Examensarbete, Högskolan på Åland, Utbildningsprogrammet för Informationsteknik

REST-API FÖR KOMMUNIKATION MED BANKID

Joakim Bergström

2023:15

Datum för godkännande: 10.5.2023
Handledare: Joakim Isaksson

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Joakim Bergström
Arbetets namn:	REST-API för kommunikation med BankID
Handledare:	Joakim Isaksson
Uppdragsgivare:	Extern uppdragsgivare

Abstrakt

Syftet med detta arbete är att modernisera en modul för BankID. Den nuvarande integrerade implementationen bryts ut till en självstående applikation med vilken det är möjligt att kommunicera med hjälp av ett REST-API. Denna applikation tar sedan hand om kommunikationen mot BankID.

Applikationen är skriven i Java och använder sig av ramverket Spring. REST-API:et genereras med hjälp av OpenAPI.

Resultatet av moderniseringen är en bättre säkerhet och en bättre struktur för framtida utveckling och testning av implementationen.

Nyckelord (sökord)

Spring, REST, BankID, OpenAPI

Högskolans serienummer:	ISSN:	Språk:	Sidantal:
2023:15	1458-1531	Svenska	33 sidor

Inlämningsdatum:	Presentationsdatum:	Datum för godkännande:
19.2.2023	7.3.2023	10.5.2023

DEGREE THESIS
Åland University of Applied Sciences

Degree Programme:	Information Technology
Author:	Joakim Bergström
Title:	REST API for communication towards BankID
Academic Supervisor:	Joakim Isaksson
Commissioned by:	External employer

Abstract

The purpose of this work is to modernize a module for BankID. The current integrated implementation is extracted into an independent application, which it is possible to communicate with using a REST API. This application then takes care of the communication with BankID.

The application is written in Java and uses the Spring framework. The REST API is generated using OpenAPI.

The result of the modernization is better security and a better structure for future development and testing of the implementation.

Keywords
Spring, REST, BankID, OpenAPI

Serial number:	**ISSN:**	**Language:**	**Number of pages:**
2023:15	1458-1531	Swedish	33 pages

Handed in:	**Date of presentation:**	**Approved:**
19.2.2023	7.3.2023	10.5.2023

INNEHÅLLSFÖRTECKNING

1. INTRODUKTION

1.1 Syfte

Syftet med detta examensarbete är att beskriva utvecklingsprocessen för att skapa ett internt REST-API-lager som hanterar all kommunikation och integration med BankID. I detta examensarbete kommer jag att gå igenom utvecklingsprocessen och de verktyg som används. När applikationen är klar kommer den att bli en fristående applikation som sköter integrationen mot BankID som sedan används av min uppdragsgivares produkter som behöver BankID för sina autentiseringsbehov.

Detta arbete handlar om att skapa anslutningspunkter för att kunna använda sig av BankIDs funktionalitet för autentisering, signering och statuskontroll. Statuskontrollen gör det möjligt att se i vilken fas en autentisering eller signering befinner sig i. Uppbyggnaden av dessa anslutningspunkter kommer att ske med design först-principen. Detta betyder att det först skrivs ett API-kontrakt, detta sker med OpenAPI-standarden. Genom detta kontrakt kan man använda sig av en OpenAPI-generator för att producera en grundstruktur för servern och klientbibliotek för att kunna integrera sig mot detta API.

1.2 Metod

Projektet började med ett möte med min handledare och med på mötet var också en utvecklare från mitt team som hade skapat själva kraven på vad som skall göras i detta arbete. På detta möte gick vi igenom hur det nuvarande systemet är uppbyggt och vad som skall genomföras vid skapandet av detta REST-API-lager. Nästa steg var att gå igenom den nuvarande implementationen för att se hur den var uppbyggd för att få en bättre överblick av vad den gjorde i praktiken. Detta behövdes göras för att se om det finns delar som kan återanvändas i den nya applikationen.

Jag har haft möjligheten att jobba som trainee ett tag redan innan detta projekt, så jag förstår hur vi i teamet arbetar och hur koden är standardiserad för företaget. Utvecklingen av detta

arbete utförs med ett agilt arbetssätt. Detta gör att projektet är uppdelat i många små delar, så kallade “stories”. När en del är utförd skall den granskas och godkännas av någon i mitt team, som kommer att ge feedback och förslag på förbättringar. Baserat på denna feedback kommer jag sedan att förfina koden och lägga upp den för granskning på nytt. När en story har blivit godkänd kommer koden att bli integrerad med testmiljön.

Applikationen kommer att skrivas i Java och använda ramverket Spring med Spring WebFluxs WebClient för att göra REST-anrop.

1.3 Avgränsningar

Jag kommer inte att beskriva hur utvecklingsmiljön ser ut och hur den är konfigurerad. Jag kommer inte heller att gå igenom hur testning av detta REST-API hanteras. Detta projekt består endast av kod för backend. Detta betyder att det inte kommer att skapas något grafiskt användargränssnitt. Koden som kommer att visas är en modifierad version av den verkliga koden.

