka olemassa olevan integraatiototeutuksen saa siirrettyä integraatioalustalle? Mitä hyötyjä alustalle siirtymisellä saadaan? Mitä kompromisseja siirtymisessä joudutaan tekemään? Ensimmäisen kysymyksen osalta selvitetään vaihtoehtoja olemassa olevan ohjelmakoodin siirtämiseksi alustalle. Koska koodia voi olla paljon ja alustasta riippuen koodia ei välttämättä suoraan sellaisenaan tueta, voi integraatioiden siirtäminen olla haastavaa. Lopulta kaikkien kysymysten avulla pohditaan, tarjoaako alustalle siirtyminen niin merkittäviä hyötyjä, että se kannattaa ohjelmoidun toteutuksen sijasta vai onko tarkoitukseen sopiva integraatiokehys perustellumpi ratkaisu.

Tutkimusmenetelmäksi työhön valikoitui tapaustutkimus, sillä tutkimuksella pyritään selvittämään vastaukset tutkimuskysymyksiin hyödyntäen yrityksen tarjoamaa yksittäistä

integraatiotapausta. Integraatioalustat ovat toteutusmielessä teknologiariippumattomia. Tämä tarkoittaa sitä, että vaikka tutkimuksessa ensisijaisesti keskitytään tietyllä tekniikalla toteutetun integraation siirtämiseen, on lopputulosten ja siirtämiseen käytettyjen menetelmien tarkoitus olla käyttökelpoisia myös teknologiasta riippumatta muissa integraatiototeutuksissa. Tämä tutkimus tehdään toimeksiantona Tampereella toimivalle Oscar Software Oy ohjelmistoyritykselle.

Toisessa luvussa käsitellään integraatioita ja niiden toteutusmenetelmiä teoriapohjalta. Teoriaosuudessa käydään läpi integraation perusperiaatteita ja myös hieman vanhempia tapoja tehdä niitä. Periaatteiden lisäksi esitellään erilaisia viestintämalleja eri arkkitehtuureille. Integraatiot esitellään korkealla tasolla ja kappaleen tarkoitus on johdatella integraatioihin, jolloin työn kulkua on helpompi seurata.

Kolmannessa luvussa käydään läpi integraatioalustan ja -kehyksen piirteitä. Lisäksi käsitellään mitä tarkoitetaan integraatioalustalla ja -kehyksellä. Edellisen luvun integraatioiden teoria liittyy olennaisesti varsinkin kehyksillä toteutettaviin integraatioihin, sillä ne tehdään itse, eikä kehys rajoita käytettävää arkkitehtuuria. Alustoilla integraatioteorian tuntemus ei ole niin suuressa roolissa, sillä ne sisältävät valmiita komponentteja, sekä usein määräävät käytettävän arkkitehtuurin. Lopuksi luvussa esitellään toteutukseen valittuja integraatioalustoja ja -kehyksiä yleisellä tasolla. Integraatioalustat ovat yleisesti hyvin samanlaisia, mutta käytännön eroja löytyy käyttöliittymistä, sekä kuinka integraatiot julkaistaan.

Toteutettavan integraatiotapauksen suunnitteluun ja yrityksen sisäisiin järjestelmiin keskitytään luvussa neljä. Suunnittelu osassa luodaan pohja integraatiotapauksen siirtämisen toteuttamiselle. Yrityksen sisäisiä järjestelmiä ei käydä kovin syvällisesti, sillä se ei ole tutkimuksen tarkoituksen kannalta olennaista. Näiden lisäksi käydään läpi alustoille määritettyjä käytännön vaatimuksia ja motivaatiota siirtämiselle.

Viidennessä luvussa aloitetaan valitun integraatiotapauksen toteuttaminen valituilla integraatioalustoilla ja vertailtavilla integraatiokehyksillä. Pohjana siirtämiselle on aiemmassa luvussa integraatiotapauksen aiemman toteutuksen pohjalta luotu prosessikaavio. Luoduista toteutuksista käydään läpi tarvittavien järjestelmän osien asennukset, toteutuksessa käytettävät elementit ja niiden määritykset sekä muita huomioita toteutukseen liittyen.

Luvussa kuusi esitellään yleisesti tutkimuksen tuloksia ja vertaillaan alustoja erilaisten laatuominaisuuksien mukaan. Laatuominaisuuksiin kuuluu virheidenhallinta, integraatiototeutusten luonnin helppous, tiedon muuntamisen yksinkertaisuus, luotujen toteutus-

ten julkaisemisen sujuvuus ja toteutusten suorituskyky. Myöhemmin luvussa tarkastellaan etukäteen määriteltyjen vaatimusten toteutumista ja käsitellään alustalle siirtymisessä saavutettavia etuja sekä kompromisseja, joita alustalle siirtyminen tuo mukanaan.

Viimeisessä luvussa tutkimuksen kulkua kerrataan ja arvioidaan saatuja lopputuloksia tutkimuskysymysten pohjalta. Arviointi sisältää tulosten käytettävyyttä yrityksen tarpeisiin ja tulevaisuuden kehityskohteita, joita ei tämän tutkimuksen aikataululla pystytty suorittamaan tai niitä ei osattu etukäteen ottaa huomioon.

2. INTEGRAATIOT

2.1 Yleisesti

Integraatio tarkoittaa yleisesti kahden, tai useamman toisistaan eristyksissä olevan kokonaisuuden, kuten ohjelmiston, sovellusten, yksittäisen ohjelman, tai laitteen välille luotua vuorovaikutusprosessia. Tällä prosessilla saavutetaan kommunikointiyhteys kokonaisuuksien välille. Kommunikaatioyhteydellä saavutetaan edelleen osapuolten yhteistoimivuus, jolla pyritään siihen, että ne toimisivat mahdollisimman saumattomasti keskenään [1]. Osapuolet voivat olla täysin eri tekniikoilla ja tietomalleilla toteutettu, jolloin on integraatioprosessin tehtävä huolehtia tiedot kohteiden vaatimissa muodoissa oikeaan osoitteeseen [10].

Nopean teknologisen kehityksen takia ohjelmistot vanhentuvat helposti. Kun teknologia on tarpeeksi vanha, voi tulla pulaa sitä osaavista työntekijöistä. Kehityksessä jälkeen jääminen aiheuttaa kehitysvelkaa, joka osaltaan voi aiheuttaa turhaa monimutkaisuutta ylläpitoon. Näistä syistä myös integraatioiden toteuttaminen on usein yrityksille hankalaa. Syitä voi olla monia: resurssit, osaaminen tai yrityksen sen hetkinen tilanne, jossa sovellus tai ohjelmisto on voinut olla yrityksen ainoa tuote, eikä rajapintoja ole aiemmin jaettu ulkopuoliseen käyttöön. Usein myös rajapintojen dokumentointi unohtuu helposti tai se jää vajaaksi. Dokumentaatioiden ollessa puutteellisia, on vaikea saada käsitystä liikkuvasta tiedosta, ohjelmien tarjoamista rajapinnoista tai niiden vaatimista tietomalleista ja sisältötyypeistä [6].

2.2 Integraatiotavat

Integraatioihin ei ole yhtä tiettyä lähestymistapaa, joka täyttäisi kaikki tarpeet. Tavat eivät sulje toisiaan pois, vaan niitä voidaan käyttää tapauskohtaisesti tarpeen mukaan. Integraatiotavalla tarkoitetaan tapaa toteuttaa integraatio kahden tai useamman järjestelmän välillä. Käytettäviä tapoja voidaan jakaa neljään isompaan korkean tason kategoriaan menetelmän mukaan; Tiedostonsiirto (engl. File Transfer), Jaettu tietokanta (engl. Shared Database), Etäältä kutsuttavat aliohjelmat (engl. Remote Procedure Invocation) ja Viestintä (engl. Messaging) [3] (s.41).

Tiedostonsiirtoon ja tiedostoihin perustuvassa integroinnissa ohjelmat luovat, käyttävät ja siirtävät yhteiskäyttöistä tietoa erillisten tiedostojen kautta. Tiedostomuoto on usein jonkin standardin mukaista, esimerkiksi pilkuilla erotellut arvot, eli CSV (Comma Separated Values) tai laajennettava merkintäkieli, kuten XML (Extensible Markup Language).

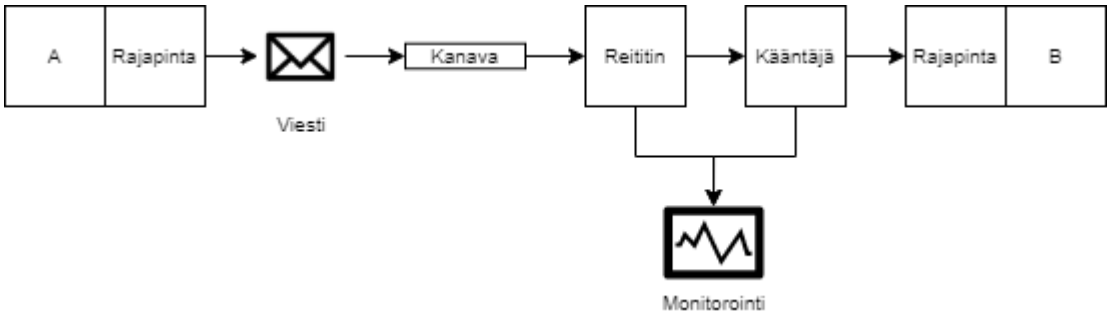
Yhdenmukaisilla tiedostostandardeilla jokainen ohjelma lukee tietoa samalla tavalla. Tiedostojen jakamisessa etuna on sen helppous. Integroitavasta osapuolesta ei tarvitse tietää sen teknistä toteutusta. Heikkouksia tällä lähestymistavalla on muun muassa, että tiedostot ovat yleisesti alttiita ylikirjoituksille, korruptoitumiselle ja poistamiselle. Tällöin tiedon katoamisen riski on kohtalaisen suuri. Lisäksi tiedostoista tulee nopeasti pullonkaula monimutkaisemmissa integroinneissa, jolloin tiedostoja pitäisi prosessoida paljon ja usein. Se lisää myös ohjelmoijan työtä, koska tiedoston lukeminen ja siihen tiedon lisääminen täytyy toteuttaa itse ja ottaa huomioon esimerkiksi erikoismerkit ja muut erityishuomiot kuten merkistökoodaukset [3] (s. 43–46).

Jaetulla tietokannalla voidaan korjata muutamia tiedostojen jakamiseen liittyviä haasteita. Ohjelmien käyttäessä samaa tietokantaa voidaan olla varmoja, että ne toimivat yhdenmukaisesti. Tietokanta mahdollistaa tiedostoihin verrattuna nopean tietojenpäivityksen, jolloin jokainen ohjelma käyttää uusinta tietoa ilman isoja viiveitä. Samaan aikaan tehdyt muokkaukset tai virheiden hallinta voidaan hoitaa transaktioiden avulla. Transaktiolla tarkoitetaan tietokantaan suoritettavaa tapahtumaketjua tai komentosarjaa, joka peruutetaan kokonaan, mikäli jokin sen osa epäonnistuu. Yhtenäisellä tietokantarakenteella ei tarvitse varmistaa, onko luettava tieto jonkin tiedostomuodon mukaista. Yleinen tietokantatoteutus on relaatiotietokanta, josta tietoa voidaan hakea relaatiotietokantojen kyselykielellä, eli SQL (Structured Query Language) -kyselyillä. Yhtenäisen rakenteen kehittäminen jokaiselle ohjelmalle sopivaksi on kuitenkin äärimmäisen hankalaa. Tämä voi johtaa monimutkaiseen rakenteeseen, jota saattaa olla vaikea ylläpitää ja tulkita. Tiedostojen tapaan tietokannasta voi tulla myös pullonkaula, jos liikennettä on paljon [3] (s. 47–49). Yhteinen tietokanta ei ole myöskään välttämättä järkevä vaihtoehto, jos integraatiota tarvitsee tehdä ulkopuolisen järjestelmän kanssa. Vaihtoehtona on kuitenkin esimerkiksi hajautettu tietokanta, tarkoittaen tietokannanhallintajärjestelmän (DBMS, Database Management System) jakamista useampaan tietokantailmentymään. Näistä ilmentymistä voidaan ulkopuoliseen käyttöön jakaa hallintajärjestelmän kautta tietokannat, jotka sisältävät ainoastaan tarvittavia tietoja. Tämä mahdollistaa, että esimerkiksi käyttäjien tallennettuja tietoja ei tarvitse jakaa ulkopuolisille tahoille. Hajauttaminen kuitenkin lisää kokonaisuuden monimutkaisuutta, eikä aina ole helppo ylläpidettävä, sillä järjestelmän toimivuus täytyy hallita, jos yksi osa pettää [5]. Yhteys tietokantaan voidaan toteuttaa joko suoralla kantayhteydellä tai siihen voidaan hyödyntää keskitettyä web-palvelua, joka suorittaa varsinaiset tietokanta kutsut. Tietokanta voi silloin itsessään olla jaoteltuna, eikä tietoa tarvitsevien osapuolten tarvitse olla siitä tietoisia, sillä ne käyttävät web-

palvelun tarjoamia rajapintoja [97]. Tässä tapauksessa integraatiotapa muuttaa muotoaan ja se muuttuu seuraavaksi esiteltäväksi etäältä kutsuttavan aliohjelman integraatiotavaksi.

Etäältä kutsuttavat aliohjelmat tarjoavat tietopohjaisen integraation sijasta integroitumisen ohjelman toiminnallisuuden mukaan. Käytännössä tämä tarkoittaa sitä, että esimerkiksi tietoa tarvittaessa kutsutaan suoraan kohdeohjelman rajapintaa, joka koostaa tiedot sen hallinnoimista lähteistä. Vaihtoehtoja rajapinnoille on useita ja samaan tietoon voidaan tarjota usealla tavalla toteutettuja rajapintoja. Ohjelmien etäältä kutsuttavaa toiminnallisuutta voidaan toteuttaa esimerkiksi Java RMI (Java Remote Method Invocation) -rajapintatoteutuksella tai tarjoamalla REST (Representational State Transfer) -arkkitehtuurin mukaisia web-palveluja [3] (s. 50). RMI toteutukset ovat sidonnaisia tiettyyn teknologiaan tai käyttöjärjestelmään, kun taas web-palvelut ovat vähemmän ohjelmointiteknologiariippuvaisia, sillä niiden tiedonsiirto perustuu standardi protokolliin [95]. Tämä integraatiotyyli helpottaa tiedon ristiriitaisuuksissa, sillä kutsuttava aliohjelma kerää tiedon yhteen, eikä aliohjelmaa kutsuva osapuoli pääse muokkaamaan tiedon rakennetta. Samalla varmistetaan tiedon eheys, koska esimerkiksi toinen ohjelma ei pääse suoraan muokkaamaan tietoa, vaan se tehdään hallitusti tiedon hallinnoivan ohjelman toimesta [3] (s. 50).

Viestipohjaiseen kommunikaatioon perustuvissa integraatiossa viestin kulkua on pilkottu toiminnallisuuden mukaan pienempiin osiin [3] (s.53). Viestipohjainen integraatio tarjoaa viestintäjärjestelmän osapuolille muita integraatiotapoja enemmän toisistaan riippumatonta kommunikaatiota. Käytännössä tämä tarkoittaa mahdollisuutta asynkroniseen toiminnallisuuteen, eli viestin lähettäjän ei tarvitse olla tietoinen kohdejärjestelmän tilasta tai olemassaolosta [96]. Hophe ja Woolf esittelevät kirjassaan malleja viestipohjaisen integraation sisältämien osien toteuttamiseen. Osia on kaikkiaan 65 ja ne ovat kategorisoitu toiminnallisuuden lisäksi sen mukaan missä kohtaa viestin kulkua se sijaitsee [3] (s.53). Kuvassa 1 on esitelty graafisesti viestin kulku integraation aikana ja osat, joista koko prosessi koostuu. Kaikkia komponentteja ei tapauksesta riippuen käytetä, mutta paremman hahmotettavuuden vuoksi ne on kaikki sisällytetty kuvaan [4].



Kuva 1 *Yleinen kulkukaavio viestipohjaisesta integraatiosta [4]*

Keskeisimmät osat viestipohjaista integraatiota ovat viesti, kanava ja rajapinta [4]. Viesti on yksinkertainen tieto-objekti, jonka mukana tietoa voidaan kuljettaa kanavalla. Viesti koostuu kahdesta osasta, otsikosta ja hyötykuormasta. Otsikko sisältää oheistietoa viestistä, kuten viestin kohteen tai mahdollisesti sen lähettäjän. Sitä käyttää pääsääntöisesti viestintäjärjestelmä, jotta viesti ohjataan oikeaan paikkaan. Hyötykuorma sisältää varsinaisen tiedon, jota ollaan siirtämässä kanavalle [3] s. 31. Kanava on viestin siirtoon tarkoitettu komponentti, joille on useita toteutusmahdollisuuksia [4]. Ne voivat olla suoria yhteyksiä toisen järjestelmän rajapintaan tai se voidaan toteuttaa myös siten, että kanava jakautuu yhdestä sisääntulokanavasta useammaksi ulostulevaksi kanavaksi tilaajien mukaan ilman, että kanavalle lähettävän tarvitsee tietää niiden olemassaolosta. Tällöin puhutaan publish-and-subscribe-kanavasta [96]. Ohjelmien liittämiseksi viestintäjärjestelmään tarvitaan erilaisia integraatiorajapintoja, joiden vastuulla on lähettäessä muuntaa tieto viestiksi kanavalle tai vastaanottaessa hyödyntää kanavalta saamaansa viestiä [4].

Tutkimuksen kannalta olennainen rajapinta on yhdyskäytävä (messaging gateway), jossa viestintään käytettävä logiikka eriytetään ohjelmasta. Eriyttäminen mahdollistaa sen, että ohjelman ei tarvitse tietää koko viestintäjärjestelmästä eikä sen yksityiskohdista. Yhdyskäytävä toimii kaksisuuntaisesti, jolloin se voi vastaanottaa viestejä myös takaisin järjestelmästä. Kanaville tulleita viestejä voidaan viestijärjestelmän sisäisesti ohjata reitittimillä erilaisten sääntöjen mukaan seuraaville kanaville. Esimerkki reitittimestä on viestinvälittäjä (message broker), jossa kanavasta saadut viestit ohjataan oikeaan kanavaan, esimerkiksi viestin otsikon perusteella. Reitityksen yhteydessä viestiä ei yleensä muokata, vaan se viedään sellaisenaan oikealle kanavalle. Viestin muokkaukseen käytetään kääntäjäkomponentteja, joilla voidaan viestin sisältö muuntaa vastaamaan kohderajapinnan tietomallia tai esimerkiksi suodattaa paljon tietoa sisältävää viestiä. Virhetilanteiden ja järjestelmän suorituskyvyn tarkkailua voidaan toteuttaa käyttämällä monitorointiin luotuja malleja, kuten hallintaväylä (Control Bus), johon kaikki komponentit on liitetty ja näin voidaan verkkoa analysoida yhden solmupisteen kautta [4].

2.3 Ratkaisumallit

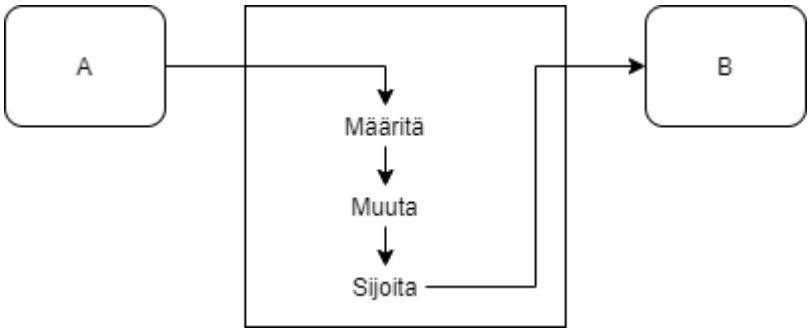
Integraatioita voidaan toteuttaa monella tapaa ja käyttökohteen mukaan niihin toimii tietynlainen ratkaisumalli. Ratkaisumallit ovat korkean tason tapoja toteuttaa integraatioita [10] (s. 69). Tutkimuksen asiayhteydessä ne eroavat integraatiotavoista siten, että ratkaisumalli kertoo enemmän integraatioarkkituurista, kuin siitä millä tavalla integraatio toteutetaan. Tässä luvussa esitellään kolme yleistä integraatiomallia: Point-to-Point, eli

pisteestä pisteeseen, Hub-and-Spoke, eli keskusmalli ja ESB (engl Enterprise Service Bus), eli palveluväylä.

2.3.1 Point-to-point – pisteestä pisteeseen

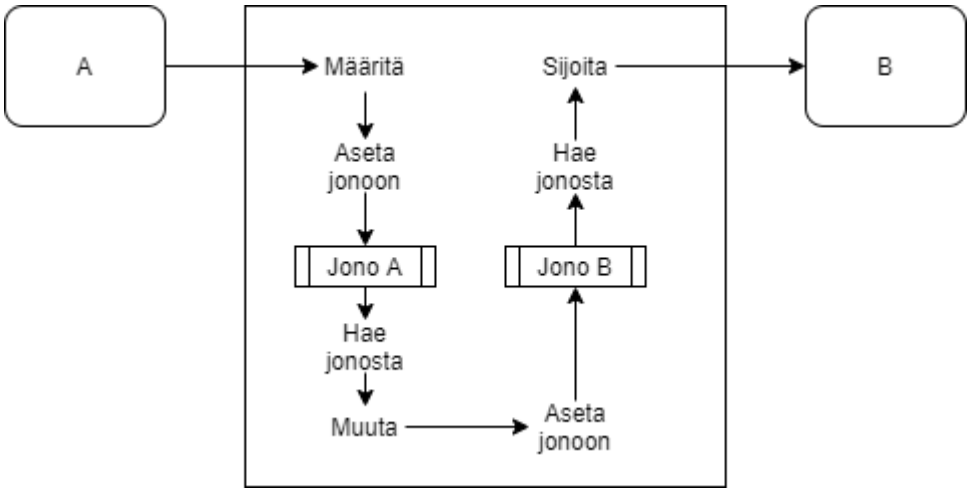
Point-to-point, eli pisteestä pisteeseen -integraatio, nimensä mukaisesti tarkoittaa kahden eri järjestelmän tai ohjelman välille luotua integraatioväylää. Tämä väylä on ainoastaan kahden osapuolen välinen, eikä siihen voida tuoda tietoa kolmannesta lähteestä, ilman että luodaan uusi yhteys. Tilanteen mukaan väylä tehdään molempiin suuntiin toimivaksi, eli se koostuu kahdesta yhteydestä [8].

Yhteyksillä on kahdenlaisia kommunikaatiotapoja, synkronisia ja asynkronisia. Synkroniset yhteydet eivät salli useampaa samanaikaista vuorovaikutusta. Sen toiminta perustuu siihen, että integraatiototeutus selvittää kohdejärjestelmän rajapinnan tietomallit ja muuntaa viestin sisältämän tiedon vastaamaan sitä [10]. Kuvassa 2 esitellään yleisellä tasolla synkronisen pisteestä pisteeseen -integraatioyhteyden kulkuprosessi.



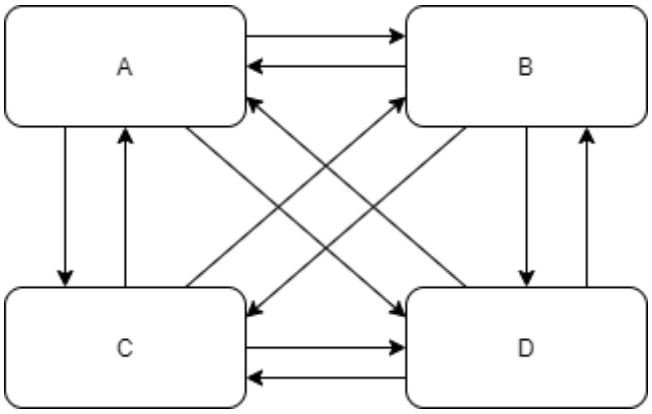
Kuva 2 *Synkronisen pisteestä pisteeseen -integraation kulku [10]*

Asynkroninen kommunikaatio mahdollistaa kahden järjestelmän suoran riippumattomuuden toisistaan. Viesti tallennetaan toteutuksen mukaan välikerrokselle, joka voi olla tietokanta tai jono. Jonoa tai tietokantaa aletaan purkaa yksi kerrallaan, ja välitetään sitä kautta kohteeseen. Tämä mahdollistaa muun muassa sen, että osapuolten ei tarvitse tietää missä tilassa toinen on, toisin sanoen onko toinen vastaanottavassa tilassa vai ei. Lisäksi viestin lähettävän osapuolen ei tarvitse jäädä odottamaan suorituksen loppumista [10]. Kuvassa 3 on esimerkki kahdella jonolla toteutetusta viestipohjaisessa asynkronisessa pisteestä pisteeseen -yhteydessä.



Kuva 3 *Asynkroninen pisteestä pisteeseen -integraatio jonoilla toteutettuna [10]*

Pisteestä pisteeseen -integraatioiden perustana on se, että tiedetään sekä kohde että lähdejärjestelmien ulospäin tarjoamat rajapinnat ja tietomallit. Näin ollen voidaan yksittäisissä tapauksissa nopeasti rakentaa ohjelmallisesti putki tiedon molempiin suuntiin siirtämiselle [8]. Yksinkertainen pisteestä pisteeseen -integraatioyhteyksien verkko on esitelty kuvassa 4.

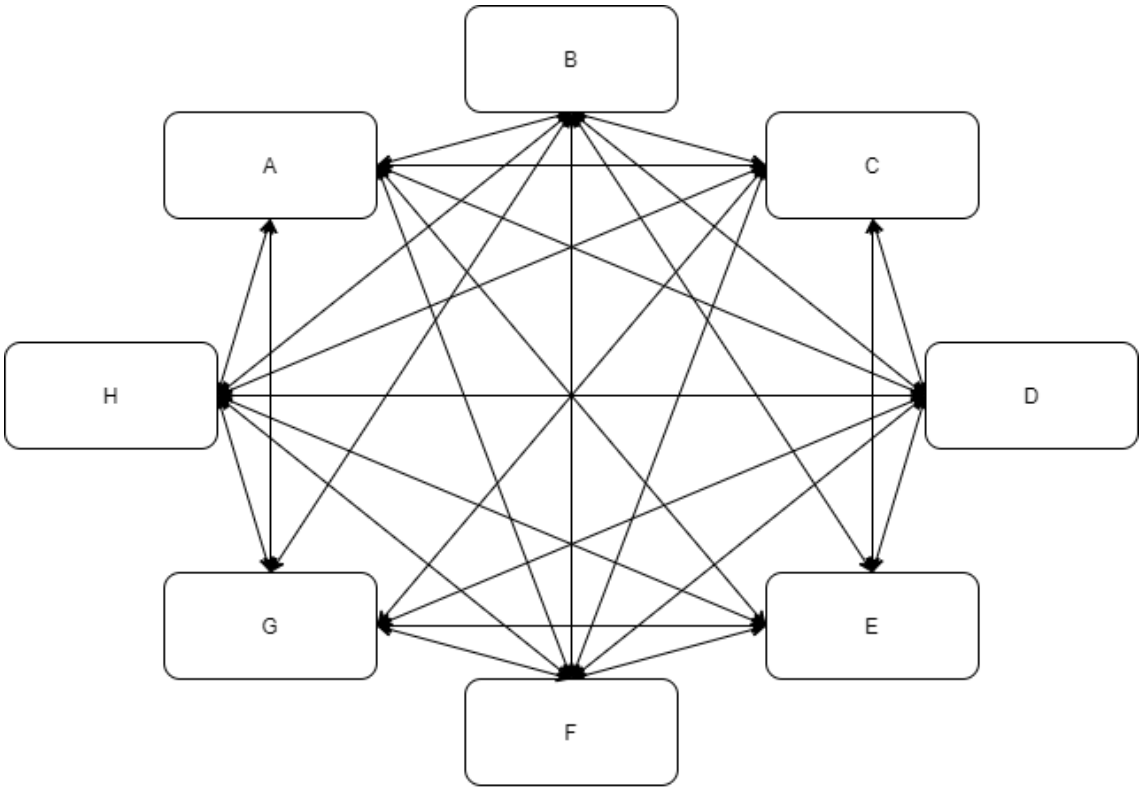


Kuva 4 *Yksinkertainen pisteestä pisteeseen -verkko [8]*

Pisteestä pisteeseen -integraatioiden hyviä puolia on niiden yksinkertaisuus, johon vaikuttavat järjestelmien rajapintojen dokumentaatio ja tekninen toteutus. Verkon koostuessa vain muutamasta järjestelmästä malli täyttää vaatimukset hyvin ja pysyy hallittavissa. Yhteyksien ollessa yksinkertaisia ja hyvin toteutettuja, raskaille integraatioarkkitehtuureille ei ole tarvetta. Yhden järjestelmän toimimattomuus ei myöskään kaada koko verkkoa, vain ne yhteydet, jotka ovat siihen liitettyjä. [8].

