Examensarbete, Högskolan på Åland, Utbildningsprogrammet för informationsteknik

UTVECKLING AV DOKUMENTÖVERFÖRINGSMODUL I JAVA/SPRING

William Eriksson

42:2016

Datum för publicering: 15.12.2016
Handledare: Agneta Eriksson-Granskog

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	William Eriksson
Arbetets namn:	Utveckling av dokumentöverföringsmodul i Java/Spring
Handledare:	Agneta Eriksson-Granskog
Uppdragsgivare:	Crosskey Banking Solutions

Abstrakt

Syftet med detta examensarbete är att lägga till funktionalitet för nedladdning av dokument och samtidigt dela upp kodbasen. Arbetet görs för att uppdragsgivaren dels ville ha funktionalitet för att ladda ner dokument, och dels för att man märkt att nuvarande implementation var undermålig för nya krav som kommit fram.

I arbetet presenteras hur den nya funktionaliteten lagts till och den nya indelningen och designen av koden. Resultatet blev som väntat en egen applikation fri från många av de externa bibliotek som den äldre implementationen var beroende av.

Nyckelord (sökord)
Java, Spring, Apache Mina, dokumentöverföring

Högskolans serienummer:	**ISSN:**	**Språk:**	**Sidantal:**
42:2016	1458-1531	Svenska	21 sidor

Inlämningsdatum:	**Presentationsdatum:**	**Datum för godkännande:**
15.12.2016	02.12.2016	19.12.2016

DEGREE THESIS
Åland University of Applied Sciences

Study program:	Information Technology
Author:	William Eriksson
Title:	Development of Document Exchange-module in Java/Spring
Academic Supervisor:	Agneta Eriksson-Granskog
Technical Supervisor:	Crosskey Banking Solutions

Abstract

The purpose of this thesis is to add functionality for downloading documents and to rewrite the old implementation. The thesis is done partly because the employer wanted functionality to download documents in their system and partly because they have noticed the current implementation was not good enough for new requirements that have come.

In this thesis there is a presentation of the solutions and design of the new functionality. The result is as expected an own application free from many of the external libraries that the old implementation were dependent on.

Keywords
Java, Spring, Apache Mina, document exchange

Serial number:	**ISSN:**	**Language:**	**Number of pages:**
42:2016	1458-1531	Swedish	21 pages

Handed in:	**Date of presentation:**	**Approved on:**
15.12.2016	02.12.2016	19.12.2016

INNEHÅLLSFÖRTECKNING

1. INLEDNING

1.1 Syfte

Syftet med detta examensarbete är att lägga till funktionalitet för nedladdning av dokument och samtidigt dela upp kodbasen som med åren fått en monolitisk struktur. Detta kommer även möjliggöra ett annat krav: att kunna köra koden på olika säkerhetsklasser av servrar. Även logiken när dokument och dokumentöverföringar skapas ska förenklas och bli helt Javabaserad.

1.2 Metod

Arbetet hade inledningsvis en grundläggande kravspecifikation men kom under arbetets gång att byggas på och specificeras genom kontinuerlig kontakt med uppdragsgivaren. Den kunskap som behövs för arbetet kommer i första hand från studier i Högskolan på Åland. Fördjupad kunskap inom de speciella teknikerna för arbetet kommer från Crosskey Banking Solutions. Utvecklingsarbetet görs i programmeringsspråket Java med ramverken Spring och MyBatis. I arbetet har stor vikt lagts vid att skriva tester då det är viktigt att försäkra sig om att koden inte bara fungerar lokalt hos utvecklaren utan även i servermiljö.

1.3 Avgränsningar

Funktionalitet för att ladda upp dokument existerar redan så den egentligen uppladdnings-koden ska inte göras på nytt, bara logiken för hur den körs igång. Den kodmässiga designen kommer till stor del att göras av uppdragsgivaren, såsom vilka gränssnitt och klasser som ska finnas, vilka beroenden som ska finnas och hur flödet i modulen ska se ut. Eftersom applikationen är en del av ett större system så är man bunden till samma ramverk och designmönster som används i resten av systemet. Databasen behöver inte modifieras från hur den ser ut före arbetet inleds vilket gör att ingen databasedesign kommer presenteras i arbetet.

