Tampereen ammattikorkeakoulu

Integraatiokehitys Apache Camel -työkalulla

Olavi Kurola

OPINNÄYTETYÖ
Huhtikuu 2021

Tieto- ja viestintätekniikka
Ohjelmistotekniikka

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tieto- ja viestintätekniikka
Ohjelmistotekniikka

KUROLA, OLAVI:
Integraatiokehitys Apache Camel -työkalulla

Opinnäytetyö 48 sivua
Huhtikuu 2021

Opinnäytetyössä perehdyttiin nykyaikaiseen integraatiokehitykseen sekä integraatioalalla perinteisesti esiintyneisiin ongelmakohtiin. Integraatiokehityksen työnkulkua parantavia tekijöitä pohdittiin muun muassa integraatioiden toteutustyylien yhtenäistämisen näkökulmasta. Lisäksi tutkittiin alalla esiintyvien lukuisten integraatiotyökalujen osuutta tähän työnkulkuun. Näistä integraatiotyökaluista valittiin tarkemman tarkastelun kohteeksi Apachen kehittämä Camel-työkalu. Tutkielman ohessa toteutettiin Digia Oyj:n sekä Tampereen korkeakouluyhteisön integraatiopalvelun yhteistyönä yksi opiskelijakorttidataa siirtävä integraatio, joka kehitettiin edellä mainittua Camel-työkalua käyttäen. Työn tavoitteina oli arvioida, miten hyvin Camel soveltuu integraatiotyökaluksi sekä muodostaa yleiskuva nykyaikaisesta integraatiokehityksestä.

Tutkielman alussa asetettu tavoite nykyaikaisen integraatiokehityksen ymmärryksen muodostumisesta saavutettiin, sillä työhön kuuluvaa integraatiota toteuttaessa jouduttiin tutustumaan moneen ajallisesti relevanttiin teknologiaan sekä arkkitehtuuriin. Tämän lisäksi työssä toteutettu integraatio saatiin suunnitellusti valmiiksi, ja kyseinen integraatio otettiin käyttöön Tampereen korkeakouluyhteisön integraatiopalvelun tuotantoympäristössä. Tätä toteutetun integraation siirtämää opiskelijakorttidataa hyödynnettiin muun muassa integraatiopalvelun kulunvalvonnan integraatioissa.

Työn lopuksi arvioitiin käytetyn Camel-työkalun kokonaistason toimivuutta integraatiokehityksessä sekä pohdittiin sen vahvuuksia ja heikkouksia. Pohdinnan ohessa todettiin, että ääritapauksissa on olemassa parempiakin vaihtoehtoja integraatiokehitykselle kuin Apache Camel, mutta Camelia pidettiin silti erittäin pätevänä yleiskäyttöisenä integraatiotyökaluna. Camelin suurimpina haastajina pääteltiin olevan valtaviksi kokonaisuuksiksi kehitetyt ESB-järjestelmät, jotka sopivat muun muassa isoihin integraatioprojekteihin Camelia paremmin lukuisten analytiikka- ja monitorointiominaisuuksiensa vuoksi. Tämän lisäksi todettiin vielä, että tällainen ESB-järjestelmä voisi sopia myös Tampereen korkeakouluyhteisön integraatiopalvelulle, jossa paisuneen projektikoon myötä kevyemmät ratkaisut, kuten käytössä oleva Camel-työkalu, saattavat jäädä ajan mittaan liian heiveröisiksi.

Asiasanat: integraatiokehitys, Apache, Camel, ESB

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in ICT Engineering
Software Engineering

KUROLA, OLAVI:
Integration Development Using Apache Camel

Bachelor's thesis 48 pages
April 2021

Integration development has long been plagued by differing implementation methods and the vast amount of different integration tools. Integrating new systems has been needlessly complicated due to every system being integrated in their own way, based on the experiences and preferences of the developers working on the project.

The purpose of this thesis was to familiarize the author with one of the integration tools that was built as a solution to this problem, as well as to get acquainted with integration development from the ground up. The tool that this thesis focused on is called Apache Camel, a lightweight integration framework developed by Apache.

In order to gain a deeper understanding of integration development, an integration was developed from the beginning with the aforementioned Camel tool. This integration was developed in collaboration with Digia Plc and the integration service of Tampere Universities (TUNI). The objective of the integration was to transfer student card data, so that the data could be used by other integrations in the TUNI integration service.

The integration project went as planned and the result was a fully-fledged integration, which was immediately utilized in the production environment of the TUNI integration service. The data transferred by the implemented integration was used by other TUNI access control integrations. A fair understanding of modern integration development and the usage of Apache Camel were also gained, and an assessment regarding the usefulness of the tool was made.

The findings indicate that Apache Camel is indeed a very efficient and lightweight tool that addresses many problems that integration development has seen in the past. Camel is found to be a part of the solution required to unify integration development practices; however, it is also found that there are other tools that are more suitable on some specific occasions. For example, heavyweight ESB integration tools that include analytical and monitoring features are found to be a better fit for bigger and more complex integration projects.

Key words: integration, development, Apache, Camel, ESB

SISÄLLYS

1 JOHDANTO 6
2 PROJEKTIN KESKEISET TEKNOLOGIAT 7
2.1 Docker 7
2.2 Java 7
2.3 MariaDB 9
2.4 Apache Maven 10
2.5 OSGi-palvelualusta 10
2.6 Apache Karaf 11
2.7 Apache Camel 11
2.8 Enterprise Service Bus 12
2.9 Apache ServiceMix 13
3 MIKSI INTEGRAATIOTYÖKALUJA TARVITAAN 14
3.1 Integraatiokehityksen ongelmat 14
3.2 Apache Camel ratkaisuna ongelmiin? 14
4 INTEGRAATION TOTEUTTAMINEN APACHE CAMELILLA 18
4.1 Kehitysympäristön tarpeet 19
4.2 OSGi-bundlejen määrittely 22
4.3 Maven-riippuvuuksien määrittely 23
4.4 Reittien kirjoittaminen 25
4.5 Java-apuluokkien hyödyntäminen 29
4.6 Integraation kääntäminen Mavenilla 33
4.7 Integraatiopaketin asentaminen ja ajo 35
4.8 Työn tulokset ja viimeistely 39
4.9 Työn analysointi 40
5 POHDINTA 43
5.1 Mietteet Apache Camelista 43
5.2 Pohdintaa projektin hyödyllisyydestä 44
LÄHTEET 46

LYHENTEET JA TERMIT

TUNI	Tampereen korkeakouluyhteisö
some	sosiaalinen media
JVM	Javan virtuaalikone (*Java Virtual Machine*)
bundle	paketti (esim. paketoitu ohjelmisto)
hot deploy	pikaista asennusta tarjoava ominaisuus
DSL	täsmäkieli (*domain-specific language*)
URI	merkkijono, jolla osoitetaan tiedon paikka (*Uniform Resource Identifier*)
open source	avoin lähdekoodi
ESB	yrityksen palveluväylä (*Enterprise service bus*)
backend	palvelinpuoli
EIP	samannimiseen kirjaan perustuvat integraatiokaaviot (*Enterprise Integration Patterns*)
AWS	Amazonin pilvipalvelutarjoama (*Amazon Web Services*)
EC2	prosessointikapasiteettia tarjoava pilvipalvelu AWS:n sisällä (*Elastic Compute Cloud*)
CSV	taulukkomainen tiedostomuoto datan säilytykseen (*Comma-separated values*)
IDE	integroitu ohjelmointiympäristö (*Integrated development environment*)
JDBC	tietokantaoperaatioita tukeva Java-kirjasto (*Java Database Connectivity*)
JAR	yhteen tiedostoon paketoitu Java-ohjelma (*Java Archive*)
SSH	salattuun tietoliikenteeseen tarkoitettu protokolla (*Secure Shell Protocol*)
JSON	yksinkertainen tiedostomuoto tiedonvälitykseen (*JavaScript Object Notation*)

1 JOHDANTO

Integraatiot ovat todella keskeisellä sijalla modernien tietojärjestelmien toimivuuden kannalta. Integraatiopioneerit Gregor Hohpe ja Bobby Woolf kirjoittavatkin kirjan Enterprise Integration Patterns (2003) kotisivuilla, että nykypäivän sovellukset harvoin elävät eristyksissä. Käyttäjät odottavat välitöntä pääsyä kaikkiin toiminnallisuuksiin, jotka saatetaan tarjota erillisten sovelluksien tai palveluiden kautta, ja jotka voivat tulla yrityksen sisältä tai sen ulkopuolelta. Sovelluksien ja palveluiden integrointi pysyy kuitenkin vielä haastavampana, kuin sen tarvitsisi olla. Kehittäjien tarvitsee taistella asynkronisuuden, osittaisten häiriöiden sekä yhteensopimattomien datamallien kanssa. (Hohpe & Woolf 2003.)

Yhä suurempi määrä järjestelmiä pyritään liittämään toisiinsa tiedonkulun sujuvuuden sekä yleisen käytettävyyden parantamiseksi. Järjestelmien keskeisen kommunikoinnin mahdollistamiseksi täytyy näiden järjestelmien ymmärtää toisiaan. Tämä ymmärrys saavutetaan integraatioilla, joiden tehtävänä on varmistaa, että jostakin lähdejärjestelmästä haettu data saapuu johonkin haluttuun kohdejärjestelmään oikeanmuotoisena ja ehjänä.

Integraatiokehityksessä on historiallisesti ollut kompastuskivenä yhdistettävien järjestelmien hyvinkin paljon toisistaan eroavat dataformaatit sekä toteutustyylit. Näitä ongelmia on pyritty ratkomaan erilaisilla integraatiotyökaluilla, joiden tavoitteina on yhtenäistää integraatioiden toteuttamistyylejä eriävien vaatimusten alla. Tämän työn tarkkailukohteena on yksi näistä lukuisista integraatiotyökaluista: tuote nimeltään Apache Camel.

Työssä perehdytään integraatiotekemiseen yleisellä tasolla oman integraatiototeutuksen kautta, sekä selvitetään kyseisen Camel-työkalun hyödyntämät ratkaisut aiemmin mainittujen integraatiokehityksen ongelmakohtien päihittämiseksi. Tutkielman tavoitteena on luoda vahva yleiskuva integraatiokehitykseen kuuluvista toimenpiteistä, sekä analysoida hieman tämän työkalun hyödyllisyyttä integraatiokehityksessä. Tutkielma tehtiin yhteistyössä Digia Oyj:n kanssa, ja varsinainen integraatio toteutettiin Tampereen korkeakouluyhteisölle (TUNI).

2 PROJEKTIN KESKEISET TEKNOLOGIAT

Integraatiokehityksessä voi tarpeiden mukaan joutua perehtymään todella moneenkin eri teknologiaan. Tarpeet tulevat lähde- ja kohdejärjestelmien hyödyntämistä lukuisista eri ratkaisuista. Myös tässä projektissa jouduttiin opettelemaan uusia teknologioita muun muassa kehitys- ja tuotantoympäristön ymmärtämiseksi, sekä integraatiokehityksen yleisen sujuvuuden takaamiseksi.

2.1 Docker

Docker on avoimen lähdekoodin moottori, joka hyödyntää käyttöjärjestelmätason virtualisointia eristääkseen applikaation sekä kaikki sen riippuvuudet omaksi paketikseen. Näitä paketteja kutsutaan *konteiksi*, ja niillä on useita hyötyjä perinteisiin toteutuksiin nähden. Tuottamalla koodia Docker-kontteihin voi ohjelmistoista tehdä käytännössä käyttöjärjestelmäriippumattomia sekä nopeasti jaettavia ja asennettavia kokonaisuuksia. (Docker overview n.d.)

Näiden hyötyjen lisäksi eräs vahva puoli tämmöisessä isolaatioparadigmassa on myös se, että lokaalissa kehitysympäristössä tuotetun ja testatun koodin voi myös olettaa toimivan testaus- sekä tuotantoympäristössä. Tämä nopeuttaa kehitystyötä huomattavasti, kun ympäristöjen välisistä eroista johtuvat ongelmatilanteet saadaan pidettyä minimissään. Dockerin sekä muiden vastaavien konttiteknologioiden suosio ohjelmistotuotannossa onkin viimeisten vuosien aikana kasvanut räjähdysmäisesti.

2.2 Java

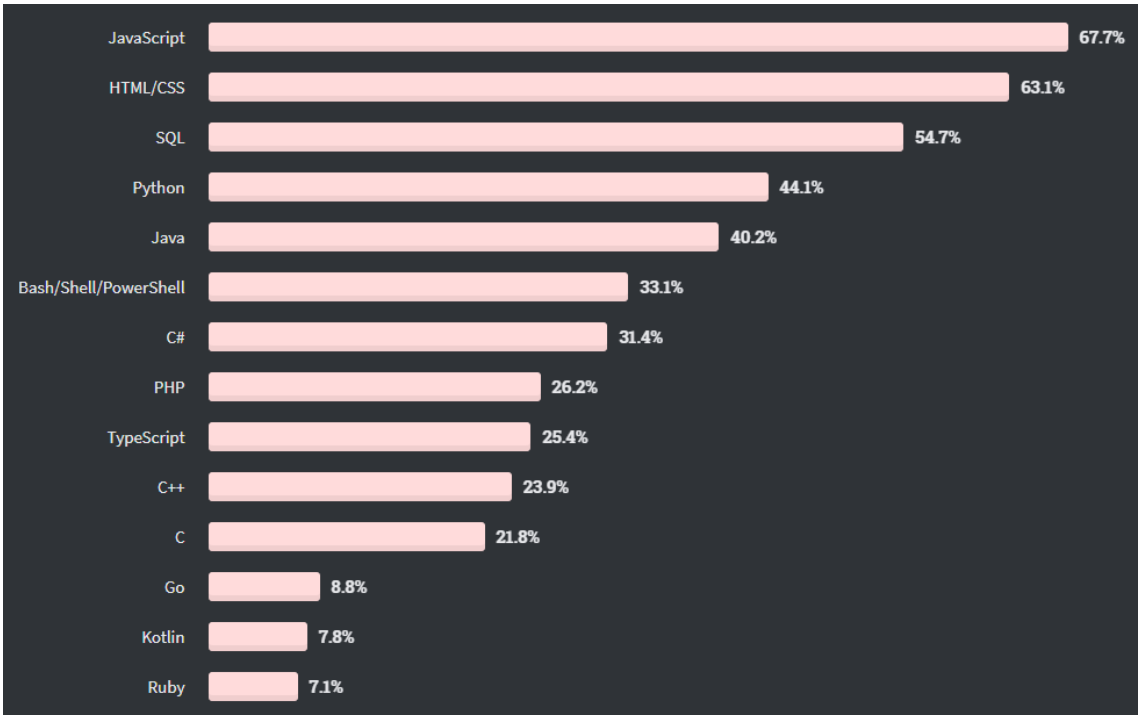
Java on alun perin Sun Microsystemsin vuonna 1995 julkaisema oliopohjainen ohjelmointikieli sekä tietojenkäsittelyalusta (Java n.d.). Java on vuosien saatossa kasvanut suosiossaan ja laajentunut ominaisuuksiltaan todella paljon, ja nykyään sitä käytetäänkin kaikenlaisien eri ohjelmien, kuten pelien, some-applikaatioiden

sekä ääni- ja videoapplikaatioiden ajamiseen. Oracle Corporation osti Sun Microsystemsin vuonna 2009, ja Oracle onkin tämän jälkeen toiminut Javan ylläpitäjänä (Oracle, 2009).

