Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Julius Koljonen

Ajastettujen toimintojen kehittäminen toiminnanohjausjärjestelmään

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

7.1.2019

Tekijä Otsikko Sivumäärä Aika	Julius Koljonen Ajastettujen toimintojen kehittäminen toiminnanohjausjärjestelmään 45 sivua 7.1.2019
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Sovellusarkkitehti Pasi Nurmenaho Lehtori Vesa Ollikainen

Insinöörityön tavoitteena oli kehittää ajastettuja toimintoja ajava ohjelma toiminnanohjausjärjestelmän rinnalle. Ajastetut toiminnot ovat eräajoja, joille on määritelty aikataulu, milloin kyseinen eräajo pitää ajaa. Työn tavoitteena oli kehittää ohjelma, joka pystyy ajamaan ajastetusti eräajoja multitenant-ympäristössä hyödyntäen yrityksen olemassa olevaa liiketoimintalogiikkaa. Työn toimeksiantajana on Suomen Cobra Systems Oy, joka tarjoaa elintarvikeyrityksille toiminnanohjausjärjestelmiä ja niiden käyttöön tarvittavaa tukea ja konsultointia.

Toimeksiantajan asettamien rajoitteiden mukaisesti työ tultiin toteuttamaan Java- ja Spring-ympäristössä. Työn alkupuolisko koostui käyttöönotettavien kirjastojen ja ohjelmistokehysten selvitystyöstä. Selvitystyön tuloksena päädyttiin työssä käyttämään Quartz-ajastuskirjastoa ja Spring Batch -eräajoohjelmointikehystä. Kyseisiä kirjastoja hyödyntäen kehitettiin ohjelma, joka täyttää suurimmalta osin sille asetetut tavoitteet. Koska eräajoja ajava ohjelma on erillinen, se tarjoaa REST-rajapinnan toiminnanohjausjärjestelmän käytettäväksi.

Työn toisena tavoitteena oli integroida työssä kehitetty ohjelmisto käytettäväksi toiminnanohjausjärjestelmässä. Toiminnanohjausjärjestelmän näkymät olivat jo ennen työn aloitusta toteutettu Vaadin-ohjelmistokehyksellä, jolloin integraatioon tarvittavat näkymät toteutettiin myös Vaadinia hyödyntäen. Työssä kehitetyn ohjelman ja ERP-sovelluksen välinen kommunikaatio toteutettiin HTTP-pyyntöjen kautta. Työssä perehdyttiin myös integraatiotestien laatimiseen Spring-ympäristössä.

Insinöörityön lopputuloksena saatiin kehitettyä eräajoja ajastava ohjelmisto, joka on helposti laajennettavissa ja jonka kehittämistä tullaan jatkamaan firman sisäisesti. Työstä saatu kokemus eri teknologioiden ja tekniikoiden suhteen tulee edesauttamaan tulevien Java- ja Spring-pohjaisten ohjelmien kehitystä.

Avainsanat	Multitenant, Java, Quartz, Spring Batch, eräajo, ajastaminen

Author Title Number of Pages Date	Julius Koljonen Development of scheduled batch processing for enterprise resource planning system 45 pages 7 January 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Pasi Nurmenaho, Software Architect Vesa Ollikainen, Principal Lecturer

The goal of the thesis was to develop an application that performs scheduled batch processing. The application runs in conjunction with an ERP (Enterprise Resource Planning) software providing functionality for running scheduled batch jobs in a multitenant environment. This thesis was made for Suomen Cobra Systems which offers mainly ERP software for food companies in Finland.

The thesis client had set several requirements for the application: the development should only be done in Java, the batch jobs run by the application should be able to re-use already developed business logic, and, preferably, the resulting application should use Spring framework. At first, the thesis work consisted of researching open source Java scheduling libraries and batch frameworks. In the end, Quartz was selected as the scheduling library whereas Spring Batch was chosen for batch processing.

The second objective was to integrate the application with the main ERP program. ERP application's views are developed using Vaadin framework, which means that the same framework needs to be used for view integration. The ERP application and the batch processing scheduler communicate via HTTP requests. The batch processing scheduler exposes a REST interface which the main ERP application would consume.

The resulting application met almost all of the requirements set by the client. Batch jobs can be scheduled, and their CRUD operations can be performed from the main ERP application. Batch jobs run fine in a multitenant environment, and new batch jobs can be easily added to the system. The development of the scheduled batch processing application is planned to continue in the future.

Keywords	Multitenant, Java, Quartz, Spring Batch, batch job, scheduling

Sisällys

Lyhenteet

1 Johdanto 1

2 Työn lähtökohdat 2

2.1 Toimintaympäristö 2
2.2 Nykytilanteen kuvaus 4
2.3 Ohjelman vaatimukset 5

3 Ajastetut toiminnot 8

3.1 Eräajot ja ajastetut toiminnot 8
3.2 Spring Batch 11
3.3 Java-ympäristön ajastuskirjastot 14
3.4 Quartz skedulointikirjasto 14

4 Ohjelman kehitys 16

4.1 Ohjelman rakenteen kuvaus ja Springin konfigurointi 16
4.2 Spring Batchin käyttöönotto ja eräajon laatiminen 17
4.3 Ajastettujen toimintojen konfigurointi multitenant-ympäristössä 18
4.4 Quartz API 20
4.5 Quartz-konfigurointi ja integrointi Spring-ympäristöön 25
4.6 Quartz- ja Spring Batch -tietokantataulut 26
4.7 REST-API 29
4.8 Tietoturva 30
4.9 Integrointi toiminnanohjausjärjestelmään 32

5 Ratkaisun arviointi 37

5.1 Ohjelman toiminnan validointi 37
5.2 Arkkitehtuuriarviointi 39
5.3 Ohjelman ja prosessin arviointi 40
5.4 Jatkokehitys 41

6 Yhteenveto 41

Lähteet 43

Lyhenteet ja käsitteet

ORM Object-relational mapping. Oliomallin mukaisen esityksen kuvaus relaatiomallin mukaiseksi esitykseksi.

API Application Programming interface eli ohjelmointirajapinta on määritelmä, jonka mukaan ohjelmat voivat kommunikoida.

ERP Enterprise resource planning. Toiminnanohjausjärjestelmä on yrityksille kohdennettu tietojärjestelmä, jolla integroidaan yritykselle oleellisia toimintoja kuten varastonhallintaa ja laskutusta.

JPA Java Persistance API on rajapinta, joka määrittelee relaatiotietokannan datan käytön Java-ohjelmissa.

Hibernate Hibernate on kirjasto, joka toteuttaa JPA:n rajapinnan määritelmät.

Maven Projektinhallinta ohjelmisto, joka kuvaa, miten ohjelmakoodi tulisi rakentaa ja mitä riippuvuuksia rakennettavalla ohjelmalla on.

Multitenant Termi jolla viitataan sovellusarkitehtuurin, jossa ohjelmisto palvelee useita asiakkaita sekä ylläpitää asiakkaiden dataa keskitetysti.

DDL Data definition language. Syntaksi jolla voidaan määrittää tietorakenteita tai skeemoja, johon muun muassa SQL-kielen CREATE-, DROP- ja ALTER-lauseet lukeutuvat.

PostgreSQL Avoimeen lähdekoodin perustuva olio-relaatiotietokantapalvelin, joka on luotettava ja sisältää paljon ominaisuuksia.

Spring Javalle suunnattu ohjelmistokehys.

Spring Batch Eräajojen laatimiseen kehitetty ohjelmointikehys.

Spring Data Ohjelmointikehys, jonka tarkoituksena yksinkertaistaa datan hakemista tietovarastosta.

Batch processing Eräajo eli ohjelma tai prosessi, joka käsittelee pitkään suuria määriä dataa.

jOOQ (Java Object Oriented Querying) Kirjasto joka generoi tietokannan rakenteesta Java-luokkia ja tarjoaa rajapinnan tietokantakyselyiden laatimiseksi Javalla.

JDBC Java Database Connectivity. Javalle laadittu rajapinta, joka määrittää standardin, miten Java-sovellus käyttää tietokantaa.

HTTP Hypertext Transfer Protocol. Verkkoselaimien ja WWW-palvelimien käyttämä tiedonsiirtoprotokolla.

REST REpresentational State Transfer. Yleinen arkitehtuurimalli rajapintojen toteuttamiseen web-palveluille.

DTO Data Transfer Object. Verkon yli välitettävä data luokka, jonka avulla kaksi tai useampi ohjelma voivat keskustella keskenään.

MDC Mapped Diagnostic Context. Lisää loki viesteihin kontekstitietoja suoritusympäristöstä, mikä auttaa virheenjäljityksessä.

JSON JavaScript Object Notation on yksinkertainen avoimen standardin tiedostomuot

1 Johdanto

Insinöörityön aiheena on kehittää asiakasyritykselle Suomen Cobra Systemsille uusi ohjelma toiminnanohjausjärjestelmän rinnalle, jolla voidaan hallita ajastettuja toimintoja kuten laskutusta ja keräilyä multitenant-ympäristössä. Suomen Cobra Systems tarjoaa elintarvikeyrityksille toiminnanohjausjärjestelmiä ja niiden käyttöön tarvittavaa tukea ja konsultointia.

Työn lopputuloksena halutaan ohjelma, jolla ajastettujen toimintojen aikatauluttaminen ja parametrisointi olisi helppoa toiminnanohjausjärjestelmän käyttöliittymältä. Käyttöliittymä toiminnanohjausjärjestelmässä on toteutettu Vaadin-ohjelmointikehyksellä, jota myös käytetään tässä työssä. Opinnäytetyössä kehitetty ohjelma ajaisi ajastetut ajot luotettavasti ja vikasietoisesti multitenant-ympäristössä. Tuotettavan ohjelman vaatimuksena on myös olemassa olevan liiketoimintalogiikan uudelleenkäyttö. Ohjelmakehityksessä käytetään ohjelmointikielenä Javaa. Aihe valittiin yrityksen tarpeen ja opinnäytetyön yhteensopivuuden mukaan.

Työn tavoitteena on kehittää toimiva ja jatkokehitettävä ratkaisu, joka tyydyttää toimeksiantajan asettamat vaatimukset ja teknologiset rajoitukset. Työssä kehitettävä ratkaisu ei saa muuttaa olemassa olevaa koodipohjaa. Työhön kuului myös kahden eri lähestymistavan arviointiratkaisun integrointi osaksi toiminnanohjausjärjestelmää vai kehitetäänkö ratkaisu omana ohjelmanaan. Kahdesta vaihtoehdosta työssä valittiin viimeisin vaihtoehto sopivimmaksi vaihtoehdoksi.

Työn vaatimuksista johtuen tuotettavan ohjelman on mahdollista kommunikoida yrityksen kehittämän toiminnanohjausjärjestelmän kanssa. Kommunikointi päätettiin toteuttaa REST-pyynnöillä, mikä tarkoitti, että tuotettavan ohjelman pitää tarjota REST-rajapinta. REST-rajapinta ratkaisee alkuperäisen ongelman eli kommunikaation tarpeen, mutta sen lisäksi se tarjoaa mahdollisuuden tuotettavan ohjelman käytölle muissa sovelluksissa alustariippumattomasti.

Työssä käsitellään, miten kehitetään Spring Boot- ja multitenant-ympäristössä ohjelma ajastettujen toimintojen hallintaan ja miten muita Springin tarjoamia kirjastoja voidaan hyödyntää kuten Spring Dataa ja Spring Securityä. Työssä tullaan käymään myös läpi

Spring Boot web-sovelluksen ja REST API -ohjelman välistä integraatiota ja kommunikointia. Lisäksi tarkastellaan myös yrityksen käyttämiä teknologioita liiketoimintalogiikan osalta (PostgreSQL, JPA/Hibernate, Spring Data), sillä kehitettävän ohjelman pitää käyttää uudelleen olemassa olevaa liiketoimintalogiikkaa.

Ennen työn aloittamista selvitettiin olemassa olevien Java-ajastuskirjastojen käyttöönottoa kuten Javan omat ajastuskirjastot sekä muita avoimen lähdekoodin kirjastoja. Lopputuloksena työssä päädyttiin käyttämään Quartz-ajastuskirjastoa sen tarjoamien ominaisuuksien vuoksi, mihin muut kirjastot eivät pystyneet vastaamaan.

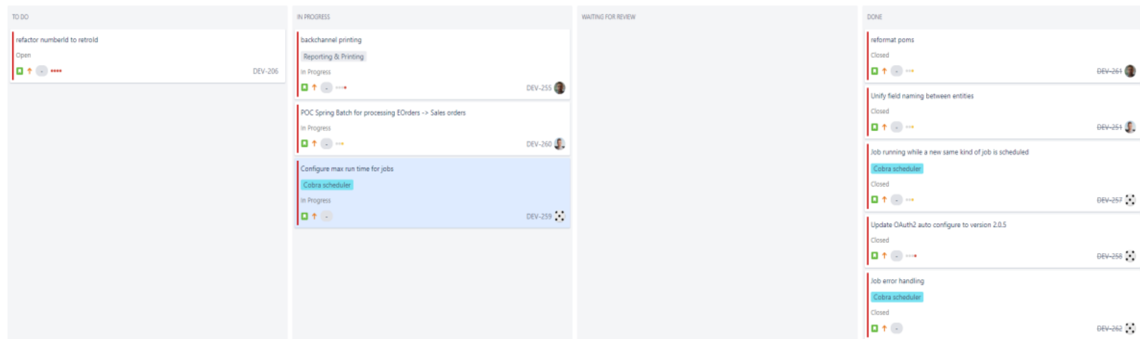
2 Työn lähtökohdat

Tässä luvussa kerrotaan yrityksen käyttämistä teknologioista, toimintatavoista sekä työlle asetetuista rajoitteista ja vaatimuksista.

2.1 Toimintaympäristö

Työ suoritetaan pääasiassa yrityksen toimitiloissa ja yrityksen antamalla kannettavalla tietokoneella. Ylläpitosyistä työn ohjelmakoodina käytetään Javaa ja kehitystyökaluna Eclipseä. Toimeksiantaja ei toivonut uuden ohjelmointikielen käyttöönottoa esimerkiksi C#:llä tai Pythonilla, sillä firman nykyinen kehitystyö toteutetaan Javalla ja uuden ohjelmointikielen tuominen kehitysympäristöön on liian suuri hinta maksettavaksi ylläpidon näkökulmasta. Tämä rajoitus tarkoittaa pääasiassa, että työssä käytettävät teknologiat kuten kirjastot ja ohjelmistokehykset ovat käytettävissä Javalla.

Kehitettävään ohjelmistoon liittyvät työtehtävät dokumentoidaan Jiralla [1]. Jira on ohjelmistokehitystiimeille suunnattu ohjelmisto ketterään ohjelmistokehitykseen.



Kuva 1. Jira sprint -lauta, jossa sprintin työtehtävät jaetaan neljään sarakkeeseen niiden edistyksen mukaan vasemmalta oikealle aloittamaton, kehityksessä, arvostelussa, suoritettu

Jira mahdollistaa työtehtävien ja sprinttien suunnittelun sekä mahdollistaa työntilaajan seurata kehitystyön etenemistä. Sprint Työn ohjelmistoon liittyvää tekninen ja muu dokumentaatio tallennetaan Atlassianin Confluence [2] -nettisivulle.

Kaikki tuotettu sovelluskoodi sijaitsee Bitbucketissa [3], joka on Atlassianin kehittämä web-pohjainen versionhallintasovellus. Firman Bitbucket-koodisäiliössä sijaitsee useita kehityshaaroja ja varsinainen päähaara ”master”, joka edustaa sovelluksen vakainta versiota. Kehitysympäristön projektinhallintaan käytetään Mavenia [4]. Maven on ohjelmisto, joka tarjoaa projektin konfigurointia, riippuvuutta ja koonnin hallintaa pom.xml-tiedoston avulla. Firman koodi on jaettu Maven-moduuleihin ja työssä kehitettävä ohjelma kehitetään uuteen Maven-moduulin. Työympäristössä näin ollen on kolme Maven-moduulia: backend, ui ja scheduler. Backend-moduuli sisältää ohjelmiston liiketoimintalogiikan, ui taas ohjelman käyttöliittymän sekä Spring-konfiguraatiot ja lopuksi scheduler-moduuli sisältää työssä kehitettävän ajastusohjelmiston lähdekoodin.