2. BAKGRUND

2.1 Uppdragsgivare

Arbetet har gjorts åt avdelningen Cards and Mobile Payments på Crosskey Banking Solutions. Det är ett Ålandsbaserat företag med sidokontor både i Sverige och Finland som erbjuder kompletta IT-lösningar för företag som vill bedriva bankverksamhet eller ge ut bank/kreditkort. Allt från utveckling och kontinuerliga uppdateringar till hårdvara och underhåll erbjuds (Crosskey, 2016).

2.2 Kortsystemet

Kortsystemet är det system som Cards and Mobile Payments-avdelningen utvecklar och underhåller. Det har hand om allt från transaktioner, betalningar, när man handlar med sitt bankkort i affären till ett webbgränssnitt som bankpersonal kan administrera kort från.

Kortsystemet skickar och tar emot filer väldigt ofta. Det kan exempelvis vara beställningsfiler som skickas till korttillverkaren. Det låter lätt men det blir genast ganska komplicerat när man tänker efter vad det ska klara av. Ska dokumentet krypteras? Ska det komprimeras? Vart ska det? Hur ska databasen rörande dokument designas? Och så vidare. Eftersom man funnit behov av att skicka och ta emot dokument från olika säkerhetsklasser av servrar så insåg man att den befintliga implementationen inte kunde klara av detta och man ville förnya detta.

2.3 Tekniker

I och med att arbetet är en del i ett större system så användes samma tekniker här som i resten av systemet, vilket är bl.a. programmeringsspråket Java (Oracle, 2016) och ramverken Spring (Pivotal, 2016) och MyBatis (The MyBatis Team, 2016). Utvecklingsmiljön som används är Intellij IDEA (JetBrains, 2016). Det är ett kraftfullt utvecklingsverktyg för Java med stöd för versionshanteringssystemet Subversion (Apache, 2016) vilket är det som används under projektet.

2.3.1 Java

Java är ett kraftfullt, objektorienterat programmeringsspråk med en uppsjö av olika ramverk och bibliotek man kan dra nytta av. Java är även plattformsoberoende, vilket betyder att koden kan exekveras på både Windows-, Mac-, och Linuxmaskiner.

2.3.2 Spring

Spring är ett av dessa ramverk. Det håller automatiskt koll på infrastrukturen i ett projekt och hjälper utvecklaren fokusera på den egentliga applikationen. Den kanske enskilt största skillnaden mellan standard Java och Spring är att i Spring används "bönor" väldigt flitigt. Kort sagt är en böna ett Javaobjekt som Spring själv har instantierat. På så vis håller ramverket koll på alla olika objekt som har skapats. För att spring ska veta vilka bönor det ska skapa så måste man definiera dem genom antingen XML- eller Javakonfiguration (Pivotal, 2016). Figur 1 visar ett exempel på hur man kan definiera en böna för Spring.

```
@Bean
public DocumentRepository documentRepository2() throws Exception {
    final IbatisDocumentRepository2 repository = new IbatisDocumentRepository2();
    repository.setDocumentTableName("DOCUMENT");
    return repository;
}
```

Figur 1. Exempel på hur man definierar en böna för Spring.

2.3.3 MyBatis

MyBatis är ett annat ramverk som underlättar hanteringen av SQL-databaser i ett Java-projekt. Det görs i detta fall med SQL-mappar som är skrivna i XML-format. I figur 2 visas ett exempel på en del av en sådan SQL-map. På detta sätt får man smidigt ut alla SQL-skript från Javakoden till sina egna filer. Med MyBatis kan man även skapa så kallade Result Maps. Dessa är objekt som MyBatis använder för att konvertera resultatet från en SQL-query till ett Javaobjekt som man kan använda i koden.

```
<sql id="selectDocumentExchange">
    select
        DE.DOCUMENT_EXCHANGE_ID, DE.DOCUMENT_ENDPOINT_ID, DE.REMOTE_PATH, DE.REMOTE_TEMP_PATH,
        DE.DOCUMENT_TYPE_ID, DE.ENCRYPTION_KEY_ID, DE.ENCRYPTION_TYPE_CODE,
        DE.ENCRYPTION_ALGORITHM_TYPE,DE.HASH_ALGORITHM_TYPE,DE.COMPRESSION_TYPE_CODE,
    from
        DOCUMENT_EXCHANGE DE
</sql>
```

Figur 2. Exempel på hur hur en sql-map med MyBatis kan se ut.

