Examensarbete, Högskolan på Åland, Utbildningsprogrammet för Informationsteknik

MODERNISERING AV SCHEMALÄGGARE FÖR BATCHJOBB

Conny Backas, Mikael Norrsjö

43:2019

Datum för godkännande: <11.12.2019>
Handledare: Joakim Isaksson

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Conny Backas, Mikael Norrsjö
Arbetets namn:	Modernisering av schemaläggare för batchjobb
Handledare:	Joakim Isaksson
Uppdragsgivare:	Crosskey Banking Solutions

Abstrakt

Syftet med examensarbetet var att på begäran av uppdragsgivaren modernisera en modul i deras system till en REST-tjänst med vilken resten av systemet skulle kunna kommunicera med genom HTTP-kommunikation.

Modulens ansvarsområde är att hantera olika automatiska batchjobb genom att erbjuda operationer för att lägga till, uppdatera och ta bort jobb. Den ansvarar även för att exekvera jobben när schemaläggningens villkor uppfylls.

I arbetet används huvudsakligen programmeringsspråket Java och ramverken Spring och Spring Boot samt Quartz för jobbhantering.

Nyckelord (sökord)
Java, Spring, Spring Boot, Gradle, Feign, REST, Quartz

Högskolans serienummer:	**ISSN:**	**Språk:**	**Sidantal:**
43:2019	1458-1531	Svenska	35 sidor

Inlämningsdatum:	**Presentationsdatum:**	**Datum för godkännande:**
<10.12.2019>	<04.12.2019>	<11.12.2019>

DEGREE THESIS
Åland University of Applied Sciences

Study program:	Information Technology
Author:	Conny Backas, Mikael Norrsjö
Title:	Modernization of Scheduler for Batch Jobs
Academic Supervisor:	Joakim Isaksson
Technical Supervisor:	Crosskey Banking Solutions

Abstract

The purpose of the thesis was, at the request of the client, to modernize a module in their system by breaking out the module into a REST-service which the rest of the system could communicate with through HTTP-communication.

The module's area of responsibility is to handle various automatic batch jobs by offering operations to add, update and delete jobs. It is also responsible for executing the job when the scheduling conditions are met.

The work was done by using the programming language Java and the frameworks Spring and Spring Boot as well as Quartz for job management.

Keywords

Java, Spring, Spring Boot, Gradle, Feign, REST, Quartz

Serial number:	ISSN:	Language:	Number of pages:
43:2019	1458-1531	Swedish	35 pages

Handed in:	Date of presentation:	Approved on:
<10.12.2019>	<04.12.2019>	<11.12.2019>

INNEHÅLLSFÖRTECKNING

1. INTRODUKTION

1.1 Syfte

Syftet med detta examensarbete är att på begäran av den externa uppdragsgivaren Crosskey Bankings Solutions modernisera arkitekturen för hantering av batchjobb i deras system. Ett batchjobb kan definieras som en samling databehandlingsuppdrag för bearbetning av datorn i en omgång (Svenska Akademiens Ordböcker, 2009). Dessa jobb kan t.ex. vara automatiskt genererade rapporter som skall genereras en viss tid på dygnet.

Funktionaliteten utgörs av schemaläggning av olika batchjobb som automatiskt skall kunna utföras beroende på vilka parametrar användaren har lagt till. Detta gör användaren genom ett webbanvändargränssnitt i deras backofficeportal.

I systemet finns ett färdigt bibliotek som sköter schemaläggningen av batchjobb. De moduler som skall använda sig av schemaläggaren behöver importera detta bibliotek. För att sedan kunna utnyttja biblioteket behöver dessa konsumerande moduler konfigurera schemaläggaren genom att ange datakällan och vilka mekanismer för datamappningen den skall använda. Det här leder till att de moduler som använder sig av schemaläggaren får en stark koppling mot databasen vilket om möjligt skall undvikas.

Vår uppgift är att skapa en självständig applikation som har en egen konfigurerad schemaläggare. Denna applikation skall i sin tur erbjuda ett programmeringsgränssnitt (REST API) och via en klient få in anrop som säger åt applikationen vilka operationer som skall utföras på schemaläggaren. Till vår uppgift tillhör även skapande av klientbibliotek och att byta ut den nuvarande implementationen mot den nya i resten av systemet.

Motiveringen för denna modernisering är genom att utveckla den här strukturen få mindre starka kopplingar och ett system som är enklare att uppdatera eller utöka med ny funktionalitet. När kommunikationen sker via en klient och ett REST API innebär det även i teorin att man kan använda olika programmeringsspråk för klienten och API:n.

En annan fördel med denna struktur är att om ett eventuellt fel skulle uppstå i någon av delarna avgränsas komplikationerna istället för att hela systemet påverkas.

1.2 Metod

Vi började med att bestämma eventuella ramverk och hjälpbibliotek som eventuellt kunde behövas för att bygga vår applikation och klient. Vid diskussion med systemarkitekter och uppdragsgivare kunde det konstateras att programspråket Java och ramverket Spring Boot var av betydelse för denna typ av applikation. Andra bibliotek som användes var Feign för implementation av klientbibliotek och Quartz som innehåller funktionalitet för schemaläggning.

Nästa steg var att noggrant gå igenom de tidigare modulernas implementation för att se hur de var uppbyggda för att få en bättre inblick i vad de gjorde i praktiken. Detta behövde göras för att bestämma vilka delar av de gamla modulernas funktionalitet som direkt kunde flyttas till vår applikation.

Under i början av projektet användes parprogrammering eftersom arbetet ännu inte hade blivit tillräckligt stort för att kunna jobba självständigt och för att få en gemensam inblick i hur systemet fungerade. Denna metod användes också när större problem uppstod. Vid mindre deluppdrag sköttes arbetet mestadels självständigt för att snabbt kunna få in ny funktionalitet i kodbasen. Här användes även versionshanteringsverktyget Git, vilket gjorde att koden kunde synkroniseras när delprojekten var klara.

1.3 Avgränsningar

På Crosskey används Vagrant som utvecklingsmiljö för driftsättning av moduler och applikationer i så kallade *Docker Containers*. I vårt arbete ingick konfiguration av en *Docker Container* som applikationen kommer att startas upp i. Projektet kommer inte att ta upp konfigurering av Vagrant eller Docker.

Ett annat område vi har valt att inte diskutera i vårt arbete är vilka de verkliga batchjobb och uppgifter är som kommer att schemaläggas via vår applikation. Arbetet kommer istället att beskriva eller visa figurer och kod med exempeldata.

2. REST

REST står för Representational State Transfer och är en arkitekturdesign som innebär erbjudande av tjänster med hjälp av webbteknologi. En webbtjänst som följer REST arkitekturen kallas även en RESTful webbtjänst (GeeksForGeeks, 2018).