Työtä tullaan toteuttamaan tiiviissä yhteistyössä firman kehitystiimin kanssa, jolloin yrityksen työtavat tulevat olemaan osa kehitystyötä. Tähän kuuluvat varsinaisesti pull-pyyntöjen laatiminen ja koodiarvioinnit. Pull-pyyntö on menetelmä, jolla tehdyt muutokset kirjoitetaan projektin versionhallinnan päähaaraan. Pull-pyyntöön yleensä liittyy koodiarvostelu, jossa joku muu kehitystiimin jäsen tarkastaa muutosten eheyden ja standardisuuden. Epäkohdan huomatessaan arvioija kommentoi asiasta muutosten alkuperäiselle tekijälle.

2.2 Nykytilanteen kuvaus

Yrityksessä ajastettujen ajojen luonti ja parametrisointi hoidetaan Alpha Managerin [5] käyttöliittymältä. Alpha manager on 1990-luvulla Cobol-ohjelmointikielellä kehitetty kattava toiminnanohjausjärjestelmä. Alpha Manageria käytetään Linux-terminaalin kautta eli kaikki käyttäjän toiminnot mukaan lukien navigointi tapahtuvat näppäimistön kautta. Nykyisen järjestelmän iän, dokumentaation puutteen ja raskaan ylläpidon takia yritys on kehittänyt uutta toiminnanohjausjärjestelmää Javalla vuoden 2017 kesästä lähtien. Uudella järjestelmällä yritys tavoittelee iäkkäämmän järjestelmän korvaamista. Uutta järjestelmää on tarkoitus tarjota verkon yli asiakkaille.

Nykyisessä järjestelmässä ajastettujen ajojen tiedot tallennetaan indeksoituun tiedostoon, josta cron-tehtävä (ks. s. 10) tarkastaa minuutin välein, onko mitään ajettavaa. Mikäli tiedostosta löytyy työ ajettavaksi, luodaan uusi prosessi ajettavasta ohjelmasta, joka suoritetaan sille annettujen parametrien mukaisesti. Ajettavat ohjelmat ovat Alpha Managerin tavoin kehitetty Cobol-ohjelmointikielellä. Jokaisella ajettavalla ohjelmalla on vastuu omasta virhehallinnasta. Ajettavat ohjelmat lataavat indeksoidusta tiedostosta ajolle määritetyt parametrit, jotka asetetaan kuvassa 2 näkyvän käyttöliittymän kautta.

Uuden järjestelmän kehitystyössä ja ympäristössä on käytössä monia eri teknologioita. Käyttöliittymäkehityksessä käytetään Javalle suunnattua avoimen lähdekoodin Vaadin-ohjelmointikehystä [6]. Vaadin mahdollistaa helpon ja nopean käyttöliittymäkehityksen Java-ohjelmoijille, sillä se kääntää Java-koodin javascriptiksi ja html:ksi. Vaadinin keskeisimmät ideat ovat abstraktoida monimutkaiset web-teknologiat ja modularisoida käyttöliittymässä käytettäviä komponentteja ohjelmoijien tehokkuuden lisäämiseksi. Kehitysympäristössä käytetään Vaadin-versiota 10.

Ohjelman runkona käytetään suosittua Spring Boot avoimen lähdekoodin ohjelmointikehystä [7]. Spring Boot koostuu Spring-ohjelmointikehyksestä. Näin ollen se sisältää Springin tarjoamat kehitystyökalut kuten riippuvuuden injektion ja datan käsittelyn. Spring boot tarjoaa lisäksi uusia ja päivitettyjä kehitystyökaluja sekä helpon ja nopean tavan ajaa sillä kehitettyjä ohjelmia. Spring boot -ohjelmaan voi sisällyttää

sulautetun web-palvelimen esim. Tomcatin[8], mikä mahdollistaa web-sovelluksen ajamisen tavallisena Java-ohjelmana.

Tällä hetkellä kehitettävä toiminnanohjausjärjestelmä on multitenant-sovellus, eli se palvelee useita asiakkaita ja käsittelee heidän dataansa keskitetysti. Multitenant-terminologiassa yksittäistä asiakasta kutsutaan tenantiksi. Multitenant-sovellukset turvaavat asiakkaiden datan käyttäjien oikeuksien mukaan eli ts. käyttäjä A ei pääse käsiksi käyttäjän B omistamaan dataan, ellei käyttäjä B myönnä oikeuksia käyttäjälle A. Multitenant-sovellus suunnittelu alkaa tietokannasta, jolle on olemassa useita lähestymistapoja. Yleisesti multitenant-sovelluksissa käytetään yhtä strategiaa seuraavista strategioista:

- Tietokanta per asiakas eli jokaisella asiakkaalla on oma tietokantansa.

- Skeema per asiakas. Sovellus käyttää yhtä tietokantaa, mutta jokaiselle asiakkaalle luodaan oma skeema tietokantaan.

- Asiakaskohtainen tunnistesarake per taulu. Kaikki sovellukseen varastoitu data on yhden tietokannan skeemassa, mutta jokaisessa skeeman taulussa on asiakassarake, jolla tunnistetaan, kenelle kyseisen rivin data kuuluu.

Yritys käyttää yllä olevista vaihtoehdoista skeema per asiakas -strategiaa. Käytännössä jokaiselle yritykselle luodaan omat skeemat, jotka tietorakenteeltaan ovat samoja eli skeemoihin luodaan taulut samoilla DDL-lausekkeilla (Data Definition Language). DDL-lausekkeilla viitataan muun muassa SQL-taulujen luonti-, päivitys- ja poistolausekkeisiin.

2.3 Ohjelman vaatimukset

Työssä kehitettävän ohjelman pitää toiminnallisuudeltaan vastata mahdollisimman paljon Alpha Managerin ajastusjärjestelmän toiminnallisuutta. Uuden ajastettavan toiminnon konfigurointi tapahtuu vanhassa järjestelmässä kuvan 2 mukaisesti.

```
pasi@lxserver6:/u/ae1x
PASIN EINESTESTI OY AB CO          JÄRJESTELMÄHUOLTO                  26.09.2018
                                   Ajastetut toiminnot
--------------------------------------------------------------------------------

 Ajotyyppi:          KELI
 Ajopäivä:           10
 Tunnit:                    0 - 23
 Minuutit:                  0   A
 Voimassaoloaika:    00.00.00 - 99.99.99
 AM-käyttäjätunnus: super
 Voimassa (K/E):     K

 Toimituspvmväli:      0+                0+
 Reittiryhmä:        kaikki
 Reittiväli:                            ÅÅÅ
 Tilausnumeroväli:        0             999999

 Asiakasnumeroväli: kaikki              kaikki
 Asiakasryhmäväli:  kaikki              kaikki

 Valikkopaikka:
```

Kuva 2. Ajastetun toiminnon luonti Alpha Managerissa

Kaikille ajoille yhteiset parametrit ovat:

- Ajotyyppi on ajettavan ohjelman nimi.

- Aikataulun tiedot kertovat, milloin ajetaan. Ajopäivä-kenttä määrittää ajopäivän, jolle voidaan asettaa numeerisesti yksittäinen ajettavapäivä su-la 0-6, arkipäivinä ajettavat ajot saadaan arvolla 10 ma-pe, arvolla 20 la-su ja arvo 99 tarkoittaa jokaista päivää. Tunnit-kenttä kertoo välin, milloin ajetaan ja minuutit-kenttä määrittää joko minuuttikohtaisesti, milloin ajetaan tai minkä minuutein välein ajetaan.

- Voimassaoloaika-kenttä kertoo, mistä päivästä lähtien ja mihin päivään asti ajastettu toiminto on voimassa.

- AM-käyttäjätunnus -kenttä kertoo, kenen nimellä ajastettava ohjelma suoriutuu.

Alpha Managerissa pystyy määrittämään ajotyyppiä kohden eri parametreja, esimerkiksi kuvassa 2 ”Toimituspvmväli”-kentästä lähtien alaspäin olevat kentät ovat kyseiselle ajettavalla ohjelmalle ominaiset parametrit. Mikäli ajotyypin arvo vaihdetaan, vaihtuisivat samalla ajokohtaiset parametrit. Kuvan 2 mukainen konfigurointi on yksi työn tavoitteista, mutta ohjelmakohtaiset parametrit jätettiin tavoitteista pois, sillä työssä tehtävän ohjelman kehitys alkaa lähes nollasta. Työn yksi päätavoitteista on eräajojen ajaminen multitenant-ympäristössä. Työn lopputuloksena kehitetyn ohjelmiston pitää toiminnallisuuksiltaan vastata asiakkaan määrittelemiä toiminnallisuuksia ja vaatimuksia seuraavasti:

- Ohjelma on käytettävissä toiminnanohjausjärjestelmän käyttöliittymältä eli käyttäjä voi luoda, päivittää, poistaa ajastettuja toimintoja käyttöliittymän kautta.
- Eräajoja pystytään ajamaan multitenant-ympäristössä.
- Työhistoria pitää olla saatavilla.
- Eräajoja voi ajaa aikatauluttamalla tai välittömästi.
- Käynnissä olevia eräajoja voi tarkastella käyttöliittymältä.
- Ohjelma on vikasietoinen sekä ilmoittaa vika- ja poikkeustilanteista.
- Ohjelma käyttää uudelleen olemassa olevaa liiketoimintalogiikkaa.

Yksi painotetuimmista ja halutuimmista ominaisuuksista on ilmoitusten saaminen vika- ja poikkeustilanteissa. Vika- ja poikkeustilanteiden alle voidaan laskea monta eri tapausta muun muassa suoritusajan ylittyminen tai suorituksen aikana tapahtunut virhetilanne. Liiketoimintalogiikan uudelleenkäyttö taas pakottaa eräajot käyttämään jo ennalta luotuja JPA-entitejä, Spring Data -repositoreja ja PostgreSQL-tietokantaa.

3 Ajastetut toiminnot

Työn keskeisenä tavoitteena oli ajastetun eräajotoiminnallisuuden kehittäminen Javalla. Ajastetun eräajotoiminnallisuuden kehittämiseksi tutustuttiin eräajoihin ja ajastettuihin toimintoihin. Tämän jälkeen tutkittiin eri tapoja ja käytäntöjä, miten halutut toiminnallisuudet voidaan kehittää suhteellisen helposti ja nopeasti sopivia kirjastoja ja ohjelmointikehyksiä käyttämällä Javalla.

3.1 Eräajot ja ajastetut toiminnot

Batch Processing eli eräajo on ohjelma tai skripti, joka lukee, käsittelee ja kirjoittaa suuria määriä dataa. Ennen eräajon suoritusta sille yleensä annetaan parametrit ja osoitetaan lähtödatan sijainti. Eräajoille annettavat parametrit yleisesti liittyvät eräajon datan filteröintiin eli eräajoja ajettaessa ei välttämättä haluta kaikkea mahdollista tietoa käsiteltäväksi. Eräajojen ominainen piirre on itsenäinen suoriutuminen ilman käyttäjän vuorovaikutusta. Tästä syystä ne ovat oleellisessa asemassa useissa järjestelmissä, sillä ne automatisoivat liiketoimintalogiikan kannalta toistuvaisimmat ja kaavamaisimmat toiminnot. Eräajot ovatkin tästä syystä yritysjärjestelmien toiminnalle kriittisiä. Eräajojen yleisimpiin käyttötapauksiin lukeutuvat muun muassa järjestelmän tietokannasta luetun datan prosessointi ja/tai tiivistäminen, jonka lopputuloksena yleensä saadaan raportti kuten lasku tai inventaarioraportti. Eräajoihin kuuluu myös datan kirjoittaminen järjestelmään kuten tilaustietoja sisältävän csv-tiedoston lukeminen ja prosessointi järjestelmän tietokantaan. Eräajot soveltuvat erittäin hyvin myös datan migraatioprosesseihin eli lähtödatan vieminen ja konvertoiminen tietovarastosta A kohdetietovarastoon B. Edellä mainituissa käyttötapauksissa voidaan sanoa. että esimerkiksi csv-tiedoston lukeminen kantaan on datamigraatioprosessi.

Oikein toteutettuna eräajo-ohjelmat voivat tarjota hyötyjä eri kokoisille organisaatioille kuten rahan- ja ajansäästön muodossa. Eräajo-ohjelmat ovat automatisoituja, nopeampia ja paljon virheettömämpiä tehtävissään kuin ihminen. Eräajot käytännössä automatisoivat datan laadinnan, organisoinnin ja raportoinnin mahdollistaen organisaatioiden työntekijöiden nopeamman työrytmin. Eräajot ovatkin kustannustehokkaita suurten datamäärien käsittelyssä, hyvässä eräajosovelluksessa eräajoja voidaan aikatauluttaa ajamaan tietyillä ajanhetkillä. [9; 10.]

Eräajojen tarjoamien hyötyjen lisäksi ohjelmoijan täytyy huomioida mahdolliset haasteet, mitä tulee eräajoihin. Eräajot eivät ole virheettömiä. Tällöin ohjelmoijan pitää huolehtia, että eräajoille on määritelty kunnolliset ja turvalliset menettelytavat virhetilanteille. Suosituimmat eräajo-ohjelmistokehykset tarjoavat virheiden hallintaa, mutta usein ohjelmoijan vastuulle jää, miten käyttää sopivaa menettelytapaa eri virhetilanteille. Kuvitellaan tilanne, että lähtödatasta luetaan virheellinen rivi eräajon suorituksen yhteydessä. Eräajon tyypistä ja datasta riippuen se joko keskeytetään tai luettu rivi ohitetaan. Liiketoiminnalle kriittiset eräajot kuten laskutus tai tilaukset eivät saa lukea virheellistä tietoa järjestelmään tiedon korruptoitumisen vuoksi. Virheellisen lähtödatan lukemisen menettelytapa on vain yksi haasteista, mitä tulee eräajoihin. [11.]

Stephen Watts ja Junaid Rehman totesivat artikkeleissaan kolme ongelmakohtaa uusiin eräajo-ohjelmistoihin liittyen. Nämä ongelmakohdat ovat eräajo-ohjelmiston virheenjäljitys, mahdolliset kehityskustannukset ja eräajo-ohjelmiston käyttöönottoon liittyvät haasteet kuten henkilökunnan koulutus uuden järjestelmän käytöstä [9; 10]. Varsinkin virheenjäljitettävyys on haastavaa eräajo-ohjelmistoille, sillä eräajot ovat itsestäänsuoriutuvia ohjelmia, jolloin niiden suoritusta on vaikea seurata.

Michael Manilla [12, s. 3] käsittelee lyhyesti kirjassaan eräajojärjestelmien yleisiä haasteita. Michael mainitsi neljä tärkeintä haastetta, jotka ovat ylläpidettävyys/käytettävyys, skaalautuvuus, saatavuus ja tietoturva. Eräajojärjestelmän pitää olla ylläpidettävä ja virhejäljitettävä. Sen pitää myös olla skaalautuva eräajojen luonteen vuoksi. Lisäksi eräajojärjestelmien pitää mahdollisesti huomioida laskentakapasiteetin saatavuus sekä varmistua, että eräajojärjestelmät eivät esimerkiksi vuoda arkaluonteista tietoa ohjelmalokiin.

Eräajojen oleellisesta asemasta johtuen on useille ohjelmointikielelle olemassa eri kirjastoja ja ohjelmointikehyksiä eräajojen luomiseksi. Java-ekosysteemissä Spring Batch [13] on yksi tunnetuimpia ohjelmointikehyksiä eräajojen laatimiseksi. Spring batch on Springin ja Accenturen yhteistyössä kehitetty ohjelmistokehys. C#-ohjelmointikielelle löytyy muun muassa Summer Batch [14], jossa Accenturen on ollut myös mukana kehityksessä. Pythonille on olemassa suosittu Spotifyn kehittämä eräajokehitysmoduuli Luigi [15], joka pyrkii tarjoamaan kattavia ominaisuuksia pitkäkestoisille eräajoille, jotka ovat riippuvuuksien hallintaa, eräajo työputkien konfigurointia ja eräajojen tilan visualisointia.

Ajastetut toiminnot ovat määriteltävän aikataulun mukaisesti ajettavia ohjelmia, skriptejä tai komentoja. Cron on yksi esimerkki tunnetuimmasta ajastinohjelmistosta, jolla voidaan aikatauluttaa cron-syntaksin mukaisesti ajettava komento tai skripti. Cron on daemon-tyyppinen apuohjelma, eli se pyörii taustalla suorittaen sille ajastettuja tehtäviä. Windows-käyttöjärjestelmän mukana tulee myös tehtävien ajastusohjelmisto, jonka avulla voidaan suorittaa määriteltyinä aikaväleinä skriptejä ja komentoja. Yleensä ajastetuilla toiminnoilla aikataulutetaan eräajoja suoritettavaksi.