Harvemmin integraatiot jäävät vain muutaman järjestelmän välille. Kuten kuvasta 4 havaitaan, neljän järjestelmän keskinäisen integroinnin seurauksena yhteyksiä tulee yhteensä 12. Molempiin suuntiin kulkevien yhteyksien määrä kasvaa kaavalla $\mathrm{n}(\mathrm{n}-1)$ [8], jossa n on järjestelmien määrä. Mikäli aiemmin esiteltyyn verkkoon tuotaisiin vielä viides

järjestelmä ja sillä pitäisi olla yhteys kaikkiin muihin järjestelmiin, kasvaa yhteyksien määrä kahdeksalla 20:een yhteyteen. Nopeasti huomataan, että järjestelmien määrän kasvaessa ylläpitotarve kasvaa merkittävästi. Kuvassa 5 on esimerkki monimutkaisemmasta pisteestä pisteeseen -integraatiosta, jossa piirretyt väylät ovat kaksisuuntaisia.



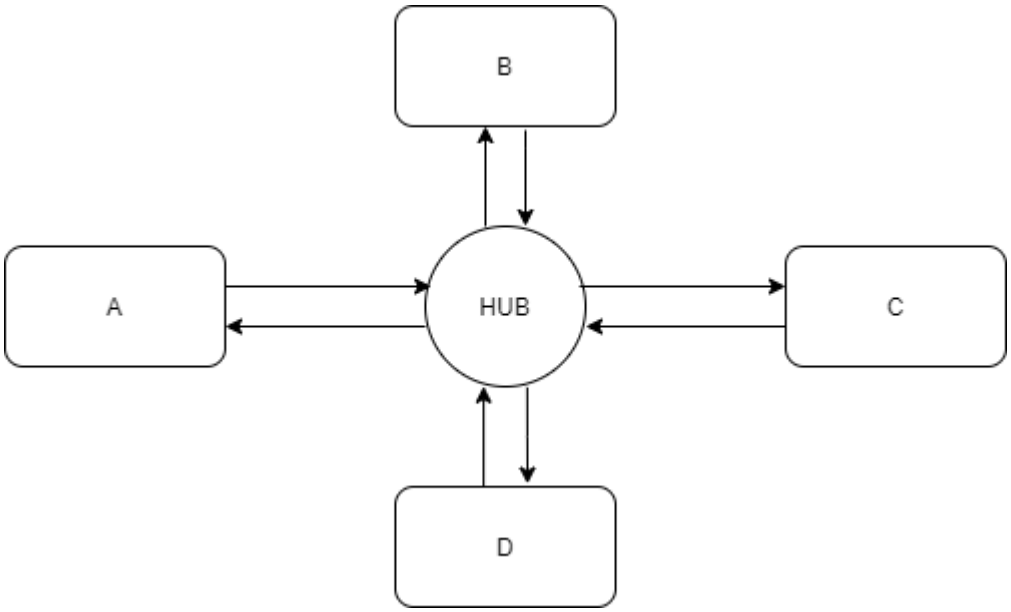
Kuva 5 *Monen järjestelmän pisteestä pisteeseen -integraatioverkko [8]*

Usean järjestelmän pisteestä pisteeseen -verkosta tulee helposti hyvin sekava ja jokainen yhteys vaatii osapuolten tarjoamien rajapintojen tuntemisen sekä ohjelmallista työtä pitääkseen ne ajan tasalla. Riski yksittäisen yhteyden vanhentumiselle suuremmissa integraatioverkoissa on suuri. Ylläpidettävyys onkin pisteestä pisteeseen -integraation suurin haaste. Suuret muutokset yhteen järjestelmään aiheuttavat korjaustöitä jokaiselle siihen tehdylle yhteydelle ja tämä tulee suurissa integraatioverkoissa kalliiksi. Mikäli on mahdollisuus sille, että integraatioverkkoa laajennetaan, pisteestä pisteeseen - integraatioita tulisi välttää, jos mahdollista [8].

2.3.2 Hub-and-Spoke – keskitetty malli

Hub-and-spoke, myös multipoint, eli keskitetty integraatiomalli tarkoittaa, että yhteydet luodaan toisen järjestelmän sijasta keskitetyn integraatiojärjestelmän kautta [10]. Viestipohjaisessa integraatiotavassa vastaavasta mallista käytetään nimitystä viestinvälittäjä (engl. message broker), joka toimii siinä viestien reitittimenä [4]. Keskeinen järjestelmä

voi olla joissain tapauksissa web-palvelin, joka sisältää integraatiologiikkaa. Tällöin mahdollistetaan SOAP- tai HTTP-standardien mukaista kommunikaatiota [95]. Toteutuksen mukaan keskeinen järjestelmä voi sisältää mukautettavaa logiikkaa ja sen päätehtävä on ohjata viestit oikeaan osoitteeseen käyttäen erilaisia reititysratkaisuja [10] (s.10-12). Kuvassa 6 on yksinkertainen korkean tason keskitetyn mallin -integraatioverkko.

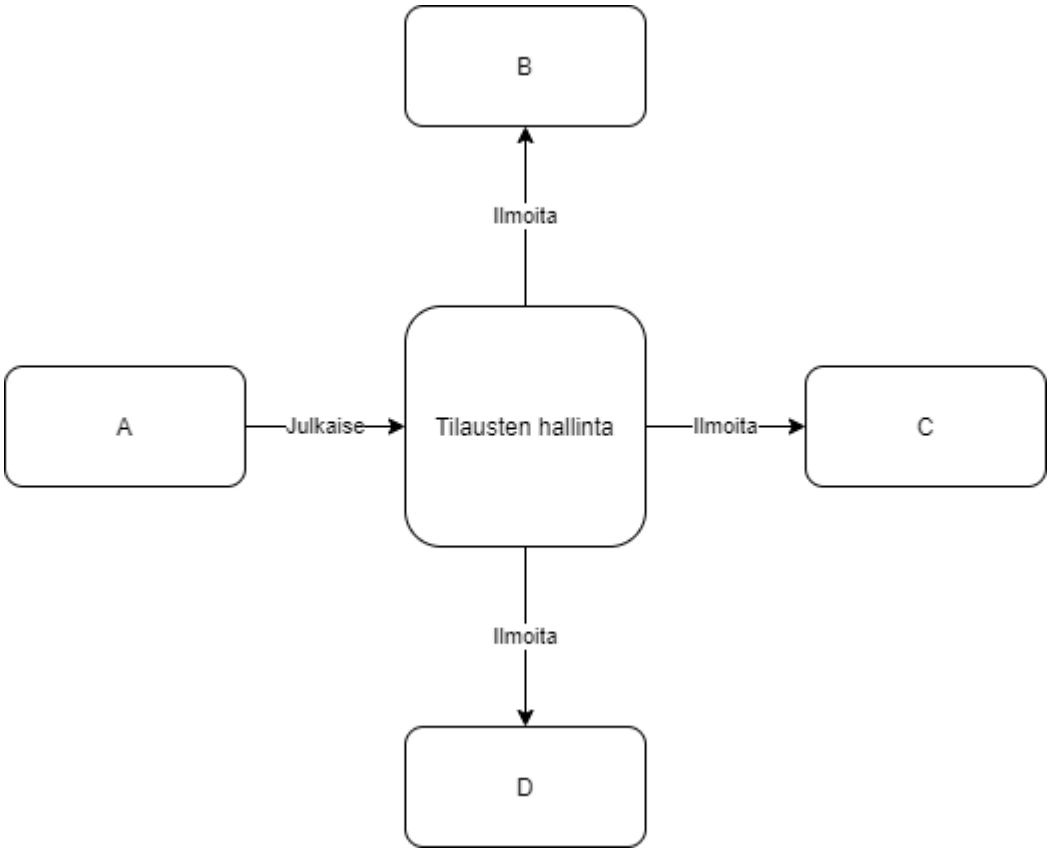


Kuva 6 *Yksinkertaistettu keskitetyn mallin integraatioverkko [10]*

Perinteisessä keskitetyssä mallissa lähdejärjestelmä määrittelee kohteensa viestin otsikosta, jonka avulla viesti ohjataan oikeaan osoitteeseen. Tämä vaatii sen, että lähdejärjestelmä tietää kohteen olemassaolosta [10] (s. 10). Yksinkertaisimmillaan seuraavasti: Järjestelmä A lähettää viestin, jonka otsikossa ilmoitetaan kohteeksi järjestelmä B. Keskeinen järjestelmä tekee tarvittavat muokkaukset, jotta tiedot ovat kohteen vaatimassa muodossa ja välittää viestin kohteeseen. Viestin kulku on vastaavanlainen kuin asynkronisissa pisteestä pisteeseen -yhteyksissä (Kuva 3), mutta tietovarasto tai jono on kaikille järjestelmille sama ja sijaitsee keskeisesti viestinvälittäjällä.

Publish-and-Subscribe on viestintätapa, jossa lähettäjä ei määrittele erikseen sen kohdetta, vaan määriteltyjen sääntöjen mukaan viestit osataan ohjata oikeille vastaanottajille. Vastaanottajat voivat tilata (engl. subscribe) jonkin määritetyn säännön mukaisen aiheen, esimerkiksi tilanne, jossa järjestelmässä asiakkaan osoite muuttuu. Tällöin asiakkaan osoitteen muuttuessa luodaan viesti kaikille tämän tapahtuman tilanneille järjestelmille. Viestintätapa voidaan toteuttaa keskeisellä tilaustenhallintajärjestelmällä, jossa lähettäjä (engl. publisher) tekee toimenpiteen, jonka perusteella keskeinen järjestelmä

osaa luoda tapahtumasta viestit sen tilanneille osapuolille [10] (s.10–12). Viestipohjaisessa integraatiotavassa on myös mahdollista toteuttaa vastaavaa toimintaa ilman keskeistä järjestelmää käyttämällä publish-subscribe-kanavaa [96]. Arkinen esimerkki Publish-and-Subscribe-periaatteesta on videopalvelu YouTube, jossa käyttäjät voivat tilata ilmoituksia sisällöntuottajan uusista videoista. Järjestelmä huolehtii, että uudesta videosta osataan ilmoittaa sen tilanneille ilman, että julkaisijan tarvitsee erikseen niitä määritellä. Edellä mainittu esimerkki on myös havainnollistettu kuvassa 7.



Kuva 7 *Yksinkertainen Publish-and-Subscribe viestintämalli hubilla toteutettuna [9]*

Suurinta etua pisteestä pisteeseen -arkkitehtuuriin nähden keskitetty malli tuo ylläpidettävyyteen. Uusia verkkoon tuotujen järjestelmien ja olemassa olevien välisiä yhteyksiä ei tarvitse tehdä erikseen, eikä aiempiin tarvitse tehdä muutoksia. Ainoastaan järjestelmän ja hubin välille luodaan yhteys ja niiden määrä kasvaa lineaarisesti [10] (s. 227). Publish-and-Subscribe reititysmallilla mahdollistetaan viestien lähettäminen useammalle vastaanottajalle kerralla. Keskitetyssä mallissa monitoroinnista tulee yksinkertaisempaa, koska kaikki liikenne kulkee yhden paikan kautta, on verkkoa helpompi tarkkailla [9].

Kuten pisteestä pisteeseen -integraatioissa, viestin otsikkoon perustuvalla reititysmallilla lähetetyt viestit ovat yksisuuntaisia. Tämä tarkoittaa sitä, että jos toisesta järjestelmästä tulleeseen kutsuun vastataan, käsitellään tämä omana viestinään. Tässä mallissa järjes-

telmällä ei siis ole logiikkaa, joka kykenisi yhdistämään paluuviestiä alkuperäiseen viestiin. Samasta syystä alkuperäisen viestin perusteella ei voida tehdä lisäreitityksiä kolmansiin järjestelmiin, sillä edellistä viestiä ei ole enää vastaanoton jälkeen saatavilla, jonka perusteella ohjauksia voitaisiin tehdä [10] (s.10–12).

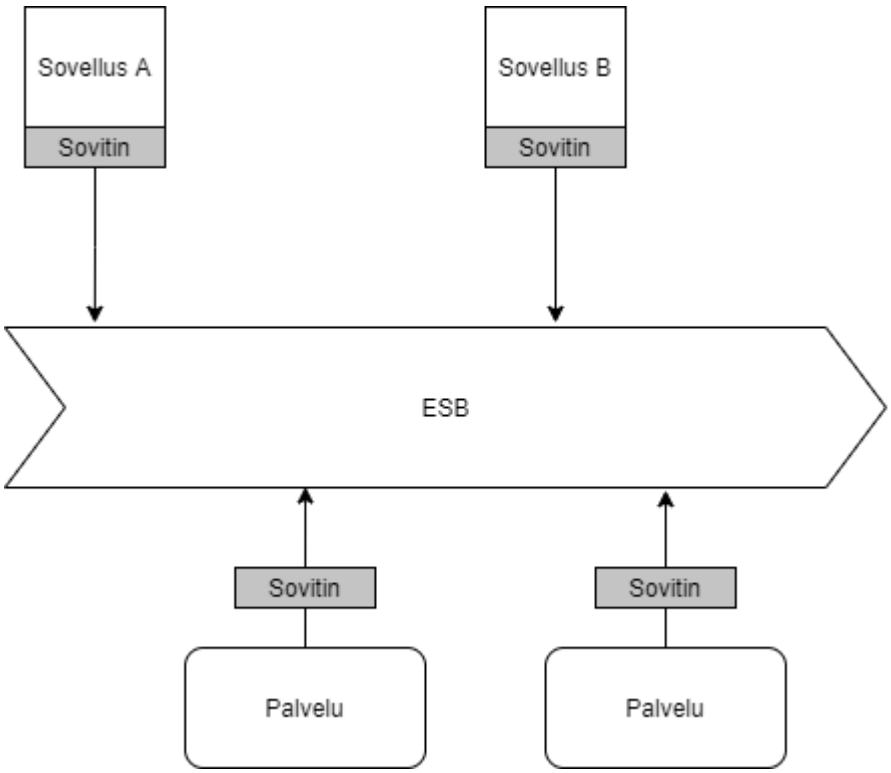
Edellä mainittuja ongelmia voidaan kuitenkin vähentää laajentamalla hubin sisälle työnkulun hallintajärjestelmä (engl. workflow management system) [10] (s.12–13). Se mahdollistaa liiketoiminnallisen logiikan lisäämisen järjestelmään, jonka avulla voidaan reitittää viestejä useamman järjestelmän läpi alkuperäisen viestin perusteella. Viestejä ei tällöin reititetä enää pelkästään viestin otsikon mukaan, vaan logiikan ja tietovaraston ansiosta ketjuutuneet viestit voidaan yhdistää toisiinsa [10] (s.225–227).

Laajennettukaan versio ei poista kaikkia keskitetyn integraatiomallin ongelmia. Kaiken toiminnan pohjautuessa yhteen solmukohtaan, aiheuttaisi tämän pettäminen koko integraatioverkon lamaantumisen. Ongelmaksi tämä muodostuu hyvin laajoissa verkoissa, joissa hubilla on valtavasti liikennettä. Riskinä on myös hallintajärjestelmässä kasvavan logiikan määrä, jolloin prosessoinnista voi tulla raskasta [10] (s. 14). Yhden solmukohdan ongelmaa voidaan lievittää hajauttamalla järjestelmää useampaan osaan, mutta tämä lisää kokonaisuuden monimutkaisuutta [95].

2.3.3 Enterprise Service Bus – palveluväylä

ESB (Enterprise Service Bus), eli palveluväylällä tarkoitetaan integraatioihin käytettävää mallia, jolle ei ole yhtä selkeää määritelmää. Sitä voidaan kuvailla kokonaisuudeksi, joka sisältää erilaisia standardeihin perustuvia kommunikointimenetelmiä keskeisen järjestelmän avulla [14] (ch 1.0). Standardipohjaiset menetelmät mahdollistavat eri protokollia ja teknologioita käyttävien ohjelmien liittämisen väylän kautta palveluihin. Liittyneet osapuolet eivät ole suoraan yhteydessä toisiinsa, vaan kommunikaatio tapahtuu palveluväylän avulla. Väylän ulospäin avoin rajapinta hyväksyy kutsut ja sisäisesti se suorittaa tarvittavat reititykset sekä muunnokset palveluiden ja toisten järjestelmien kutsumiseksi [14]. Käytännössä ohjelmat voivat liittyä väylälle käyttäen sovittimia, joilla määritetään esimerkiksi kohdepalvelu ja käytettävä protokolla millä väylälle yhdistetään [12]. Tiedot väylän sisällä liikkuvat yhteneväisessä muodossa, usein käytetään XML-formaattia perustuen luotuun tietomalliin [14] (ch 1.7). Mikäli saapuva tieto ei ole XML-formaatissa, sisällytetään väylän sisälle muuntaja, joka muuntaa saapuvan tiedon yhtenäisen mallin mukaisesti XML-tiedoksi [14] (ch 4.4.1). Yhtenäinen tietomuoto helpottaa tiedolle tehtä-

viä operaatioita ja tiedon yhdistämistä väylän sisällä [14] (ch 1.7). Koska liittymärajapinnat ovat samoja kaikille liittyville osapuolille, voidaan niitä lisätä ja poistaa helposti vaikuttamatta muihin järjestelmiin [14]. Yleiskuva palveluväylän toiminnasta on kuvassa 8.



Kuva 8 *Palveluväylän yleiskuva [14] (ch. 1.7.2, muokattu)*

Etuina palveluväylässä ovat yhtenäinen liittymisväylä kaikille osapuolille eri tekniikoista huolimatta, jolloin niitä on helppo lisätä ja poistaa [13]. Sisäisen rakenteen ansiosta se skaalautuu keskitettyä mallia paremmin suurelle määrälle yhteyksiä [14]. Erilaiset reititysmahdollisuudet tuovat myös monipuolisuutta, sillä väylä voi silloin tarjota useampia viestinvälitys menetelmiä samanaikaisesti. Lisäksi väylä voi tuoda ylimääräisen kerroksen tietoturvallisuutta, sillä väylän sisällä voidaan myös toteuttaa käyttäjien oikeuksien hallintaa keskitetysti hyödyntämällä yrityksen turvallisuuspolitiikkaa [13].

Heikkoutena palveluväylällä on keskitetyn mallin tapaan yhden solmukohdan ongelma, eli koska järjestelmä on käytännössä yhden järjestelmän varassa, sen pettäminen lamauttaa koko integraatiojärjestelmän [13]. Tätä ongelmaa voidaan lievittää kuitenkin hajauttamalla yksittäinen kokonaisuus useampaan toisiinsa liittyneeseen väylään [14]. Laaja järjestelmä tuo luonnollisesti jonkin verran hitautta verrattuna pisteestä pisteeseen mallin yhteyksiin, eikä monimutkainen järjestelmä ole kannattava, jos integroitavia osapuolia on vähän [13].

3. INTEGRAATIOALUSTAT JA -KEHYKSET

3.1 Yleistä

Aiemmin siiloutuneita ohjelmia integroitiin lukuisilla ohjelmien välisillä pisteestä pisteeseen -integraatioilla. Myöhemmin pisteestä pisteeseen -integraatioiden heikkouksien minimoimiseksi alettiin hyödyntämään muun muassa väliohjelmistoja, kuten viestijonoja, viestinvälittäjiä ja etäältä kutsuttavia proseduureja. Lopulta erillisinä toteutuksina näistä muodostuu usein monimutkaisia kokonaisuuksia, ja ne vaativat suuria muutoksia lähde- ja kohdejärjestelmiin. Niiden aiheuttamat kehitys- ja ylläpitokustannukset lisäsivät tarvetta keskitetyille integraatiojärjestelmille, jolloin alettiin puhua EAI (Enterprise Application Integration), eli yrityssovellusten integraatioista. EAI-menetelmällä pyritään luomaan tiedonvälittäjiä, jotka ovat keskitettyjä ohjelmariippumattomia integraatiokokonaisuuksia. Nämä kokonaisuudet sisältävät standardoituja liitinkomponentteja ja käyttötarkoituksen mukaan rajattua integraatiologiikkaa [16]. Pilvipalveluiden lisääntyessä ja SaaS-jakelumalliin nojautuvien yritysten määrän kasvaessa integraatioihin on tullut uusia haasteita. Esimerkiksi SaaS-ohjelmia hyödyntävällä yrityksellä ei välttämättä ole halua, resursseja tai tietotaitoa luoda monimutkaisia integraatiojärjestelmiä. Integraatioalustoilla pyritään ratkaisemaan näitä uusia haasteita tarjoamalla toimintoja pilvipohjaisesti siten, että integraatioita tarvitsevan yrityksen ei tarvitse hankkia siihen tarvittavaa infrastruktuuria [15].

Integraatioalusta, eli iPaaS (integration Platform as a Service) on Gartnerin määritelmän mukaan kokoelma pilvipalveluita, mahdollistaen integraatiokulkujen kehittämisen, suorituksen ja hallinnan. Integraatiokuluilla yhdistetään yhden tai usean organisaation välillä mitä tahansa yhdistelmää prosesseista, palveluista, sovelluksista tai tiedosta, olivat ne yrityksen sisäisiä tai pilvipohjaisia [17].

Integraatioalustojen käytössä tarvitaan hyvin harvoin, jos ollenkaan, ohjelmointia. Pääasiassa käyttäminen koostuu valmiista elementeistä, joita asetellaan ja määritellään graafisen käyttöliittymän kautta [16]. Valmiita elementtejä ovat esimerkiksi liitännäiset suosittuihin kolmannen osapuolen palveluihin, kuten Amazonin pilvipalvelut tai toiminnanohjausjärjestelmiin, kuten SAP [21]. Jotkin alustat tarjoavat mahdollisuutta toteuttaa myös omia liitinelementtejä. Esimerkiksi Boomin AtomSphere ja MuleSoftin Anypoint Platform tarjoavat mahdollisuuden luoda itse täysin mukautettavia liitinelementtejä käyt-

täen Javalle luotuja kehitystyökaluja (SDK, Software Development Kit) [18],[19]. Itse luotujen elementtien lisäksi, esimerkiksi Boomi-alustalla, voidaan joitakin elementtejä mukauttaa omilla skripteillä [20].

Integraatioalustoista käytetään joskus myös nimitystä EiPaaS (Enterprise integration Platform as a Service), joka tarjoaa laajempia toimintoja yritysintegraatioihin. Usein iPaaS ja EiPaaS erotetaan toisistaan siten, että jälkimmäisen ei tarvitse olla täysin pilvipohjainen, vaan voi sisältää paikallisia komponentteja erilaisissa tapauksissa [16].

Integraatiokehyksellä (engl. integration framework) tarkoitetaan yksinkertaisesti ohjelmistokehystä (software framework), joka tarjoaa uusia ominaisuuksia ohjelmistoteknologiaan, kuten ohjelmointikieleen tai toiseen kehykseen. Kehys tarjoaa rungon, jonka avulla voidaan helpottaa itse tehtävää työtä. Kehys ei tarjoa valmista toteutusta, vaan esimerkiksi oliopohjaisissa teknologioissa abstrakteja luokkia, joita voidaan periyttää osaksi omia luokkia. Kehykset tarjoavat myös valmiita luokkia, joita voidaan hyödyntää suoraan toteutuksissa [22] (s.54).

3.2 Tutkittavat alustat

Integraatiolustoista ja niiden toiminnoista oli toimeksi antavalla yrityksellä ja kirjoittajalla vain vähän ennalta tietoa, joten pääasiassa niiden valinnat perustuvat tarjolla oleviin yleisimpiin ja suosittuihin vaihtoehtoihin. Alustoja on tarjolla paljon eri tarjoajilta, ja etukäteen tutkittuna useat niistä ovat käytännössä hyvin samanlaisia käyttää. Ne sisältävät paljon vastaavaa toiminnallisuutta ja niitä yhdistää graafinen käyttöliittymä, jonka avulla integraatiokulku rakennetaan käyttäen erilaisia valmiita osia. Tällaisia osia ovat esimerkiksi integraatioiden käynnistykseen käytettävä HTTP-kuuntelija, joka nimensä mukaisesti odottaa saapuvaa kutsua ja sen saadessaan aloittaa integraatiovirran. Eroja alustojen väliltä löytyy lähinnä termeissä, käyttöliittymässä, helppokäyttöisyydessä ja kuinka varsinainen integraatiototeutus saadaan julkaistua.

Ennen varsinaisen siirrettävän integraatiotapauksen valintaa tutkittiin alustoja yleisesti dokumentaatioiden perusteella. Yhdistäminen sisäverkkoon oli yrityksen puolesta yksi vaikuttava tekijä alustan valinnalle, koska kaikissa integraatioissa tarvitaan resursseja, jotka sijaitsevat yrityksen omassa verkossa. Yrityksellä on myös muita vaatimuksia alustojen toteutuksien suhteen, jotka esitellään seuraavassa luvussa. Näillä vaatimuksilla ei ollut merkittävää vaikutusta vertailuun valittuihin alustoihin, sillä alustojen ollessa hyvin samankaltaisia todettiin näiden toteutumisen selviävän parhaiten käytännössä. Ennalta

arvaamattomasti yhdeksi määrittäväksi tekijäksi valintojen kannalta ilmaantui kokeilutunnusten saaminen, sillä osaan ei saatu tunnuksia lainkaan tai niissä kesti pitkään. Tämä aiheutti sen, että vertailuun valittiin ne alustat, joihin tunnukset tulivat ensimmäisenä.

Huomioitavia poisjätettyjä alustoja olivat esimerkiksi Jitterbitin iPaaS-alusta ja kotimainen HiQ Oy:n Frends-alusta. Jitterbitin sivustolla mainittiin alustan Cloud Studion esittelyssä olemassa olevan koodin uudelleenkäyttöä [23]. Sinne ei saatu kuitenkaan lainkaan kokeilutunnuksia, joten tätä ei päästy kokeilemaan ja alusta täytyi jättää pois tämän tutkimuksen vertailusta. Frends-alusta jäi pois aikarajoitteisuuden vuoksi. Pois jättämiseen vaikutti lisäksi, että sen käyttäminen vaikutti dokumentaatioiden mukaan hyvin samankaltaiselta kuin muilla valituilla alustoilla, jolloin sen valitseminen ei olisi tuonut merkittävää arvoa tutkimuksen kannalta.

