Examensarbete, Högskolan på Åland, Utbildningsprogrammet för Informationsteknik

INTEGRATION AV BEFINTLIGT SYSTEM MOT BOOMI

- en skräddarsydd programvara i Java

Tobias Östman

2022:03

Datum för godkännande: 15.02.2022
Handledare: Björn-Erik Zetterman

EXAMENSARBETE
Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Tobias Östman
Arbetets namn:	Integration av befintligt system mot boomi - en skräddarsydd programvara i Java
Handledare:	Björn-Erik Zetterman
Uppdragsgivare:	Gambit/Kund

Abstrakt

Syftet med detta examensarbete är att designa och implementera en anslutningshaterare till plattformen Boomi, en så kallad *Custom connector*. Anslutningshanteraren skall ha en gRPC-klient som tar kontakt med en gRPC-server.

Utveckling av anslutningshanteraren har skett i Java med hjälp av Gradle för att bygga ihop anslutningshanteraren.

Resultatet är en implementation av en anslutingshanterare till platformen boomi som kan dra nytta av gRPC och anropa det specificerade funktionerna. Anslutningshanteraren kan även själv formatera in och ut data.

Nyckelord (sökord)

Boomi, gRPC, Boomi Custom Connector, Reflection, Java

Högskolans serienummer:	ISSN:	Språk:	Sidantal:
2022:03	1458-1531	Svenska	32

Inlämningsdatum:	Presentationsdatum:	Datum för godkännande:
14.12.2021	15.12.2021	15.02.2022

DEGREE THESIS
Åland University of Applied Sciences

Degree Programme:	Bachelor of Information Technology/Bachelor of Engineering
Author:	Tobias Östman
Title:	Development of a Custom Boomi Connector - a custom software in Java
Academic Supervisor:	Björn-Erik Zetterman
Commissioned by:	Gambit/Customer

Abstract

The purpose of this thesis is to design and implement a custom connector to the platform Boomi. The custom connector must have a gRPC client that communicates with a gRPC server.

The chosen programming language used was Java with the help of Gradle to build the custom connector together.

The result is an implementation of a custom connector to the Boomi platform that can take advantage of gRPC and call the specified endpoints. The custom connector can also format the data.

Keywords

Boomi, gRPC, Boomi Custom Connector, Reflection, Java

Serial number:	ISSN:	Language:	Number of pages:
2022:03	1458-1531	Swedish	32

Handed in:	Date of presentation:	Approved:
14.12.2021	15.12.2021	15.02.2022

INNEHÅLLSFÖRTECKNING/TABLE OF CONTENTS

1. INTRODUKTION

1.1 Syfte

Syftet med detta examensarbete är att designa och implementera en anslutningshanterare till plattformen Boomi, en så kallad *Custom connector*. En anslutningshanterare är en klient som uppfyller en grundfunktion i Boomi. Denna grundfunktion är hur man kommer hämta eller skicka den data som Boomi har manipulerat. Anslutningshanteraren kommer att uppfylla ett behov hos kunden som har valt att vara anonym i detta arbete. Behovet är att flytta data från ett system till ett annat och manipulera den data som skickas från en anslutningshanterare till den anslutningshanterare som görs i detta examensarbete. Anslutningshanteraren skall ha en gRPC-klient som tar kontakt med en gRPC-server. gRPC är fjärranslutna procedur-anrop-system utvecklat av Google som används för att skicka och ta emot data.
Anslutningshanteraren skall kunna hantera olika operationer, såsom *Get, Update, Delete, Create* och presentera informationen som kommer från dessa operationer.

1.2 Kravspecifikation

I detta examensarbete skall dessa punkter fullgöras:

1. Implementera en anslutningshanterar till plattformen Boomi
2. Anslutningshanteraren skall fungera med gRPC
3. Anslutningshanteraren skall kunna göra anrop till specificerade funktioner
4. Anslutningshanteraren skall kunna formatera in- och ut-data
5. Anslutningshanteraren skall vara krypterad med ett SSL-certifikat

1.3 Metod

En lista med vanliga definitioner och akronymer finns med i bilagorna.
Utvecklingsmiljön kommer att vara Boomi[1]-specifikt vilket betyder att programmeringsspråket kommer att vara Java, version 8. Senare versioner av Java stöds ej av

[1] https://boomi.com/

Boomi då detta arbete utförs hösten 2021. Verktyget för att bygga ihop helheten för testning kommer vara Gradle[2], vilket medför fördelen att det går att automatisera delar av processen att kompilera om gRPC samt bygga relevanta JAR-filer.

Det naturliga valet av integrerad utvecklingsmiljö (IDE[3]) kommer att vara Intellij. Företaget Gambit[4] använder sig av Jetbrains-paketet så professionella versionen finns tillgänglig.

Utvecklingen kommer att ske stegvis, som metod används *Evolutionary prototyping* som är ett sätt att testa sig fram och se vad som fungerar (Hekmatpour, 1987). På kundens begäran kommer jag att verifiera att en funktion har implementerats rätt för att sedan ladda upp det till testmiljön och säkerställa att funktionen fungerar enligt kraven. Om funktionen presterar ett bra resultat, gå vidare till nästa funktion/krav om kraven ej uppfylls, gå tillbaks och förbättra för att sedan återkomma till testningen. Testmiljön har kunden satt upp till vårt förfogande. Kunden är ett anonymt företag som har anlitat Gambit att göra detta projekt.

1.4 Avgränsningar

Eftersom gRPC-delen av projektet utvecklas av en tredje part kommer inte alla slutpunkter (endpoints) vara fungerande. Ur säkerhetssynpunkt saknas ännu ett SSL-certifikat för kryptering, vilket behöver kompletteras före produktions release. Tillsvidare körs anslutningshanteraren i klartext, utan kryptering.

1.5 Uppdragsgivare

Uppdragsgivaren för det här examensarbetet var Gambit Group. Gambit är ett IT-företag som levererar olika digitaliseringslösningar. Deras arbetsområde sträcker sig från mindre till större företag. Företaget har ca. 50 anställda, huvudkontoret finns i Vasa. Deras övriga kontor finns i Jakobstad och Varkaus (About us - gambit, 2019).

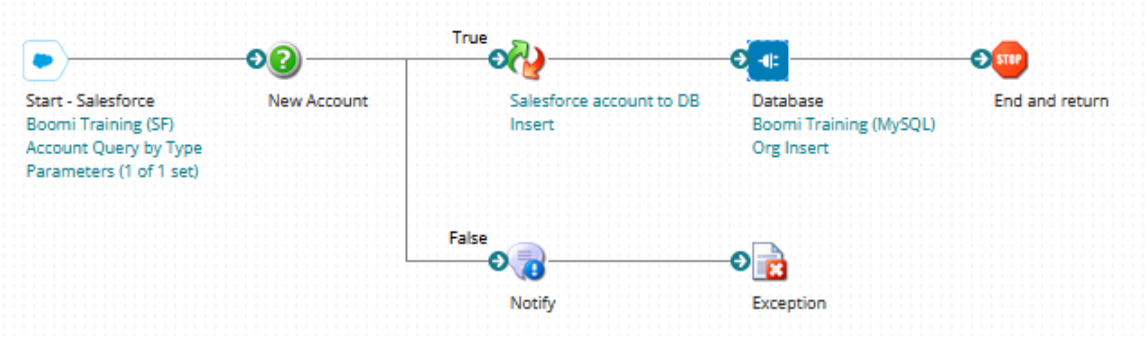
[2] https://gradle.org/
[3] https://en.wikipedia.org/wiki/Integrated_development_environment
[4] Gambit är företaget som arbetet görs åt, https://www.gambitgroup.fi/about-us/

2. TEKNISK PLATTFORM

2.1 Boomi

Boomi är en integrationsplattform som specialiserar sig i *integration platform as a service*, datahantering och data förberedelser (Dignan, 2019). Företaget började året 2007 med målet att skapa konfigurations baserad integration (Metz, 2011) .

I Boomi ritar man upp sina processer med hjälp av olika färdigt definierade verktyg. I Figur 1 visas en bild av en enkel process i Boomi. Denna process tar data från källan *Salesforce* och kontrollerar ifall det är ett nytt konto som hämtats. Ifall det inte är ett nytt så skapar processen ett undantag, annars skickar processen datan till en databas för att lagras (Boomiverse, 2021).



Figur 1. En simpel Boomi-process (Boomiverse, 2021)

2.2 Boomi Connectors

Connectors[5] i Boomi-plattformen är färdig-definierade verktyg för att koppla upp sig mot FTP, DISK, HTTP, databaser, Salesforce, Google G Suite osv. Dessa finns då färdigt i Boomi redo att användas. De innehåller 3 viktiga delar (Boomi AtomSphere Documentation, n.d).