Javassa on paljon kehitystä sekä sen käytettävyyttä helpottavia ominaisuuksia JVM:n ansiosta. JVM tarkoittaa Javan virtuaalikonetta (Java Virtual Machine*)*, ja se on hyvin kriittinen komponentti Java-koodin suorittamista ajatellen. JVM-spesifikaation (2015) mukaan kaikki Java-koodi käännetään ensiksi Javan tavukoodiksi, jota sitten suoritetaan JVM:n avulla. Näin ollen JVM-toteutus vaaditaan kaikista laitteista, joissa Java-koodia halutaan ajaa. (JVM-spesifikaatio, 2015.) JVM:ään sisäänrakennetuista kehitystä auttavista ominaisuuksista merkityksellisimpiä on varmasti muistinhallintaa helpottava automaattinen roskienkeruu, joka pyyhkii mm. tarpeettomiksi tulleita muistivarauksia pois ja vapauttaa näin tilaa tuleville tarpeille.

Edellä mainitun ominaisuuden ansiosta, että Java-ohjelmat käännetään Javan tavukoodiksi ja niitä ajetaan virtuaalikoneen sisällä, on Java myös järjestelmäriippumaton alusta. Toisaalta hiekkalaatikkotyylinen ratkaisu tarjoaa Javalle myös vahvan turvallisuusaspektin, sillä virtuaalikoneessa ajetut ohjelmat ovat eristettynä muusta käyttöjärjestelmästä, eivätkä ne täten, ainakaan teoriassa, pääse luvattomasti käsiksi ulkopuolisiin ohjelmiin tai tietoihin. (Johari n.d.)

Java on perinteisesti nauttinut suuresta suosiosta kehittäjien keskuudessa, ja yksi hyvä mittari tälle on Stack Overflow:n vuosittain järjestämät kehittäjäkyselyt, joissa kysellään ekstensiivisesti mm. kehittäjien työnkuvista, käytettävistä ohjelmointikehyksistä sekä -kielistä, sekä muutamista henkilökohtaisista asioista. Vuoden 2020 tilastot liittyen käytettyihin ohjelmointikieliin löytyy seuraavan sivun kuvasta 1.



KUVA 1. Kehittäjien käyttämät ohjelmointikielet Stack Overflow:n vuoden 2020 kehittäjäkyselystä (Stack Overflow, 2020)

Kuvasta voi nähdä, että Java on viidenneksi käytetyin ohjelmointikieli kyselyyn vastanneiden keskuudessa vuonna 2020, yli 40 prosenttia kehittäjistä käyttäessä tätä kieltä. Suurin syy Javan tämänhetkiseen suosioon lienee Android-käyttöjärjestelmän valtava osuus älypuhelimissa käytetyistä käyttöjärjestelmistä, sillä Android pohjautuu Javaan. Java on toki myös perinteisissä sovellutuksissa vieläkin suosittu ja arvostettu, sillä kieltä päivitetään jatkuvasti. Muun muassa viime vuosikymmenenä lisätyt lambda-funktiot sekä stream-operaatiot ovat vieneet kieltä modernimpaan ja helppokäyttöisempään suuntaan. Javaa käytetään esimerkiksi huippusuositun suoratoistopalvelu Netflixin palveluiden ajamiseen (Baum n.d.).

2.3 MariaDB

MariaDB on yksi suosituimmista avoimen lähdekoodin relaatiotietokantajärjestelmistä, ja se pohjautuu MySQL-tietokantaohjelmistoon. Se on kehitetty alkuperäisten MySQL-kehittäjien toimesta. MariaDB on pyritty kehittämään tehokkaaksi, luotettavaksi, sekä helppokäyttöiseksi. Teknisesti ansioituneet henkilöt voivat

MariaDB:n open source -periaatteiden ansiosta tehdä omia kontribuutioita projektiin. MariaDB:tä käyttävät muun muassa Google, Wikipedia sekä Amazon Web Services. (Saive, 2020.)

Viimevuosien merkittävimpiä lisäyksiä MariaDB:hen on sen lisääntyneet klusterointimahdollisuudet Galera Cluster -teknologian avulla. Galera Cluster on tietokantareplikointiratkaisu, joka tähtää käytännössä välittömään, synkronoituun tietokantareplikointiin kytkettyjen tietokantojen välillä (MariaDB n.d.). Tietokantojen klusterointi on datan säilyvyyden ja eheyden sekä järjestelmän suorituskyvykkyyden kannalta todella tärkeä asia. Näiden parannuksien myötä ei siis ole ihmekään, että MariaDB on noussut erittäin houkuttelevaksi vaihtoehdoksi tietokantajärjestelmänä varsinkin integraatioalalla.

2.4 Apache Maven

Apache Maven on ohjelmistoprojektien hallintaan liittyvä työkalu, jota käytetään pääasiassa Java-projektien hallintaan ja käännöksiin. Mavenin päällimmäinen tavoite on helpottaa ja nopeuttaa kehittäjien ymmärrystä kehitettävän projektin senhetkisestä tilasta. Mavenissa on tämän lisäksi panostettu käännösprosessin suoraviivaisuuteen, ja se toimiikin myös käännösautomaatiotyökaluna. (What is Maven? n.d.)

Maven hyödyntää projektin rakentamisessa POM-nimistä (*project object model*) XML-pohjaista tiedostoa, johon tulee kirjata tietoa projektista ja tarvittavista konfiguraatioista tietyn syntaksin mukaisesti. Esimerkiksi projektin käyttämät riippuvuudet, kuten kolmannen osapuolen kirjastot, täytyy määritellä em. POM-tiedostoon.

2.5 OSGi-palvelualusta

OSGi (*Open Services Gateway initiative*) on palvelualusta, joka mahdollistaa Java-ohjelmistojen rakentamisen hyvin modulaarisella tyylillä. OSGi pyrkii modu-

laarisuudellaan sekä vähentämään ohjelmistojen monimutkaisuutta, että mahdollistamaan applikaatioiden ja palveluiden etähallintaa ja järjestelmäriippumattomuutta.

OSGi-palvelualustan avulla luodaan ja hallinnoidaan modulaarisia Java-komponentteja, joita kutsutaan *bundleiksi*. Kehittäjät voivat OSGi-spesifikaatiota ja -työkaluja hyödyntäen luoda yhden tai useamman bundlen. OSGi määrittelee itse elinkaaren näille bundleille tarpeiden mukaan ja isännöi näitä bundleja kontin sisällä. Hallitsemalla itse bundlejen säilytystä voi OSGi varmistua siitä, että jokainen komponentti on riittävän eristettynä, mutta omaa myös pääsyn kaikkiin tarvittaviin riippuvuuksiin. (Tyson, 2020.)

2.6 Apache Karaf

Apache Karaf on käytännössä OSGi-ympäristö, joka tarjoaa OSGi-toiminnallisuuksien lisäksi hieman käyttöönottoa ja hallintaa helpottavia ominaisuuksia. Karaf on kehitetty OSGi-palvelualustan standardi-implementaation päälle, mutta varsinaista OSGi-tietämystä ei tarvita Karafia käytettäessä. Tämän OSGi-pohjaisuuden sekä lisäominaisuuksiensa ansiosta Karaf on erittäin joustava ratkaisu esimerkiksi integraatiotarpeisiin. (Karaf-manuaali n.d.)

Karaf tarjoaa lisäominaisuuksina muun muassa oman terminaalin, oman keskitetyn lokitusjärjestelmän, etäpääsyn terminaaliin SSH-tunnistautumisen avulla, sekä kansiosiirron kautta toimivan ns. *hot deploy* -ominaisuuden, jolla saa asennettua ja päivitettyä ajettavia ohjelmia suoraan Karafiin pelkästään siirtämällä ajotiedostot *deploy*-kansion sisälle. (Karaf-manuaali n.d.)

2.7 Apache Camel

Apache Camel on hyvin monipuolinen avoimen lähdekoodin integraatio-ohjelmistokehys, joka pyrkii tekemään integraatiokehityksestä helpompaa ja saavutettavampaa kehittäjille. Pohjimmiltaan Camel perustuu Enterprise Integration Patternien toteuttamiseen. *Enterprise Integration Patterns* (2003) on Gregor Hohpen ja

Bobby Woolfin kirjoittama, paljon integraatioalalla tunnustusta saanut kirja, joka kuvaa teknologiariippumattomasti 65 erilaista integraatiokaavaa. Nämä kaavat esittävät yleisiin integraatio-ongelmiin ratkaisuja sekä erilaisia design-vaihtoehtoja. (Hohpe & Woolf, 2003.)

Camel on kehitetty niin, että reititys- ja välityssääntöjä voi kirjoittaa montaa eri DSL:ää käyttäen. Camelin suosituimmat kielet ovat Java-pohjainen Fluent API, sekä Blueprint ja Spring XML-pohjaiset kielet. Blueprint-spesifikaatio kuuluu OSGi-palvelualustalle, eli toisin sanoen Blueprint DSL:llä voi kirjoittaa suoraan Camel-reittejä OSGi-bundleiksi. Spring DSL:llä hallitaan Spring Framework -ohjelmistokehyksen komponentteja. (Camel DSL n.d.)

Camel käyttää ekstensiivisesti myös URI-merkkijonoja kytkeytyäkseen suoraan melkeinpä mihin tahansa kuljetus- tai viestintämalliin, kuten HTTP:hen (What is Camel? n.d.). Cameliin voi tämän lisäksi määritellä omia komponentteja tai dataformaatteja.

Java DSL:ää käytettäessä Camel tukee vahvasti JavaBeanien käyttöä antamalla työkaluja sekä käytettävien metodien määrittelemiseen että viestin käsittelyn ohjaukseen. JavaBeanit ovat eräänlaisia tietyn syntaksin ja sääntöjen mukaan luotuja Java-luokkia. JavaBeaneihin voi Camelissa siis määritellä, mitä osaa viestistä halutaan käyttää, ja minkälaiseen formaattiin se mahdollisesti konvertoidaan. Camel tarjoaa em. etujen lisäksi myös monipuolisen tuen toteutettujen reittien yksikkötestaamiseen.

2.8 Enterprise Service Bus

Enterprise service bus, lyhyemmin *ESB*, on eräänlainen yrityskäyttöön soveltuvan palveluväylän malli, jossa keskitetty ohjelmistokomponentti kykenee toteuttamaan integraatioita palvelinpuolen (*backend*) järjestelmiin. Tähän ESB-järjestelmän mahdollistamaan integraatiokyvykkyyteen kuuluu muun muassa datamallien kääntäminen, reitittäminen, sekä pyyntöjen teko. ESB-järjestelmät kykenevät myös luomaan näille integraatioille yksinkertaisia ja helppokäyttöisiä palveluliittymiä, joita muut sovellukset voivat käyttää. (IBM, 2019.)

ESB-järjestelmissä on yleensä näiden edellä mainittujen toimintojen lisäksi sisäänrakennettuna erilaisia analytiikkaominaisuuksia, joiden tavoitteina on auttaa havainnoimaan yrityksen palveluiden läpi kulkevaa dataa sekä osoittaa mahdollisia kehityskohteita.

ESB-järjestelmät ovat tyypillisesti rakennettu valmiiksi kokonaisuuksiksi, joten näiden järjestelmien komponenttien vaihtaminen toisiin ei yleensä onnistu lainkaan ilman suurta vaivannäköä ja työtä. Tästä johtuen ESB-järjestelmät voivat olla hyvinkin joustamattomia ja vaatia potentiaalisesti paljonkin opetteluaikaa, mikäli järjestelmän käyttämät komponentit eivät ole kehittäjille entuudestaan tuttuja. Myös Apache tarjoaa oman ESB-järjestelmän, nimeltään Apache ServiceMix.

2.9 Apache ServiceMix

Apachen tarjoama yrityskäyttöön valmis palveluväylä, ServiceMix, on avoimen lähdekoodin ESB-järjestelmä. ServiceMix yhdistää neljä keskeistä Apachen integraatiotuotetta yhdeksi voimakkaaksi palvelualustaksi, jonka avulla voi kehittää omia integraatioratkaisuja. Nämä ServiceMixin sisältämät tuotteet ovat Apachen Camel- ja Karaf-teknologioiden lisäksi myös CXF- sekä ActiveMQ-teknologiat. (ServiceMix n.d.)

CXF on eräänlainen avoimen lähdekoodin ohjelmistokehys palveluiden kehitystä varten, kun taas ActiveMQ on Java-pohjainen viestien välityspalvelin sovellusten väliseen kommunikointiin. CXF- ja ActiveMQ-teknologiat eivät suoranaisesti liity tähän työhön, mutta Camelin sekä Karafin kuuluminen ServiceMix-palveluväylään toisaalta tarkoittaa sitä, että näiden kahden työkalun välillä on vahvasti integroituja ominaisuuksia. Esimerkkinä tällaisesta integroidusta ominaisuudesta on integraatioiden perustason monitorointi Karafissa. Cameliin on hyvinkin helppo ja nopea kirjoittaa mm. tärkeitä työvaiheita lokimuotoon, jotka Karaf pystyy siististi näyttämään omasta konsolinäkymästään.

3 MIKSI INTEGRAATIOTYÖKALUJA TARVITAAN

3.1 Integraatiokehityksen ongelmat

Kuten tämän tutkielman johdannossa mainittiin, integraatiokehityksessä on historiallisesti ollut ongelmakohtana sovellettujen ratkaisujen monimuotoisuus. Yhdistettävien palveluiden erilaisuudet ja eriävät tarpeet ovat luoneet tilanteen, jossa on tapauskohtaisesti räätälöity uniikkeja integraatioratkaisuja tarvitseville osapuolille. Tämä lähestymistapa on saanut aikaan sen, että sovellettuja arkkitehtuureja ei ole juurikaan voinut hyödyntää muissa projekteissa muiden tahojen kohdalla, vaan uusien integraatioiden toteuttamista on jouduttu aloittamaan hyvinkin pitkälti alusta asti.