2.3.4 Gradle

Gradle är ett kraftfullt verktyg som underlättar byggande, testande och körning av bland annat Java-projekt (Gradle Inc, 2016).

Man ställer in vilka beroenden olika moduler och applikationer har i dedikerade .gradle-filer:

```
compile project (":cbs-card-document-web")
```

På samma sätt lägger man även in externa beroenden:

```
compile "org.springframework:spring-core:$spring4Version"
```

3. SKAPA APPLIKATIONEN

En del av arbetet är att flytta filerna som hanterar dokumentöverföringar till en egen applikation. Tidigare har modulen för batch-jobb hanterat detta men som tidigare nämnts så fungerar det inte längre med den funktionalitet man önskar att denna modul ska klara av. Därför börjar den här delen av arbetet med att skapa upp den applikationen.

Sättet jag väljer att göra detta på är att kopiera en liknande modul för att få med projektstruktur såsom byggfiler, mappstrukturer, osv. Eftersom denna applikation är en del av en större system får man hålla i minnet att lägga till den nya applikationen i eventuella gradle-filer som länkar till alla applikationer.

3.1 Initialisering av Spring

För att Spring ska veta att det ska köra i den nya applikationen är det viktigt att göra en fil som kör igång Spring. Detta kan antingen göras via en XML-konfiguration, då gör man en fil som heter *web.xml* där Spring kommer att leta efter konfigurationer som den ska göra för att starta. Eller så gör man det med Javakod. Då kommer Spring leta efter en subklass till *WebApplicationInitializer* i vilken man gör all konfiguration/initialisering som man behöver. Figur 3 visar ett exempel på hur man kan göra detta. Notera att i figuren har klassen *AbstractSecurityWebApplicationInitializer* använts. Detta är i sin tur en förlängning av *WebApplicationInitializer* som används i Securitydelen av Springramverket.

```
@Configuration
@EnableWebMvc
public class InitWebApplication extends AbstractSecurityWebApplicationInitializer {

    @Override
    protected void beforeSpringSecurityFilterChain(ServletContext servletContext) {
        // Gör den initialisering man behöver
    }
}
```

Figur 3. Exempel på hur man via Javakod kan konfigurera spring.

3.2 Flytt av filer

När destinationsapplikationen för filerna är skapad så kan man flytta filerna. Själva flyttandet är lika lätt som det låter eftersom IntelliJ kommer att hålla reda på att göra samma ändring i versionshanterinssystemet. Man markerar filer och drar dom dit man vill att de ska flyttas. Men det är här man börjar stöta på problem i och med hur klasserna är beroende av varandra. En del problem med beroenden kommer dyka upp i och med att den tidigare implementationen har varit beroende av äldre moduler och ramverk som man nu vill försöka göra sig av med. Dessa problem kommer lösas genom att skriva om mekanismen för hur överförningarna skapas och körs igång. Detta kommer behandlas i kapitel 4.1 och 4.2.

4. REFAKTORERING AV APPLIKATIONEN

Med målet att göra hela dokumentöverföringsprocessen enklare så behöver man skriva om stora delar av den redan existerande koden. Tidigare har modulen varit baserad på Batch-modulen i Spring och har därför blivit baserad runt tanken om batch-jobb, alltså att överföringarna körts klumpvis. Integrationsramverket Mule (Mulesoft, 2016) har också använts för att sätta upp flöden för alla de olika överföringsjobben. Dessa flöden definieras i XML-filer. När mer och mer överföringingar har lagts till i systemet har dessa XML-filer blivit för stora och besvärliga att jobba med. Detta ska nu skrivas om så att allting blir Javabaserat och hela processen blir lättare att följa. Samtidigt kommer applikationen också skrivas så att endast en överföring i taget hanteras.