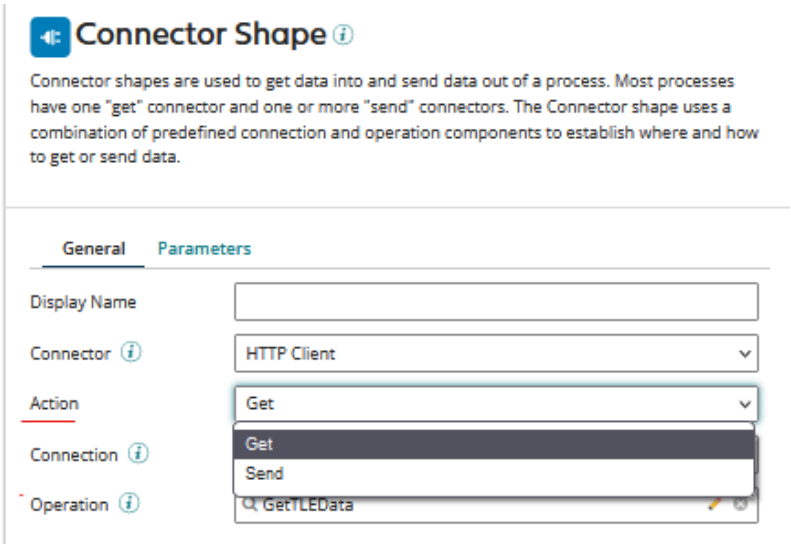
- En *Action* - vad ska connectorn göra.
- En *Connection* - vart ska connectorn göra ett anrop
- En *Operation* - vad skall connectorn göra med den data som tas emot

[5] https://help.boomi.com/bundle/connectors

Detta är de tre byggstenarna som en connector innehåller, och kan användas till olika användningsfall.

2.2.1 Action

En action[6] är olika typer av operationer som kan köras t.ex. *Get, Send, Delete, Put, Query*. Det finns oftast flera olika *actions* att välja mellan. Som man ser i figur 2 finns det två alternativ för en *HTTP Client connector, get* och *send*. Alternativet för handling som väljs ger en grund för vad som skall hända med data som skickas/efterfrågas.

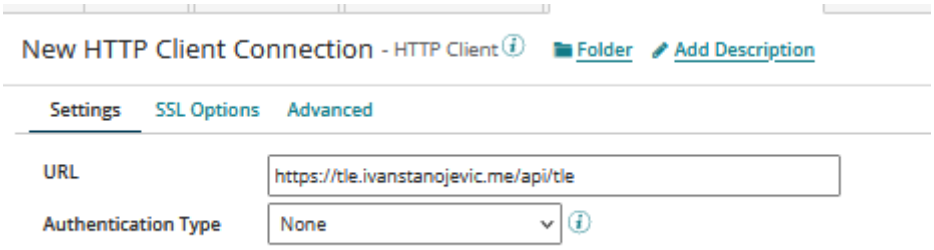


Figur 2. En connector shape

2.2.2 Connection

En connection[7] ger anslutningshanteraren ett ställe att skicka / hitta data som skickas / efterfrågas. På figur 3 sätter man en URL för HTTP-klienten som den då kan skicka eller efterfråga data från.

[6]hittas under *connector configuration* https://help.boomi.com/bundle/connectors
[7]hittas under *connector components* kapitlet https://help.boomi.com/bundle/connectors

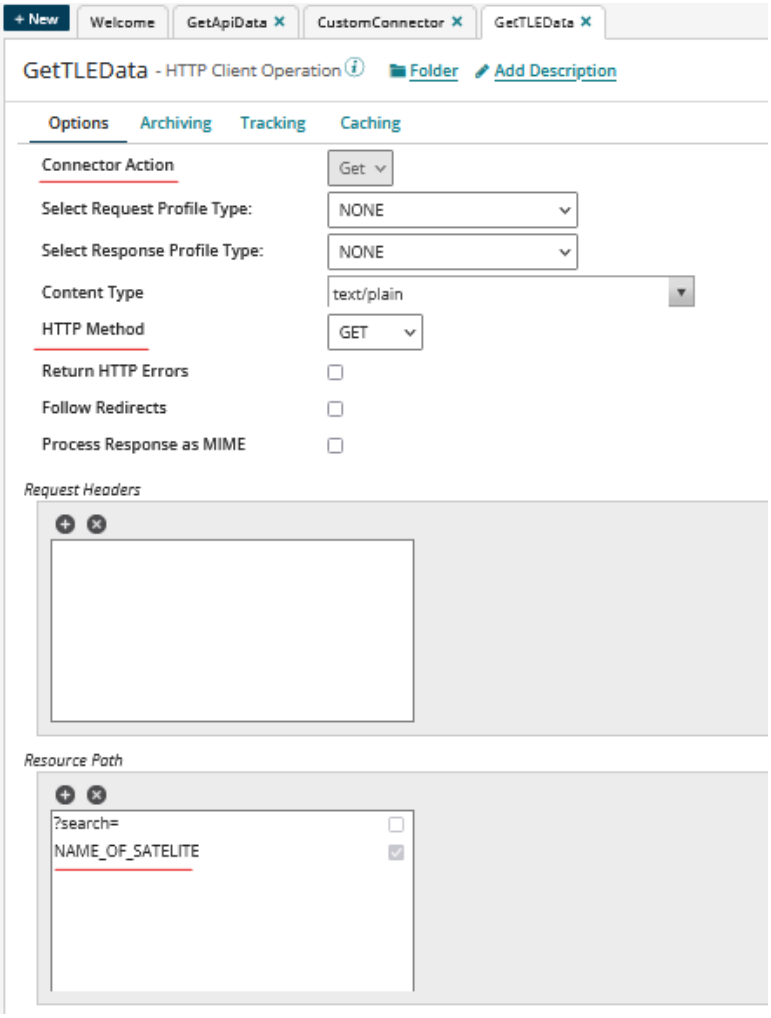


Figur 3. Konfigurationsanvändargränssnit för en anslutningshanterares anslutningsfält

2.2.3 Operation

En operation[8] definierar vad applikationen skall göra med datan den får in. I operationen kan man vara riktigt specifik på vad man vill att anslutningshanteraren skall göra med datan. I figur 4 ser man att *connector action* är *Get*. Applikationen kommer att hämta ut data. Då kan man också sätta http methoden till *GET*[9] och sedan specificerar man ännu vilka sökparametrar man vill att anslutningshanteraren skall använda. Se i figur 4 på fältet *Resource Path*

[8]hittas under *connector components* kapitlet https://help.boomi.com/bundle/connectors
[9] https://www.hjp.at/doc/rfc/rfc1945.html



Figur 4. Konfigurations-användargränssnitt för en anslutningshanterare operationsfält

2.3 gRPC

gRPC (remote procedure calls) är byggt av Google 2015 som nästa generation av RPC-infrastrukturen Stubby. RPC är ett sätt att kommunicera mellan system, där syftet är att optimera för mindre mängd data i överföringarna (Bagci and Kara, 2016). Det använder sig av HTTP/2 som transport-lager och *Protocol buffers* som beskrivningsspråk. Det finns en hel del funktioner såsom (Indrasiri and Kuruppu, 2020).

- autentisering (eng authentication)
- tvåvägs strömning (eng bidirectional streaming)

- flödeskontroll (eng flow control)
- blockerande eller icke blockerande bindningar (eng blocking or non blocking bindings)
- annulleringar och timeouts (eng cancellation and timeouts)
- genererar plattformsoberoende klient- och server bindningar för många språk

2.3.1 RPC

Står för *remote procedure call* och innebär att en dator kör en subrutin i ett annat adressutrymme. Detta kan liknas vid att man skickar ett mail åt en vän och frågar ifall hen kan skicka ett par nybakade kakor åt dig. I praktiken så tar servern emot ett par parametrar som t.ex (jag vill ha kakor, skicka dem åt mig), och sedan sköter servern resten den bakar kakorna åt dig och skickar resultatet. RPC är ett *request-response*-protokoll som börjar med att en klient skickar iväg en förfrågan till en känd server. Servern skickar då tillbaka ett svar och applikationen fortsätter köras. Så länge servern behandlar förfrågan är klienten i ett blockat läge och väntar på ett svar ifall inte implementation tillåter asynkrona förfrågningar (Birrell and Nelson, 1984).