Tämän ongelmakohdan selättämiseksi kaksi IT-alan asiantuntijaa, Gregor Hohpe & Bobby Woolf, lähtivät kasaamaan eräänlaista kaaviokokoelmaa yleisimmistä integraatiotarpeista sekä -tilanteista, tavoitteenaan yhtenäistää integraatioalan menetelmiä ja toteutustyylejä. Tämä yhtenäistämistyö kantoi hedelmää vuonna 2003 julkaistun Enterprise Integration Patterns -kirjan muodossa. Kirjasta ja siinä puhutuista integraatiokaavoista tuli nopeasti integraatioalan referoiduimpia teoksia. Kirjassa on esitetty 65 erilaista integraatiotilannetta eräänlaista kaaviokieltä käyttäen. Kirja ei siis määrää tai rajaa käytettäviä ohjelmointikieliä tai työkaluja, vaan se yksinkertaisesti ohjeistaa, kuinka tällaisissa yleisissä, toistuvissa integraatiotilanteissa voidaan ratkaisuja muodostaa. Monet integraatioalalle tehdyt työkalut pohjautuvatkin juuri näihin Hohpen ja Woolfin kirjaamiin integraatiokaavoihin. Yksi näistä tällaisista työkaluista on juuri Apache Camel.

3.2 Apache Camel ratkaisuna ongelmiin?

Apache Camel on yksi näistä Hohpen ja Woolfin em. Enterprise Integration Patterneja hyödyntävistä ja toteuttavista työkaluista, joten Camel on myös eräänlainen ratkaisu tähän historialliseen integraatioalan pulmaan. Lähimenneisyydessä suosiota nauttineet Enterprise service bus (ESB) -integraatiomallit, jotka toimivat

eräänlaisina yrityksen palveluväylinä, ovat myös tähdänneet yhdistämään integraatioalan levällään olevia toimintamalleja ja -menetelmiä, mutta itsenäisissä avoimen lähdekoodin integraatioratkaisuissa on kuitenkin ehdottomasti omat hyvät puolensa.

Pitkään väliohjelmistojen kehittäjänä toiminut Anton Goncharov esimerkiksi kertoo Camelin hyödyistä avaavassa nettiartikkelissaan, että ESB:eitä on liki mahdoton ottaa nopeasti käyttöön, vaan ne vaativat pitkän opetteluajan, sekä niiden joustavuus on rajoitettua sisäänrakennettujen työkalujen ja ominaisuuksien vuoksi. Goncharov jakaa samassa artikkelissa mielipiteensä siitä, kuinka kevyet avoimen lähdekoodin integraatioratkaisut ovat hänen mielestään näihin ESB:eihin verrattuna paljon parempia vaihtoehtoja paremman skaalautuvuuden ja joustavuuden takia. (Goncharov n.d.)

Goncharov myös täsmentää kirjoittamassaan artikkelissaan Camelin hyvistä puolista poimimalla omasta mielestään kaksi hyödyllisintä ja tärkeintä ominaisuutta, jotka löytyvät Camelista. Ensimmäinen korostettu ominaisuus koskee Camelin luontaista mallia kirjoittaa reittejä. Integraatioreitit kirjoitetaan Camelissa ikään kuin eri palasista koostuviksi kiinteiksi putkistoiksi, jonka ansiosta datan kulkemista on erittäin helppo havainnoida sekä seurata. Toinen Goncharovin mainitsemista hyödyistä on Camelin suora soveltuvuus moniin erilaisiin suosittuihin rajapintoihin. Esimerkiksi datan haku Apache Kafka -tiedonprosessointialustalta, Amazonin AWS EC2 -pilvipalvelujen monitorointi, sekä integrointi Salesforce-ohjelmistoihin kaikki onnistuvat ns. out of the box, eli hyödyntämällä pelkästään Camelin mukana tulevia komponentteja ja adaptereita. (Goncharov n.d.)

Tietotekniikan moniosaaja, mm. JavaOne sekä ApacheCon -konferensseissa puhunut blogikirjoittaja Kai Wähner kirjoittaa tietoteknisiä artikkeleita julkaisevan blogisivuston DZone:n artikkelissaan ”When to Use Apache Camel?” hyvin ammattimaisesti Camelin vahvuuksista ja heikkouksista. Päällimmäisenä hän kirjoittaa blogissaan arvostavansa Camelin yhdenmukaisuutta, sillä valituista teknologioista tai ohjelmointikielistä riippumatta Camelin idea pysyy samana: Camel-integraatioissa on aina olemassa tuottaja (producer), kuluttaja (consumer), päätepisteitä, EIP-kaavioiden toteutusta, sekä mahdollisia lisätoiminnallisuuksia (kustomoidut prosessorit, parametrit ym.). Tämän yhdenmukaisuuden lisäksi Wähner

mainitsee muina tärkeinä Camelin ominaisuuksina tuen virheenkäsittelylle sekä automaatiotestaukselle, ja ylistää samalla Camelin laajennusta JUnit-testikehyksestä muun muassa sen helppokäyttöisyydestä. Lopuksi Wähner listaa artikkelissa vielä Camelin muita lukuisia hyötyjä, kuten sen valmiutta tuotantokäyttöön, skaalautuvuutta, transaktiotukea sekä monitorointimahdollisuuksia. (Wähner, 2011.)

Vastakohtana Kai Wähner kirjoittaa samassa blogikirjoituksessa myös tilanteista, joissa Camelia ei hänen suositustensa mukaan kannata käyttää. Hän mainitsee muun muassa tapauksista, joissa integroitavien järjestelmien määrä tai laajuus on hyvin pieni, sekä tapauksista, joissa jo heti alussa tiedetään, että integraatioprojekti tulee olemaan valtava skaalaltaan. Wähner huomauttaa myös, että kummassakaan näistä skenaarioista ei ole varsinaisesti kyse Camelin soveltumattomuudesta, vaan pikemminkin muiden, tarkasteltavan integraatioprojektin skaalaan paremmin sopivien teknologioiden olemassaolosta. Hänen mukaansa todella pieniin projekteihin ei ole järkeä lähteä opettelemaan Camelin saloja, joka on varmasti paikkansapitävä toteamus. Pienempien projektien kohdalla on fiksumpaa käyttää tunnettuja kolmannen osapuolen kirjastoja kuten Apache Commons IO:ta sen sijaan, että lähtee opettelemaan alusta asti kokonaisen työkalun käyttöä. (Wähner, 2011.)

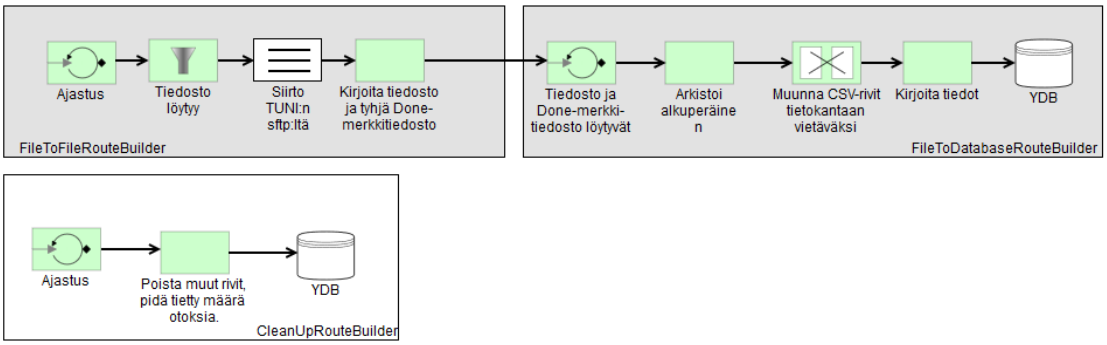
Isojen projektien kohdalla tämä tilanne on päinvastainen; Wähner suosittelee valtavankokoisille projekteille kevyiden Camelin kaltaisten integraatioratkaisuiden sijaan raskaita ESB-toteutuksia. ESB:eiden käyttöönotto on hieman jähmeää ja hidasta mikäli aikaisempaa kokemusta käyttöönotettavasta ESB-tuotteesta ei ole. ESB:t kuitenkin maksavat itsensä takaisin pitkässä juoksussa, sillä niiden graafiset käyttöliittymäominaisuudet sekä monitorointimahdollisuudet helpottavat ja nopeuttavat työnkulkua huomattavastikin työkalun käytön hallitseville kehittäjille. (Wähner, 2011.)

Aikaisemmissa kappaleissa referoitu Kai Wähnerin teksti on toki jo suhteellisen vanhaa, mutta tilanne integraatioalan työkalujen välillä ei ole muuttunut mitenkään dramaattisesti. Camel on edelleen hyvä valinta keskikokoisille, kevyttä ratkaisua hakeville projekteille, kun taas ESB-järjestelmien kehittyessä ja kasvaessa ne ovat entistäkin sopivimpia isoihin projekteihin. Vaikka edellä mainittuihin

tiettyihin tilanteisiin on olemassa parempiakin ratkaisuja kuin Apachen Camel-työkalu, ei se silti tarkoita sitä, etteikö myös Camelia voisi käyttää näissä tilanteissa. Joissakin tapauksissa tämä voi olla jopa suotavaa, mikäli Camelin käyttö tuo muita hyötyjä. Esimerkiksi osaamisen kartuttaminen, tai avoimen lähdekoodin työkalujen suosiminen voivat olla syitä pysyttäytyä Camelissa.

4 INTEGRAATION TOTEUTTAMINEN APACHE CAMELILLA

Tätä tutkielmaa varten toteutettiin yksi integraatio alusta alkaen. Tämän integraation tehtävänä oli lukea opiskelijakortteja personoivan yrityksen, Antenna Oy:n, tuottama CSV-data, ja siirtää tämä data integraatiopalvelun välimuistitietokantaan, eli YDB:hen. Siirrettyä dataa oli tarkoitus hyödyntää välimuistitietokannasta mm. kulunvalvonnan integraatioissa. Integraatio tehtiin Tampereen korkeakouluyhteisölle. Työnkulkuun kuului myös integraation EIP-kaavion luominen ja sen lisääminen integraation wikisivuille. Tämän Antenna-integraation EIP-kaavio löytyy kuviosta 1.



KUVIO 1. Antenna-integraation EIP-kaavio

Kuvion EIP-kaaviossa on määriteltyinä kaikkien kolmen käytetyn reitin toimintavaiheet, mutta kaksi noista käytetyistä reiteistä (FileToFileRouteBuilder, FileToDatabaseRouteBuilder) olivat TUNI-integraatiopalvelun parent-projektista tulevia yleiskäyttöisiä reittejä, jotka vaativat käytännössä vain parametrien konfiguroinnin. Tänne parent-projektiin oli toteutettu monia muitakin TUNI-integraatiopalvelun integraatioissa toistuvia komponentteja, kuten virheenkäsittelyluokkia, joita pystyi ottamaan käyttöön sisällyttämällä parent-projektin toteutettavaan integraatioprojektiin, ja konfiguroimalla tarvittavat parametrit käytetylle komponentille. Tässä työssä parent-projektista hyödynnettiin pelkästään näitä em. yleiskäyttöisiä reittejä. Kuvion FileToFileRouteBuilder-reitin kuvauksessa mainitulla sftp:llä tarkoitetaan tässä TUNI-integraationpalvelun tiedostopalvelinta.

4.1 Kehitysympäristön tarpeet

Camel tarjoaa integraatiokehitykseen useita eri DSL-vaihtoehtoja, joten kehittäjät voivat hyödyntää projekteissa omia haluamiaan integroituja ohjelmointiympäristöjä (IDEä). Etuja IDEn käytössä ovat muun muassa tiedostorakenteiden ja navigoinnin selkeys, sekä älykäs täydennys kirjoitettavien reittien ja luokkatiedostojen kohdalla. Esimerkiksi jos Camelin Java DSL:ää käytettäessä hyödyntää jotakin Javaa tukevaa IDEä, niin ohjelmointiympäristö voi auttaa syntaksin ja avainsanojen kanssa, kunhan tarvittavat Apachen kirjastot ovat sisällytettynä projektiin. Ohjelmointiympäristöksi valittiin tässä työssä JetBrainsin kehittämä IntelliJ IDEA -tuote, sillä koko muu kehitystiimi käytti tätä kyseistä IDEä.

Ennen kuin varsinaiseen kehitykseen pääsi uppoutumaan, tarvitsi IDEn lataamisen ja tutustumisen lisäksi vielä perehtyä myös projektin kehitysympäristöön. Kehitysympäristön kannalta tärkeimmässä asemassa oli Docker, sillä koko kehitysympäristöä ajettiin Docker-konttien sisällä. Docker on onneksi suhteellisen suoraviivainen ja helppokäyttöinen palvelu, joten projektin Wikiin kirjoitettuja asennusohjeita seuraamalla pääsi jo todella pitkälle tietämättä Dockerista juurikaan mitään entuudestaan.

Dockerin käyttöönottoa ja kehitysympäristön asennusta helpotti projektissa hyödynnetty Docker Compose -työkalu. Compose on työkalu, jolla voidaan määritellä ja ajaa monikonttisia Docker-applikaatioita. Tämä tapahtuu käyttämällä docker-compose-nimistä YAML-konfiguraatiotiedostoa applikaatiossa tarvittavien palvelujen määrittelemiseen. Composen konfiguraatiotiedoston lisäksi tarvittiin myös *Dockerfile*, jonka sisälle määritellään applikaation pystyttämiseen tarvittavat komennot. (Compose-dokumentaatio n.d.)