Ajastinohjelmistojen lisäksi on olemassa ajastinkirjastoja, joita ohjelmistokehittäjät voivat hyödyntää integroimalla niitä ohjelmistoprojekteihinsa. Ajastinkirjasto voi olla houkuttelevampi vaihtoehto valmiiseen ajastin ohjelmistoon verrattuna, sillä se mahdollistaa ohjelmoijan kehittää ajastettuja toimintoja, jotka suoraan hyödyntävät olemassa olevaa sovelluslogiikkaa. Kirjastojen lisäksi monissa ohjelmointi kielissä on valmiiksi ajastustoiminnallisuutta kuten Javassa ja JavaScriptissa. Ajastinohjelmiston käyttö voi taas houkuttaa nopean käyttöönoton puolesta. Tämä tosin riippuu ympäristöstä kuten mitä halutaan ajastaa.

Ajastettujen toimintojen järjestelmän kehitykseen liittyy kuitenkin riskejä, joita kehittäjän kannattaa varoa ja pyrkiä ratkomaan jo ennalta. Jarkko Tuikka [16, s.9-13] kertoo diplomityössään, mitä ongelmakohtia vaikeasti ylläpidettävässä ajastusjärjestelmässä on, mikäli se ei ole keskitetty ja standardoitu vaan hajautettu ja suunnittelematon. Diplomityössä käsiteltävän vanhan ajastusjärjestelmän ominaiset piirteet olivat:

- Huono dokumentaatio ajettavista ajoista, niiden sijainnista tiedostojärjestelmässä ja niiden suorittamista työtehtävistä.

- Ei yhdenmukaisuutta. Tehtävät toteutettu eri tekniikoilla eli osa töistä ajetaan omina prosesseinaan eli ohjelminaan ja osa taustapalveluina.

- Suoritettavat tehtävät ohjelmoitu eri ohjelmointikielillä.

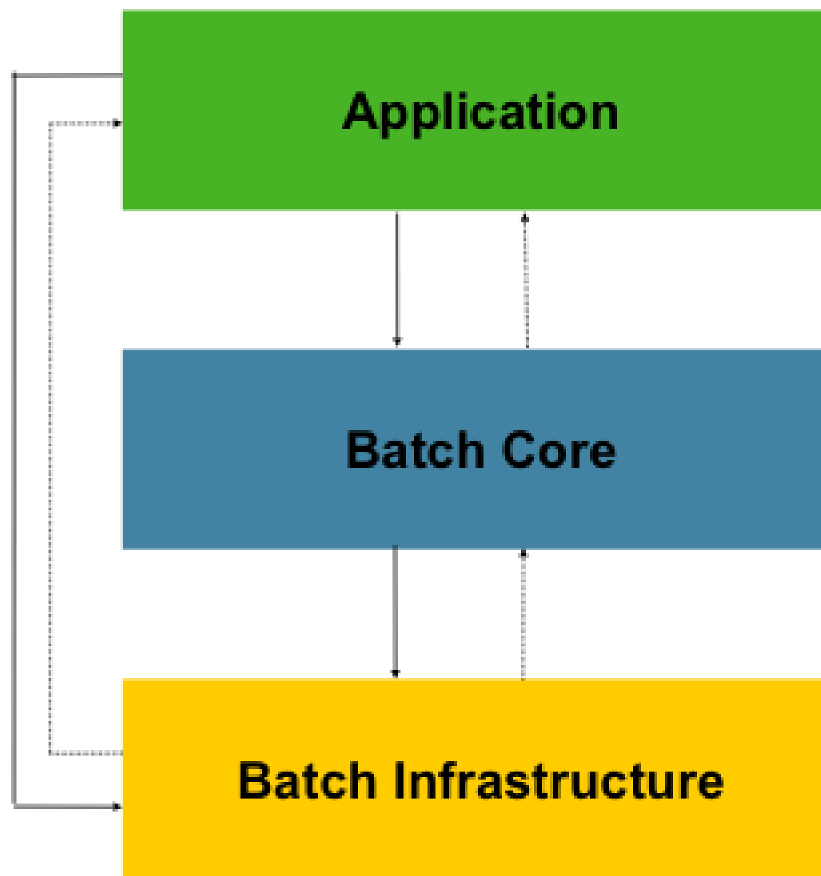
Kuvailtu järjestelmä on erittäin haastavaa ylläpitää, minkä takia ongelma ratkaistiin kehittämällä keskitetty ajastettuja toimintoja ajava järjestelmä. Tästä syystä kehittäjän pitää keskittyä yhdenmukaisuuteen luotettavilla työvälineillä ajastettuja toimintoja ajavan

ohjelman kehittämiseksi, jotta ylläpidon näkökulmasta ohjelma on helposti ymmärrettävissä ja ylläpidettävissä.

3.2 Spring Batch

Spring Batch [13] on Java-ympäristölle laadittu kevyt avoimen lähdekoodin ohjelmistokehys, joka pyrkii tarjoamaan kehitystyökalut luotettavien ja vikasietoisten eräajojen laatimiseksi. Spring Batch on kehitetty Springin (yhtiön) ja Accenturen välisessä yhteistyössä. Tarkoituksena oli luoda Spring Batchista yksi standardoiduista tavoista luoda eräajoja Javalla yritystasoisissa ympäristöissä. Spring Batch on rakennettu Spring-ohjelmistokehyksen päälle ja näin ollen Spring Batchilla laaditut eräajot hyötyvät Springin ominaisuuksista kuten riippuvuuksien injektoinnista. Spring Batch tarjoaa monia ominaisuuksia ja konfiguraatioita eräajojen laatimiseen esim. miten dataa käsitellään/muunnetaan suorituksen aikana, kuinka iso osa sisääntulodatasta prosessoidaan kerralla sekä virhetilanteiden menettelytapa. Spring Batch pyrkiikin tarjoamaan yleisimmille eräajokäyttötapauksille vaihtoehtoja ja konfiguraatioita, mikä tekee siitä ominaisuuksiltaan kattavankin ohjelmistokehyksen.

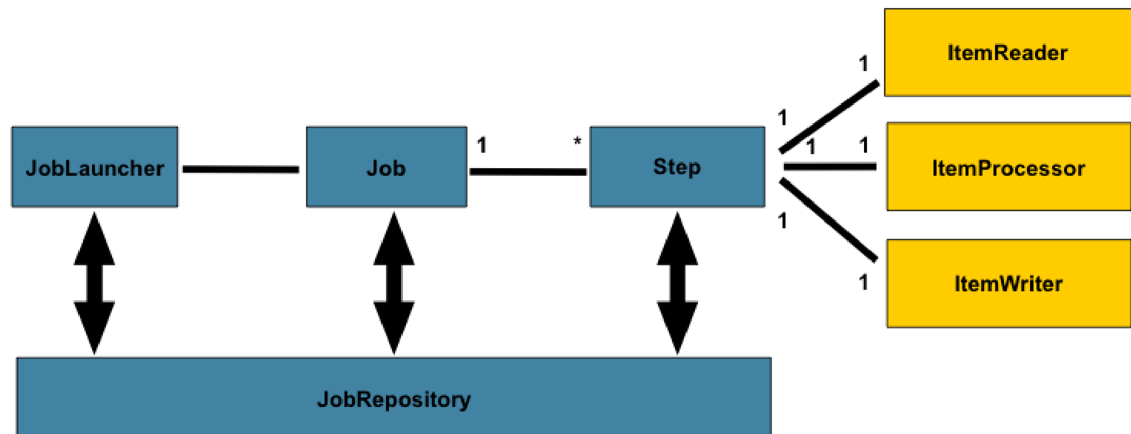
Spring Batch pyrkii tarjoamaan kehittäjille helpon tavan määrittää eräajoja abstraktion, rajapintojen ja arkkitehtuurin kautta. Batchin arkitehtuuri voidaan jakaa kolmeen kerrokseen kuvan 3 osoittamalla tavalla. Sovelluskerros (Application), joka sisältää kehittäjän kirjoittaman koodin mukaan lukien Spring Batch -ohjelmistokehyksellä laaditut eräajot. Batch Core tarjoaa eräajojen hallintarajapintaa toteuttavat luokat kuten JobLauncher, joka nimensä mukaisesti käynnistää sille annetun eräajon. Batch Infrastructure sisältää eräajojen suorituksen ja virhehallinnan kannalta oleelliset lukijat, kirjoittajat ja rajapinnat.



Kuva 3. Spring Batch yksinkertaistettu kerrosarkkitehtuuri [17]

Spring Batch ei itsessään sisällä aikataulutusta vaan aikataulutuksen lisäämiseksi on ohjelmoijan toteutettava oma toteutus tai otettava käyttöön ajastuskirjasto. Batchin dokumentaatiosta [18] voidaan havaita dokumentaation laatijoiden suosittelevan seuraavia ajastinkirjastoja: Quartz, Tivoli ja Control-M.

Spring Batchin eräajojen hallinta ja arkkitehtuuri voidaan osoittaa kuvan 4 mukaisesti.



Kuva 4. Spring Batchin yleinen arkkitehtuuri [19]. Kuvassa näkyvät keskeisimmät rajapinnat, joita ohjelmoija tulee käyttämään Spring Batchin kanssa.

JobLauncher-rajapinnan tehtävänä on käynnistää eräajoja annetuilla parametreilla. Työ (Job) on itse varsinainen eräajo, joka koostuu yhdestä tai useammasta loogisesta operaatiosta tai askeleesta (Step). Askel on yksittäinen ja itsenäinen eräajon operaatio, joka kapseloi lukijat, prosessoijat ja kirjoittajat, joita se käyttää datan hallitsemiseen. Askel voi olla toteutukseltaan yksinkertainen tai monimutkainen. Esimerkiksi datan lukeminen tiedostosta suoraan kantaan on yksinkertainen eräajo-operaatio, jonka voi toteuttaa yhdellä askeleella. ItemReader eli lukija vastaa tietolähteestä lukemisen ohjelman välimuistiin ja luetun tiedon muuntamisen ohjelman ymmärtämään muotoon. Spring Batch tarjoaa valmiita lukijoita, jotka lukevat eri tietolähteiden kautta dataa kuten JSON, Hibernate, Spring Data repositoryt ja tekstitiedosto. Kehittäjät voivat myös luoda omia lukijoita, mikäli Spring Batchin valmiit toteutukset eivät ole yhteensopivia eräajon käsittelemän datan kanssa. ItemProcessor tai prosessoija yleensä kohdistavat prosessoitavaan olioon liiketoimintalogiikkaoperaatioita tai konvertoi luokan x luokaksi y. Yksinkertaisissa stepeissä prosessoija ei ole tarpeen. ItemWriter eli kirjoittaja vastaa eräajon käsittelemän datan kirjoittamisesta määriteltyyn tietosäiliöön. Kuten lukijan kohdalla Spring Batch tarjoaa myös valmiita kirjoittajia, jotka kykenevät toimimaan samoissa tietolähdeformaateissa kuin lukijatkin. JobRepository on rajapinta, jonka kautta Spring Batch kirjoittaa ja päivittää tietokantaan eräajojen ja steppien tietoja kuten aikaleimat, suorituskontekstit ja tilat.

3.3 Java-ympäristön ajastuskirjastot

Javalle on tarjolla useita ajastuskirjastoja ja luokkia. Springin API tarjoaa myös tavan luoda ja konfiguroida ajastettuja ajoja TaskScheduler-rajapinnan tai @Scheduled-annotaation kautta. @Scheduled-annotaatio tarjoaa tavan määrittää yksinkertaisia ajastettuja toimintoja esimerkiksi ajo minuutin välein, mutta @Scheduled ei tarjoa tarvittavia edistyneitä toimintoja kuten uudelleenajastus, viimeisin ja seuraava laukaisuaika, toimintojen parametrisointi ja toimintojen pysyväistallennus. TaskScheduler on @Scheduled-annotaatiota kehittyneempi ja tarjoaa API:n ajastaa toimintoja. TaskSchedulerin rajapinnan päälle olisi mahdollista rakentaa tähän työhön vaadittava ohjelma, mutta syy sen poisjättämiseksi oli sama kuin @Scheduled-annotaatiolla edistyneiden toimintojen ja asetusten puute.

Java Source Net -sivustolta [20] selailtiin käyttökelpoisia avoimen lähdekoodin ajastuskirjastoja projektiin, joista suurin osa oli vanhoja ja ylläpitämättömiä. Db-scheduler [21] -kirjastoa kokeiltiin tutkimusvaiheessa, mutta yleisen dokumentaation ja API-dokumentaation puute sekä huonolta vaikuttava API estivät kirjaston käyttöönoton. Työluokan määrittämisessä pitää periytyä luokasta, jolla on oma konstruktori, jossa on useita parametreja ja itse suoritusmetodi on myös sekava.

Java EE (Enterprise edition) tarjoaa oman skedulointitoteutuksen, joka on varteenotettava vaihtoehto työssä käytettäväksi teknologiaksi. Java EE:n tarjoama Timer Service sisältää paljon ominaisuuksia ajastustoiminnallisuuden kehittämiseen kuten ajastustietojen pysyväistallennuksen [22]. Timer Service ei ole tosin suoraan käytettävissä Spring-ympäristössä, jonka takia sitä ei otettu käyttöön projektissa.

Tutkimusvaiheessa havaittiin, että suurin osa avoimen lähdekoodin ajastuskirjastoista oli ylläpitämättömiä ja vanhoja. Tutkituista projekteista Quartz-skedulointikirjasto oli ylläpidetty ja käytetty.

3.4 Quartz skedulointikirjasto

Quartz [23] on Javalle suunnattu ilmainen avoimen lähdekoodin skedulointikirjasto, jonka avulla voidaan laatia Java-ohjelmiin ajastettuja toimintoja. Quartz on ominaisuuksiltaan

rikas kirjasto, jonka voi integroida käytännössä mihin tahansa Java-ohjelmaan. Quartz on Terracota Oy:n kehittämä ja ylläpitämä. Quartz on Java-maailmassa suosittu ja käytetty. Siitä on myös olemassa C#-portti nimeltä Quartz.NET [24]. Quartz tarjoaa yksinkertaisen ajastuksen lisäksi edistyneitä ominaisuuksia ja toiminnallisuuksia kuten töiden parametrisointia, pysyväistallennusta ja uudelleenajastusta. Quartz-töiden ja aikataulujen pysyväistallennus mahdollistaa niiden tilan "muistamisen" sovelluksen sammuttamisen ja uudelleenkäynnistyksen yhteydessä.

Quartzin toimintaperiaate keskittyy aikataulutuksen ja varsinaisen tehtävän työn erillistäminen kahdeksi rajapinnaksi Trigger ja Job, eli ts. Trigger vastaa milloin työ tehdään ja Job miten työ tehdään. Käyttämällä Trigger- ja Job-rajapintoja voidaan määrittää ajastettuja toimintoja, jotka suorittavat määritellyn työn aikataulunsa mukaisesti. Quartzilla voi määrittää yksinkertaisia ja toistuvia aikatauluja, mutta kehittyneempien aikataulujen laatiminen tapahtuu Quartzin cronmaisen syntaksin avulla. Varsinainen työ ajetaan Quartzin työsäikeessä, eli jokainen Quartzin käynnistämä työ on asynkroninen.

Quartz on konfiguroitavissa properties-tiedoston avulla (ks. s. 25). Yleensä Quartzin yksinkertainen käyttö ei vaadi ohjelmoijalta Quartzin konfigurointia, mutta mikäli on tarvetta Quartzin edistyksellisimmille toiminnoille, konfiguraatio on tarpeen. Edellä mainittu töiden ja aikataulujen tilan pysyväistallennus on yksi esimerkki, sillä ohjelmoijan pitää osata valita oikea tietokanta ajuri Quartzille, mikäli ohjelmoija haluaa hyödyntää tietokantaan tallennettujen tietojen tarjoaman hyödyn. Oletuksena Quartz tallentaa töiden ja aikataulujen tietoja ohjelman välimuistissa, mikä on nopeampi kuin tietokantaan tallentaminen, mutta kaikki välimuistiin tallennetut tiedot katoavat ohjelman sammuttua. Näin ollen jää ohjelmoijan konfiguroitavaksi, kumpaa töiden ja aikataulutietojen tallennustapaa käytetään. Opinnäytetyössä tullaan käymään osa Quartzin konfiguraatioiden arvoista ja niiden vaikutuksista.