4.1 Skapa JMS-lyssnare

För att förstå detta kapitel behöver man känna till två tekniker: Java Messaging Service, JMS, och Java Naming and Directory Interface, JNDI, (Oracle, 2016). JMS är ett meddelande-API för Java som gör att man kan skapa meddelanden att skicka till andra klienter, eller i detta fall, andra applikationer. JNDI är ett Java API för att hämta resurser med hjälp av bara ett namn. Med detta kan man smidigt hämtar resurser utan att behöva göra sig beroende av andra delar av systemet.

Mekanismen som kör igång hela dokumentöverföringen ska lyssna på en JMS-kö som den får med hjälp av JNDI. När ett meddelande kommer till den kön med specifika parametrar så kommer mekanismen köra igång och starta överföringarna på basis av de parametrarna. JMS-lyssnaren skapas med Spring annotationer. Då får man väldigt ren kod och det är tydligt hur den fungerar. För att lyssnaren ska veta vad den ska göra när den får en sökväg till en destination och hur den ska skapa kopplingar mot leverantören av JMS-meddelanden måste man definiera en *ContainerFactory* åt lyssnaren. Figur 4 illustrerar hur en JMS lyssnare baserad på Spring annotationer kan se ut. Observera att man inte behöver peka ut en specifik *ContainerFactory*. Finns ingen definierad så letar Spring efter en böna som heter *jmsListenerContainerFactory*.

```
@JmsListener(containerFactory = "myContainerFactory",
             destination = "java:/jndi/path/to/my.queue")
public void triggerExchange(Message data) {


}
```

Figur 4. JMS-lyssnare med hjälp av Spring annotationer.

4.2 Förnyad batchlogik

Tidigare har klassen som gör överföringen gjort en mängd överföringar per körning. Men den ska skrivas om så att den bara hanterar en enda överföring och istället anropas flera gånger. För att möjliggöra detta ska logiken för hur överföringarna delas upp och körs igång skrivas om. Två huvudsakliga moduler kommer skapas: en service som JMS-lyssnaren anropar vilken kommer hantera skapandet av överföringsjobb och en hanterare för överföringsjobben. Tjänsten som skapas ska ta emot de parametrar man skickade i JMS-meddelandet och på basis av det söka upp överföringar i databasen och skapa Java-objekt av dessa. Därefter skickas jobben till hanteraren som tar hand om hur de exekveras.

4.2.1 Trådpooler

Hanteraren använder sig av så kallade trådpooler för körningen av överföringarna. En trådpool är som namnet antyder ett antal trådar som är samlade bakom ett gränssnitt. Det kommer skapas en pool för varje extern ändpunkt. Eftersom det för varje extern ändpunkt i databasen sparas hur många parallella uppkopplingar den kan hantera blir det naturligt att låta trådpoolens storlek, alltså hur många trådar som ingår i poolen, bestämmas från det. Då får man en snygg struktur i koden så att en trådpool kör mot en viss ändpunkt. Och varje överföring mot den ändpunkten körs i en egen tråd i motsvarande trådpool. I Figur 5 kan vi se logiken för hur trådpooler skapas.

```
private synchronized void queue(final TransferTask task) {

    DocumentEndpoint endpoint = task.getEndpoint();

    if(!executors.keySet().contains(endpoint)) {
        final int maxPoolSize = endpoint.getConnectionLimit();
        final ThreadPoolExecutor executor =
                (ThreadPoolExecutor) Executors.newFixedThreadPool(maxPoolSize);

        executors.put(endpoint, executor);
    }

    task.updateTransferStatus(TransferStatus.WAITING);
    executors.get(endpoint).submit(task);
}
```

Figur 5. Logiken för hur trådpooler skapas.