Docker ajaa nämä komennot joka kerta kontteja pystyttäessä, mikä säästää huomattavasti aikaa ja päänvaivaa, kun komentoja ei tarvitse manuaalisesti yksi kerrallaan kirjoitella komentokehotteeseen. Seuraavan sivun kuvasta 2 löytyy kuvankaappaus projektissa käytetyn MariaDB-näköistiedoston määrittelystä Docker Composen sisällä.

```
db:
  container_name: mariadb
  image: mariadb:10.2.8
  ports:
    - "3306:3306"
  environment:
    MYSQL_ROOT_PASSWORD: ******
  volumes:
    - ./config:/etc/mysql/conf.d
    - ./db:/var/lib/mysql
```

KUVA 2. Docker Compose -tiedostosta löytyvä MariaDB:n konfiguraatio

Kuten kuvasta 2 voi nähdä, Composeen määritellään muun muassa halutun näköistiedoston versionumero sekä käytettävä portti Dockerin sisällä. Dockerfilen sekä Docker Compose -tiedoston ollessa konfiguroituina siirryttiin seuraavaan vaiheeseen, eli Docker-konttien käynnistämiseen. Kehitysympäristö käynnistettiin kirjoittamalla käyttöjärjestelmän komentokehotteeseen (tässä työssä Windows PowerShell) Dockerin juurihakemistossa avainsanat ”docker-compose up” (kuva 3).

```
PS C:\tuniproj\t3itg_docker_devel_env> docker-compose up -d
Creating network "t3itg_docker_devel_env_default" with the default driver
Creating mariadb                      ... done
Creating activemq                     ... done
Creating sftpin                       ... done
Creating t3itg_docker_devel_env_smtp_1 ... done
Creating karaf                        ... done
PS C:\tuniproj\t3itg_docker_devel_env>
```

KUVA 3. Docker-konttien käynnistys Compose-työkalua käyttäen

Kuvassa näkyvillä olevaan käynnistyskomentoon lisätty -d-lippu tarkoittaa *detached*-tilaa, eli kyseisellä lipulla Docker Composea käynnistäessä kontit jätetään ajoon taustalle. Tässä tilassa kontit ajetaan alas pelkästään käyttäjän manuaalisesta syötteestä.

Kun näitä Composella käynnistettyjä Docker-kontteja ei tarvita enää, tai kun ne halutaan ajaa jostain muusta syystä alas, kirjoitetaan komentokehotteeseen ”docker-compose down” seuraavan sivun kuvan 4 tapaisesti.

```
PS C:\tuniproj\t3itg_docker_devel_env> docker-compose down
Stopping karaf                          ... done
Stopping sftpin                         ... done
Stopping mariadb                        ... done
Stopping activemq                       ... done
Stopping t3itg_docker_devel_env_smtp_1 ... done
Removing karaf                          ... done
Removing sftpin                         ... done
Removing mariadb                        ... done
Removing activemq                       ... done
Removing t3itg_docker_devel_env_smtp_1 ... done
Removing network t3itg_docker_devel_env_default
PS C:\tuniproj\t3itg_docker_devel_env>
```

KUVA 4. Docker-konttien synkronoitu alasajo Compose-työkalua käyttäen

Kuten kuvassa 4 näkyvästä tulosteesta voi nähdä, kontille ominaiset näköistiedostot ajetaan rauhanomaisesti alas ennen kuin ne poistetaan konttien sisältä. Sitä mukaan, kun kontteja tarvitsi käyttää uudestaan, tehtiin käynnistysprosessi alusta kuvan 3 käynnistyskomentoa hyödyntäen.

Projektin kehityksessä käytettiin ohjelmointikielenä Javaa, joten kehitysympäristön toimivuuden kannalta täytyi asentaa myös oikeanlainen Java-versio. Java-versioita on nykyään lukuisia, ja niitä voi selailla sekä ladata Oraclen omilta latussivuilta. Kaikkien Java-pakettien mukana tulee tarvittavat riippuvuudet (kuten JVM) Java-applikaatioiden ajamiseen, mutta esimerkiksi JDK-paketeissa (*Java SE Development Kit*) tulee tarvittavien riippuvuuksien lisäksi myös erilaisia hyödyllisiä kehittäjätyökaluja debuggaamista ynnä muuta varten. Tampereen korkeakouluyhteisön integraatiopalvelussa käytettiin Javan versiota 8, joten yhteensopivuusongelmien välttämiseksi myös tässä työssä otettiin käyttöön kyseinen versio.

Tarvittavien työkalujen asennusten sekä konfigurointien jälkeen alkoi työn varsinainen toteutusvaihe. Työ aloitettiin yksinkertaisesti luomalla uusi Apache Maven-projekti IntelliJ IDEAan, ja täydentämällä projektirunkoa tarvittavilla tiedostoilla. Tarvittavia tiedostoja olivat muun muassa Blueprint-tiedosto OSGi-bundlejen luontia varten, POM-tiedosto Mavenin konfigurointia varten sekä erillinen konfiguraatiotiedosto tiettyjä muuttujia ja parametrejä varten. Nämä tiedostot olivat

jokaisessa integraatiopalvelun integraatiossa läsnä, joten mallia näiden tiedostojen alullepanoon pystyi ottamaan projektin versionhallinnasta löytyvistä muista integraatioista.

4.2 OSGi-bundlejen määrittely

Kaikki integraation ajamiseen tarpeelliset komponentit määriteltiin Blueprint-tiedostoon Mavenia varten, jotta projektin tiedostoista pystyttiin oikeaoppisesti kasaamaan ajettava OSGi-bundle käännöksen myötä. Camel käyttää tähän määrittelyyn omaa kustomoitua XML-nimiavaruutta. Määrittely on tehty syntaktisesti hyvin yksinkertaiseksi. Esimerkiksi kirjoitetut JavaBeanit määritellään Blueprint-tiedostoon käyttäen *bean*-avainsanaa, ja antamalla tälle JavaBeanille tarpeelliset parametrit sekä muuttujat. Kuvassa 5 on nähtävillä esimerkki yhden integraatiossa käytetyn JavaBeanin määrittelystä.

```
<bean id="ydbDao" class="fi.tampere3.it.itg.antenna_ydb_opiskelijakortit.dao.YdbDao">
    <argument ref="jdbcTemplate"/>
    <property name="keepBatches" value="${keep.old.batches}"/>
</bean>
```

KUVA 5. Data access object -luokan määrittely Blueprint-tiedostossa

Kuvassa näkyvä määrittely koskee integraation data access object (DAO) -luokkaa. DAO on suunnittelumalli, jonka tarkoituksena on auttaa mm. tietokantaoperaatioiden tekemisessä vaarantamatta tietokannan turvallisuuden kannalta tärkeää tai herkkää tietoa. Tässä työssä tietokantaoperaatiot DAO-luokassa toteutettiin JDBC:n avulla. JDBC, koko nimeltään *Java Database Connectivity*, on Java-rajapinta, jonka avulla pystytään luomaan yhteys haluttuun tietokantaan sekä tekemään sinne tietokantaoperaatioita (Javatpoint n.d.). Tämän em. DAO-luokan määrittelyssä on erikseen annettu argumentti JDBC:n käyttöä varten, sekä nimetty property tietokantaerien säilytystä varten. Property on tässä yhteydessä eräänlainen ko. luokassa hyödynnetty muuttuja. Tämän luokan Blueprint-määrittelyn kohdalla propertyn arvo tulee konfiguraatiotiedostosta, siksi arvo on kirjoitettu $-merkin ja kaarisulkujen sisään.

Blueprint-tiedostoon määritellään myös Camelissa käytettävä konteksti, joka kapseloi sisäänsä tämän kontekstin alle määritellyt reitit. Reittien kapselointi tällä tavalla tuo mukanaan muutamia integraation hallinnointia helpottavia etuja. Camel-konteksteja hyödynnetään muun muassa integraatioiden elinkaarien ohjauksessa, sillä esimerkiksi saman kontekstin alla olevat reitit pystytään pysäyttämään yhdellä stop-komennolla (Camel Lifecycle n.d.). Kuvassa 6 näkyy tämän työn Blueprint-tiedostoon määritelty Camelin konteksti.

```
<camelContext id="antenna_ydb_opiskelijakortit-ctx" xmlns="http://camel.apache.org/schema/blueprint"
    <routeBuilder ref="downloadAntennaStudentDataRouteBuilder"/>
    <routeBuilder ref="updateAntennaStudentToDatabaseRouteBuilder"/>
    <routeBuilder ref="batchCleanUpRouteBuilder"/>
</camelContext>
```

KUVA 6. Camel-kontekstin määrittely Blueprint-tiedostossa

Kuvassa näkyvät referenssit routeBuilder-entiteetteihin sisältävät kaikki tässä työssä käytetyt reitit. Toisin sanoen kaikki tähän työhön toteutetut reitit kuuluvat tämän yhden Camel-kontekstin alle. Camelin 2.x-versioissa on mahdollista käyttää useampaa Camel-kontekstia yhden asennettavan integraation sisällä, mutta Camelin GitHub-migraatio-ohjeiden (2019) mukaan tämä tuki on poistettu Camel 3.x -versioissa. Camel 3.0:ssa ja siitä eteenpäin on siis suositeltua käyttää vain yhtä kontekstia per asennettava integraatio. (Apache Camel Migration Guide, 2019.)

4.3 Maven-riippuvuuksien määrittely

Mavenin toiminnan kannalta kriittisin kohta on Mavenin hyödyntämä POM-tiedosto. Tämä XML-formaatin tiedosto sisältää kaiken Mavenin tarvitseman tiedon projektista ja sen konfiguroinnista, joiden avulla Maven rakentaa projektin. Jos projektiin lisäillään joitain kolmansien osapuolten kirjastoja, täytyy näiden kirjastojen riippuvuudet muistaa lisätä pom.xml-tiedostoon, tai muuten Maven ei osaa hakea projektiin tarvittavia riippuvuuksia rakennusvaiheessa.

Tässä työssä POM-tiedoston konfigurointi oli sangen suoraviivaista; kaikissa integraatioprojektin integraatioissa käytettiin likimain samoja kirjastoja sekä riippuvuuksia, ja uusien kolmansien osapuolten kirjastojen lisääminen projektiin olikin

kiellettyä ilman erillistä hyväksyntää. POM-tiedoston konfigurointi aloitettiin itse POM-tiedoston ominaisuuksista, kuten merkistökoodauksen menetelmän ja POM-version määrittelyillä. Näiden jälkeen POM-tiedostoon täytyi määritellä parent-projektin tiedot, sillä kuten luvun neljä pääotsikon alla mainittiin, työssä käytettiin komponentteja, jotka löytyivät vain tästä parent-projektista. Näitä yleiskäyttöisiä komponentteja pystyi käyttämään sisällyttämällä tämä parent-projekti työstetyn projektin POM-tiedostoon. Kuvassa 7 on nähtävillä tämän projektin POM-tiedoston ensimmäisten rivien määrittelyt.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>fi.tampere3.it.itg</groupId>
        <artifactId>t3-itg-parent</artifactId>
        <version>1.6.2</version>
    </parent>

    <artifactId>antenna_ydb_opiskelijakortit</artifactId>
    <version>1.0.5-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>T3 antenna_ydb_opiskelijakortit integration</name>
```

KUVA 7. Projektin POM-tiedoston alkupäästä löytyvää määrittelytietoa

Edellä mainittujen parent-projektin ym. määrittelyjen lisäksi kuvassa näkyy parent-lohkon alla myös *artifactId*, *version* ja *packaging* -nimillä varustettuja tietokenttiä integraatioon liittyen. Nämä kentät auttavat Mavenia rakentamaan tiedostolle oikeanlaisen metadatan, kuten versionumeron sekä nimen, ja määrittelevät tiedoston paketointitavan.

Seuraavalta sivulta kuvasta 8 löytyy Apache Camel -integraatioille hyvin tyypillisiä riippuvuuksia, kuten pohjatoiminnallisuuksia tarjoava *camel-core*-moduuli, joka mahdollistaa mm. Javan DSL:n käytön (Camel Core n.d.).

```
<dependencies>
    <!-- Camel -->
    <dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-core</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-blueprint</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-sql</artifactId>
    </dependency>
```

KUVA 8. Muutamia projektin riippuvuuksia määriteltynä POM-tiedostossa

Muita kuvassa näkyviä riippuvuuksia camel-coren lisäksi ovat camel-blueprint sekä camel-sql -moduulit. Blueprint-moduuli tuo mukanaan tarvittavat rajapinnat OSGi-bundlejen hallinnointiin, ja SQL-moduuli tuo projektiin tarvittavat JDBC-työkalut tietokantakyselyjen tekemistä varten.

4.4 Reittien kirjoittaminen

Camelin reitit ovat järjestyksessä ajettavia toimintosarjoja, jotka kuluttavat ja prosessoivat sanomia. Camel-reitti alkaa aina kuluttajalla ja sitä seuraa päätepisteiden sekä mahdollisten prosessoreiden ketju. (Donohue n.d.)

Reittien yleisenä päätehtävänä on siirtää dataa pisteestä A pisteeseen B. Camel-reittien matkalle voi kirjoittaa omaa logiikkaa hyödyntäviä apuluokkia tai prosessoreita, jos esimerkiksi lähdejärjestelmästä tulevan datan formaattia täytyy jollain tavalla muokata kohdejärjestelmään sopivaksi. Tässä työssä reittejä tarvittiin kolme kappaletta, kolmeen eri käyttötarkoitukseen.

Yhden reitin tuli hoitaa tarvittavien CSV-tiedostojen siirto Antennan palvelimilta TUNI-palvelimille, toinen reitti tarvittiin siirtämään CSV-tiedostoissa ollut data TUNI-integraatiokehityksen välimuistitietokantaan ja kolmas reitti piti luoda vanhan datan säännölliseen ja täsmälliseen poistoon. Näistä reiteistä kahden ensimmäisen kohdalla pystyttiin hyödyntämään TUNI-integraatiokehityksen parent-projektia, jossa oli määriteltynä valmiit pohjat tämmöisille peruslaatuisille tiedostojärjestelmien välisille siirroille sekä välimuistitietokantasiirroille. Nämä valmisreitit saatiin valjastettua käyttöön yksinkertaisesti määrittelemällä tarvittavat konfiguraatioparametrit näille reiteille projektin OSGi Blueprint -tiedostoon.

Kolmannen reitin kohdalla toteutus jouduttiin kuitenkin tekemään itse. Tietokannan vanhan datan siivoamisen kohdalla määrittely oli tälle integraatiolle vastaavanlainen: kaikki muut tätä integraatiota koskevat tietokannan erät, paitsi kolme viimeisintä, piti poistaa päivittäin tiettynä ajanhetkenä. Poistoajankohta valittiin senhetkisen integraatiopalvelun kuormitustasojen perusteella, eli päivittäiseksi poistoajankohdaksi valittiin noin keskiyö. Ajankohta määriteltiin integraatioon ladattavaan konfiguraatiotiedostoon cron-muodossa. Cron on eräänlainen Linux-järjestelmille tyypillinen ajastusmekanismi, jolla voidaan ajastaa tehtäviä suoritettavaksi haluttuihin kellonaikoihin. Cron toimi tälle kolmannelle reitille laukaisevana tekijänä, eli siivoustoimenpiteisiin ryhdyttiin joka päivä konfiguraatiossa määriteltynä ajankohtana. Siivousreitin voi nähdä kokonaisuudessaan seuraavalta sivulta (kuva 9).