2.3.2 Protocol buffers

Protocol buffers (protobuf) är ett *open-source*-bibliotek som används för att serialisera strukturerad data. Google utvecklade *Protocol buffers* för att lätt kunna generera kod för flera programmeringsspråk. (Kalman, M, 2013).

2.3.3 Services

Som många andra RPC-system är gRPC-baserat på att definiera olika tjänster som då specificerar vilka metoder som finns tillgängliga. Det finns fyra olika service-metoder (Core concepts, architecture and lifec…, n.d):

- Klienten skickar ett meddelande till servern, servern svarar med ett meddelande
- Klienten skickar ett meddelande till servern, servern svarar med en ström
- Klienten skickar en ström, servern svarar med ett meddelande
- Klienten skickar en ström, servern svarar med en ström

En ström är i princip ett flöde av data som kan liknas vid att läsa en bok medan boken blir skriven. Medan ett meddelande kan liknas vid en färdig bok som kan läsas direkt.
Se figur 5 för exempel på en service tagen från gRPC-dokumentationen.

```
// The greeter service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}
```

Figur 5. En enkel *service*-definition

2.3.4 Meddelanden

Ett message[10] är en logisk struktur som innehåller en serie med *name-value pairs* som kallas *field* eller fält på svenska. *name-value pairs* är en kompakt mängd av data där namnet användes för att identifiera data-värdet. Se figur 6 för exempel av ett *message* taget från gRPC-dokumentation

```
message Person {
  string name = 1;
  int32 id = 2;
  bool has_ponycopter = 3;
}
```

Figur 6. En enkel definition av ett *message*

2.4 Utvecklingsmiljön

2.4.1 Java

Java[11] är ett objektorienterat programmeringsspråk som använder sig av JVM (Java virtual machine) för att tolka och köra kod (Lopez, 2007). Java är ett så kallat general-purpose programmeringsspråk vilket ger möjlighet till, *write once, run anywhere* (WORA). Detta

[10]https://developers.google.com/protocol-buffers/docs/overview
[11] https://www.oracle.com/java/technologies/introduction-to-java.html

innebär att kompilerad Javakod ska kunna köras på vilken maskin som helst (Blom et al, 2008). Boomi använder sig också av Java så det naturliga valet / enda valet är Java som programmeringsspråk.

2.4.1.1 Java-reflektion

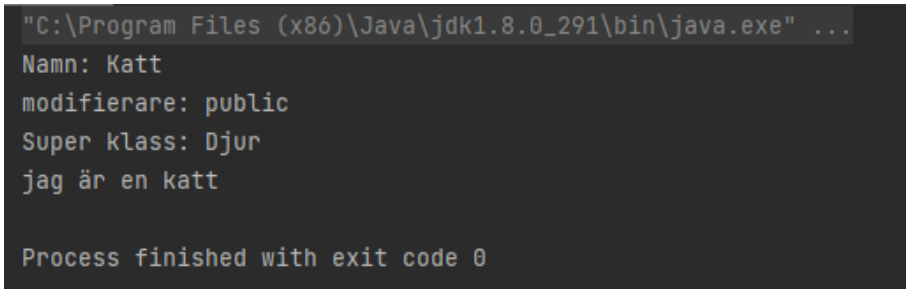
Reflektion är ett sätt att inspektera och modifiera klasser, gränssnitt, metoder och fält under körtid. Detta snabbar upp utvecklingsprocessen och gör flödet mera flexibelt (Li et al. 2019).

Ett enkelt exempel på hur reflektion fungerar. I exemplet finns en basklass Djur som får förbli tomt sedan har exemplet en klass Katt som utvidgar klassen djur. Klassen Katt innehåller också en metod *display* som skriver ut strängen “jag är en katt”. Om man då skapar en *main* klass med en statisk funktion *main* så kan man testa reflektionen. Figur 7 visar ett enkelt exempel på hur reflektion fungerar och figur 8 visar resultatet från en körning.

```
public class main {
    public static void main(String[] args) {
        try {

            Katt katt = new Katt();

            Class obj = katt.getClass();

            // hämta ut objektets namn
            String name = obj.getName();
            System.out.println("Namn: " + name);

            // hämta bestämningen (public/private/static/protected)
            int modifier = obj.getModifiers();

            // konvertera bestämningen till en sträng
            String mod = Modifier.toString(modifier);
            System.out.println("modifierare: " + mod);


            Class superClass = obj.getSuperclass();
            System.out.println("Super klass: " + superClass.getName());
            // Hitta metoden som matchar strängen
            Method method = obj.getMethod( name: "display");
            // kalla på metoden med hjälp av "invoke" och vår instansierade klass som object
            method.invoke(katt);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
public class Katt extends Djur {
    public void display() {
        System.out.println("jag är en katt");
    }
}

public class Djur {
}
```

Figur 7. *Main* Klassen, Djur klassen och Katt klassen



Figur 8. Utskriften för exemplet

Så med hjälp av reflektion kan man få ut mycket information om en klass under körtid och till och med anropa metoder med hjälp av strängar.

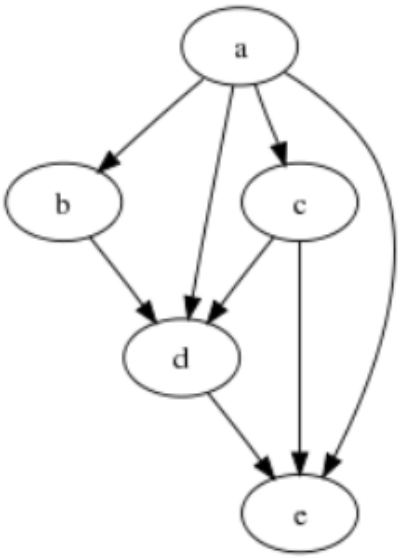
2.4.2 Gradle

Gradle[12] är ett automationsverktyg som stöder många språk så som

- Java[13]
- Kotlin[14]
- Groovy[15]
- C/C++[16]

Gradle skapades för att bygga flera "mindre" projekt samtidigt som sedan kombineras till ett större. Dessa byggprocesser kan köras i serie eller parallellt. Inkrementella byggprocesser stöds också genom att bestämma vilka delar av bygg-trädet som redan är uppdaterade Gradle använder sig av riktad acyklisk graf för att bestämma i vilken ordning den skall bygga så kallade *Tasks* (What is Gradle?, n.d). Figur 9 visar en riktad acyklisk graf, På figuren finns det linjer som kallas kanter och bollarna som kallas vertex. Från varje vertex förutom sista går det kanter till en eller flera vertexer vilket Gradle då använder för att avgöra vilka *Tasks* som ska byggas.

[12] https://docs.gradle.org/current/userguide/what_is_gradle.html
[13] https://www.oracle.com/java/technologies/introduction-to-java.html
[14] https://kotlinlang.org/api/latest/jvm/stdlib/
[15] https://en.wikipedia.org/wiki/Apache_Groovy
[16] C https://en.wikipedia.org/wiki/C_(programming_language), C++ https://en.wikipedia.org/wiki/C%2B%2B



Figur 9 En riktad acyklisk graf

2.4.4 Designmönster

2.4.4.1 Data transfer object (DTO)

Ett *Data transfer object* är ett objekt som transporterar data mellan två olika processer. Objektet skall inte innehålla någon logik med det undantaget att man får implementera logik för serialisering och deserialisering. Ett exempel på tillämpningsområde för detta är ifall man har en process som tar upp en hel del resurser och vill få ner på mängden data som blir flyttad så ska man använda sig av ett "DTO" (Archiveddocs, n.d).

2.4.4.2 Factory mönstret

*"*Factory metoden definierar ett gränssnitt för att skapa objekt men låter härledda klasser bestämma vilket objekt som ska skapas*"* (Mu and Jiang, 2011).

3. APPLIKATIONSUTVECKLING

3.1 Förberedelser

Innan någon kod alls skrevs så gjorde jag i samarbete med beställaren en hel del testprocesser. Vilket gav mig en grund till vad som kan göras och vad som behöver omvärderas.

Första handledningen jag började med var ett simpelt och klassiskt *Hello world*[17] program. Som mitt andra test så gjorde jag en *bridge of death*-anslutningshaterare som då refererar till kända engelska tv programmet *Monty Python*[18]. I processen skickade man in variabler till anslutningshanteraren och baserat på dessa variabler så slapp man över bron eller så blev man kastad ner i ravinen. Som mitt sista test gjorde jag *GetCustomer* slutpunkten som då skall hämta ut en kund baserat på ett ID.