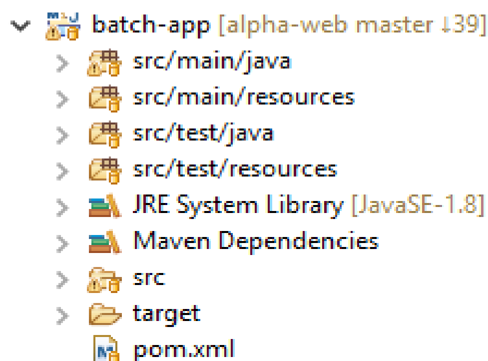
Quartzin etu muihin ilmaisiin Javan avoimen lähdekoodin ajastinkirjastoihin on valtava. Quartz on helppo integroida osaksi käytännössä mitä tahansa Java-ohjelmaa ja on suoraan käyttövalmis. Quartz tarjoaa monia ominaisuuksia perustarpeista edistyksellisimpiin ominaisuuksiin kuten klusteroinnin, jossa ajetaan useampaa Quartzilla kehitettyä ajastinohjelmistoa. Quartz on jo käytössä monissa eri projekteissa ja on suosittu kirjasto Java-sovelluksille. Tästä syystä internetissä on saatavilla paljon

oppimismateriaalia Quartzista. Muun muassa Atlassianin Confluence käyttää aikataulutuksessa Quartzia [25]. Suosionsa myötä jopa Springille on kehitetty apuluokkia Quartzin käyttöä ja integrointia varten. Edellä mainituista syistä Quartz valittiin kehitettävän ohjelman ajastinkirjastoksi sen tarjoamien ominaisuuksien, suosion, dokumentaation ja valmiin Spring-integraation takia.

4 Ohjelman kehitys

4.1 Ohjelman rakenteen kuvaus ja Springin konfigurointi

Ohjelman kehitys aloitettiin luomalla uusi Maven-projekti, jonka riippuvuuksiksi asetettiin tarvittavat Spring-moduulit (Spring Boot, Spring Security ja Spring Batch), Quartz-ajastuskirjasto sekä yrityksen back-end -moduuli, joka sisältää tarvittavan liiketoimintalogiikan. Työ organisoidaan alla olevan kansiorakenteen mukaisesti.



Kuva 5. Työn kansiorakenne. Kansio src/main/java sisältää työn ohjelmakoodin. Kansio src/main/resources sisältää ohjelmassa käytettävät properties-tiedostot Springille ja Quartzille sekä tietokannan luontiskriptit. Kansio src/test/java sisältää ohjelmalle laaditut testit ja sitä vastaava properties-kansio sisältää properties-tiedostot Springille ja Quartzille.

Kuvassa 5 nähdään, että kansiorakenne seuraa Mavenin kansiorakennestandardia.

Springin konfigurointi aloitettiin application.properties-tiedoston luomisella, jossa voidaan määritellä ohjelman konfiguraatiot avain-arvopareina. Spring hakee aina

määritetyt konfiguraatiot kyseisestä properties-tiedostosta. Spring konfiguroitiin ajamaan ohjelma portissa 8090 ja ottamalla käyttöön työssä käytettävä tietokanta eli PostgreSQL.

4.2 Spring Batchin käyttöönotto ja eräajon laatiminen

Tässä luvussa tarkastellaan, miten Spring Batchilla voidaan laatia yksinkertaisia eräajoja. Edellisessä luvussa 3.2 käytiin läpi, että Spring Batchin eräajon laatimiseksi ohjelmoijan pitää määrittää eräajolle loogiset operaatiot eli askel (Step) ja askeleen tarvitsemat apuluokat eli lukija, prosessoija ja kirjoittaja.

Spring Batchin käyttöönotto tapahtuu yksinkertaisesti asettamalla @EnableBatchProcessing-annotaatio ohjelman pääkonfiguraatioluokkaan. Kyseinen annotaatio alustaa Spring Batchin kannalta kriittiset luokat kuten JobRepository. Lisäksi seuraavaa properties-määrettä käytettiin spring.batch.job.enabled=false, kyseinen property estää Spring Batchia laukaisemasta jokaista määriteltyä eräajoa ohjelman käynnistyksen jälkeen. Spring Batch käynnistää oletuksena ohjelmakoodissa määritellyt eräajot, mutta koska eräajot halutaan ajaa käyttäjän pyynnöstä, tämä oletus poistettiin.

Yksi yleisimpiä malleja eräajon laatimisille Spring Bathiclla on yksittäisen konfiguraatioluokan luonti. Kyseinen konfiguraatioluokka käärii sisäänsä eräajon askeleeseen liittyvät apuluokat sekä itse askeleen ja työn rakentamisen. Kyseistä eräajon laatimismallia tullaan työssä hyödyntämään, sillä se eristää eräajon laatimisen yhteen luokkaan ja näin ollen helpottaa ylläpitoa tulevaisuudessa. Esimerkkikoodissa 1 havainnollistetaan, miten yksinkertainen eräajo komponentteineen voidaan määritellä yhdellä konfiguraatioluokalla.

```
@Configuration
Public class ExampleBatchConfig{

     @Bean
     ItemReader getItemReader(Tietovarasto tietovarasto)

     @Bean
     ItemProcessor getItemProcessor()

     @Bean
     Step getStep(ItemReader reader, ItemProcessor processor)

     @Bean
     Job getJob(Step step)
```

```
}
```

Esimerkkikoodi 1. Yksinkertaistettu koodiesimerkki eräajon laatimisesta. ExampleBatchConfig-luokka sisältää @Bean-annotoituja metodeja, joita Spring käyttää eräajo-olion eli Jobin luomiseen. Jobin luonnissa Spring kutsuu ensin ItemReader- ja ItemProcessor-metodeja, jotta se saa tarvittavat argumentit getStep-metodin kutsumiseen. Luotu askel välitetään getJob-metodille, joka luo Spring Batch -eräajo-olion. Laadittu eräajo sisältää vain yhden askeleen, joka lukee määritellystä tietovarastosta datan ja prosessoi luetut rivit.

Esimerkkikoodista 1 ilmenee, että kehittäjän vastuulla on eräajoon liittyvien komponenttien alustaminen, kun taas Spring-ohjelmointikehys hoitaa komponenttien riippuvuuksien injektoimisen.

4.3 Ajastettujen toimintojen konfigurointi multitenant-ympäristössä

Yksi työn tavoitteista oli liiketoimintalogiikan uudelleenkäytettävyys multitenant-ympäristössä. Tämä saavutetaan käyttämällä olemassa olevia Spring Data Repositoreja ja niitä käyttäviä Serviceitä eräajoissa. Käyttäjien StepUp ja LukLed mukaan Repositori ja Service määrittelevät kaksi eri kerrosta ja näin ollen eroavat toisistaan. Repositori määrittelee, mitä tietovarastoon kohdistettavia operaatioita on olemassa, ja Service kapseloi varsinaisen liiketoimintalogiikan, joka käyttää Repositorin tarjoamia operaatioita [26].

Eräajojen konfigurointi multitenant-ympäristöön toteutettiin käyttämällä Hibernate-kirjaston tarjoamia suhteellisen valmiita ratkaisuja. Hibernate on JPA-rajapinnan toteuttaja eli se on Javalle kehitetty ORM-kirjasto (Object-relational mapping). ORM tarkoittaa oliomallin mukaisen esityksen kuvaaminen relaatiomallin mukaiseksi esitykseksi. Tässä luvussa kerrotaan, miten Hibernate konfiguroidaan käyttämään skeemapohjaista multitenant-strategiaa. Hibernaten konfigurointi vaatii seuraavat neljä toimenpidettä:

- Multitenant strategian määrittäminen application.properties-tiedostossa.
- Multitenant-ympäristössä tietokantayhteyksiä hallinnoivan MultiTenantConnectionProvider-rajapinnan toteuttaminen.

- Hibernatelle skeemojen tunnisteita palauttava CurrentTenantIdentifierResolver-rajapinnan toteuttaminen.

- Rekisteröidä luodut toteuttajat Hibernatille properties-tiedoston kautta.

Skeemapohjainen multitenant-strategia valitaan Hibernaten käytettäväksi kirjoittamalla seuraava avain-arvopari properties-tiedostoon:

```
spring.jpa.properties.hibernate.multiTenancy=SCHEMA
```

Seuraavaksi ohjelmoijan vastuulla on luoda toteutukset MultiTenantConnectionProvider- ja CurrentTenantIdentifierResolver-rajapinnoille. Ohjelmoijan tulee olla tietoinen, että toteuttajaluokat ovat Hibernaten hallinnoimia, eivätkä Springin. Tämä käytännössä estää riippuvuusinjektion käytön toteuttajaluokissa.

Hibernate-konfiguroinnin jälkeen tarvitaan mekanismi, jonka avulla työsäie käyttää oikeaa skeemaa eräajon suorituksessa. Mekanismin pitää säilyttää skeemantunniste säiekohtaisesti, sillä työsäikeiden suoriutuminen eri skeemoissa samanaikaisesti on erittäin todennäköistä. Lisäksi CurrentTenantIdentifierResolver-rajapinnan toteuttaja ei ole Springin hallinnoima, jolloin sitä ei voi hakea Springin sovelluskontekstista skeematunnisteen asetusta varten. Ratkaisuksi osoittautui Javan tarjoama ThreadLocal-luokka, joka säilyttää sille asetetun muuttujan säiekohtaisesti. ThreadLocal säilytetään staattisena muuttujana, johon työsäie tallentaa skeeman tunnisteen ennen työn aloittamista. ThreadLocal-luokka takaa, että säikeen asettama arvo on nähtävissä vain siinä säikeessä, joka alkuperäisen arvon asetti. ThreadLocal efektiivisesti eristää säikeet toisistaan, jolloin vältytään perinteisiltä monisäikeisyyden ongelmilta. Näin ollen jokainen työsäie voi turvallisesti asettaa oman skeeman tunnisteen arvon eräajon alussa vaikuttamatta muihin säikeisiin.

Hibernate tulisi käyttämään saatua skeeman tunnistetta valittuun skeemaan kohdennetun tietokantayhteyden saamiseksi. MultiTenantConnectionProvider-rajapinnan tehtävä on tarjota ja sulkea tietokantayhteyksiä Hibernatille, jotka osoittavat tiettyyn skeemaan. Eräajojen pääsy tietokantaan menee Spring datan hallinnoimien Repository-rajapintojen läpi. Repositoreissa tietokantakyselyt yleensä määritellään muun muassa jOOQ:lla, kyselymetodeilla (Query methods) tai natiiveilla SQL-kyselyillä.

jOOQ [27] kartoittaa tietokannanrakenteen Java-luokiksi, joiden avulla voidaan luoda tyyppiturvallisia tietokantakyselyitä. Repositoreissa määritetyt kyselyt eivät ota kantaa, mihin skeemaan ne kohdistetaan, vaan se jää Hibernatin selvitettäväksi MultiTenantConnectionProvider toteuttajaa hyödyntäen. MultiTenantConnectionProvider-rajapinnan toteuttaminen ei itsessään esittänyt suuria ongelmia. Toteuttava luokka kapseloi sisäänsä javax.sql.DataSource-olion, jonka kautta haettiin tarvittavia yhteyksiä.

Tarpeellisen konfiguraation ja ohjelmakoodin tekemisen jälkeen Spring Batchilla laaditut eräajot pystyisivät hyödyntämään olemassa olevaa liiketoimintalogiikkaa multitenant-ympäristössä. Eräajot pystyisivät käyttämään olemassa olevia Spring Data Repositoreja ja Springin hallinnoimia service-luokkia. Näin ollen kaksi ohjelman kehityksen tavoitteista olisi saavutettu liiketoimintalogiikan uudelleen hyödyntäminen ja eräajojen ajo multitenant-ympäristössä.

4.4 Quartz API

Tässä osiossa käydään syvällisemmin läpi Quartzin API:a ja toiminnallisuutta. Quartzin keskeisimmät rakennuspalikat ovat Scheduler-, Trigger-, Job-, JobDetail- ja JobStore-rajapinnat. Quartz tarjoaa lisäksi kuuntelijarajapintoja, joita voidaan kytkeä Quartziin ohjelman käynnistyksen tai suorituksen aikana.

Trigger on käytännössä mekanismi, joka kertoo Quartzille, milloin siihen kytketty JobDetail pitää suorittaa. JobDetail on rajapinta, joka edustaa Job-rajapintaa toteuttavaa luokkaa. JobDetailin tehtävänä on työhön liittyvän datan kapselointi. Triggeri voi olla kytkettynä vain yhteen JobDetailiin, mutta yhdellä JobDetaililla voi olla useita Triggereitä eli käytännössä Triggerin ja JobDetailin välillä vallitsee yhden suhde moneen -yhteys. Triggereillä voi myös välittää yksinkertaisia parametreja niiden laukaisimille Job-rajapinnan toteuttaneille työluokille. Tämä onnistuu JobDataMapin avulla, joka hyväksyy yksinkertaisia avain-arvopareja esim. primäärilukuja ja merkkijonoja. Koska Triggereitä voi olla montakin, Quartz lajittelee ne TriggerKey-luokan avulla, joka on käytännössä kahden merkkijonon yhdistelmä Triggerin nimi ja Triggerin ryhmä. Triggerin nimi on yksilöivä Triggerin ryhmän sisällä eli ts. voi olla olemassa kaksi triggeriä samalla nimellä vain, jos ne ovat eri ryhmissä. Trigger itsessään on rajapinta, ja näin ollen Quartz tarjoaa

valmiita trigger-toteutuksia, jotka tarjoavat eri tapoja ajastuksen määrittämiseksi. Työssä käytetään Quartzin tarjoamaa CronTrigger-luokkaa, jolla voidaan määrittää aikataulutus Quartzin cron-syntaksilla [28]. CronTriggeriin liittyvä TriggerKey-luokka määrittyy seuraavasti: Triggerin nimi on käyttäjän antama merkkijono ja Triggerin ryhmä on skeeman nimi, joka asetetaan riippuen siitä, millä Tenantilla tai yrityksessä käyttäjä on aktiivinen Triggeriä luotaessa.

Quartz määrittelee Job-rajapinnan, jonka Java-luokan pitää toteuttaa, mikäli se halutaan ajastaa. Job-rajapinta asettaa vain kaksi vaatimusta toteuttavalle luokalle. Luokassa pitää olla tyhjä konstruktori ja luokan pitää toteuttaa execute-metodi, johon voi käytännössä ohjelmoida mitä tahansa toiminnallisuutta. Execute-metodi on varsinainen metodi, jota kutsutaan Quartzin työsäikeessä, kun Job-rajapinnan toteuttava Java-luokka pääsee suoritettavaksi. Esimerkkikoodi 2 havainnollistaa miltä yksinkertainen Quartz-työluokan toteutus näyttää.

```
public class ExampleJob implements Job {
      @Override
      public void execute(JobExecutionContext context) throws JobExecutionEx-
      ception {
              //Haluttu toiminnallisuus tähän
      }
}
```

Esimerkkikoodi 2. ExampleJob-luokka toteuttaa Quartzin Job-rajapinnan jolloin se on ajettavissa Quartzin työsäikeessä. Execute-metodi ajetaan työsäikeessä.

Execute-metodi saa argumenttina JobExecutionContext-olion, joka nimensä mukaisesti kapseloi suoritusympäristön tietoja ja suoritukselle annettuja parametreja. JobExecutionContext-oliolta saa MergedJobDataMap-rajapinnan, joka on yhdistelmä JobDetailin ja Triggerin JobDataMap-olioista. JobDataMapit toimivat Map-tietorakenteen mukaisesti eli avain-arvo-pareina ja niiden tarkoituksena on säilyttää Triggerille tai Jobille annettavia parametreja.