```
import com.digia.camel.component.iiris.IirisProducer;
import org.apache.camel.builder.RouteBuilder;

public class BatchCleanUpRouteBuilder extends RouteBuilder {

    @Override
    public void configure() {

        onException(Exception.class)
                .to("iiris:Failure?info=${exception.stacktrace}")
                .markRollbackOnly();

        // Ajastettu reitti, joka poistaa vanhat erät tietokannasta
        from( uri: "scheduler:cleanup?scheduler=spring&" +
                "scheduler.cron={{antenna_ydb_opiskelijakortit.cleanup.cron}}")
                .routeId("antenna_ydb_opiskelijakortit-cleanup-rt")

                .setProperty(IirisProducer.MDC_SOURCE, constant( value: "{{iiris.cleanup.source}}"))
                .setProperty(IirisProducer.MDC_DESTINATION, constant( value: "{{iiris.cleanup.destination}}"))

                .to("iiris:New?info=Aloitetaan poistamaan vanhoja eriä taulusta: uudet_opiskelijakortit")
                .to("bean:ydbDao?method=deleteOldBatches()")

                .to("iiris:Done?info=Vanhojen erien poisto suoritettu.")
                .end();
    }
}
```

KUVA 9. Vanhojen erien siivoamiseen tarkoitettu reitti ja sen sisältö

Kuvassa 9 näkyvän reittitoteutuksen oleellisimmat kohdat löytyvät kommenttirivin alapuolelta (kahdella kauttaviivalla alkava koodirivi), alkaen reitin käynnistämisestä, joka tapahtuu Camelin from-avainsanaa hyödyntäen. Reitissä määritellään cron-ajastuksen lisäksi mm. reitin identifioimista helpottava tunniste, *routeId*, sekä lähde- ja kohdejärjestelmäarvot Digian kehittämälle valvontajärjestelmälle, Iirikselle. Reitin alkumäärittelyjen jälkeen esiintyvillä *to*-avainsanoilla ohjataan viestiä eteenpäin määriteltyihin päätepisteisiin. Tässä reitissä kaksi näistä kolmesta päätepisteestä kuuluu Iiris-komponentille, eli kyseessä on yksinkertaisten lokiviestien lähettäminen Iiris-valvontajärjestelmään. Näiden Iiris-päätepisteiden välistä löytyvä bean-päätepiste ohjaa viestin YdbDao-nimisen luokan deleteOldBatches-metodiin, jossa tietokannan poistologiikka varsinaisesti toteutetaan. Kuva em. metodin toteutuksesta löytyy alta (kuva 10).

```
public void deleteOldBatches() {
    iirisLogger.processing( msg: "Poistetaan vanhat erät, pidetään " +
            keepBatches + " viimeisintä otosta.");
    jdbcTemplate.update(DELETE_OLD_BATCHES_SQL, keepBatches);
}
```

KUVA 10. Vanhojen erien poistossa hyödynnetty deleteOldBatches-metodi

Kuten edeltävän sivun kuvasta 10 voi huomata, kyseessä on erittäin yksinkertainen metodi, jolla on vain kaksi toiminnallisuutta: lähettää Iiris-valvontajärjestelmään viesti integraation aikomuksesta poistaa vanhat erät, sekä JDBC-kirjastoa hyödyntävä metodikutsu, joka saa parametrikseen poistossa hyödynnetyn SQL-lauseen sekä säilytettävien tietokantaerien määrän. Poistossa hyödynnetty SQL-lauseke löytyy alta (kuva 11).

```
DELETE FROM
  uudet_opiskelijakortit
WHERE
  batch_time IN (
    SELECT batch_time
    FROM (
            SELECT DISTINCT batch_time AS batch_time
            FROM uudet_opiskelijakortit
            ORDER BY batch_time DESC
            LIMIT ?, 99999) foobar)
```

KUVA 11. Vanhojen erien poistossa käytetty SQL-lauseke

Kuvan SQL-lausekkeessa näkyvä kysymysmerkki (?) viittaa dynaamiseen SQL-parametriin. Tällaisia parametrejä sisältäviä SQL-lausekkeita kutsutaan dynaamisiksi SQL-lausekkeiksi, ja rakenteellisesti nämä dynaamiset SQL-lausekkeet konstruoidaan ja ajetaan vasta ajoaikana syötettyjen parametrien pohjalta (Gibbs n.d.). Dynaamisten SQL-lausekkeiden käytön MariaDB:ssä mahdollistaa työssä hyödynnetty JDBC-kirjasto.

Säilytettävien tietokantaerien määrä, sekä muutaman toisenkin muuttujan tiedot tuodaan ohjelmakoodin puolelle projektin juurikansiossa sijaitsevasta konfiguraatiotiedostosta. Kuvakaappaus tästä konfiguraatiotiedostosta on nähtävissä seuraavan sivun kuvasta 12.

```
# Vanhojen erien putsaukseen liittyvät asetukset

antenna_ydb_opiskelijakortit.cleanup.cron=* * 3 * * *
keep.old.batches=3


# Iiris-lokituksen parametrit

iiris.cleanup.source=ydb
iiris.cleanup.destination=ydb
```

KUVA 12. Projektikansion konfiguraatiotiedostoon määriteltyjä muuttujia

Säilytettävien tietokantaerien määrään käytetyn muuttujan lisäksi kuvan 12 kaappauksesta voi huomata cron-ajastuksen muuttujan, sekä kuvan alaosasta löytyvät Iiris-valvontajärjestelmän parametrit. Näihin konfiguraatiotiedostoihin on tässä integraatioprojektissa tapana laittaa yleisiä integraatiopalvelun parametrejä, kuten Iiris-valvontajärjestelmän lähde- ja kohdejärjestelmätietoja, sekä muita integraation ajoon vaikuttavia pääparametreja. Mikäli tässäkin tutkielmassa työstettyyn integraatioon olisi toteutettu datan siirtoon käytettävät reitit itse, löytyisi konfiguraatiotiedostosta esimerkiksi tiedonsiirtoon käytettyjen palvelimien tiedostopolkuihin liittyvät parametrit. Työssä hyödynnettyjen parent-projektin valmisreittien takia myös konfiguraatiotiedosto on kooltaan maltillisempi.

4.5 Java-apuluokkien hyödyntäminen

JavaBeaneja voi hyödyntää myös Camel-reittejä toteutettaessa. Reiteille voi kirjoittaa JavaBeaneihin johtavia päätepisteitä, joiden kautta voidaan ohjata sanoma vaikkapa uudelleenrakennettavaksi tai rikastettavaksi JavaBeanin sisälle. JavaBeaneja käytetään Camelissa myös muun muassa mallien kirjoittamiseen, jotka antavat esimerkiksi tietokannasta haetulle datalle eräänlaisen muotin, johon data voidaan asettaa.

Tässä työssä näitä JavaBeaneja käytettiin ympäri integraatiota, mm. uuden datan syöttämiseen tietokantaan, vanhojen tietokantaerien poistoon sekä yleisenä mallina siirrettävälle datalle. Kuvasta 13 voidaan nähdä tässä työssä käytetyn mallitiedoston sisältö.

```
import org.apache.camel.dataformat.bindy.annotation.CsvRecord;
import org.apache.camel.dataformat.bindy.annotation.DataField;
import org.apache.commons.lang.StringUtils;

@CsvRecord(separator =  ";", skipField = true, skipFirstLine = true)
public class StudentCard {

    @DataField(pos = 1, trim = true, required = true)
    private String studentNumber;

    @DataField(pos = 6, trim = true, required = true)
    private String studentCardNumber;

    public String getStudentNumber() {
        return StringUtils.defaultIfEmpty(studentNumber,  defaultStr: null);
    }

    public void setStudentNumber(String studentNumber) {
        this.studentNumber = studentNumber;
    }

    public String getStudentCardNumber() {
        return StringUtils.defaultIfEmpty(studentCardNumber,  defaultStr: null);
    }

    public void setStudentCardNumber(String studentCardNumber) {
        this.studentCardNumber = studentCardNumber;
    }
}
```

KUVA 13. Opiskelijakorttien datan muottina käytetyn malliluokan sisältö

Tässä tutkielmassa työstetty integraatio on pohjimmiltaan hyvin yksinkertainen, ja kuten yllä olevasta mallitiedoston kuvastakin voi huomata, määriteltyjä datakenttiä ei tiedostossa ole kuin kaksi. On hyvinkin normaalia, että isommissa integraatioprojekteissa mallitiedostoja on monia, ja näiden tiedostojen sisällä olevia datakenttiä voi olla useitakin kymmeniä.

Kuvassa näkyvät keltaiset annotaatiot liittyvät suosittuun Camel-integraatioissa käytettyyn Bindy-kirjastoon. Tätä kirjastoa käytetään datakenttien alustamisessa

tiettyyn dataformaattiin. Tässä integraatiossa lähdejärjestelmästä tuleva data on CSV-formaatissa, joten datakentät alustettiin myös CSV-muotoon. Tämä alustaminen tehtiin CsvRecord-annotaation avulla, joka näkyy mallitiedoston yläosassa ennen itse luokan määrittelyä. Tämän annotaation sisälle annettiin mm. tieto datakentät erottavasta merkistä (tässä puolipiste), sekä ohjeistettiin Bindy aina ohittamaan CSV-tiedoston ensimmäinen rivi, sillä lähdejärjestelmästä tulleisiin CSV-tiedostoihin oli määritelty otsikkorivi, joka ei sisällä varsinaista siirrettävää dataa.

DataField-annotaatiot luokan jäsenten määrittelyn yhteydessä kertovat Bindylle, mistä kohdasta CSV-tiedoston riviä kyseinen data otetaan, poistetaanko datasta tyhjät välit, ja pidetäänkö datakenttää pakollisena tietona. Esimerkiksi tässä tapauksessa required-parametrille on annettu arvo *true*, joten integraation ajo päättyy virheeseen, mikäli Bindy ei kykenekään löytämään tätä datakenttää CSV-tiedostosta.

Em. mallitiedoston lisäksi integraatiossa käytettiin Java-apuluokkia myös integraation logiikan toteutukseen. Itse toteutettuun Java-apuluokkaan kirjoitettiin muun muassa opiskelijakorttien datan syöttäminen tietokantaan. Implementaatio tästä toiminnallisuudesta löytyy kuvasta 14.

```
public void insertStudentCards(Object messageBody) {

    if (messageBody == null) {
        // Heitetään poikkeus rollback:ia varten
        throw new IllegalArgumentException("Message should not be null");
    }

    List<StudentCard> studentCards = (List<StudentCard>) messageBody;

    batchTime = Timestamp.valueOf(LocalDateTime.now());

    jdbcTemplate.batchUpdate(INSERT_STUDENT_CARDS_SQL, studentCards,
            BATCH_SIZE, this::setParameters);
}
```

KUVA 14. Opiskelijakorttien tietokantaan syöttämisen apuna käytetty Java-metodi

Edeltävän sivun kuvassa nähtävillä oleva metodi saa parametrikseen Camel-reitin puolelta tulevan bodyn, eli siirrettävän datan varsinaisen sisällön, ja tämän jälkeen alustaa kyseisen datamassan listaksi opiskelijakortteja. Tämän jälkeen metodi kirjaa käsiteltävän erän ajankohdan talteen muuttujan sisälle, ja päivittää tietokannan samaa JDBC-kirjastoa hyödyntäen. JDBC:n päivitysmetodin sisältä löytyvä BATCH_SIZE-muuttuja on määritelty saman Java-apuluokan yläosassa (kuvan ulkopuolella), ja tällä muuttujalla voidaan säätää käsiteltävien dataerien kokoa. Mahdollisuus dataerien koon säätämiselle on kätevää esimerkiksi silloin, kun halutaan varmistua, että tietokanta-ajo ei vie liikaa resursseja, tai jotta tietokantaan kerkeäisi tallentumaan mahdollisimman paljon dataa ennen potentiaalisen vian ilmenemistä.

Koodin kirjoitushetkellä ei tietenkään voitu tietää minkälaista dataa tietokantaan pitää päivittää, joten myös päivitysmetodissa piti hyödyntää dynaamisia SQL-lausekkeita. Edellä mainitun JDBC-metodin viimeiseksi parametriksi annettu funktio on toiminnallisuudeltaan todella yksinkertainen, mutta kokonaisuuden toimivuuden kannalta silti hyvin tärkeä. Tämä kuvassa 15 näkyvä setParameters-funktio mahdollistaa datan dynaamisen käytön tietokantapäivityksissä.

```
private void setParameters(PreparedStatement ps, StudentCard studentCard) throws SQLException {

    ps.setTimestamp( parameterIndex: 1, batchTime);
    ps.setString( parameterIndex: 2, studentCard.getStudentNumber());
    ps.setString( parameterIndex: 3, studentCard.getStudentCardNumber());
}
```

KUVA 15. Tietokantaan syöttämisessä käytettyjen parametrien asettaminen

Funktion ainoana tehtävänä on asettaa Camel-reitiltä saatu ja Java-apuluokassa prosessoitu data osaksi dynaamista SQL-lauseketta. Funktio lukee tämän annetun datan, ja tallettaa tiedot PreparedStatement-luokan muuttujiksi. PreparedStatement on java.sql-pakettiin kuuluva luokka, josta luodut oliot edustavat esikäännettyjä SQL-lausekkeita (PreparedStatement-luokan dokumentaatio n.d.). JDBC osaa hyödyntää näitä olioita SQL-lausekkeiden luonnissaan. SQL-lauseke, jota tämän työn tietokannan päivityksessä käytetään, löytyy seuraavan sivun kuvasta 16.

