

UTVECKLING AV PLATTFORM FÖR HANTERING AV PARTNERSKAP

Måns Nygård



2021:41

Datum för godkännande: 30.09.2021
Handledare: Björn-Erik Zetterman

EXAMENSARBETE

Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Måns Nygård
Arbetets namn:	Utveckling av plattform för hantering av partnerskap
Handledare:	Björn-Erik Zetterman
Uppdragsgivare:	Crosskey Banking Solutions

Abstrakt

Under senare år har det växande samarbetet mellan banker och partner uppmärksamrats, samtidigt som behovet av hanteringen av dessa partner ökat. Genom uppdragsgivaren Crosskey Banking Solutions är syftet med detta examensarbete att utveckla och utöka hanteringen av Fintech-partner i deras kortsystem.

Med programmeringsspråket Java implementerades i det befintliga kortsystemet en abstraktion (entitet) för partner, där abstraktionen används för att etikettera data. Därtill erbjuds möjligheten att säkerställa en säker filtrering av data när partner-användare hämtar data från web-services med hjälp av utnyttjandet av Spring Security säkerhetsroller och den nya partner-entiteten.

Nyckelord (sökord)

Partner, Web Services, Java, Spring, MyBatis, säkerhetsroller

Högskolans serienummer:	ISSN:	Språk:	Sidantal:
2021:41	1458-1531	Svenska	33 sidor

Inlämningsdatum:	Presentationsdatum:	Datum för godkännande:
19.09.2021	30.09.2021	30.09.2021

DEGREE THESIS

Åland University of Applied Sciences

Study program:	Information Technology
Author:	Måns Nygård
Title:	Development of Platform for Management of Partnership
Academic Supervisor:	Björn-Erik Zetterman
Technical Supervisor:	Crosskey Banking Solutions

Abstract

In recent times there has been an increasing collaboration between banks and partners. Because of this growing partnership, there has also emerged a need to manage these partners in a more nuanced way. Through the employer, Crosskey Banking Solution, the purpose of this degree thesis is to develop and expand the management of Fintech-partners in their card system.

With the help of the programming language Java, an abstraction (entity) was implemented in the existing card system to label data. With the help of Spring Security roles in addition to the new entity, a correct filtering of the data fetched from the web services was ensured for the partner users.

Keywords

Partner, Web Services, Java, Spring, MyBatis, safety roles

Serial number:	ISSN:	Language:	Number of pages:
2021:41	1458-1531	Swedish	33 pages

Handed in:	Date of presentation:	Approved on:
19.09.2021	30.09.2021	30.09.2021

INNEHÅLLSFÖRTECKNING

1. INTRODUKTION	5
1.1 Syfte	5
1.2 Metod	5
1.3 Avgränsningar	6
2. VERKTYG OCH RAMVERK	7
2.1 Spring	7
2.1.1 IoC och Spring Beans	7
2.2 MyBatis	8
2.3 Flyway	11
2.4 Gradle	12
3. UTVECKLINGSPROCESSEN	14
3.1 Arkitekturella riktlinjer	15
3.2 Introduceringen av Partner-entiteten	17
3.2.1 Databas och domänobjekt	17
3.2.2 Repository och servicelager	18
3.3 Tillägget av partner-id på entiteterna	19
3.3.1 En första startpunkt	20
3.4 Kopiering av partner-id mellan entiteter	22
3.5 Filtrering av data med hjälp av säkerhetsroller	23
3.5.1 Validering av säkerhetsroll	25
3.5.2 Inkapsling av säkerhetsinformationen	27
3.5.3 Filtreringsflödet	28
4. SLUTSATSER	30
KÄLLOR	31

1. INTRODUKTION

1.1 Syfte

Via min externa uppdragsgivare Crosskey Banking Solutions¹ (CBS) är syftet med detta examensarbete att utveckla och utöka hanteringen av Fintech²-partner i deras kortsystem. Till följd av ett allt mer växande samarbete mellan CBS-kunder och Fintech-bolag, har det under senare år uppstått ett ökat behov av en mer nyanserad och komplex hantering av dessa partner, vilket kortsystemet för tillfället saknar.

I och med den växande involveringen av partner kan utformandet av en mer mångfacetterad partnerhantering motiveras med ett flertal punkter. I dagens läge är systemet uppbyggt på ett sådant sätt där en egentlig *Partner-entitet*³ saknas. Avsaknaden av denna entitet komplicerar framtida vidareutveckling av kunders samarbeten med olika partner med avseende på exempelvis säkerhet, skalbarhet och rapportering. Implementeringen av Partner-entiteten, samt inrättningen av denna i det befintliga systemet, är min huvuduppgift i detta examensarbete. Förloppet av arbetsprocessen är det som jag redogör för i kommande kapitel.

1.2 Metod

Mitt arbetsområde ingår i ett omfattande projekt för partnerutveckling inom CBS. Ett första steg i min arbetsprocess var därför att studera olika typer av dokumentation för projektet i stort för att skapa mig en bättre helhetsförståelse samt få en tydligare bild av vad som förväntas av mitt bidrag i slutändan.

¹ Crosskey Banking Solutions (CBS) är ett finskt mjukvaruföretag som erbjuder finansiella lösningar till banker och andra finansiella institutioner inom Norden. CBS var tidigare den interna it-avdelningen på moderföretaget Ålandsbanken Abp (Crosskey Banking Solutions, 2015).

² Enligt Kagan (2020) brukar Fintech beskrivas som den nya teknologi som försöker såväl förbättra och automatisera leveransen som användningen av finansiella tjänster. Grundtanken är att Fintech ska användas för att hjälpa företag, företagsägare samt konsumenter att bättre hantera sin finansiella verksamhet och sina processer genom att utnyttja anpassad mjukvara och algoritmer (Kagan, 2020).

³ En entitet kan beskrivas som ett enkelt domänobjekt. Oftast representerar en entitet en tabell i en relationsdatabas där varje rad motsvarar en unik instans av denna entitet. En entitet implementeras mer än sällan som en klass (Oracle, 2013b).

Till följd av att jag under en period har haft en traineetjänst på CBS, är jag väl bekant med kodbasen, arbetsformerna och verktygen som används vid utvecklingsprocessen. Av den anledningen behövde jag således inte ägna någon tid åt att lära mig dessa. Med detta sagt kommer jag däremot i följande kapitel att redogöra för de verktyg och hjälpmedel som hade en betydande roll för mitt arbete. Värt att lyfta fram redan här är att jag inte blev tilldelad en “separat” miljö att utveckla i. Allt arbete som jag utförde under projektets gång gick nämligen igenom samma processer som det dagliga arbetet på avdelningen. Detta betyder att jag exempelvis redogjorde för mitt arbete under så kallade *daily standups* - att min kod granskades genom *code-reviews* och testades i specifika testmiljöer av medlemmar i teamet som jag arbetade i.

Det som återstod innan jag kunde börja med den faktiska programmeringen var att anordna ett möte med en av arkitekterna för kortsystemet, tillika min handledare för projektet. Vi utformade, enligt det agila tillvägagångssättet, så kallade *stories*⁴ för mitt projekt där kravspecifikationer för de olika delmomenten definierades. Eftersom detta var ett projekt som fortfarande var i startgroparna fördes även en öppen diskussion angående designval då ingenting var skrivet i sten vid det här skedet. Diskussionerna med arkitekten skulle visa sig vara återkommande genom hela min arbetsprocess, och dessa har bidragit till en mängd såväl spännande som upplysande tankar och idéer kring bland annat designval.

1.3 Avgränsningar

I detta examensarbete har jag valt att inte redogöra för hur utvecklingsmiljön ser ut, med andra ord de verktyg som används av utvecklarna på CBS för arbetet med moduler och testning av dessa. Jag kommer dessutom inte visa den egentliga koden, eller den tillhörande datan från de olika tabellerna i databaserna. Dessa val kan motiveras med att jag inte anser det avgörande för förståelsen av hur själva arbetsprocessen har gått till. Däremot presenteras diagram, pseudokod samt exempeldata där det kan vara nödvändigt, för att belysa arbetets olika moment på ett mer lättöverskådligt sätt.

⁴ Stories eller *User-Stories* är vanliga inom det agila arbetssättet och består av korta beskrivningar av de krav eller förfrågningar som en slutanvändare har på den färdiga implementationen. Dessa mindre stories brukar ofta tillhöra en samling, även kallad *epic* (Atlassian, u.å.).

2. VERKTYG OCH RAMVERK

I detta kapitel presenteras de mest framstående verktyg och ramverk som användes under arbetets gång. Systemet jag arbetade inuti var stort och uppbyggt av en mängd olika moduler. I samband med systemets storlek stötte jag förvisso på fler ramverk och bibliotek, men de som jag har valt att redogöra för här är de som har spelat en central roll genom hela arbetets fortskridande. Kodbasen jag arbetade i var skriven i Java och IntelliJ IDEA, vilket är Crosskeys standard-utvecklarmljö. Detta är också anledningen till varför jag också använde samma verktyg i mitt examensarbete.