2. NUVARANDE IMPLEMENTATION

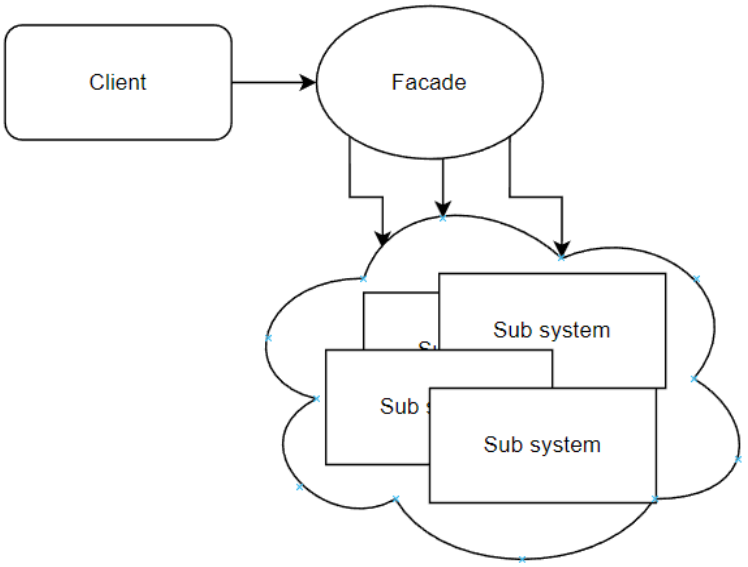
2.1 Bakgrund

Den nuvarande implementationen hämtar hela BankID-modulen som ett beroende via Gradle och sedan anropar klienten en fasad för att utföra t.ex en autentisering. Detta leder till att BankID-modulen inte fungerar som ett självstående projekt utan den måste alltid hämtas som ett bibliotek till projektet. Projekt tillhör en del av ett större moderniseringsprojekt som syftar till att skapa fristående moduler och därigenom förenklas testningen av dessa. Den nuvarande implementationen använder sig av designmönstret Fasad.

2.2 Designmönstret Fasad

Designmönstret Fasad är ett mjukvarudesignmönster som ger ett förenklat gränssnitt till ett komplext system. Det används för att tillhandahålla ett enhetligt gränssnitt till en uppsättning gränssnitt i ett delsystem, vilket gör att klienter kan interagera med delsystemet genom fasaden i stället för att interagera med delsystemet direkt (*Facade*, 2022). Figur 1 ger en överblick av detta.

För att implementera designmönstret definieras vanligtvis en fasadklass som erbjuder ett gränssnitt till det komplexa systemet. Fasadklassen delegerar förfrågningar från klienter till lämpliga subsystem objekt och kan även utföra ytterligare uppgifter såsom loggning, felhantering eller validering av indata.



Figur 1. Strukturen för designmönstret Fasad

3. REST

REST (Representational State Transfer) är en arkitekturstil för att bygga webb-API:er. Ett webb-API är ett gränssnitt som definierar hur webbaserade applikationer kommunicerar med varandra. API:er som bygger på REST-arkitekturen definieras som RESTful API:s.

REST API:er är designade för att vara lättviktiga och lätta att använda, och de är vanligtvis baserade på HTTP-protokollet och använder vanliga HTTP-metoder och statuskoder för att förmedla information om statusen för en begäran. De använder också HTTP-rubriker och svarskoder för att ge ytterligare information om begäran och resursen som nås (*What Is Restful API*, u.å.).

3.1 Principer för arkitekturen

För att använda REST-arkitekturen finns det ett antal principer som skall följas för att man skall kunna kalla sitt API ett RESTful API. Dessa sex principer är följande (*What Is REST*, 2018):

1. **Enhetligt gränssnitt**
   Det skall finnas ett enhetligt gränssnitt mot servern som gör att alla applikationer som kommunicerar med den givna servern gör det på samma sätt. Detta uppnås genom följande begränsningar: Varje resurs som är involverad i interaktionen mellan servern och klienten skall vara unikt identifierad i gränssnittet. Servern tillhandahåller enhetliga representationer av resurser och klienten använder dessa representationer för att ändra tillståndet på servern.

2. **Klient och server**
   Genom att tilldela klienten och servern olika ansvarsområden kan komponenterna utvecklas oberoende av varandra. Denna uppdelning förbättrar portabiliteten och skalbarheten både för klienten och servern.

3. **Tillståndslös**

   Kommunikationen mot servern skall vara tillståndslös. Detta betyder att själva begäran från klienten till servern måste innehålla all information som behövs för att behandla begäran. På grund av detta behöver servern inte utnyttja tidigare lagrad information, vilket ger servern bättre tillgänglighet eftersom den inte behöver upprätthålla sessionen.

4. **Tillfällig lagring**

   Svaren från serven skall inkludera om de kan bli mellanlagrade eller inte. Detta innebär att klienten som skickat en förfrågan kan spara responsen i sitt mellanlager, vilket möjliggör att klienten kan använda sin mellanlagrade data istället för att skicka en ny förfrågan till servern. För servern innebär detta att den inte behöver svara på onödiga förfrågningar och detta förbättrar serverns prestanda och tillgänglighet.

5. **Lagerarkitektur**

   Applikationsarkitektur kan vara ihopsatt av flera olika lager i en hierarki där varje komponent inte kan se mera än det direkt underliggande lagret som den integreras med.

6. **Kod på begäran (Valbart)**

   Denna princip är valfri och innebär att klientens funktionalitet utökas genom att med en förfrågan kunna hämta körbar kod från servern.

3.2 Kommunikation mellan klient och server

I REST-arkitekturen skickar klienter förfrågningar för att hämta eller modifiera resurser och servern svarar på dessa förfrågningar. En förfrågan består av en HTTP-metod som definierar vilken typ av åtgärd som skall göras, en sökväg till resursen (URL), en HTTP-rubrik som innehåller metadata och en valfri meddelandetext som innehåller data (*What Is REST?*, u.å.).

De grundläggande HTTP-metoderna för en klientförfrågning är:

- GET - Hämtning av en resurs

- POST - Skapande av en resurs
- PUT - Uppdatering av en resurs
- DELETE - Radering av en resurs

När servern tar emot en förfrågan från klienten försöker den uppfylla förfrågan och svarar sedan med en HTTP-svarskod och i vissa fall också med en meddelandetext som innehåller ett svar på förfrågan.

De vanligaste svarskoderna är följande:

- 200 (OK) - HTTP-förfrågan var lyckad
- 201 (SKAPAD) - Skapande av resursen var lyckad
- 400 (DÅLIG FÖRFRÅGAN) - Förfrågningen kan inte behandlas för att indatan var felaktig
- 404 (EJ HITTAD) - Resursen kunde inte hittas
- 500 (INTERNT SERVERFEL) - Ett allmänt svar för ett oväntat fel

4. RAMVERK OCH VERKTYG

4.1 Ramverket Spring

Ramverket Spring är ett applikationsramverk med öppen källkod för Java som tillhandahåller en omfattande uppsättning funktioner för att bygga applikationer. Ramverket är fokuserat på att tillhandahålla kärnfunktioner för att utveckla företags Java-applikationer, inklusive stöd för beroendeinjektion (*eng. Dependency Injection, DI*), dataåtkomst och webbapplikationer (*Spring Framework*, 2019).