3.2 Konfigurationsfiler

3.2.1 Connector-config.xml

Konfigurationsfilen connector-config.xml[19] definierar startpunkten i vårt program. I figur 10 ser man hur konfigurationsfilen gjordes.

```
<?xml version="1.0" encoding="UTF-8"?>
<GenericConnector>
    <connectorClassName>PesConnector.PesConnector</connectorClassName>
</GenericConnector>
```

Figur 10. Konfigurationsfilen *connector-config.xml*

I dessa filer kan man också definiera versioner av programmet med mera. I början av utvecklingen av projektet är det inte nödvändigt, så det hålls avskalat till en början.

[17] https://en.wikipedia.org/wiki/%22Hello,_World!%22_program
[18] https://en.wikipedia.org/wiki/Monty_Python
[19] https://help.boomi.com/bundle/connectors/page/int-Connector_configuration_file.html

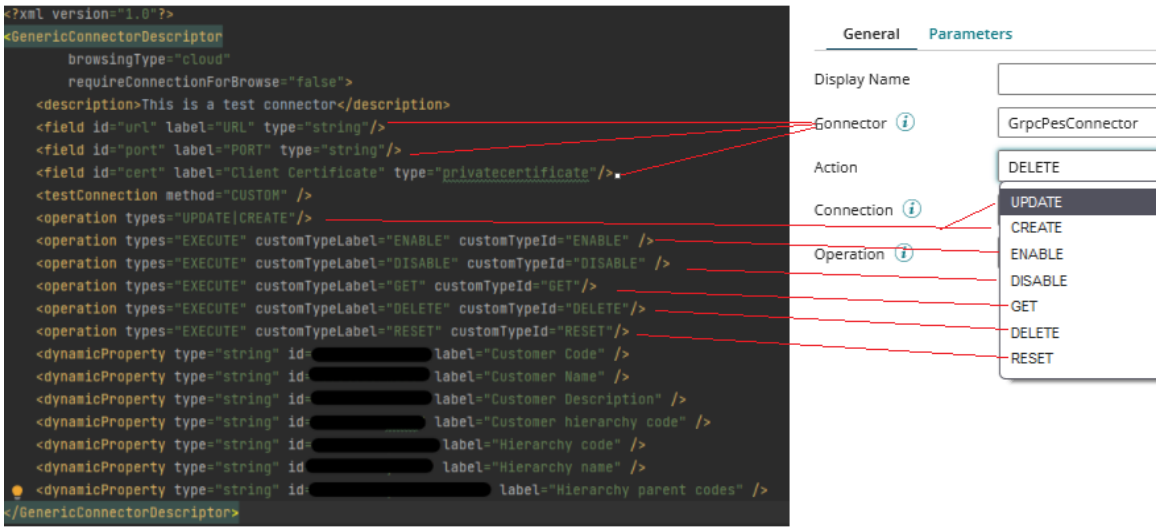
3.2.2 Connector-descriptor.xml

Konfigurationsfilen connector-descriptor.xml[20] definierar hur anslutningshanteraren skall se ut samt vilka typer av funktionalitet som skall finnas med. Det vill säga

- fält där man kan ta in strängar, certifikat, lösenord…
- *testConnection*, vilket är ett sätta att kolla uppkopplingen mot t.ex en databas
- operationer, vilka typer av grundfunktioner som skall finns tillgängliga
- dynamiska variabler, som man kan sätta i Boomi

Det finns många fler valmöjligheter men dessa listade ovan användes i projektet.

Figur 11 visar descriptor-filen som används i projektet samt de röda strecken visar vilka konfigurations-fält som hör ihop med användargränssnittet i Boomi.



Figur 11. Konfigurationsfilen *connector-descriptor.xml* och vilka fält som hör ihop med vad i Boomi

[20] https://help.boomi.com/bundle/connectors/page/int-Describing_the_capabilities_of_a_custom_connector.html

3.3 Grund-klasserna

3.3.1 Connector-Klass

Detta är en så kallad (entry point på engelska) eller "startställe" på svenska. Denna ska då ärva en klass som heter *baseConnector*[21]. Denna klass kräver att man implementerar en metod som heter *createBrowser*[22]. I denna klass skall också alla funktioner som tidigare definierats i connector-descriptor-xml filen också implementeras. Se figur 12 för implementationen.

```java
public class PesConnector extends BaseConnector {

    private final Logger logger = Logger.getLogger(PesConnector.class.getName());

    @Override
    public Browser createBrowser(BrowseContext context) {
        return new PesBrowser(new PesConnection(context));
    }

    @Override
    public Operation createCreateOperation(OperationContext context) {
        return new PesCreateOperation(createConnection(context));
    }

    @Override
    protected Operation createUpdateOperation(OperationContext context) {
        return new PesUpdateOperation(createConnection(context));
    }

    @Override
    protected Operation createExecuteOperation(OperationContext context) {
        return new PesExecuteOperation(createConnection(context));
    }

    private PesConnection createConnection(BrowseContext context) {
        return new PesConnection(context);
    }


}
```

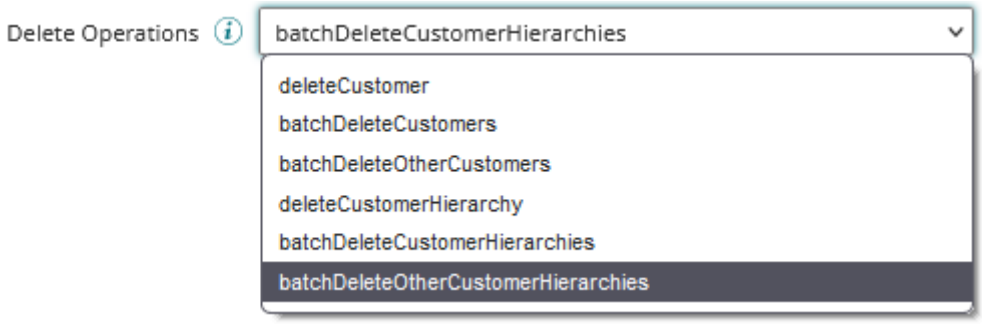
Figur 12. Innehållet av *PesConnector*-filen

[21] https://boomisdkjavadoc.s3.amazonaws.com/javadoc/2.12.0/com/boomi/connector/util/BaseConnector.html
[22]https://boomisdkjavadoc.s3.amazonaws.com/javadoc/2.12.0/com/boomi/connector/api/Connector.html#createBrowser-com.boomi.connector.api.BrowseContext-

3.3.2 Browser-klass

Browser-klassen[23] är den som styr vad en anslutningshanteraren kan importera, exportera. baserat på olika objekttyper. En objekttyp består av ett id och en etikett. Dessa ger dig möjlighet att i Boomi välja olika förfrågnings-profiler eller svar-profiler. Objekt-definitioner använder sig av objekttyper för att hitta rätt profil.

I detta fall användes browser-klassen för att gruppera olika funktioner eftersom gRPC tjänsten hade flertal av samma funktion med lite annan funktionalitet. I figur 13 finns de olika *delete*-funktioner som existerar i anslutningshateraren. I figur 14 finns implementationen som visar hur objekt-definitionerna blir hämtade.



Figur 13. Tillåtna *delete operations*

```
@Override
public ObjectDefinitions getObjectDefinitions(String objectTypeId, Collection<ObjectDefinitionRole> roles) {

    OperationSupplier supplier = new OperationSupplier();
    IOperationDefinitions definitions = supplier.supplyDefinitions(Optional.ofNullable(getContext().getCustomOperationType())
            .orElse(getContext().getOperationType().name()));

    ObjectDefinitions defs = definitions.createObjectDefinitions(objectTypeId);
    return defs;
}
```

Figur 14. Implementation av *getObjectDefinitions*

[23] https://boomisdkjavadoc.s3.amazonaws.com/javadoc/2.12.0/com/boomi/connector/api/Browser.html

3.3.3 Connection-klass

I *connection*-klassen[24] tar man ut konfigurations-fält från descriptor-filen och använder dem för att bygga upp en uppkoppling mot det man vill ansluta till. Man kan då instansiera denna klass i andra klasser för att få ut en uppkopplings-sträng. Se figur 15

```
public class PesConnection extends BaseConnection {

    private static final String BASE_URL_FIELD = "url";
    private static final String PORT_FIELD = "port";
    private static final String CERT_FIELD = "cert";

    public PesConnection(BrowseContext context) { super(context); }

    public String getBaseUrl() { return getContext().getConnectionProperties().getProperty(BASE_URL_FIELD); }

    public String getPort() { return getContext().getConnectionProperties().getProperty(PORT_FIELD); }

    public PrivateKeyStore getCert() {
        return getContext().getConnectionProperties().getPrivateKeyStoreProperty(CERT_FIELD);
    }

}
```

Figur 15. Implementation av *PesConnection*

3.3.4 Operations-klasserna

Dessa klasser är specifikt namngivna efter den grundfunktion som den skall utföra. T.ex man har så kallade *create*-operationer. Då ska klassen implementera *BaseUpdateOperation*[25]-klassen. Ifall man skulle ha en så kallad *delete*-operation skulle den implementera *BaseDeleteOperation*[26]. I detta fall implementerar dock alla funktioner *BaseUpdateOperation*. På grund av att reflektion kommer att användas behöver parametrarna vara så generella som möjligt.