En RESTful webbtjänst ger möjlighet för system att hämta eller skicka data baserat på en uppsättning enhetliga och förbestämda regler. Kommunikationen mellan webbtjänst och system går via kommunikationsprotokollet Hypertext Transfer Protocol, även kallat HTTP (GeeksForGeeks, 2018).

2.1 Arkitekturens riktlinjer

Likt andra arkitekturer har REST ett visst antal riktlinjer vilka skall uppfyllas för att kunna definiera det som en äkta RESTful webbtjänst. Dessa riktlinjer är sex stycken och lyder enligt följande (Fielding, 2000):

1. **Enhetligt gränssnitt**
   Riktlinjen ställer krav på att det skall finnas ett enhetligt gränssnitt mot servern som gör att alla applikationer som kommunicerar med den givna servern gör det på samma sätt oavsett om applikationen är en webbsida, mobilapplikation etc.

2. **Tillståndslös**
   Kommunikationen mot servern skall vara tillståndslös. Detta innebär att själva förfrågan bör innehålla tillståndet i sig själv med all information som behövs för att servern skall kunna klara av att utföra den givna förfrågan. Denna information kan finnas i adressen, parametrarna eller i form av rubriker i själva förfrågan. Den här tillståndslösa kommunikationen ger servern bättre tillgänglighet eftersom den inte behöver upprätthålla eller uppdatera sessionen.

3. **Tillfällig lagring**

   Svaren från servern skall inkludera om de kan mellanlagras eller inte. Detta innebär att klienten som skickat förfrågan kan spara responsen i sin cache. Detta innebär att framtida likadana förfrågningar kan hämtas direkt från cache istället för att skicka en förfrågan till servern igen. I och med denna riktlinje ökas serverns tillgänglighet och prestanda då den slipper svara på onödiga förfrågningar. En nackdel med denna riktlinje kan dock vara att klienten hämtar gammal data ur sin cache.

4. **Klient och server**

   Denna riktlinje säger att det skall finnas en klientsida och en serversida. Klientsidan bryr sig inte om att lagra data utan skickar endast förfrågningar för de resurser som den behöver. Serversidan äger resurser och ger ut dessa resurser till de som behöver dem utan att bry sig om vad de skall användas till. Mer om klient och server nämns i kapitel 3.1.3.

5. **Lagerarkitektur**

   Denna princip innebär att en applikationsarkitektur kan vara sammansatt av en hierarki med flera skikt där varje skikt i hierarkin inte känner till de andra skiktens existens. Detta innebär att det kan finnas många servrar som ligger mellan klienten och slutservern. Dessa mellanservrar förbättrar tillgängligheten.

6. **Kod på begäran (Valbart)**

   Denna riktlinje är valfri och innebär att klientens funktionalitet utökas genom att den med en förfrågan hämtar körbar kod av servern. Detta kan vara kod i form av till exempel JavaScript.

2.2 Separation mellan server och klient

En fördel med REST-arkitekturen är hur man implementerar klienten och servern helt oberoende av varandra och utan att den ena vet om den andra. I och med denna separation blir det enklare att uppdatera, omstrukturera eller göra ändringar på den ena utan att den

andras funktionalitet eller operation påverkas. Det enda kravet för att de kan hållas separerade är att servern och klienten håller sig till reglerna för kommunikationen och vet vilka format datan som skall skickas eller tas emot skall ha (Codecademy, 2019b).

Ett exempel på denna separation finns i vårt arbete där användargränssnittsmodulen är en klient och jobbhanteringsmodulen är en server. När användaren i backofficeportalen klickar på användargränssnittets “visa jobb” knapp så skickas ett HTTP-anrop till jobbhanteringsmodulen med en förfrågning om att hämta jobb. Jobbhanteringsmodulen svarar med att skicka tillbaka en lista med alla jobb som användargränssnittet kan ta emot och visa. Användargränssnittsmodulen behöver inte veta av eller vara i närheten av någon affärslogik och jobbhanteringsmodulen behöver inte veta något om användargränssnittslogik.

2.3 Kommunikation med förfrågning och respons

I REST arkitekturen kommunicerar klient och server i form av klientens förfrågan och serverns respons.

En klients förfrågan är uppbyggd av (Codecademy, 2019b):

- En adress till en av serverns resurser
- Ett förfrågningshuvud som innehåller information om förfrågan
- Ett HTTP-verb som bestämmer vad det är för typ av förfrågan klienten vill göra
- En icke obligatorisk meddelandekropp som beroende på förfrågningstypen kan innehålla data

De fyra vanligaste HTTP-verben som används i en klients förfrågan är följande (Codecademy, 2019a):

1. **GET** - hämta en specifik resurs eller en samling av resurser. Kan innehålla förfrågningsparameter i form av t.ex. ett id för att välja rätt resurs.

2. **POST** - skapa en ny resurs. Innehåller oftast en meddelandekropp med information om resursen som skall skapas.

3. **PUT** - uppdatera en specifik resurs. Förfrågan skickas oftast med t.ex. ett id som parameter för att välja ut rätt resurs att uppdatera.

4. **DELETE** - ta bort en resurs. Även här används t.ex. ett id för att välja rätt resurs att ta bort.

Dessa fyra HTTP-verben kallas även för CRUD operationer. CRUD är en förkortning för Create, Read, Update, Delete (Codecademy, 2019a).

3. RAMVERK OCH VERKTYG

3.1 Ramverket Spring

Ramverket Spring gör det enkelt att skapa företagsapplikationer i programspråket Java. Utgående från vilken typ av applikation som skall byggas kan man med hjälp av Spring enkelt implementera olika applikationsstrukturer (Pivotal, 2019b).

3.1.1 Spring Beans

I Spring finns objekt som bildar ryggraden för en applikation. Dessa objekt kallas för bönor och sköts om av en komponent som heter *Spring IoC container*. Denna komponent har i uppgift att konfigurera, skapa, bygga och sköta om bönornas livscykel. Komponenten får information om hur den skall göra detta genom att läsa av konfigurationsmetadata. Metadatan tillhandahålls av programmeraren i form av konfigurationsfiler av olika slag, t.ex. Java-konfiguration eller XML-konfiguration (Pivotal, 2019a).

En bönkonfiguration kan bestå av följande (Pivotal, 2019a):

- Ett paketadresserat klassnamn som oftast är adressen till själva klassimplementationen.
- Definition av beteende i form av initialisering och destruktionsmetoder osv.
- Referenser till samarbetsbönor eller andra bönor som den aktuella bönan är beroende av för att fungera.
- Definition av konstruktorargument och egenskaper för bönan som skapas.

3.1.2 Beroendeinjektion

Beroendeinjektion är en princip som används i Spring-projekt och innebär att objekten själva skall tydliggöra vilka beroenden de behöver för att kunna fungera. Detta betyder att objekten via funktionsargument, konstruktorargument eller via konfigurerade egenskaper får in de andra objekten de behöver arbeta med (Pivotal, 2019a).