Lopulta päädyttiin kolmeen toisistaan tarpeeksi eroavaan alustaan, joita vertailemalla saadaan kartoitettua tarpeita pitämällä samalla alustojen määrä kohtuullisena. Valittujen alustojen tarjoajat löytyvät Gartnerin Magic Quadrant Leader-sarakkeesta, johon vuosittain valitaan edistyneimmät integraatioalustat [1]. Seuraavaksi esitellään valittuja alustoja yleisesti, jonka jälkeen esitellään myös käytetyt integraatiokehykset.

3.2.1 Boomi AtomSphere

Boomi AtomSphere on Boomi-liiketoimintayksikön luoma integraatioalusta. Yksikön toiminnot ovat olleet Dell Technologies yrityksen omistuksessa vuodesta 2010, mutta ne myytiin toukokuussa 2021 Francisco Partners sijoitusyritykselle [24],[25]. Alusta tarjoaa integraatiotyökalujen lisäksi rajapintojen hallintaa, keskitettyä tiedonhallintaa ja pilvipohjaisten sovellusten luontia [26].

Pilvipohjaisia sovelluksia alustalla voidaan luoda Boomi Flow:n avulla. Boomi Flow sisältää selainpohjaisen web-käyttöliittymän luomaan kokonaisvaltaisia sovelluksia web, mobiili sekä IoT-alustoille [27]. Alusta tarjoaa työkalut tietojenhakuun, API-hallintaan, käyttöliittymän toteuttamiseen ja julkaisuun [28]. Sovelluksista voidaan hyödyntää alustan integraatiopalveluita käyttämällä Flow-palveluita, jotka voivat aktivoida luotuja integraatiototeutuksia [29].

Boomin integraatiopalveluiden keskiössä on integraatioprosessi. Prosessi on graafinen esitysmuoto, joka kuvaa sisään tulevan dokumentin kulkua. Tähän kulkuun kuuluu olennaisesti sille tehtävät operaatiot ja lopulta dokumentti lähetetään joko eteenpäin tai palautetaan prosessin kutsujalle [30]. Prosessit julkaistaan paikallisesti tai alustan pilvipalvelun kautta ylläpidettävillä ajoympäristöillä [31].

Graafisessa käyttöliittymässä integraatiokulkuja toteutetaan raahaamalla elementtejä paikoilleen ja tekemällä niihin määrityksiä. Elementeistä käytetään alustalla nimitystä kuvio (engl. shape) ja näitä kuvioita on monenlaisia eri tarpeisiin ja osaa niistä voidaan muokata itse tehdyillä komentosarjoilla, eli skripteillä. Kuvioita on monenlaisia ja ne sisältävät komponentteja integraation käynnistykseen, tiedon manipulointiin, prosessin ohjaukseen ja ulkoisten palveluiden yhdistämiseen [33].

Alusta tarjoaa julkaisustrategioinaan hybridimallia tai täysin pilvipohjaista julkaisumallia. Hybridimallissa palvelimelle asennetaan integraatiot suorittava ajoympäristö. Omalle palvelimelle asennettaessa saadaan ajoympäristö sisäverkkoon, jolloin voidaan yhdistää yrityksen verkossa sijaitseviin rajapintoihin ja tietokantoihin. Paikallisesti toimivia ajoympäristöjä on muutamia erilaisia; Atom, Molecule ja private Atom Cloud [34]. Atom on kevyenä dynaamisena suoritusmoottorina toimiva Java-sovellus, jolla julkaistuja integraatioprosesseja voidaan suorittaa. Koska Atom on yksittäinen suoritusyksikkö ja se voidaan asentaa vain yhdelle koneelle, siinä ei ole sisäänrakennettua vikasietoisuutta tai kuorman hallintaa [35]. Molecule on joukko Atom-yksiköitä, joita voidaan asentaa useammalle koneelle ja näin mahdollistaa parempaa virhesietoisuutta ja kuorman hallintaa eri koneiden välillä [36]. Atom Cloud sen sijaan on kokoelma Molecule-ilmentymiä, joka hajautuneen rakenteen ansiosta mahdollistaa usean käyttäjäryhmän pääsyn samoihin resursseihin. Atom ja Molecule ovat yksittäisinä ilmentyminä käytössä ainoastaan yhdelle käyttäjäryhmälle. Atom Cloudia on mahdollista käyttää pilvipohjaisesti Boomin ylläpitämänä tai sitä voidaan suorittaa omilla palvelimilla, jolloin puhutaan private Atom Cloudista. Atom Cloudit tukevat oletuksena hajautettua suoritusmenetelmää, jolloin ajoympäristö voi suorittaa useampaa integraatioprosessia samanaikaisesti [37]. Molecule-ajoympäristössä tämä ei ole oletuksena päällä, mutta se voidaan erikseen määritellä aktiiviseksi [36].

3.2.2 Anypoint Platform ja Mule

Anypoint Platform on MuleSoft yrityksen tarjoama integraatioalusta. Anypoint Platform alustan pohjana on Mule ESB-palveluväylä, joka toimii suoritusmoottorina integraatioalustalle ja se on saatavilla myös erikseen. Mulesta löytyy kaksi versiota; ilmainen avoimen lähdekoodin yhteisöversio ja maksullinen yritysversio. Alusta tuo Mule-palveluväylään pilvipohjaisia lisäominaisuuksia kuten: APIen hallinta, ajoympäristöjen hallinta, lokien hallinta ja selainpohjaisen graafisen käyttöliittymän integraatioiden luontiin. Mule-ohjelmiksi kutsuttavia integraatiototeutuksia voidaan myös luoda käyttäen Anypoint Stu-

dio-ohjelmointiympäristöä. Luodut ohjelmat ovat sellaisenaan julkaistavissa myös integraatioalustalla hyödyntäen erilaisia julkaisustrategioita tai viemällä ne alustalle muokattavaksi [7].

Julkaisustrategioita Anypoint alustalla on neljänlaisia; CloudHub, Hybrid, Private Cloud tai Runtime Fabric. CloudHub julkaisuympäristö nimensä mukaisesti tarkoittaa sitä, että Anypoint-alusta toimii täysin pilvipohjaisesti ja mitään ei tarvitse asentaa omiin ympäristöihin. Alusta tarjoaa graafisen käyttöliittymän integraatiokulkujen luontiin, josta ne voidaan julkaista ja testata suoraan pilviympäristössä. Hybridijulkaisussa suorittavaa osaa, eli Mule-ajoympäristöä ylläpidetään omalla palvelimella. Ajoympäristö voidaan lisätä pilvipalvelun hallintapaneeliin, jolloin sitä voidaan monitoroida ja julkaista Mule-ohjelmia pilvipalvelun käyttöliittymän kautta [38].

Private Cloud ja Runtime Fabric ovat julkaisuratkaisuja, joissa alustaa tai osia siitä hallinnoidaan omassa infrastruktuurissa. Private Cloudin osalta tämä tarkoittaa itse alustan ylläpitämistä, mutta Runtime Fabricilla ylläpidetään hybridiratkaisun tapaan integraatioita suorittavia osia. Erona kuitenkin laajemmin skaalautuva suoritusjärjestelmä [38].

3.2.3 Azure Logic Apps

Azure Logic Apps on Microsoftin pilvipohjainen integraatiopalvelu, jolla voidaan luoda integraatiototeutuksia käyttäen graafista käyttöliittymää. Tuote on osa Azuren integraatiopalveluita, joihin kuuluu sen lisäksi Service Bus, API Management, Event Grid, Functions ja Data Factory. Jokaisella on hieman oma käyttötapauksensa ja ne on tapauksen mukaan lyhyesti esitelty taulukossa 1 [40].

Taulukko 1 *Azuren integraatiopalvelut käyttötapauksineen [40]*

Käyttötapaus	Integraatiopalvelu
Työnkulun luonti ja yritysprosessien hallinta palveluiden kokonaisvaltaiseen integroimiseen	Logic Apps
Sovellusten ja palveluiden yhdistäminen viestipohjaisen työnkulun luontiin	Service Bus
Rajapintojen julkaisu ja ylläpito tarjoamaan yhteys yrityksen taustajärjestelmiin	API Management

Tapahtumapohjaisten prosessien hallinta ja reitittäminen palveluiden yhdistämiseen	Event Grid
Tapahtumapohjaisten tilattomien toiminnallisuuksien luonti	Functions
Tietopohjaisten integraatioiden luonti hyödyntäen graafista käyttöliittymää	Data Factory

Palveluista valittiin Logic Apps siitä syystä, että se vastaa toiminnaltaan muiden tarjoajien integraatioalustoja, jolloin se on paremmin vertailtavissa muihin. Koska kaikki tuotteet kuuluvat samaan tuoteperheeseen, niitä voidaan hyödyntää myös keskenään.

Alustan termistössä luotavasta toteutuksesta käytetään nimeä työnkulku (engl. workflow), joka tarkoittaa peräkkäisiä askelia, joissa määritellään jokin tehtävä tai prosessi. Työnkulku alkaa erilaisien liipaisimien kautta, joka herättää askeleiden suorittamisen aloittavan toiminnon [41]. Itse luotua koodia voidaan Logic Apps -toteutuksella suorittaa käyttäen Azure Functions -pilvifunktioita tai sisäänrakennetulla koodielementillä [42],[43].

Alustalla integraatiototeutuksissa käytettäviä resursseja tehdään omissa luontinäkymissään resurssienhallinnan kautta. Kun kaikki tarvittavat resurssit on luotu ja määritelty, voidaan integraatiokulut suunnitella yhdessä näkymässä. Toteutuksien julkaisu ja suoritus tapahtuu pilvipohjaisesti, eikä tähän ole suoritusmielessä muita vaihtoehtoja. Kun toteutus on kerran julkaistu, kaikki muutokset julkaistaan automaattisesti, mikäli tallennuksessa ei tullut virheitä.

Alustan hybridimallissa sisäverkkoon ei sijoiteta suorittavia komponentteja vaan pääsy toteutetaan asentamalla yhdyskäytävä, joka ainoastaan ohjaa liikenteen sen kautta mahdollistaen sisäverkon resurssien kutsumisen pilvipalvelusta. Esimerkiksi sisäverkossa sijaitsevan HTTP-rajapinnan yhteyttä varten täytyy luoda mukautettava liitinelementti, jolle määritellään rajapinnan yhteysasetuksia ja yhteyden kulkevan yhdyskäytävän kautta [44].

3.3 Verrattavat integraatiokehykset

Yrityksen vahvan Java-osaamisen takia vaihtoehdot integraatiokehyksille olivat JVM-pohjaisia (Java Virtual Machine) toteutuksia. Lopulta myöskään muita varteenotettavia, toiseen teknologiaan pohjautuvia kehyksiä ei löytynyt. Kokeiltavaksi valittiin kaksi suosit-

tua avoimen lähdekoodin kehystä, Spring Integration ja Apache Camel. Molempia voidaan toteuttaa XML-pohjaisesti tai sopivalla täsmäkielellä (DSL, Domain-specific language). Täsmäkieli on johonkin tiettyyn käyttötarkoitukseen luotu kielivariantti, jota ei voida yleisesti käyttää, ainakaan suoraan, muissa kehyksissä tai ohjelmointikielissä. Se voi perustua johonkin ohjelmointikieleen sisältäen hieman eroavaa syntaksia [45]. Molemmat kehykset sisältävät Java-pohjaisen täsmäkielen, mutta ne eivät ole keskenään yhteensopivia, syntaksi eroista johtuen.

Kehykset eroavat toisistaan käytännössä vain vähän ja molemmilla on mahdollisuudet luoda täysin vastaavia toteutuksia. Apache Camelilla on laajuuden ja mukautettavuuden kannalta hieman enemmän etuja. Se tukee Spring Bootia, jolloin pystytään hyödyntämään Spring-ohjelmistokehyksen automaattisia luokkien määrityksiä ja ohjelmia on nopeampi luoda [46]. Camelista löytyy myös komponentti, jolla voidaan luoda silta olemassa olevien Spring Integration -toteutusten rajapinnoille [47]. Spring Integration mahdollistaa olemassa olevaan Spring-toteutukseen viestipohjaista toiminnallisuutta ja se sopiikin erittäin hyvin valmiiseen Spring ekosysteemiin [48].

4. LÄHTÖTILANNE JA SUUNNITTELU

4.1 Integrointitapaus

Integrointitapauksena käytetään Unifaunin APIConnect-palveluihin kuuluvaa REST-rajapintaa toimitusten lähettämiseen ja rahtikirjojen tulostamiseen. Onnistuneesta kutsusta luodaan tarvittavat tulosteet kuten lähetyslaput. Rajapintaa käyttääkseen tarvitsee olemassa olevat tunnukset, jotka tutkimukseen saadaan yritykseltä [49]. Yritys hyödyntää Unifaunin palvelua muun muassa varastonhallinnassa (WMS, Warehouse Management System) tuotteiden toimituskirjausten luonnin yhteydessä.

Luotuja lähetyksiä voidaan tarkastella käyttämällä siihen tarkoitettua rajapintaa tai Unifaun Online-palvelua, jossa niiden tietoja voidaan selainpohjaisella sovelluksella tarkastella, tarvittaessa tulostaa lähetystarroja tai ladata niistä pdf-kopioita. Unifaun Onlinessa voidaan hallita muutakin lähetyksiin liittyvää, kuten tiedot mahdollisista lähettäjistä, vastaanottajista ja käytettäviä kuljetusliikkeitä. Lähetystenhallinnan lisäksi palvelussa voidaan määritellä esimerkiksi tulostusasetuksia [50]. Esimerkkeinä tulosteista kuvassa 9 on Unifaun Onlinesta tulostetut DB Schenkerin ja Postin lähetystarrat.

Kuva 9 *Unifaun Onlinesta tulostetut lähetystarrat*

4.2 Yrityksen järjestelmät ja integraatioratkaisu

Yrityksen omat tuotteet käyttävät yhteistä Oracle-tietokantaa, jonka kautta sovellukset ovat integroituneet keskenään. Oracle-tietokannoissa käytetään PL/SQL (Procedural Language for SQL) -varianttia perinteisestä SQL-kyselykielestä. Kielivariantilla on muutamia syntaksi ja toimintaeroja, mutta ne eivät ole tutkimuksen aiheen kannalta olennaisia, joten niitä ei käsitellä tarkemmin. Tiedon tallennus ja lukeminen tapahtuu joko suorilla SQL-kyselyillä tietokannanhallintasovelluksessa, kuten esimerkiksi DBeaver tai SQL Developer, käyttäen säilöttyjä aliohjelmia (engl. stored subprograms) tai käyttäen PL/SQL-paketteja. Aliohjelmilla tarkoitetaan joko funktiota tai proseduuria. Näiden ero on se, että funktio palauttaa suorittamisen jälkeen jonkin paluuarvon, kun taas proseduurit eivät palauta mitään takaisin. Paketti tarkoittaa kokonaisuutta, joka sisältää loogisesti toisiinsa liittyviä määritelmiä kuten tyyppejä, muuttujia sekä aliohjelmia. Näiden pakettien

tarkoituksena on mahdollistaa PL/SQL-koodin uudelleenkäyttöä muiden ohjelmien välillä [51].

Logiikan yhtenäistämiseksi yritykseltä löytyy tällä hetkellä jatkuvasti kehityksessä oleva Javaan perustuva Spring ohjelmistokehyksen mukainen sisäinen rajapinta. Kokonaisuus hyödyntää useita Springin kirjastoja, kuten Cloud, sekä Security tärkeimpinä mainintoina. Rajapintaa voidaan kutsua joko REST API:n kautta tai käyttäen sille Javalla toteutettua asiakaskirjastoa. Rajapinta on merkittävässä roolissa integraatioiden toteutuksessa, sillä sitä käytetään tarvittavien tietojen hakemiseen yrityksen tietokannasta.

Yrityksen tarjoama integraatiojärjestelmä on myös Java-pohjainen ratkaisu, joka mukautetaan integroitavan järjestelmän ja asiakkaan tarpeisiin. Toiminnaltaan kokonaisuus on vastaa keskitettyä ratkaisumallia, sillä se tarjoaa ulospäin rajapinnan ja huolehtii tarvittavat muunnokset ohjelman sisällä. Näin ollen kutsuvan ohjelman ei tarvitse tietää integraatiologiikkaa. Koska toteutus ei perustu mihinkään kehykseen, yhdenmukaista tapaa tehdä ei ole. Merkittävimpiä kirjastoja käytössä on JAX-WS SOAP-kutsuille, JAX-RS REST-kutsuille ja FasterXML:n Jackson, jolla voidaan luoda Java-luokista esimerkiksi JSON-formaatissa merkkijonoja. Tämä järjestelmä on tutkimuksen varsinaisena kohteena, eli etsitään moderneja vaihtoehtoja sen sisältämien toteutusten siirtämiseksi integraatioalustalle ja verrattaville integraatiokehyksille.

Yrityksen integraatiotapauksen toteutus on osa suurempaa integraatioihin keskittynyttä projektia. Se sisällytetään muista integraatioista poiketen pääprojektiin JAR-pakettina, joka pääprojektin tapaan puretaan Apache Tomcat -sovelluspalvelimella. Kun paketti on purettu, aukeaa rajapinta sen kutsumiseen. Yritykseltä löytyy Unifaunin lähetykseen myös vanhempi SOAP-palvelu, mutta tutkimuksessa keskitytään REST-rajapintaa hyödyntävään palveluun.

4.3 Ongelman kuvaus ja motivaatio

Yrityksen useat tuotteet pohjautuvat Javaan, jonka versiot vaihtelevat tuotteesta riippuen. Tutkimuksessa käytettävä integraatiototeutus käytti työn alkaessa Javan kehitystyökalujen (JDK, Java Development Kit) versiota 7. Tämä on julkaistu vuonna 2011 ja sen pääasiallinen tuki on päättynyt, joten se oli vanhentunut ja aiheutti jatkuvasti kehityskuormaa. Useat projektit, kuten kesken työn myös tutkittava integraatioprojekti, on yrityksen sisällä kuitenkin päivitetty hieman uudempaan Java JDK 8 versioon, joka on laajennetun tuen julkaisu (LTS, Long-Term-Support). Sen pääasiallinen päivitystuki on myös kuitenkin päättymässä. Moderni integraatioalusta tai -kehys voisi tuoda helpotusta päivityskuormaan tai Java version nostamisessa seuraavaan LTS-julkaisuun, joka on

kirjoitus hetkellä JDK 11. Ajan tasalla olevat teknologiat varmistavat osaltaan myös viimeisimmät tietoturvapäivitykset [52].

Koska integraatiototeutus räätälöidään asiakkaan ja integroitavan järjestelmän tarpeiden mukaisesti, ne ovat tyypillisesti hyvin mukautettuja. Yhtenäisellä alustalla tai kehyksellä voitaisiin eliminoida toistuvasti samojen asioiden tekemistä ja kierrättää toiminnallisuutta. Tällöin voidaan tapauskohtaisesti mukautettavat osat tehdä erikseen omina pienempinä kokonaisuuksina valmiin pohjan päälle. Yhdenmukaisesti toteutettujen rajapintojen ansiosta myös niiden dokumentoinnista tulee yhdenmukaisempaa ja yksinkertaisempaa. Alusta voisi tarjota rajapintojen dokumentaatioita automaattisesti tai tarjota työkaluja niiden luomiseen.

Vähän ohjelmointia vaativilla integraatioalustoilla yksinkertaisia integraatioita voisi olla mahdollista tehdä muidenkin, kuin kehittäjien. Silloin alustan täytyisi olla mahdollisimman yksinkertainen käyttää vähemmällä teknisellä osaamisella, jonka ansiosta integraatioiden luontiin voitaisiin hyödyntää laajemmin organisaation työvoimaa.

Yleisesti tunnettu alusta tai kehys toisi myös rekrytointimielessä yritykselle uusia mahdollisuuksia. Uusien osaajien etsiminen olisi suoraviivaisempaa, sillä hakuprosessissa voidaan tarkennetusti hakea ehdokkaita tietyn integraatio osaamisen perusteella, esimerkiksi Boomi- tai Spring Integration -kehittäjä. Lisäksi suositulle alustalle on usein tarjolla konsultointiapua ulkoisista yrityksistä, kuten Solita [53]. Tällöin uuden projektin käynnistäminen sujuu tarvittaessa järjestelmän tuntevan tahon avulla.

4.4 Toteutuksen vaatimukset

Alustoille ja kehyksille määriteltiin vaatimuksia, joita niistä tulisi löytyä onnistuneen lopputuloksen saavuttamiseksi. Tärkeimmät vaatimukset ovat yhdenmukainen toteuttaminen, modernit autentikointitavat ja perinteisten integraatiotapojen tukeminen, kuten tiedostojen käyttäminen yleisimpien tiedostoformaattien avulla. Yleisiä tiedostoformaatteja ovat esimerkiksi CSV ja XML. Lisäksi toteutuksille määriteltiin vaihtoehtoisia vaatimuksia, jotka tuovat lisäarvoa aiempaan toteutukseen nähden. Lisäarvoa tuovia ominaisuuksia ovat muun muassa automaattiset rajapintadokumentaatiot, muiden pilvipalveluiden yhteensopivuus ja yksinkertaisten integraatioiden toteutus muiden kuin kehittäjien toimesta.

Vaatimukset koskevat lähtökohtaisesti integraatioalustoja, sillä valitut kehykset perustuvat Java ohjelmointikieleen, johon vaadittavia ominaisuuksia on mahdollista toteuttaa

itse. Integraatiokehyksillä toteutettujen ohjelmien vertailuissa otetaan kuitenkin huomioon kehyksien tuomia helpottavia lisäominaisuuksia vaatimusten suhteen, varsinkin yhdenmukaisen toteutuksen osalta.

Jotta toteuttaminen olisi johdonmukaista, tulisi alustalla toteuttaminen olla arkkitehtuurillisesti yhdenmukaista. Tämä edesauttaa huomattavasti integraatioiden ylläpitoa ja toteuttaminen on helpompaa, kun tarvitsee opetella ainoastaan perusteet, jonka jälkeen muut toteutukset tehdään samoilla periaatteilla. Näin voidaan keskittyä varsinaisen tapauksen toteuttamiseen ja siihen liittyvien ongelmien ratkaisuun. Integraatioalustoille esitellyn määritelmän mukaan ja alustoja etukäteen tutkimalla tämä pitäisi lähtökohtaisesti toteutua kaikilla valituilla alustoilla. Yhdenmukaisuuteen koettiin myös vaikuttavan käyttöliittymän helppokäyttöisyys, sillä jos navigointi valikoiden välillä on intuitiivista, pystytään keskittymään oikeisiin asioihin ja saavuttamaan yhteneväisiä lopputuloksia.

Moderneista autentikointitavoista ollaan kiinnostuneita erityisesti OAuth 2.0 -protokollan valtuutusvirroista (authorization flow) ja niiden hyödyntämistä integraatiorajapintakutsujen autentikoinnissa. Yritykseltä löytyy palvelu valtuutusavaimien hakuun käyttäjätunnusten avulla. Nykyinen integraatiototeutus on suojattu Basic Auth -menetelmällä ja integraatiotapauskohtaisesti tunnuksia haetaan tietokannasta.

Nykyinen integraatiototeutus sisältää myös integraatioita järjestelmiin, joissa integraatio toteutetaan tiedostojen avulla, käyttäen esimerkiksi CSV-tiedostoformaattia. Kutsuja tehdään myös käyttäen SOAP (Simple Object Access) -protokollaa, jossa tieto kulkee XML-formaatissa. Koska integraatiot ovat yhä käytössä, alustan tulisi kyetä tukemaan näitä esimerkkitapauksia, jotta niiden toteutukset voidaan siirtää alustalle myöhemmin.

Yrityksellä on olemassa AWS-pilviympäristö, jonka avulla jatkossa rajapintoja tarjotaan ulkopuoliseen käyttöön. Mikäli integraatioalustalla on mahdollista helposti yhdistää pilvipalveluihin, toisi se jatkossa lisäarvoa. Yhdistäminen voi tapahtua esimerkiksi jonkin liitännäiselementin kautta integraatioprosessien luonnissa.

4.5 Teknisen toteutuksen suunnittelu

Suunnittelu alkoi eristämällä aiemman toteutuksen pohjalta integraatiotapauksen kannalta keskeiset tapahtumat. Näiden perusteella luotiin liitteen 1 mukainen tapahtumakaavio, jota käytettiin pohjana kaikille toteutuksille.

Nykyinen integraatiototeutus hyödyntää yrityksen tietokannasta haettavia tietoja. Yrityksen sisäinen rajapinta sekä tietokanta vaativat kutsujen tulevan samasta verkosta. Näin ollen täytyy alustan, tai kehyksen avulla rakennetun ohjelman sijaita yrityksen palvelimilla tai sillä on oltava muuten pääsy yrityksen sisäverkkoon. Integraatiokehyksillä tätä

rajoitusta ei lähtökohtaisesti ole, sillä niitä suoritetaan pääsääntöisesti itse, eli kehittäjän tietokoneella tai yrityksen palvelimella. Tällöin ohjelmalla on automaattisesti pääsy yrityksen verkkoon ja sitä kautta tietokantaan.

Sisäverkkoon päästään alustasta riippuen erilaisilla strategioilla. Yksi strategia on ylläpitää koko alustaa omilla palvelimilla [39]. Toinen käytetty strategia on hybridimalli, josta löydettiin kaksi erilaista varianttia. Ensimmäisessä variantissa pilvipalveluun yhdistetään erikseen palvelimella ajossa olevia paikallisia ohjelmia, jotka kykenevät suorittamaan integraatioprosessit suoraan palvelimella [35]. Toinen hybridivariantti on yhdyskäytävä (engl. gateway), joka yhdistää pilvipalvelun, sekä sisäverkossa sijaitsevat resurssit, jolloin päästään esimerkiksi hakemaan käytettävää tietoa yrityksen omasta tietokannasta. Yhdyskäytäväratkaisussa integraatiototeutuksia ei usein suoriteta paikallisesti palvelimella, vaan ainoastaan käytetään sisäverkossa sijaitsevia resursseja [44].

Lähetykseen tarvittavat tiedot haetaan yrityksen sisäisen rajapinnan avulla. Tämän rajapinnan taakse tehdään osana tutkimusta vastaavaa logiikkaa, kuin yrityksen nykyisessä integraatiototeutuksessa. Rahtikirjan tunnuksella tehdään tarvittavat PL/SQL-kutsut ja tiedot yhdistetään ulospäin tarjottavaksi objektiksi. Integraatioprosessin tehtävä on muuttaa saatavat tiedot Unifaun-lähetyksen rajapinnan vaatimaan JSON muotoon.

Pääsy ulkoverkosta voidaan tarvittaessa toteuttaa tarjoamalla edellä mainittua rajapintaa yrityksen julkisen standardirajapinnan kautta. Ensisijaisesti tämän tutkimuksen alussa päätettiin, että käytettävän alustan olisi hyvä toimia sisäverkossa, jolloin esimerkkitapauksen toteuttaminen olisi yksinkertaisempaa. Kaikki toteutukseen valitut integraatioalustat tarjoavat mahdollisuutta ylläpitää alustaa, tai sen suorittavia komponentteja omilla palvelimilla, jolloin pääsy sisäverkkoon on mahdollista.

Ratkaisumalleista vaihtoehdoiksi nousi suunnitelmien perusteella keskitetty malli tai ESB, riippuen alustan tai kehyksen tarjoamista mahdollisuuksista. Integraatiotapoja voidaan hyödyntää tapauskohtaisesti, mutta pääasiassa pyritään hyödyntämään viestipohjaista integraatiotapaa. Valituilla malleilla mahdollistetaan keskitettyä toimintaa, jolloin rajapintaa kutsuvan ohjelman ei tarvitse tietää kohteen rajapinnan vaatimuksia ja se jää integraatiototeutuksen hoidettavaksi.