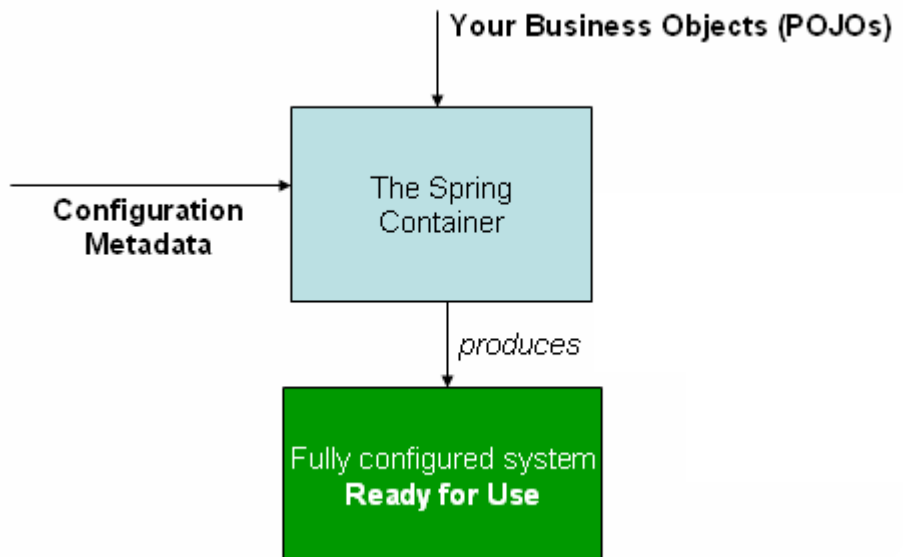
2.1 Spring

Spring är ett open source-ramverk för programmeringsspråket Java med avseende att簡simplifiera utvecklandet av företagsapplikationer (Pivotal, 2021a). Tack vare Springs förmåga att enkelt kunna integreras med andra ramverk, exempelvis Hibernate och MyBatis, har det sedan år 2004 varit ett mycket populärt webbramverk (Ginanjari & Hendayun, 2019). Utöver den enkla integrationen med andra ramverk har Spring även vunnit mark i och med erbjudandet av betydligt enklare och mer produktiva lösningar i utvecklandet av webbapplikationer än det tidigare brett använda Java API:et Enterprise Java Beans, idag namngivet Jakarta Enterprise Beans (Arthur & Azadegan, 2005).

2.1.1 IoC och Spring Beans

Spring's infrastruktur bygger på deras *Inversion of Control Container* (IoC) som innehåller alla de objekt, eller *Spring Beans* (sv. bönor), som är nödvändiga för en Spring-applikation. Gränssnittet, även kallat *applikationskontexten*, som representerar denna container har som uppgift att instansiera, konfigurera och bygga bönorna. Hur detta ska utföras definieras i konfigurationsfiler som bidrar med metadata till applikationskontexten. Metadata i dessa konfigurationsfiler är exempelvis ett paketaddresserat klassnamn som pekar på den egentliga klassimplementationen av bönan. Ett annat exempel på metadata är referenser till andra bönor

som konfigurationsbönan är beroende av (Pivotal, 2021b). Figur 1 nedan illustrerar en bild av hur ovannämnda flöde ser ut.



Figur 1. Illustration av IoC (Pivotal, 2021b).

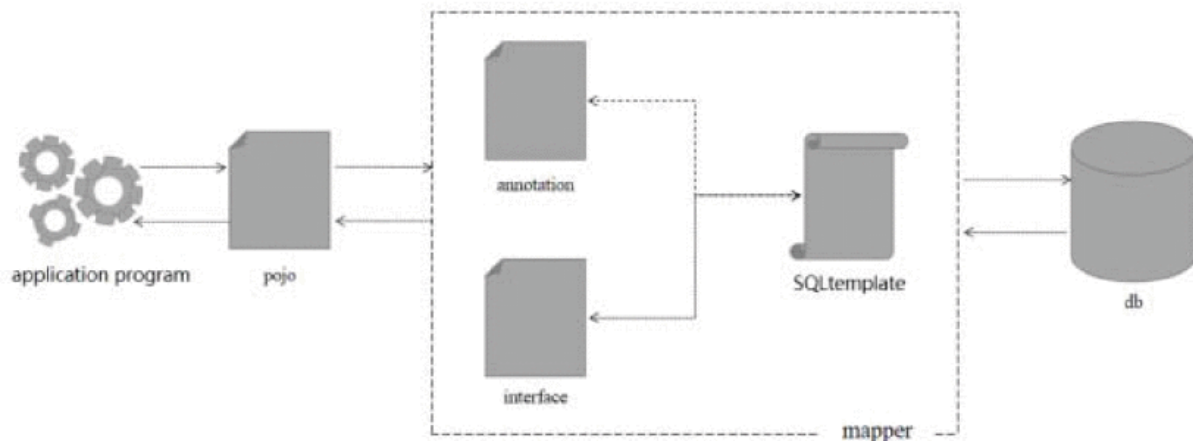
Springs egna namn på detta koncept är *Dependency Injection* (sv. beroendeinjektion), vilket är en process där objekt definierar sina beroenden. Med beroenden avses i detta fall de övriga objekt som objektet i fråga behöver för att fungera. Denna beroendeinjektion sker genom exempelvis konstruktor-argument eller genom argument till en factory-metod. Containern injekterar sedan dessa beroenden när den skapar ett objekt, eller en böna. Det är alltså bönan själv som styr över instansieringen eller platsen av dess beroenden (Pivotal, 2021b). Arthur och Azadegan (2005) jämför IoC:s koncept med det berömda talesättet från Hollywood: “Don’t call us, We’ll call you” (Arthur & Azadegan, 2005).

2.2 MyBatis

MyBatis är ett *persistence*-ramverk med stöd för egenproducerad SQL-kod. Ramverket används för att kunna eliminera all *JDBC*-kod⁵ samt den manuella konfigurationen av parametrar och databasresultat. Detta sker med hjälp av XML-kod, eller annotationer, för att

⁵ JDBC, eller Java Database Connectivity, är ett API som implementeras i applikationer för att få tillgång till databaser genom Java-kod. API:et hjälper användaren att upprätta en databasförbindelse, skicka förfrågningar och utföra uppdateringar samt bearbeta resultat från databasoperationer (Oracle, 2020a).

konfigurera sammanlänkningen från primitiver, gränssnitt och *POJOs* (Plain Old Java Object) till databasobjekt (MyBatis, 2020a). Figur 2 nedan visar en grundläggande modell av ramverkets uppbyggnad där “*SQLtemplate*” representerar den XML-fil där SQL-koden skrivs.



Figur 2. Modell av ramverket MyBatis (X.Shiyong et al, 2020).

Inuti XML-filerna, eller mallarna, kombineras ifyllning av parametrar, konvertering av datatyper samt dynamisk SQL-kod till snabba och flexibla databasförfrågningar (X.Shiyong et al., 2020). Genom de fåtaliga element som existerar inom ramverkets XML-filer behöver inte användaren skriva upprepad *JDBC*-kod, och kan således fokusera på den egentliga SQL-koden (MyBatis, 2020b). Figur 3 illustrerar en *resultMap* inuti en MyBatis XML-fil och figur 4 (s. 10) visar exempel på en lagrad operation inuti samma fil.

```

<resultMap id="userResultMap" type="User">
  <id property="id" column="user_id" />
  <result property="username" column="user_name"/>
  <result property="password" column="hashed_password"/>
</resultMap>

```

Figur 3. Exempel på en simpel resultMap inuti en MyBatis XML-fil (MyBatis, 2020b).

Förklaringen av kodexemplet i figur 3 ovan lyder enligt:

1. `<resultmap id="userResultMap" type="User">`

Exempel på en deklaration av standard-elementet *resultMap*. Denna deklaration förklarar hur konverteringen från databasinformationen till Java-klassen ska utföras. Elementet kräver ett unikt id och till vilken Java-klass som den hämtade datan ska konverteras till. I detta fall är det klassen *User*.

2. `[...] property="username" column="user_name"/>`

Property hänvisar till attributnamnet i Java-klassen och *column* hänvisar till vilken databaskolumn som ska motsvara attributets värde.

```
<select id="selectUsers" resultMap="userResultMap">
  select user_id, user_name, hashed_password
  from some_table
  where id = #{id}
</select>
```

Figur 4. Kodexempel på en lagrad operation inuti en MyBatis XML-fil (MyBatis, 2020b).

Förklaringen av kodexemplet i figur 4 ovan lyder enligt:

1. `<select id = "selectUsers" resultMap="userResultMap">`

Denna tag visar på en av MyBatis standard-element, nämligen en *select-tag* med ett unikt id. Följande attribut anger till vad resultatet ska mappas till. Vad gäller detta fall är det *userResultMap* som är illustrerad i figur 3.