4.1.1 Beroendeinjektion

Beroendeinjektion är ett mjukvarudesignmönster som gör att ett objekt kan förses med dess beroenden. I ramverket Spring används beroendeinjektion för att hantera beroenden mellan objekt och för att tillhandahålla ett sätt för objekt att få de beroenden de behöver på ett flexibelt sätt, utan att skapa hårdkodade kopplingar till andra objekt. Med beroendeinjektion fås bättre omstruktureringsmöjligheter och bättre strukturerad kod. Objektet letar inte upp sina beroenden och känner inte till platsen eller klassen för beroenden. Som ett resultat blir klasser lättare att testa, särskilt när beroenden är gränssnitt eller abstrakta basklasser, som gör det möjligt att skapa objekt som simulerar beteendet av den riktiga implementationen i enhetstester. Detta kallas för en mock implementation (*Core Technologies*, u.å.-a).

4.1.2 Spring Beans

I ramverket Spring är en böna ett Java-objekt som hanteras av Spring IoC-behållaren (Inversion of Control). Spring IoC-behållaren är ansvarig för att skapa och hantera bönornas livscykel, inklusive att instansiera dem, injicera deras beroenden och hantera deras livscykel (t.ex. att förstöra dem när de inte längre behövs) (*Core Technologies*, u.å.-b).

För att definiera en böna i Spring skapas vanligtvis en Java-klass som representerar bönan och kommenterar den med annotationen @Bean. Se figur 2. Det är också möjligt att använda andra annotationer, som @Configuration för att indikera bönans roll i applikationen (*Core Technologies*, u.å.-c).

```java
@Configuration
public class DefaultConfigurationService {

    @Bean
    public ExampleService exampleService() {
        return new ExampleService();
    }

    @Bean
    public ProductService productService(final ExampleService exampleService) {
        return new ProductService(exampleService);
    }
}
```

Figur 2. Exempel på Java-konfiguration av bönor

När en böna har definierats kan Spring IoC-behållaren användas för att hantera den. Du kan till exempel använda behållaren för att injicera beroenden i bönan, eller för att hämta bönan från behållaren och använda den i din applikation.

Bönor kan vara beroende av andra bönor för att fungera korrekt. När du definierar en böna kan du specificera dess beroenden genom konstruktorinjektion, vilket innebär att IoC-behållaren ser till att rätt bönor finns tillgängliga när en instans av klassen blir skapad. Ett exempel på detta visas i figur 3. Om klassen innehåller flera konstruktorer måste den huvudsakliga konstruktorn vara markerad med @Autowired-annotationen så att IoC-behållaren vet vilken konstruktor som den skall använda när den instansierar klassen. Spring IoC-behållaren kommer sedan att ta hand om att injicera beroenden i bönan när den skapas (*Core Technologies*, u.å.-d).

```java
@Service
public class DefaultInventoryService implements InventoryService {

    private ProductService productService;

    public DefaultInventoryService(final ProductService productService) {
        this.productService = productService;
    }
}
```

Figur 3. Konstruktorinjektion med ProductService-bönan från figur 2

4.1.3 Spring WebClient

WebClient är en klient för att göra HTTP-förfrågningar och har stöd för icke-blockerande anrop och är byggd på Reactor Netty-biblioteken. WebClient introducerades i Spring 5 och ersätter den befintliga RestTemplate-klienten. Med WebClient är det möjligt att göra synkrona eller asynkrona HTTP-förfrågningar som kan integreras direkt i din befintliga Spring-konfiguration (*Web on Reactive Stack*, u.å.).

4.2 Spring Boot

Spring Boot är ett Java-baserat verktyg som används för att bygga fristående Java-applikationer. Spring Boot ger ett antal fördelar för utvecklare som bygger applikationer med ramverket Spring. Dessa inkluderar stöd för inbyggda servrar, hantering av konfigurationsegenskaper, samt ett antal verktyg och funktioner som gör det enkelt att bygga, testa och distribuera Spring-baserade applikationer. (*What Is Java Spring Boot?*, u.å.).

4.3 Swagger

Swagger är ett ramverk med öppen källkod för att designa, bygga och dokumentera API:er. Den tillhandahåller en uppsättning verktyg och bibliotek för att designa och dokumentera API:er, samt ett användargränssnitt (UI) för att utforska och interagera med API:er. Swagger är baserad på OpenAPI-specifikationen (OAS), ett standardsätt för att beskriva funktionerna och funktionaliteten för ett API (*About Swagger Specification*, u.å.).

Swagger ger ett antal fördelar för API-utvecklare och användare. För API-utvecklare tillhandahåller den en uppsättning verktyg och bibliotek för att designa och dokumentera API:er på ett konsekvent och standardiserat sätt. Det tillhandahåller också ett användargränssnitt för att testa och utforska API:er, vilket kan vara användbart för felsökning och utveckling av API:er. Swagger kan också generera klientbibliotek för ett API för många olika programmeringsspråk (*Documenting Your Existing APIs: API Documentation Made Easy with OpenAPI & Swagger*, u.å.).

4.3.1 OpenAPI-specifikation

OpenAPI-specifikationen är ett standardsätt för att beskriva funktionerna och funktionaliteten hos ett API. Det är ett maskinläsbart format som specificerar endpoints, operationer och parametrar för ett API, såväl som datastrukturerna som används av API:et. OpenAPI-specifikationen definieras vanligtvis med OpenAPI-formatet, som är baserat på JSON Schema-standarden och använder JSON eller YAML för att definiera API:er (*OpenAPI Specification*, u.å.).