5. INTEGRAATION SIIRTÄMINEN

5.1 Alustat

Koska yksikään valittu alusta ei tue suoraan olemassa olevan ohjelmakoodin siirtämistä sellaisenaan tai muokattuna alustan käyttöön, aloitetaan toteuttaminen aiemmin luodun prosessikaavion mukaisesti, joka on liitteessä 1.

Tiedon muuntaminen on Azuren Logic Apps -toteutusta lukuun ottamatta helppo vedä ja pudota ratkaisu, jossa graafisella käyttöliittymällä lähdetiedon kenttä raahataan vastaavaan kohdetiedon kenttään. Siltikään ilman ongelmia ei selvitty. Alustat eivät oletusasetuksilla välittäneet HTTP-kutsujen luomia virheviestejä tai järjestelmän virheistä ei saatu usein selviä syitä ongelmiin. Ratkaisuja täytyi etsiä järjestelmän lokeista tai alustan omilta foorumeilta. Dell Boomin HTTP-kutsu elementti tarjosi asetuksista mahdollisuutta HTTP-virheviestien palauttamiselle, mutta silti alusta ei välitä näitä viestejä kutsujalle oletuksena.

Esimerkkitiedoston pohjalta luotuun JSON-malliin täytyy tehdä muutoksia kenttien tyyppimäärityksiin. Vapaaehtoisille kentille lisättiin varsinaisen tietotyypin lisäksi null tyyppi, sillä monilla alustoilla suoritus katkesi, jos kentän toteutuneeksi arvoksi tuli null, eikä tätä ollut malliin määriteltynä.

5.1.1 Boomi

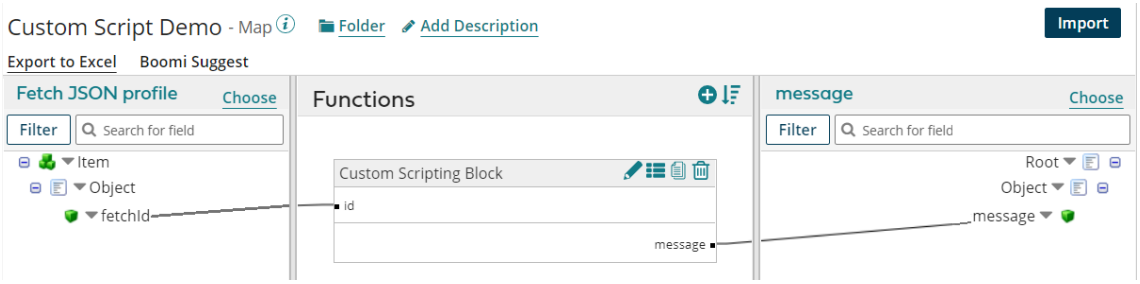
Ajoympäristön asennus

Integraatioprosesseja paikallisesti suorittaa Atom, joka asennetaan palvelimelle asennusohjelman avulla. Asennus sisältää kaiken tarvittavan integraatioprosessien suorittamista varten. Atom-asennuksen yhdistäminen alustalle tapahtuu asennusohjelmassa kirjautumalla tunnuksilla, tai syöttämällä lataussivulla näytettävä yksilöivä avain. Onnistuneen asennuksen jälkeen Atom suoritetaan palveluna kohdepalvelimella [63]. Atom-ilmentymä näkyy yhdistämisen jälkeen alustan hallintanäkymässä. Hallinnassa nähdään reaaliajassa kaikkien ilmentymien tila ja versiotiedot. Lisäksi näistä ilmentymistä voidaan esimerkiksi hakea niiden tuottamia lokitiedostoja halutulta aikaväliltä tai katsoa sille julkaistut integraatioprosessit ja niiden kuuntelijat.

Tiedon muuntaminen

Alustalla tiedon siirtämisen ja muuntamisen keskeisiä komponentteja ovat erilaiset profiilit. Ne mahdollistavat tiedon muuntamisen helposti yhdistämällä profiilin kenttiä toisiinsa, sekä luotaviin ajonaikaisiin muuttujiin voidaan valita arvoksi suoraan profiilista löytyvä kenttä. Tutkimuksen osalta käytetään JSON-profiileja ja niitä voidaan luoda luontinäkymässä kuten muitakin resursseja. Niiden rakenne voidaan määrittää joko suoraan JSON-mallina tai esimerkkitiedoston avulla. Esimerkkitiedoston pohjalta alusta muuttaa tiedon mahdollisimman tarkasti JSON-malliksi ja edelleen profiiliksi. Profiilit ovat prosessin aikana yleisesti käytettyjä ja niillä saadaan helposti tehtyä tarkastuksia tiedon oikealle rakenteelle. Profiileja alustalla on JSON-formaatin lisäksi esimerkiksi database, XML, EDI tai tiedosto. Tiedostoprofiili voi olla esimerkiksi CSV-muotoinen [56].

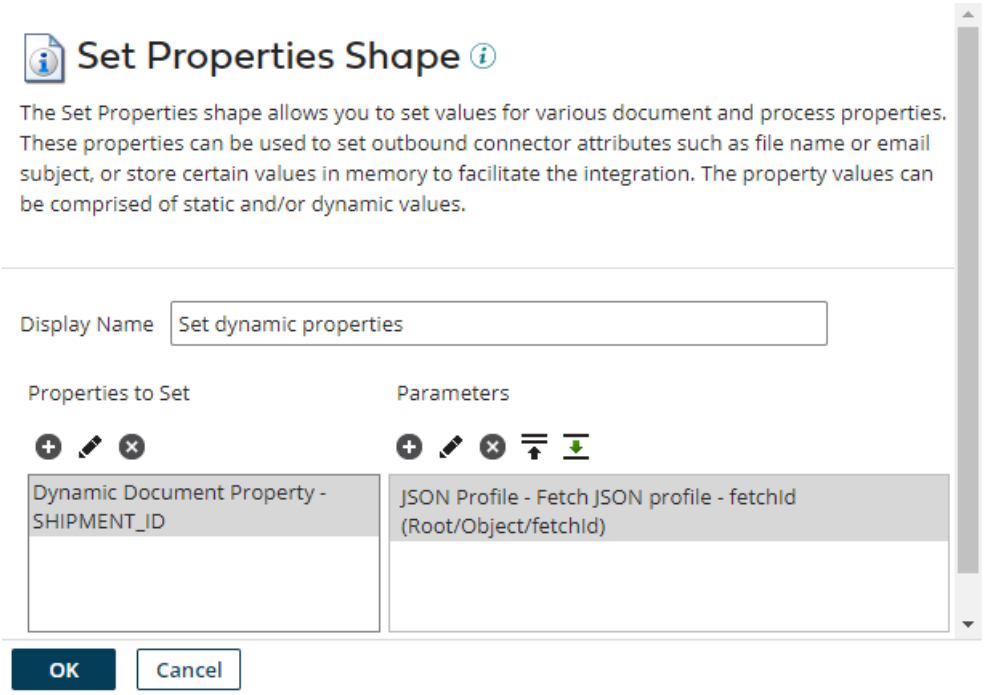
Varsinainen muuntaminen tapahtuu alustan Map-elementillä, jossa sisään ja ulos tulevan tiedon rakenne valitaan tässä tapauksessa JSON-profiileilla. Varsinainen muuntamisoperaatio on yksinkertainen, eli raahataan muunnettavan tiedon kenttiä toisella puolella olevan ulostulo profiilin kenttiin. Valittujen profiilien ei tarvitse olla JSON-profiileja, vaan muuntaminen onnistuu kaikkien eri profiilien välillä [64]. Kuvassa 10 on esimerkki muunnoksesta, jossa välissä on mukautettu skriptilohko.



Kuva 10 *Tiedon muokkaukseen käytettävä Map-elementti*

Map-elementti huolehtii kenttien muuntamisen profiilin JSON-mallin mukaisesti. Jos määritellyssä mallissa ei kuitenkaan oteta kaikkia mahdollisia tyyppejä huomioon heittää elementti virheen, jos todellinen tieto ei vastaa hyväksyttyjä tyyppejä. Tämä huomattiin tietokannasta tulleilla null arvoilla, joka korjaantui määrittämällä JSON-malliin null tyypit hyväksytyksi.

Tiedon muuntamisen lisäksi ajonaikaisia muuttujia voidaan luoda käyttämällä Set Properties -elementtiä. Kuvassa 11 on esimerkki JSON-profiilin yksittäisen kentän talletaminen ajonaikaiseen muuttujaan.



Kuva 11 *Rahtikirjanumeron asettaminen muuttujaan Set Properties-elementissä*

Integraatioiden luonti

Integraatioiden toteuttaminen tapahtuu kokonaisuudessaan graafisen käyttöliittymän kautta, eikä ulkopuolista ohjelmointiympäristöä tarvita. Toteutus tehdään alustan Build, eli luontinäkymässä, jonka sisällä voidaan luoda ja muokata käytettäviä resursseja kuten esimerkiksi API-määrityksiä, integraatioprosesseja, JSON-profiileja ja yhteyspohjia. Aluksi luodaan integraatioprosessi, jota voidaan myöhemmin myös kutsua API-määrityksen avulla.

Jokainen integraatioprosessi alkaa aloituselementillä, joka voidaan määritellä usealla tavalla. Se voi olla liitin, joka voidaan edelleen määritellä kuuntelijaksi. Tämä kuuntelija odottaa erilaisten protokollien kutsuja, esimerkiksi Web-protokollat kuten HTTP ja SOAP tai FTP [32]. Liitin voidaan myös määritellä vastaanottamaan tai lähettämään jonkin ulkoisen palvelun tietoja, kuten SAP-toiminnanohjausjärjestelmän [54]. Toinen vaihtoehto elementille on yhteistyökumppanilta saadun EDI-tiedon käsittelijä, jossa saatu dokumentti on jonkun EDI-standardin, kuten X12, mukaista. Näiden lisäksi aloituselementti voi välittää edelliseltä prosessilta saatu tiedon eteenpäin prosessin käsiteltäväksi. Viimeinen vaihtoehdossa aloituselementti ei vastaanota mitään tietoa, vaan prosessi alkaa tyhjästä dokumentista [32]. Tutkimuksessa aloituselementiksi määriteltiin Web Service-kuuntelijaksi, joka voidaan halutessaan määritellä SOAP- tai REST-rajapinnaksi. Rajapinnaksi valittiin REST, koska se tukee useampaa tiedostoformaattia, tärkeimpänä

JSON-formaatti. SOAP-rajapinta tukee ainoastaan XML-formaattia. Aloituselementin Web Service määrityksiin valitaan missä muodossa tuleva tieto pitäisi olla ja tarkistaessa elementti heittää virheen, jos tieto ei vastaa määrityksiä [55].

Kun aloituselementti ja sen avaaman rajapinnan määritykset ovat tehty, voidaan siirtyä seuraavaan vaiheeseen. Aloituselementissä on määritelty sisään tuleva JSON-profiili, jossa sisältönä tulee rahtikirjanumero hyötykuorman mukana. Rahtikirjanumero asetetaan ajonaikaiseen muuttujaan Set Properties -elementin avulla myöhempää käyttöä varten.

Seuraavaksi täytyy luoda yrityksen sisäisen rajapinnan kutsumiseen käytettävä HTTP-liitinelementti. Elementti määritellään vastaavaan tyyliin kuin aloituselementti ja siihen määritellään erikseen uudelleenkäytettävät yhteys- ja toimintomääritykset. Yhteysmäärityksiin lisätään käytettävän palvelimen pohjapolku ja sen todentamisasetukset [57]. Todentamiseen voidaan käyttää tavallista tunnus ja salasana paria tai määritellä se käyttämään OAuth-todentamismenetelmää [58]. Jälkimmäistä kokeiltiin myös onnistuneesti ja saatiin valtuutustunniste yrityksen OAuth-palveluista. Lopulta tutkimuksessa tätä ei käytetty, koska molemmat rajapinnat tukevat ainoastaan tavallista todentamismenetelmää. Uudelleenkäytettävyyden takia yhteysmäärityksissä ei oteta vielä kantaa varsinaisiin kutsuasetuksiin tai käytettäviin profiileihin. Toimintomäärityksiin määritellään tarkemmin sisään ja ulos tulevat JSON-profiilit, käytettävä HTTP-metodi, mahdolliset kutsuotsikot ja polkuun voidaan tarvittaessa lisätä dynaamisia muuttujia tai jos rajapinnan URL-osoitteeseen täytyy lisätä tekstiä. Samaa elementtiä käytetään myös Unifauniin tehtävään kutsuun omilla määrityksillä [57].

Kun yrityksen sisäiseen rajapintaan tarvittava elementti on luotu, siirrytään muuntamaan saatu tieto Unifaun-rajapinnan vaatimaan muotoon. Muuntaminen tapahtuu aiemmin esitellyllä Map-elementillä. Kun Map-elementti on määritelty, luodaan uusi HTTP-liitinelementti Unifaun-kutsua varten. Yhteysmäärittelyihin lisätään Unifaun-rajapinnan pohjaosoite ja tunnukset autentikointiin, jotta näitä asetuksia voidaan käyttää uudelleen toisen päätepisteen kanssa. Toimintomäärityksiin täsmennetään päätepisteen lopullinen polku, lisätään käytettävät kutsuotsikot ja sisään tulevaksi profiiliksi Map-elementistä saatava JSON. Ulos tuleva profiili on sama, joka määritellään aloituselementissä sen käyttäjälle palauttavaksi profiiliksi. Vastauksen välittämiseksi aloituselementille, lisätään prosessin jokaisen polun viimeiseksi Return Document -elementti. Elementillä ei ole mitään asetuksia, vaan se ilmaisee prosessin päättymisen ja palauttaa vastauksen kutsujalle.

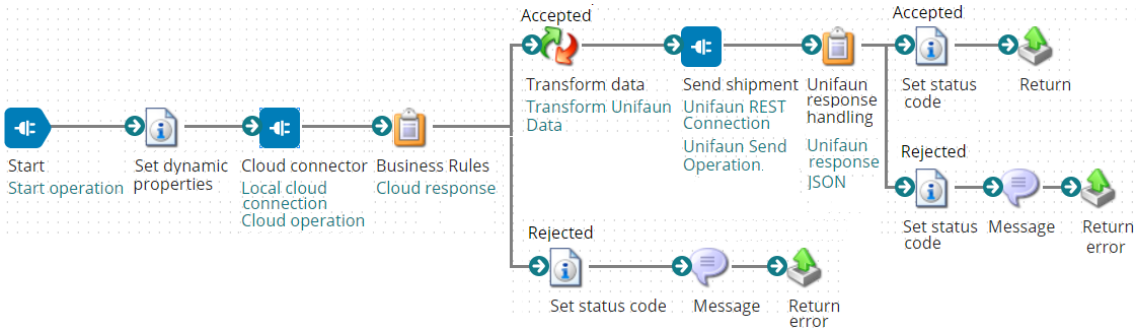
Toteutusta testatessa havaittiin virheiden hallinnan olevan oletuksena puutteellista. Lokeja luotiin useampiakin, mutta ne olivat usein kovin vajaita. Esimerkiksi tiedon muuntamisessa käytetty Map-elementti ei onnistunut muuntamaan dataa, jolloin virheen paikannus, eli debug-loki, ei tarjonnut oikeastaan mitään yksiselitteistä syytä mistä epäonnistuminen johtui. Usein virhe johtui null-tyyppien hallinnasta, jolloin tiedon asettaminen profiilin mukaisesti epäonnistui. Rajapinnoista tulleet virheet saatiin hallittua käyttämällä Business Rules -elementtiä. Elementtiin määriteltiin sääntö, jossa rajapinnasta saadun vastauksen HTTP-tilakoodi pitää alkaa numerolla 2, eli sen täytyy olla onnistunut kutsu, kuten esimerkiksi 200 OK tai 201 CREATED. Virheen sattuessa, eli tilakoodin ollessa edellä mainitun säännön vastainen, virheviesti rakennettiin käyttämällä Message-elementtiä ja tilakoodi määritetään käyttämällä aiemmin esiteltyä Set Properties -elementtiä. Elementin avulla tilakoodia vastaavaan prosessimuuttujaan sijoitetaan edellisestä HTTP-rajapintaelementistä saatu tilakoodi. Onnistuneille kutsuille operaatio tehdään myös siksi, että huomattiin ulos tulevan tilakoodin olevan 200 OK, eikä Unifaun-rajapinnan vastauksessa tuleva tilakoodi.

HTTP-kuuntelijan avaama rajapinta on sellaisenaan käytettävissä, mutta muista alustoista poiketen prosessista tehdään myös API-palvelu. Syy tähän ratkaisuun on alustan tarjoama automaattinen Swagger-dokumentaatio API-palveluille. API-komponentti luodaan samassa luontinäkymässä kuin integraatioprosessit ja sille voidaan määritellä rajapinnan käynnistämä prosessi, polku ja käytettävät profiilit. Profiileille täytyy tehdä muutoksia oletuksena luotuun rakenteeseen. Kaikkien JSON-profiilien pääelementtien nimeksi tulee automaattisesti Root, jolloin alustan API-komponentin määrityksissä ei pystytä erottamaan kahta profiilia toisistaan. Kun pääelementit on nimetty toisistaan eroavaksi, täytyy palvelimelle asentaa API-yhdyskäytävä, jolle API-komponentteja voidaan julkaista.

Yhdyskäytävän asentaminen tapahtuu ajoympäristön tapaan yksinkertaisesti asennusohjelman avulla, johon voidaan syöttää tunnukset tai käyttää yhdyskäytävän lisäysnäkymässä tarjottavaa tunnistetta [60]. Onnistuneen asennuksen jälkeen yhdyskäytävä on näkyvissä hallintapaneelissa. Jotta olemassa oleva Atom-ajoympäristö voidaan liittää yhdyskäytävään, sille täytyy tehdä ympäristön siirtämismääritykset (Environment Migration). Esivaatimuksena Atom-ilmentymälle täytyy muuttaa sen API-tyyppi Advanced tilaan ja tunnistautumistavaksi Gateway [61]. Siirtämismääritysten jälkeen valittu Atom-ilmentymä toimii kyseisen yhdyskäytävän ajoympäristönä.

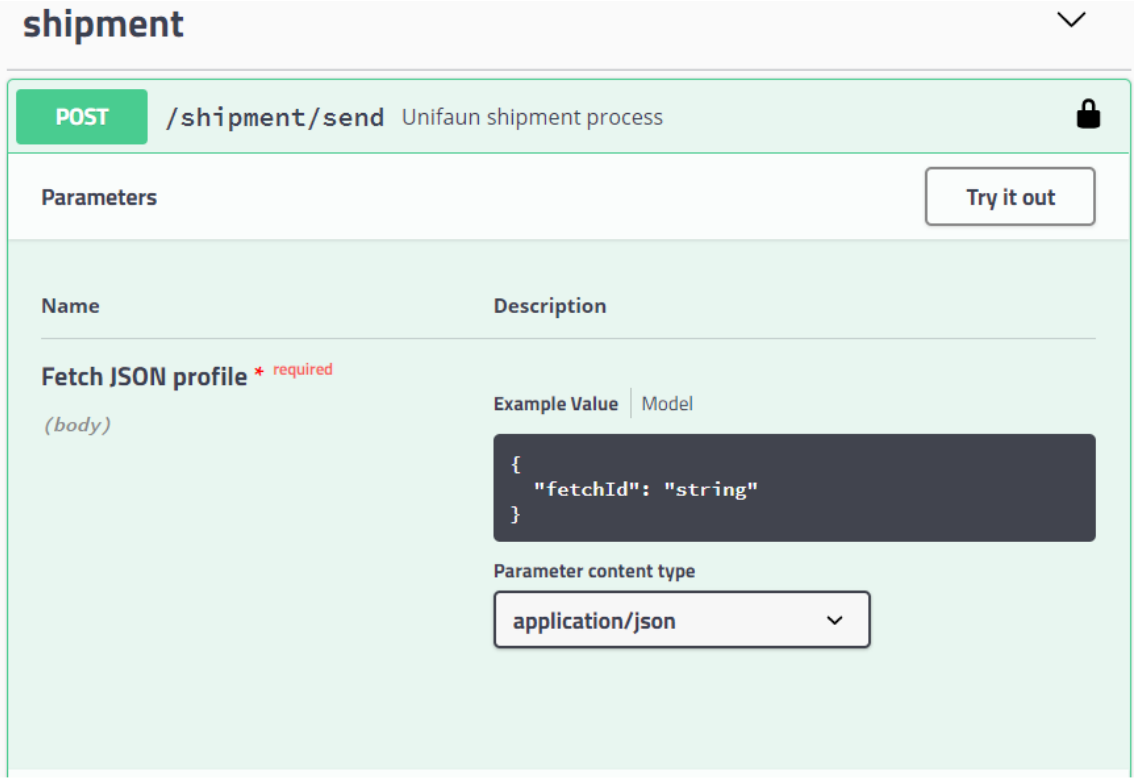
Integraatioiden julkaiseminen

Lopullinen integraatioprosessi ja sen elementit ovat nähtävillä kuvassa 12. Prosessi voidaan nyt julkaista ajoympäristöön. Luontinäkymän yläreunasta löytyy testaukseen ja julkaisuun painikkeet, josta ne voidaan suorittaa. Web Service-kuuntelijalla luodun aloituselementin heikkoutena on se, että niitä prosesseja ei voida testata käyttöliittymän kautta, vaan prosessi pitää julkaista olemassa olevaan ajoympäristöön ja testata sitä julkaistussa versiossa. Julkaisusta luodaan ensin paketoitu komponentti, joka julkaistaan ajoympäristössä. Paketit huolehtivat automaattisesti versioinnista ja versiointinumeron kasvattamisesta, mutta julkaistaessa versionumero voidaan myös määrittää itse [59]. Julkaisuponnahdus ikkunassa valitaan paketille kohdeajoympäristö ja onnistuessaan rajapinta on valmis kutsuttavaksi.



Kuva 12 *Lopullinen Boomi-integraatioprosessi*

API-palvelun julkaisu tehdään integraatioprosessin tapaan luontinäkymän yläreunassa olevasta julkaisunapista. Kutsu tapahtuu nyt API-palvelun lisäyksen yhteydessä määritettyyn polkuun. Julkaistujen API-palveluiden hallinnasta päästään tutkimaan REST-rajapintojen päätepisteitä graafisena Swagger-dokumentaationa, joka on nähtävissä kuvassa 13 [62].



Kuva 13 *Automaattisesti luotu Swagger dokumentaatiosivu*

Huomioita

Boomilla integraatiototeutusten tekeminen tapahtuu pääasiassa yhden näkymän kautta, joka tuo hyvin yhtenäisen kokemuksen. Monipuolisuutta tuo mahdollisuus luoda omaa logiikkaa muunnosten oheen ja itse muuntamisprosessi on varsin yksinkertainen.

Lokitus alustalla on kuitenkin keskinkertaista, eikä Atom-ilmentymän virhelokeista saatu useinkaan lopullisia syitä havaittua. Myös virheidenhallinta alustalla on oletuksena puutteellista ja alustan omista elementeistä tulleet virheet saadaan lopulta paremmin välittämällä ne suoraan vastauksena HTTP-kutsuun. Lisätyllä virheidenhallinnalla saadaan myös rajapintojen virheet hyvin hallittua ja välitettyä kutsujalle.

5.1.2 Anypoint

Ajoympäristön asennus

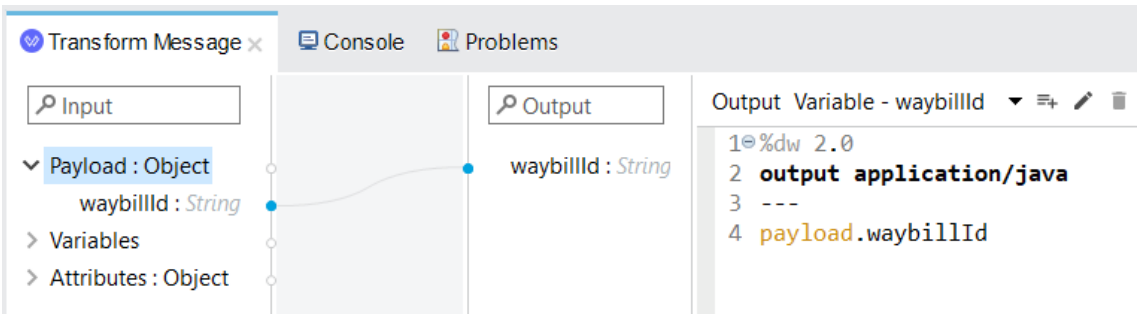
Anypoint-alustan julkaisustrategiaksi toteutukselle valittiin hybridimalli, jolloin saadaan suorittava ajoympäristö sisäverkkoon ja päästään käsiksi yrityksen sisäiseen rajapintaan. Toinen varteenotettava vaihtoehto olisi ollut esittelyssä mainittu Private Cloud -strategia, jossa koko alustaa ja sen komponentteja ylläpidetään itse omilla palvelimilla.

Tätä koetettiin toteuttaa, mutta lopulta sivuilta löydettyjen ohjeiden perusteella sitä ei saatu järkevän ajan kuluessa toimimaan ja päädyttiin hybridimalliin. Hybridijulkaisua varten asennettiin erillinen Mule-ajoympäristö, joka suorittaa sille julkaistuja ohjelmia, joita kutsutaan Mule-ohjelmiksi. Asennus tapahtui lataamalla se ZIP-pakettina ja purkamalla se johonkin hakemistoon.

Ajoympäristön lisääminen alustan hallintapaneeliin tapahtuu sivuston Servers-välilehdeltä, jossa käytössä olevat palvelimet ovat listattuina. Uutta palvelinta liittäessä antaa käyttöliittymä komennon, josta löytyy tarvittava komentolippu, sekä yhdistämiseen tarvittava tunniste. Kun komento on suoritettu onnistuneesti, Anypoint-alusta ja paikallisesti toimiva ajoympäristö vaihtavat salausavaimia ja lisätty palvelin näkyy käytettävien palvelimien listassa.

Tiedon muuntaminen

Alustan Transform, eli muuntajaelementti, huolehtii tiedon muokkaamisen lisäksi muuttujien luonnin sisään tulevasta hyötykuormasta. Prosessissa kulkevaa tietoa kuvataan käyttämällä tietotyyppejä, joita voidaan luoda halutessaan omia, tai käyttää alustan tarjoamia valmiita vaihtoehtoja. Muuntajan graafinen komponentti tuottaa automaattisesti oikealle puolelle näkyviin muokkausoperaatiota vastaavaa DataWeave-ohjelmakoodia. Koodia voidaan tässä muuttaa, jolloin myös graafinen näkymä päivittyy, mikäli muutokset pystytään näyttämään. Koodin muokkaaminen mahdollistaa tarkistuksien lisäämisen, kuten onko jokin arvo tyhjä merkkijono ja antaa sen perusteella esimerkiksi oletusarvo. Kuvassa 14 on muuntajaelementti, jossa saapuvassa hyötykuormassa tuleva rahtikirjanumero muutetaan järjestelmän muuttujaksi.