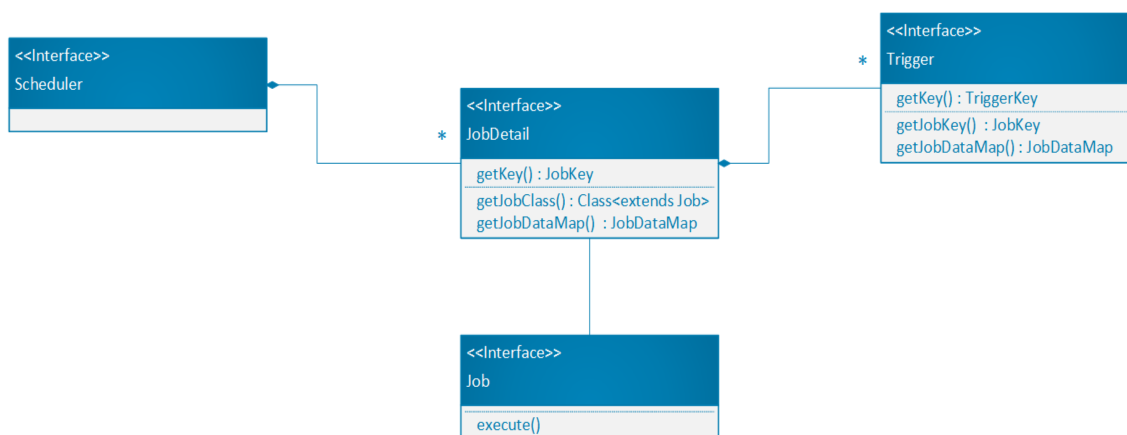
Job-rajapinnan toteuttava luokka pitää ensin rekisteröidä Quartzin skedulerille ennen suoritukseen pääsemistä. Rekisteröinnissä luodaan JobDetail-instanssi, jolle annetaan suoritettavan luokan nimi, työkohtaiset parametrit sekä tunniste eli JobKey-luokka, jonka toimintaperiaate on samanlainen kuin viimeksi mainitulla TriggerKey-luokalla.

```
JobDetail job1 = JobBuilder.newJob(ExampleJob.class)
```

```
.withIdentity("jobName1", "jobGroup1")
.build();
```

Esimerkkikoodi 3. Kuvassa oleva koodi rakentaa uuden JobDetail-instanssin, jolle annetaan suoritettavaksi työluokaksi ExampleJob, JobKeyn nimeksi jobName1 ja JobKeyn ryhmäksi jobGroup1

Varsinainen ajastettujen toimintojen hallinta kuten rekisteröinti ja poisto tapahtuu Quartzin tarjoaman Scheduler-rajapinnan kautta. Scheduler-rajapinta on käytännössä Quartzia käyttävän ohjelman tapa kommunikoida Quartzin kanssa. Quartzin kehittäjät suosittelevatkin kaiken töiden ja laukaisijoiden hallinnan tapahtuvan Scheduler-rajapinnan kautta. Tässä työssä kehitettävä ohjelma tulee myös kommunikoimaan Quartzin kanssa Scheduler-rajapinnan kautta.



Kuva 6. Quartzin olennaisimmat rajapinnat ja luokat yksinkertaistetusti. Scheduler-rajapinta tarjoaa metodeja Jobdetailien ja Triggereiden hallinnointiin. JobDetail edustaa ajettavaa työtä ja siihen liittyviä parametreja. Job-rajapinta edustaa varsinaista ajettavaa työtä. Trigger liittyy yhteen JobDetail instanssiin ja määrittää, milloin kyseisen JobDetailin Jobia ajetaan.

Quartz tarjoaa Job- ja Trigger-rajapintojen elinkaaritapahtumille kuuntelijarajapintoja, joilla voidaan pitää kirjaa ohjelman suorituksesta. Nämä rajapinnat ovat JobListener ja TriggerListener. JobListener-rajapinta kuuntelee töihin liittyviä elinkaaritapahtumia kuten työn aloittaminen, työn vetoaminen (työ ei päässyt suoritukseen) ja työn suoriutuminen. Esimerkkikoodi 4 näyttää JobListener-rajapintaan kuuluvat metodit.

```
public interface JobListener {
      String getName();
      void jobToBeExecuted(JobExecutionContext context);
      void jobExecutionVetoed(JobExecutionContext context);
      void jobWasExecuted(JobExecutionContext context,
```

```
            JobExecutionException jobException);
}
```

Esimerkkikoodi 4. JobListener-rajapinta, johon on määritelty kolme elinkaari metodia, joita Quartz-vuorottaja kutsuu tilanteen mukaisesti. GetName-metodin tulee palauttaa kuuntelijalle uniikki merkkijono eli Quartz-vuorottajaan ei saa rekisteröidä kahta kuuntelija, jotka palauttavat saman merkkijonon.

JobListener-rajapinta määrittelee merkkijonon palauttavan getName-metodin, jonka tarkoituksena on erottaa toteuttavaluokka muista kuuntelijoista. Työssä käytettävien kuuntelijaluokkien getName-metodi toteutettiin palauttamalla konkreettisen kuuntelija luokan nimi. TriggerListener-rajapinta toimii käytännössä samalla tavalla kuin JobListener erona on, että TriggerListener nimensä mukaisesti kuuntelee Triggereiden elinkaari tapahtumia. Esimerkkikoodissa 5 näytetään, mitä Quartzin TriggerListener-rajapintaan kuuluu.

```
public interface TriggerListener {
    String getName();
    void triggerFired(Trigger trigger, JobExecutionContext context);
    boolean vetoJobExecution(Trigger trigger, JobExecutionContext context);
    void triggerMisfired(Trigger trigger);
    void triggerComplete(Trigger trigger, JobExecutionContext context, Com-
pletedExecutionInstruction triggerInstructionCode);
}
```

Esimerkkikoodi 5. TriggerListener-rajapinta, joka kuuntelee Triggereiden elinkaari tapahtumia. Rajapinnan toteuttajalla on sama velvollisuus kuin JobListener-toteuttajalla eli getName-metodin tulee palauttaa uniikki merkkijono Quartz-vuorottajalle.

Kehittäjän on hyvä ottaa huomioon kuuntelija rajapintojen kutsumisjärjestys. Tämä yksityiskohta on oleellista tietää varsinkin silloin, jos halutaan upottaa omaa toiminnallisuutta Quartziin. Taulukko 1 havainnollistaa, missä järjestyksessä Quartz kutsuu kuuntelijoiden metodeja onnistuneessa työn suorituksessa.

Taulukko 1. Quartz-kuuntelijoiden kutsumisjärjestys. Vasen sarake kertoo kuuntelija rajapinnan nimen ja oikea sarake kyseisen rajapinnan metodin nimen.

Kutsumisjärjestys on ylhäältä alaspäin eli ensimmäisen rivin metodi kutsutaan ensimmäisenä.

Rajapinnan nimi	Kutsuttava metodi
TriggerListener	triggerFired
TriggerListener	vetoJobExecution
JobListener	jobToBeExecuted
JobListener	jobWasExecuted
TriggerListener	triggerComplete

Kuuntelijat ovat hyödyllisiä, sillä ne tarjoavat ohjelmoijille mahdollisuuden asettaa omaa logiikkaa ja vaikuttaa Quartz-vuorottajan työnkulkuun. Esimerkiksi TriggerListenerin vetoJobExecution() -metodi palauttaa totuusarvon, jonka perusteella Quartz-vuorottaja päättää, mikäli lauenneeseen Triggeriin liittyvä Jobi pääsee suoritukseen. Kuuntelijoiden tarjoamia mahdollisuuksia hyödynnettiin haluttujen toiminnallisuuksien kehityksessä. Halutut toiminnallisuudet, jotka toteutettiin kuuntelijoita hyödyntäen ovat maksimisuoritusaika, työn virheellisestä suorituksesta johtuva tulevien samojen töiden estäminen ja samojen töiden rinnakkainen suoritusesto.

Quartz käyttää JobStore-toteutusta töiden ja aikataulutietojen tallentamiseen. Viitaten lukuun 3.4 JobStoren määrittäminen kertoo Quartzille, millä tavalla sen täytyy tallentaa tietoa. RamJobStore tallentaa kaikki Quartzin tiedot välimuistiin ja näin ollen on erittäin nopea, mutta tiedot katoavat ohjelman sammumisen yhteydessä. JDBCJobStore

määrittää tiedon tallentamisen JDBC-rajapinnan kautta. JDBCJobStore otettiin projektissa käyttöön, koska työn vaatimuksena oli eräajojen ja aikataulutietojen tallentaminen. Koska tiedon tallennus kulkee JDBC-rajapinnan kautta, pitää määrittää Quartzille delegaattiajuri, joka vastaa halutun tietokannan JDBC-toiminnoista. Työssä käytetään PostgreSQL-tietokantaa, niin sitä vastaava Quartzin tarjoama PostgreSQL JDBC -delegaatti (org.quartz.impl.jdbcjobstore.PostgreSQLDelegate) otettiin käyttöön.

4.5 Quartz-konfigurointi ja integrointi Spring-ympäristöön

Quartz on mahdollista konfiguroida properties-tiedostolla. Quartzin sivuilta löytyy dokumentaatiota Quartzin konfiguroimiseen [29]. Ohjelmassa Quartzin-konfiguraatiotiedoston nimi on quartz.properties, ja se sijaitsee src/main/resources-kansiossa. Kuvassa 7 havainnollistetaan, miltä Quartz properties -tiedosto näyttää.

```
1 #These properties are loaded by QuartzSchedulerConfiguration
2
3 #Quartz
4 org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
5 org.quartz.threadPool.threadCount = 4
6 org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread = true
7
8  #specify the jobstore used
9  #unneeded?
10 org.quartz.jobStore.driverDelegateClass = org.quartz.impl.jdbcjobstore.PostgreSQLDelegate
11 org.quartz.jobStore.useProperties=false
12
13 #quartz table prefixes in the database. NOTE! When Transaction errors occur verify this value is correct
14 org.quartz.jobStore.tablePrefix = batch_app.qrtz_
15 org.quartz.scheduler.misfirePolicy =  doNothing
16 org.quartz.jobStore.misfireThreshold = 1000
17 #misfireThreshold uses milliseconds
18 org.quartz.jobStore.isClustered = false
```

Kuva 7. Esimerkki Quartz properties -tiedostosta.

Tiedostossa määritellään oleellisia asioita kuten työsäikeiden määrä, tietokanta-ajuri Quartz tietojen pysyväistallennukselle ja laukaisemattomuuden kynnys (misfireThreshold), joka on millisekunneissa. Quartz properties -tiedosto ladataan sovellukseen käynnistyksen yhteydessä QuartzSchedulerConfiguration-luokassa, jossa luodaan uusi SchedulerFactory, joka rakentaa uuden Quartz Scheduler -instanssin Quartz properties -tiedoston ja Springin tietojen pohjalta. Luodun Quartz Schedulerin tehtävänä on hallita ja ajaa QuartzBatchJob-luokkaa, joka on projektissa käytetty Quartz-työluokka.

Spring Batchilla luodut eräajot käynnistetään QuartzBatchJob-luokassa. QuartzBatchJob on työssä kehitetty luokka, joka toteuttaa Quartzin Job-rajapinnan eli sillä on execute-metodi, joka ajetaan työsäikeessä. Execute-metodin suoritus voidaan jakaa kolmeen vaiheeseen. Ensimmäisessä vaiheessa ladataan eräajoon liittyvät parametrit, toisessa vaiheessa ladatut parametrit asetetaan ThreadContextiin ja Spring Batchiin ja viimeisessä vaiheessa Spring Batch -eräajo käynnistetään. QuartzBatchJobista luodaan uusi instanssi aina, kun uusi eräajo pitää ajaa.

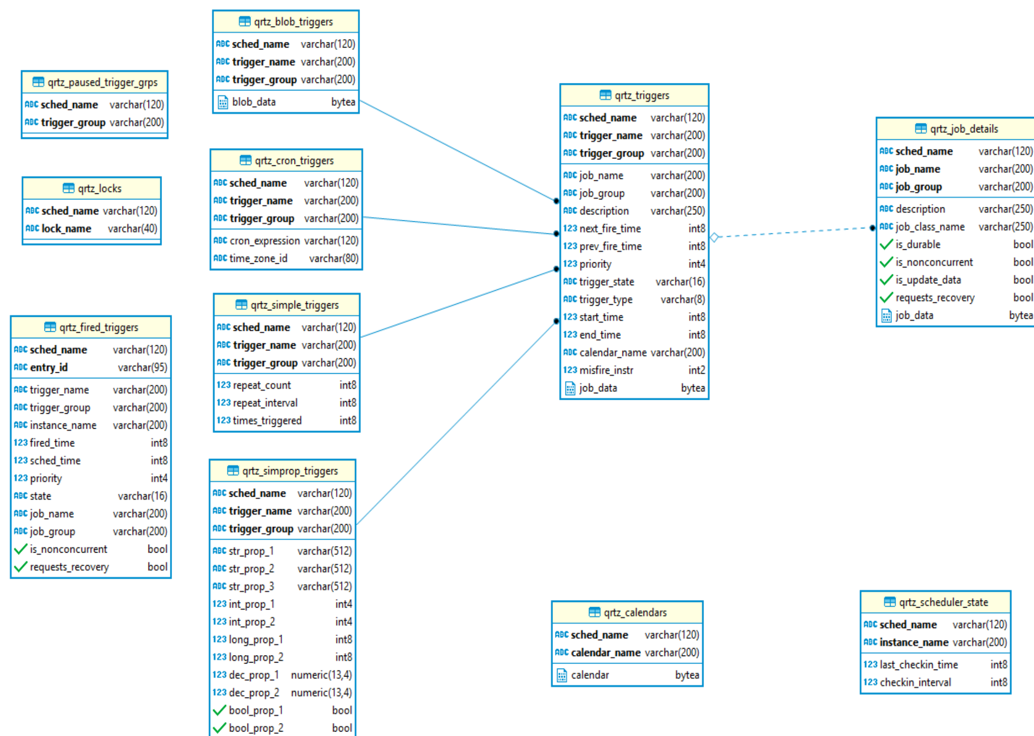
Uuden QuartzBatchJobin luonnin yhteydessä on tärkeää injektoida tarpeelliset Spring Batch -komponentit Quartz-työluokkaan, mutta Spring ei tarjoa tälle suoraa tukea. Toimivaksi osoittautuneen ratkaisun esitti Brian Matthews. Hän loi uuden luokan, joka periytyy SpringBeanJobFactory-luokasta ja toteuttaa ApplicationContextAware-rajapinnan [30]. Ideana on luoda luokka, joka luo uusia Quartz-työluokkia ja saa vastaan Springin hallinnoiman sovelluskontekstin, josta se voi hakea tarvittavat oliot riippuvuusinjektiota varten.

Tässä vaiheessa on kasassa tarpeelliset luokat, kytkennät ja properties-määritykset Quartzin hyödyntämiselle, mutta tärkein vielä puuttuu. Se on luokka, joka varsinaisesti kommunikoisi Quartz API:n kanssa Scheduler-rajapinnan kautta. Tämä luokka on SchedulerService, ja se vastaa useista tehtävistä kuten eräajojen aikataulutuksesta, välittömien eräajojen ajosta, eräajojen poistosta, eräajon päivittämisestä ja eräajotietojen kartoittamisesta Quartzin ja muun järjestelmän välillä. SchedulerServicen suurimmat vastuualueet ovatkin Quartzin kommunikoinnin kanssa ja voidaankin sanoa sen olevan ohjelman sydän.

4.6 Quartz- ja Spring Batch -tietokantataulut

Työssä kehitettävän ohjelman tallentamat Quartzin ja Spring Batchin tiedot tallennetaan samaan tietokantaskeeman. Tämä ratkaisu tuntui luontevalta, sillä se keskittää Quartzin ja Spring Batchin tietovarastot yhteen skeeman. Vastaavasti Quartzin ja Spring Batchin tietojen tallentaminen toiminnanohjauksen hallinnoimiin skeemoihin ei tuntunut luontevalta, sillä se tarkoittaa sitä, että jokaiseen skeemaan joudutaan luomaan Quartz- ja Spring Batch -tietokantatauluja. Keskitetyn skeeman valinta helpottaa ja nopeuttaa myös kehitystyötä.

Quartz ei tarjoa suoraa tapaa generoida tietokantatauluja vaan ne pitää hakea manuaalisesti Quartzin github-repositoristä [31]. Quartzin github-repositorissa on saatavilla useita sql-skriptejä eri tietokannoille. Valittava Sql-skripti määräytyy luonnollisesti käytettävän tietokannan mukaan eli tässä tapauksessa PostgreSQL. Quartzin taulujen sql-skripti tallennettiin src/main/properties-kansion alle, mistä Flyway [32] käyttää kyseistä skriptiä tietokantamigraatiossa. Flyway on avoimeen lähdekoodiin perustuva tietokantamigraatio työkalu. Quartzin taulujen ER-kaavio näyttää kuvan 8 mukaiselta.