Med beroendeinjektion fås mer strukturerad kod och bättre omstruktureringsmöjligheter, speciellt om beroendet är av typen abstrakt klass eller gränssnitt eftersom objekten då inte känner till exakt placering i projektstrukturen eller klass av sina beroenden (Pivotal, 2019a).

Autowiring är en funktion som låter Spring automatisk injicera beroenden i klasser. Genom att definiera en autowriringannotation tillsammans med ett objekt så söker Spring upp rätt beroende i applikationskontexten och injicerar dessa (JavaTPoint, 2019). I figur 1 ser vi ett exempel på klass som initialiserar en medlemsvariabel med hjälp av autowiring.

```
1   package com.example.demo;
2
3   import org.springframework.beans.factory.annotation.Autowired;
4
5   public class Cat {
6
7       @Autowired
8       private Color color;
9
10      public Color getColor() {
11          return color;
12      }
13  }
```

Figur 1. Kodexempel på Autowiring med annotation.

3.2 Spring Boot

Spring Boot är ett ramverk som utökar Spring. Ramverket skapades för att eliminera *boilerplatekod* som tillkommer vid konfigurering av ett vanligt Spring projekt (Baeldung, 2018). Boilerplatekod kan definieras som en sektion av kod som bör vara inkluderad på många ställen med små eller inga modifikationer (Zaveri, 2018).

Fördelar med Spring Boot:

- Erbjuder startpaket med bibliotek som underlättar byggnad och konfiguration av applikationen (Baeldung, 2018).

- Innehåller inbyggd webbserver för att underlätta distribution av applikationen (Baeldung, 2018).
- Förminskar antalet kodrader och utvecklingstid (Tutorialspoint, 2019).
- Ger förenklad beroendehantering av tredjepartsbibliotek (Tutorialspoint, 2019).
- Tillhandahåller autokonfigurering (Tutorialspoint, 2019).

I projektet används Spring Boot för att enkelt kunna implementera applikationen med hjälp av startpaket och autokonfigurering. En självständig applikation innebär även större skalbarhet, vilket gör att man kan skapa en ny instans till systemet vid fall av överbelastning etc.

3.3 Quartz

Quartz är ett mångsidigt bibliotek med öppen källkod för schemaläggning av batchjobb vilket kan integreras med en Java-applikation. Quartz kan användas till att skapa mindre och enklare eller större mer komplexa jobbscheman för körning av några hundra till flera tusentals jobb (Software AG, 2018).

Quartz är ett bibliotek som kan användas ifall en applikation eller ett system har uppgifter som skall utföras automatiskt vid en given tidpunkt (Software AG, 2018). Ett exempel på en sådan situation kan vara en tandläkarklinik som skickar ut SMS påminnelser till kunder med bokade tider. När användaren lägger in en ny bokning i systemet så schemaläggs ett jobb som skickar kunden ett SMS med en påminnelse 24 timmar före den bokade tiden.

Fördelar med Quartz (Software AG, 2018)**:**

- Möjlighet att exekvera kod efter ett givet schema, t.ex. en tidpunkt varje dag eller endast bankdagar osv. Jobben som schemaläggs kan vara vilken som helst typ av Javaklasser som implementerar ett jobbgränssnitt som Quartz erbjuder.
- Erbjuder en jobbavslutningsprocedur som tillhandahåller schemaläggaren med information om huruvida jobbet lyckats eller inte. Denna procedur kan även instruera schemaläggaren med uppgifter baserat på resultatet, t.ex. att köra jobbet på nytt.

- Innehåller ett gränssnitt vilket kan användas för att implementera olika tekniker för att förvara och spara jobb.
- Applikationer kan implementera något av de erbjudna gränssnitten för monitorering och kontroll av jobbexekveringen.

I projektet används Quartz eftersom detta bibliotek användes i den tidigare arkitekturen och därmed fanns stora delar av funktionaliteten färdigt implementerad. Quartz passar bra till banksystem eftersom man enkelt kan schemalägga t.ex. jobb för utskick av kreditfakturor eller rapporter av olika slag.

3.4 Feign

Feign är ett hjälpbibliotek för HTTP-klienter i programspråket Java. Feign skapades med inspiration från andra klientbibliotek som Retrofit, WebSocket m.m. Målet med detta hjälpbibliotek är att minska ner på komplexiteten för kommunikation mellan HTTP-programmeringsgränssnitt och klient (Baeldung, 2016).

Med hjälp av Feign kan man enkelt implementera en HTTP-klient med minimalt antal rader kod. Utvecklaren behöver endast implementera ett gränssnitt med funktionsdeklarationer och annotationer som erbjuds av Feign. I dessa funktionsdeklarationer anges returtyp och eventuella parametrar som skall skickas i HTTP-frågan likt figur 2. Annotationerna anger HTTP-förfrågningstyp i form av t.ex. någon av CRUD-operationerna som nämnts i kapitel 2.3. Med annotationerna anges även vilken destination HTTP-förfrågningarna skall gå till och själva implementationen av funktionerna sköter Feign automatiskt (Baeldung, 2016).

```java
package com.example.demo;

import feign.RequestLine;

public interface MyClient {

    @RequestLine("POST /application/cars")
    void saveCar(CarDTO car);

    @RequestLine("GET /application/cars/{carId}")
    CarDTO getCar(int carId);
}
```

Figur 2. Kodexempel på ett Feign gränssnitt.

Förklaringen av kodexemplet i figur 2 lyder enligt följande:

1. *public interface MyClient*

   Detta innebär att man har skapat ett gränssnitt som heter *MyClient*.

2. Första metoden:
   a. *@ReguestLine* är en annotation som Feign tillhandahåller. Denna annotation berättar vilken destination som skall användas.
   b. *POST* är HTTP-förfrågningstypen och anger att det skickas med data som skall sparas.
   c. */application/cars* är destinationen som skall användas.
   d. *void saveCar(CarDTO car)* är en metoddeklaration, som består av följande:
      i. *void* är returtypen och anger att metoden inte returnerar något.
      ii. *saveCar* är namnet på metoden och *CarDTO car* säger att man har ett objekt *car* av klassen *CarDTO* som skall sparas.

3. Andra metoden:
   a. *GET* är HTTP-förfrågningstypen vilken anger att man vill hämta data.
   b. */application/cars/{carId}* är destinationen och i detta fall skickas det med en variabel *carId* som säger vilket *car* objekt man vill hämta.
   c. *CarDTO getCar(int carId)* är en metodsdeklaration som består av följande:

i. *CarDTO* är returtypen och anger att denna metod returnerar ett objekt av klassen *CarDTO*.
ii. *getCar* är namnet på metoden. Denna metod har ett funktionsargument i form av ett id med datatypen Integer som anges för att hämta ett specifikt *CarDTO* objekt.

3.5 Gradle

Gradle är ett automatiseringsverktyg för byggnationer av applikationer. När större applikationer skapas blir det snabbt nödvändigt att använda sig av ett byggnationsverktyg för att underlätta olika steg då en applikation byggs upp. Gradle kan skapa olika processer genom att slå ihop flera delsteg (Gradle Inc, 2019d).