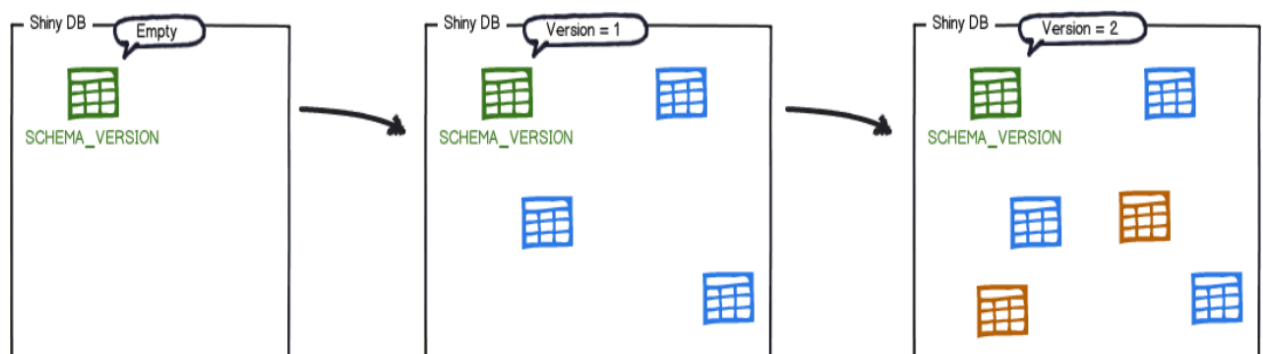
2. `[...] where id = #{id}`

Ett exempel på ifyllning av parametrar, där `#{id}` är metodparametern för den motsvarande Java-metoden med det överensstämmande namnet, *selectUsers*. Denna Java-metod deklareraras oftast i ett gränssnitt.

2.3 Flyway

Flyway är ett open source-verktyg för databas-migration och versionshantering. Ramverket stöder SQL, och har databas-specifik syntax, vilket möjliggör att man enkelt kan skriva databasförfrågningar i SQL-filer som sedan verkställs i korrekt ordning när exempelvis en applikation startas. Ramverket erbjuder även plugin för Spring Boot-applikationer (Flyway 2020a).

Figur 5 nedan visar ett simpelt flöde från en tom databas till en databasstruktur två versioner senare.



Figur 5. Illustration av ett flöde i Flyway där nya tabeller har adderats med hjälp av migration i de olika versionerna (Flyway, 2020b).

För att hålla reda på databasens tillstånd och vilka migrationer som redan har blivit applicerade skapar Flyway en “bokföringstabell”. Denna tabell består av metadata för att hålla reda på kontrollsummor för migrationer samt information om migrationen var lyckad eller ej (Baeldung, 2021a). Figur 6 belyser strukturen för en sådan tabell.

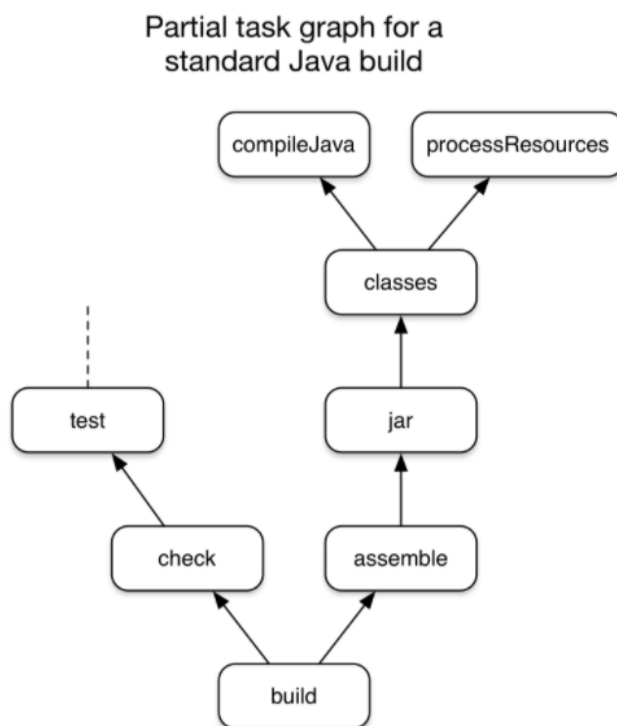
installed_rank	version	description	type	script	checksum	installed_by	installed_on	execution_time	success
1	1	Initial Setup	SQL	V1__Initial_Setup.sql	1996767037	axel	2016-02-04 22:23:00.0	546	true
2	2	First Changes	SQL	V2__First_Changes.sql	1279644856	axel	2016-02-06 09:18:00.0	127	true

Figur 6. Exempel på en “bokföringstabell” innehållande metadata för två SQL-migrationer (Flyway, 2020b).

2.4 Gradle

Gradle är ett hjälpverktyg som automatiserar byggandet av applikationer och är ofta förekommande för applikationer skrivna i Java. Hur byggnadsprocessen ska gå till för en applikation definieras i build-filer som beskriver i vilken ordning uppgifter ska utföras. Dessa uppgifter kan exempelvis vara hämtning av tredjepartsbibliotek eller kompilering av en specifik modul (Gradle Inc, 2021a).

Figur 7 illustrerar hur en byggnadsprocess med ett antal deluppgifter kan se ut i en Java-applikation.

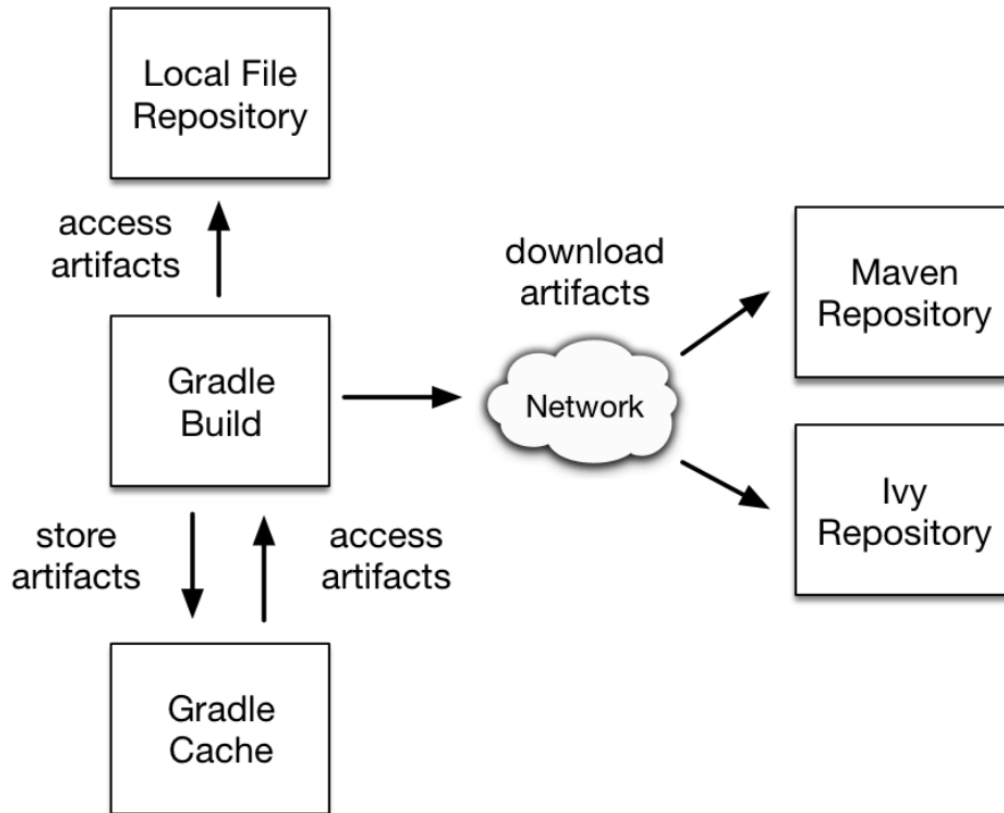


Figur 7. Exempel på en uppgiftsgraf i Gradle. Pilarna beskriver beroenden, det vill säga i vilken ordning uppgifterna bör utföras. (Gradle Inc, 2021a).

En stor fördel med att använda Gradle som byggnadsverktyg är dess inbyggda hantering av beroenden. Då större projekt sällan är uppbyggda som en enda stor monolit, utan ofta är

uppdelade i flera moduler och använder sig av importerade bibliotek, behövs ett tillvägagångssätt för att hantera dessa beroenden (Gradle Inc, 2021b).

Figur 8 nedan ger en förenklad bild av hur Gradles beroendehantering ser ut.



Figur 8. Överblick av Gradles beroendehantering (Gradle Inc, 2021b).

3. UTVECKLINGSPROCESSEN

I Crosskeys övriga banksystem existerar redan en abstraktion för de olika partnerna. Kortsystemet, som detta examensarbete utförs i, stödjer endast banker inom samma miljö eller installation, exempelvis finska eller svenska banker. Partnerskapet mellan en bank och en partner komplicerar dock det “normala” sättet att behandla data i systemet. Att hantera en partner som en bank blir således inte hållbart i detta scenario.