4.2.1 Skapande av överföringsjobb

För att göra det ytterligare mer flexibelt så tar trådpoolerna emot objekt som implementerar gränssnittet *TransferTask* och låter sedan både upp- och nedladdningsjobb implementera detta gränssnitt. Då kan man smidigt gå genom alla överföringsregler i databasen och kontrollera vilken typ av överföring det är frågan om och skapa ett jobb på basis av det. Sedan lägger man till dessa till korrekt trådpool. Ned- och uppladdningsjobb ska dock skapas på lite olika sätt. Uppladdningsjobb skapas precis så som man tror. Man pekar ut ett dokument, en destination, och ställer in diverse parametrar t.ex. komprimering, kryptering, eller om det ska få något speciellt filnamn. Nedladdningsprocessen är lite mer komplicerad. Den behandlas i kapitel 5.1. I Figur 6 kan vi se hur ett överföringsjobb skapas.

```
for (final DocumentExchange exchange : exchanges) {
    TransferTask task = null;

    if (DocumentExchangeType.UPLOAD.equals(exchange.getExchangeType())) {
        task = new DocumentUploadTask(endpoint, exchange);
    }
    else if (DocumentExchangeType.DOWNLOAD.equals(exchange.getExchangeType())) {
        task = new DocumentDownloadTask(endpoint, exchange);
    }
    transferTaskManager.queue(task);
}
```

Figur 6. Skapande av överföringsjobb.

4.3 Skapande av överföringar

Logiken för hur överföringar har lagts till i databasen har tidigare gjorts genom ett batch-jobb som går genom databasen och söker efter nya dokument. Om det hittas ett nytt dokument sedan senaste körning som har en överföringsregel associerad till sig skapas en överföring som en rad i databasen.

Den logiken är väldigt bristfällig på många sätt. Exempelvis, vad händer om ett dokument skapas mitt när det jobbet håller på att köras? Därför ska den bytas ut mot en ‘händelse-driven’ logik. Det vill säga när ett dokument skapas håller det själv reda på om det ska skickas någonstans och lägger till sig i databasen. I och med att överföringar är definierade baserat på dokumenttyp görs detta genom att dokumentet ser vilken typ det är och på basis av det skapar upp en överföring i databasen.

Med detta lättar man upp flödet för hur dokument hanteras. Men den största förbättringen är att det blir mycket mer pålitligt om en överföring skapas i databasen direkt när dokumentet skapas utan att man behöver vänta på att ett batch-jobb ska köras.

5. DOKUMENTNEDLADDNING

5.1 Skapande av nedladdningsjobb

Nedladddningsprocessen börjar med att lista alla filer i en katalog hos den externa ändpunkten. Subkataloger kan också inkluderas ifall man specifierar det. Vill man inte ha med alla filer i katalogen så går det att skapa ett reguljärt uttryck för att matcha mot de filnamn man är ute efter. Sedan skapas det ett tomt dokument i databasen för varje fil som hittas. Nu när man har ett 'destinationsdokument' i databasen kan man skapa nedladdningsjobb för dem. Nedladdningarna skickas därefter till hanteraren för överföringsjobben där de matchas ihop med trådpoolen för respektive ändpunkt och köas upp för körning. På det här sättet behöver man inte veta exakta filnamn eller själv skapa separata jobb för alla filer i en katalog.

5.2 Dokumentnedladdning

Nedladdningen börjar med att ett *Transport*-objekt skapas. Detta objekt sköter uppkoppling och filhantering på den externa ändpunkten. När man använder *Transport*-objektet för att ladda ner en fil så kommer innehållet i form av en *InputStream*. Det är ett standard Javaobjekt som representerar en ström av bytes. Man kan inte se på filinformationen om filen är komprimerad och/eller krypterad. Därför måste man lita på fördefinierade filspecifikationer som sändaren av filerna har angivit. Beroende på detta avkrypteras filen och packas upp. Krypteringsalgoritmer som stöds för tillfället är vanlig pgp-kryptering, antingen med eller utan signering (Phil Zimmermann, 1991). Komprimeringsformat som stöds är de standardiserade zip och gzip. Figur 7 visar flödet för den egentliga nedladdningen.