```
INSERT INTO uudet_opiskelijakortit
    (batch_time, opiskelijanumero, opiskelijakortin_numero)
VALUES (?, ?, ?)
```

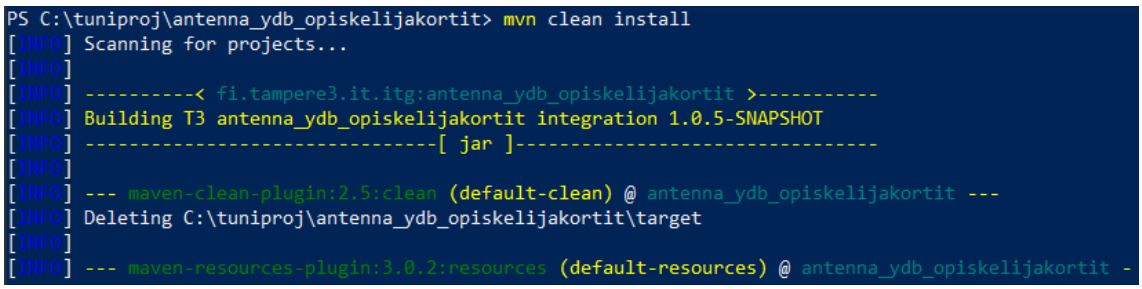
KUVA 16. SQL-komento uusien opiskelijakorttien syöttämisestä tietokantaan

Kuten koodia katsomalla voi nähdä, kyseessä on jälleen dynaaminen SQL-lauseke. Tietokantaan syötettäviä arvoja ei voitu koodin kirjoitusaikana vielä tietää, joten ne korvattiin kysymysmerkkinotaatiolla, eli toisin sanoen SQL-parametreilla. Tieto erän käsittelyajankohdasta, prosessoitavien opiskelijoiden opiskelijanumeroista sekä kyseisten opiskelijoiden opiskelijakorttien numeroista sijoitetaan JDBC-kirjaston ja Java-toiminnallisuuksien avulla näiden parametrien tilalle. Parametrien korvaamisen jälkeen JDBC ajaa SQL-lausekkeen projektissa määriteltyyn tietokantaan, ja tämän jälkeen tiedon pitäisi olla päivittynyt, mikäli ajonaikaisia virheitä ei tietokantapäivityksen aikana ilmene.

4.6 Integraation kääntäminen Mavenilla

Kun integraation varsinainen toteutus saatiin valmiiksi, täytyi toteutuksen toimivuutta tietysti testata jotenkin. Luvussa 4.1 kuvattiin jo tarpeelliset toimenpiteet kehitysympäristön pystyttämiseksi, toisin sanoen toimiva kehitysympäristö oli tässä vaiheessa jo konfiguroituna. Jäljelle jäi enää varsinaisen integraation asennustoimenpiteiden tekeminen, eli projektin rakentaminen Mavenilla, Mavenin luoman JAR-ajotiedoston syöttäminen Karafille, sekä integraation käynnistäminen Karafissa.

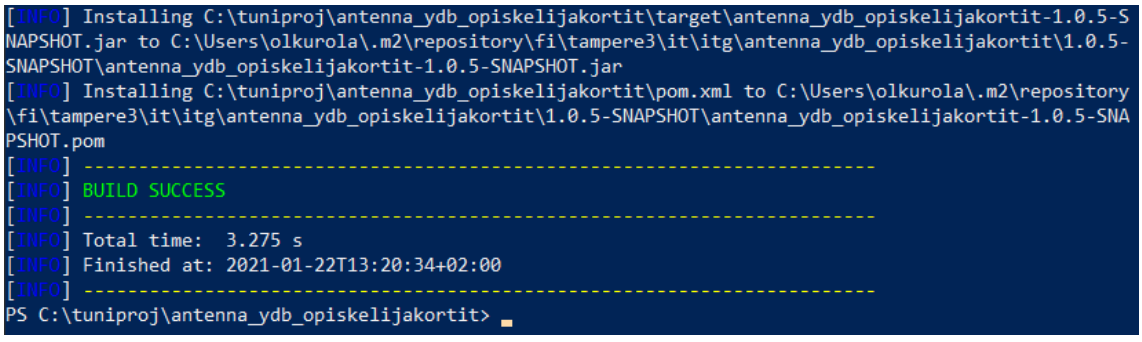
Projekti käännettiin Mavenilla Windows-käyttöjärjestelmän PowerShell -komentotulkkia käyttämällä. Kääntämiseen käytettiin komentoa *mvn clean install* (seuraava sivu, kuva 17).



KUVA 17. Projektin rakentamisen käynnistyessä ilmestyvää tulostusta

Annettu komento muodostuu kolmesta eri avainsanasta, joista ensimmäinen (mvn) kutsuu Mavenin ajotiedostoa, toinen (clean) siivoaa mahdolliset projektin aikaisemmista käännöksistä jääneet tiedostot ja kolmas (install) kääntää, paketoi ja asentaa projektin Mavenin paikalliseen repositorioon. Tänne paikalliseen repositorioon asennettuja projekteja voi muun muassa käyttää muiden projektien riippuvuuksissa ilman ylimääräisiä määrittelyjä tai asennuksia, sillä Maven tunnistaa nämä repositoriosta löytyvät projektit automaattisesti. Esimerkiksi integraatiopalvelun parent-projekti asennettiin tällä tavoin Mavenin paikalliseen repositorioon kehitysympäristön asennuksen yhteydessä, jotta parent-projekti voitaisiin helposti sisällyttää kaikkiin sitä tarvitseviin integraatiototeutuksiin.

Käännöksen yhteydessä Maven tarkistaa projektin muun muassa käännösvirheiden kannalta (esimerkiksi puuttuvat määrittelyt POM-tiedostosta) sekä mahdolliset projektiin toteutetut yksikkötestit. Mikäli käännösvirheitä ei ole ja yksikkötestit menevät läpi, Maven ilmoittaa käännöksen onnistumisesta viestillä *BUILD SUCCESS* (kuva 18).



KUVA 18. Käännöksen yhteydessä luotujen tiedostojen asennus sekä tuloste onnistuneesta käännöksestä

Lopputulosteessa Maven ilmoittaa muun muassa tiedostojen asennuskohteet sekä käännökseen kuluneen ajan. Jos käännös ei jostain syystä onnistu, Maven ilmoittaa myös havaituista ongelmakohdista komentokehotteen tulosteella kuvan 19 mukaisesti.

```
[INFO] 8 errors
[INFO] -------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  1.500 s
[INFO] Finished at: 2021-01-22T13:30:15+02:00
[INFO] ------------------------------------------------------------------------
[ERROR] Failed to execute goal org.apache.maven.plugins:maven-compiler-plugin:3.6.1:compile (default-c
ompile) on project antenna_ydb_opiskelijakortit: Compilation failure: Compilation failure:
[ERROR] /C:/tuniproj/antenna_ydb_opiskelijakortit/src/main/java/fi/tampere3/it/itg/antenna_ydb_opiskel
ijakortit/model/StudentCard.java:[3,52] package org.apache.camel.dataformat.bindy.annotation does not
exist
```

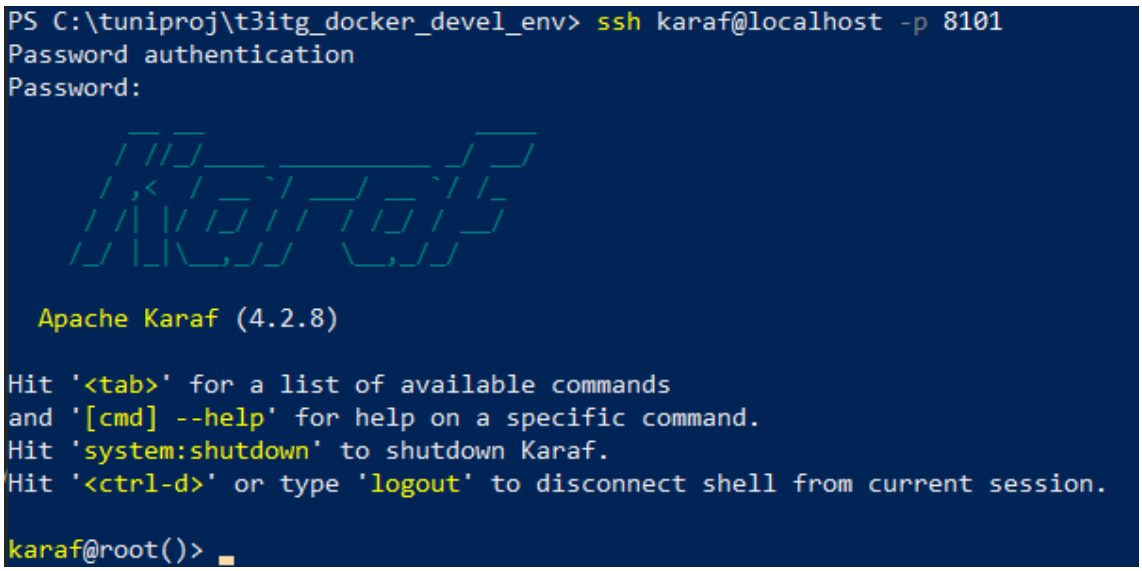
KUVA 19. Käännöksen epäonnistuessa ilmestyvää tulostusta

Kyseisessä testimielessä aiheutetussa virhetapauksessa POM-tiedostosta kommentoitiin pois eräs datamuunnoksiin käytetty riippuvuus nimeltä *Bindy*. Em. virhetulostuksesta voikin havaita, että Mavenin käännös epäonnistui juuri tuon puuttuneen Bindy-paketin takia.

Käännöksen onnistuessa Maven paketoi koko projektin yhteen JAR-tiedostoon, jonka voi syöttää Karafille joko hot deploy -ominaisuutta hyödyntäen, tai antamalla manuaalisen komennon asennukselle Karafin konsolissa, niin kuin tässä työssä tehtiin. Jotta paketoitu integraatio voitiin Karafin konsolista manuaalisesti asentaa, täytyi Karafiin ensin kirjautua.

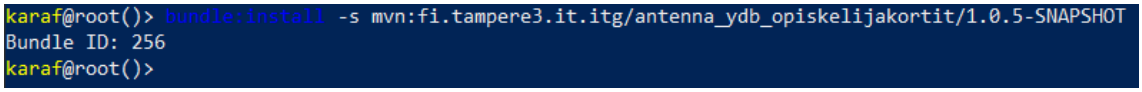
4.7 Integraatiopaketin asentaminen ja ajo

Karafiin pystyttiin nyt ottamaan yhteys SSH-varmennusta hyödyntäen, sillä Karaf oli jo käynnistetty aiemmin Docker Composen käynnistyskomennon yhteydessä. SSH-varmennukselle tarpeelliset konfiguraatioarvot olivat valmiiksi asetettuina Compose-konfiguraatiotiedoston sisällä olevaan Karaf-näköistiedostoon. Composen konfiguraatiotiedostossa oli Karafin käyttöön asetettu portti 8101, ja Docker-kontteja ajettiin paikallisesti, joten yhteyden muodostukseen käytetty ssh-komento rakennettiin näiden tietojen pohjalta. Komennon syöttö sekä siitä seuraava Karafin päävalikon tulostus voidaan nähdä seuraavalta sivulta (kuva 20).



KUVA 20. Yhdistäminen Karafiin SSH-varmennuksella, sekä onnistuneesta kirjautumisesta syntyvä päävalikon tulostus

Karafin päävalikon tulostuksen jälkeen voi konsoliin syöttää haluttuja komentoja. Jotta aikaisemmin käännetty integraatio saatiin asennettua Karafiin, täytyi Karafille antaa asennuskomento. Pakettien eli bundlejen asennus Karafiin suoritetaan *bundle:install*-komennon avulla (kuva 21).



KUVA 21. Integraation asennus Karafiin, ja Karafin integraatiolle määräämä ID

Annettuun bundle:install-komentoon on liitettynä lippu ’-s’, joka tarkoittaa sitä, että bundle käynnistetään automaattisesti asennuksen onnistuttua ilman erillistä käyttäjän syötettä. Tämän lipun jälkeen tuleva osuus on Mavenin omaa syntaksia, jossa Mavenille ilmoitetaan POM-tiedostossakin määriteltyjen tietojen mukaisesti (groupId, artifactId, versionumero) käsiteltävän integraation tiedot. Näiden tietojen avulla Maven osaa muun muassa valita oikean projektin sekä oikean versionumeron asennettavaksi paketiksi.

Eräs kätevä komento, jonka Karafille voi antaa, on ’list’, jonka myötä Karaf tulostaa komentokehotteeseen muun muassa kaikki käynnissä olevat paketit, niiden tilat ja versionumerot, sekä paketeille määrätyt ID-numerot. Lvl on lyhenne sa-

noista *start-level*, ja tämä tarkoittaa järjestystä, jossa paketit käynnistetään. Kuvassa 22 nähdään ensimmäiset kymmenisen riviä työssä käytetyn kehitysympäristön Karafin bundlejen listauksesta.