En orsak till att inte hantera partner som banker i kortsystemet beror på dupliceringen av data och konfiguration som skulle ske på grund av partnerskapet. En annan aspekt som också försvårar tillvägagångssätt med att hantera partner som banker är att alla de individuella bankerna i systemet “äger” sina kunder. Detta är dock inte fallet för en partner eftersom kunderna till en partner juridiskt sett tillhör banken som ingår i partnerskapet. Utöver ovannämnda orsaker skulle integrationen mellan kortsystemet och de övriga systemen försvåras om partnerhanteringen skedde på olika sätt. I och med tillskottet av en separat partner-entitet närmar sig således kortsystemet de övriga systemen, vilket är ett av huvudmålen för projektet som detta examensarbete ingår i.

Nedan presenteras de enligt CBS huvudsakliga orsakerna till behovet av en egen partner-dimension:

- **Säkerhet.** I och med ett ökande partnersamarbete är en filtrering av data tillhörande dessa partner essentiell. Olika partner i kortsystemet bör endast ha tillgång till uppgifter som är sammankopplade med deras egna produkter. En ny partner-dimension förenklar detta.
- **Administration.** Att lägga till och att ta bort produkter sammankopplade med olika partner förblir lättare då partner-produkter för tillfället endast är åtkomliga med hjälp av bestämda produktidentifikatorer som försvårar hanterandet av dessa.
- **Spara tid.** Införandet av en ny partner i kortsystemet skulle vara en betydligt mindre tidskrävande process än att införa exempelvis en ny bank.

- **Premium API:er.** Möjliggör säkerheten gällande partner-data för de kommande API-förändringarna i kortsystemet.
- **Rapportering.** Att konstruera filer och rapporter för olika partner kommer även att förenklas i och med den tydligare etiketteringen av partner-datat som behövs för att framställa dessa filer och rapporter.

3.1 Arkitekturella riktlinjer

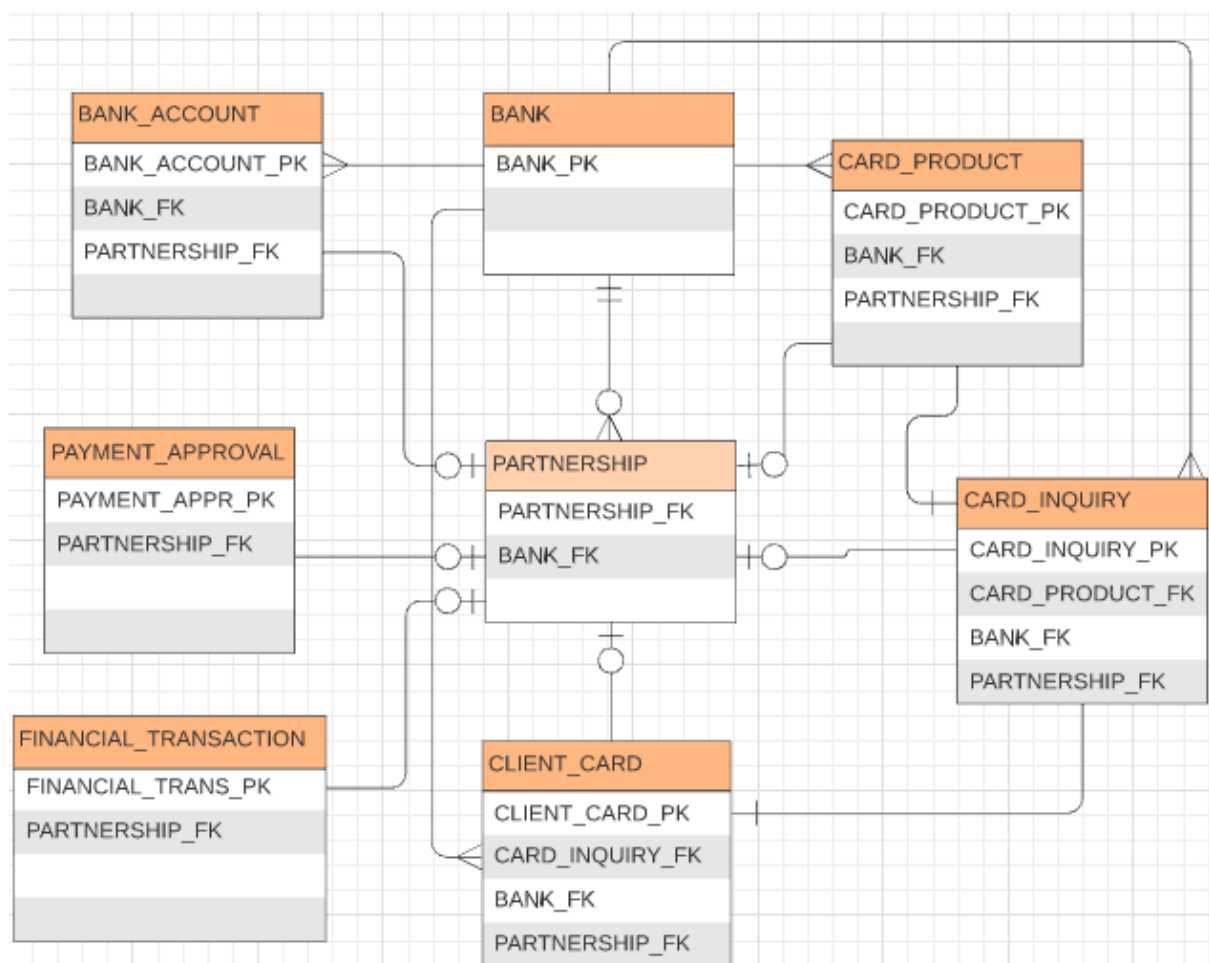
Innan jag började med själva utvecklingsprocessen tog jag del av den dokumentation gällande de arkitekturella riktlinjerna som de projektansvariga hade lagt fram. Detta dokument var endast riktlinjer, och en del designval var därför inte slutgiltiga, vilket i sin tur ledde till ett antal modifieringar under projektets gång. Härnäst presenterar jag den önskade strukturen för den nya partnerskaps-hanteringen, där ord med versaler refererar till databastabeller som illustreras i figur 9 (s.16) i form av ett ER-diagram (Entity Relationship Diagram).

Varje bank (BANK) med den nya partnerskaps-dimensionen aktiverad ska kunna ha en eller flera partner knutna till sig med ett unikt id. Samtidigt kommer varje produkt inom den banken att tilldelas ett partner-id, där detta id kommer representera en utomstående partner eller banken själv. Då en kortansökan från en partner-användare mottages ska en kontroll utföras ifall användaren är autentiserad för partnern i fråga. Ifall användaren är autentiserad ska partner-id:et från CARD_PRODUCT, som representerar ett betalkort i det här exemplet, bli kopierat och lagrat i CARD_INQUIRY som representerar en kortansökan. Vid det tillfället en kortansökan blir godkänd bör partner-id:et kopieras till de nya insättningar i CLIENT_CARD samt BANK_ACCOUNT, vilka representerar kundens kort och kortkonto. När autentiseringar i samband med transaktioner utförs för betalkort eller konton knutna till ett partner-id ska de tillagda raderna i PAYMENT_APPROVAL samt PAYMENT_TRANSACTION även innehålla samma partner-id.

Under arbetets gång bestämdes det att även tabellen, som inte illustreras i ER-diagrammet, STATEMENT skulle smyckas med ett partner-id. I denna tabell sparas information om

kreditfakturer. I samband med att ett partnerkonto med tillhörande partner-id ska faktureras kopieras således id:et från kontot när en insättning i tabellen görs.

Det ovannämnda flödet ska således bidra till en automatisk “taggning” av databasuppgifter tillhörande respektive partner. Den tydliga etiketteringen av data förenklar i sin tur datafiltreringen eftersom det finns ett konkret attribut att göra själva filtreringen på. För att utföra datafiltreringen implementerades en åtkomstkontroll. Åtkomstkontrollen ska med hjälp av bestämda säkerhetsroller för partner-användare av systemets API:er jämföra dessa mot det unika partner-id:et och därefter filtrera bort det data som användaren inte bör ha åtgång till. I figur 9 nedan illustreras ER-diagrammet som beskrevs ovan. Syftet med diagrammet är att åskådliggöra vilka redan implementerade entiteter som kom att påverkas av den nya Partner-entiteten.



Figur 9. ER-diagram efter implementerad Partner-tabell (PARTNERSHIP).

3.2 Introduceringen av Partner-entiteten

Kortsystemets kodbas är uppbyggd av en mängd olika moduler för att förenkla uppdelningen av olika entiteter och deras tillhörande serviceklasser, konfigurationsfiler med mera. Det fastställdes att den nya Partner-entiteten borde implementeras i *Company*-modulen där abstraktionen för bankerna huserar. Valet av denna modul baserade sig på idén om att de båda entiteterna i stora drag torde bidra till samma sak; en tydligare uppdelning av vilket data som tillhörde vart.