Kuva 14 *Muuttujan asettaminen Mulen Transform-elementissä*

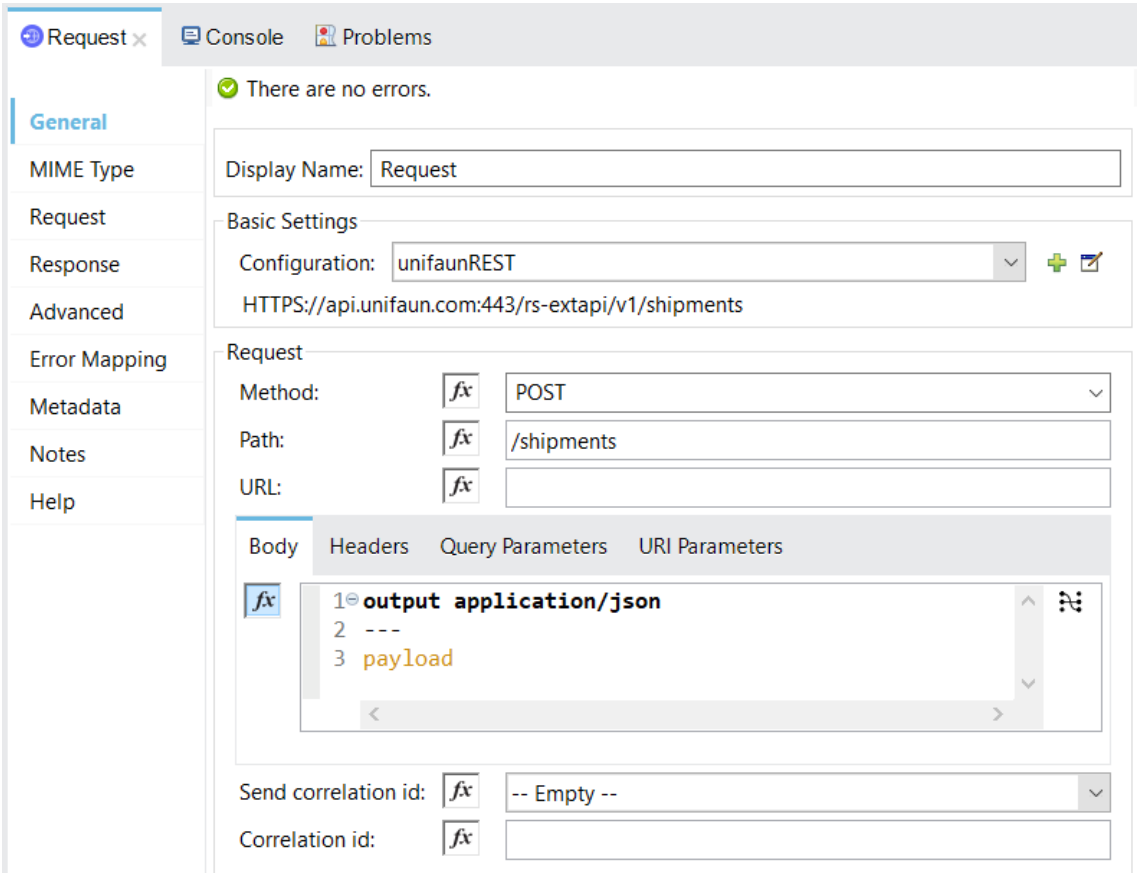
Integraatioiden luonti

Julkaisuun tarvittavien osien asennuksen jälkeen voidaan aloittaa integraatiotapauksen toteuttaminen. Kehitysvaiheessa Mule-ohjelmia voidaan suorittaa Anypoint Studion

avulla. Anypoint Studio on MuleSoftin kehittämä ohjelmointiympäristö rajapintojen ja integraatioiden suunnitteluun. Se perustuu avoimen lähdekoodin Eclipse-ohjelmointiympäristöön ja siinä on sisäänrakennettu Mule-ajoympäristö, jossa kehityksessä olevia ohjelmia voidaan testata [65]. Näin ei tarvitse julkaista ohjelmaa ennen kuin se on valmis tai sen toimivuutta tuotantoympäristössä halutaan kokeilla.

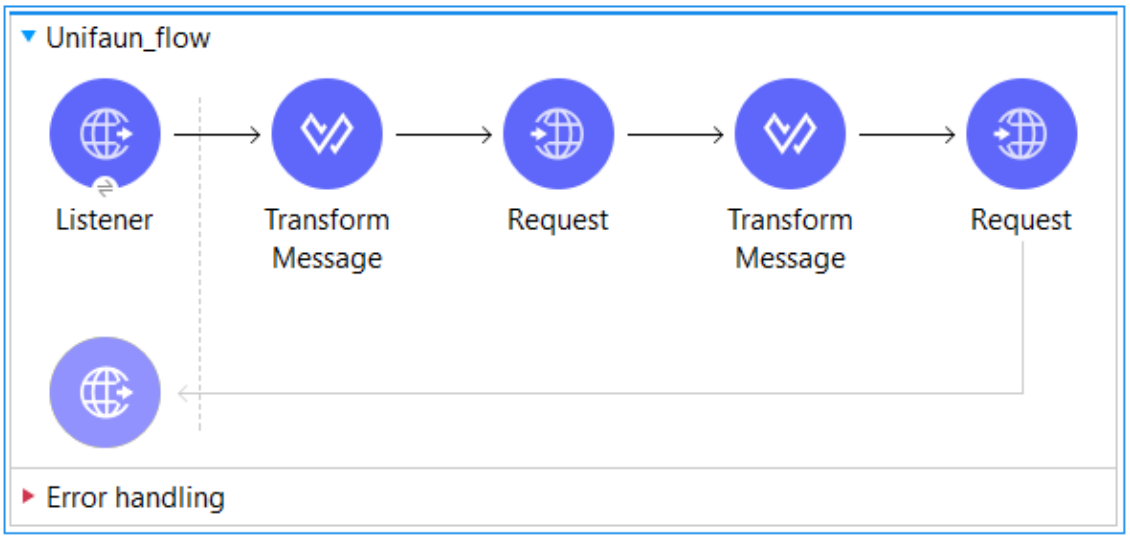
Integraation käynnistämiseen voidaan käyttää kuuntelijoita, jotka odottavat tietyn protokollan kutsuja kuten TCP, UDP tai HTTP [66]. Toteutuksessa käytettiin HTTP-kuuntelijaa, johon määriteltiin rajapinnan perusasetuksia, kuten isäntäkoneen IP-osoite ja portti. Lisäasetuksissa voidaan myös sallia vain tietyt HTTP-verbit ja työssä asetuksiin määritettiinkin POST ainoaksi sallituksi kutsutavaksi. Määritykset viimeistellään asettamalla rajapinnan osoite, johon kutsut tehdään [67]. Kutsun sisällössä kuljetetaan rahtikirjan numero, jota ollaan toimittamassa. Tämä numero kerätään hyötykuormasta ja sijoitetaan se ajonaikaiseen muuttujaan käyttäen alustan muuntaja elementtiä.

Yrityksen sisäiseen rajapintaan tapahtuva kutsu valmistellaan tarkoitukseen sopivalla HTTP-pyyntöelementillä. Asetukset ovat vastaavanlaiset kuin kuuntelijan osalta ja ensin sille määritellään perusasetukset. Näihin perusasetuksiin sijoitetaan rajapinnan isäntäpalvelimen asetuksia kuten IP-osoite, portti, autentikointi ja pohja-URL. Perusasetuksia voidaan käyttää uudelleen muissa vastaavissa elementeissä. Esimerkiksi jos samalla palvelimella sijaitsevan rajapinnan toiseen päätepisteeseen tehdään kutsu, voidaan käyttää kyseiselle palvelimelle aiemmin luotuja perusasetuksia. Päätepisteen kohdentava URL-osoitteen loppuosa määritellään tämän jälkeen jokaisen elementin omiin asetuksiin. Samaa elementtiä käytetään Unifaunin palveluihin tehtäviin kutsuihin, jonka asetukset ovat nähtävissä kuvassa 15.



Kuva 15 *Unifauniin käytetyn HTTP-kutsuelementin asetuksia*

Kun Unifaun-kutsun suorittava elementti on määritetty, palauttaa integraatiototeutus rajapinnoista saadut vastaukset kutsujalle. Oletuksena kuuntelijan virheidenhallinnan kautta virheestä luodaan hyvin yksinkertainen viesti. Jotta vastaukseen voidaan määrittää tarkemmat virheviestit, täytyy kuuntelijaan määritellä virheviestin asetuksia. Käytännössä tämä tarkoittaa sitä, että oletuksena välitettävästä virheobjektista valitaan palautettavaksi haluttu tilakoodi ja virheen hyötykuorma. Onnistuneen kutsun kohdalla elementissä tulee oletuksena hyötykuorma määriteltynä, mutta tilakoodin osalta määriteltiin sen palauttavan oletuksena 201 Created. Elementtien ollessa määriteltynä ja oikeilla paikoillaan on integraatiototeutus valmis julkaistavaksi. Lopullinen prosessi näyttää kuvan 16 mukaiselta.



Kuva 16 *Lopullinen Mule-integraatioprosessi*

Integraatioiden julkaiseminen

Toteutus voidaan nyt julkaista asennettuun ajoympäristöön. Julkaisua varten ohjelman saa ohjelmointiympäristöstä ulosvietyä JAR (Java Archive) -pakettimuodossa. Tämä paketti voidaan sellaisenaan julkaista aktiivisessa Mule-ajoympäristössä käyttäen Maven-lisäosaa tai komentoliittymäsovelluksella. Komentoliittymäsovellus on asennettavissa NPM-pakettienhallinnan avulla ja vaatii toimiakseen NodeJS-ohjelmistoympäristön [68]. Anypoint-pilviympäristön kautta paketti voidaan julkaista käyttäen sovelluksen julkaisuikkunaa, jossa syötetään sovellukselle nimi, valitaan julkaisukohde ja asennettava pakettitiedosto. Julkaisukohde voi olla vain käynnissä oleva paikallinen ajoympäristö tai pilviympäristö ja ne tulevat pudotusvalikkoon valittavaksi. Sovellus lisätään joko ZIP- tai JAR-paketointitiedostona, jonka palvelin purkaa ja suorittaa ajoympäristössä [69].

Kun julkaistu toteutus on käynnistynyt, voidaan luotua rajapintaa testata HTTP-kutsulla käyttäen esimerkiksi Postman API-testaustyökalua. Onnistuneesta kutsusta saadaan Unifaun-rajapinnalta vastauksena hyötykuormasta palvelun linkki tulostettavalle kollilapulle ja luodun lähetyksen tiedot.

Huomioita

Alustalla integraatioiden luonti ja tiedon muuntaminen tapahtuu hyvin vastaavalla tavalla kuin Boomilla. Tiedot muunnetaan ja käsitellään molemmissa erilaisten tietoprofiilien avulla. Erona on kuitenkin esimerkiksi muuntamisessa automaattisesti luotu skripti, jota

voidaan muokata tarvittaessa. Huomattavaa tässä on kuitenkin se, että graafinen näkymä voi mennä usein sekaisin liiallisesta muokkaamisesta. Julkaisun osalta Anypoint-alusta saa hieman moitetta, sillä sisäverkkoon yhteydessä olevat integraatioprosessit täytyy luoda paikallisesti omalla koneella. Tämä aiheuttaa sen, että toteutus täytyy julkaista esimerkiksi luomalla projektista asennettava JAR-paketti ja lisätä se erillisen graafisen käyttöliittymän kautta ajoympäristöön. Jos toteutus olisi täysin pilvipohjainen, julkaisu onnistuisi selainpohjaisen sovelluksen kautta.

Virhetilanteissa Mulen elementtien tuottamat virheet voivat olla vaikealukuisia. Lopulta virheen taustalla on kuitenkin looginen selitys ja usein keskustelufoorumeilta löytyi apua virheiden ratkaisemiseen. Kuvan 17 virheessä kyse on siitä, että muuntajaan merkitty JSON-malli oli muuttunut ja sen seurauksena DataWeave-skripti on väärän mallinen.

```
ERROR 2021-04-24 16:12:26,489 [[MuleRuntime].uber.03: [testing].testing2Flow.CPU_INTENSIVE @40a71fb8]
********************************************************************************
Message               : "You called the function 'Value Selector' with these arguments:
  1: String ("{\"shippingUnits\":[{\"date\":1569237675000,\"shippingUnitNumber\":\"1\",\"a...)
  2: Name ("printConfig")

But it expects one of these combinations:
  (Array, Name)
  (Array, String)
  (Date, Name)
  (DateTime, Name)
  (LocalDateTime, Name)
  (LocalTime, Name)
  (Object, Name)
  (Object, String)
  (Period, Name)
  (Time, Name)
Trace:
  at main (Unknown)" evaluating expression: "%dw 2.0
```

Kuva 17 *Mulen tiedon muunnoksessa tullut virhe lokitiedostossa*

JSON-profiiliin usein tehdyt muutokset aiheuttivat sekaannusta automaattisesti muodostettavalle DataWeave-skriptille. Muutokset päivittyvät muuntajaelementin graafisessa näkymässä, mutta satunnaisesti muutokset eivät näkyneet skriptissä, jolloin se ei enää vastannut graafista käyttöliittymää. Tyhjentämällä kentille määritetyt muunnokset ja määrittelemällä ne uudelleen saatiin ongelmatilanne korjattua.

5.1.3 Azure

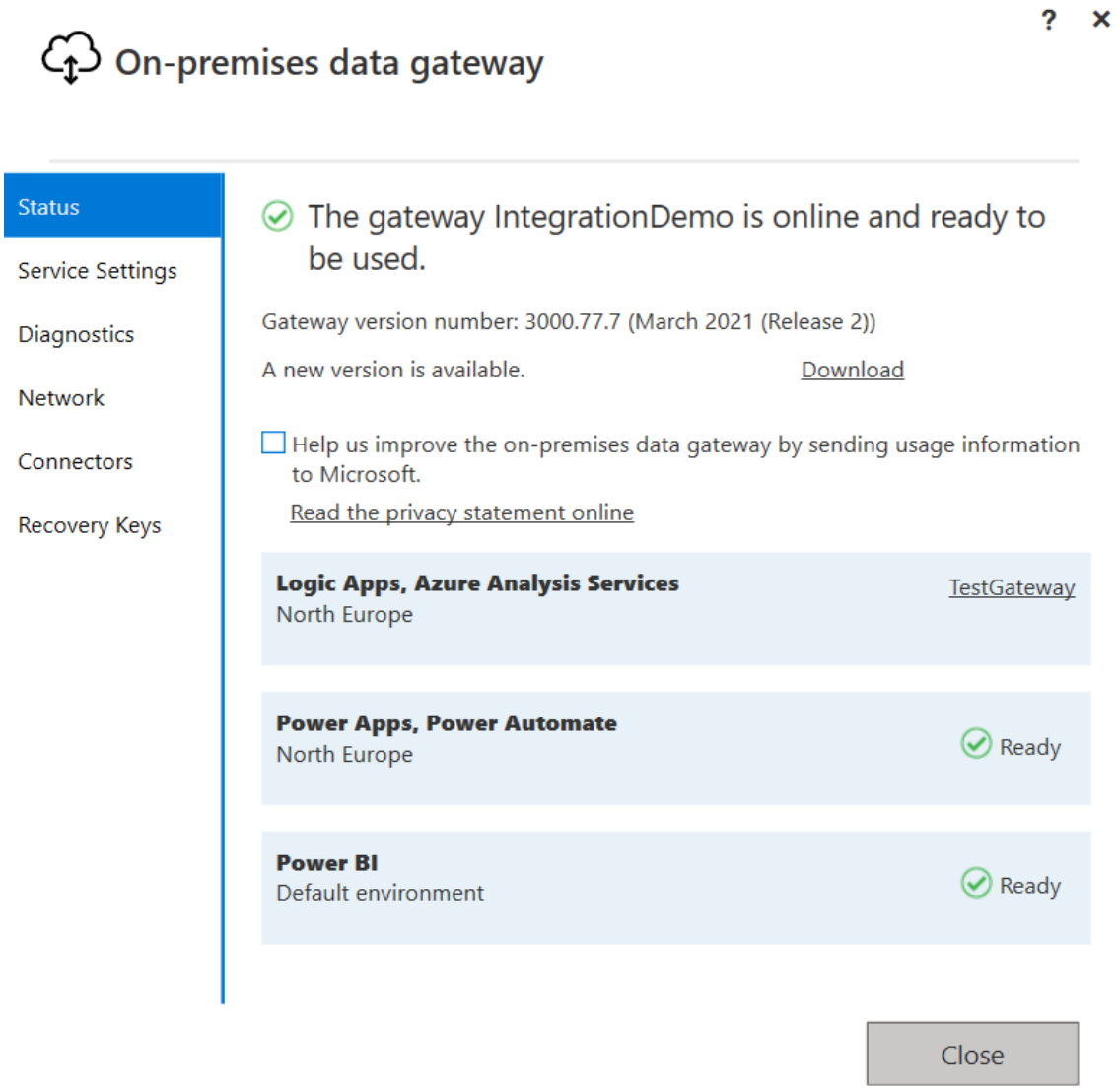
Ajoympäristön asennus

Sisäverkon resursseihin pääsyyn käytetään yhdyskäytävää. Yhdyskäytävä eroaa muista hybridimalleista siten, että integraatiototeutuksen suorittamisen sijasta se toimii siltana tiedonsiirrolle, mahdollistaen tiedonsiirron ja yhteyden salauksen keskitetysti sen kautta.

Yhdyskäytävä on asennettavissa ainoastaan Windows isäntäkoneille, mutta toimeksi antavalla yrityksellä on näitä jo ennestään, jolloin rajoitus ei aiheuta estettä sen käyttämiselle.

Yhdyskäytävän asennukseen on Microsoftin dokumentaatioissa yksityiskohtaiset ohjeet. Ennen aloittamista tulee olla käytössä aktiivinen Azure-tili. Asennusprosessi on yksinkertainen ja tapahtuu käyttäen Windows asennusohjelmaa [70]. Kun asennusohjelma on asentanut onnistuneesti yhdyskäytävän komponentit, seuraavaksi liitetään yhdyskäytävä aiemmin luodulle Azure-käyttäjätilille kirjautumalla asennusohjelman yhteydessä. Vaihtoehtoina tämän jälkeen on rekisteröidä uusi tai siirtää olemassa oleva yhdyskäytävä asennettavalle palvelimelle. Asennus viimeistellään luomalla palautusavain, jota voidaan myöhemmin käyttää yhdyskäytävän asentamiseen toiselle isäntäkoneelle, palauttamiseen tai haltuunottoon toiselta palvelimelta. Maantieteelliseksi sijainniksi tulee oletusarvoisesti käyttäjätilille määritelty sijainti. Tämä voidaan vaihtaa tarvittaessa asennuksen viimeistelyn yhteydessä.

Kun asennus on onnistunut, siirrytään kuvan 18 mukaiseen yleisnäkymään, josta nähdään palveluiden tila. Tässä näkymässä tulee myös ilmoitus mahdollisista päivityksistä ja ne voidaan asentaa suoraan sen kautta. Sivuvalikoissa voidaan muun muassa tehdä vianmäärityksiä ja vaihtaa tai luoda palautusavaimia.



Kuva 18 *Logic Apps -yhdyskäytävän yleisnäkymä*

Nyt kun yhdyskäytävä on asennettu isäntäkoneelle ja liitetty Azure käyttäjätiliin, jotta sitä voidaan käyttää muissa palveluissa, se täytyy lisätä resurssiksi Azuren resurssienhallintaan [71]. Tämä onnistuu portaalin kotinäkymässä, joko hakemalla suoraan hakukentästä tai painamalla resurssien lisäyspainiketta. Resurssin valinnan jälkeen siirrytään näkymään, jossa täytetään haluttavan yhdyskäytävän määritykset. Tilille liitetyt tilaukset ja yhdyskäytävän asennukset tunnistetaan automaattisesti ja ne tulevat pudotusvalikkoon valittavaksi.

Tiedon muuntaminen

Logic Apps -toteutuksella tiedon muuntamiseen ei ole graafista käyttöliittymää, jossa tiedot voitaisi yhdistellä vetämällä. Yksinkertaista JSON-tiedon muuntamista ja rakenteen

tarkastelua voidaan tehdä käyttäen sisäänrakennettuja Data Operations-luokasta löytyviä Parse JSON- tai Compose -elementtejä [72]. Näissä elementeissä muuntaminen tehdään antamalla JSON-malli, joko määrittelemällä se itse tai käyttämällä esimerkkitietoa sen muodostamiseen. Muuntamiseen voidaan myös hyödyntää Azure Functions -toteutuksia. Monimutkaisempaa muuntamista varten, käytetään toteutuksessa kuitenkin Liquid-mallipohjia [73]. Liquid on Shopify-yrityksen ylläpitämä mallikieli (engl. template language) [74]. Näissä malleissa JSON-tietoon täydennetään tulevat tiedot dynaamisesti, käyttäen muuttujia ja ohjausvirtoja, kuten jos – muuten ja silmukkarakenteita [73]. Mallille tulevan tiedon rakenne ja ominaisuudet vastaavat käytettävää JSON-tietoa, jota halutaan muuntaa. Pohjan luonnin jälkeen se täytyy lisätä resursseihin liittämällä se erikseen luotavaan integraatiotiliin.

Integraatioiden luonti

Yhdyskäytävän lisäyksen jälkeen on sisäverkkoon yhdistämiseen tarvittavien komponenttien määritykset valmiit ja lisätään integraatiotapaukseen liittyviä elementtejä. Aluksi resursseihin lisätään mukautettu liitin (Logic App Custom Connector). Tätä elementtiä tarvitaan yrityksen sisäisen rajapinnan kutsumiseen ja sen määritykset koostuvat kolmesta osasta. Yleisiin asetuksiin määritellään yhdistettävän rajapinnan tyyppi, käyttääkö liitin yhdyskäytäväyhteyttä ja rajapinnan isäntäkoneen pohjaosoite. Turvallisuusmäärityksiin voidaan tarvittaessa lisätä rajapinnassa käytettävä autentikointimenetelmä. Rajapinnan määrityksiin luodaan rajapinnasta löytyviä päätepisteitä ja niiden kutsuvaatimukset sekä palautusarvot. Liittimeen tehdyistä määrityksistä generoidaan automaattisesti Swagger-dokumentaatio. Määritykset voidaan tuoda aiemmasta toteutuksesta käyttäen olemassa olevia OpenAPI-määrityksiä JSON muodossa.

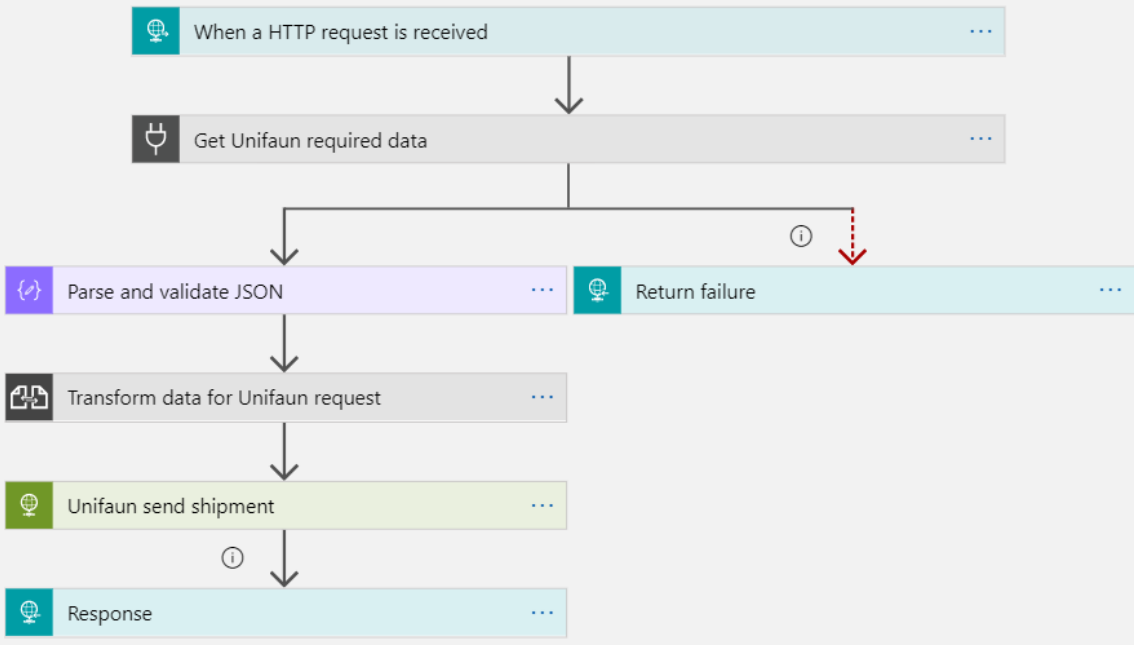
Integraatiototeutus aloitetaan luomalla resursseihin uusi Logic App -sovellus, joka tehdään Azuren aloitussivun kautta. Uutta toteutusta luodessa voidaan käyttää valmiiksi löytyviä pohjia tai luoda työnkulku tyhjästä projektista itse.

Prosessin käynnistämiseen käytetään aloituselementteinä erilaisia liipaisimia. Vaihtoehtoja on monia, mutta toteutuksella käytetään HTTP-liipaisinta. Tämä aktivoi integraation, kun sen luoma rajapinta vastaanottaa HTTP-kutsun. Määrityksiin annetaan ainoastaan vastaanotettavan hyötykuorman JSON-malli, jonka avulla alusta voi suorittaa tarkastuksia saapuvan tiedon rakenteelle. Rajapinta aukeaa Azuren pilvipalveluihin, eikä sille tarvitse määritellä erillisiä yhteysasetuksia.

Tapahtumakaavion mukaan edetään tiedonhankinta vaiheeseen ja lisätään aiemmin luotu mukautettava liitin osaksi työnkulkua. Liitintä lisätessä annetaan yhteyteen tarvittavat tunnukset ja valitaan yhdyskäytävä, johon yhdistetään. Kun yhteys on määritelty, annetaan tarvittavat parametrit ja kutsuotsikot.

Yrityksen sisäisestä rajapinnasta tulevan tiedon rakenne voidaan vahvistaa Parse JSON -komponentilla, käyttäen sille määriteltyä JSON-mallia. Toteutusta tehdessä huomattiin, että liittimellä tehdylle rajapintakutsulle täytyy tehdä erillinen virheiden hallinta, jotta yrityksen rajapinnan tuottamat virheviestit voidaan välittää kutsujalle. Tämä voidaan toteuttaa määrittämällä rinnakkainen suoritushaara edellisen vaiheen epäonnistuessa.

Haetun tiedon ollessa oikeassa muodossa, voidaan lisätä elementti, joka muuntaa sisään tulevan tiedon aiemmin luodun Liquid-pohjan mukaisesti. Muuntamisen lisäksi elementti vahvistaa rakenteen käyttäen määriteltyä JSON-mallia. Onnistuneen vahvistuksen jälkeen valmistellaan kutsu Unifaunin rajapintaan käyttäen HTTP-kutsu elementtiä. Viestiosaan sijoitetaan muunnettu tieto sekä määritellään tarvittavat otsikot ja autentikointiin tunnukset. Lopuksi rajapinnan tuottama vastaus lähetetään takaisin kutsujalle. Vastauselementtiin määritellään sen suorittuvan myös virhetilanteessa, jolloin saadaan välitettyä rajapinnan virheviestit käyttäjälle. Lopullinen integraatiokulku näyttää sen jälkeen kuvan 19 mukaiselta.



Kuva 19 *Lopullinen integraatioprosessi Logic App designer -näkymässä*

Integraatioiden julkaiseminen

Integraatiototeutusta ei tarvitse erikseen julkaista, vaan tallennuksen onnistuessa rajapinta on käytettävissä ja aktiivinen. Omien testien perusteella automaattisessa päivityksessä voi esiintyä haittana se, että ennen päivitystä oikeaan aikaan lähetetty kutsu saattaa katketa ja riippuen toteutuksesta aiheuttaa ei toivotun lopputuloksen.