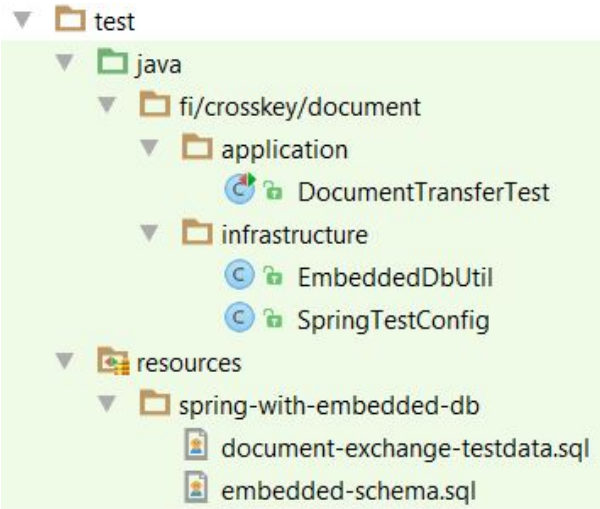
```
@Override
public void run() {
    try {
        String remoteFilename = "fileNameOnExternalEndpoint.txt";
        TransportFactory transportFactory = new TransportFactory();
        Transport transport = transportFactory.createTransport(documentEndpoint);

        InputStream downloadStream = transport.download(remoteFilename);
        decompressStream(downloadStream);
        decryptStream(downloadStream);

        Document document = createDocument();
        document.write(downloadStream);
        document.makePersistent();

    } catch (Exception e) {
        // handle exceptions...

    }
}
```

Figur 7. Flöde för dokumentnerladdning

6. TESTA APPLIKATIONEN

6.1 Teststruktur

Alla enhetstester används med det välbeprövade ramverket JUnit (JUnit, 2016) som även stöds av Spring, vilket gör att man kan använda Java-bönor som är definierade i Springs Config-filer. Detta gör att man säkert vet att det är samma objekt som körs både i testfallen och "på riktigt". Enhetstesterna körs mot en lokal databas. Detta så att man exakt kan styra hur databasen ser ut vid varje test så man vet exakt vad för data man jobbar med.



Figur 8. Design för testerna

På figuren ser vi hur allt hänger tillsammans i testerna.

- *embedded-schema.sql* definierar strukturen på databasen, dvs. tabeller och beroenden.
- *document-exchange-testdata.sql* fyller databasen med data som man kan använda i sina tester.
- *EmbeddedDbUtil.java* en hjälpklass som innehåller metoder för att smidigare hantera databasen, t.ex. *public void resetDatabaseContent();*
- *SpringTestConfig.java* är config-filen är testspecifika bönor definieras, här skapas bl.a. kopplingen till testdatabasen.
- I *application* paketet läggs sen simpelt alla testsviter, då har man åtkomst till testdatabasen och alla funktioner den kommer med.

6.2 Unit-tester

Unit-tester är tänkta att vara korta och tydliga tester. Dessa testar i detalj en specifik klass eller modul i taget så att den fungerar som man tänker sig. Med dessa tester ser man dock inte hur olika moduler beter sig tillsammans utan bara var för sig. Därför kommer även integrationstester att skrivas i kapitel 6.3.

6.2.1 Documenttransfertest

Dessa tester försäkrar sig om att funktionaliteten kring hur överföringar skapas i databasen fungerar korrekt. Detta genom att skapa olika ändpunkter och överföringsregler, och efter det skapa en överföring på basis av det och kolla att allt fick förväntade resultat.

Scenarion som testas är att överföringstabellen i databasen uppdateras när:

- dokument skapas, med och utan specifik ändpunkt
- överföring skapas
- överföring uppdateras

6.3 Integrationstester

Till skillnad från unit-testerna så är integrationstesterna mer övergripande. Därför kör de inte mot en egen databas utan mot samma databas som alla andra testmiljöer. De kan också bli lite längre i och med att dessa är flera moduler som testas tillsammans. Dessa tester kommer också att köras kontinuerligt med automationsverktyget Jenkins. Det betyder att med jämna mellanrum kommer Jenkins att bygga om hela systemet och sedan köra genom alla integrationstester. Med detta kan man på ett tydligt sätt se att de ändringar man har gjort inte bara fungerar lokalt på sin egen maskin utan också i en servermiljö.

6.3.1 Apache Mina SSHd

I och med att integrationstesterna ska koppla upp sig mot en server för att ladda upp filer så skulle testerna bli väldigt externt beroende av en server som den kan koppla upp sig på port 22 mot (sftp, krypterat filöverföringsprotokoll). Ett sätt att kringgå detta kan man göra med en del av Apacheprojektet Mina. Närmare bestämt SSHd-delen vilket är ett Javabibliotek som låter en skapa t.ex. servrar och klienter med simpel Javakod och köra sftp-trafik mellan dem. Då blir testerna inte beroende av någon extern server och man kan smidigt med biblioteket sedan verifiera filerna på servern.