3.2.1 Databas och domänobjekt

Det första steget som behövde tas i och med skapandet av den nya Partner-entiteten var att ta bort den befintliga databastabellen med samma namn. Detta var en tabell som inte längre var i användning, och hade i ett tidigare skede syftat på något annat än det jag nu skulle implementera. För att säkerställa en riskfri borttagning av nuvarande främmande nycklar till andra tabeller behövde jag kontrollera med en *Application Manager* (AM) ifall den gamla tabellen innehöll någon data i produktionsmiljön. Efter att detta hade undersökts kunde jag med hjälp av ett Flyway SQL-script ta bort den gamla tabellen samt radera kod som refererade till denna i MyBatis-filerna.

Efter borttagandet av den gamla tabellen var det dags för inkorporerandet av den nya Partner-tabellen. Även detta utfördes med hjälp av Flyway. Den nya tabellen skulle bland annat innehålla kolumnen PARTNERSHIP_FK (primärnyckel) samt BANK_FK, vilket var den främmande nyckeln som refererade till BANK. Primärnyckeln kom att implementeras som en 32-tecken lång GUID (Globally Unique Identifier).

Domänobjektet, eller i det här fallet Partner-klassen, byggdes som en *immutable entity*. Ett objekt som är *immutable* är ett objekt som, när det väl är skapat, inte längre kan ändras. Detta blev det självklara designvalet eftersom Partner-entiteten egentligen bara behövde hämtas i samband med datafiltrering. Utöver detta bestämdes det att partner endast skulle adderas och uppdateras med hjälp av manuella databasoperationer. Figur 10 nedan (s.18) visar en förenklad implementation av Partner-klassen med hjälp av *Lombok*-annotationer.

```

@Value
@Builder
@RequiredArgsConstructor
public class Partner {

    String partnerId

    Long companyId

    //...
}

```

Figur 10. Partner-klassen implementerad som en immutable entity med hjälp av Lombok annotationer.

Lombok är ett populärt Java-bibliotek som gör det enklare att generera template kod som bland annat *getters*, *setters* och *loggingsvariabler* (Project Lombok, 2021). Partner-klassen, som figuren ovan visar, smyckades med hjälp av några av dessa annotationer. *@Value* är annotationen som antyder att klassen bör vara immutable eftersom klassens fält automatiskt deklarerar som “*private final*” samt att inga *setter*-metoder skapas.

3.2.2 Repository och servicelager

Ett MyBatis-repository behövde implementeras för att kommunicera med databasen. För att systemet skulle veta vilket repository samt xml-fil med databasoperationer som hörde ihop med det nya domänobjekt behövde detta definieras i en Spring-konfigurationsfil. Här skapades en Spring-böna i form av klassen *MapperFactoryBean* där repository-gränssnittet *PartnerRepository* sedan lades till. I detta gränssnitt definierades sedan metoder som överensstämde med de databasoperationerna i MyBatis xml-filen med namnet “*partner-mapper.xml*”. Operationerna som implementerades var:

- *Partner getPartner(@Param("partnerId") String partnerId)*
- *List<Partner> getPartnersByCompanyId(@Param("companyId") Long companyId)*
- *Partner getPartnerByCode(@Param("partnerCode") String partnerCode)*
- *boolean isPartnershipEnabled(@Param("companyId") Long companyId)*

Abstraktionslagret ovanpå databaslagret implementerades i form av en enstaka serviceklass med namnet *PartnerService*. Serviceklassen kom i princip att fungera som repository-gränssnittets förlängda arm eftersom dess enda roll var att anropa repository-gränssnittet. De operationer som implementerades här var således de samma som nämndes ovan. Efter att dessa två lager hade färdigställts utformades enhetstester för att säkerställa att alla operationer fungerade som tänkt för både repository-gränssnittet och serviceklassen.

3.3 Tillägget av partner-id på entiteterna

Som rubriken avslöjar var det nu dags att introducera det nya partner-id:et på alla de entiteter som redogjordes för i ER-diagrammet ovan (s. 16). Detta kom att visa sig vara en lång process på grund av duplikationer av dessa entiteter och repository. När den nya kolumnen lades till i de befintliga tabellerna behövde alla processer kopplade till dessa tabeller uppdateras. För att enklast beskriva detta flöde kan tillvägagångssättet delas upp i följande steg:

1. Uppdatera befintlig tabell med den nya kolumnen `PARTNER_ID` samt definiera en ny främmandenyckel till Partner tabellen som refererade till `PARTNER_ID` i den tabellen.
2. Uppdatera MyBatis xml-filen som skötte förfrågningarna till associerad tabell. Detta rörde sig om ändringar i *resultMapen* och databasoperationerna definierade i dessa filer.
3. Lägga till den nya medlemsvariabeln, *partnerId*, till det domänobjekt som var sammanlänkat med tillhörande *resultMap* i xml-filen. I de fall där inte Lombok-annotationer användes behövde manuella implementationer av *getters* och *setters* tillämpas.
4. Uppdatera de lokala tabellerna som enhetstesterna använder sig av. Dessa tabeller utnyttjar Springs *in-memory-databaser* och är alltså lokala databaser som endast används då enhetstesterna körs för att slippa uppkopplingar till de riktiga databaserna.

Efter ovannämnda ändringar var det viktigt att följa med avdelningens *Jenkins-pipeline*⁶ för att kontrollera att ändringarna inte hindrade systemet att byggas korrekt eller ifall integrationstesterna misslyckades. Till följd av de ovannämnda dupliceringar som domänobjekten och repositories hade, krävdes ett antal försök innan systemet fungerade felfritt, då dessa nya ändringar kom att förbises på ett antal vitala ställen i koden.

3.3.1 En första startpunkt

När strukturen för Partner-entiteten var på plats krävdes en första startpunkt för det faktiska introducerandet av partner-id:et på entiteterna som skulle vara till förfogande för den framtida filtreringen av partner-data. Som tidigare nämnts i stycket om de arkitekturella riktlinjerna, var produkt-entiteten den essentiella utgångspunkten. För att göra det möjligt att ändra partner på befintlig produkt, eller koppla samman en partner i och med införandet av en ny produkt, var jag tvungen att göra ändringar i kortsystemets portal. Denna portal, eller *backoffice* som den också kallas, är kortsystemets gränssnitt för de användare som jobbar med kort-procedurer. Dessa användare är oftast bankpersonal, men även CBS-anställda använder portalen för arbetsrelaterade uppgifter.

Riktlinjerna för hur detta skulle implementeras rent praktiskt var få, vilket ledde till att jag behövde göra en ordentlig genomgång av den befintliga koden som skötte om sidorna för de produktbaserade uppgifterna. Det jag fick information om att behöva utföra var:

1. Lägga till funktionalitet (backend/frontend) för att kunna koppla ihop en partner med en produkt på sidan för informationsuppdatering av en produkt.
2. Lägga till funktionalitet (backend/frontend) för att kunna koppla ihop en partner med en produkt på sidan för skapandet av en ny produkt.
3. Lägga till funktionalitet (backend/frontend) för att visa produktens tillhörande partner på sidan för produktöversikt.

⁶ En *Jenkins Pipeline*, eller bara Pipeline, är ett samlat system av plugins för att enkelt kunna övervaka källkoden i systemet samt automatisera processen från versionskontroll till leverans för kund. Varje ändring som görs i koden går igenom denna "tunnel" av kvalitetskontroller för att kunna säkerställa en säker och kvalitativ slutleverans (Jenkins, u.å).

Portalen är uppbyggd med hjälp av tredjepartsbiblioteket *Liferay*, som är ett populärt ramverk för att utveckla just portaler. Jag kommer inte i detalj att beskriva hur ramverket fungerar, utan endast nämna den inbyggda funktionen *portlets*. *Portlets* är *Liferays* implementation på webbapplikationer som hanterar förfrågningar och genererar svar som kan visas i webbläsare, exempelvis HTML. Dessa *portlets* är dessutom självständiga och kan presenteras på en webbsida i form av enskilda komponenter (Liferay, 2019). Produkt hade en egen *portlet* och det var här jag behövde göra mina ändringar.

Inom kontexten för *product-portlet* fanns en färdig *controller class* som hanterar förfrågningar och producerar svar; *ProductManagementController*. Klassens ansvar är att ta emot förfrågningar för att sedan utföra logiska operationer och till sist skicka iväg svar som kan renderas på tillhörande produktsidor i portalen. Arkitekturen påminner alltså mycket om det klassiska *MVC*-konceptet (Model View Controller). Metoderna i kontrollklassen utnyttjar också Spring Web för att koppla ihop förfrågningar med rätt metod samt binda ihop värden med *model-attributes* för att enklare kunna “transportera” värden mellan processer. Figur 11 illustrerar hur en metod i en kontrollklass kan se ut som både använder *portlets* och Spring web.

```
@RolesAllowed("SOME_ROLE")
@RequestMapping(params = "action=saveSomething")
public void saveSomething(
    final ActionRequest request,
    final ActionResponse response,
    @ModelAttribute("SomethingDTO") final SomethingDTO dto,
    final Model model)
```

Figur 11. Exempel på en metoddeklarering för en operation i en kontrollklass.