Som ett exempel kan en process se ut likt nedan:

- Steg 1: “Hämta tredjepartsbibliotek”.
- Steg 2: “Kompilera kod”.
- Steg 3. ”Kör tester”.

Gradle definierar hur de olika stegen är relaterade till varandra, t.ex. steg 1 måste utföras före steg 2 kan starta, som kan ses på rad 12 i figur 3. Man definierar sina byggprocesser och applikationens beroende till tredjepartsbibliotek i en build.gradle fil (Gradle Inc, 2019d).

```
apply plugin: "java"

repositories{
    mavenCentral()
}

dependencies{
    compile 'com.google.guava:guava:11.0.2'
}

task taskX {
    dependsOn 'taskY'
    doLast {
        println 'taskX'
    }
}

task taskY {
    doLast {
        println 'taskY'
    }
}
```

Figur 3. Ett kodexempel på en build.gradle fil.

I figur 3 visas en typisk uppbyggnad av en build.gradle fil för en java-applikation och nedan förklaras koden.

1. *apply plugin: "java"*
   Här används en färdig plugin för java-applikationer. Genom denna plugin fås tillgång till färdigt utvecklade processer som kan användas av Javaprojekt (Gradle Inc, 2019c).

2. *repositories*
   Funktionen definierar varifrån tredjepartsbibliotek skall hämtas som man vill använda sig av i applikationen. I detta fall hämtas biblioteken från Maven Central som är en förvaringstjänst som erbjuder tredjepartsbibliotek (Gradle Inc, 2019a).

3. *dependencies*

   Anger vilka tredjepartsbibliotek som skall användas i applikation. I exemplet anges att man bör använda sig av guava version 11.0.2. *Compile* anger att applikationen behöver tillgång till biblioteket under kompileringen processen av applikationen (Gradle Inc, 2019a).

4. *taskX* och *taskY*

   För att demonstrera hur man skapar egna steg i en byggprocess har två steg skapats: *taskX* och *taskY*. I exemplet *taskX* ses raden *dependsOn 'taskY'*, vilket anger att *taskX* måste vänta på att *taskY* har utförts klart innan den själv får börja köras (Gradle Inc, 2019d).

4. UTVECKLINGEN AV APPLIKATIONEN

4.1 Planeringen av projektet

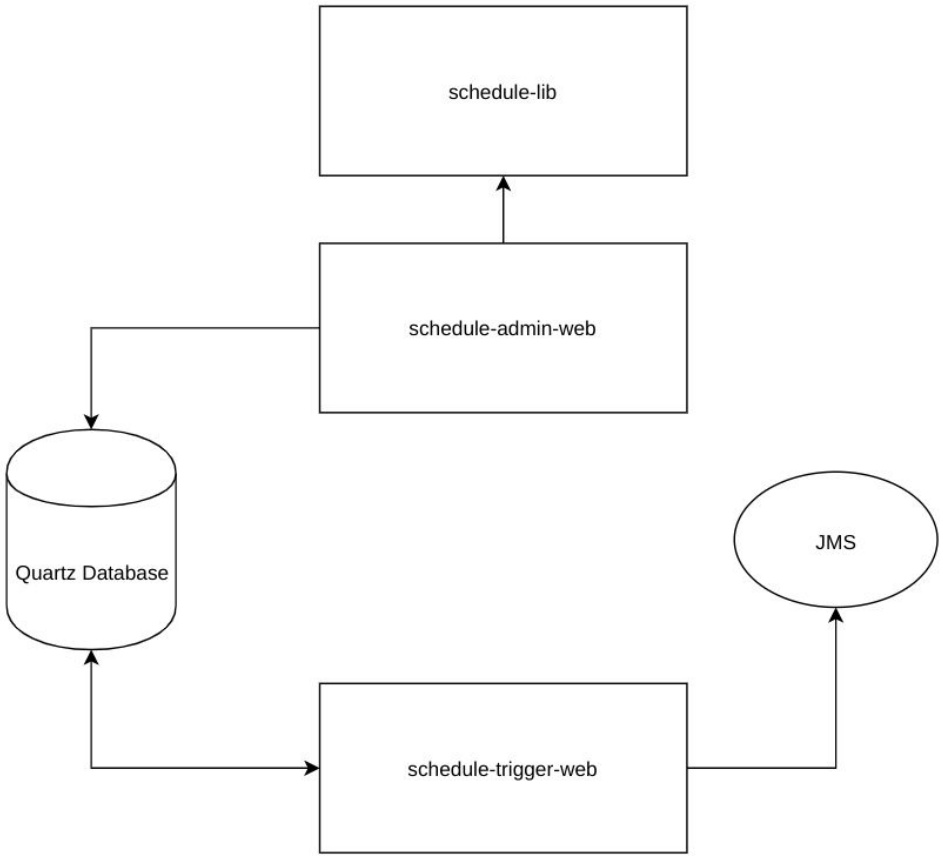
Projektet började med att kontakta systemarkitekterna på Crosskey och berätta att det fanns ett intresse av att göra ett examensarbete åt företaget. När systemarkitekterna sedan hade diskuterat med varandra kallades vi in till ett möte för att diskutera eventuella projekt tillsammans med dem och bestämma vilket som skulle passa oss bäst både svårighetsmässigt och tidsmässigt. Vid mötet framkom förslaget om att uppdatera en befintlig del av systemet genom att bryta ut funktionalitet till en egen REST-tjänst. REST-arkitektur var bekant för oss sen tidigare, vilket var en av orsakerna att arbetet valdes.

Arbetet inleddes med att se över vad applikationen hade för syfte och samtidigt utforska den tidigare arkitekturen. Det visade sig att användargränssnittet och databaslagret var tätt sammankopplade. Denna arkitektur visas i figur 4. Under samma process undersöktes det även vilka moduler som skulle beröras av vårt arbete. Det konstaterades att det huvudsakligen rörde sig om tre moduler:

- **schedule-lib** bestod av logik för att schemalägga och för att manuellt kunna köra batchjobb men för att möjliggöra användning av logiken behövde konsumerande bibliotek ange databaskällan som skulle användas.