Kuva 8. Quartzin ER-kaavio

Quartzin taulujen tallentamat tiedot ja käyttötapaukset ovat seuraavanlaiset:

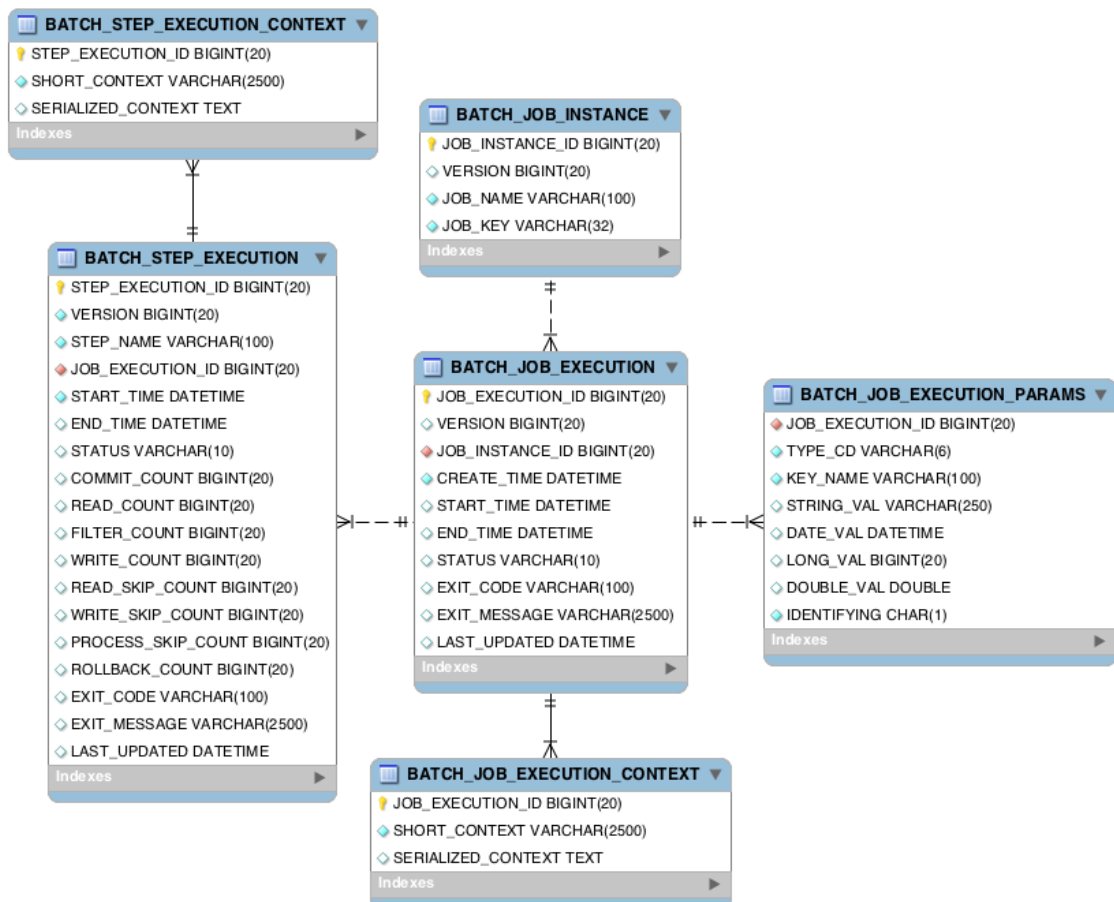
- Qrtz_triggers-taulu sisältää yleistä tietoa kaikista Quartzin triggereistä mukaan lukien Tirggereille annetut parametrit. Taulussa on tieto Triggereiden

seuraavasta ja viimeisimmästä käynnistysajasta (next_fire_time ja prev_fire_time) UNIX-aikana.

- Qrtz_cron_trigger ja qrtz_simple_trigger säilyttävät nimensä mukaisia trigger-tietoja esimerkiksi qrtz_cron_trigger-taulu sisältää Cron Triggerille ominaisia tietoja kuten Quartz syntaksisen cron-merkkijonon. Yksinkertaisen Triggerin (SimpleTrigger) tietoja säilytetään qrtz_simple_trigger-tietokantataulussa, joka sisältää SimpleTriggerin laukaisuaikavälin.

- Qrtz_job_detail sisältää töiden eli Jobien tiedot mukaan lukien annetut parametrit ja ajettavan työluokan nimi.

Spring Batch tallentaa ja päivittää eräajojen tiloja tietokantaan. Spring Batch käyttää alla olevan kuvan mukaista ER-kaaviota.



Kuva 9. Spring Batchin käyttämät taulut ER-kaaviona [33]

Kuvasta 9 voidaan huomata Spring Batchin hallinnoivan töiden ja askelten tiloja tietokannan avulla. Kyseisiä tauluja voidaan käyttää arkistointiin ja eräajohistorian lukemiseen, sillä Spring Batch ei poista ajettuihin eräajoihin liittyvää tietoa. Historiatietoja voidaan lukea ainakin kahdella tavalla. Ensimmäinen vaihtoehto on lukea suoraan tarvittavat tiedot tietokantatauluista. Toinen tapa on lukea tarvittavat tiedot Spring Batch API:n kautta. Työssä päädyttiin ensimmäiseen ratkaisuun, sillä tietokantanäkymällä saa nopeasti tietokannasta halutut historiatiedot skeeman mukaan. Tietokannan näkymä näyttää seuraavanlaiselta.

job_execution_id	tenant	exit_code	job_name	trigger_name	start_time	end_time
1	cobra	COMPLETED	READ_PRODUCTS_BATCH	ADHOC	2018-11-19 15:35:44	2018-11-19 15:35:44
2	cobra	COMPLETED	READ_PRODUCTS_BATCH	ADHOC	2018-11-19 15:35:47	2018-11-19 15:35:47
3	cobra	COMPLETED	IMPORT_EORDER_BATCHES	ADHOC	2018-11-20 09:22:06	2018-11-20 09:22:10

Kuva 10. Eräajojen historian lukemiseen käytetty tietokantanäkymä. Näkymällä haetaan ja muodostetaan koostetaulu Spring Batchin luomasta metadatasta. Näkymä hakee tarvittavat tiedot kahdesta Spring Batchin hallinnoimasta tietokantataulusta, joita ovat Batch_Job_Execution_Params ja Batch_Job_Execution.

4.7 REST-API

Työssä kehitettävä ohjelma päätettiin kehittää omaksi ohjelmakseen. Tästä on seuraavia hyötyjä kuten kuormituksen tasaaminen eli raskaita eräajoja ajava ohjelma eri palvelimella ei vaikuta toiminnanohjauksenjärjestelmän toimintaan ja suorituskykyyn. Päätökseen erillistää työssä kehitettävä ohjelma vaikutti myös toiminnanohjausjärjestelmän nykyinen arkkitehtuuri Springin istuntoon liittyen. Springin istuntoon liittyvät tiedot ovat säiekohtaisia eli ts. työsäikeet eivät pääse käsiksi näihin tietoihin. Toiminnanohjausjärjestelmän multitenant-toteutus nojaa Springin istuntoon. Näin ollen työsäikeet eivät pystyisi toimimaan kyseisessä järjestelmässä ilman raskasta ja virhealtista refaktorointia.

REST (REpresentational State Transfer) on yleinen arkkitehtuurimalli rajapintojen toteuttamiseen web-palveluille. Kehitettävän ohjelman päälle toteutettiin REST-rajapinta, joka tarjoaa päätepisteitä ajastettujen toimintojen hallintaan. Kahden järjestelmän kommunikointi tapahtuu HTTP-pyyntöjen (Hypertext Transfer Protocol)

kautta. HTTP on yleinen tiedonsiirtoprotokolla, jota verkkoselaimet ja WWW-palvelimet käyttävät. Tavoitteena on toteuttaa CRUD-operaatiot ajastetuille toiminnoille ja tämän lisäksi mahdollisuus luoda välittömästi ja kerran ajettavia eräajoja, hakea tietoa, mitä eräajoja ajetaan kyseisellä hetkellä ja eräajojen historiaa skeemakohtaisesti.

Spring-ohjelmistokehys tarjoaa ”RestController”-annotaation, joka merkitsee luokan käsittelemään HTTP-pyyntöjä sekä tekee luokan oliosta Springin hallinnoiman, mikä mahdollistaa injektioriippuvuuden. Annotoituun luokkaan voidaan lisätä REST-API päätepisteitä annotoiduilla metodeilla. Työssä määriteltiin TriggerController-luokka, joka vastasi ajastettujen toimintojen hallintaan kohdistuvista HTTP-pyynnöistä. TriggerController-luokka sisältää myös lähetetyn tiedon validointia ja virheidenhallintaa. TriggerController-luokka ohjelmointiin noudattamaan yleisimpiä HTTP-standardeja kuten palvelimen vastauskoodien ja HTTP-metodien kuten GET ja POST käyttö. Multitenant-tuelle tarpeellinen skeeman nimen arvo välitetään HTTP-pyynnön otsikkotiedoissa (header), jonka perusteella tunnistetaan, mitä skeemaa vasten kysely tehdään. TriggerController käytännössä määrittelee REST-rajapinnan ajastettujen toimintojen hallitsemiseen. Sen tärkein tehtävä on hoitaa Jackson-kirjaston [34] avulla HTTP-pyynnön mukana tulevan JSON-hyötykuorman sarjallistaminen ja purkaminen. TriggerController-luokka hoitaa myös HTTP-pyyntöjen validoinnista, virheidenhallinnasta ja HTTP-vastausten lähettämisestä tilakoodin kera. TriggerController ohjaa kelvolliset HTTP-pyynnöt SchedulerService-luokkaan, joka vastaa kommunikoinnista Quartz API:n kanssa.

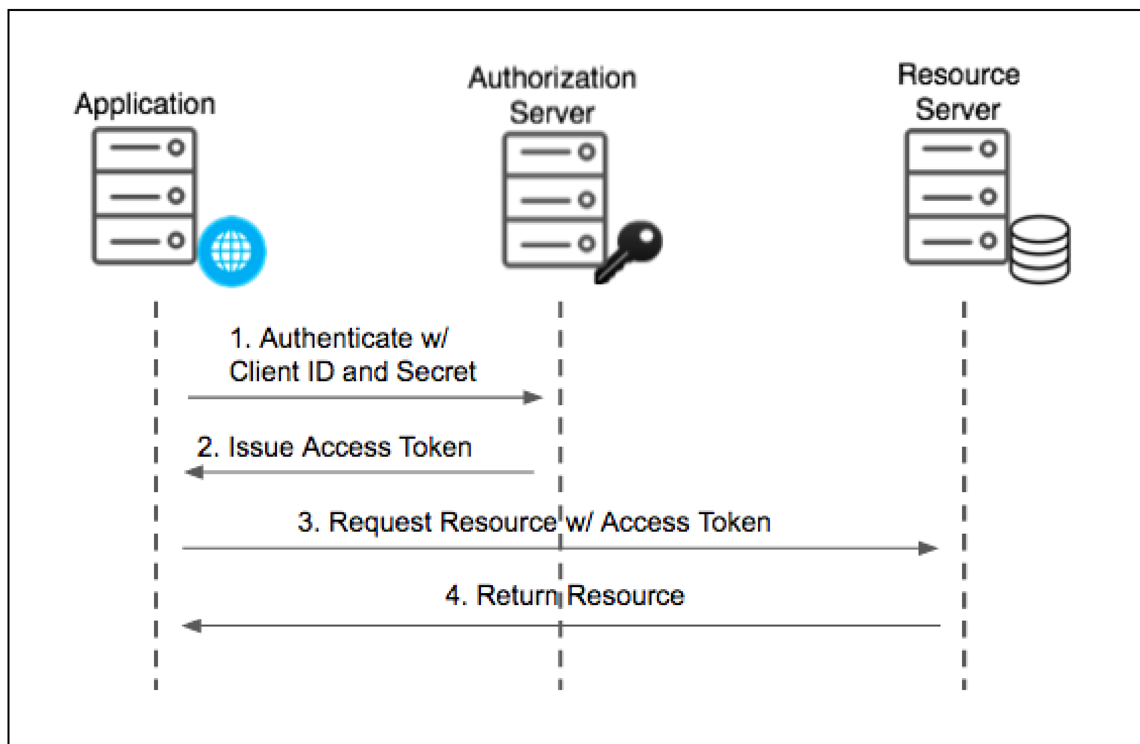
Toiminnanohjausjärjestelmä sekä ajastavia toimintoja ajava ohjelma kommunikoivat HTTP-pyyntöjen välityksellä. HTTP-pyynnössä välitetään dataa ajastetuista toiminnoista JSON (JavaScript Object Notation) -muodossa. Koska molemmat ohjelmat ovat Spring-pohjaisia voidaan molemmissa ohjelmissa hyödyntää Springin mukana tulevaa Jackson-kirjastoa JSON-hyötykuorman sarjallistamisessa ja purkamisessa.

4.8 Tietoturva

Työssä päädyttiin käyttämään OAuth2-autentikointiprotokollaa REST-API:n suojaamiseksi. Syyt OAuth2-protokollan valitsemiseksi olivat käytettävyys muissa mahdollisissa tulevissa yrityksen REST-API -sovelluksessa ja sen standardisuus.

OAuth2 tarjoaa eri auktorisointiprotokollia kuten password, code_grant, implicit ja client credentials. OAuth2-auktorisointiprotokollana päätettiin käyttää Client credentialsia, jolla vain toiminnanohjausjärjestelmä auktorisoidaan käyttämään REST-API:a. Client credentials valittiin protokollaksi, koska muuten sisäänkirjautuminen pitäisi ulkoistaa Cobra Schedulerille, jota toimeksianataja ei toivonut.

OAuth2-auktorisointiprotokollaksi valittiin OAuth2:n määrittämä ”client credentials” -protokolla, jolla pääasiassa auktorisoidaan REST-API:a käyttävä ohjelma eli tässä tapauksessa toiminnanohjausjärjestelmä.



Kuva 11. Client credentials -protokolla kuvalla havainnollistettuna [35].

Itse OAuth2-protokollan käyttöönotto Spring-sovelluksessa on helppoa. Kehittäjän tarvitsee vain lisätä Springin tarjoama OAuth2-kirjasto riippuvuudeksi. Kirjaston nimi on spring-security-oauth2-autoconfigure, joka automaattisesti konfiguroi lähes valmiin valtuutuspalvelimen (Authorization server). Kehittäjän konfiguroitavaksi jääkin OAuth2-tunnistuksessa käytettävien Client-tietojen määrittäminen ja kyseisten tietojen tallennuspaikan valinta. Kehitystyön nopeuttamiseksi Client-tiedot sekä OAuth2-valtuutuspalvelun myöntämät tokenit tallennettiin ohjelman välimuistiin. Työssä

laadittava REST-API määriteltiin resurssipalvelimeksi (Resource server), joka olisi OAuth2-valtuutuspalvelimien suojaama. Valtuutus- ja resurssipalvelin ovat osa samaa työssä kehitettävää ohjelmaa.

4.9 Integrointi toiminnanohjausjärjestelmään

Toiminnanohjausjärjestelmä perustuu Javaan ja sen avoimen lähdekoodin teknologioihin kuten Vaadiniin ja Spring Bootiin. Näin ollen tavoitteena on kehittää Vaadinilla käyttöliittymä, joka kuluttaa työssä kehitetyn ohjelman REST-rajapintaa. REST-rajapinnan käyttö edellyttää http-client -kirjaston käyttöä, mutta onneksi sekä Javan ekosysteemi sekä Spring tarjoavat hyödynnettäviä http-client -kirjastoja. Integrointi toiminnanohjausjärjestelmään oli tarkoitus toteuttaa luomalla Vaadinilla käyttöliittymä, joka kutsuisi Springin hallitsemaa Service-luokkaa, joka kapseloisi varsinaisen http-client -kirjaston.

Vaadinilla käyttöliittymän toteuttaminen oli helppoa jo olemassa olevan kokemuksen ja haluttujen näkymien yksinkertaisuuden vuoksi. Tarvittavia näkymiä tulisi olemaan kolme: ajastettujen toimintojen hallinta, eräajojen seuranta ja eräajojen suoritusloki. Jokainen näkymä tulisi olemaan ruudukkopohjainen taulukko, josta voidaan näyttää oleelliset tiedot. Ruudukon sarakkeet ovat myös lajiteltavissa. Vaadinilla ruudukkojen tekeminen on yksinkertaista.

```
Grid<ScheduledJob> scheduledJobsGrid = new Grid<>()
scheduledJobsGrid.addColumn(scheduledJobsGrid::getTriggerName).setCompara-
tor(scheduledJobsGrid::getTriggerName)
```

Esimerkkikoodi 6. Kuvassa oleva koodi luo uuden ruudukon, jossa on vain yksi lajiteltava sarake. Metodit addColumn ja setComparator ottavat parametrina funktionaalisen rajapinnan ValueProvider<S, T>, joka konvertoi lähtö olion tyyppiä S muotoon T. Kuvassa olio ScheduledJob konvertoidaan tyypiksi String. Sarakkeen arvot määräytyvät getTriggerName-metodin tuloksen mukaan.