[24] https://boomisdkjavadoc.s3.amazonaws.com/javadoc/2.12.0/com/boomi/connector/util/BaseConnection.html
[25] https://boomisdkjavadoc.s3.amazonaws.com/javadoc/2.12.0/com/boomi/connector/util/BaseUpdateOperation.html
[26] https://boomisdkjavadoc.s3.amazonaws.com/javadoc/2.12.0/com/boomi/connector/util/BaseDeleteOperation.html

3.4 Proto-filerna

3.4.1 Slutpunkter

Slutpunkterna eller (endpoints på engelska) är de som kommer att ta emot olika förfrågningar och styra dem vidare till rätt funktion och sedan svara med ett “JSON”-meddelande. Implementeringen av slutpunkterna i tabell 1 gjordes med hjälp av reflektion där man då lätt kunde implementera flera metoder som anropar samma funktion men ger in en annan kontext. Detta drog ner på antalet rader kod och gjorde filen mera hanterbar. I figur 16 finns implementationen för funktionen som hanterar alla enskilda kundförfrågningar.

Tabell 1. Slutpunkter för en enskild kund

Slutpunkter för modifikation av en kund	
Slutpunktsnamn	**Beskrivning**
GetCustomer	Hämtar ut en kund baserat på id
AddCustomer	Sätter till en kund
SetCustomer	Sätter till en kund eller uppdaterar befintlig
UpdateCustomer	Uppdaterar en befintlig kund
DeleteCustomer	Tar Bort en kund baserat på id
EnableCustomer	Sätter en kund status till aktiv baserat på id
DisableCustomer	Sätter en kund status till inaktiv baserat på id

```
public void executeOperationRequest(UpdateRequest request, OperationResponse response, String operationMethod) throws InvocationTargetException, IllegalAccessException,
    Object objResponse;
    Class[] parameterTypes = new Class[1];
    parameterTypes[0] = Customermessages.CustomerHierarchy.class;

    for (ObjectData data : request) {
        Customermessages.CustomerHierarchy customerHierarchy = createProtoCustomerHierarchyDynamicProps(data, this.context.getOperationType());
        Method method = this.client.getClass().getMethod(operationMethod, parameterTypes);
        objResponse = method.invoke(this.client, customerHierarchy);

        if (objResponse instanceof Common.DefaultResponse) {
            Common.DefaultResponse hierarchyDefRes = (Common.DefaultResponse) objResponse;
            Utils.handleResponse(response, data, JsonFormat.printer().print(hierarchyDefRes),
                    hierarchyDefRes.getResponseRows( index: 0));
        } else if (objResponse instanceof Customermessages.CustomerHierarchyQueryResponse) {
            Customermessages.CustomerHierarchyQueryResponse customerHierarchyQueryResponse = (Customermessages.CustomerHierarchyQueryResponse) objResponse;
            Utils.handleResponse(response, data, JsonFormat.printer().print(customerHierarchyQueryResponse),
                    customerHierarchyQueryResponse.getResponseRows( index: 0));
        }
    }
}
}
```

Figur 16. En överblick av *executeOperationRequest*

Slutpunkterna för flera kunder finns i tabell 2 och gjordes på samma sätt med hjälp av reflektions-funktionen. Skillnaderna är dock när man hanterar flera kunder behöver man mappa en kund till ett kundobjekt. Det skapades alltså en *CustomerDTO* för att hantera mappningen av en kund till ett kund objekt. Med hjälp av detta *Data transfer object* kunde man lätt använda sig av en "JSON"-översättare för att mappa datan rätt. Se figur 17 för implementation. Allt man behöver göra är att läsa in allt som en "JSON"-räcka och sedan skicka in det till *customerMapper* som då sköter om mappningen till ett *CustomerBatch*-objekt.

Tabell 2. slutpunkter för en flera kunder

Slutpunkter för modifikation av flera kunder	
Slutpunktsnamn	**Beskrivning**
BatchAddCustomers	Lägger till flera kunder
BatchSetCustomers	Lägger till eller uppdaterar flera kunder
BatchUpdateCustomers	Uppdaterar flera kunder baserat på id
BatchDeleteCustomers	Tar bort flera kunder baserat på id
BatchEnableCustomers	Sätter flera kunder till status aktiv baserat på id
BatchDisableCustomers	Sätter flera kunder till status inaktiv baserat på id

```
private Customermessages.CustomerBatch customerMapper(String input) {
    Gson gson = new Gson();
    Type customerListType = new TypeToken<ArrayList<CustomerDTO>>() {
    }.getType();
    ArrayList<CustomerDTO> customers = gson.fromJson(input, customerListType);
    Customermessages.CustomerBatch.Builder customerBatchBuilder = Customermessages.CustomerBatch.newBuilder();

    for (CustomerDTO customerDTO : customers) {
        if ((context.getOperationType() == OperationType.CREATE || context.getOperationType() == OperationType.UPDATE)
                && customerDTO.getName() == null) {
            throw new ConnectorException("Name cannot be null on update or create");
        }
        Customermessages.Customer tempCustomer = Customermessages.Customer.newBuilder()
                .setCode(customerDTO.getCode())
                .setName(Optional.ofNullable(customerDTO.getName()).orElse( other: ""))
                .setDescription(Optional.ofNullable(customerDTO.getDescription()).orElse( other: ""))
                .setCustomerHierarchyCode(Optional.ofNullable(customerDTO.getCustomer_hierarchy_code()).orElse( other: ""))
                .setStatus(Optional.ofNullable(customerDTO.getStatus()).orElse(Common.Status.ACTIVE))
                .build();
        customerBatchBuilder.addCustomers(tempCustomer);
    }
    return customerBatchBuilder.build();
}
```

Figur 17. Implementation av *customerMapper*

3.4.2 Kompilation av proto-filer

För att kunna ta nytta av de definierade proto-filerna behöver man kompilera dem för att få färdig genererad kod. För att kompilera koden användes Gradle, se figur 18.

```
protobuf {
    protoc { artifact = "com.google.protobuf:protoc:${protocVersion}" }
    plugins {
        grpc { artifact = "io.grpc:protoc-gen-grpc-java:${grpcVersion}" }
    }
    generateProtoTasks {
        all()*.plugins { grpc {} }
    }
}
```

Figur 18. Gradle-kompilation av proto-filer

I projektet behövde man även kunna använda de genererade filerna. Därför definierades *sourceSets* som informerar *integrated development enviroment (IDE)* om att detta är ett kodarkiv man kan ta nytta av. Se figur 19.

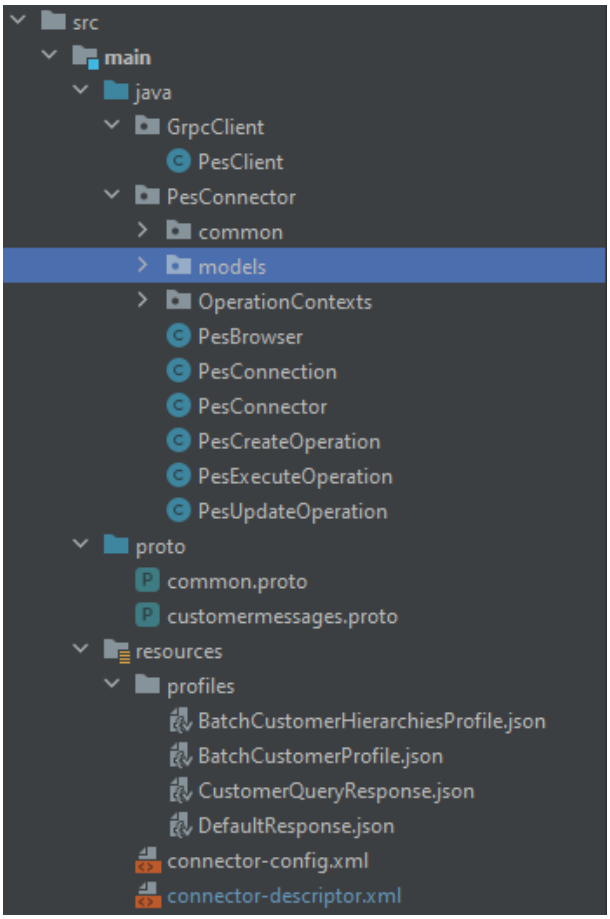
```
// Inform IDEs like IntelliJ IDEA, Eclipse or NetBeans about the generated code.
sourceSets {
    main {
        java.srcDirs += 'build/generated/source/proto/main/grpc'
        java.srcDirs += 'build/generated/source/proto/main/java'
    }
}
```