6.3.2 Uppladdningstest

För att verifiera att funktionaliteten efter att dokumentuppladningslogiken fortfarande fungerar som förväntat ska ett integrationstest för detta skrivas. Testet ska använda Apache Mina (Apache, 2015) för att lägga upp en lokal sftp-server. Sedan ska den peka ut denna som ändpunkt för en dokumentuppladdning och köra igång överföringsjobbet. Efter det använder man Minas API för att manuellt nå filsystemet på den lokala servern och väntar på att dokumentet man laddar upp skapas där.

6.3.3 Nedladdningstest

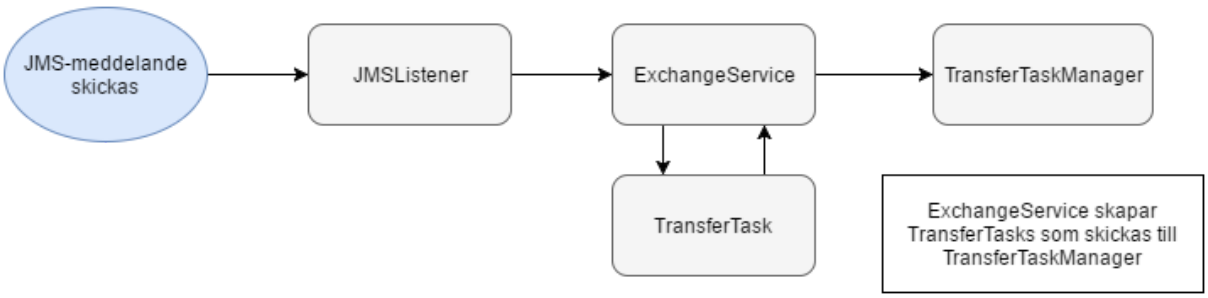
Det här integrationstestet använder igen Apache Mina för att skapa en lokal sftp-server. Sedan laddas ett dokument upp på den manuellt med Minas eget API. Överföringsregler skapas som pekar ut dokumentet på den lokala sftp-servern. Därefter kör man igång överföringen, väntar, och ser ifall dokumentet dyker upp i databasen. Kommer dokumentet fram dit så har man försäkrat sig om att nedladdningsfunktionaliteten fungerar som tänkt då man har testat det "på riktigt" från början till slut. I Figur 9 visas integrationstestet för att testa nedladdningsfunktionaliteten.

```
@Test
public void testDocumentDownload() throws IOException {
    createTestData();

    // create file on sftp server
    final String fileData = "Data for testfile... 0123456789!#¤%&";
    sftpClient.mkdir("directory");
    OutputStream os = sftpClient.write("directory/auto_test.txt");
    os.write(fileData.getBytes());

    triggerDocumentExchangeJob();

    await().pollInterval(2, TimeUnit.SECONDS)
            .atMost(60, TimeUnit.SECONDS)
            .until(documentIsDownloaded());

    Document document = documentTestDriver.fetchLatestDocument();

    assertEquals("Document content should be equal.", document.getFileData(), fileData);
}
```

Figur 9. Integrationstestet för dokumentnedladdning.

7. RESULTAT

Resultatet blev lyckat så när arbetet var klart hade en ny applikation skapats som innehöll all funktionalitet för dokumentöverföring. Detta gör det lättare att underhålla i och med att man vet var i systemet funktionaliteten finns och möjliggör också att man kan köra koden på olika säkerhetsklasser av servrar. En del externa beroenden har försvunnit i och med förbättringarna som har gjorts på den redan befintliga koden. Detta har också gjort flödet för hur dokument skapas mycket lättare att följa då det är helt Javabaserat. Pålitligheten för dokumentöverföring har också ökat tack vare förbättringarna med att börja använda händelse-driven logik istället för 'polling'-logik och att det nu finns kontinuerligt körande integrationstest för överföringslogiken. Med dessa förbättringar är det också möjligt att skapa ett webbgränssnitt så att man kan inspektera dokumentöverföringar utan att behöva ha några databaskunskaper, vilket kommer underlätta för personerna som testar systemet. Flödesdiagrammet i figur 10 visar hur flödet ser ut när överföringsjobb skapas.