Huomioita

Alustalla on verraten enemmän sen ulkopuolella luotavia osia, jolloin toteuttamisesta tulee työläämpää. Kuten tiedon muuntamiseen käytettävä Liquid-pohja tai sisäverkkoon mukautettava liitin, täytyy ne luoda erikseen omien valikkojen kautta. Esimerkiksi Liquid-pohjille ei ole erillistä näkymää niiden tekemiseen, vaan ne täytyy tehdä omalla koneella. Positiivisena asiana mukautettava muuntaminen lisää ohjelmoijan vapautta, sekä mahdollistaa jo muuntamisvaiheessa omaa logiikkaa. Käytännössä tämä kuitenkin eliminoi muunnosten luonnin muilta kuin kehittäjiltä. Syntaksi pohjille oli ennalta tuntematon, mutta Microsoftin dokumentaatiosta löytyy esimerkki, jossa on esitelty integraatiotapauksen muuntamisen kannalta tarvittavat esimerkit.

Lopulta tuntemattomaksi jääneestä syystä virheiden käsittely Logic App -toteutuksella kestää pitkään. Verraten onnistuneen kutsun ja vastauksen välillä kesti yleensä noin kaksi sekuntia, mutta mikäli integraatiokulussa tuli virhe mukautetulla liittimellä tehdyssä kutsussa, saattoi vastaus kestää välillä lähes kaksi minuuttia. Kauan kestäneestä kutsusta on esimerkki kuvassa 20.

Kuva 20 *Kauan kestänyt kutsu Logic App -rajapintaan Postmanissa*

Tämä esiintyi ainoastaan tämän elementin kutsuissa tapahtuneilla virheillä, eikä sisäänrakennetulla HTTP-kutsuelementillä esiintynyt lainkaan viivettä virhetilanteessa. Syyksi epäiltiin sitä, että liittimen toteutus on yhdyskäytäväyhteyden varassa, jolloin siihen otetaan yhteys tässä tapauksessa oman koneen kautta. Onnistuneessa kutsussa tätä viivettä ei kuitenkaan esiinny ja yhteys kulkee samaa reittiä, joten täysin vedenpitävä hypoteesi tämä ei ole.

5.2 Kehykset

Molemmille kehyksille projektit luotiin käyttäen Spring Initializr -projektinluontityökalua. Spring Initializr luo valittujen komponenttien pohjalta Spring Boot -aloitusprojektin. Spring Boot on ohjelmistokehys, joka pohjautuu Spring ohjelmistokehykseen. Se tarjoaa nopean tavan luoda projekteja ja tarjoaa pohjan ohjelmien suorittamiselle. Spring Boot pyrkii automaattisesti täydentämään tarvittavia komponentteja projektin luokkapolusta hyödyntäen valittua riippuvuudenhallinta työkalua. Se muun muassa määrittelee riippuvuuksien mukaan tarvittavan sisäänrakennetun sovelluspalvelimen, jonka avulla ohjelmat suoritetaan. Esimerkiksi Spring MVC-ohjelmille oletuksena määritetään Apache Tomcat -sovelluspalvelin [75],[76].

Spring Initializr -työkalua voidaan käyttää sille luodulla verkkosivulla tai sen sisältävillä ohjelmistoympäristöillä. Ohjelmointiympäristönä käytettiin JetBrainsin Intellij Idea Ultimatea, josta työkalu löytyy sisäänrakennettuna. Spring Initializr:lla voidaan luoda Maven tai Gradle riippuvuuksienhallintaan perustuvia projekteja, joille voidaan ohjelmointikielenä käyttää Javan sijasta myös Kotlinia tai Groovya. Luonnin yhteydessä voidaan lisätä myös muita ulkopuolisia riippuvuuksia, joita tarvitsee ja ne lisätään oletuskirjastojen oheen riippuvuudenhallinnan käyttämään määrittelytiedostoon [75]. Molempiin toteutuksiin valittiin kieleksi Java, sen versioksi JDK 11 ja projektin tyypiksi Maven. Lisäksi ulkopuolisena riippuvuutena lisätään Lombok-kirjasto, jonka toiminnasta myöhemmin lisää. Työkalu luo automaattisesti Mavenin käyttämän pom.xml määrittelytiedoston, kansiorakenteen sekä ohjelman 1 mukaisen Spring Boot -pääohjelman, jonka jälkeen ohjelma on heti suoritusvalmis.

```
package fi.oscar.camel.springboot.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

Ohjelma 1. *Spring Initializr-työkalun luoma pääohjelma Camel-toteutukselle.*

Camel-integraatiototeutuksen pohjaksi valittiin myös Spring Boot-projekti, sillä sen avulla voidaan keskittyä tutkimuksen kannalta olennaisemmin integraatiotapauksen toteuttamiseen. Projektien luonnin jälkeen aloitetaan varsinaiset toteutukset valituille integraatiokehyksille.

Molemmat integraatiokehykset tarjoavat viestipohjaisia integraatioita ja toteuttavat teoriaosuudessa mainittuja integraatiomalleja. Integraatiototeutusta suunnitellessa voidaan hyödyntää kuvan 1 mukaista kulkua määrittämään tarvittavia komponentteja. Varsinkin molemissa Spring-toteutuksissa vastaavat malleissa käytetyt nimeämiset hyvin kehyksen tarjolla olevia elementtejä.

Tiedon muuntaminen tehdään molemmilla kehyksillä käyttäen Java-luokkia, jotka vastaavat rakenteeltaan tarvittavia JSON-tietoja. Varsinaiset muunnokset luokkien välillä suoritetaan luokkametodin tai luokan rakentajan avulla. Luokkien ominaisuuksien saanti- ja asetusmetodit sekä luokan rakentaja määritetään käyttäen Lombok-kirjastoa. Lombokin määritysten tarkoituksena on vähentää kirjoitettavaa koodia luomalla yksinkertaiset ja usein toistuvat operaatiot automaattisesti ohjelman käännösvaiheessa. Esimerkkinä ohjelmassa 2 näkyy luokalle lisätyt Lombok-määritykset annotaatioilla. Data-määritys sisältää ominaisuuksien saanti- ja asetusmetodien lisäksi toString-, equals- ja hashCode-metodit sekä parametrirakentajan. Data-määrityksen tuoma parametrirakentaja luodaan luokalle final avainsanalla merkityistä ominaisuuksista. Koska rakentajaan haluttiin kaikki ominaisuudet, lisätään AllArgsConstructor -määritys, joka luo rakentajan kaikkien ominaisuuksien mukaan [77].

```
import lombok.Data;
import lombok.AllArgsConstructor;

@Data
@AllArgsConstructor
class PrintConfig {
    private String target1Media;
    private String target1Type;
}
```

Ohjelma 2. *Esimerkki tietoluokasta Lombok-määrityksillä*

Toteutuksissa JSON-tiedon muuntamiseksi käytetään siihen tarkoitettuja kirjastoja. Yleinen ja toteutuksien kohdalla oletuksena valikoitunut kirjasto tähän tarkoitukseen on FasterXML:n Jackson, joka osaa muuntaa sille annettavan tavallisen Java-objekti (POJO, Plain Old Java Object) JSON-objektiksi tai päinvastoin [79],[80]. Muita vaihtoehtoja vastaavilla ominaisuuksilla on esimerkiksi Googlen Gson ja Apache Johnson [78]. On myös mahdollista luoda omia JSON-muuntaja toteutuksia laajentamalla jotakin olemassa olevaa kirjastoa tai tehdä se kokonaan itse.

5.2.1 Spring Integration

Spring-integraatiokehyksellä on toteutuksiin useampia vaihtoehtoja. Yleisin toteutustapa on XML-pohjainen määritys, joka noudattaa XSD-pohjaista nimiavaruutta [81]. Toinen usein käytetty vaihtoehto on Java DSL, jossa integraatiokulkuja luodaan määritysluokan metodeissa käyttäen annotaatioita ja syntaksia, jossa viestirajapintakomponenttien nimet vastaavat suurilta osin XML-määrityksissä käytettäviä nimeämisiä [82]. Näiden kahden lisäksi integraatiototeutuksia on mahdollista luoda tekemällä itse toteutukset käytettävistä viestirajapinnoista. Tämä vaihtoehto vaatii eniten määrityksiä, mutta näihin voidaan hyödyntää kehyksen tarjoamia annotaatioita. Kaikki esitellyt vaihtoehdot ovat keskenään yhteensopivia, eli toteutuksessa voidaan sekoittaa erilaisia tapoja tarpeen mukaan [81].

Tutkimuksen yhteydessä tehtiin vertailun vuoksi kaksi toteutusta, jotka edustavat määritystavoiltaan ääripäitä. Ensimmäinen niistä tehtiin XML-määrityksillä, jossa viestien muokkaamiseen käytetään Java-luokkien metodeja määrittämällä ne palvelunaktivointirajapinnaksi XML-määrityksiin. Toinen toteutus tehtiin käyttäen kehyksen annotaatioita ja rajapintaluokkia suoraan luomalla niistä itse toteutukset. Annotaatioilla varustetut rajapinnat ja niiden riippuvuudet tunnistetaan, sekä niistä luodaan pääohjelman käynnistyksen yhteydessä ilmentymät automaattisesti [83].

XML-toteutus on kahdesta toteutusvaihtoehdosta virtaviivaisempi ja vähäsanaisempi, jossa määritykset tehdään XML-tiedostoon, joka tunnistetaan Spring Bootilla luodussa pääohjelmassa käyttäen ImportResource-annotaatiota. Määritystiedostoon luodaan aluksi käytettävät kanavat, jotka ovat tässä tapauksessa suorakanavia yksinkertaisuuden vuoksi. Lisäksi kutsuttaviin rajapintoihin käytetään yhdyskäytävä komponentteja, joka sisään tulevan viestin lisäksi lähettää määritellystä rajapinnasta saadun vastausviestin määriteltyyn kanavaan. Toinen vaihtoehto rajapinnoille on adapter, eli sovitinkomponentti. Sovitin eroaa yhdyskäytävästä siten, että se vastaanottaa viestejä sisään tulo kanavasta, mutta se ei palauta rajapinnan tuottamaa vastausta eteenpäin ulostulo kanavalle. Yhdyskäytäviä on kahdenlaisia; saapuvaan ja ulosmenevään liikenteeseen. Määritykset ovat muuten samat, mutta saapuvan liikenteen yhdyskäytävään asetetaan polku, josta yhdyskäytävä voidaan aktivoida. Toisin sanoen se avaa rajapinnan integraatioprosessin aloitukseen. Ulosmenevän liikenteen yhdyskäytävää voidaan käyttää ulkopuolisten rajapintojen kutsumiseen.

Aluksi saapuvan liikenteen yhdyskäytävään (inbound gateway) tulevan kutsun perusteella otsikossa siirrettävä rahtikirjanumero lähetetään viestin mukana määritellylle ulostulokanavalle. Service Activator-komponentti kuuntelee sille asetettua sisääntulo kanavaa ja suorittaa määritellyn metodin, jossa saapuvaa viestiä voidaan muokata ja lähettää

edelleen ulostulokanavaan. Ensimmäisessä metodissa määritellään viestiin rajapinnassa käytettävät tunnistautumisotsikot, jonka jälkeen tiedot haetaan yrityksen rajapinnasta viestissä kulkevan rahtikirjanumeron avulla. Kutsu yrityksen rajapintaan tehdään Apachen HTTP-asiakaskirjaston avulla siihen luodussa palvelukutsussa. Kutsu olisi voitu tehdä myös ulostulevan liikenteen yhdyskäytävällä, mutta koska otsikoita haluttiin muokata suoraan tiedon haun yhteydessä, päädyttiin tähän ratkaisuun. Haetut tiedot ja muokatut otsikot lähetetään viestin mukana muuntamisen aloittavaan kanavaan.

Tietojen muuntamiseksi JSON-muotoon käytetään json-to-object-transformer-muuntajaa. Oletuksena muuntaja käyttää FasterXML:n Jackson kirjastoa toteutuksen pohjana, mutta tätä on mahdollista mukauttaa käyttämään muita kirjastoja tai omaa toteutusta. Muuntajalle määritellään sisääntulokanava, josta saapuu tietoa viestin hyötykuormassa JSON-muodossa. Muuntaja käsittelee tiedon ja muuntaa sen sille määritellyksi Java-luokaksi ja lähettää sen viestin mukana ulostulo kanavalle. Tämä kanava aktivoi seuraavan palvelun, jossa valmistellaan kutsu Unifaunin rajapintaan. Ulospäin suuntavan liikenteen yhdyskäytävälle johtavalle kanavalle lähetettävään viestiin lisätään hyötykuormaksi luokka, joka on yrityksen rajapinnasta haetun tiedon perusteella Unifaun rajapinnan vaativaan muotoon muunnettu. Muuntamiseen käytettiin luokkarakentajaa, johon tulee parametrinä luokka, joka vastaa yrityksen rajapinnan tietoja.

Muuntamisen jälkeen aktivoituu yhdyskäytävä (outbound gateway), joka suorittaa kutsun ulospäin Unifaunin rajapintaan saapuvan viestin otsikoiden ja hyötykuorman avulla. Rajapinnasta saatu vastausviesti voidaan palauttaa suoraan sisään tulevan liikenteen yhdyskäytävälle määriteltyyn ulostulokanavaan. Tiedosta haluttiin tulostaa lokiin tietoja, joten se kierrätettiin vielä ylimääräisen kanavan kautta, joka aktivoi palvelun, jossa tietoja tulostetaan ja hyötykuorma lähetetään eteenpäin kanavalle, josta se palautetaan kutsujalle yhdyskäytävän kautta.

Toinen Spring Integration toteutus on vastaavilla komponenteilla luotu kuin XML-toteutus, mutta niille tehtävät määritykset on pääosin tehty itse. Annotaatioilla automaattisesti määrittyviä komponentteja ovat palvelunaktivointirajapintoihin asetetut metodit ja kanavat. Yhdyskäytävät määritettiin manuaalisesti käyttäen tarjolla olevia rajapintaluokkia. Sisään saapuvan liikenteen yhdyskäytävälle käytettiin HttpRequestHandlingMessagingGateway-luokkaa ja ulospäin suuntautuvan liikenteen yhdyskäytävälle käytettiin HttpRequestExecutingMessageHandler-luokkaa. Unifaunin rajapinnan yhdyskäytävän tekemä kutsu aktivoidaan palvelunaktivointirajapinnaksi määritellyllä metodilla, kun sille määritettyyn kanavaan saapuu viesti. Kanavia on toteutuksella sama määrä kuin aiemmassa toteutuksessa ja kulku on muutenkin vastaava kuin edellisellä. Yhdyskäytävät on

myös mahdollista toteuttaa käyttäen MessagingGateway ja Gateway annotaatioita, mutta toteutukseen haluttiin ottaa vertailun vuoksi myös itse määriteltyjä komponentteja.

Toteutusten käynnistyessä ovat rajapinnat kutsuttavissa niille määritetyissä päätepisteissä. Päätepisteet on määritelty käyttämään HTTP POST-metodia ja kutsuminen tapahtuu testatessa Postman-työkalun avulla. Kehyksellä tehtyihin toteutuksiin on mahdollista luoda mukautettavaa virheidenhallintaa, jos tälle esiintyy tarvetta. Tutkimuksen tapauksen osalta virheidenhallintaan ei tarvinnut tehdä muutoksia ja molemmissa toteutuksissa rajapinnoista tulleet virheet välittyvät vastauksen mukana.

5.2.2 Apache Camel

Camel sisältää muutamia vaihtoehtoja määritysten tekemiseen, kuten työssäkin käytetty REST DSL ja Java DSL [84]. REST DSL -määrityksillä voidaan helposti määritellä rajapinta ja asettaa REST-operaatioiden yleisimpiä asetuksia, kuten rajapinnan päätepisteen hyväksymät HTTP-metodit (GET, POST jne). Tätä varten projektiin pitää tuoda camel-rest-dsl Maven riippuvuus [85].

Integraatiot toteutetaan luomalla reittejä, joiden kautta ohjelma etenee. Näihin reitteihin on kirjoitushetkellä kaksi vaihtoehtoa; XML-pohjainen määritys tai luokkaan tehtävät Java DSL määritykset. Vaihtoehdoista valitaan jälkimmäinen, sillä se sopii luonnostaan osaksi Spring Boot -ohjelmistokehystä, jolla projekti on luotu [84].

Jotta reittejä voidaan luoda, täytyy reiteistä vastaavan luokan laajentaa RouteBuilder abstraktiluokkaa ja ylikirjoittamalla kyseisen luokan configure-metodia, jossa määritykset tehdään [84]. Aluksi lisätään REST-määritykset ja avataan rajapinta haluttuun polkuun, johon voidaan myös määritellä osoite ja portti, mutta Spring Bootin ansiosta näihin tulee oletuksena pääohjelmalle tulevat määritykset [85]. Spring määrityksien mukaisesti reittiluokalle lisätään @Component-annotaatio, jolloin luokka tunnistetaan automaattisesti osaksi pääohjelmaa. Pääohjelmaan ei tarvitse lisätä reittiluokalle alustuksia tai edes luoda siitä erikseen ilmentymää [86].

Reitin määritys alkaa from-avainsanalla ja sen sisälle parametrinä tulee mistä kanavasta viesti tulee. Kanavana voi olla esimerkiksi suorakanava tai tiedoston polku. Vastaanottavan reitin oheen voidaan tehdä tarvittavia muokkauksia saataviin viesteihin, kuten otsikoiden asettamista. Kaikki haluttavat operaatiot voidaan toteuttaa yhdelle reitille, mutta uudelleen käytön vuoksi jokainen operaatio tehdään omaksi reitikseen [84]. REST DSL -määritysten mukaisesti HTTP POST -kutsun saadessaan siirrytään reitille, jossa hae-

taan yrityksen sisäisestä rajapinnasta tiedot muuntamista varten. Ennen rajapintaa kutsumista asetetaan tunnistautumisotsikko ja JSON-tiedon otsikot, sen jälkeen kutsu rajapintaan tehdään toD-reittimäärityksellä. Onnistuneen kutsun jälkeen hyötykuorma siirretään to-reittimäärityksellä kanavalle, joka vastaa tiedon muuntamisesta Unifaunin vaatimaan muotoon. Reittimääritysten to ja toD erot ovat siinä, että jälkimmäisessä rajapinta rakennetaan dynaamisesti saatavien parametrien mukaan. Kun taas to-määrityksellä annetaan staattinen rajapinta tai kanava, joka ei muutu parametrien mukaan [87].

Heikkoutena dynaamisella rajapinnan rakentamisella on se, että jokaisesta eri reittiparametristä luodaan oma ilmentymä, jolloin resurssien käyttö kasvaa. Tätä voidaan hillitä määrittämällä sallittujen rajapintailmentymien määrää tai jos mahdollista käyttää staattista polkua ja määrittää parametrit viestin otsikkoina, eli käyttää to-määritystä reiteillä. Jälkimmäistä vaihtoehtoa ei toimi tutkimuksen tilanteessa, koska päätepisteelle määritettiin bridgeEndpoint-ominaisuus päälle, joka ei huomioi tiettyjä otsikoita ja käyttää määritellyn rajapinnan osoitetta Camel-ilmentymän sijaan [87]. Mikäli rahtikirjanumero, jolla tiedot haetaan, otettaisiin yrityksen rajapinnassa vastaan kutsun hyötykuormassa, reittiparametrit voitaisiin jättää pois. Tällöin staattisen rajapinnan käyttö olisi ollut mahdollista.

Muuntamisesta vastaava reitti vastaanottaa viestin, jonka hyötykuorman se muuttaa ensin JSON-muodosta sitä vastaavaksi Java-luokaksi. Muuntamiskirjastona käytetään aiemmin mainittua Jacksonia [88]. Yrityksen rajapinnan tiedosta muodostettu luokka voidaan muuttaa reittimäärityksissä käyttäen bean-komentoa. Tällä komennolla voidaan liittää Java-luokat viestijärjestelmään ja käyttää määritetyn luokan metodeja [89]. Tässä tapauksessa Unifaun-tietoon käytettävän luokan transform-metodia, joka parametrinä ottaa vastaan yrityksen rajapinnasta saadun tietoluokan ja muuntaa sen Unifaunille sopivaksi. Lopuksi metodin vastaus muunnetaan jälleen JSON-muotoon käyttäen samaa Jackson-kirjastoa. Muuntamisprosessin koodi on nähtävissä ohjelmassa 3. Muunnettu tieto lähetetään viestin hyötykuormassa kanavaan, josta se siirtyy Unifaunin lähetyksestä vastaavalle reitille.