Figur 19. Få Gradle att visa kompilerade mappar som användbara mappar

3.5 Filstruktur

3.5.1 Grundstrukturen

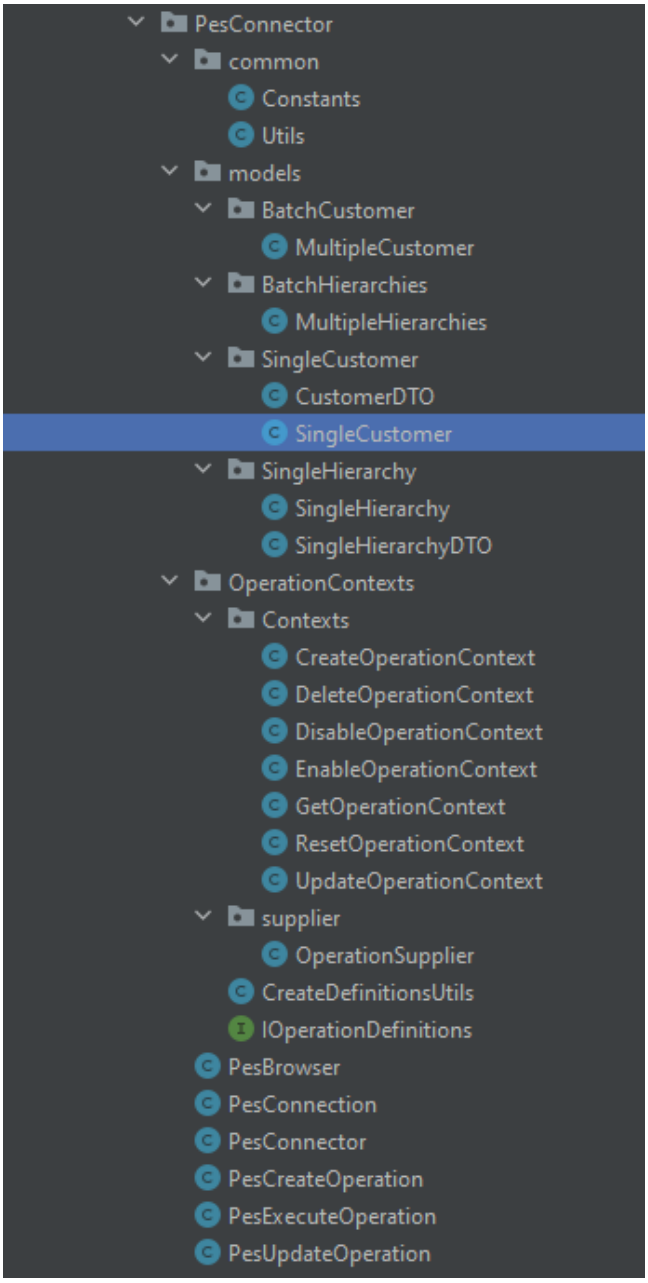
Projektet är delat i två delar, Pes-klienten och anslutningshateraren. Det gjordes två huvudmappar: Den första var en *GrpcClient*. Inne i denna mapp sätter man allt som har med uppkopplingen mot Grpc servern att göra. Andra är *PesConnector* där man sätter allt som har med logiken för anslutningshanteraren att göra. Se figur 20.



Figur 20. Grundstrukturen för projektet

3.5.2 Connector-strukturen

Inne i *PesConnector*-mappen gjordes även ett par andra mappar: *Common* dit man sätter allt som många filer/klasser behöver. Models där har man största delen av logiken och *Data transfer objects* (DTO) och till sist *OperationContexts* som innehåller våra operations-kontexter. Se figur 21.



Figur 21. Filstrukturen för connector-delen

3.6 Implementering av designmönster

3.6.1 Factory mönstrets implementation

Factory mönstret implementerades med namnet *OperationSupplier* därför att det var mera beskrivande av vad funktionen faktiskt gjorde. Som funktionsparameter användes en sträng för att kunna skapa en klass baserat på strängens värde. Man kommer även undan kravet att

ha en härledd klass för att skapa klasserna. Det gav oss möjlighet att instansiera olika klasser baserat på vad man satte som input parameter, se figur 22.

```
public class OperationSupplier {

    private static final Map<String, Supplier<IOperationDefinitions>> DEFINITIONS_SUPPLIER;

    static {
        final Map<String,Supplier<IOperationDefinitions>>
                definitions = new HashMap<>();
        definitions.put("CREATE", CreateOperationContext::new);
        definitions.put("UPDATE", UpdateOperationContext::new);
        definitions.put("GET", GetOperationContext::new);
        definitions.put("DELETE", DeleteOperationContext::new);
        definitions.put("ENABLE", EnableOperationContext::new);
        definitions.put("DISABLE", DisableOperationContext::new);
        definitions.put("RESET", ResetOperationContext::new);
        DEFINITIONS_SUPPLIER = Collections.unmodifiableMap(definitions);
    }

    public IOperationDefinitions supplyDefinitions(String operationType){
        Supplier<IOperationDefinitions> operationDefinitionsSupplier = DEFINITIONS_SUPPLIER.get(operationType);

        if(operationDefinitionsSupplier == null){
            throw new ConnectorException("Invalid operation type: " + operationType);
        }
        return operationDefinitionsSupplier.get();
    }
}
```

Figur 22. Implementation av *OperationSupplier*

3.6.2 DTO mönstrets Implementation

Två klasser skapades *CustomerDTO* och *CustomerHierarchyDTO* dessa användes för att mappa *JSON* data till *CustomerDTO* objekt. Tack vare designmönstret så drog vi ner på antal rader kod eftersom det inte behövdes någon konverterings funktion. Se figur 23 för användningsfallet av design mönstret

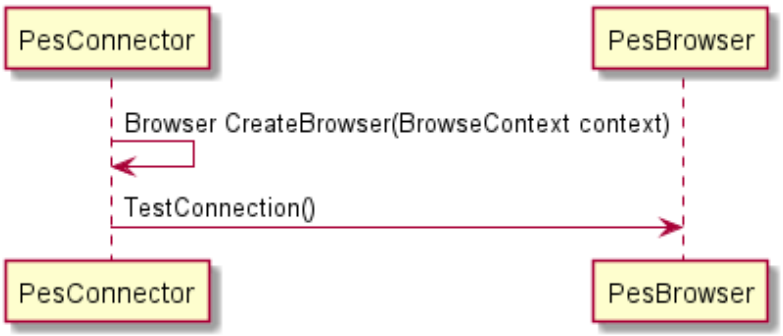
```
private Customermessages.CustomerBatch customerMapper(String input) {
    Gson gson = new Gson();
    Type customerListType = new TypeToken<ArrayList<CustomerDTO>>() {
    }.getType();
    ArrayList<CustomerDTO> customers = gson.fromJson(input, customerListType);
```

Figur 23. Funktionen för att mappa *JSON* till en räcka med *CustomerDTO*:s.

3.7 Projektflödet

3.7.1 Test uppkopplingen mot servern

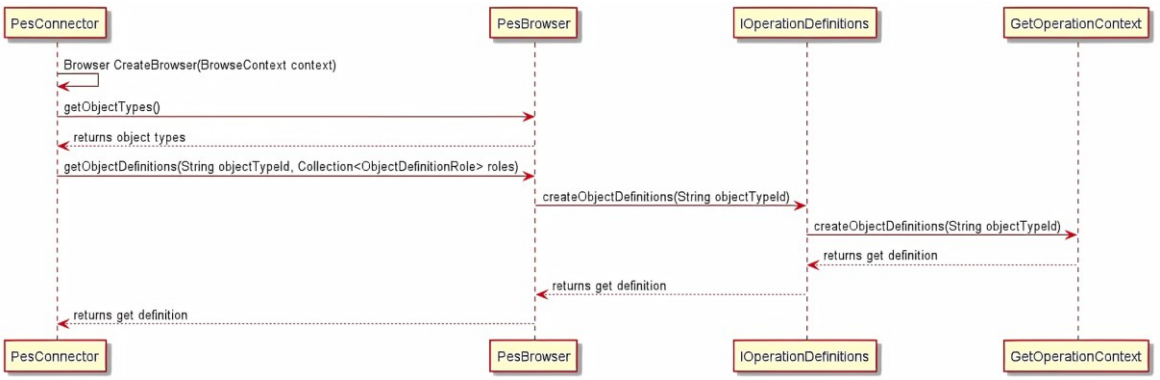
Flödet på figur 24 motsvarar en användare som trycker på knappen *Test connection* i Boomi. Då startas detta flöde. Allt börjar med *PesConnectorn* som skapar en instans av *PesBrowser*-klassen och anropar metoden *testConnection*. *testConnection* skickar då ett anrop till servern och väntar på ett svar ifall det lyckas returneras *true,* om inte *false*.