OpenAPI-specifikationen är avsedd att vara ett språkoberoende, standardiserat sätt att beskriva API:er. Den kan användas för att definiera API:er för så gott som alla programmeringsspråk eller ramverk, och används ofta i utvecklingen av RESTful API:er. OpenAPI-specifikationen kan användas för att generera kod, dokumentation och andra artefakter relaterade till ett API, vilket gör det lättare för utvecklare att bygga och konsumera API:er. Det är en öppen standard och specifikationen underhålls av OpenAPI Initiative (*Documenting Your Existing APIs: API Documentation Made Easy with OpenAPI & Swagger*, u.å.).

Figur 4 - 6 illustrerar hur en struktur kan se ut för en OpenAPI-specifikation skriven i YAML.

```yaml
openapi: 3.0.0
info:
  title: OpenAPI specification example
  description: A basic OpenAPI specification example
  contact:
    email: example@example.com
  version: 1.0.0
servers:
  - url: https://example.com/api/v1
tags:
  - name: student
    description: Everything about students
```

Figur 4. Informationsdelen av en OpenAPI-specifikation

```yaml
paths:
  /student/{id}:
   get:
    tags:
      - student
    description: Find student by id
    operationId: getStudentById
    parameters:
      - name: id
        in: path
        description: Id of student to return
        required: true
        schema:
          type: integer
          format: int64
    responses:
      200:
        description: Successful operation
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/Student"
      400:
        description: Invalid id supplied
      404:
        description: Student not found
```

Figur 5. Sökvägsdelen där anslutningspunkterna specificeras

```yaml
components:
  schemas:
    Student:
      type: object
      properties:
        id:
          type: integer
          format: int64
          example: 123456
        firstname:
          type: string
          example: Firstname
        lastname:
          type: string
          example: Lastname
        dob:
          type: string
          format: date
          example: 01.01.2000
```

Figur 6. Komponentdelen där scheman specificeras

I tabell 1 kan man se förklaringar till de olika specifikation egenskaperna från exemplen i figurerna 4 - 6.

Tabell 1. Beskrivning av egenskaper från figurerna 4 - 6 ovan

Egenskaper	***Beskrivning***
openapi	Vilken Swagger-version som används.
info	Metadata om API:et som t.ex titel, kontaktinfo, licenser osv.
servers	Innehåller en inledande sökväg (*eng. base url*) för anslutningspunkterna.
tags	Taggar används för att gruppera olika anslutningspunkter.
paths	Innehåller information om respektive anslutningspunkt och dess operationer. Dessa läggs till sökvägen (*base url*) som är definierad i serverobjektet. I figur 4 och 5 definieras en HTTP GET-operation för anslutningspunkten https://example.com/api/v1/students/{id} där {id} kan vara ett godtycklig 64-bitars heltal.
response	Innehåller svaren från operationen. Varje svar mappas till en HTTP-statuskod. Om en lyckad operation har genomförts returneras statuskoden 200.
content	Innehåller mediatyper som förbrukas av operationen och med $ref kan man returnera ett eget objekt. I exemplet används application/json vilket betyder att responsen kommer att vara strukturerad med ett JSON-format.
components	Den plats där alla återanvändbara objekt såsom scheman finns.
schemas	Definierar in- och utdatatyper som kan vara objekt eller primitiva datatyper

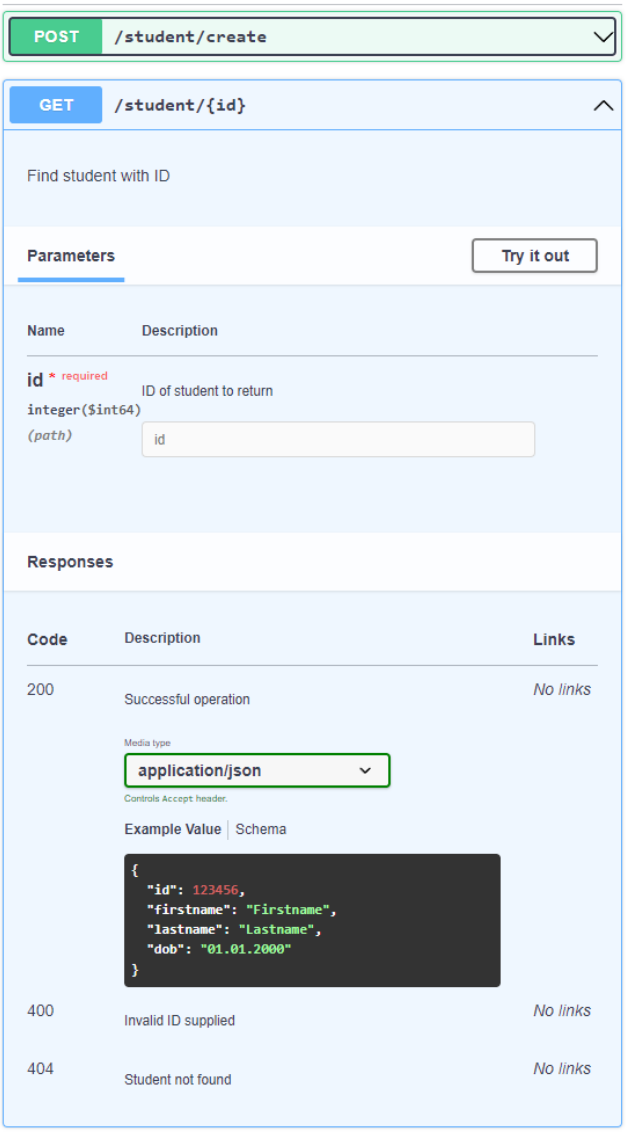
4.3.2 OpenAPI Generator

OpenAPI generator är ett verktyg med öppen källkod som kan generera kod både för klient- och serversidan samt dokumentation baserad på OAS. Detta kan vara användbart för utvecklare som bygger API:er och vill generera kod som överensstämmer med specifikationen, såväl som för utvecklare som konsumerar API:er och vill använda ett klientbibliotek som genereras från specifikationen (*OpenAPI Generator*, u.å.).