Jag lade till den nya *PartnerServicen* i kontroll-klassen för att kunna hämta och addera partner-id:et till *DTO*-objekten⁷ som användes för att bygga upp formulären som skulle visas på produktsidorna. Detta var formulär för att skapa en ny produkt samt formulär för att ändra

⁷ *DTO:s* eller *Data Transfer Objects* är enkla objekt utan någon innehållande “business logik” som transporterar information mellan processer för att minska antalet metoanrop (Baeldung, 2021b)

informationen om en befintlig produkt. Genom att kontrollera ifall användaren som använde portalen var inloggad via en bank som hade partnerdimensionen påslagen med hjälp av *PartnerService*-metoden, *isPartnerShipEnabled*, kunde jag sedan hämta ut och lägga till den tillgängliga partner-informationen till DTO:n som sedan skulle visas på sidorna.

Tillägget på de bägge sidorna blev en “dropdown meny”, med tillgängliga partner för produkten, vilka användaren kunde välja mellan. För att säkerställa att ett korrekt partner-värde angetts vid slutgiltig produktkonfiguration uppdaterade jag valideringsprocessen för formuläret. Ifall ett felaktigt värde hade blivit inmatat av användaren visas ett felmeddelande på sidan och användaren ombeds kontrollera valet. I och med att produkten också skulle uppdateras i databasen var det extra viktigt att partner-värdet var korrekt. Det sista som återstod var att lägga till informationen om produktens valda partner på översiktssidan och partner-id:et hade nu fått ett fotfäste i kortsystemet. Nu var det dags för implementeringen för överförandet av partner-id:et mellan de återstående entiteterna.

3.4 Kopiering av partner-id mellan entiteter

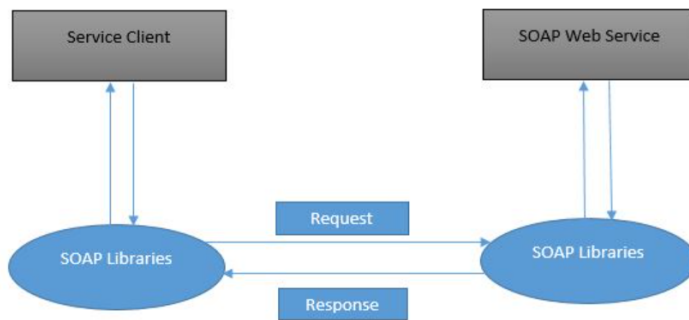
Det som återstod innan datafiltreringen kunde implementeras var tillämpningen av logiken för att kopiera partner-id:et mellan entiteter då de skapas, eller med andra ord sparas i tillhörande tabeller. Nu skulle flödet som beskrevs i kapitel 3.1, arkitekturella riktlinjer, således realiseras i kod. I likhet med processen för adderande av partner-id-kolumnen var även detta en tidskrävande fas som krävde ett flertal försök innan det fungerade som tänkt. Med det sagt behövdes inte en stor mängd kod eller några banbrytande ändringar i nuvarande logik, utan ett omfattande sökande i kodbasen efter de ställen där kopieringen behövde ske i och med databasinsättningarna. En viktig faktor till den extensiva sökningen var den otillräckliga kunskap jag besatt av de moduler där mycket av kopieringen skulle ske, men också de många variationerna av dessa abstraktioner som existerar i systemet komplicerade arbetet. Ett exempel för att belysa detta är de fem till sex olika sätten en transaktion kan tillkomma på.

Om *card product* hade haft en betydande roll i föregående fas då en första “kopiering” av partner-id:et implementerades, hade kortansökningar (*card inquiry*) en betydande roll i denna fas. Via processen för hanteringen och godkännandet av kortansökningar i kortsystemet behövde partner-id:et kopieras från produkten till ansökan för att därefter kopieras till två andra entiteter; *client card* och *bank account*. Speciellt *bank account* visade sig användas frekvent i framtida kopiering och filtrering för att hämta ut partner-id:et. Som nämnts tidigare behövdes ingen komplex logik för att lösa kopieringen mellan entiteterna, utan oftast kunde kopieringen ske genom att “föra över” värdet mellan DTO:s. I de fall partner-id:et inte fanns att tillgå från de objekt som förekom i skedet då kopieringen behövde ske, fanns det ofta andra attribut som kunde användas för att hämta ut tillhörande entitet med ett partner-id, exempelvis *bank account* eller *client card*. Efter att mina ändringar hade genomgått hela flödet av testning, för att kontrollera att partner-id:et nu befann sig på de förväntade uppgifterna i databasen, var det dags att implementera logiken för filtreringen av denna data i samband med hämtandet av den från partner-användare.

3.5 Filtrering av data med hjälp av säkerhetsroller

Den nya datafiltreringen skulle implementeras i kortsystemets externa API:er, som för tillfället består av *SOAP Web Services*, men i allt större utsträckning har eller ska bli ersatta av *REST* API:er. För att skapa en bättre överblick tänkte jag inleda detta avsnitt med att beskriva tre huvudkoncept, eller verktyg, som används för att bygga upp kortsystemets web services; *SOAP*, *WSDL* och *JAX-WS*.

SOAP står för *Simple Object Access Protocol* och är ett standardiserat XML-protokoll för att designa och utveckla web services. Eftersom protokollet är XML-baserat blir server-klient modellen språk-och plattformsoberoende. Servern kan exempelvis vara uppbyggd av Java medan klienten utnyttjar .NET (JournalDev, 2021). Figur 12 nedan illustrerar ett flödesdiagram för en web service.



Figur 12. Arkitekturen av en standard SOAP-baserad web service (Jaxenter, 2015)

WSDL eller *Web Service Definition Language* är ett XML-baserat kontrakt som beskriver hur en web service är uppbyggd och vilka operationer som gränssnittet erbjuder. Kontraktet beskriver således hur en klient bör utforma förfrågan till en webservice, *input*, samt utformandet av svaret från servicen, *output* (IBM, 2021). Figur 13 visar hur en tillgänglig web service-operation kan definieras i en WSDL-fil.

```

<definitions ...>
  ...
  <portType name="EmployeeService">
    <operation name="getEmployee">
      <input
        wsam:Action="http://jaxws.baeldung.com/EmployeeService/getEmployeeRequest"
        message="tns:getEmployee" />
      <output
        wsam:Action="http://jaxws.baeldung.com/EmployeeService/getEmployeeResponse"
        message="tns:getEmployeeResponse" />
      <fault message="tns:EmployeeNotFound" name="EmployeeNotFound"
        wsam:Action="http://jaxws.baeldung.com/EmployeeService/getEmployee/Fault/EmployeeNotFound" />
    </operation>
    ...
  </portType>
  ...
</definitions>
  
```

Figur 13. Beskrivning av operation "getEmployee" i web service "EmployeeService" (Baeldung, 2020c).

Det tredje och sista konceptet är *JAX-WS* (*Jakarta XML Web Service*). Detta är teknologin som används för att binda ihop de två övriga koncepten till en konkret web-service i Java-kod. Teknologin, eller API:et, hjälper till att definiera vilken Java-metod som reflekterar WSDL-operationen som SOAP-meddelandet anropar. API:et översätter även

SOAP-meddelandet till korrekta inparametrar i Java-metoden samt omvandlar returvärdet från Java metoden till förväntat SOAP-svar med hjälp av WSDL-kontraktet (Oracle, 2013c). Figur 14 exemplifierar hur en web service *endpoint* i Java kan se ut.

```
@WebService
public interface EmployeeService {
    @WebMethod
    Employee getEmployee(int id);

    @WebMethod
    Employee updateEmployee(int id, String name);

    @WebMethod
    boolean deleteEmployee(int id);

    @WebMethod
    Employee addEmployee(int id, String name);

    // ...
}
```