```
from("direct:transform")
    // Take cloud data from body
    .unmarshal().json(JsonLibrary.Jackson, CloudData.class)
    // Transform cloud data using class transform method
    .bean(UnifaunData.class, "transform")
    // Transform result back to json
    .marshal().json(JsonLibrary.Jackson, UnifaunData.class)
    .to("direct:send");
```

Ohjelma 3. *Camel-reittimääritys muuntamiselle*

Lopullinen reitti Unifaun-rajapinnan kutsumiseen on vastaava kuin yrityksen rajapintaan. Ensin määritetään tarvittavat tunnistautumis- ja JSON-tiedon otsikot, jonka jälkeen kutsu rakennetaan samalla toD-määrityksellä Unifaunin rajapinnan osoitteeseen. Kanavalta saadun viestin hyötykuorma siirtyy automaattisesti HTTP-kutsuun mukaan ja onnistuneen kutsun jälkeen lopputulosta ei siirretä enää seuraavalle kanavalle, vaan määritykset palauttavat automaattisesti vastauksen integraatiorajapinnan kutsujalle.

Määritykset ovat valmiit ja integraatioreittejä voidaan testata tekemällä HTTP POST -kutsu määriteltyyn REST-päätepisteeseen. Kutsut tehdään jälleen Postmanilla ja lopputuloksena onnistuneesta kutsusta saadaan HTTP-tilakoodi 201 Created ja linkit lähetystulosteisiin. Virheet toteutus hallitsee oletuksena hyvin ja se palauttaa rajapinnasta virhettä vastaavan tilakoodi ja virheviestin.

5.3 Yhteenveto

Luvussa käytiin läpi integraatiotapauksen siirtämisen vaiheita erilaisilla alustoilla ja kehyksillä. Käyttöohjeen sijasta vaiheiden etenemisen yhteydessä kirjattiin havaintoja, joita toteutusta tehdessä huomattiin. Havaintojen ja etukäteen asetettujen vaatimusten perusteella voidaan suorittaa vertailua näiden välillä. Tulokset ja vertailu tehdään seuraavassa luvussa.

Teoria oli siirtämisen aikana vahvassa roolissa varsinkin kehyksillä toteuttaessa, sillä niiden komponentit perustuivat siellä esiteltyihin malleihin. Komponenttien lisäksi kehyksillä on mahdollisuus vaikuttaa integraation ratkaisumalliin, sillä ne tehdään itse. Alustoilla varsinaisesti teoriaa ei tarvittu toteuttaessa, eikä ratkaisumallia ole mahdollista muuttaa, koska ne ovat valmiiksi luotuja. Tutkimukseen valituilla alustoilla ratkaisumalli on lähimpänä palveluväylää, sillä sen tarjoamaan runkoon tuodaan palveluita ja muita resursseja.

Kuten teoriassa esitettiin, ovat käytettävät tavat ja komponentit hyvin tapauskohtaisia. Käytännössä tämä esiintyi viestijärjestelmän suunnittelussa, jossa integraation kululle on useita vaihtoehtoja ja niistä paras vaihtoehto on tarpeesta kiinni. Alustoillakin on käytännössä mahdollista hyödyntää kaikkia integraatiotapoja, vaikka ratkaisumalliin ei voida vaikuttaa. Kehyksillä valinnanvaraa oli luonnollisesti vielä enemmän, jolloin päästään vaikuttamaan toteutuksen jokaiseen vaiheeseen. Oli se sitten viestin reitityskomponentti tai kanava, jota pitkin viesti kuljetetaan eteenpäin.

6. TULOKSET

6.1 Yleistä tuloksista

Alustojen yleistä vertailua varten koostettiin erilaisia ominaisuuksia, jotka koskevat alustan yleisiä toiminnallisuuksia. Arviointi on tehty sen perusteella, kuinka hyvin alustat pärjäsivät eri ominaisuuksien osalta ja tulokset löytyvät taulukosta 2. Vertailtavina laatuominaisuuksina olivat virheidenhallinta, integraatiototeutusten luonnin helppous, tiedon muuntamisen yksinkertaisuus, luotujen toteutusten julkaisemisen sujuvuus ja toteutusten suorituskyky ajallisesti mitattuna. Alustavasti vertailtavat ominaisuudet koskevat suorituskykyä lukuun ottamatta alustoja, mutta kehyksillä luoduista toteutuksista esitellään muutamia huomioita. Suorituskykyä verrataan erillisenä osana lopuksi, sillä siinä vertaillaan alustojen ja kehyksien lisäksi myös aiempaa toteutusta sekä suoraa kutsua Unifaun-rajapintaan.

Taulukko 2 *Alustojen tarkasteltavia ominaisuuksia*

Tarkasteltava ominaisuus	Boomi	Anypoint	Logic Apps
Virheidenhallinta	-	-	-
Integraatioiden luonti	+	+	-
Tiedon muuntaminen	+	+	-
Integraatioiden julkaisu	+	-	+

Alustoilla tehdyissä toteutuksissa huomattiin, että virheidenhallinta olisi pitänyt ottaa huomioon heti aluksi. Kun kutsuja tehtiin Unifaunin suuntaan ja sieltä saatiin virheviestinä esimerkiksi 400 Bad Request, palautettiin Unifaunin rajapinnasta vastauksessa virhetiedot, mutta virheiden hallinnan puuttuessa alustat eivät oletuksena välittäneen virheviestiä takaisin vaan palauttivat alustan luoman oletusvirheviestin, josta ongelmaa ei saatu esille. Boomi-alustalla rajapintaan tehty kutsu palautti onnistuneen 200 OK vastauksen, vaikka prosessin sisällä tapahtui virhe. Huomioiden perusteella luotiin liitteen 2 mukainen paranneltu prosessikaavio, jossa virheidenhallinta otetaan huomioon. Paranneltua kaaviota voidaan käyttää mallina myös muihin vastaavanlaisiin integraatiotapauksiin, muuttamalla ainoastaan tapahtumien selitteitä.

Alustat ovat integraatiokulkujen luonnin osalta hyvin samankaltaisia. Pääasiallinen työ tapahtuu keskitetysti yhden graafisen käyttöliittymän kautta, mutta alustakohtaisesti tässä oli myös vaihtelua. Anypoint- ja Boomi-alustalla tarvittavia resursseja luodaan samasta näkymästä ja elementit löytyvät sivuvalikosta valittavaksi. Azuren Logic Apps -alustalla oli paljon luotavia resursseja, jotka tehtiin luontinäkymän ulkopuolella omien va-

likoiden kautta. Tämä aiheuttaa muihin verrattuna ylimääräistä työtä. Kun kaikki tarvittavat resurssit ja komponentit on luotu, saadaan integraatioprosessi viimeisteltyä tähän tarkoitetussa Logic App Designer -näkymässä.

Tiedon muuntaminen tapahtuu Boomi- ja Anypoint-alustalla muuntamiselementin graafisella näkymällä. Logic Apps -toteutuksella muuntaminen tapahtui itse luotavan muuntamispohjan avulla, jossa käytetään Liquid -mallikielen merkintätyylejä. Liquid-mallipohjilla saavutetaan korkea mukautettavuus, sillä muuntamisen ohessa voidaan käyttää ohjausrakenteita, kuten ehtoja ja silmukoita. Verrattuna graafisesti tapahtuvaan muuntamiseen tämä tuo kuitenkin ylimääräisen kerroksen monimutkaisuutta. Anypoint -alustalla muuntamista voidaan mukauttaa hieman, esimerkiksi tekemällä tarkastuksia hyötykuorman tiedoille. Mukauttaminen tapahtuu tekemällä muutoksia suoraan generoituun Data Weave -skriptiin. Tämä kuitenkin saattaa sekoittaa graafisen näkymän, jolloin sitä ei voida enää käyttää yhtä luotettavasti. Boomin muuntamisprosessissa on tiedon yhdistämisen lisäksi mahdollisuus luoda mukautettavaa logiikkaa suoraan elementtiin tekemällä siihen erillisiä skriptilohkoja.

Integraatioprosessien julkaiseminen oli kaikilla alustoilla virtaviivaista ja tarvittavien paikallisesti toimivien osien asentaminen tarjottujen ohjeiden avulla yksinkertaista. Logic App -alustalla oli muista poikkeavasti automaattinen julkaisu, joka tarkoittaa sitä, että aina tallennuksen onnistuessa integraatiototeutuksen aktivoiva rajapinta on käytettävissä ilman erillistä julkaisuprosessia. Boomilla tehty toteutus pystytään helposti julkaisemaan painikkeella samasta näkymästä, kuin missä prosesseja luodaan. Anypoint-alustalla julkaisu täytyy tehdä hallintapaneelista löytyvän oman käyttöliittymän kautta, asentamalla integraatiototeutus JAR-pakettimuodossa. Tämän lisäksi toteutus on mahdollista julkaista käyttämällä komentoliittymäohjelmaa tai Maven-liitännäistä. Muihin alustoihin verrattuna julkaisu Anypoint-alustalla on monipuolisista vaihtoehdoista huolimatta monimutkaisempaa, joka arvioinnissa vähentää sen tulosta.

Kehyksillä tehdyt toteutukset vaativat alustoja enemmän taustatyötä, sillä niiden käyttäminen pohjautuu vahvasti teoriaosuudessa esiteltyihin malleihin. Spring Integrationilla käytettävät komponentit on nimetty vastaamaan viestipohjaisia integraatiomalleja, jolloin niiden toimintatapoja ymmärrettiin jo etukäteen teoriapohjalta. Camelilla kyseinen integraatiotapaus saatiin toteutettua hyvin pienillä määrityksillä, mutta valitulla täsmäkielellä käytettyjen elementtien termit eivät vastanneet yhtä hyvin integraatiomalleja. Tällöin jouduttiin toimintatapoja etsimään enemmän

Kehysten vaihtoehtoisilla täsmäkielillä toteuttaessa haasteeksi ilmaantui dokumentaatioiden yksipuolisuus. Spring Integrationin dokumentaatioissa oli eniten esimerkkejä XML-

määrityksille ja muilla tavoilla esimerkkien löytäminen oli haastavampaa. Camel-komponenttien dokumentaatioissa suosittiin Java-täsmäkieltä, mutta XML-määrityksille löytyi useassa tapauksessa myös esimerkit. Kun toteutuksen sai valmiiksi jollain täsmäkielellä, oli muilla tavoilla toteuttaminen helpompaa. Spring Bootin hyödyntäminen molemmilla kehyksillä nopeutti ohjelmien luontia merkittävästi, koska aikaa ei kulunut esimerkiksi sovelluspalvelimen määrityksiin ja voitiin keskittyä integraatiologiikan toteuttamiseen.

Alustoilla ja kehyksillä luotujen toteutuksien suoritusnopeudessa oli keskenään paljonkin vaihtelevuutta. Jotta eroavaisuuksista saadaan parempi kuva, tehtiin kymmenen HTTP-kutsua jokaiseen rajapintaan Postman API-testaustyökalulla. Vertailun vuoksi kutsut tehtiin uusien toteutusten lisäksi myös vanhan toteutuksen ja suoraan Unifaunin rajapintaan. Tulokset koostettiin taulukkoon 3, johon tulosjoukoille laskettiin keskiarvo ja mediaani.

Taulukko 3 *Suorituskykyvertailut ja niistä kerätyt tilastoarvot*

Kutsu	Mule	Boomi Web Service kuuntelija	Boomi API endpoint	Azure	Spring DSL	Spring XML	Camel	Suora kutsu	Aiempi toteutus
1 (s)	0.924	0.256	0.624	1.441	0.449	0.414	1.075	0.213	1.172
2 (s)	0.893	0.251	0.548	1.983	0.517	0.391	1.344	0.170	1.350
3 (s)	1.109	0.268	0.588	1.856	0.435	0.387	0.332	0.149	1.248
4 (s)	0.349	0.245	0.504	1.251	0.475	0.407	0.411	0.142	0.921
5 (s)	0.315	0.223	0.510	0.897	0.440	0.354	0.406	0.155	0.917
6 (s)	0.361	0.263	0.498	1.522	0.680	0.318	0.368	0.154	0.874
7 (s)	0.340	0.222	0.564	1.443	0.415	0.397	0.419	0.149	1.302
8 (s)	0.314	0.306	0.523	2.17	0.357	0.427	0.409	0.132	1.029
9 (s)	0.329	0.242	0.531	1.201	0.352	0.405	0.404	0.123	0.825
10 (s)	0.381	0.332	0.553	0.882	0.386	0.358	0.419	0.130	0.902
Keskiarvo (s)	0.532	0.261	0.544	1.465	0.451	0.386	0.559	0.152	1.054
Mediaani (s)	0.355	0.254	0.540	1.442	0.438	0.394	0.410	0.149	0.975

Mediaani laskettiin siitä syystä, että voidaan eliminoida satunnaisesti tapahtuvia viiveitä, jolloin tulosjoukkoon saattaa tulla poikkeuksellisen suuria arvoja, jotka vääristävät keskiarvoa. Vertaillessa keskiarvoa ja mediaania ei tällä otannalla tullut kovinkaan suuria eroja, mutta poikkeuksiakin löytyy. Mule- ja Camel-toteutuksilla muutamassa ensimmäisissä kutsussa kestot ovat huomattavasti suurempia kuin lopuissa. Vaikka tarkkaa syytä tapahtuneelle ei löydetty, epäiltiin sen liittyvän juuri käynnistetyn ilmentymän taustaprosessien käynnistymisestä, sillä alkupään kutsut on tehty juuri käynnistetyille toteutuksille.

Heikoin suorituskyky molempien tilastoarvojen valossa oli Azuren Logic Apps -toteutus noin 1.5 sekunnilla ja paras Boomin Web Service -kuuntelijalla tehty toteutus noin 0.25 sekunnilla. Vertailuarvoiksi vanhan toteutuksen kutsun keston keskiarvo ja mediaani oli noin 1 sekunti. Unifaunin rajapintaan tehdyillä kutsuilla molemmat tilastoarvot olivat noin

0.15 sekuntia. Kaikilla muilla, paitsi Logic Apps -toteutuksella, saadaan yli puolet nopeampi suoritusaika verrattuna vanhaan toteutukseen. Boomilla tehtyyn API-toteutukseen tulee suoritusaikaa noin puolet lisää verrattuna kuuntelijalla aktivoitavaan prosessitoteutukseen. Viiveen epäiltiin johtuvan siitä, että API-toteutus kutsuu prosessia, jossa aloituselementtinä on mainittu Web Service -kuuntelija. Lopulta kuuntelijaa ei rajapintatoteutuksen tilanteessa tarvita, sillä prosessi aktivoidaan REST-rajapinnan kautta ja kuuntelija tuo oletettavasti turhaan viivettä suoritukseen.

Saatuihin tuloksiin on huomioitava kuitenkin se, että uudet toteutukset ovat demototeutuksia ja niistä voi olla jäänyt jotain käytännön asioita huomioimatta. Lisäksi nopeuteen voi vaikuttaa positiivisesti, että tiedon haku ja koostaminen suoritetaan integraatiototeutuksen sijasta yrityksen sisäisessä rajapinnassa. Tästä huolimatta tulokset ovat lupaavia ja niistä voidaan tehdä johtopäätös, että lähes kaikki uusista vaihtoehdoista toisivat suorituskykyyn merkittäviä parannuksia.

6.2 Vaatimusten toteutuminen

Pääasiassa vaatimukset toteutuivat kohtalaisen hyvin alustasta riippuen. Vaatimusten toteutumisesta luotiin taulukko 4, jossa koostettuna vaatimukset ja niiden toteutuminen. Taulukosta löytyy myös kommentit niille kohdille, joissa vaatimus on toteutunut osittain tai ei lainkaan.

Taulukko 4 *Alustojen vaatimuksien toteutuminen*

Vaatimus	Boomi	Huomiot	Anypoint	Huomiot	Logic Apps	Huomiot
OAuth	✓		✓		✓	
Yhtenäinen tapa tehdä	✓		✓		Osittain	Elementtejä luodaan monessa paikassa erilaisin määrityksin
Perinteiset integraatiome-netelmät	✓		✓		Osittain	Mahdollista toteuttaa, mutta ei sisällä valmista toiminnallisuutta
Dokumentaati-ot	Osittain	Dokumentaatiot saadaan luotua API-toteutuksen kautta, mutta ei prosesseille	Osittain	Dokumentaatiot pitää luoda erillisen APISpec-projektin kautta	Osittain	Itseluoduista rajapintaliittimistä saadaan OpenAPI-dokumentaatio
Yhdistäminen pilvipalveluihin	✓		✓		✓	
Integraatioiden luonti ilman teknistä taustaa	Osittain	Jos yhteysasetukset on tehty valmiiksi pohjiksi	×	Pilvipohjaisia julkaisuja voidaan, mutta vaatii yhteysasetuspohjat. Sisäverkossa toimivia integraatioita ei voida tehdä selainpohjaisella editorilla	×	Määritykset ja elementtien luonnit usein hyvin teknisiä

Vaatimuksissa mainituista OAuth-valtuutusvirroista löytyy kaikilta alustoilta elementtejä tai määrityksiä niiden hyödyntämiseen. Koska tutkimuksessa keskitytään enemmän integraatiotapauksien siirtämiseen alustoille, ei käytännön toteutuksissa hyödynnetty näitä valtuutusvirtoja, mutta tutkittiin mahdollisuuksia niiden lisäämiseksi integraatioprosesseihin. Azurella valtuutus tapahtuu ensisijaisesti Active Directoryn avulla. Tämä ei ole tutkimuksen yritykselle ongelma, sillä heillä on käytössään Windows-ympäristö ja kaikkien työntekijöiden käyttäjiä hallitaan Active Directoryn kautta. Mukautettaviin liittimiin voidaan OAuth-määrityksiä tehdä ulkopuolisen tarjoajan kautta, kuten Google ja GitHub tai määrittää oma. Itse alustalle kirjautumiseen ja käyttäjäoikeuksiin voidaan hyödyntää yrityksen olemassa olevia Microsoft tunnuksia.

Yhtenäinen tapa tehdä toteutuksia saavutetaan alustoilla graafisen käyttöliittymän ja valmiiden elementtien kautta. Tässä on alustakohtaisesti hieman vaihtelua varsinkin vaatimuksissa mainitun helppokäyttöisyyden osalta, joka vaikuttaa kokemukseen merkittävästi. Azurella integraatiototeutuksella käytetyt komponentit tehdään omina resursseina niille tarkoitetuissa näkymissä. Alustalla on paljon luotavia komponentteja ja tämä aiheuttaa aluksi sekavan käyttökokemuksen. Muuntamiseen täytyi lisäksi tehdä itse Liquid-

mallinnekielellä pohja, jolla tiedot voidaan muuntaa eri rakenteiseksi ja tämä toi työskentelyyn lisää monimutkaisuutta. Pohjan lisäksi resursseihin täytyi luoda integraatiotunnukset, joihin Liquid-elementtien määrityksiä voidaan liittää. Tunnukset vaaditaan, ennen kuin näihin liittyviä resursseja voidaan käyttää Logic App -toteutuksella. Anypoint ja Boomi olivat alustoista lähimpänä toisiaan toteutusten luonnin näkökulmasta. Kunhan alustojen tarvittavat paikallisesti suoritettavat komponentit oli asennettu, tapahtui pääsiallinen työskentely yhdessä näkymässä. Boomi oli lopulta kaikista alustoista helpoin käyttää, sillä myös integraation julkaisu tapahtui tästä samasta näkymästä. Anypoint-alustalla ainoastaan julkaisu täytyy tehdä eri näkymästä tai käyttää toteutuksen yhteydessä mainittuja tarkoitukseen sopivia työkaluja. Kehyksillä sisäänrakennetut kirjastot ja täsmäkielet mahdollistavat yhteneväistä logiikkaa, jota on helppo käyttää uudelleen. Camel-toteutukseen lisäarvoa tuo mahdollisuus hyödyntää Spring-ohjelmistokehyksen automaattisia määrittely annotaatioita, jolloin koodia tulee vähemmän ja se on oletettavasti selkeälukuisempaa.

Perinteiset integraatiot erilaisilla tiedostoformaateilla onnistuvat Boomi ja Anypoint-alustoilla luomalla erilaisia tietueita määrittäen niiden tietotyypit. Yleisiä tuettuja formaatteja ovat JSON, XML, CSV ja tavallinen merkkijono. Näitä tietueita voidaan tämän jälkeen hyödyntää palautusarvoina rajapinnoille tai muuntamisessa lähde- ja kohdetietueena. Logic App -toteutuksella esimerkiksi CSV-tiedon muuntamiseen ei ole valmista ratkaisua, mutta alustalta löytyy elementtejä, joilla käsittely on mahdollista.

Dokumentaatiomenetelmiä tehdyille rajapinnoille löytyi, mutta niiden luomisen yksinkertaisuus vaihteli alustakohtaisesti. Boomi ei tuota pelkästään prosessissa kuuntelijalla luodusta rajapinnasta dokumentaatiota. Jos toteutus tehdään API-palvelun kautta siten, että alkuperäisesti luotu prosessi käynnistetään REST-rajapintaan tulleen kutsun kautta, kykenee alusta luomaan automaattisesti rajapinnalle Swagger -dokumentaation. Anypoint -alustalla ei dokumentointia tehdä automaattisesti, mutta dokumentaatioita voidaan luoda APISpec-projektin avulla käyttäen Anypoint Studiota tai selainpohjaisesti alustan verkkosovellusta. Logic Apps -alustalla ei dokumentaatioita saanut luotua muille kuin itse luoduille liittimille.

Pilvipalveluihin löytyy alustoilta valmiita yhteysliitännäisiä, mutta niiden määrä ja kohteet vaihtelevat alustakohtaisesti. Logic Apps -toteutuksilla on luonnollisesti kattavat vaihtoehdot Azure-pilvipalvelun osalta, mutta ainoastaan tapahtumien kuuntelijoita Amazon Redshift, S3 ja SQS palveluihin, eikä lainkaan Google Cloud -liitännäisiä [90]. Boomilla ja Anypointilla löytyy useampia liitännäisiä Amazonin ja Azuren lisäksi joitakin myös Google Cloud ympäristöön [21],[91]. Kehyksiltä löytyy joihinkin pilvipalveluihin kirjastoja, joita voidaan toteutuksissa hyödyntää. Springille löytyy valmis AWS-lisäosa, jota voidaan

käyttää Java-määrityksien lisäksi XML-määrityksien kautta [92]. Apache Camel tarjoaa pilvipalveluihin hieman enemmän valmiita komponentteja, joihin kuuluu Amazonin lisäksi Azuren palveluita. Näitä valmiita komponentteja ja niiden rajapintoja voidaan käyttää esimerkiksi reiteissä kohdekanavina [93].

Alustoilla tekeminen yksinkertaistuu merkittävästi, mutta erityisesti liitinelementtien määrityksissä täytyy olla jonkin verran teknistä asiantuntemusta. Esimerkiksi tuntemusta käytettävistä tietoliikenneprotokollista ja niissä käytettävistä tietotyypeistä ja -malleista. Tästä syystä integraatioiden toteuttamista ei voida täysin suositella henkilöille ilman jonkinlaista teknistä taustaa. Erityisesti Logic Apps -alustalla, jossa valmiiden komponenttienkin määritykset ovat usein hyvin teknisiä, eikä alustalla esimerkiksi ole HTTP-yhteyksille uudelleenkäytettäviä pohjia.

Teknisen osaamisen vaatimuksia voidaan kuitenkin madaltaa Boomi-alustan kohdalla, jos yhteysasetuksia ja muita uudelleen käytettäviä asetuskomponentteja on määritelty valmiiksi. Tällöin konseptien tunteminen ei ole niin tärkeää ja yksinkertaisia integraatioita voi teoriassa tehdä vähälläkin teknisellä osaamisella. Boomilla helpotusta tuo myös julkaisun helppous, sillä se tapahtuu samasta näkymästä kuin prosessien luonti. Anypoint-alustalla integraatioiden luonti ilman teknistä taustaa on mahdollista, mutta sisältää jonkin verran haasteita, jolloin käytännössä se ei todennäköisesti ole järkevää. Suurin haaste on se, että sisäverkon resursseja hyödyntäviä elementtejä ei voida testata selainpohjaisen graafisen käyttöliittymän kautta, vaan siihen tarvitaan koneelle asennettava Anypoint Studio.

6.3 Parannukset edelliseen toteutukseen verrattuna

Suurin positiivinen muutos, jonka useimmat alustat tuovat on yksinkertaisuus. Graafiset komponentit ovat pääosin nimetty hyvin, jolloin niistä saa dokumentaation tutkimisella selville käyttötarkoituksen. Määritykset ovat pääsääntöisesti virtaviivaisia, varsinkin konseptit tuntevalle kehittäjälle. Esimerkiksi HTTP-kuuntelijan määrityksissä asetetaan osoite, portit, käytössä olevat HTTP-metodit ja missä muodossa hyötykuorma vastaanotetaan. Itse tehtävien määritysten lisäksi alustat tarjoavat hyödyllisiä lisätarkastuksia, kuten saapuvan kutsun hyötykuormalle tehtävä rakenteen tarkastus ja sen perusteella voidaan suoritus pysäyttää, jos rakenne ei täsmää asetuksissa määritettyä mallia. Vastaavaa tarkistusta tehdään jatkuvasti muissakin elementeissä, jolloin tiedon tarkastuksia ei tarvitse tehdä itse lainkaan.

Toinen huomattava etu alustoilla on yhteysasetusten uudelleen käyttäminen. Käytännössä kaikki yhteyksiä sisältävät elementit koostuvat kahdesta osasta, eli pohja- ja käyttöasetuksista. Pohja-asetukset sisältävät pohjan yhteyksille, sisältäen määritykset isäntäpalvelimen pohjaosoitteelle, portille ja autentikointimenetelmille. Jokainen käyttöasetus koostuu pohja-asetuksien päälle tulevista täsmentävistä määrityksistä, kuten rajapinnan päätepisteen lopullinen osoite, hyötykuorman määritykset, otsikot ja käytettävät parametrit. Käytännössä tämä siis tarkoittaa, että jos esimerkiksi palvelimella sijaitsevan REST-rajapinnan kahta päätepistettä halutaan integraatioissa käyttää, riittää silloin yksi pohja-asetus ja eri päätepisteille tehdään käyttöasetuksiin yksilöivät määritykset, jotka pohjautuvat samaan pohjamääritykseen.

Alustat sisältävät paljon valmiita liitännäiselementtejä suosittuihin palveluihin, kuten Amazon, Salesforce, SAP, JIRA ja Slack. Liitännäiselementtien avulla ei tarvitse tietää määrityksien lisäksi, kuinka kommunikointi kyseisen palvelun kanssa tapahtuu käytännössä. Näin voidaan helpommin tarvittaessa kytkeä palveluja toteutuksiin eikä niiden tekemiseen tarvita niin syvällistä tuntemusta kuin ohjelmallisesti toteutettuna. Valikoima liitännäisistä vaihtelee alustakohtaisesti, mutta esimerkiksi Anypoint-alustalla voi liitännäisiä lisäksi ladata alustan kauppapaikasta. Kaupasta hankitut liitännäiset ovat kolmannen osapuolen toteuttamia, mutta niille voidaan hakea MuleSoftin sertifikaatti [94].

Kehyksillä usein toistuvaa ohjelmointityötä saadaan minimoitua käyttämällä kehyksen tarjoamia rajapintaluokkia. Lisäksi ne sisältävät viestipohjaisten integraatiomallinteiden mukaista toiminnallisuutta ja termistöä, jolloin toteutuksia voidaan suunnitella helposti teoriapohjalta. Tutkimuksen integraatiotapaus saatiin lopulta toteutettua kehyksien eri vaihtoehdoilla hyvin pienellä määrällä ohjelmointia verrattuna aiempaan toteutukseen.

Yrityksen puolesta noussut etu tunnetulle alustalle tai kehykselle siirtymiselle toisi mahdollisuuden rekrytoida paremmin integraatio-osaajia. Tätä tutkittiin käymällä läpi työpaikkailmoituksia hakemalla alustojen nimillä. Lopputuloksena löytyi joitain työpaikkoja pelkästään alustan osaajille, esimerkiksi työnimikkeillä Dell Boomi-asiantuntija. Kehyksien osalta niitä näkyi Java-kehittäjä työnimikkeiden kuvauksissa. Johtopäätelmänä alusta tai kehys voi tuoda etua rekrytointimielessä kohdennetun työpaikkailmoituksen myötä.

6.4 Kompromissit

Alustoille siirtyminen ei integraatiotapauksen yhteydessä tuonut mukanaan juurikaan havaittavia kompromisseja itse toteutuksen lopputuloksen kannalta. Alustoilla on kuitenkin paljon komponentteja ja hieman alustakohtaisesti näitä saatettiin luoda useammassa paikassa, jolloin opeteltavaa on aluksi paljon. Eniten haasteet koskevat toteuttamista ja

kuinka paljon yksityiskohtiin halutaan vaikuttaa. Vanhaan toteutukseen nähden hallintaa joudutaan toteutuksen osalta antamaan enemmän alustoille, sillä elementtien toteutuksiin ei voida vaikuttaa. Joidenkin elementtien yhteydessä on mahdollista toteuttaa mukautettua toimintaa esimerkiksi skripteillä. Kehyksillä toteutukseen yksityiskohtiin voidaan vaikuttaa paljon enemmän, mutta osa toiminnoista jätetään kehyksen taustaprosessien hoidettavaksi. Hallinnan antaminen aiheuttaa esimerkiksi sen, että toteutuksen suorituskykyyn tai pullonkauloihin ei voida vaikuttaa juuri ollenkaan.

Versionhallinta on alustojen tapauksessa jätettävä pääsääntöisesti järjestelmän hallittavaksi. Poikkeuksena on Anypoint-alusta, jossa Anypoint Studiolla luotuja Mule-projekteja voidaan lisätä itse omaan versionhallintaan, kuten GitHub. Muissa alustoissa on mahdollisuuksia tuoda ulos yksittäisiä elementtien tai rajapintojen määrityksiä. Lopulta versionhallinnan jättäminen alustalle ei ole ongelma, varsinkaan yrityksillä, joilla muuta versionhallintaa ei ole olemassa.

7. LOPPUPÄÄTELMÄT

Työssä tehtiin yrityksen vanhan integraatiototeutuksen pohjalta sitä vastaavat toteutukset useammalla integraatioalustalla, sekä kahdella integraatiokehyksellä. Työ alkoi perehtymällä integraatioihin teoriapohjalta, jotta toteutusvaiheessa olisi mahdollisimman kattava pohja integraatioista, niiden käsitteistä ja toteutusvaihtoehdoista. Käsitteiden tunteminen auttoi varsinkin integraatiokehyksillä toteuttaessa, jolloin integraatio-ohjelman suunnittelussa ymmärrettiin mahdollisia toteutustapoja ja arkkitehtuurivaihtoehtoja etukäteen. Alustoilla teorian tunteminen oli pienemmässä roolissa, sillä alustat eivät antaneet merkittävästi mahdollisuuksia vaikuttaa käytettävään integraatioarkkitehtuuriin. Lopulta tämä tuntui luonnolliselta, sillä toteuttamisesta on pyritty tekemään mahdollisimman yksinkertaista eikä arkkitehtuuriin tullut tarvetta tehdä muutoksia. Alustojen toiminta on lähellä palveluväylää, koska siihen tuodaan palveluja erilaisten liitinkomponenttien mukaan ja näitä voidaan prosessissa yhdistää, ilman suoraa yhteyttä toisiinsa.

Teoria osuuden jälkeen oli alun perin tarkoitus tutkia useita alustoja, sen jälkeen rajata vaihtoehtoja kriteerien mukaan ja niiden perusteella testata muutamaa alustaa. Lopulta alustat osoittautuivat hyvin samankaltaisiksi, jolloin järkevä määrä testattavia alustoja täyttyi nopeasti. Tarkempi tutkiminen aloitettiin järjestyksessään niillä alustoilla, missä sisäverkkoon yhdistäminen on mahdollista ja joihin saatiin ensin kokeilutunnukset. Tunnusten saaminen osoittautui määrääväksi tekijäksi valinnoissa tutkimuksen aikarajoitteisuuden takia, sillä alustojen tunnusten saamisessa kesti yhdestä päivästä viikkoihin, tai niitä ei saatu lainkaan. Taustalla tässä epäiltiin olevan kokeilutunnuksia hakevien autenttisuuden varmistaminen. Integraatiokehyksien valinnassa päädyttiin yleisesti suosittuihin ja yrityksen ennalta ehdottamiin avoimen lähdekoodin Spring Integrations- ja Apache Camel -kehyksiin.