4.3.3 Swagger UI

Swagger UI är en samling HTML-, JavaScript- och CSS-resurser som dynamiskt genererar en dokumentation från ett Swagger-kompatibelt API. I figur 7 visas hur OAS-dokumentation kan se ut i Swagger UI (*Swagger UI Repository*, u.å.).

Swagger UI ger en visuell representation av ett API:s anslutningspunkter och operationer, samt möjligheten att interagera med API:et genom ett enkelt webbaserat gränssnitt. Detta kan utnyttjas av utvecklare som skapar eller använder API:er, eftersom det ger ett sätt att enkelt testa och utforska API:s funktionalitet (*REST API Documentation Tool*, u.å.).



Figur 7. Exempel på hur dokumentation ser ut i Swagger UI

4.4 BankID

BankID är den största lösningen för elektronisk identifiering (eID) i Sverige som företag använder för att verifiera sina kunders identitet online på ett säkert och bekvämt sätt. Endast personer med svenskt personsignum kan ansöka om BankID. Det används ofta för uppgifter som att signera dokument eller göra finansiella transaktioner, och kan också användas för att få tillgång till statliga tjänster online (*About BankID*, u.å.).

5. UTVECKLING AV APPLIKATIONEN

5.1 Förberedelser

Innan utvecklingen av applikationen kunde påbörjas uppdelades specifikationen i mindre bitar, genom att skapa Jira stories som beskriver uppgifterna. Jira är ett projektledningsverktyg som hjälper till att följa upp arbetsprocessen och buggspårning. Jira används också som ett verktyg för att dokumentera mina framsteg och hålla reda på vad som har blivit gjort.

Arbetet var uppdelat i tre delar: autentisering, signering och statuskontroll. Om man läser BankID-dokumentationen så kallas statuskontrolldelen där “collect”. Statuskontrolldelen används för att kolla upp i vilket skede autentisering eller signeringen befinner sig i. Varje del var också indelad i mindre delar, t.ex autentiseringsdelen bestod av skapande av OpenAPI-specifikationen, implementation av serverns logik och implementera klienterna att använda den nya integrationen.

5.2 Struktur och utvecklingsmetodik

Strukturen för detta arbete består av två projekt. Det första projektet innehåller endast OAS YAML-filen som beskriver API-strukturen. Detta projekt används sedan för att producera grundstrukturen för servern och klientbibliotek för integration mot servern. Det andra projektet är själva API-implementationen, detta projekt innehåller integrationen mot BankID och omstrukturering av in- och utdata.

För utveckling av REST API:n användes design först-principen vilket innebär att man designar API-kontraktet först genom att skriva strukturen för API:et innan man skriver någon kod alls. Detta kontrakt skrivs med OpenAPI-specifikationen (OAS). För detta arbete fanns det inget behov av att skapa ett nytt tomt projekt eftersom det redan finns en implementation mot BankID som använder sig av denna RESTful implementation, så det jag behövde göra var att skriva kod .

För att få en bra överblick av vilka anslutningspunkter, operationer och resurser (dataobjekt) API:et skulle behöva gick jag igenom den tidigare implementationen.

5.3 Gränssnitt mot BankID

I detta stycke går jag igenom några av de viktigaste parametrarna som skickas till BankID och vad de har för funktion. Dokumentationen för detta finns på BankIDs hemsida (*Interface description*, u.å.).

Alla förfrågningar mot BankID sker med HTTP-metoden POST och datan skall vara uppbyggd i JSON-format. I tabell 2 visas parametrarna som behövs till autentisering och signeringsförfrågningen.

Tabell 2. Parametrarna för autentisering och signeringsförfrågningen

Namn	***Nödvändig***	***Beskrivning***
personalNumber	Nej	Svenskt personnummer, 12 tecken långt, århundrade måste vara inkluderat. Om personnummer inte är inkluderat måste klienten startas med autoStartToken som finns med i svaret. Se tabell 4.
endUserIp	Ja	IP-adressen från enheten som används för anropet, kan vara IPv4 eller IPv6.
requirement	Nej	Innehåller kravparametrar för att beskriva hur signaturen måste skapas och verifieras. Typiska krav är t.ex att man måste använda Mobilt BankID eller att man får använda identifikation med fingeravtryck. Alla krav finns uppräknade på BankIDs hemsida.

Det finns även några tilläggsparametrar. Dessa används för att visa information i textformat till slutanvändaren om vad som håller på att göras. För signeringen måste det finnas userVisibleData som beskriver vad som skall signeras och sedan visas till slutanvändaren. För autentiseringen är denna inte en nödvändig parameter att inkludera.

Tabell 3. Tilläggsparametrar för förfrågningen till autentisering och signering

Namn	***Nödvändig***	***Beskrivning***
userVisibleData	Nej / Ja	Text som visas till användaren under en autentisering eller signering. Texten måste vara UTF-8 och base64-kodad.
userNonVisibleData	Nej	Data som inte visas till kunden. Måste vara base64-kodad.

Om förfrågning av autentisering eller signering godkänns kommer BankID att svara med ett JSON-formaterat svar. I responsen finns parametern autoStartToken som används för att skapa en start-url som sedan hämtar QR-koden som visas till slutanvändaren.

Tabell 4. Svar som returneras om en autentisering eller signering godkänts

Namn	***Beskrivning***
orderRef	Referens för denna beställning. Används för att få ut statusen i statuskontrollanropet.
autoStartToken	Används för att skapa start url:n för att hämta QR-koden.

I tabell 5 visas statuskontrollförfrågningen som endast tar emot en parameter orderRef. Detta är samma referens som finns i svaret från en godkänd autentisering eller signering.

Tabell 5. Parametrar som ingår i en statuskontrollförfrågning

Namn	***Beskrivning***
orderRef	Referensen som fås från svaret av en autentisering eller signering.