```
karaf@root()> list
START LEVEL 100 , List Threshold: 50
 ID | State  | Lvl | Version            | Name
 22 | Active |  80 | 4.2.8              | Apache Karaf :: OSGi Services :: Event
 45 | Active |  80 | 1.9.3.1            | Apache ServiceMix :: Bundles :: jasypt
 46 | Active |  80 | 1.4.4              | OPS4J Pax JDBC Generic Driver Extender
 47 | Active |  80 | 1.4.4              | OPS4J Pax JDBC Config
 48 | Active |  80 | 1.4.4              | OPS4J Pax JDBC Pooling Support Base
 49 | Active |  80 | 1.0.0.201505202023 | org.osgi:org.osgi.service.jdbc
 50 | Active |  80 | 0.4.1              | Camel Iiris Component
 54 | Active |  80 | 2.9.9              | Jackson-dataformat-CSV
 56 | Active |  80 | 2.9.9              | Jackson datatype: JSR310
 61 | Active |  80 | 7.4.1              | Microsoft JDBC Driver for SQL Server
 63 | Active |  50 | 1.6.1              | JavaMail API
```

KUVA 22. Listaus Karafin käynnissä olevista prosesseista

Kuvan 22 bundlejen nimilistauksessa on näkyvillä useita integraatiokehityksessä käytettyjä Java-komponentteja, kuten Jackson- ja JDBC-riippuvuudet, joita käytetään mm. JSON-datan ja tietokantaoperaatioiden hallintaan. Aiemmin asennuksen yhteydessä (kuva 21) Karaf antoi tuoreelle integraatiolle uuden, uniikin ID-numeron. Tämä ID oli 256. Kun Karafin aktiivisten prosessien listauksessa navigoidaan alas tuoreimpien bundlejen kohdalle, voi vasta-asennetun bundlen tilatiedot havaita muiden bundlejen joukosta listan alimpana (kuva 23).

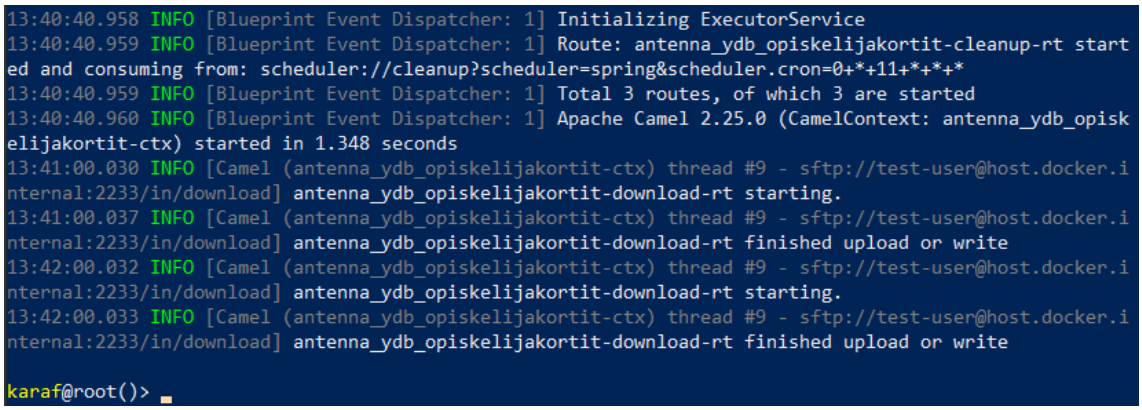
```
254 | Active |  80 | 0.9.1          | JSON Web Token support for the JVM
255 | Active |  80 | 3.8.1          | Ehcache 3
256 | Active |  80 | 1.0.5.SNAPSHOT | T3 antenna_ydb_opiskelijakortit integration
karaf@root()>
```

KUVA 23. Tässä työssä toteutetun integraation tilatiedot

Kuvan tulostuksesta voidaan huomata, että tämän uuden integraation tila on *Active*. Toisin sanoen paketin asennus on onnistunut, ja integraatio on lähtenyt onnistuneesti käyntiin. Jos esimerkiksi paketin asennuksen tai käynnistyksen yhteydessä tulisi vakavia ongelmia, näkyisi tuon paketin tilana listauksessa ”Failed”.

Karafin oman keskitetyn lokitusjärjestelmän ansiosta voidaan asennus- ja käynnistysprosessin etenemistä sekä integraation ajonaikaista suorittamista seurata

suoraan näistä Karafin lokeista. Lokitiedot saa näkyviin kirjoittamalla Karafin konsoliin *log:display*. Kuvassa 24 on näkyvillä lokitulostusta muun muassa tässä työssä toteutetun integraation käynnistämisestä sekä sen ajonaikaisista suorituksista.



KUVA 24. Lokitulostusta ajanhetkeltä, jolloin integraatio asennettiin

Kuten kuvan 24 tulostuksesta voi lukea, työstetyn integraation kaikki kolme reittiä saatiin käynnistettyä onnistuneesti. Tämän lisäksi aikaleimojen 13:41 ja 13:42 kohdalla voidaan huomata geneerinen ajonaikainen aloitus- ja lopetustulostus. Nämä tulostukset tulevat työssä hyödynnetyn parent-projektin valmisreiteistä. Tulostuksista voi päätellä, että integraatio käynnistyy sekä ajaa itsensä loppuun oikeaoppisesti, sillä alku- ja lopputulostuksen väliin ei ole tullut virhetulostuksia. Myös cron-ajastus toimii oletetulla tavalla, sillä integraatio ajastettiin konfiguraatiotiedostossa ajautumaan joka minuutti, ja kuvan tulostuksen mukaisesti aikaleimat poikkeavat minuutin verran toisistaan.

Kun integraation toimivuudesta oli varmistuttu, saatiin se siirtää kehitysputkessa eteenpäin, eli testausvaiheeseen. Käytännössä kehittäjän tehtäviin kuuluva integraation toimivuuden varmistaminen tarkoittaa juuri edellä kuvattua integraation kääntämistä Mavenilla sekä asentamista Karafiin. Näiden toimien myötä voidaan todeta se, että koodissa ei ole asennusta tai ajoa estäviä suurempia virheitä. Lopulliset logiikkatestit tehdään testaajien toimesta testausympäristössä, jossa pyritään varmistumaan siitä, että integraatio esimerkiksi siirtää oikeanlaista dataa juuri oikeanlaisessa muodossa.

4.8 Työn tulokset ja viimeistely

Integraation valmistuttua se siirrettiin eteenpäin testattavaksi. Testauksessa integraatio ajettiin testiympäristön datan läpi ja siihen tehtiin erilaisia pistotestejä, jotta integraation toimivuudesta eri tilanteissa sekä erilaisilla datan kombinaatioilla voitiin varmistua. Varsinaisen testiympäristön testit tehtiin TUNI-integraatiopalvelun integraatioasiantuntijoiden toimesta, sillä vain heille oli määritelty oikeudet testiympäristön dataan.

Ekstensiivisen testauksen jälkeen, kun TUNI-integraatioasiantuntijat olivat todenneet integraation toimivaksi, se siirrettiin odottamaan seuraavaa releasea eli julkaisua. Integraatioiden julkaisuja pyrittiin tekemään tietyin väliajoin, joka vaihteli noin viikosta kahteen viikkoon. Integraation julkaisuun kuului integraation asentaminen tuotantoympäristöön, sekä tuotantoympäristön konfiguraatiomuutokset kyseisen integraation osalta. TUNI-integraatioasiantuntijat hoitivat myös tämän tehtävän, sillä tuotantoympäristöön oli hyvin rajatut pääsyoikeudet.

Integraation julkaisun yhteydessä oli tärkeä varmistaa, että integraatiolle jo määrittelyvaiheessa luodut wikisivut olivat julkaisukelpoisessa kunnossa. Näille integraatiopalvelun integraatiokohtaisille wikisivuille kirjattiin tyypillisesti vähintään integraation siirtämä data ja sen formaatti, integraation EIP-kaavio sekä yleiskuvaus integraation tehtävästä. Monimutkaisemmissa ja isommissa integraatiototeutuksissa käytettiin myös paljon vapaata tekstiä kuvaamaan toteutusta, jotta mahdollisten jatkokehittäjien olisi helpompi ja nopeampi ymmärtää integraation toteutuslogiikkaa.

Työstämäni Antenna-integraation julkaisuvaihe sujui ongelmitta, ja julkaisun jälkeen integraation ajo aloitettiin saman tien tuotantoympäristössä. Integraatio jää siis tarkkailemaan TUNI-integraatiopalvelun tiedostopalvelimille saapuvia Antennan tiedostoja, ja laukaisee integraatioon sisällytetyt valmisreitit käyntiin, kun oikean nimiset tiedostot löytyvät tarkkaillusta kansiosta. Vanhojen erien poistoreitin laukaisu tapahtuu konfiguroitavissa olevan cron-ajastuksen kautta, eli vanhoja eriä käydään poistamassa niin usein, kuin TUNI-integraatiopalvelun tuotannon ylläpitäjät itse haluavat. Normaalisti näitä integraatioiden poistoreittejä ajetaan noin kerran päivässä.

Tämän integraation prosessoimat datamassat viedään eteenpäin TUNI-integraatiopalvelun käyttämään välimuistitietokantaan, jota muut TUNI-integraatiot hyödyntävät tietolähteenä omiin integraatiotarpeisiinsa. Tätä Antenna-opiskelijakorttidataa käy välimuistitietokannasta hakemassa ja hyödyntämässä mm. TUNI-integraatiopalvelun kulunvalvonnan integraatiot.

4.9 Työn analysointi

Integraation toteuttamista helpotti huomattavasti TUNI-integraatiopalvelun hiottu kehitysprosessi. Integraation vaatimusmäärittelyä hoitivat TUNI-integraatioasiantuntijat jo hyvissä ajoin ennen integraation siirtämistä kehitystilaan. Tämän vaatimusmäärittelyn yhteydessä tehtiin myös wikisivut integraatiolle integraatiopalvelun wikitilaan. Näiltä wikisivuilta löytyi vaatimusmäärittelyn lisäksi muutakin kehitystä ohjaavaa tietoa, kuten siirrettävän datan formaatti sekä esimerkkidataa. Kehitystä aloittaessa oli helppo päästä alkuun pelkästään integraation wikisivuja silmäilemällä.

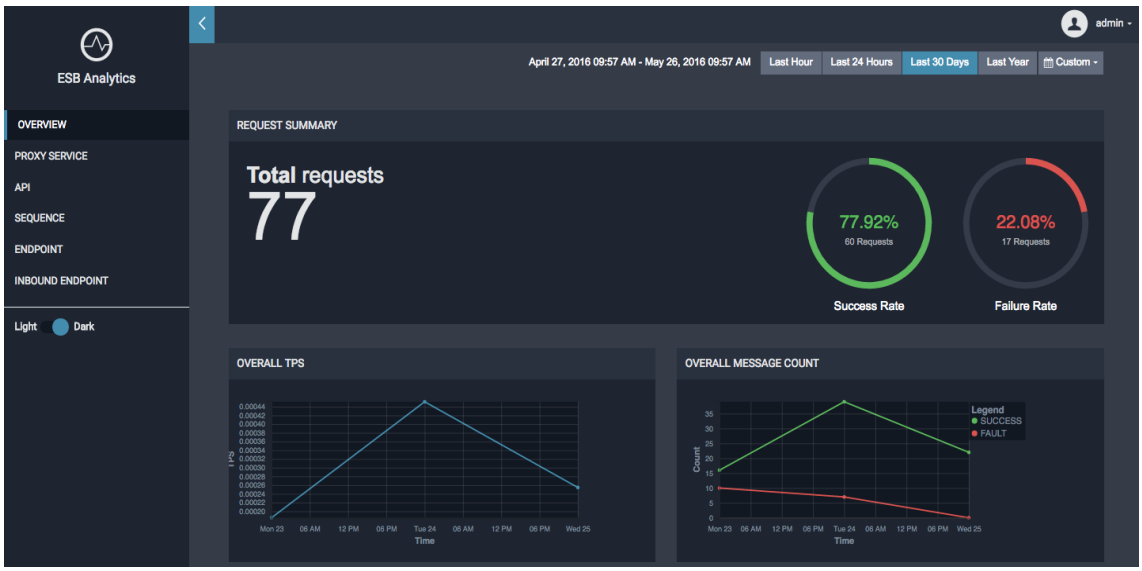
Myös kehitysputken loppupuoli oli hyvin kaavailtu. Integraation ollessa valmis se testattiin saman integraatioasiantuntijan toimesta, joka määrittelynkin oli tehnyt. Tällä varmistuttiin siitä, että integraatio toimi juuri niin kuin sen kuuluikin toimia. Lopuksi tuotantoympäristöön julkaisun teki vielä oma, julkaisuihin erikoistunut henkilö. Kehitysputki tuntui kaiken kaikkiaan oikein toimivalta ainakin kehittäjän näkökulmasta, sillä kehittäjänä sain keskittyä juuri siihen, mihin minut oli palkattukin, eli kehitystyöhön.

Työstämäni integraatio asennettiin muidenkin TUNI-integraatiopalvelun integraatioiden lailla OSGi-bundlena Apache Karafiin. Tämä mikropalvelumainen ajatusmalli omina paketteina ajettavista integraatioista oli varmasti hyvin järkevä idea TUNI-integraatiopalvelun aloittaessa toimintansa, mutta korkeakoulujen fuusioitumisen yhteydessä tapahtuva räjähdysmäinen integraatiotarpeen kasvu saattoi osoittautua suuremmaksi, kuin mitä osattiin odottaa. TUNI-integraatiopalvelussa on tämän tutkielman kirjoitushetkellä jo yli 100 integraatiota, joka on jo varsin iso määrä. Nämä Camelilla toteutetut ja Karafin sisällä pyörivät integraatiot ovat toki

äärimmäisen kevyitä ja sinänsä helppoja hallita yksittäisinä paketteina. Kokonaisuuden jatkaessa paisumistaan ei voi kuitenkaan olla miettimättä, kuinka jokin monipuolisempi ESB-järjestelmä voisi sovita tähän kasvaneeseen integraatioprojektiin paremmin.

Tässä tutkielmassa toteutetun Antenna-integraation kohdalla tilanne oli sama kuin muillakin TUNI-projektin integraatioilla, eli käytännössä ainoat lähteet integraation ajonaikaisen toimivuuden tarkistamiseen olivat Karafin omat lokit sekä Digian Iiris-järjestelmään integraatiosta lähetetyt lokiviestit. Näiden lokitietojen avulla voi tietysti selkeästi nähdä, onko integraatio suoriutunut ajostaan, vai onko se epäonnistunut tiedon siirrossa, mutta kaikenlainen lisätieto jää kuitenkin saamatta. Tietoa esimerkiksi integraation suoriutumisprosentista ei Karafista mitenkään automaattisesti näe. Ainut tapa tällaisen tiedon saamiseen tässä ympäristössä olisi tehdä jokin räätälöity komponentti, joka tarkkailisi vaikka integraation lokitietoja, ja sen avulla tekisi jotain yhteenvetoa onnistuneista ja epäonnistuneista suorituskerroista.

TUNI-integraatiopalvelussa tätä ongelmaa lieventää hieman Digian ylläpitämän Iiris-valvontajärjestelmän hyödyntäminen, jossa juuri näistä Iirikseen lähetetyistä lokiviesteistä kerätään statistiikkaa. Tätä statistiikkaa visualisoidaan Iiriksen käyttöliittymässä muun muassa erilaisilla pylväsdiagrammeilla sekä ympyräkaavioilla, mutta ei se silti vastaa kokonaisvaltaisten ESB-järjestelmien tarjoamien analytiikkatyökalujen tasoa. ESB-järjestelmiä kehitetään alusta alkaen kokonaisuuksiksi, jolloin ESB:n komponenttien välillä löytyy vahvoja integraatioita, joiden avulla pystytään näyttämään monipuolista dataa ESB:n eri komponenteista. Esimerkki WSO2-integraatiomoottorin tarjoamasta ESB-analytiikkatyökalun kojelaudasta löytyy seuraavalta sivulta (kuva 25).



KUVA 25. WSO2-integraatiomoottorin ESB-analytiikkatyökalun kojelauta (WSO2 n.d.)

Kuten kuvasta voi huomata, kyseisen integraatiomoottorin kojelauta tarjoaa ESB:mäisellä tavalla monenlaista graafista näkymää tuotteen integraatiokomponenttien toiminnasta. Tällainen mahdollisesti hyvin hyödyllinenkin lisätieto integraatioiden toimintahistoriasta jää tällä hetkellä saamatta TUNI-integraatiopalvelun integraatioista, joka on mielestäni tällä hetkellä kyseisen projektin heikoin kohta toteutettujen integraatioiden osalta.

Tämä havainto ei ole jäänyt huomaamatta myöskään TUNI-integraatioasiantuntijoiden keskuudessa, sillä myös TUNI-integraatioarkkitehdit ovat lähiaikoina miettineet mahdollisia ratkaisuja tähän integraatioiden hallintaan liittyvään ongelmaan jatkuvasti kasvavassa tuotantoympäristössä. Mikäli integraatioiden monitorointia sekä hallintaa saataisiin vielä parannettua uhraamatta liikaa integraatioratkaisujen keveyttä, alkaisi TUNI-integraatiopalvelun toiminta näyttämään erittäin hyvältä arkkitehtuurillisesta näkökulmasta katsottuna.