Ennen varsinaista toteuttamista vanhan toteutuksen pohjalta luotiin prosessikaavio, jossa esitellään tärkeimmät askeleet valitulle integraatiotapaukselle. Tutkimuksen integraatiotapaus on periaatteiltaan yksinkertainen, mutta sisältää sopivasti erilaisia askelia ja komponentteja, jotta alustojen erot tulevat esiin. Ennen uusien integraatiototeutusten luontia aikaa kului vielä yrityksen sisäisen rajapinnan tietojen hakuun tarkoitetun päätepisteen suunnittelussa. Aiemmin integraatiototeutus haki itse tarvittavat tiedot tietokannasta, mutta luodun päätepisteen myötä tiedot haetaan keskitetysti yhdellä kutsulla. Tämä yksinkertaistaa integraation luomisprosessia, kun uudelle toteutukselle jää vastuulleen vain tarvittaessa muuttaa haettavia tietoja ja välittää ne haluttuun kohteeseen.

Toteuttaminen alustoilla oli lopulta yksinkertaista, mutta aluksi aikaa kului oikeiden elementtien etsimiseen ja dokumentaatioiden lukemiseen. Koska alustat olivat periaatteiltaan samanlaisia, sujui loppujen toteuttaminen paljon vaivattomammin, kun yksi toteutus saatiin valmiiksi. Toteutuksen lisäksi varsinkin virheidenhallinta aiheutti lisätyötä. Lokitiedot alustoilla olivat parhaimmillaankin keskinkertaisia, eivätkä ne useinkaan tarjonneet valaisevia tietoja virheiden syistä. HTTP-kutsuissa sattuneet virheet eivät oletuksena välittyneet vastausten mukana, vaan näihin piti luoda erillisiä käsittelyjä, jotta myös virheviestit saadaan välitettyä. Näistä ongelmista kuitenkin positiivisena asiana huomattiin prosessikaavioon ottaa mukaan myös virheiden hallinta, jolloin kaavion lopputulos on lopulta hyödyllisempi.

Alustat tarjosivat lopullisen integraatiototeutuksen julkaisuun mahdollisuuksia joko suorittaa paikallista ajoympäristöä tai ylläpitää alustaa omalla palvelimella. Täysin pilvipohjainen vaihtoehto on myös mahdollinen, mutta tämän tutkimuksen yhteydessä haluttiin päästä yrityksen sisäverkossa sijaitseviin resursseihin käsiksi, jolloin täytyi käyttää edellä mainittuja hybridiratkaisuja. Kaikista toteutuksista lopputuloksena saatiin rajapinta, jota voidaan kutsua HTTP-kutsuilla, aivan kuten aiempaakin integraatiototeutusta.

Tutkimuksen tarkoituksena oli saada vastauksia seuraaviin tutkimuskysymyksiin:

1. **Kuinka olemassa olevan integraatiototeutuksen saa siirrettyä integraatioalustalle?**
2. **Mitä hyötyjä alustalle siirtymisellä saadaan?**
3. **Mitä kompromisseja siirtymisessä joudutaan tekemään?**

Kysymyksiin saatujen vastauksien avulla voidaan laajemmin pohtia, tarjoaako alustalle siirtyminen niin merkittäviä hyötyjä, että se kannattaa ohjelmoidun toteutuksen sijasta vai olisiko tarkoitukseen sopiva integraatiokehys perustellumpi ratkaisu.

Kuinka olemassa olevan integraatiototeutuksen saa siirrettyä integraatioalustalle?

Koska koodia ei voida suoraan siirtää alustoille on vaihtoehtona eristää kyseessä olevan integraatiotapauksen kulku esimerkiksi kaaviona, jolloin se on teknologiariippumattomassa muodossa. Kaavion tai jonkin muun prosessista luodun dokumentin pohjalta toteutus on helppo tehdä alustasta tai kehyksestä riippumatta. Graafisia komponentteja on alustasta riippuen mahdollista mukauttaa, jolloin vanhaa logiikkaa voidaan tapauskohtaisesti hyödyntää alustalla. Kokeilujen perusteella on usein tarkoituksenmukaisempaa pyrkiä käyttämään alustan alkuperäisiä elementtejä, koska silloin integraatiokulkuja on helpompi ymmärtää ja ne pysyvät selkeämpänä.

Mitä hyötyjä alustalle siirtymisellä saadaan?

Alustan yksi merkittävimpiä hyötyjä on nopea tapa luoda integraatioita. Graafisen käyttöliittymän kautta voidaan osat raahata paikoilleen ja ainoastaan määrittää komponenttien tarvittavat asetukset. Edun varsinkin SaaS-jakelumallia hyödyntävillä yrityksillä tuo se, että integraatioiden toteuttaminen voidaan tehdä hyvin pienellä infrastruktuurilla. Mikäli integraatiototeutuksilla ei ole tarvetta päästä sisäverkkoon alustat pystyvät toimimaan täysin pilvipohjaisesti, jolloin niitä varten ei tarvitse hankkia esimerkiksi omia palvelimia. Yksinkertaisimmissa tapauksissa integraatioita voivat luoda muutkin kuin kehittäjät. Onnistuakseen tämä kuitenkin usein vaatii hieman teknistä osaamista tai sen, että elementtien teknisemmät osat, kuten palvelinmääritykset on tehty valmiiksi, jolloin niitä voidaan vain sijoittaa näkymään. Suorituskykyparannukset ovat edelliseen toteutukseen verrattaessa merkittäviä. Parhaassa tapauksessa Boomin integraatioprosessilla saavutettiin noin neljä kertaa nopeampi suoritusaika. Ainoa poikkeus on Azuren Logic App -toteutus, joka oli lopulta hitaampi kuin aiempi toteutus. Suorituskyvystä huolimatta Azuren vahvuuksia on kuitenkin sen liittyminen muuhun Microsoft-ekosysteemiin. Jos yritys nojautuu vahvasti Microsoftin tuotteisiin, kuten Azure-pilvipalveluihin ja Active Directory -käyttäjien hallintaan, voi silloin ratkaisevana tekijänä olla suorituskyvyn sijasta ennalta tuttu ympäristö ja siihen sulautuva integraatioiden toteutusalusta.

Mitä kompromisseja siirtymisessä joudutaan tekemään?

Alustoilla ei ole hirveästi merkittäviä kompromisseja toteutuksen suhteen, ainakaan jos verrataan ilman integraatiokehystä luotuun kokonaisuuteen. Alustojen sisältäessä paljon graafisia komponentteja, hallinta jää alustan hoidettavaksi, eikä lopulliseen toteutukseen tai sen suorituskykyyn voida vaikuttaa yhtä paljon, kuin jos se olisi tehty ohjelmoimalla. Mukauttaminen onkin alustojen yleisin haaste verrattuna itse luotuihin toteutuksiin. Ohjelmoinnin vähentyessä merkittävästi, joillekin kehittäjille graafisen käyttöliittymän käyttäminen pidemmällä aikajaksolla voi olla epämiellyttävää. Tähän vaikuttavat luonnollisesti kehittäjän mieltymykset. Versionhallinta alustojen integraatiototeutuksilla jää myös pääsääntöisesti alustan hallintaan.

Tutkimuskysymysten yhteenveto

Alustalla tai kehyksellä saavutettavaan lisäarvoon vaikuttavat ominaisuudet, jotka integraatioiden toteuttamisessa koetaan tärkeimmiksi. Silloin voidaan pohtia, tarvitaanko kokonainen uusi järjestelmä integraatioiden tuottamiseen. Uuden järjestelmän käytön opettelu vaatii oman prosessinsa, vaikka integraatioiden tekeminen helpottuukin, kun järjestelmän hallitsee. Toisaalta voidaan miettiä parantaisiko tarkoitukseen räätälöity kehys aiemman toteutuksen heikkouksia niin merkittävästi, että kokonainen uusi järjestelmä voi

tuoda enemmän sekaannusta tai se on kustannuksellisesti perusteeton. Lopulta yleistä vastausta siihen kannattaako integraatioalusta aina itse ohjelmoidun toteutuksen sijasta ei voida antaa. Vaikuttavia tekijöitä on niin paljon, että päätös on tehtävä tapauskohtaisesti ottaen huomioon tarpeet integraatioiden suhteen. Esimerkiksi jos yritys nojautuu SaaS-jakelumalliin tuotteissaan, ei ole mielekästä luoda integraatiototeutusta itse. Mikäli taas integraatioissa on saatava itse vaikuttaa toimintamalleihin ja varsinaiseen arkkitehtuuriin, on kehys parempi vaihtoehto. Joka tapauksessa yhtenäinen ratkaisu on kannattava ylläpidon ja päivitettävyyden kannalta, oli se sitten alusta tai kehys.

Tutkimuksen yrityksen kannalta alustalle tai kehykselle siirtyminen tuo merkittäviä parannuksia nykytilanteeseen. Verrattavista alustoista selkeästi parhain vaihtoehto yrityksen tarpeisiin on Boomi, sillä se oli helpoin ottaa käyttöön ja käyttökokemus oli miellyttävä. Lisäksi sen suorituskyky oli vertailun parhaimmistoa. Sisäverkossa toimiva ajoympäristö on kevyt ja se voidaan asentaa sekä Windows että Linux palvelimille. Ajoympäristöllä on myös vaihtoehtoja kuormanhallinnalle muuttamalla Atom-ajoympäristö monipuolisempiin Molecule- tai Atom Cloud -suoritusympäristöihin. Jälkimmäinen ympäristö mahdollistaa esimerkiksi asiakkaiden hallinnan eri käyttäjäryhminä, jolloin voidaan samoille resursseille tehdä ryhmäkohtaista hallintaa. Kehyksien kohdalla vanhoja koodeja voidaan hyödyntää, sillä yrityksen aiempi toteutus on tehty Javalla. Tämä mahdollistaa vanhan toteutuksen siirtämisen uudelle kehykselle paloissa, sekoittamatta vanhaa toimintaa. Kehyksistä Spring Integration on yritykselle luonnollisempi vaihtoehto, sillä heillä on useassa toteutuksessa Spring-ohjelmistokehys käytössä ja kyseinen integraatiokehys sulautuu tähän hyvin. Kehys tuo sekä suorituskyky parannuksia että helpompia tapoja luoda integraatioita hyödyntämällä teoriapohjan mukaisia malleja. Camel-integraatiokehys tuo myös parannuksia, mutta ei ole yhtä luonnollinen vaihtoehto, vaikka sekin on mahdollista yhdistää Spring Integration-kehyksen kanssa. Näiden lisäksi potentiaalinen vaihtoehto on Anypoint-alustan taustalla toimiva Mule ESB, josta on avoimen lähdekoodin vaihtoehto ja sille voidaan integraatiototeutuksia luoda käyttämällä ilmaisversiota Anypoint Studiosta. Näiden kokemuksien perusteella loppupäätelmänä on se, että yritykselle on kannattavaa siirtää aiempi toteutus alustalle tai kehykselle. Lopulta kumpi tahansa valitaankin ovat hyödyt joka tapauksessa merkittäviä ja valinta tulee tehdä tarpeen mukaan.

Jatkokehitystä tutkimuksen pohjalta olisi hyödyntää OAuth-valtuutusvirtoja osana integraatiototeutuksia. Alustoilla sen käyttäminen on helpompaa, sillä siihen löytyy valmiita elementtejä. Kehyksillä niiden hyödyntäminen täytyy luoda itse, mutta Spring Integration -kehyksellä voidaan tähän kuitenkin hyödyntää Spring-ohjelmistokehyksen Spring Security -projektia. Tehty prosessikaavio osoittautui jatkon kannalta hyväksi työkaluksi, sillä

yhdelläkään alustalla ei voitu käyttää suoraan vanhan toteutuksen ohjelmakoodia. Tärkeimpien askelien ollessa määriteltynä, voitiin keskittyä niiden toteuttamiseen alustan tai kehyksen tarjoamilla elementeillä. Lisäksi kaavio mahdollistaa integraatioiden uudelleen luomisen minkä tahansa teknologian pohjalta. Eli jatkossa integraatioita voitaisiin suunnitella kaavioiden tai muun teknologiariippumattoman dokumentoinnin avulla. Tällöin järjestelmän päivittäminen ja uusien integraatioiden luonti tulee yksinkertaisemmaksi, kun tärkeimmät tapahtumat on eristetty etukäteen.

LÄHTEET

[1] Gartner - Magic Quadrant, 2020. Saatavissa: https://www.gartner.com/doc/reprints?id=1-2481C9JS&ct=200923&st=sb

[2] D. Chen, G. Doumeingts, F. Vernadat - Architectures for enterprise integration and interoperability: Past, present and future, Computers in Industry Research Journal, Volume 59, Issue 7, 2008, Pages 647-659. https://doi.org/10.1016/j.compind.2007.12.016

[3] G. Hophe, B. Woolf - Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley Educational Publishers Inc, 2004

[4] G. Hophe, B. Woolf - Enterprise Integration Patterns: Messaging Patterns. Saatavissa: https://www.enterpriseintegrationpatterns.com/patterns/messaging/index.html (Viitattu 07/2021)

[5] D. McLeod, D. Heimbigner - A federated architecture for database systems, Proceedings of the May 19-22, 1980, national computer conference, 1980. https://doi.org/10.1145/1500518.1500561

[6] D. R. Ferreira, Enterprise Systems Integration, Springer-Verlag Berlin Heidelberg, 2013.

[7] What is Mule ESB, MuleSoft, verkkosivu. saatavissa: https://www.mulesoft.com/resources/esb/what-mule-esb (Viitattu 08/2021)

[8] T. Gulledge, T. (2006), "What is integration?", Industrial Management & Data Systems, Vol. 106 No. 1, pp. 5-20. https://doi.org/10.1108/02635570610640979

[9] A. Nghiem - Web Services Part 6: Models of Integration, Informit, 2002. Saatavissa: https://www.informit.com/articles/article.aspx?p=28713

[10] C. Bussler - B2B integration, Springer-Verlag Berlin Heidelberg, 2003

[11] D. Draheim, Business Process Technology, Springer-Verlag Berlin Heidelberg, 2010

[12] B. A. Christudas, Service Oriented Java Business Integration, Packt Publishing, 2008

[13] E. B. Fernandez, N. Yoshioka, Two Patterns for Distributed Systems: Enterprise Service Bus (ESB) and Distributed Publish/Subscribe, In Proceedings of the 18th Conference on Pattern Languages of Programs (PLoP '11). Association for Computing Machinery, New York, NY, USA, Article 8, 1–10, 2011. https://doi.org/10.1145/2578903.2579146

[14] D. A. Chappell, Enterprise Service Bus, O'Reilly Media, Inc. 2004

[15] M. Marian, iPaaS: Different Ways of Thinking, Procedia Economics and Finance, Volume 3, 2012, Pages 1093-1098. https://doi.org/10.1016/S2212-5671(12)00279-1

[16] N. Ebert, K. Weber, S. Koruna - Integration Platform as a Service, Bus Inf Syst Eng 59, 375–379 (2017). https://doi.org/10.1007/s12599-017-0486-0

[17] Gartner Glossary – Integration Platform as a Service (iPaaS), https://www.gartner.com/en/information-technology/glossary/information-platform-as-a-service-ipaas (Viitattu 07/2021)

[18] Boomi AtomSphere Documentation - Building a custom connector. Saatavissa: https://help.boomi.com/bundle/connectors/page/int-Building_your_own_custom_connector.html (Viitattu 07/2021)

[19] Anypoint Documentation - About Anypoint Connector DevKit. Saatavissa: https://docs.mulesoft.com/connector-devkit/3.9/ (Viitattu 07/2021)

[20] Boomi AtomSphere Documentation – Custom scripting. Saatavissa: https://help.boomi.com/bundle/integration/page/c-atm-Custom_scripting.html (Viitattu 07/2021)

[21] Boomi AtomSphere Documentation – Application connectors. Saatavissa: https://help.boomi.com/bundle/connectors/page/c-atm-Application_connectors.html (Viitattu 08/2021)

[22] Dirk Riehle. Framework Design: A Role Modeling Approach. Ph.D. Thesis, No. 13509. Zürich, Switzerland, ETH Zürich, 2000. Saatavilla: https://riehle.org/computer-science/research/dissertation/diss-a4.pdf

[23] Jitterbit kotisivu - Cloud Studio. Saatavissa: https://www.jitterbit.com/platform/cloud-studio (Viitattu 06/2021)

[24] Dell to Acquire Boomi; Adds Industry's No. 1 Integration Cloud™ Solution to SaaS Capabilities, uutisartikkeli, 2010. Saatavissa: https://www.dell.com/learn/us/en/uscorp1/secure/2010-11-02-boomi

[25] Francisco Partners and TPG to Acquire Boomi from Dell Technologies, PR Newswire, 2021. Saatavissa: https://www.prnewswire.com/news-releases/francisco-partners-and-tpg-to-acquire-boomi-from-dell-technologies-301281744.html

[26] Boomi kotisivu – Platform. Saatavissa: https://boomi.com/platform (Viitattu 07/2021)

[27] Boomi AtomSphere Documentation - Boomi Flow, Getting Started. Saatavissa: https://help.boomi.com/bundle/flow/page/c-flo-Getting_Started.html (Viitattu 07/2021)

[28] Boomi AtomSphere Documentation - Boomi Flow, Visual Overview. Saatavissa: https://help.boomi.com/bundle/flow/page/c-flo-Visual_Overview.html (Viitattu 07/2021)

[29] Boomi AtomSphere Documentation – Boomi Integration, Flow Service components. Saatavissa: https://help.boomi.com/bundle/integration/page/c-atm-Flow_Service_components.html (Viitattu: 07/2021)

[30] Boomi AtomSphere Documentation – Boomi Integration, Process building. Saatavissa: https://help.boomi.com/bundle/integration/page/c-atm-Process_building.html (Viitattu 07/2021)

[31] Boomi AtomSphere Documentation – Boomi Integration, Boomi Integration service. Saatavissa: https://help.boomi.com/bundle/integration/page/c-atm-Boomi_Integration_application.html (Viitattu 07/2021)

[32] Boomi AtomSphere Documentation – Boomi Integration, Start shape. Saatavissa: https://help.boomi.com/bundle/integration/page/r-atm-Start_shape.html (Viitattu 06/2021)

[33] Boomi AtomSphere Documentation – Boomi Integration, Process shapes. Saatavissa: https://help.boomi.com/bundle/integration/page/r-atm-Process_shapes.html (Viitattu 06/2021)

[34] Boomi AtomSphere Documentation – Boomi Integration, Atoms, Molecules, and Atom Clouds. Saatavissa: https://help.boomi.com/bundle/integration/page/int-Atoms_Molecules_and_Atom_Clouds.html (Viitattu 06/2021)

[35] Boomi AtomSphere Documentation – Boomi Integration, Atoms. Saatavissa: https://help.boomi.com/bundle/integration/page/c-atm-Atoms.html (Viitattu 06/2021)

[36] Boomi AtomSphere Documentation – Boomi Integration, Molecules. Saatavissa: https://help.boomi.com/bundle/integration/page/c-atm-Molecules.html (Viitattu 06/2021)

[37] Boomi AtomSphere Documentation – Boomi Integration, Atom Clouds. Saatavissa: https://help.boomi.com/bundle/integration/page/c-atm-Atom_Clouds.html (Viitattu 06/2021)

[38] Anypoint Documentation - Deployment Options. Saatavissa: https://docs.mulesoft.com/runtime-manager/deployment-strategies (Viitattu 06/2021)

[39] Anypoint Documentation - Installing Anypoint Platform On Premises Edition. Saatavissa: https://docs.mulesoft.com/private-cloud/1.1/installing-anypoint-on-premises (Viitattu 06/2021)

[40] Microsoft Azure - Integration Services. Saatavissa: https://azure.microsoft.com/en-us/product-categories/integration/ (Viitattu 06/2021)

[41] Azure Logic Apps documentation – What is Azure Logic Apps? Saatavissa: https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-overview (Viitattu 06/2021)

[42] Azure Logic Apps documentation – Create and run your own code from workflows in Azure Logic Apps by using Azure Functions. Saatavissa: https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-azure-functions (Viitattu 06/2021)

[43] Azure Logic Apps documentation – Add and run code snippets by using inline code in Azure Logic Apps. Saatavissa: https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-add-run-inline-code (Viitattu 06/2021)

[44] Azure Logic Apps documentation – What is an on-premises data gateway? Saatavissa: https://docs.microsoft.com/en-us/data-integration/gateway/service-gateway-onprem (Viitattu 06/2021)

[45] D. Spinellis, Notable design patterns for domain-specific languages, 2001 https://doi.org/10.1016/S0164-1212(00)00089-3

[46] Apache Camel Spring Boot – About. Saatavissa: https://camel.apache.org/camel-spring-boot/latest/ (Viitattu 07/2021)

[47] Apache Camel Components, Spring Integration. Saatavissa: https://camel.apache.org/components/latest/spring-integration-component.html (Viitattu 07/2021)

[48] Spring Integration - Overview of Spring Integration Framework. Saatavissa: https://docs.spring.io/spring-integration/reference/html/overview.html (Viitattu 07/2021)

[49] Unifaun REST API Documentation – Shipments. https://api.unifaun.com/rs-docs/#/#shipments (Viitattu 08/2021)

[50] Unifaun Online – Johdanto. https://help.unifaun.com/uo-fi/fi/johdanto.html (Viitattu 08/2021)

[51] Oracle Database Express Edition 2 Day Developer Guide - 5 Using Procedures, Functions, and Packages. Saatavissa: https://docs.oracle.com/cd/B25329_01/doc/appdev.102/b25108/xedev_programs.htm (Viitattu 06/2021)

[52] Oracle - Oracle Java SE Support Roadmap. https://www.oracle.com/java/technologies/java-se-support-roadmap.html (Viitattu 05/2021)

[53] Solita, Pilvipohjainen Dell Boomi - helppo ja nopea integraatiopalvelu. Saatavissa: https://www.solita.fi/dell-boomi/ (Viitattu 08/2021)

[54] Boomi AtomSphere Documentation – Boomi Integration, Connector components. Saatavissa: https://help.boomi.com/bundle/integration/page/c-atm-Connector_components.html (Viitattu 05/2021)

[55] Boomi AtomSphere Documentation – Connectors, Web Services Server operation. Saatavissa: https://help.boomi.com/bundle/connectors/page/r-atm-Web_Services_Server_operation.html (Viitattu 05/2021)

[56] Boomi AtomSphere Documentation – Boomi Integration, Profile components. Saatavissa: https://help.boomi.com/bundle/integration/page/c-atm-Profile_components.html (Viitattu 05/2021)

[57] Boomi AtomSphere Documentation – Connectors, HTTP Client connector. Saatavissa: https://help.boomi.com/bundle/connectors/page/r-atm-HTTP_Client_connector.html (Viitattu 05/2021)

[58] Boomi AtomSphere Documentation – Connectors, Adding an HTTP Client connection using OAuth 2.0. Saatavissa: https://help.boomi.com/bundle/connectors/page/t-atm-Adding_an_HTTP_Client_connection_that_uses_OAuth_2_0.html (Viitattu 05/2021)

[59] Boomi AtomSphere Documentation – Boomi Integration, Packaged Components. Saatavissa: https://help.boomi.com/bundle/integration/page/int-Packaged_components.html (Viitattu 05/2021)

[60] Boomi AtomSphere Documentation – API Management, Downloading the local Gateway installer. Saatavissa: https://help.boomi.com/bundle/api_management/page/api-Downloading_the_local_Gateway_installer.html (Viitattu 05/2021)

[61] Boomi AtomSphere Documentation – API Management, Migrating an environment to or from an API Gateway. Saatavissa: https://help.boomi.com/bundle/api_management/page/api-Migrating_environments_to_or_from_API_gateways.html (Viitattu 05/2021)

[62] Boomi AtomSphere Documentation – API Management, Accessing the Swagger Visualization Portal. Saatavissa: https://help.boomi.com/bundle/api_management/page/t-api-Accessing_the_Swagger_Visualization_Portal.html (Viitattu 05/2021)

[63] Boomiverse, Knowledge Article, How to Install a Local Atom, Molecule, or Cloud, 2018. Saatavissa: https://community.boomi.com/s/article/HowtoInstallaLocalAtomMoleculeorCloud

[64] Boomi AtomSphere Documentation – Boomi Integration, Mapping elements from source to destination. Saatavissa: https://help.boomi.com/bundle/integration/page/t-atm-Mapping_elements_from_source_to_destination.html (Viitattu 05/2021)

[65] Mulesoft kotisivu - Anypoint Studio. Saatavissa: https://www.mulesoft.com/platform/studio (Viitattu 05/2021)

[66] Anypoint Documentation – Connectors, Sockets Connector. Saatavissa: https://docs.mulesoft.com/sockets-connector/1.1/ (Viitattu 05/2021)

[67] Anypoint Documentation – Connectors, HTTP Listener Reference. Saatavissa: https://docs.mulesoft.com/http-connector/1.5/http-listener-ref HTTP-kuuntelija (Viitattu 05/2021)

[68] Anypoint Documentation – Runtime Manager, Anypoint Platform CLI 3.x. Saatavissa: https://docs.mulesoft.com/runtime-manager/anypoint-platform-cli (Viitattu 05/2021)

[69] Anypoint Documentation - Mule Runtime, Deploy Mule Applications. Saatavissa: https://docs.mulesoft.com/mule-runtime/4.3/deploying (Viitattu 05/2021)

[70] Azure Logic Apps documentation – Install on-premises data gateway for Azure Logic Apps. Saatavissa: https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-gateway-install (Viitattu 05/2021)

[71] Azure Logic Apps documentation – Connect to on-premises data sources from Azure Logic Apps. Saatavissa: https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-gateway-connection (Viitattu 05/2021)

[72] Azure Logic Apps documentation – Perform data operations in Azure Logic Apps. Saatavissa: https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-perform-data-operations (Viitattu 05/2021)

[73] Azure Logic Apps documentation – Transform JSON and XML using Liquid templates as maps in Azure Logic Apps. Saatavissa: https://docs.microsoft.com/en-

us/azure/logic-apps/logic-apps-enterprise-integration-liquid-transform (Viitattu 05/2021)

[74] Liquid dokumentaatiosivu. Saatavissa: https://shopify.github.io/liquid/ (Viitattu 05/2021)

[75] Spring Quickstart Guide. Saatavissa: https://spring.io/quickstart (Viitattu 06/2021)

[76] Spring guides - Building an Application with Spring Boot. Saatavilla: https://spring.io/guides/gs/spring-boot/ (Viitattu 06/2021)

[77] Project Lombok – Lombok features. Saatavilla: https://projectlombok.org/features/all (Viitattu 06/2021)

[78] Apache Camel - User manual, JSON. Saatavissa: https://camel.apache.org/manual/latest/json.html (Viitattu 06/2021)

[79] Spring Integration Reference Guide – Transformer, JSON Transformers. Saatavissa: https://docs.spring.io/spring-integration/reference/html/transformer.html#json-transformers (Viitattu 06/2021)

[80] Github – FasterXML, Jackson. Saatavissa: https://github.com/FasterXML/jackson (Viitattu 06/2021)

[81] Spring Integration Reference Guide – Configuration. Saatavissa: https://docs.spring.io/spring-integration/docs/current/reference/html/configuration.html (Viitattu 06/2021)

[82] Spring Integration Reference Guide – Java DSL. Saatavissa: https://docs.spring.io/spring-integration/docs/current/reference/html/dsl.html#java-dsl-basics (Viitattu 06/2021)

[83] Spring Framework Documentation - Core. Saatavissa: https://docs.spring.io/spring-framework/docs/current/reference/html/core.html (Viitattu 06/2021)

[84] Apache Camel - User manual, Java DSL. Saatavissa: https://camel.apache.org/manual/latest/java-dsl.html (Viitattu 06/2021)

[85] Apache Camel - User manual, REST DSL. Saatavissa: https://camel.apache.org/manual/latest/rest-dsl.html (Viitattu 06/2021)

[86] Spring Framework Java-doc – Component Annotation. Saatavissa: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html (Viitattu 06/2021)

[87] Apache Camel - Components, Enterprise Integration Patterns, To D. Saatavissa: https://camel.apache.org/components/latest/eips/toD-eip.html (Viitattu 06/2021)

[88] Apache Camel - Components, Data Formats, JSON Jackson. Saatavissa: https://camel.apache.org/components/latest/dataformats/json-jackson-dataformat.html (Viitattu 06/2021)

[89] Apache Camel - Components, Bean. Saatavissa: https://camel.apache.org/components/latest/bean-component.html (Viitattu 06/2021)

[90] Microsoft - Connector reference overview. Saatavissa: https://docs.microsoft.com/en-us/connectors/connector-reference/ (Viitattu 08/2021)

[91] Anypoint Documentation – Connectors. Saatavissa: https://docs.mulesoft.com/connectors/ (Viitattu 08/2021)

[92] GitHub – Spring Projects, Spring Integration Extension for Amazon Web Services. Saatavissa: https://github.com/spring-projects/spring-integration-aws (Viitattu 08/2021)

[93] Apache Camel - Components. Saatavissa: https://camel.apache.org/components/latest/index.html (Viitattu 08/2021)

[94] Mulesoft, Anypoint Connector certification program. Saatavilla: https://www.mulesoft.com/platform/cloud-connectors/certified (Viitattu 08/2021)

[95] S. U. A. Ayubi, N. Nugrahaningsih, Centralized-Star Architecture of Web Service Node as Integration Solution in Complex Organization, 2009, Proceedings of iiWAS2009. https://doi.org/10.1145/1806338.1806451

[96] V. Sauermann, Experience paper about application of Enterprise Integration Patterns in an industry context, 2016, Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPlop '16). https://doi.org/10.1145/3011784.3011787

[97] E. Kaneshima, R. T. V. Braga, Patterns for Enterprise Application Integration, 2012, IX Latin America Conference on Pattern Languages of Programming. https://doi.org/10.1145/2591028.2600811

LIITE 1: ALKUPERÄINEN PROSESSIKUVAAJA

LIITE 2: PARANNELTU PROSESSIKAAVIO