Från statuskontrollen får man ett svar som innehåller data om vad som håller på att hända i detta ögonblick.

Tabell 6. Svar som returneras för status om förfrågningen godkänts

Namn	***Beskrivning***
orderRef	Referensen man använde för förfrågningen.
status	Kan vara pending: beställningen är pågående, failed: någonting gick fel, complete: beställningen gick igenom och completionData är tillgängligt.
hintCode	Anges bara om status är pending eller failed. Används för att beskriva vad som gick fel om status = failed eller vad som håller på att göras om status = pending.
completionData	Anges bara om beställningen har blivit godkänd. Innehåller information om personen som t.ex personnummer, namn, vilken enhet man har använt BankID från och annan metadata.

Om förfrågningen misslyckas så att HTTP-koden motsvarar 4xx eller 5xx kommer BankID att returnera en respons som beskriver problemet. Se tabell 7.

Tabell 7. Svar som returneras om ett fel inträffade.

Namn	***Beskrivning***
errorCode	Kod som motsvarar ett problem i textformat.
details	Kort meddelande om felet.

5.4 Anslutningspunkter

Det behövs en anslutningspunkt per operation vilket betyder att det behövs en för autentisering, en för signering och till sist en för statuskontroll. Dessa anslutningspunkter definieras i OAS-filen. I figur 8 kan man se hur anslutningspunkten för autentisering är definierad. Alla anrop kommer att ske med HTTP-metoden POST. När man använder POST kommer dataobjektet att läggas i kroppen av anropet (eng. request body). I figur 8 ser vi att autentiseringen tar in ett objekt som skall vara strukturerat i JSON-format. För att förfrågningen skall lyckas finns det även några HTTP-rubriker som måste inkluderas. HTTP-rubrikerna innehåller metadata som används i loggningsflödet för debugging. I svaret

definieras olika HTTP-statuskoder, om förfrågningen lyckades returneras statuskoden 200. I svaret finns ett objekt som innehåller svaret från BankID, se tabell 4. Statuskoden 400 returneras om det skulle uppstå problem med kommunikationen mellan klienten och servern. Statuskoden 500 returneras om något fel uppstår på servern om t.ex BankID skulle svara med en fel respons. Dessa två felstatuskoder returnerar också ett objekt i sitt svar med information om vad som har hänt.

Alla tre anslutningspunkter, autentisering, signering och status är uppbyggda på samma sätt som i figur 8. Enda skillnaden är vilken sorts objekt som tas in av POST-anropet och vad som returneras om statuskoden är 200.

```yaml
/v1/authenticate:
  post:
    parameters:
      - name: X-Calling-Application
        in: header
        schema:
          type: String
        required: true
      - name: X-User-ID
        in: header
        schema:
          type: string
      - name: X-External-Session-ID
        in: header
        schema:
          type: String
      - name: X-Tenant
        in: header
        schema:
          $ref: "#/components/schemas/Tenant"
        required: true
    requestBody:
      required: true
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/BankIdAuthenticateRequest"
    responses:
      200:
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/BankIdAuthenticateResponse"
      400:
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/ValidationProblemDetails"
      500:
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/InternalServerErrorProblemDetails"
```

Figur 8. OAS för anslutningspunkten för autentisering

5.4.1 Autentisering

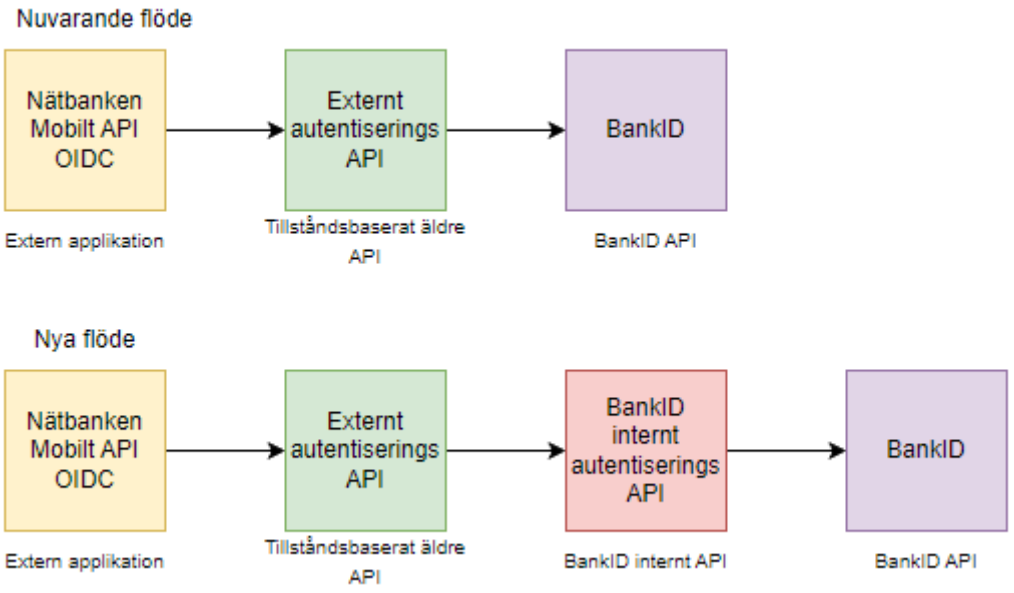
Autentiseringen är det som används när slutanvändaren, alltså kunden, öppnar inloggningssidan och trycker på en knapp för att logga in t.ex i nätbanken. Efter det så behandlas datan och skickas till BankID för att hämta QR-koden som slutanvändaren kan läsa av med sin BankID mobilapp. Se figur 9 för en överblick av det nuvarande och det nya flödet.

5.4.2 Signering

Signeringen är det som används när slutanvändaren skall godkänna en betalning eller signera ett dokument. Anropet sker när slutanvändaren trycker på en knapp för att t.ex. bekräfta en betalning. Signerings- och autentiseringsflödena ser liknande ut.