Äskeisen esimerkkikoodin mukaan määriteltiin jokaiselle näkymälle omat ruudukot. Näkymän ruudukoissa käytettävät oliot olisivat tosin erilaiset, mikä johtaa ruudukkojen rakenteen eroamiseen näkymien välillä. Kuvassa 12 havainnollistetaan karkeasti, miltä kaikki kolme näkymää näyttävät selaimessa.

Hae ajastetut ajot | Luo ajastettu ajo | Luo välittömästi ajettava työ

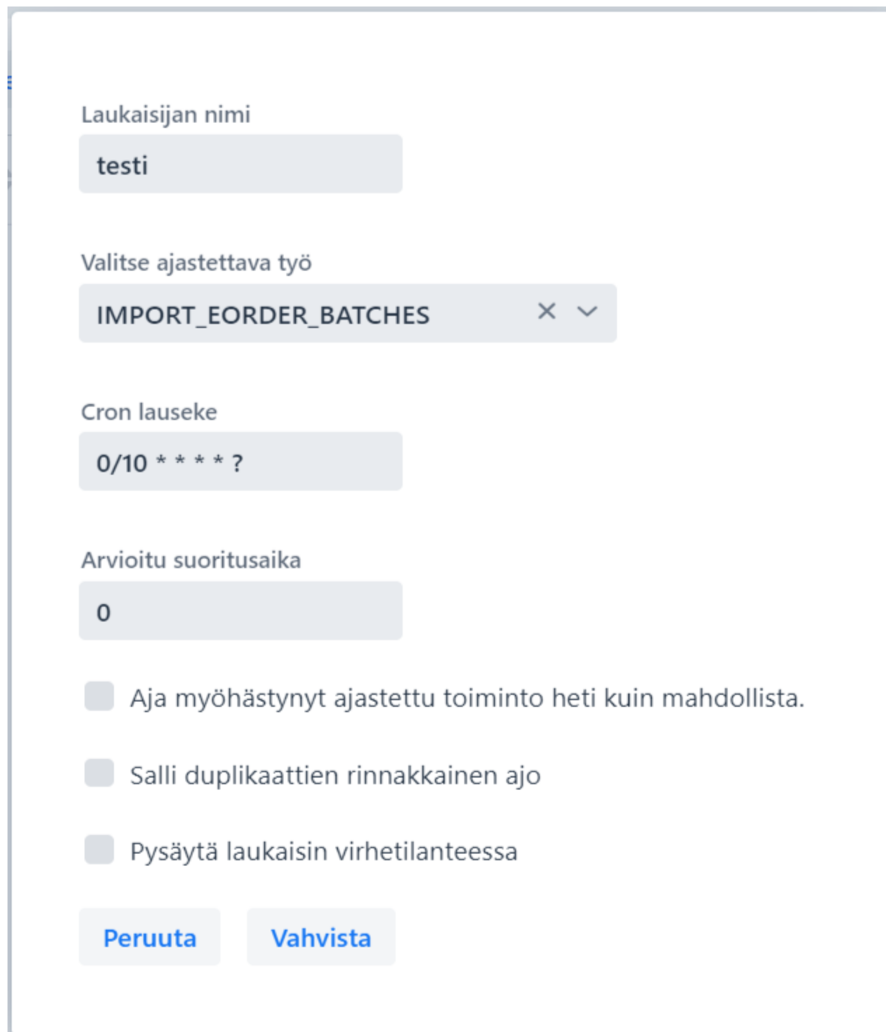
Laukaisijan nimi	Työn nimi	Tenant	Seuraava käynnistysaika	Viimeisin käynnistysaika	Laukaisijan tila	Aja myöhästynyt ajastettu toi	Salli duplikaattien rinnakkaine
process	PROCESS_EORDERS	cobra	3.12.2018 11:17:00	3.12.2018 11:16:35	NORMAL	false	false
testi	READ_PRODUCTS_BATCH	cobra	3.12.2018 11:16:40	3.12.2018 11:16:30	NORMAL	true	false

Kuva 12. Ajastettujen toimintojen hallinta näkymä. Ajastettujen toimintojen tiedot näytetään ruudukossa, jonka sarakkeet ovat lajiteltavissa.

Kahdessa muussa näkymässä tietoa välitetään käyttäjälle ruudukkoa käyttämällä viimeksi mainitun kuvan mukaisesti.

Integrointiin kuului uusien luokkien laatiminen tarkemmin uusien DTO-luokkien (Data Transfer Object) tekeminen. DTO-luokkia käytännössä käytetään tiedon siirtämiseksi verkon yli eli ajastusohjelma antaisi HTTP-pyynnön vastauksena dto-luokkia JSON-muodossa, jotka konvertoidaan pääohjelmassa Vaadin-ruudukoille käytettäviksi. DTO-luokkia luotiin kolme kappaletta: ScheduledJob, ExecutingJob ja ExecutedJob.

Osana integrointia toimeksiantaja halusi helpon dialogin, jolla voisi helposti laatia, poistaa ja päivittää ajastettuja toimintoja. Tämän lisäksi haluttiin antaa käyttäjälle mahdollisuus konfiguroida eräajoja yksinkertaisilla asetuksilla. Työssä kehitettiin seuraavanlainen dialogi eräajojen laatimiseen.



Kuva 13. Uuden ajastettavan toiminnon luonti dialogi. Käyttäjälle on saatavilla erilaisia konfiguraatiovaihtoehtoja

Kuvassa 13 näkyvät dialogin konfiguraatiovaihtoehdot voidaan havainnollistaa taulukon 2 mukaisesti.

Taulukko 2. Tässä taulukossa havainnollistetaan ajastetun toiminnon konfiguraatiodialogin valintoja ja niihin liittyviä vaikutuksia

Laukaisijan nimi	Uniikki merkkijono, joka on käyttäjän määriteltävissä. Nimi on uniikki vain yritys kohtaisesti eli ts. yrityksillä A ja B voi olla samanniminen ajastettu toiminto.
Ajastettava työ	Varsinainen Spring Batchilla luotu eräajo.
Cron-lauseke	Quartz-syntaksin mukainen cron-lauseke.
Arvioitu suoritusaika	Maksimisuoritusaika sekunteina eräajolle. Mikäli suoritusaika ylittyy, siitä tulee loki merkintä.
Myöhästynyt ajastettu toiminto ajetaan heti kuin mahdollista	Mikäli ajastettu toiminto ei jostain syystä pääse suoritukseen sille aikataulutettuna aikana, tämä asetus kertoo, mikäli se ajetaan myöhemmin vai unohdetaan.
Sallitaan duplikaattien rinnakkainen ajo	Pääseekö toinen samanlainen ajastettu ajo suoritukseen.

Pysäytetään laukaisin virhetilanteessa	Pysäytetään ajastettu toiminto, mikäli se kohtaa virhetilanteen, joka johtaa ajon kaatumiseen. Pysäytetty laukaisin ei aja eräajoja ollenkaan.

Työn alussa hyödynnettiin Springin tarjoamaa Oauth2RestTemplate-luokkaa http-pyyntöjen lähettämiseen. Oauth2RestTemplate periytyy RestTemplate-luokasta, joka abstraktoi pois http-pyyntöihin liittyvät monimutkaisuudet ja suoraviivaistaa pyyntöjen laatimista ohjelmakoodissa. Periytymisen myötä Oauth2RestTemplate tuo lisäominaisuuksia, joiden avulla voidaan hakea automaattisesti auktorisointipalvelimelta voimassa oleva token suojatun Scheduler-palvelimen käyttöä varten ja hallita ohjelmalle myönnettyjä tokeneita. Oauth2RestTemplate vaikutti alkuun hyvältä ratkaisulta, mutta kehityksen aikana huomattiin, että kyseinen luokka heittää aina ajonaikaisen poikkeuksen, mikäli palvelimelta saapuu http status koodi 4xx-5xx. Tästä syystä jokainen REST-kutsu jouduttaan käärimään try-catch -lohkoon, jossa on monta catch-lohkoa eri virhetilanteiden varalta.

Viimeksi mainitusta syystä johtuen työn kehityksen aikana perehdyttiin Retrofit2 http client -kirjastoon [36]. Tarkoituksena oli arvioida, onko kannattavaa korvata jäykältä tuntuva Oauth2RestTemplate Retrofit2:lla. Retrofit2 on suosittu Java pohjainen http-client -kirjasto, jota käytetään varsinkin Javalla toteutetuissa Android-ohjelmissa. Retrofit2:lla REST-API rajapinnat määritellään Java-rajapinnoiksi, joita voidaan hyödyntää Java-ohjelmakoodissa. Retrofitin korjasi OauthRestTemplaten jäykkyyden, sillä se ei heittänyt ajonaikaista poikkeusta, mikäli HTTP-pyynnön tila koodina oli 4xx-5xx. OAuth2 token haetaan Retrofitissä Authenticator-rajapinnan kautta, jonka toteuttaminen oli helppoa.

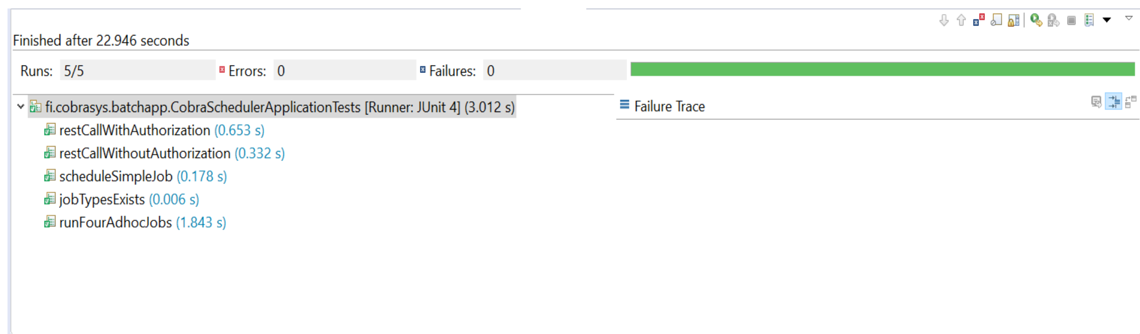
Työssä kehitettävät näkymät käyttävät Retrofit2:lla kehitettyä ScheduledJobService-luokkaa ajastettujen toimintojen tiedon ja tilan hallintaan. Tämän luokan olio injektoidaan riippuvuutena Vaadinilla kehitettyihin näkymiin. ScheduledJobService delegoi tiedon hakupyynnöt retrofit2-rajapinnalle, joka suorittaa varsinaisen http-pyynnön. ScheduledJobService välittää pyynnön vastauksen näkymälle oikeassa muodossa.

5 Ratkaisun arviointi

5.1 Ohjelman toiminnan validointi

Kehityksen aikana ohjelmaa testattiin enimmäkseen manuaalisesti, mutta muutamia automatisoituja testejä laadittiin myös. Manuaalinen testaaminen suoritettiin käytännössä asettamalla eräajoja suoritukseen joko välittömästi tai aikataulutetusti. Eräajojen suoritusta seurattiin lokituksen avulla, mikä antoi testauksen ja vianjäljityksen kannalta tärkeää tietoa kuten missä säikeessä ja millä ajanhetkellä lokitus tehtiin. Lokituksen tiedon välityksen parantamiseksi käytettiin lokituksen kanssa MDC:tä (Mapped Diagnostic Context) [37]. MDC:n ideana on parantaa lokituksen antamaa tietoa lisäämällä lokiviestiin ajonaikaisia kontekstitietoja. Koska eräajot ajetaan eri säikeissä, niillä on eriävät ajonaikaiset kontekstitiedot kuten mitä skeemaa vasten eräajo ajetaan ja mikä on ajettavan eräajoluokan nimi. MDC-konfigurointi toteutettiin Baeldungin esittämän SL4J- ja Logback-ratkaisun mukaisesti, sillä ratkaisussa käytetyt lokitusteknologiat ovat projektin kanssa samat [38]. Lokitus MDC-konfiguraation kanssa on osoittautunut tärkeäksi osaksi ohjelman toiminnan validointia, sillä sen avulla pystytään hyvin seuraamaan ohjelman ja eräajojen suoritusta kontekstitietoineen.

Manuaalisen testauksen lisäksi työssä kehitettiin testiympäristö integraatiotestausta varten. Motiivina oli luoda testejä, jotka pyörisivät mahdollisimman samanlaisessa tilassa ja ympäristössä kuin kehitysympäristö, jolloin voidaan olla suhteellisen varmoja testien luotettavuudesta. Testiympäristön kehityksessä käytettiin Springin tarjoamia työkaluja integraatiotestausta varten. Käytännössä testin alussa käynnistetään ohjelma samalla tavalla kuin normaalisti, eli testauksen aikana sovellus suoriutuu käytännössä samassa tilassa käyttäen samoja ajureita ja konfiguraatioita kuin normaalisti. Kuvasta 14 nähdään, että varsinaisia testimetodeja on viisi, joista osa testaa tietoturvaa ja osa ajastettujen toimintojen lisäämistä ja niiden souriutumista.



Kuva 14. Testit ajetaan käynnissä olevaa ohjelmaa vasten. Ylävasemmalla näkyy kokonaisaika testien suoritukselle, mikä on noin 23 sekuntia, josta suurin osa on ohjelman käynnistykseen kulunutta aikaa. Testit ovat ajettavissa omalta koneelta, mutta pääasiassa niitä ajetaan Jenkins-koontipalvelimella.

Kuva 14 havainnollistaa testien läpipääsyn testattavaa ohjelmaa vasten. Testeissä käytännössä testataan ohjelman OAuth2-suojausta, eräajo-olioiden saatavuutta Spring-sovelluskontekstista ja Quartzin ajamien eräajojen suoriutumista.