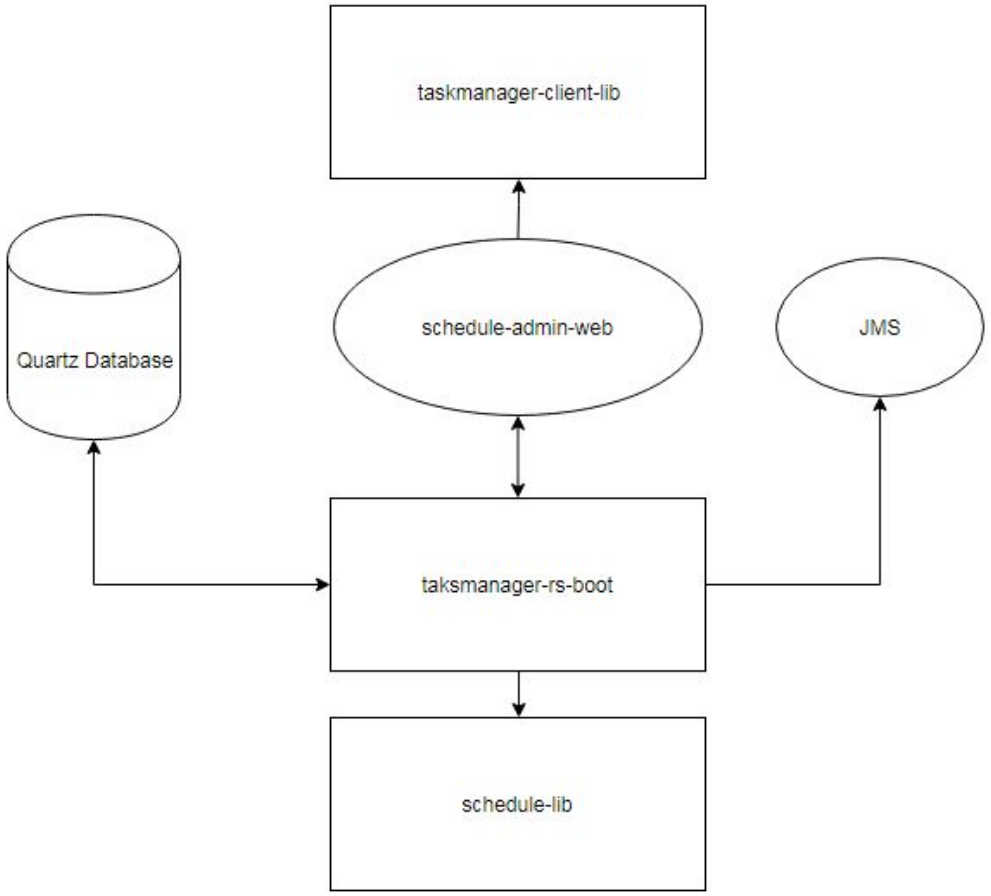
- **schedule-admin-web** är en huvudmodul som sköter om användargränssnittet. Detta bibliotek använde sig av schedule-lib för att schemalägga batchjobb.

- **schedule-trigger-web** har uppgiften att kontinuerligt kontrollera när batchjobben skall köras och skickade sedan informationen om exekveringen vidare med hjälp av JMS. JMS står för Java Messaging System och används för att skicka meddelande mellan applikationer.



Figur 4. Modulerna i den tidigare arkitekturen.

När arkitekterna förklarade hur den nya arkitekturen skulle se ut visade det sig att huvudsyftet med projektet var att bryta isär den här sammankopplingen med hjälp av REST-arkitekturen som visas i figur 5. Samtidigt konstaterades det att schedule-trigger-webs funktionalitet helt och hållet kunde flyttas in i den nya applikationen.



Figur 5. Den nya arkitekturen där trigger-web har flyttats in i taskmanager-rs-boot.

Nästa steg var att gå igenom de olika ramverk och verktyg som skulle användas enligt Crosskey standarder. Vissa verktyg och ramverk var bekanta från skolan och vissa var helt nya vilket innebar att vi behövde lägga ned tid för att få en förståelse över hur de fungerar.

Nästa del i planeringsfasen var att tillsammans börja gå igenom koden för att få samma uppfattning och för att hjälpa varandra ifall att det var någonting den ena parten inte förstod. Under granskningen av systemet konstaterades det att denna del av systemet inte hade blivit uppdaterad under en längre tid. Vi undersökte även vilka kopplingar modulerna hade till resten av systemet, då det framgick att de även används på andra ställen. Ett annat problem som konstaterades var att det saknades implementation för att kunna köra applikationen i utvecklingsmiljön.

4.2 Spring Boot-applikationen

4.2.1 Implementation av applikationen

När koden hade granskats påbörjades arbetet med implementationen genom att skapa ett Spring Boot-projekt med hjälp av Spring Initializr. Detta är en webbtjänst där man enkelt och snabbt kan skapa ett Spring Boot-projekt. Tjänsten ger tillgång till olika val såsom vilket byggnadsverktyg som skall användas, vilken version av Spring Boot man vill använda samt om man vill inkludera några tredjepartsbibliotek direkt när Spring Boot-applikationen skapas. När man är färdig kan man enkelt ladda ner projektet.

Valet blev att använda startpaketen spring-starter-web och spring-starter-undertow eftersom Crosskey har egna riktlinjer vilka bör följas när man utvecklar och konfigurerar nya Spring Boot-applikationer. Valet att använda Undertow hörde till en av dessa riktlinjer eftersom Undertow har färre sårbarheter än t.ex. webbservern Apache Tomcat som ingår som standard i Spring Boot. Valet av byggnationsverktyg för applikationen blev Gradle eftersom Crosskey använder sig av detta verktyg. Spring Boot-applikationen kommer med en egen Gradle wrapper. En Gradle wrapper ger utvecklare möjlighet att snabbt kunna börja jobba på sina projekt eftersom de inte behöver installera Gradle på sina lokala maskiner (Gradle Inc, 2019b). Eftersom Crosskey redan har en Gradle wrapper för hela systemet kunde den medföljande wrappern raderas. Fördelen med att ha en wrapper för hela systemet är att det möjliggör att kunna köra samtliga Gradle tasks för alla undermoduler samtidigt istället för att köra dessa var för sig, men valet att köra en specifik modul finns fortfarande kvar.

Efter att ha tagit bort Gradle wrappern behövde vårt modulnamn läggas till i systemets settings.gradle fil. En settings.gradle fil har som huvudsyfte att definiera alla inkluderande submoduler som tillhör samma gradle projekt. Detta gjordes med syftet att Gradle wrappen för hela systemet skulle veta att det fanns en ny modul registrerad så att de Gradle tasks som fanns definierade i systemets huvudsakliga build.gradle fil kunde användas.

4.2.2 Implementation av databasen

Följande steg var att skapa ett databasschema för vår modul eftersom det i planeringsfasen konstaterades att det saknades stöd för att köra den dåvarande lösningen lokalt på våra egna maskiner. Vi insåg att detta skulle innebära problem som vi behövde lösa tidigt i utvecklingsfasen eftersom databasen var en viktig del av vårt projekt. Vi granskade Crosskeys testmiljö och konstaterade att den dåvarande lösningen jobbade mot ett eget databasschema istället för det primära databasschemat. Utifrån denna information beslöts att för utvecklingsmiljön också skapa databasstrukturen i ett separat schema för applikationen istället för att försöka införa strukturen i det primära databasschemat.