5.4.3 Statuskontroll

Anslutningspunkten för statuskontroll anropas med jämna mellanrum efter påbörjad autentisering eller signering för att få en uppdaterad status för hur beställningen fortskrider. Anropen till anslutningspunkten för statuskontroll fortsätter tills parametern "status" har blivit tilldelat värdet *failed* eller *complete*. Om beställningen misslyckas kommer kedjan att avbrytas och ett felmeddelande kommer att visas till slutanvändaren. Men om beställningen lyckas kommer klienten att få parametern i svaret med *completionData*.



Figur 9. BankID har blivit utbrutet till ett eget API

5.5 Implementation av API-strukturen

Verktyget OpenAPI generator bygger upp ett Java-gränssnitt och modellklasser baserat på förfrågningarna och resurserna som definierats i OAS-filen. Verktyget genererar ett gränssnitt för varje tagg man har skapat i sin OAS-fil. I figur 10 visas det genererade gränssnittet.

```
@javax.annotation.Generated(value = "", date = "")
@Validated
@Api(value = "", description = "")
public interface BankIdApi {

    /**
     * POST /v1/authenticate : Do authentication towards BankID
     *
     * @param xCallingApplication       (required)
     * @param xTenant                   (required)
     * @param bankIdAuthenticateRequest (required)
     * @param xUserID                   (optional)
     * @param xExternalSessionID        (optional)
     * @return OK (status code 200)
     *         or Invalid input (status code 400)
     *         or Internal Server error (status code 500)
     */
    @ApiOperation(
            value = "", nickname = "", notes = "", response = BankIdAuthenticateResponse.class, tags = {""}
    )
    @ApiResponses(value = {
            @ApiResponse(code = 200, message = "", response = BankIdAuthenticateResponse.class),
            @ApiResponse(code = 400, message = "", response = ValidationProblemDetails.class),
            @ApiResponse(code = 500, message = "", response = InternalServerErrorProblemDetails.class)
    })
    @RequestMapping(
            method = RequestMethod.POST,
            value = "v1/authenticate",
            produces = {"application/json"},
            consumes = {"application/json"}
    )
    ResponseEntity<BankIdAuthenticateResponse> authenticate(
            @ApiParam(value = "", required = true) @RequestHeader(value = "", required = true) String header1,
            @ApiParam(value = "", required = true) @RequestHeader(value = "", required = true) Tenant header2,
            @ApiParam(value = "", required = true) @Valid @RequestBody BankIdAuthenticateRequest request,
            @ApiParam(value = "", required = true) @RequestHeader(value = "", required = false) String header3,
            @ApiParam(value = "", required = true) @RequestHeader(value = "", required = false) String header4
    );
}
```

Figur 10. Genererade gränssnittet med autentiseringsmetoden

För att implementera detta gränssnitt som innehåller grundstrukturen för servern skapar man en REST kontrollklass som man markerar med annotationen @RestController. Denna klass är en förfrågningshanterare där varje metod mappas mot en anslutningspunkt som returnerar ett ResponseEntity-objekt.

ResponsEntity-objektet representerar ett HTTP-svar som inkluderar statuskod, rubriker och data. I figur 11 visas implementationen av anslutningspunkten POST /v1/authenticate som definierades i figur 8. Denna metod använder en autentiseringstjänst för att utföra en autentisering. Metoden returnerar datan med statuskoden 200.

```
@RestController
public class BankIdController implements BankIdApi {

    private final AuthenticateService authenticateService;

    public BankIdController(final AuthenticateService authenticateService) {
        this.authenticateService = authenticateService;
    }

    @Override
    public ResponseEntity<BankIdAuthenticateResponse> authenticate(final String xCallingApplication,
                                                                   final Tenant xTenant,
                                                                   final BankIdAuthenticateRequest request,
                                                                   final String xUserID,
                                                                   final String xExternalSessionID) {
        final BankIdAuthenticateResponse bankIdAuthenticateResponse = authenticateService
                .authenticate(request, xTenant);

        return ResponseEntity.ok(bankIdAuthenticateResponse);
    }
}
```

Figur 11. Klass som implementerar det genererade gränssnittet

Om något fel skulle uppstå under processen så att ett svarsobjekt inte kan genereras kommer detta att kastas som ett *BankIdWebServiceException* på servern och detta tas upp av Springs undantagshanterare. Detta kan t.ex inträffa om BankID returnerar en fel respons. I OAS-filen i figur 8 visas att om statuskoden är 500 skall ett svar med ett objekt *InternalServerErrorProblemDetails* returneras. I figur 12 kan vi se hur detta objekt byggs upp.

```
@ExceptionHandler(BankIdWebServiceException.class)
protected ResponseEntity<Object> handleBankIdWebserviceException(final BankIdWebServiceException ex) {
    return buildResponseEntity(InternalServerErrorProblemDetails.builder()
            .instance(URI.create("uuid:" + UUID.randomUUID()))
            .timestamp(Instant.now())
            .title(ex.getErrorCode())
            .details(ex.getDetails())
            .status(500)
            .build());
}
```

Figur 12. Metod för att bygga en ResponsEntity med statuskod 500

5.6 Klientbibliotek

Syftet med klientbiblioteket är att de moduler som konsumerar en REST-tjänst inte behöver ha någon konfiguration för kommunikationen mot tjänsten. Det är biblioteket som är ansvarigt för att veta vilken adress man skall ansluta till. Detta betyder att alla applikationer som använder sig av detta bibliotek blir en klient. Genom att använda sig av ett klientbibliotek undviks även potentiella framtida synkroniseringsproblem.