Figur 24. Flödet för operationen *testConnection*

3.7.2 Hämta specifika operations-definitioner

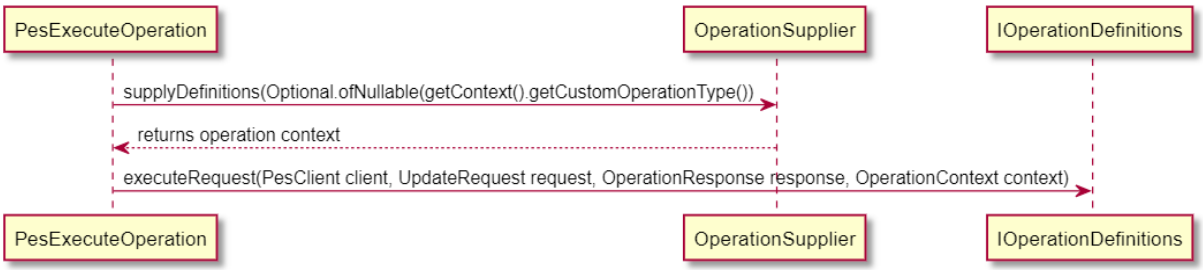
I detta fallet vill man ha ut operationsdefinitioner för *GET*-metoden. Allt börjar med att *PesConnector* skapar en *PesBrowser*-klass och gör ett anrop till metoden *getObjectTypes.* Den returnerar då en lista på olika definitioner som finns. Denna lista innehåller olika *response*- eller *request*-profiler. Efter det anropar den *getObjectDefintions* med den specifika kontexten man vill ha. I detta fallet kör den på en enkel blank *response*-profil vilket innebär att datan kan vara lite vad som helst och behöver inte följa ett mönster. Processen anropar då *IOperationsDefintions* som är ett gränssnitt som skickar anropet vidare till en *Get* operation eftersom det är kontexten som den fått. Den returnerar då en lista med lagliga GET-operationer som *PesConnectorn* sedan tar emot och visar åt användaren, se figur 25.



Figur 25. Flödet för att få en specifik objekt-definition

3.7.3 Flödet för *execute request*

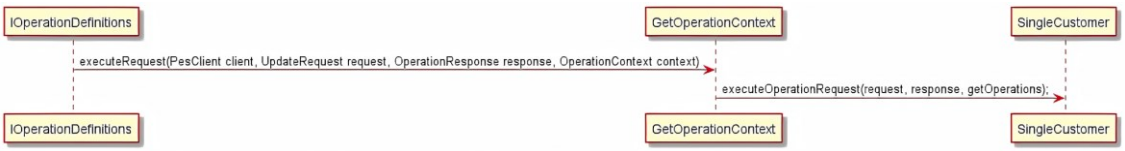
Inne i *PesExecuteOperation,* se figur 26, börjar processen med att skapa en *PesClient* som den behöver för att anropa servern. Sedan skapar den en *OperationSupplier* klass och anropar metoden *supplyDefinitions*. Den går igenom en lista med definierade kontexter och väljer ut den som matchar med strängen som skickades med i anropet. *OperationSupplier* returnerar kontexten processen vill ha som den skickar vidare till *IOperationDefintions*.



Figur 26. Flödet för *pesExecuteOperation*

IOperationDefintions skickar anropet vidare till rätt kontext som i detta fallet blir *GetOperationContext* som sköter om alla *GET*-operationer. Inne i *GetOperationContext* kommer processen att ta ett val ifall det är en eller flera kunder eller hierarkier man vill hämta ut. Efter att det blivit bestämt skapas *SingleCustomer*-klassen och anropar metoden *executeOperationRequest*.

Inne i *SingleCustomer* har man reflektions-funktionen som kollar vad processen har för metod som den vill köra och vad för *response* den kommer att ha. Efter det anropas metoden och processen körs till sitt slut, se figur 27.



Figur 27. Flödet för *IOperationDefinitions*

3.8 Hur reflektion blev använt

I figur 28 finns *operationMethod* som används för att få ut den riktiga metoden som processen sedan anropar. Som i exemplet i kapitlet "Vad är Java-reflektion" går processen genom samma sak här. Genom att hitta en metod baserat på vår *operationMethod* och sedan anropar funktionen med hjälp av *invoke*-kommandot. Detta ger oss möjlighet att anropa alla funktioner som blivit definierade av proto-filerna på samma ställe förutom två specialfall.

```
public void executeBatchOperationRequest(UpdateRequest request, OperationResponse response, String operationMethod) {
    Object objResponse;
    Class[] parameterTypes = new Class[1];
    parameterTypes[0] = Customermessages.CustomerHierarchyBatch.class;

    for (ObjectData data : request) {
        InputStream inputData = data.getData();
        String input = new BufferedReader(
                new InputStreamReader(inputData, StandardCharsets.UTF_8))
                .lines()
                .collect(Collectors.joining( delimiter: "\n"));
        Customermessages.CustomerHierarchyBatch customerHierarchyBatch = customerHierarchyMapper(input);
        try {
            Method method = this.client.getClass().getMethod(operationMethod, parameterTypes);
            objResponse = method.invoke(this.client, customerHierarchyBatch);
            Common.DefaultResponse customerDefRes = (Common.DefaultResponse) objResponse;
            handleResponse(response, data, JsonFormat.printer().print(customerDefRes), customerDefRes.getResponseRows( index: 0));
        } catch (Exception e) {
            throw new ConnectorException("Something went wrong: " + e.getMessage());
        }
    }
}
```

Figur 28. Implementationen av *executeBatchOperationRequest*

3.9 Ut- och indata

3.9.1 Formatering och validering av indata

Formateringen av indata gjordes på så sätt att för parti-operationer skall indata vara i en "JSON"-räcka, se figur 29. Indata för enskilda operationer sätter man i ett konfigurations-fält i Boomi.

```
[{
    "code": "13371",
    "name": "Tobys test",
    "description": "Test",
    "status": "ACTIVE"
  }, {
    "code": "13372",
    "name": "Tommys test Updated",
    "description": "Tommy test description",
    "status": "ACTIVE"
  }, {
    "code": "13373",
    "name": "Jeremias test updated",
    "description": "Jeremias test description",
    "status": "ACTIVE"
  }]
```


Figur 29. In data i "JSON"-format

Valideringen sköts med några hjälp funktioner i *Utils*-klassen. *Utils*-klassen innehåller alla funktioner som används av flera filer eller klasser.

Det finns två saker man behöver beakta när man validerar indata: Det första är ifall *code* är tomt skall det automatiskt misslyckas. Det andra är ifall man försöker skapa en ny kund eller en ny hierarki så får inte namnet vara tomt. Se figur 30 för implementationen.

```
private static boolean validate(String code, String name, OperationType type) {
    if (code == null && code.isEmpty()) {
        throw new ConnectorException("Customer/Hierarchy code cannot be null or empty");
    }
    if (type == OperationType.CREATE) {
        if (name == null && name.isEmpty()) {
            throw new ConnectorException("Customer/Hierarchy code cannot be null or empty");
        }
    }
    return true;
}
```

Figur 30. Implementationen av *validate*

3.9.2 Formatering av utdata

Formateringen av utdata sköts av *Utils*-klassen i den klassen gjordes en *handleResponse* som skall var så generell att man kan skicka in vilket response som helst och kunna skriva ut det. Det gjorde att man behövde formatera data före processen skickar in den till *handleResponse*. Datan är så pass enkel att en "JSON"-formaterare kan hantera det, se figur 31.

```
try {
    Method method = this.client.getClass().getMethod(operationMethod, parameterTypes);
    objResponse = method.invoke(this.client, customerBatch);
    Common.DefaultResponse customerDefRes = (Common.DefaultResponse) objResponse;
    handleResponse(response, data, JsonFormat.printer().print(customerDefRes), customerDefRes.getResponseRows( index: 0));
} catch (Exception e) {
    throw new ConnectorException("Something went wrong: " + e.getMessage());
}
```