Utvecklarna på Crosskey har egna databasscheman för att inte utvecklarnas egna databasändringar skall störa de andra utvecklarnas arbete. För att migrera det egna databasschemat används Flyway. Flyway är ett verktyg som erbjuder versionshanteringskontroll för databasmigrationer. Crosskey använder Flyway i syftet att så få ändringar som möjligt görs manuellt i databasen. För att utvecklarna skall få de nya ändringarna i sina databasscheman skriver man SQL-skripts och kör Flyway som migrerar utvecklarnas databasscheman till den senaste versionen. På detta sätt säkerställs det att alla utvecklare jobbar mot samma version av databasstrukturen.

Efter konsultation med arkitekter och andra utvecklare framgick det att det inte fanns något hjälpverktyg för att skapa strukturen för vårt databasschema, så detta behövde utföras manuellt. Det fanns några färdiga SQL-skript med databasfrågor som använts tidigare till förfogande. Detta möjliggjorde att vi enkelt kunde kopiera dessa databasfrågor och manuellt importera dem till databasen via en terminal.

4.2.3 Återanvändning av befintlig kod

Efter att databasen hade skapats påbörjades arbetet med att implementera Spring Boot-applikationen genom att analysera vilka nuvarande delar som kunde fortsätta att användas. Detta gjordes för att inte uppfinna något som redan fanns. Under processen konstaterades att det tidigare nämnda biblioteket schedule-lib skulle komma att användas i den här Spring Boot-applikationen eftersom biblioteket erbjöd en färdig Spring-böna vars

funktionalitet innehöll alla operationer för schemaläggning av batchjobb. Dessa operationer var följande:

- Lägga till olika jobb till databasen.
- Radera jobb från databasen.
- Redigera befintliga jobb.
- Funktionalitet för att kunna köra jobb direkt.
- Funktionalitet för att köra jobb en specifik tidpunkt.

Biblioteket hade även två bönor som tillsammans konfigurerade bönan med schemaläggningsfunktionalitet vilket de konsumerande modulerna kunde använda sig av. Konfigureringen av bönorna gjordes via XML vilket är ett föråldrat tillvägagångssätt som enligt Crosskeys anvisningar inte är att rekommendera eftersom det gör koden svårläst och otydlig. Detta visas i figur 6.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="color" class="com.example.demo.Color"/>

    <bean id="car" class="com.example.demo.Car">
        <constructor-arg ref="color"/>
    </bean>

</beans>
```

```java
package com.example.demo.spring.config;

import ...

@Configuration
public class ExampleConfiguration {

    @Autowired
    Color color;

    @Bean
    public Car car(){
        return new Car(color);
    }
}
```

Figur 6. Jämförelse mellan en XML konfiguration (ovan) och Java konfiguration (nedan).

Bönorna som skapades i bibliotekets konfiguration:

- ***scheduler***

  Detta är ett Quartscheduler-objekt som skapas baserat på olika egenskapsparametrar.
- ***taskDao***

  Detta är ett datatillgångsobjekt vilket används för kommunikation mot databasen.
- ***taskManager***

  Detta object skapades med hjälp av de två tidigare nämnda *scheduler* och *taskDao* bönorna som konstruktorparametrar och är den böna som erbjöd schemaläggningsfunktionalitet.

Denna design krävde att de konsumerande modulerna hade konfigurationen för vilken databas som skulle användas. Detta var ej optimalt eftersom det fanns flera konsumentmoduler och varje modul behövde därmed konfigurera databaskällan var för sig.

Eftersom det hade beslutats att behålla schedule-lib importerades detta bibliotek med hjälp av Gradle. XML konfigurationen av bönorna byttes ut mot Java konfigurering. Under samma process valdes det att flytta över konfigureringen av databas källan från biblioteket till den nya applikationen eftersom det endast är den som skall veta vilken databaskälla som används. De konkreta klasserna som bönorna byggdes upp av lämnades kvar i schedule-lib.

4.2.4 Konfigureringen av applikationen

Crosskey har egenutvecklade hjälpbibliotek som är specifikt designade för att underlätta konfigureringen av sina Spring Boot-applikationer. Fördelen med denna design är att man enkelt kan ändra hur Spring Boot-applikationerna skall konfigureras på ett och samma ställe, istället för att manuellt konfigurera varje Spring Boot-applikation var för sig.

För att använda dessa bibliotek hade det skapats ett bibliotek vars enda uppgift var att importera de andra hjälpbiblioteken. Detta gjordes genom att i applikationens build.gradle fil definiera biblioteket med uppgiften att importera dom andra biblioteken för att transitivt få in alla de andra hjälpbiblioteken vilket innebar att vi kunde använda oss av dem. Detta innebar

även att vi kunde ta bort biblioteken starter-undertow och starter-web eftersom de inkluderades transitivt genom biblioteket vars enda uppgift var att importera hjälpbiblioteken.

4.2.5 Implementationen av REST API

Efter att ha gjort alla konfigurationer, skapat en databas med Quartz-tabeller och även lyckats få Spring Boot-applikationen att köra en webbserver kunde arbetet med att implementera ett programmeringsgränssnitt (API) påbörjas. API står för Application Programming Interface och används för att hämta, lämna eller uppdatera information mellan olika applikationer eller system (Sturgeon, 2018).

Arbetet inleddes med att identifiera de operationer som behövde finnas tillgängliga för klienterna. Detta gjordes genom att granska den föregående arkitekturens kod vilket resulterade i operationerna i tabell 1.

Tabell 1. Programmeringsgränssnittets operationer med tillhörande förfrågningstyper.

Operation	***Förfrågningstyp***
Lägg till nytt jobb	POST
Hämta alla jobb	GET
Hämta alla jobb baserat på grupp	GET
Uppdatera jobb	PUT
Ta bort jobb	DELETE
Pausa jobb	PUT
Återuppta pausat jobb	PUT
Köra jobb direkt	PUT
Hämta grupper	GET
Hämta slutförda jobb inom ett tidsintervall	GET

Programmeringsgränssnittet i denna applikation skulle vara ett REST API enligt beskrivningen i kapitel 3. När man bygger ett REST API med ramverket Spring implementeras en kontrollerklass som annoteras med *@RestController*. I denna kontrollerklass definieras alla destinationer man vill ha tillgängliga samt implementerar deras funktionalitet (Pankaj, 2019).

Följande steg var att bestämma destinationsadresserna för dessa operationer. Detta gjordes med inspiration från tidigare API:er i Crosskeys system samt med förslag från andra utvecklare. Destinationsadresserna lades sedan in i operationernas annotationer som berättar vilken HTTP-förfrågningstyp samt eventuella inparametrar operationen har. Annotationen kan även definiera vad operationen producerar och konsumerar likt figur 7.