Spring tarjoaa valmiita tapoja määrittää integraatiotestejä annotaatioiden kautta. Tästä syystä testiluokka eli ”CobraSchedulerApplicationTests” annotoitiin seuraavasti esimerkkikoodin 7 osoittamalla tavalla.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.DEFINED_PORT)
@TestExecutionListeners(listeners = { TestExecutionListener.class }, mergeMode
= MergeMode.MERGE_WITH_DEFAULTS)
public class CobraSchedulerApplicationTests {
      //Testi koodia
}
```

Esimerkkikoodi 7. Annotaatioiden avulla voidaan alustaa ja konfiguroida integraatiotesti ympäristö. Lisäksi testiin liitetään oma TestExecutionListener-luokka, joka kuuntelee testien elinkaari tapahtumia. Esimerkiksi testiluokan suoriutuminen voidaan havaita afterTestClass()–metodia hyödyntäen.

Esitetyt annotaatiot esimerkkikoodissa 7 käytännössä käynnistävät ohjelman testausta varten ja antavat hyödyllisiä työkaluja testaajille kuten riippuvuuksien injektoiminen testiluokkaan ja Springin hallinnoimien Java-olioiden ylikirjoituksen.

Testien haasteellisimmat osat ovat Quartzin ajastukseen liittyvä testaaminen ja monisäikeisessä ympäristössä testaaminen. Quartz-ajastin kutsuu staattista metodia System.currentTimeMillis, joka hakee käyttöjärjestelmältä nykyisen Unix-ajan. Tämä on

ongelmallista, sillä testaaminen eri ajastuksille on erittäin vaikeaa. Esimerkiksi eräajon ajaminen tunnin välein johtaa siihen, että testi tulosta joudutaan odottamaan pahimmassa tapauksessa tunnin ajan. Tätä ongelmaa ei tosin saatu ajan puitteissa täysin ratkaistua vaan testeissä tyydyttiin ajastamaan eräajo kahden sekunnin välein.

Toinen mainituista haasteista oli testaaminen monisäikeisessä ympäristössä. Haluttiin testeissä havaita, että eräajoja ajavat säikeet suorittavat eräajot onnistuneesti ja pääsäie eli testejä ajava säie jää odottamaan eräajojen suoriutumista. Tämä ratkaistiin käyttämällä CountDownLatch-luokkaa, joka pysäyttää pääsäikeen, kunnes CountDownLatchin sisäinen laskuri saavuttaa arvon nolla. CountDownLatch-oliolle annetaan haluttujen eräajo suoritusten määrä, joita pääsäie jää odottamaan. Eräajoa ajava säie laskee laskurin arvoa yhdellä aina suorituksen päätteeksi. Testin aikana pääsäie odottaa eräajojen suoriutumista vain muutaman sekunnin ajan. Mikäli eräajojen suoritus kestää odotusaikaa pidempään, testi epäonnistuu.

5.2 Arkkitehtuuriarviointi

Ohjelma eroteltiin omaksi ohjelmakseen toiminnanohjausjärjestelmästä, mikä antoi ajastusohjelmistolle vapaammat kädet ohjelman kehitykselle ja suunnittelulle. Erottamiselle oli useita syitä, joita tässä luvussa luetellaan. Lopuksi arvioidaan lopullista ratkaisua ja arkkitehtuuria.

Mahdollisesti suurin syy siirtää ajastusohjelmisto toiminnanohjausjärjestelmästä oli toiminnanohjausjärjestelmän nykyinen tapa säilyttää tärkeitä tietoja istuntokohtaisesti Springissä ja kuinka paljon sen multitenant-konfiguraatio nojaa tähän ratkaisuun. Istuntotiedoissa, joita pidetään yllä palvelimella, sisältäisi skeeman tunnisteen, joka kertoo, minkä skeeman dataa käyttäjä käsittelee. Toiminnanohjausjärjestelmä ei ole sovellettavissa useiden eräajojen ajamiselle, koska nykyinen toiminnanohjausjärjestelmän arkkitehtuuri nojaa istunnon tietoihin ja niiden käyttöön muissa järjestelmän osissa. Lisäksi käyttäjän kirjautuessa ulos kaikki käyttäjänistuntoon liittyvät tiedot hävitetään aina, mikä on toinen ongelma, mikäli eräajoja pystytään ajamaan nykyisellä ratkaisulla. Koska toimeksiantaja toivoi mahdollisimman vähän muutoksia olemassa olevaan koodipohjaan ja nykyisen toiminnanohjausjärjestelmän

refaktorointi ajastetuille toiminnoille yhteensopivaksi on haastavaa ja äärimmäisen riskialtista, päädyttiin viimeksi mainittuun ratkaisuun eli uuden ohjelman luominen.

Uuden ohjelman luonti tuo mukanaan positiivisia puolia ja ominaisuuksia kuten kuorman jako. Mikäli ajetaan ajastusohjelmistoa eri palvelimella kuin varsinaista toiminnanohjausjärjestelmää, voidaan eräajoja ajaa omalla palvelimella vaikuttamatta toiminnanohjausjärjestelmän suorituskykyyn. Toinen positiivinen puoli on vapaammat kädet ohjelman kehitykseen liittyen, joka nopeuttaa kehitys- ja suunnittelutyötä.

Negatiivisina puolina voidaan pitää kommunikaation kehitystä eli pitää keksiä tapa, miten nykyinen ERP-järjestelmä kommunikoi ajastusohjelmiston kanssa. Retrofit2- ja REST-API-teknologioiden oppiminen ja omaksuminen Spring-ympäristössä vei aikaa, mutta kyseisiä teknologioita tullaan mahdollisesti käyttämään muissa projektin osa-alueissa.

5.3 Ohjelman ja prosessin arviointi

Työssä kehitetty ensimmäinen versio tyydytti suurilta osin toimeksiantajan asettamat tavoitteet ja ominaisuudet. Nykytilanteesta on hyvä jatkaa ajastusohjelmiston kehittämistä. Ilmoitustoiminnallisuus oli ainoa aikataulusyistä pois jätetty haluttu ominaisuus. Toimeksiantaja on ollut tyytyväinen työn lopputulokseen.

Ohjelman kehitykseen osallistui yrityksen muu kehitystiimi heidän koodiarvostelujen, rakentavien kritiikkien, palavereiden ja neuvojen avulla. Tämä tietenkin oli iso osa suunnittelu ja kehitystyötä. Varsinkin palaverit muun kehitystiimin kanssa olivat tärkeitä, sillä ne tarjosivat tavan vaihtaa ideoita työhön liittyen kuten teknisten ongelmien ratkaisemista tai uuden kirjaston käyttöönottoa. Työn kehitysprosessiin kuului myös demotilaisuuksia ja muita palavereita, joissa sai palautetta ja toimeksiantajan toiveita.

Firman sisäinen infrastruktuuri ja toimintatavat auttoivat kehitys- ja suunnittelutyössä. Jiran avulla pystyi pitämään kirjaa tarpeellisista tehtävistä, mikä antoi toimeksiantajalle tavan seurata työn edistymistä. Pull-pyyntöjen kommentointi ja koodiarvostelut sekä Jenkins-koontipalvelu edesauttoivat työn laatua.

5.4 Jatkokehitys

Tämän työn tuloksena syntynyttä ohjelmaa on tarkoitus kehittää jatkossa firman sisällä. Jatkokehityksen piiriin kuuluu muiden haluttujen ominaisuuksien kehittäminen ja testien laatiminen. Ajastusohjelmiston tuloksena halutaan Alpha Managerin vastaavat ajastettujen toimintojen konfigurointiominaisuudet. Eräajokohtaiset parametrit ovat yksi isoimmista haasteista ja ominaisuuksista jatkokehitystä ajatellen. Toinen haluttu ominaisuus oli ilmoitussysteemi, joka ilmoittaa tuelle eräajojen poikkeustilanteista kuten virhetilanteista tai pitkittyneistä suoritusajoista.

Tulevaisuudessa tullaan laatimaan enemmän uusia eräajoja Spring Batchilla. Tämä vaatii tulevaisuudessa eräajojen luomisen lisäksi niihin kohdistuvien testien luomista. Tällä hetkellä eräajojen toimivuutta on testattu seuraamalla eräajon suoritusta lokeista ja tarkastelemalla eräajon tuloksia. Tämä prosessi pitää automatisoida tulevaisuudessa, sillä jokaisen eräajon yksittäinen manuaalinen testaaminen kestää pitkään ja siihen käytetty aika kasvaa sitä mukaan, mitä enemmän uusia eräajoja luodaan.

Jatkokehityksestä vielä mainittakoon, että eräajojen luojan käyttäjätiedot halutaan tallentaa. Eli voidaan tarkastaa, kuka on luonut jonkin kyseisen eräajon. Lisäksi tulevaisuudessa halutaan tarkastella ja analysoida lokitiedostoja eräajoihin liittyen.

6 Yhteenveto

Insinöörityössä oli tarkoitus kehittää ajastettuja toimintoja ajava ohjelma kehitteillä olevan toiminnanohjausjärjestelmän rinnalle. Työllä oli tarkoitus toteuttaa toiminnallisuuksiltaan ja ominaisuuksiltaan Alpha Managerin ajastettuja toimintoja vastaava ohjelma. Työn alussa toimeksiantaja asetti rajoitteita ja vaatimuksia insinöörityössä kehitettävälle ohjelmalle, jotka työn lopputulos tyydyttää. Tärkeimmät rajoitteet ja vaatimukset olivat seuraavat:

- Uuden ohjelman tarjoamat palvelut pitää olla käytettävissä toiminnanohjausjärjestelmän käyttöliittymältä.

- Työn lopputuloksena kehitetty ohjelma käyttää uudelleen olemassa olevaa liiketoimintalogiikkaa uudelleenkäyttö eräajoissa.

- Ohjelma kykenee ajamaan eräajot multitenant-ympäristössä.

- Työssä kehitettävä ohjelma on Javalla toteutettu.

Työn alkuvaiheeseen liittyi sopivien kirjastojen ja ohjelmointikehysten etsiminen, mitkä tarjoavat paljon tarvittavia ominaisuuksia valmiiksi, ovat käytettyjä ja joilla olisi mahdollista aloittaa kehitystyö nopeasti. Valitut kirjastot ajastettujen toimintojen kehittämiseksi olivat Quartz ja Spring Batch. Quartz on ajastuskirjasto Javalle, kun taas Spring Batch oli eräajojen laadintaan tarkoitettu ohjelmistokehys. Työssä käytiin läpi, miten Quartz ja Spring Batch toimivat ja miten ne voidaan integroida keskenään Spring-ympäristössä, josta voidaan kehittää ajastettuja toimintoja ajava ohjelma. Työssä jouduttiin myös käymään läpi, miten kaksi Spring-sovellusta voivat kommunikoida keskenään. Kommunikointi päätettiin toteuttaa REST-rajapinnan avulla, jota ajastavia toimintoja ajava ohjelma tarjoaa toiminnanohjausjärjestelmän käytettäväksi.

Työn lopputuloksena saatiin halutut perustoiminnallisuudet ja ominaisuudet täyttävä ohjelmisto, jota on tarkoitus jatkokehittää firman sisällä eteenpäin. Ohjelmiston tarjoamat palvelut ovat käytettävissä toiminnanohjausjärjestelmän käyttöliittymältä ja muutkin toiminnallisuudet vastaavat toimeksiantajan vaatimuksia. Ainoastaan ilmoitussysteemin kehitys jäi ajan puitteissa toteuttamatta työstä, sillä kunnollisen ja toimivan ilmoitussysteemin kehittäminen vie paljon aikaa.

Työ on antanut paljon kokemusta eri teknologioista ja tekniikoista, sillä se koostui täysin uuden ohjelman kehittämisestä ja integroimisesta toiseen järjestelmään. Kokemus mitä tästä työstä saatiin, on mahdollisesti hyödynnettävissä muissa mahdollisissa henkilökohtaisissa tai toimeksiantajan projekteissa.

Lähteet

1 Atlassian. Jira. Verkkoainesto.<https://fi.atlassian.com/software/jira>. Luettu 10.12.2018.

2 Atlassian. Confluence.Verkkoaineisto. <https://fi.atlassian.com/software/confluence>. Luettu 10.12.2018.

3 Atlassian. Bitbucket. Verkkoaineisto. <https://bitbucket.org/product> . Luettu 10.12.2018.

4 Apache. Maven. Verkkoaineisto. <https://maven.apache.org/>. Luettu 10.12.2018.

5 Suomen Cobra Systems Oy. Alpha Manager. Verkkoaineisto. <https://www.cobrasystems.fi/ohjelmistot/alpha-manager/>. Luettu 10.12.2018.

6 Vaadin. Vaadin. Verkkoaineisto. <https://vaadin.com/>. Luettu 10.12.2018.

7 Pivotal. Spring Boot. Verkkoaineisto. <https://spring.io/projects/spring-boot>. Luettu 10.12.2018.

8 Apache. Tomcat. Verkkoaineisto. <http://tomcat.apache.org/>. Luettu 10.12.2018.

9 Watts, Stephen. 2017. What is Batch Processing? Batch Processing Explained. Verkkoaineisto. <https://www.bmc.com/blogs/what-is-batch-processing-batch-processing-explained/>. 20.7.2017. Luettu 28.11.2018.

10 Rehman, Junaid. What are advantages and disadvantages of batch processing systems. N.d. Verkkoaineisto. <http://www.itrelease.com/2012/12/what-are-advantages-and-disadvantages-of-batch-processing-systems/>. Luettu 12.12.2018.

11 Sarhan, Ashraf. 30.5.2016. Spring Batch Exception Handling Example. Verkkoaineisto. <https://examples.javacodegeeks.com/enterprise-java/spring/batch/spring-batch-exception-handling-example/>. Luettu 20.12.2018

12 Minella, Michael. 13.7.2011. Pro Spring Batch.

13 Ward ym. Spring Batch - Reference Documentation. Verkkoaineisto. <https://docs.spring.io/spring-batch/4.0.x/reference/html/index.html>. Luettu 10.12.2018.

14 Blu Age. Summer Batch. Verkkoaineisto. <https://www.bluage.com/summer-batch>. Luettu 10.12.2018.

15 Spotify. Luigi. Verkkoaineisto. < https://github.com/spotify/luigi>. Luettu 10.12.2018.

16 Tuikka, Jarkko. 2016. Ajastettuja tehtäviä suorittavan järjestelmäylläpidollisen työkalun määrittely ja toteutus. Tampere. Verkkoaineisto. <https://dspace.cc.tut.fi/dpub/bitstream/handle/123456789/23646/tuikka.pdf?sequence=1&isAllowed=y>. 1.2016. Luettu 13.12.2018.

17 Ward ym. Spring Batch Layers. Kuva. <https://docs.spring.io/spring-batch/4.0.x/reference/html/images/spring-batch-layers.png>. Luettu 13.12.2018.

18 Ward ym. 2018. Spring Batch Introduction. Verkkoaineisto. <https://docs.spring.io/spring-batch/4.0.x/reference/html/spring-batch-intro.html#spring-batch-intro>. 7.3.2018. Luettu 21.9.2018.

19 Ward ym. Spring Batch Reference Model. Kuva. https://docs.spring.io/spring-batch/4.0.x/reference/html/images/spring-batch-reference-model.png. Luettu 13.12.2018.

20 Java Source Net. Verkkoaineisto. <http://java-source.net/open-source/job-schedulers>. Luettu 9.12.2018.

21 Karlsson, Gustav. db-scheduler. Verkkoaineisto. <https://github.com/kagkarlsson/db-scheduler>. Luettu 9.12.2018.

22 Testa, Gualtiero. Java EE schedulers. Verkkoaineisto. <https://gualtierotesta.wordpress.com/2016/10/02/java-ee-schedulers/>. Luettu 12.12.2018.

23 Terracota. Quartz Job Scheduler. Verkkoaineisto. <http://www.quartz-scheduler.org/>. Luettu 10.12.2018.

24 Lahma, Marko. Quartz Enterprise Scheduler .NET. Verkkoaineisto. <https://www.quartz-scheduler.net/>. Luettu 10.12.2018.

25 Atlassian. Package com.atlassian.confluence.schedule.quartz. <https://docs.atlassian.com/atlassian-confluence/5.9.4/com/atlassian/confluence/schedule/quartz/package-summary.html>. Verkkoaineisto. Luettu 9.12.2018.

26 Nimimerkit StepUp, LukLed. 2011. Difference between Repository and Service Layer? Verkkoaineisto. <https://stackoverflow.com/questions/5049363/difference-

between-repository-and-service-layer/5049454#5049454>. 19.2.2011. Luettu 7.12.2018.

27 Data Geekery. jOOQ. Verkkoaineisto. <https://www.jooq.org/>. Luettu 20.12.2018.

28 Terracota. Cron Trigger Tutorial. Pvm ei saatavilla. Verkkoaineisto.<http://www.quartz-scheduler.org/documentation/quartz-2.x/tutorials/crontrigger.html>. Luettu 12.12.2018.

29 Terracota. Quartz Configuration Reference. Pvm ei saatavilla. Verkkoaineisto. <http://www.quartz-scheduler.org/documentation/quartz-2.x/configuration/>. Luettu 7.12.2018.

30 Matthews, Brian. 2011. Inject application context dependencies in Quartz job beans. Verkkoaineisto. <http://blog.btmatthews.com/?p=40>.24.9.2011. Luettu 7.12.2018.

31 Nimimerkit Zemian Deng ym. tables_postgres.sql. Verkkoaineisto. <https://github.com/quartz-scheduler/quartz/blob/master/quartz-core/src/main/resources/org/quartz/impl/jdbcjobstore/tables_postgres.sql>. 4.7.2017. Luettu 9.12.2018.

32 Boxfuse. Flyway. Verkkoaineisto. <https://flywaydb.org/>. Luettu 20.12.2018.

33 Ward ym. Spring Batch Meta-Data ERD. Kuva. <https://docs.spring.io/spring-batch/4.0.x/reference/html/images/meta-data-erd.png>. Luettu 20.12.2018.

34 FasterXML. Jackson. Verkkoaineisto. <https://github.com/FasterXML/jackson>. Luettu 9.12.2018.

35 Pivotal. Oauth client credentials. Kuva. <https://docs.pivotal.io/p-identity/1-4/images/oauth_client_credentials.png>. Luettu 9.12.2018.

36 Square. Retrofit. Verkkoaineisto. <https://square.github.io/retrofit/>. Luettu 12.12.2018.

37 Ceki Gülcü ym. 2017. Verkkoaineisto. <https://logback.qos.ch/manual/mdc.html>. Luettu 12.12.2018.

38 Baeldung. 2018. Improved Java Logging with Mapped Diagnostic Context (MDC). Verkkoaineisto. <https://www.baeldung.com/mdc-in-log4j-2-logback>. 4.11.2018. Luettu 8.12.2018.