Figur 10. Flödesdiagram för skapande av överföringsjobb.

Ett nytt ramverk för att smidigt skapa lokala sftp-servrar i ren Javakod har också blivit testat. I och med att man gärna vill att tester ska vara så oberoende som möjligt av 'världen utanför' så är detta inte bara väldigt användbart för integrationstesterna skriva i detta arbete utan kommer även vara till nytta för andra integrationstester som förlitar sig på att externa servrar finns.

Sedan självklart också huvudmålet med arbetet, att systemet på ett metodiskt och flexibelt vis kan ladda ner dokument som en del när överföringsjobbet körs. Implementationen för avkryptering och uppackning av filer finns på plats men är inte helt färdigt testat ännu så just de specifika bitarna kan eventuellt komma att ändras om någon bug hittas.

8. SLUTSATSER

I detta examensarbete har en modul som hanterar dokumentöverföringar uppdaterats och utökats. Tidigare var det en del av de olika batchjobben men har nu blivit en helt skild applikation i ett större system. Flödet för hur dokument skapas har även förenklats tack vare att externa beroenden har försvunnit och allt nu är Javabaserat. Tidigare var bara uppladdning av dokument möjligt men det har nu utökats till att det även är möjligt att ladda ner dokument. Integrationstest som kontinuerligt körs har skapats så att man hela tiden kan försäkra sig om att applikationen fungerar. Utvecklingen har gjorts i programmeringsspråket Java, med ramverken Spring och MyBatis. Som utvecklingsmiljö har verktygen IntelliJ IDEA och Oracle SQL Developer använts.

8.1 Fortsättningsarbete

Efter att detta arbete är utfört finns det redan planer på att implementera en REST-service för att kunna hämta information om överföringar med hjälp av webbanrop. Kortfattat är REST ett sätt att hantera data med hjälp av webbtjänster. Samt att göra ett webbgränssnitt för att smidigt kunna se överföringar som gjorts i systemet. Detta skulle underlätta mycket för de personer som testar systemet i och med att de nödvändigtvis inte har kunskap om hur man hämtar data ur databaser.

8.2 Egna reflektioner

Projektet hade redan från start en snäv tidsram. Därför var det tråkigt att jag redan i början stötte på en del problem jag inte hade räknat med till exempel att lägga till filer för att starta Spring när applikationen körs igång. Ett annat problem uppstod i och med hur integrationstesterna är upplagda så hade IntelliJ svårt att förstå projektstrukturen vilket ledde till att det tog ett tag innan jag ens kunde köra redan fungerande integrationstester.

Lyckligtvis flöt utvecklingen av nedladdningsfunktionalitetet på utan större problem. Det jag fastnade på där var hur skapandet av nerladdningsjobb skulle fungera i och med att jag hade missat en detalj i hur beställaren ville att nerladdningen skulle fungera. Dessutom hade jag problem med hur jag skulle hantera skapande och rensande av testdata som testerna använder sig av.

KÄLLFÖRTECKNING

Apache. (2015). Apache mina SSHd. Retrieved from http://mina.apache.org/sshd-project/. November 2016

Apache. (2016). Subversion. Retrieved from https://subversion.apache.org/. November 2016

Crosskey. (2016). Crosskey. Retrieved from https://www.crosskey.fi/. Oktober 2016

Gradle Inc. (2016). Gradle. Retrieved from https://gradle.org/. November 2016

JetBrains. (2016). IntelliJ IDEA. Retrieved from https://www.jetbrains.com/idea/. Oktober 2016

JUnit. (2016). JUnit. Retrieved from http://junit.org/junit4/. Oktober 2016

Oracle. (2016). Jndi. Retrieved from http://www.oracle.com/technetwork/java/jndi/index.html. Oktober 2016

Phil Zimmermann. (1991). Pgp. Retrieved from https://philzimmermann.com/EN/essays/index.html. November 2016

Pivotal. (2016). Spring framework. Retrieved from http://spring.io/. Oktober 2016

The MyBatis Team. (2016). MyBatis. Retrieved from http://www.mybatis.org/mybatis-3/. Oktober 2016