```
1   package com.example.demo;
2
3   import ...
6
7   @RestController
8   @RequestMapping("/api/example/")
9   public class ExampleController {
10
11      @Autowired
12      private CatManager catManager;
13
14      @PostMapping(value = "cat", consumes = MediaType.APPLICATION_JSON_VALUE)
15      public void add(@RequestParam Cat cat){
16          catManager.addCat(cat);
17      }
18
19      @GetMapping(value = "cat/{catId}", produces = MediaType.APPLICATION_JSON_VALUE)
20      public Cat get(int catId){
21          return catManager.getCat(catId);
22      }
23
24      @PutMapping(value = "cat", consumes = MediaType.APPLICATION_JSON_VALUE)
25      public void update(@RequestParam Cat cat){
26          catManager.update(cat);
27      }
28
29      @DeleteMapping(value = "cat/{catId}")
30      public void delete(int catId){
31          catManager.delete(catId);
32      }
33  }
```

Figur 7. Exempelkod för en REST-kontroller med tillhörande operationer och dess annotationer.

Slutligen återstod arbetet med att implementera de operationer som definierats. Som tidigare nämnts fanns färdig funktionalitet för schemaläggning av batchjobb tillgänglig för återanvändning vilket innebar att det enkelt gick att styra vidare anropen från API:ets operationer. Detta gjordes genom att med hjälp av Autowire-annotation injicera ett objekt som tillhandahåller denna funktionalitet till kontrollern och sedan använda detta objekts metoder för att vidarebefordra anropen likt figur 7.

Slutligen testades API:ets operationer med Postman. Detta är ett verktyg som används vid testning av REST API:er för att utföra HTTP-anrop med valfria förfrågningstyper mot en angiven HTTP-adress (Postman Inc, 2019).

4.2.6 Uppdatering av Spring och Spring Boot

Under projektets gång påbörjades uppdateringen till Spring 5 och Spring Boot 2 för resten av systemet, vilket gjorde att även vårt projekt behövde uppdateras. Med uppdateringarna uppstod möjligheten att kunna använda sig av biblioteket spring-starter-quartz som inte hade något stöd för Spring Boot versioner under 2.0. Genom att lägga till biblioteket spring-starter-quartz i build.gradle kunde man reducera konfigurationen av bönorna eftersom spring-starter-quartz automatiskt skapar en schedule-böna när applikationen startas. Bönan använder sig av applikationens konfigurationsfiler (application.properties/application.yaml) för att veta hur den skall konfigureras.

4.3 Utvecklingen av klientbiblioteket

Syftet med ett klientbibliotek är att de konsumerande modulerna inte behöver ha någon konfiguration för att kommunicera med en REST-tjänst. Det är själva klientbiblioteket som är ansvarigt för att veta vilken adress man skall sända till, vilka protokoll som skall användas samt sköta om t.ex auktoriseringar. Detta betyder att alla moduler eller applikationer som använder sig av detta bibliotek blir en klient.

4.3.1 Fördelar med ett klientbibliotek

Genom att använda sig av ett klientbibliotek undviks även potentiella framtida problem. Exempelvis kan vi tänka oss ett scenario där tio olika klienter kommunicerar med samma REST-tjänst utan ett klientbibliotek och klienterna måste konfigurera serveradressen samt vilken port som skall användas var för sig. Efter ett år ändras portnumret som skall användas för att kommunicera med REST-tjänsten, vilket leder till att det finns tio olika ställen där portnumret behöver ändras. Genom att använda ett klientbibliotek centraliseras all konfiguration till ett och samma ställe, vilket underlättar för nästa utvecklare som skall ändra någonting med konfigureringen.

4.3.2 Implementation av klientbiblioteketet

Denna process inleddes med att skapa ett nytt bibliotek i systemet. Eftersom det i planeringsfasen framgick att hjälpbiblioteket Feign skulle användas vid skapandet av klientbiblioteket angavs detta i build.gradle för att importera biblioteket. Eftersom alla miljöer har en separat konfiguration skapades även en egen YAML-fil som skulle innehålla konfigurationsdata för de olika miljöerna. T.ex för utvecklingsmiljön så är adressen till "http://localhost:8080/applikation" medan adressen i testmiljön kan vara "http://testMiljö:1234/applikation". Se figur 8.

```yaml
spring:
  profiles: localdev

environment: development

server:
  address: localhost/applikation
  port: 8080

---

spring:
  profiles: tst

environment: test

server:
  address: testMiljö/applikation
  port: 1234

```

Figur 8. Exempel på en Yamlkonfiguration för två olika miljöer.

Nästa steg var att skapa en konfigurationsklass som skulle innehålla konfigureringen av bönor för klientbiblioteket.

En böna hade hand om att skapa ett konfigurationsdataobjekt. Den data som används för att skapa detta objekt hämtas från tidigare nämnda YAML-fil. För att veta vilken miljö man befinner sig i hämtas en miljövariabel. Miljövariabler är variabler som beskriver miljön som program körs i.

En annan böna skapar själva klienttjänsten med hjälp av Feign. I denna böna behövs det också definieras vilket gränssnitt som de konsumerande biblioteken skall använda sig av samt vilket dataformat objekten skall serialiseras och deseriliariseras till för att kommunicera med andra applikationer. Serialisering innebär att ett objekt konverteras till ett angivet format innan det skickas mellan olika applikationer. Deserialisering är motsatsen till serialisering, dvs att återskapa ursprungsobjektet från den serialiserade datan (GeeksForGeeks, 2016). I detta arbete används JSON-formatet vid serialisering och deserialisering.

Med denna design behöver de konsumerande modulerna endast importera klientbiblioteket i sina build.gradle-filer och injicera bönan som skapar själva klienttjänsten genom autowiring för att kunna använda sig av funktionaliteten som REST-tjänsten erbjuder.

4.3.3 Polymorfisk deserialisering

Ett problem som uppstod när testningen av klientbibliotek påbörjades var att deserialiseringen av jobbobjekten misslyckades. Detta berodde på att en medlemsvariabel i jobbobjektet inte kunde mappas till någon speciell klass pga. att medlemsvariabeln hade en datatyp i form av ett gränssnitt. Detta gränssnitt med namnet Frequency har i syfte att ange på vilket sätt jobbet skall schemaläggas.

Gränssnittet Frequency fungerar som ett markeringsgränssnitt och innehåller ingenting. Frequency implementeras i sin tur av en abstrakt klass som heter AbstractFrequency. Denna abstrakta klass har fyra stycken subklasser med olika funktionalitet för att schemalägga jobb som t.ex. schemaläggning av engångsjobb med ett speciellt datum och tid för exekvering,

återkommande exekveringar baserat på ett intervall eller exekvering baserat på ett uttryck i form av t.ex. månadens sista dag varje månad kl. 09:00.

Problemet berodde på att det inte hade angetts hur en instans av Frequency skulle mappas och detta behöver anges manuellt i koden då en polymorfisk deserialisering skall ske. Detta gjordes manuellt genom att förse gränssnittet Frequency med annotationer som beskriver vilka konkreta klasser som implementerar denna typ likt figur 9.