Figur 31. *HandleRespone*

Väl inne i *handleResponse,* se figur 32, kollar processen ifall det uppstått något fel. ifall det har det anropar processen metoden *addFailure*[27]. Ifall det inte är ett fel anropas metoden *addSuccess*[28]. Metoderna *addFailure* och *addSuccess* gör i princip samma sak bara att en returnerar ett rött meddelande medan andra returnerar ett grönt.

```
public static void handleResponse(OperationResponse response, ObjectData data, String customerRes,
                                  Common.ResponseRow responseStatus) {
    if (responseStatus.getResponseType() == Common.ResponseMessageType.ERROR
            || responseStatus.getResponseType() == Common.ResponseMessageType.WARNING) {
        ResponseUtil.addFailure(
                response,
                data,
                responseStatus.toString(),
                ResponseUtil.toPayload(customerRes));
    } else {
        ResponseUtil.addSuccess(
                response,
                data,
                responseStatus.toString(),
                ResponseUtil.toPayload(customerRes));
    }
}
}
```

Figur 32. Implementationen av *handleResponse*

[27]https://boomisdkjavadoc.s3.amazonaws.com/javadoc/2.12.0/com/boomi/connector/api/ResponseUtil.html#addFailure-com.boomi.connector.api.OperationResponse-com.boomi.connector.api.TrackedData-java.lang.String-

[28]https://boomisdkjavadoc.s3.amazonaws.com/javadoc/2.12.0/com/boomi/connector/api/ResponseUtil.html#addSuccess-com.boomi.connector.api.OperationResponse-com.boomi.connector.api.TrackedData-java.lang.String-

4. SLUTSATSER

4.1 Resultat

Målet med detta examensarbete var att implementera en anslutningshaterare till plattformen Boomi som passade in på kundens beställning. Kunden var nöjd med demon av projektet, som genomfördes inom tidsramen för projektet.

Enligt kravspecifikationen har jag implementerat en anslutingshanterare till platformen Boomi som kan dra nytta av gRPC och anropa alla specificerade funktionerna. Anslutnings-hanteraren kan även själv formatera in- och utdata.

En punkt som fattas från kravspecifikationen är ett SSL-certifikat som jag inte fått tag i ännu. Detta görs av en tredje part så kunden väntar ännu på deras sista steg för att lyckas med projektet.

4.2 Reflektioner

Detta har varit ett intressant projekt att få utveckla. En stor del av dokumentationen är rätt så dålig ännu från Boomis sida. Stora delar av detta projekt har gått ut på *evolutionär prototyping* även kallat testa sig fram och se vad som fungerar. Jag har lärt mig mycket om anslutningshateraren, Boomi, gRPC och utveckling för ett företag. Detta projekt skulle kunna bli gjort på så många sätt. Man skulle helt kunna skippa reflektionen av funktionerna eller reflektera de genererade funktionerna från gRPC istället för att ha en mellanman för att hantera anropen.

Det som skulle vara intressant är ifall man istället för att behöva ha proto-filerna i projektet så skulle man kunna ha dem i Boomi där man laddar upp en proto-fil och så kompileras det av connectorn och ger ut de funktioner som finns specificerade i proto-filen.

Boomi har varit ett intressant verktyg att jobba med och det är otroligt enkelt att skapa smidiga dataförflyttningar eftersom man bara ritar upp en process som man vill att skall flytta filer från system a till b.
Normalt sköts såna överflyttningar för hand eller via något script som gjorts i något programmeringsspråk vilket gör överskådligheten mycket svårare än en uppritad process.
Dock så finns det ännu mycket brister i Boomis dokumentation och det finns inte så mycket stöd för mera avancerade funktioner som t.ex om man vill ändra en fils format så måste man göra det med ett script.

KÄLLFÖRTECKNING

About us - gambit. (2019, January 29). https://www.gambitgroup.fi/about-us/

Archiveddocs. (n.d.). *Data Transfer Object*. Retrieved November 16, 2021, from https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649585(v=pandp.10)?redirectedfrom=MSDN

Bagci, H., & Kara, A. (2016). A lightweight and high performance remote procedure call framework for cross platform communication. *International Conference on Software Engineering and Applications*, *2*, 117–124. https://pdfs.semanticscholar.org/b1f9/1eb075b0a17ad26d1b2bafa0229daf1091e0.pdf

Birrell, A. D., & Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems*, *2*(1), 39–59. https://doi.org/10.1145/2080.357392

Blom, S., Book, M., Gruhn, V., Hrushchak, R., & Köhler, A. (2008). *Write Once, Run Anywhere*. https://ul.qucosa.de/landing-page/?tx_dlf[id]=https%3A%2F%2Ful.qucosa.de%2Fapi%2Fqucosa%253A32152%2Fmets

Boomi AtomSphere Documentation. (n.d.). Retrieved November 16, 2021, from https://help.boomi.com/bundle/connectors

Boomi iPaaS solutions & tools for cloud connected business. (2019, March 22). https://boomi.com/

Boomiverse. (2021, March 21). https://train.boomi.com/login

Core concepts, architecture and lifecycle. (n.d.). Retrieved December 2, 2021, from https://grpc.io/docs/what-is-grpc/core-concepts/

Dignan, L. (2019, December 17). *Boomi acquires Unifi Software, aims to meld data catalog, discovery with its integration platform*. ZDNet. https://www.zdnet.com/article/boomi-acquires-unifi-software-aims-to-meld-data-catalog-discovery-with-its-integration-platform/

Gradle Build Tool. (n.d.). Retrieved November 16, 2021, from https://gradle.org/

Hekmatpour, S. (1987). Experience with evolutionary prototyping in a large software project. *SIGSOFT Softw. Eng. Notes*, *12*(1), 38–41. https://doi.org/10.1145/24574.24577

hjp: doc: RFC 1945: Hypertext Transfer Protocol -- HTTP/1.0. (n.d.). Retrieved November 30, 2021, from https://www.hjp.at/doc/rfc/rfc1945.html

Indrasiri, K., & Kuruppu, D. (2020). *gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes*. "O'Reilly Media, Inc." https://play.google.com/store/books/details?id=8c3LDwAAQBAJ

Kálmán, M. (2013). ProtoML: A rule-based validation language for Google Protocol Buffers. *8th International Conference for Internet Technology and Secured Transactions (ICITST-2013)*, 188–193. https://doi.org/10.1109/ICITST.2013.6750189

Li, Y., Tan, T., & Xue, J. (2019). Understanding and Analyzing Java Reflection. *ACM Transactions on Software Engineering and Methodology*, *28*(2), 1–50. https://doi.org/10.1145/3295739

Lopez, E. (2007). *Introduction to Java*. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States). https://www.osti.gov/servlets/purl/1724140

Metz, C. (2011, April 14). *Meet Dell, your internet service provider*. The Register. https://www.theregister.com/2011/04/14/dell_and_boomi/

Mu, H., & Jiang, S. (2011). Design patterns in software development. *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, 322–325. https://doi.org/10.1109/ICSESS.2011.5982228

What is Gradle? (n.d.). Retrieved December 8, 2021, from https://docs.gradle.org/current/userguide/what_is_gradle.html

BILAGOR/APPENDICES

API	Application Programming Interface[29]
HTTP	Hypertext Transfer Protocol[30]
FTP	File Transfer Protocol[31]
DISK	Disk v2 connector[32]
XML	Extensible Markup Language[33]
WORA	Write once, run anywhere[34]
IDE	Integrated development environment[35]
Endpoint	A function or procedure call that is part of an API in software engineering[36]
Request	A request to an API[37]
Response	A response from an API[38]
RPC	Remote Procedure Call[39]
JSON	JavaScript Object Notation[40]
IDE	Integrated development environment[41]

[29] https://en.wikipedia.org/wiki/API
[30] https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[31] https://en.wikipedia.org/wiki/File_Transfer_Protocol
[32] https://help.boomi.com/bundle/connectors/page/int-Disk_v2_connector.html
[33] https://en.wikipedia.org/wiki/XML
[34] https://en.wikipedia.org/wiki/Write_once,_run_anywhere
[35] https://en.wikipedia.org/wiki/Integrated_development_environment
[36] https://en.wikipedia.org/wiki/Endpoint
[37] https://en.wikipedia.org/wiki/API
[38] https://en.wikipedia.org/wiki/API
[39] https://en.wikipedia.org/wiki/Remote_procedure_call
[40] https://en.wikipedia.org/wiki/JSON
[41] https://en.wikipedia.org/wiki/Integrated_development_environment