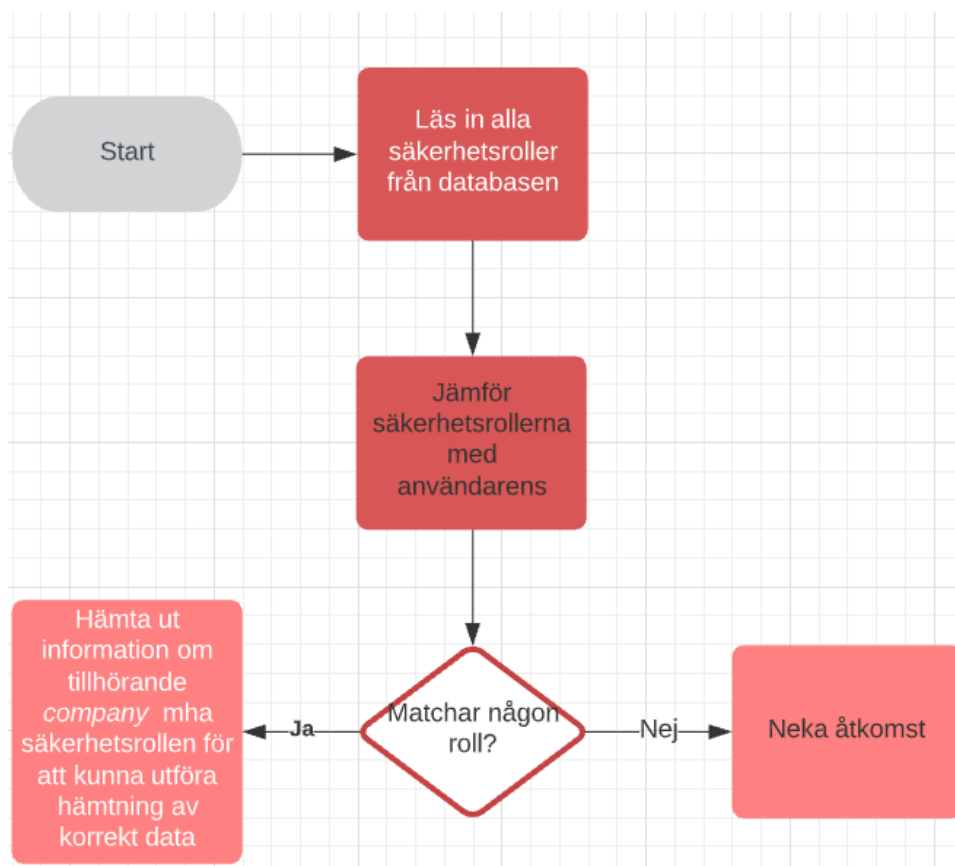
Figur 14. En deklarerad web service endpoint som använder JAX-WS annotationer (Baledung, 2020c)

3.5.1 Validering av säkerhetsroll

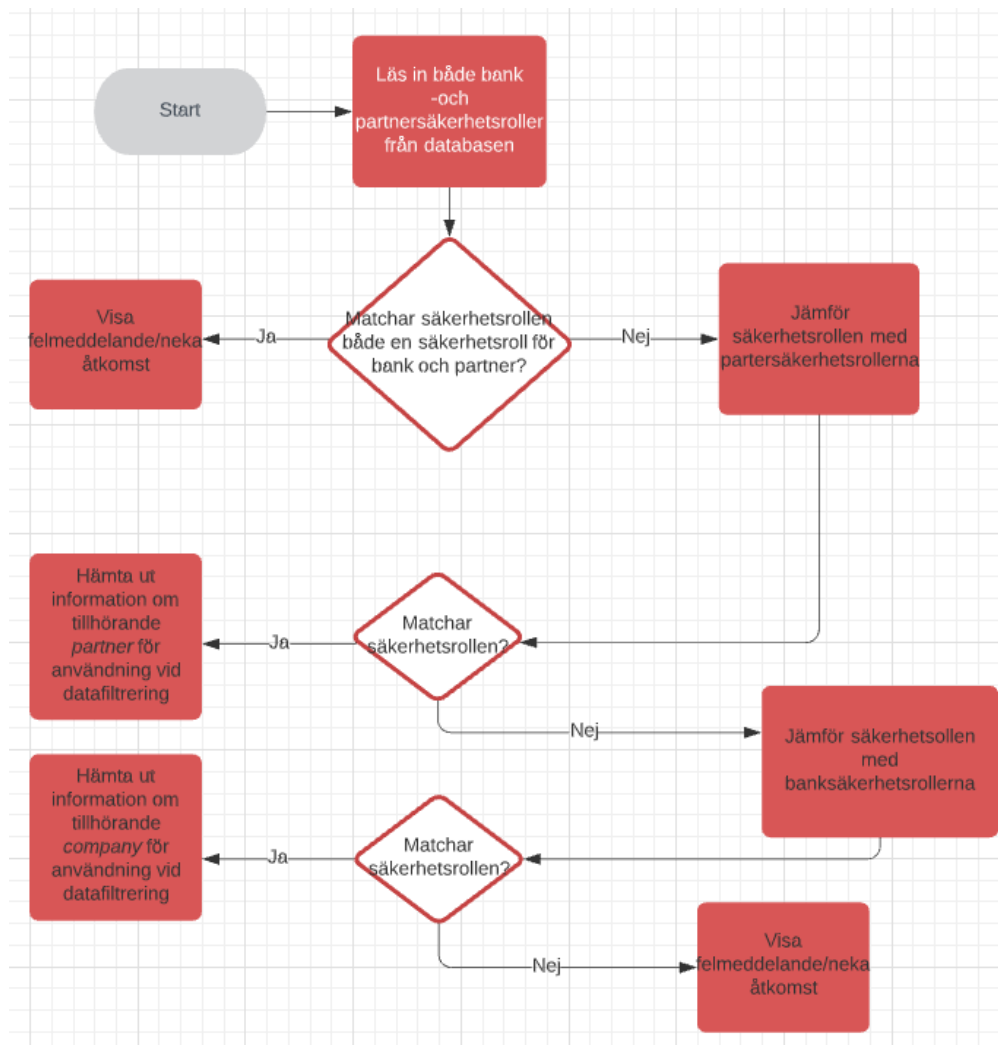
Ett första steg i filtreringen av data var att implementera ett sätt att validera säkerhetsrollen som en partner-användare hade i samband med att ett anrop till någon av web-servicarna gjordes. Modulen som innehåller alla web service klasser, utnyttjade en redan implementerad validering för banker. Alla externa användare av API:erna har en *Spring Security Role*⁸ som refererar till den bank som de försöker hämta ut information om. Detta kan röra sig exempelvis om hämtning av information om konton, transaktioner eller kunder. Figur 15 (s. 26) visar ett flödesdiagram hur valideringen av säkerhetsroller såg ut innan partner-valideringen implementerades.

⁸ *Spring Security* är ett ramverk för autentisering och åtkomstkontroll som har blivit standard för säkerhetsaspekter i Spring-applikationer. För att lättare kunna hantera åtkomst till specifika tjänster har ramverket ett utvecklat stöd för säkerhetsroller (Pivotal, 2021c).

För att validera partner-användare behövde nya säkerhetsroller läggas till i databasen. Utöver detta behövde jag även utveckla en ny logik med anledning av att först kontrollera ifall det var en partner-användare eller en bank-användare som försökte använda web-servicen. Förutom denna kontroll var det också nödvändigt att kontrollera att en användare inte hade både en bank-och en partnersäkerhetsroll tilldelad till sig eftersom det skulle leda till en felaktig filtrering av data. För att enklare beskriva det nya flödet är figur 16 (s.27) till förfogande.



Figur 15. Flödesdiagram av ursprunglig validering av säkerhetsroller.



Figur 16. Flödesdiagram över det nya flödet för validering av säkerhetsroller.

3.5.2 Inkapsling av säkerhetsinformationen

Då valideringen för användarens säkerhetsroll hade gjorts var det önskvärt att samla den nödvändiga informationen om den banken eller partnern som användaren tillhör. Orsaken till detta är för att underlätta filtreringen och kontrollen över datat. Genom att samla all information i ett och samma objekt, som endast skapades en gång då en användare anropade en web service metod, skulle detta leda till mer välstrukturerad kod samt bättre prestanda. Till detta syfte introducerades klassen *UserContext* samt det nya gränssnittet *UserContextService*. Klassen skapades som en *immutable entity* eftersom objektet bara skapades en gång och informationen som objektet innehöll endast var till för att hämtas. Gränssnittet

implementerades med en metod, *getUserContext*. Figur 17 visar implementeringen av klassen *UserContext*.

```
@Value
@Builder
public class UserContext {
    private Long companyId;
    private String partnerId;
    private String tenant;
    private boolean isPartnerUser;
    private Long userId;
    private String userEmail;
}
```

Figur 17. Klassen *UserContext*.

3.5.3 Filtreringsflödet

Det första hindret jag stötte på när ändringarna för filtreringsflödet skulle göras i de nästan 20 till antalet web service-klasserna var det osammanhängande sättet de alla var implementerade på. I princip alla endpoint metoder i web servicerna hade ett unikt sätt att behandla input och output på. En förklaring till denna variation var att en del var klasserna var gamla och implementationen således var en aning föråldrad. En annan, och kanske den mest betydande faktorn, var att de olika web service-klasserna i kortsystemet behandlade hämtningen, uppdaterandet samt skapandet av väldigt olika typer av data.