```
1   package com.example.demo;
2
3   import ...
6
7   @JsonInclude(JsonInclude.Include.NON_EMPTY)
8   @JsonTypeInfo(use = JsonTypeInfo.Id.CLASS, property = "@class")
9   @JsonSubTypes({
10          @JsonSubTypes.Type(value = SpecifiedTime.class, name = "SpecifiedTime"),
11          @JsonSubTypes.Type(value = Interval.class, name = "Interval"),
12          @JsonSubTypes.Type(value = Every.class, name = "Every")})
13  public interface Frequency {
14      /*This is just an empty marking interface*/
15  }
```

Figur 9. Gränssnittet Frequency med annotationer för polymorfisk deseralisering.

När objektet sedan deserialiseras jämförs datan i variabeln med dessa annotationer för att mappa ut rätt klass. I figur 10 ses en *@JsonTypeName* annotation vilket anger konkreta klassers *TypeName*.

```
1   package com.example.demo;
2
3   import com.fasterxml.jackson.annotation.JsonTypeName;
4
5   @JsonTypeName("Interval")
6   public class Interval extends AbstractFrequency {
7
8       /*.....*/
9   }
```

Figur 10. En konkret klass med @JsonTypeName annotation.

5. SLUTSATSER

Resultatet av projektet blev att vi som planerat har en självständig Spring Boot-applikation som erbjuder en REST-tjänst med operationer för schemaläggning av batchjobb samt ett funktionellt klientbibliotek som konsumerande bibliotek kan använda sig av för att kommunicera med REST-tjänsten. Logiken för Spring Boot-applikationen är klar men det finns fortfarande en del konfiguration kvar att göra innan applikationen är redo för test- och produktionsmiljö. Den främsta orsaken till detta är att det fanns väldigt många nya olika ramverk och verktyg att lära sig, vilket var relativt tidskrävande. En annan orsak var även att Spring och Spring Boot versionerna uppdaterades i resten av systemet under processens gång vilket medföljde en del komplikationer för utvecklingen och testningen av applikationen. Den återstående konfigurationen kommer att åtgärdas i efterhand.

Projektet visade sig vara mer komplext än vad vi trodde vid arbetets inledning. Detta ledde till att vi valde att använda parprogrammering under största delen under projektet. Denna arbetsmetod gav oss möjlighet att jobba mer effektivt då vi kunde diskutera idéer och förslag direkt med varandra under implementeringsfasen.

Moderniseringen av schemaläggaren genom REST-arkitektur blev som förväntat mer särkopplad än den föregående arkitekturen. Största fördelen med denna särkoppling är att det i framtiden enkelt går att byta ut eller göra ändringar i schemaläggarens logik utan att konsumerande bibliotek påverkas av detta. En annan fördel är att en ny applikation eller modul enkelt kan importera klientbiblioteket och därmed kan få funktionaliteten som REST-tjänsten erbjuder utan att ha egna konfigurationer som tidigare arkitektur krävt.

Arbetet har varit väldigt lärorikt för oss båda eftersom vi har fått arbetat med många olika verktyg och ramverk som var helt okända för oss. Projektet har gett oss en erfarenhet som vi kommer ha nytta av i arbetslivet framöver eftersom dagens riktlinjer om hur större applikationer skall byggas baserar sig på den arkitektur som vi har implementerat i det här arbetet.

KÄLLOR

Baeldung. (2016, September 25). Intro to Feign. Retrieved October 9, 2019, from Baeldung website: https://www.baeldung.com/intro-to-feign

Baeldung. (2018, September 29). A Comparison Between Spring and Spring Boot. Retrieved October 8, 2019, from Baeldung website: https://www.baeldung.com/spring-vs-spring-boot

Codecademy. (2019a). What is CRUD? Retrieved October 23, 2019, from Codecademy website: https://www.codecademy.com/articles/what-is-crud

Codecademy. (2019b). What is REST? Retrieved October 19, 2019, from Codecademy website: https://www.codecademy.com/articles/what-is-rest

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* (University of California, Irvine Doctoral dissertation). Retrieved from https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

GeeksForGeeks. (2016, January 13). Serialization and Deserialization in Java. Retrieved November 21, 2019, from GeeksforGeeks website: https://www.geeksforgeeks.org/serialization-in-java/

GeeksForGeeks. (2018, December 9). REST API Architectural Constraints. Retrieved October 19, 2019, from GeeksforGeeks website: https://www.geeksforgeeks.org/rest-api-architectural-constraints/

Gradle Inc. (2019a). Declaring Dependencies. Retrieved November 7, 2019, from https://docs.gradle.org/6.0.1/userguide/declaring_dependencies.html

Gradle Inc. (2019b). The Gradle Wrapper. Retrieved November 7, 2019, from https://docs.gradle.org/6.0.1/userguide/gradle_wrapper.html

Gradle Inc. (2019c). The Java Plugin. Retrieved November 7, 2019, from https://docs.gradle.org/6.0.1/userguide/java_plugin.html

Gradle Inc. (2019d). What is Gradle? Retrieved November 7, 2019, from docs.gradle.org website: https://docs.gradle.org/6.0.1/userguide/what_is_gradle.html

JavaTPoint. (2019). Autowiring in Spring. Retrieved November 8, 2019, from www.javatpoint.com website: https://www.javatpoint.com/autowiring-in-spring

Pankaj. (2019). Spring RestController. Retrieved November 10, 2019, from JournalDev website: https://www.journaldev.com/21536/spring-restcontroller

Pivotal. (2019a). Core Technologies. Retrieved October 26, 2019, from https://docs.spring.io/spring/docs/5.1.9.RELEASE/spring-framework-reference/core.html?fbclid=IwAR0wVGTZ9OqdPzVPuK5ufov4iWUGj4-0AIlR7222nomatmDOT0TA48Pdyp4

Pivotal. (2019b). Spring Framework Overview. Retrieved October 26, 2019, from https://docs.spring.io/spring/docs/5.1.9.RELEASE/spring-framework-reference/overview.html

Postman Inc. (2019). The Collaboration Platform for API Development. Retrieved November 10, 2019, from Postman website: https://www.getpostman.com/

Software AG. (2018). Quartz Enterprise Job Scheduler. Retrieved October 8, 2019, from http://www.quartz-scheduler.org/documentation/2.3.1-SNAPSHOT/introduction.html

Sturgeon, P. (2018, March 9). Talk Human to Me: What is an API? Retrieved October 30, 2019, from Codecademy News website: https://news.codecademy.com/what-to-know-about-apis/

Svenska Akademiens Ordböcker. (2009). Retrieved July 10, 2019, from svenska.se website: https://svenska.se/so/?sok=batch&pz=4

Tutorialspoint. (2019). Spring Boot - Introduction. Retrieved October 8, 2019, from https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm

Zaveri, M. (2018, February 1). What is boilerplate and why do we use it? Necessity of coding style guide. Retrieved August 10, 2019, from freecodecamp website: https://www.freecodecamp.org/news/whats-boilerplate-and-why-do-we-use-it-let-s-check-out-the-coding-style-guide-ac2b6c814ee7/