5 POHDINTA

5.1 Mietteet Apache Camelista

Tämän tutkielman tarkoituksena oli tutustua integraatioalan työnkulkuun sekä sen ongelmiin, ja evaluoida Apache Camel -työkalun sopivuutta osana integraatiokehityksen teknologiavalikoimaa. Tutkielmaa voi pitää monilta osin onnistuneena, sillä tämän työn myötä saavutettiin hyvä ymmärrys integraatioalan perusteista, saatiin arvioitua valitun Camel-työkalun pätevyyttä sekä onnistuttiin toteuttamaan yksi integraatiokokonaisuus osana tätä tutkielmaa.

Camelin hyvien puolien osilta voin allekirjoittaa luvussa 3.2 mainitut Anton Goncharovin sekä Kai Wähnerin kommentit Camelista. Omien kokemusteni perusteella nostaisin itsekin Camelin vahvimmaksi puoleksi varmasti Camelin yhdenmukaisuuden. Integraatiototeutusten toistuvat rakenteet, joissa on aina tuottaja, kuluttuja sekä päätepisteet, auttavat hahmottamaan esimerkiksi muiden kehittäjien tekemien integraatioiden tarkoitusta ja toimintalogiikkaa. Omassa oppimisprosessissani huomasin, kuinka kunnon perehtyminen yhteen Camel-toteutukseen avasi ovet myös kaikkiin muihinkin Camelilla toteutettuihin integraatioihin. Camel-integraatioiden tyypillisen työnkulun havainnollistumisen jälkeen näitä integraatiototeutuksia oli helppo seurata ja ymmärtää.

Oman Camel-käyttöni perusteella voinkin todeta, että Camel todella tuntuu yhdeltä vahvalta ratkaisulta tähän integraatioalaa vaikeuttavaan pulmaan, jossa yhdenmukaisuuden puute luo tarpeetonta kaaosta kehittäjien sekä arkkitehtien keskuudessa. Itselleni ei ole vielä tullut vastaan yhtäkään tilannetta, jossa Camel ei olisi soveltunut ratkaisuksi jotakin integraatiototeutusta tehdessä, mutta mainittakoon kuitenkin, että integraatiokehityksen kokemukseni on vielä sen verran vähäistä, että omista mielipiteistäni puuttuu luonnollinen objektiivisuus tämän asian kohdalla. Uskon vahvasti siihen, että kaikelle on oma aika ja paikkansa, ja esimerkiksi raskaahkot ESB-ratkaisut ajavat varmasti asiansa ESB:eille suotuisissa ympäristöissä, eli monimutkaisissa laajan skaalan projekteissa.

Camelin arviointi tässä työssä perustui vahvasti omiin käyttökokemuksiin sekä alan ammattilaisten mielipiteisiin, joten tämän tutkielman tuloksena syntyneitä observaatioita ei voidakaan pitää konkreettisena faktatietona Camelin paremmuudesta, vaan enemmänkin suuntaa antavana tutkimuksena sen pohjatason soveltuvuudesta integraatiotyökaluna. Kattavien, objektiivisiin faktatietoihin perustuvien vertailujen tekemisessä olisi mennyt hyvin paljon aikaa muun muassa vertailtavien teknologioiden opettelemiseen, sillä Camelin suurimmat kilpailijat ovat varmasti raskaat mutta monipuoliset ESB-ratkaisut, joiden peruskäyttökin vaatii jo paljon perehtymistä itse käytettävään tuotteeseen.

On syytä myös huomioida, että työn tarkoituksena ei ollutkaan tehdä teknologioiden välillä syvällisiä vertailuja absoluuttisen paremmuuden selvittämiseksi, vaan tavoitteena oli yksinkertaisesti tutustua töissä käytettyyn Camel-työkaluun, ja pohtia sen pätevyyttä integraatioalalla ilmenevien ongelmien ratkaisemiseen. Kevyitä vertailuja muihin teknologioihin käytettiin vain osoittamaan Camelin vahvuuksia ja heikkouksia sekä antamaan perspektiiviä perinteisistä integraatiotyökaluista.

5.2 Pohdintaa projektin hyödyllisyydestä

Tutkielman osana toteutettua integraatioprojektia työstettäessä tuli opittua paljon uutta ei pelkästään integraatiokehityksen, vaan myös ohjelmistotuotannon kannalta. TUNI-integraatiopalveluprojektissa työskennellessä pääsi näkemään muun muassa tuotantopuoleen ja kehitysputkeen liittyviä ratkaisuja todella läheltä. Esimerkiksi testiympäristön sekä tuotannon julkaisuihin liittyviin automaatioratkaisuihin tuli tutustuttua, sillä projektissa käytetyn Jenkins-automaatiopalvelimen toiminta vaati muun muassa sen, että versionhallinnan etärepositorioon pusketut muutosjulkaisut oli Git-tägätty asianmukaisella versionumerolla. Nämä integraatiokehitystehtävät olivat ensimmäiset työtehtäväni ohjelmistoalalla, joten arvokasta tietoa esimerkiksi ohjelmistotuotannon prosesseista sekä yleisistä apuvälineistä tuli myös saatua.

Eräs toinen esimerkki työnkulun optimoinnin sekä suoraviivaistamisen tiimoilta TUNI-projektissa liittyy integraatiopalvelun kehittämään, ja tässäkin työssä hyödynnettyyn parent-projektiin. Integraatiototeutuksissa toistuvien rakenteiden havainnoiminen sekä niiden eristäminen omiksi komponenteikseen oli arkkitehtuurillisesti nerokas idea, joka toi mukanaan useita hyötyjä. Kehittäjien ei tarvinnut yksinkertaisten, toistuvien toteutusten kohdalla keksiä joka kerta pyörää uudestaan, vaan työnkulkua saatiin nopeutettua kierrättämällä näitä yleiskäyttöisiä ratkaisuja. Esimerkkinä tästä toimintatavasta on tässäkin työssä käytetty TUNI-integraatiopalvelun parent-projektista löytyvä yleiskäyttöinen tiedonsiirtokomponentti, jota hyödyntämällä vältyttiin erillisen tiedonsiirtoreitin kirjoittamisesta. Geneerinen tiedonsiirtoreitti oli jo toteutettuna parent-projektiin, ja määrittelemällä tälle komponentille vain oikeat parametrit saatiin tämä komponentti valjastettua myös tämän projektin käyttöön. Näin toimimalla säästyttiin muun muassa turhalta työltä ja bugeilta, sekä saatiin yhtenäistettyä integraatioratkaisuja entisestään.

Aikoinaan minulle mainostettiin hakemaani Digian integraatiokehittäjän paikkaa eräänlaisena väylänä ohjelmistoalalle. Kerrottiin muun muassa siitä, kuinka integraatioalalla työskenteleminen voi avata silmiä tietoteknisten toteutusten ja kokonaisuuksien osalta. Työssä tarjottiin näköalapaikkoja suurien järjestelmien välisiin kommunikaatioratkaisuihin, joiden ymmärtäminen helpottaisi ohjelmistoalalla työskentelyä. Tämän tutkielman sekä ohessa toteutetun integraatiotyön perusteella voin hyvin mielin todeta, etteivät nämä lupaukset jääneet vain mainospuheeksi. Integraatiokehittäjän tehtävät ovat jo näinkin lyhyellä aikavälillä tuottaneet erittäin hedelmällisiä tuloksia, ja hyöty tulee varmasti näkymään myös tulevissa ohjelmistoalan työtehtävissä.

Kuten tämänkin tutkielman johdannossa mainittiin, nykypäivän sovellukset harvoin elävät eristyksissä. Kuuluipa työnkuvaan sitten web-sovellusten kehittäminen, testiautomaatioiden toteuttaminen, tai jotkin muut ohjelmistokehitystehtävät, on järjestelmien väliseen kommunikointiin toteutettujen ratkaisujen ymmärtäminen silti kullanarvoista. Näihin toteamuksiin sekä henkilökohtaisiin kokemuksiin nojaten voin ehdottomasti suositella integraatioalaa kaikille aloitteleville ohjelmistoalan työntekijöille, jotka haluavat kehittää omaa näkemystään tietoteknisistä kokonaisuuksista, ja jotka haluavat luoda vahvan pohjan ohjelmistokehityksen tietämykselle.

LÄHTEET

Apache Camel Migration Guide. 2019. GitHub-dokumentti. Julkaistu 11.7.2019. Päivitetty 30.1.2021. Luettu 12.4.2021.
https://github.com/apache/camel/blob/master/docs/user-manual/modules/ROOT/pages/camel-3-migration-guide.adoc

Baum, D. n.d. Always on. GoJava-sivuston artikkeli. Luettu 12.4.2021.
https://go.java/netflix.html

Camel Core. n.d. Apache Camel -sivuston dokumentaatiosivu. Luettu 12.4.2021.
https://camel.apache.org/manual/latest/camel-core.html

Camel DSL. n.d. Apache Camel -työkalun dokumentaatiosivu. Luettu 18.1.2021.
https://camel.apache.org/manual/latest/dsl.html

Camel Lifecycle. n.d. Apache Camel -sivuston dokumentaatiosivu Luettu 12.4.2021.
https://camel.apache.org/manual/latest/lifecycle.html

Compose-dokumentaatio. n.d. Overview of Docker Compose. Docker-palvelun dokumentaatiosivu. Luettu 15.1.2021.
https://docs.docker.com/compose/

Docker overview. n.d. Docker-palvelun dokumentaatiosivu. Luettu 13.1.2021.
https://docs.docker.com/get-started/overview/

Donohue, T. n.d. What are Camel Routes? Tom Donohuen blogikirjoitus. Luettu 12.4.2021.
https://tomd.xyz/camel-routes/

Goncharov, A. n.d. Streamline Software Integration: An Apache Camel Tutorial. Toptal-asiantuntijasivuston artikkeli. Luettu 5.3.2021.
https://www.toptal.com/apache/apache-camel-tutorial

Gibbs, M. n.d. What is Dynamic SQL? Study-opetussivuston kirjoitus. Luettu 12.4.2021.
https://study.com/academy/lesson/what-is-dynamic-sql.html

Hohpe, G. & Woolf, B. 2003. Enterprise Integration Patterns -kirjan kotisivu. Luettu 31.1.2021.
https://www.enterpriseintegrationpatterns.com/

IBM. 2019. ESB (Enterprise Service Bus). IBM-yhtiön verkkoartikkeli. Julkaistu 18.7.2019. Luettu 12.4.2021.
https://www.ibm.com/cloud/learn/esb

Java. n.d. What is Java technology and why do I need it? Java-sivuston artikkeli. Luettu 15.1.2021.
https://www.java.com/en/download/help/whatis_java.html

Javatpoint. n.d. Java JDBC Tutorial. Javatpoint-opetussivuston kirjoitus. Luettu 24.2.2021.
https://www.javatpoint.com/java-jdbc

Johari, A. n.d. What is Java? A Beginner's Guide to Java and Its Evolution. Edureka-opetussivuston blogikirjoitus. Päivitetty 26.11.2020. Luettu 12.4.2021.
https://www.edureka.co/blog/what-is-java/

JVM-spesifikaatio. 2015. The Java® Virtual Machine Specification. Luku 1 (Introduction). Oracle-sivuston Java-dokumentaatiosivu. Julkaistu 13.2.2015. Luettu 12.4.2021.
https://docs.oracle.com/javase/specs/jvms/se8/html/index.html

Karaf-manuaali. n.d. Apache Karaf Container 4.x – Documentation. Apache Karaf -dokumentaatiosivu. Päivitetty 4.11.2020. Luettu 28.1.2021.
http://karaf.apache.org/manual/latest/

MariaDB. n.d. What is MariaDB Galera Cluster? MariaDB-sivuston dokumentaatiosivu. Luettu 25.1.2021.
https://mariadb.com/kb/en/what-is-mariadb-galera-cluster/

Oracle. 2009. Oracle Buys Sun. Oracle-sivuston lehdistötiedote. Julkaistu 20.4.2009. Luettu 11.4.2021.
https://www.oracle.com/corporate/pressrelease/oracle-buys-sun-042009.html

PreparedStatement-luokan dokumentaatio. n.d. Oracle-sivuston Java-dokumentaatiosivu. Luettu 12.4.2021.
https://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html

Saive, R. 2020. What is MariaDB? How Does MariaDB Work? Tecmint-asiantuntijasivuston blogikirjoitus. Julkaistu 25.8.2020. Luettu 20.1.2021.
https://www.tecmint.com/what-is-mariadb-how-does-mariadb-work/

ServiceMix. n.d. Apache ServiceMix -tuotteen kotisivu. Luettu 12.4.2021.
https://servicemix.apache.org/

Stack Overflow. 2020. Stack Overflow:n vuotuinen kehittäjäkysely. Luettu 20.3.2021.
https://insights.stackoverflow.com/survey/2020

Tyson, M. 2020. What is OSGi? A different approach to Java modularity. InfoWorld-asiantuntijasivuston artikkeli. Julkaistu 13.8.2020. Luettu 27.1.2021.
https://www.infoworld.com/article/3543072/what-is-osgi-java-modularity-with-the-open-service-gateway-initiative.html

What is Camel? n.d. Apache Camel -työkalun dokumentaatiosivu. Luettu 18.1.2021.
https://camel.apache.org/manual/latest/faq/what-is-camel.html

What is Maven? n.d. Apache Maven -työkalun dokumentaatiosivu. Luettu 21.1.2021.
https://maven.apache.org/what-is-maven.html

WSO2. n.d. Analyzing WSO2 ESB with the Analytics Dashboard. WSO2-tuotteen dokumentaatiosivu. Luettu 12.4.2021.
https://docs.wso2.com/display/ESB500/Analyzing+WSO2+ESB+with+the+Analytics+Dashboard

Wähner, K. 2011. When to Use Apache Camel? DZone-asiantuntijasivuston artikkeli. Julkaistu 5.6.2011. Luettu 5.3.2021.
https://dzone.com/articles/when-use-apache-camel