5.6.1 Användning av klientbibliotek

Om man har använt OpenAPI-specifikationen betyder det att man kan använda sig av OpenAPIs kodgenerator för att generera ett klientbibliotek. Det kommer att generera en klass per tagg som har specificerats i OAS-filen. I figur 13 illustrerar hur ett genererat klientbibliotek kan se ut. Den genererade klassen är *bankIdApi* som innehåller våra anslutningspunkter som vi kan gå emot. Denna funktion tar in alla parametrar som var specificerade i OAS-filen.

```
@javax.annotation.Generated(value = "", date = "")
public class BankIdApi {

    private ApiClient apiClient; // <-- Also a genereted class

    public BankIdApi(ApiClient apiClient) {
        this.apiClient = apiClient;
    }

    private WebClient.ResponseSpec authenticateRequestCreaton(String header1,
                                                              Tenant header2,
                                                              BankIdAuthenticateRequest request,
                                                              String header3,
                                                              String header4) throws WebClientResponseException {
        if (/* Verify that all required parameters are set */) {
            throw new WebClientResponseException(/* Error Stuff */);
        }

        /* Creating maps for header, cookies, forms, parametes etc */

        return apiClient.invokeAPI( path: "v1/authenticate", HttpMethod.POST /* Insertion of the maps */);
    }

    2 related problems
    public Mono<BankIdAuthenticateResponse> authenticate(String header1,
                                                         Tenant header2,
                                                         BankIdAuthenticateRequest request,
                                                         String header3,
                                                         String header4) throws WebClientResponseException {
        return authenticateRequestCreaton(header1, header2, request, header3, header4)
                .bodyToMono(new ParameterizedTypeReference<BankIdAuthenticateResponse>() {});
    }
}
```

Figur 13. Genererat klientbibliotek (mycket kod bortklippt)

I figur 14 visas ett exempel på hur det kan se ut att anropa anslutningspunkten för autentiseringen. Springs *webClient* använder sig av asynkrona anrop för att hämta responsdata. Därför används blockmetoden i figur 14 för att vänta på hela anropet innan programmet kan gå vidare.

```
public BankIdAuthenticateResponse login(final BankIdAuthenticateRequest request) {
    return bankIdApi.authenticate(
            X_CALLING_APPLICATION,
            Tenant.A,
            request,
            getUserID(),
            getSessionId()
    ).block();
}
```

Figur 14. Exempel på hur man kan använda sig av klientbiblioteket

6. SLUTSATSER

6.1 Resultat

Detta arbete var en del av ett större moderniseringsprojekt som handlar om att skapa fristående moduler och därigenom förenkla testningen. BankID-implementationen var en av de lättare delarna att bryta ut för att skapa en fristående applikation.

Applikationen kommer också att gå ut i produktionsmiljön. I början kommer den att köras parallellt med den tidigare versionen. Om någonting skulle gå fel kan vi med hjälp av en på/av-brytare i en databastabell bestämma vilken implementation som skall användas.

6.2 Framtida arbete

I framtiden kommer säkert en ändring av detta projekt att ske. För tillfället kallar detta API på en proxy som hanterar några säkerhetsaspekter. Men proxyn returnerar all respons den får från BankID med en HTTP-statuskod 200 även om BankID skulle svara med en statuskod på 400. Ändringen skulle bli att proxyn skulle skicka vidare samma statuskod som den får av BankID.

Projektet använder sig inte för tillfället av Spring Boot utan körs på våra interna servrar med hjälp av JBoss. Eftersom detta projekt är följande i kö för teamets produkter att få bli placerade i AWS molntjänster är applikationen ändå redan anpassad för Spring Boot. Om man testar applikationen lokalt kan man välja att köra den som en Spring Boot-applikation eller att distribuera den på sin lokala JBoss-server.

6.3 Egna reflektioner

Att integrera ny kod med befintlig kod och få allt att fungera i slutändan är ibland en utmaning. Men annars har detta projekt gått bra framåt och allting i dagsläge verkar fungera utan problem. Arbetet har också lärt mig mycket om olika verktyg som Springs webflux och generering av kod med OpenAPI-generatorn.

REFERENSER

About BankID. (u.å.). Hämtad 28 januari 2023, från https://www.bankid.com/en/privat/om-bankid

About Swagger Specification. (u.å.). Hämtad 28 januari 2023, från https://swagger.io/docs/specification/about/

Core Technologies. (u.å.-a). Hämtad 28 januari 2023, från https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-dependencies

Core Technologies. (u.å.-b). Hämtad 28 januari 2023, från https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans

Core Technologies. (u.å.-c). Hämtad 28 januari 2023, från https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-basics

Core Technologies. (u.å.-d). Hämtad 28 januari 2023, från https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-autowired-annotation

Documenting Your Existing APIs: API Documentation Made Easy with OpenAPI & Swagger. (u.å.). Hämtad 28 januari 2023, från https://swagger.io/resources/articles/documenting-apis-with-swagger/

Facade. (2022, januari 1). https://refactoring.guru/design-patterns/facade

Interface description. (u.å.). Hämtad 28 januari 2023, från https://www.bankid.com/utvecklare/guider/teknisk-integrationsguide/graenssnittsbeskrivning

OpenAPI Generator. (u.å.). Hämtad 28 januari 2023, från https://openapi-generator.tech/

OpenAPI Specification. (u.å.). Hämtad 28 januari 2023, från https://swagger.io/specification/

REST API Documentation Tool. (u.å.). Hämtad 28 januari 2023, från https://swagger.io/tools/swagger-ui/

Spring Framework. (2019, augusti 19). App Architecture; TechTarget.

https://www.techtarget.com/searchapparchitecture/definition/Spring-Framework

Swagger UI repository. (u.å.). Hämtad 28 januari 2023, från https://github.com/swagger-api/swagger-ui

Web on Reactive Stack. (u.å.). Hämtad 28 januari 2023, från https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-client

What is Java Spring Boot? (u.å.). Hämtad 28 januari 2023, från https://www.ibm.com/topics/java-spring-boot

What is REST? (u.å.). Codecademy. Hämtad 28 januari 2023, från https://www.codecademy.com/article/what-is-rest

What is REST. (2018, maj 29). REST API Tutorial; Lokesh Gupta. https://restfulapi.net/

What Is Restful API. (u.å.). https://aws.amazon.com/what-is/restful-api/