Filtreringen var däremot tacksam i de web-servicerna där hämtningen bestod av entiteter som hade ett partner-id på sig från början. Här behövde jag inte ändra på logiken gällande hämtningen av entiteterna från databasen och filtreringen kunde således ske i efterhand. Detta rörde sig exempelvis om hämtningen av informationen gällande konton, kort- och kontotransaktioner samt auktoriseringar. I dessa *web service endpoints* introducerade jag en ny hjälpklass, *PartnerAuthorizationValidator*. Hjälpklassens huvudsakliga roll var att jämföra partner-id:et från *UserContext*-objektet med partner-id:et på de objekt som användaren försökte hämta och filtrera ut de objekt där detta villkor inte var sant. För att kunna anropa en och samma filtreringsmetod för de objekt som denna “enkla” filtreringslogik lämpade sig för, skapade jag ett gränssnitt som de alla implementerade; *PartnerResource* med metoden *getPartnerId*. Filtreringsmetoden illustreras i figur 18.

```

public <T> void filterEntities(
    final String partnerId,
    final List<T> entitiesToFilter,
    final Function<T, String> partnerIdExtractor
) {
    entitiesToFilter.removeIf(entity ->
        !partnerId.equals(partnerIdExtractor.apply(entity)));
}

```

Figur 18. Filtreringsmetoden i *PartnerAuthorizationValidator* som utnyttjar den inbyggda java funktionen *removeIf* i *java.util*.

Filtreringen i de övriga web-servicarna utfördes på varierande sätt. I en del fall var det fördelaktigt att använda något av de attribut som användaren skickade som input till metoden. Dessa attribut var oftast primärnycklar eller unika id:en som kunde användas för att hämta ut sammankopplade entiteter med ett partner-id för att sedan jämföra detta partner-id med det tillhörande i *UserContext*-objektet och således kontrollera ifall användaren var behörig för de uppgifterna som försöktes hämta. Ett annat tillvägagångssätt som också utnyttjades var att “skicka med” partner-id:et från *UserContext* till databaslagret för att utföra själva filtreringen då databasoperationerna gjordes. Dessa databasoperationer utnyttjade ofta *join-operationer* för att samla ihop data från olika tabeller till outputen för web servicen i fråga. I dessa scenarion kunde således SQL-påståenden sköta filtreringen genom att endast hämta ut information om de insättningar där ett korrekt partner-id fanns.

De ovannämnda lösningarna för datafiltreringen var de mest centrala för den sista delen i mitt arbete över partnerskapshantering, utöver ett fåtal specialfall som jag har valt att inte redogöra. Detta eftersom de centrala lösningarna som jag har redogjort för belyser filtreringsflödet på ett mer lättöverskådligt sätt. Det fanns även en del web service-klasser som partner-användare inte skulle komma att använda. I dessa klasser implementerade jag bara det nya *UserContext*-konceptet för att få en enhetlig kod i modulen.

4. SLUTSATSER

Detta examensarbete resulterade i en ny abstraktion för Partner i Crosskey Banking Solutions kortsystem. Den nya abstraktionen, eller entiteten, har skapat bättre förutsättningar för att lättare kunna lägga till nya partner med tillhörande produkter, samt gett en automatiserad lösning för etikettering av data som tillhör dessa partner. Utöver detta har också ett tillvägagångssätt utvecklats för att säkerställa att korrekt data hämtas och visas för de partner-användare som utnyttjar kortsystemets web-services.

Projektet har varit tidskrävande i och med att ändringar har varit nödvändiga i en utbredd del av kodbasen. Dock har detta erbjudit möjligheten att få undersöka stora delar av kortsystemet och således gett mig en ökad förståelse av dataflöden och designval. Jag har även fått chansen att bekanta mig vid nya tekniker och ramverk, exempelvis *SOAP* web-services. De diskussioner jag har haft med min handledare, tillika arkitekt på kortavdelningen, har varit betydelsefulla för arbetets fortskridande, samtidigt som de har gett mig en bättre förståelse för koddesign och kortsystemet som helhet.

Inför de framtida ändringarna av kortsystemets nya API:er kommer en fortsatt hantering av partner vara nödvändig. Det som jag har redogjort för i detta arbete kan förhoppningsvis bidra med en mall eller en grund för den partnerskaphantering som kommer att vara aktuell i samband med att de framtida API:erna ska realiseras.

KÄLLOR

Atlassian. (u.å.). Stories, epics and initiatives. Hämtad 21.08.2021 från:
<https://www.atlassian.com/agile/project-management/epics-stories-themes>

Arthur, J., & Azadegan S. (2005). Spring Framework for rapid open source J2EE Web Application Development: A case study. *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Network*.
<https://doi-org.ha.idm.oclc.org/10.1109/SNPD-SAWN.2005.74>

Baeldung. (2021a). Database Migrations with Flyway. Hämtad 21.08.2021 från:
<https://www.baeldung.com/database-migrations-with-flyway>

Baeldung. (2021b). The DTO Pattern. Hämtad 05.09.2021 från:
<https://www.baeldung.com/java-dto-pattern>

Baeldung. (2020c). Introduction to JAX-WS. Hämtad 11.09.2021 från:
<https://www.baeldung.com/jax-ws>

Crosskey Banking Solutions. (2015). About Crosskey. Hämtad 22.08.2021 från:
<https://www.crosskey.fi/our-story/>

Flyway. (2020a). Overview. Hämtad 18.08.2021 från:
<https://flywaydb.org/documentation/>

Flyway. (2020b). How Flyway works. Hämtad 21.08.2021 från:
<https://flywaydb.org/documentation/getstarted/how>

Ginanjar, A., & Hendayun, M. (2019). Spring Framework Reliability Investigation Against Database Bridging Layer Using Java Platform. *Procedia Computer Science*, 161, 1036-1045. <https://doi-org.ha.idm.oclc.org/10.1016/j.procs.2019.11.214>

Gradle Inc. (2021a). What is Gradle? Hämtad 22.08.2021 från: https://docs.gradle.org/current/userguide/what_is_gradle.html

Gradle Inc. (2021b). Dependency management in Gradle. Hämtad 22.08.2021 från: https://docs.gradle.org/current/userguide/core_dependency_management.html

IBM. (2021). What is WSDL?. Hämtad 11.09.2021 från: <https://www.ibm.com/docs/en/app-connect/11.0.0?topic=services-what-is-wsdl>

Jaxenter. (2015). Creating SOAP Web Services using JAX-WS. Hämtad 11.09.2021 från: <https://jaxenter.com/creating-soap-web-services-using-jax-ws-117689.html>

Jenkins. (u.å). What is Jenkins Pipeline. Hämtad 04.09.2021 från: <https://www.jenkins.io/doc/book/pipeline/>

JournalDev. (2021). JAX-WS Tutorial. Hämtad 11.09.2021 från: <https://www.journaldev.com/9123/jax-ws-tutorial>

Kagan, J. (2020). Financial Technology - Fintech. *Investopedia*. Hämtad 02.03.2021 från: <https://www.investopedia.com/terms/f/fintech.asp>

Liferay. (2019). Introduction to Portlets. Hämtad 04.09.2021 från: <https://help.liferay.com/hc/en-us/articles/360018159431-Introduction-to-Portlets>

MyBatis. (2020a). Introduction. Hämtad 02.03.2021 från: <https://mybatis.org/mybatis-3/>

MyBatis. (2020b). Mapper XML Files. Hämtad 02.03.2021 från:

<https://mybatis.org/mybatis-3/sqlmap-xml.html>

Oracle. (2020a). Lesson: JDBC Introduction. Hämtad 04.03.2021 från:

<https://docs.oracle.com/javase/tutorial/jdbc/overview/index.html>

Oracle. (2013b). The Java EE 6 Tutorial, Entities. Hämtad 21.08.2021 från:

<https://docs.oracle.com/javaee/6/tutorial/doc/bnbqa.html>

Oracle. (2013c). The Java EE 6 Tutorial, Building Web Services with JAX-WS. Hämtad 11.09.2021 från:

<https://docs.oracle.com/javaee/6/tutorial/doc/bnayl.html>

Pivotal. (2021a). Spring Framework Overview. Hämtad 10.03.2021 från:

<https://docs.spring.io/spring-framework/docs/5.3.5-SNAPSHOT/reference/html/overview.html#overview-history>

Pivotal. (2021b). Core Technologies. Hämtad 10.03.2021 från:

<https://docs.spring.io/spring-framework/docs/5.3.5-SNAPSHOT/reference/html/core.html#beans>

Pivotal. (2021c). Authorization, Servlet Applications. Hämtad 15.09.2021 från:

<https://docs.spring.io/spring-security/site/docs/5.2.11.RELEASE/reference/html/authorization.html>

Project Lombok. (2021). Project Lombok. Hämtad 01.09.2021 från:

<https://projectlombok.org/>

Shiyong, X., Tianxiang, D., Rongzheng, Z., & Rongsen, W. (2020). Research On Mybatis Mapper Model Based On SQL Template. *2020 International Conference on Computer Engineering and Application (ICCEA)*, 502–505.

<https://doi-org.ha.idm.oclc.org/10.1109/ICCEA50009.2020.00112>