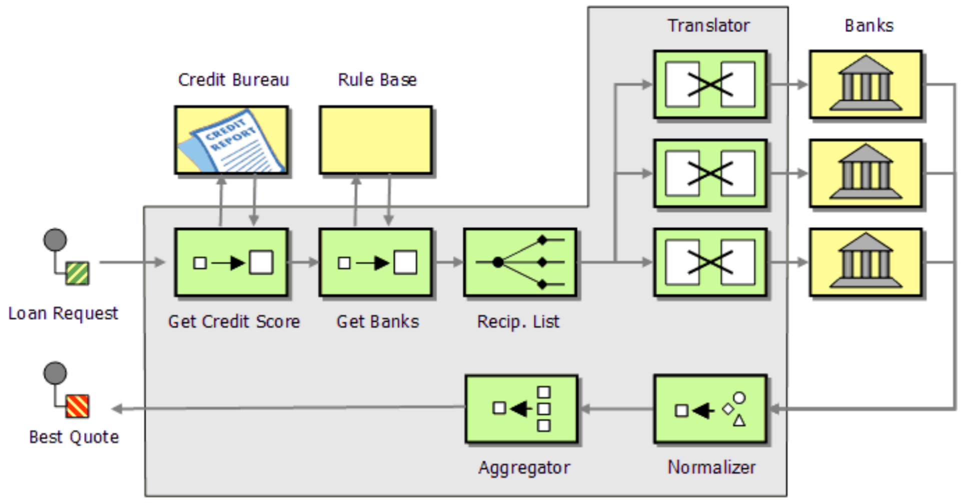
Anders Geer Jakobsen & Mohammed Joseph Salameh

Project: Loan Broker

Introduction

The project assigned involved implementing a Loan broker, capable of responding with a best quote based on a Loan Request from a customer. The project consists of four key components:



(Picture 1: Loan Broker)

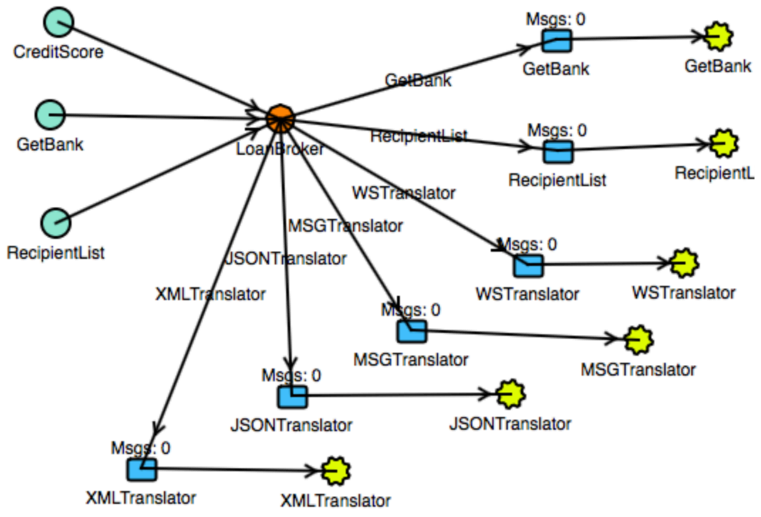
- Consumer/Customer
- Loan Broker (Grey Box: Our main implementation)
- Credit Bureau (already implemented WS)
- Banks (4; 2 of them already implemented JSON/XML through RabbitMQ)

Procedure (In Short)

The *Consumer/Customer* makes requests for loan quotes and receives a quote based on his Credit Score. The credit score is determined by the *Credit Bureau*, that provides the Loan Broker a service that determines the *Consumers/Customers* credit score based on his/hers Social Security number(SSN). And based on that Credit Score, the *Loan Broker* (which acts as the central process manager that coordinates message' and processes between the different components (grey box)) determines the best quote and submits it to each Bank that computes an interest rate according to the loan parameters and ultimately returns the result back to the consumer.

Design Overview

The loan broker itself is a web service, that is divided into messaging components that will be accessed through a simple web site. Our main infrastructure design and communication



(Picture 2: Infrastructure Messaging)

Is based on one Exchange *LoanBroker* for internal communication between components. It is implemented as a Direct, where its routing is based on queues whose “binding key” exactly matches the “routing key” of the message. In such setup, whenever *CreditScore* publishes a message to the exchange *LoanBroker* with a “routing key” GetBank, it will be routed to queue *GetBank*. Now *GetBank* should also be able to publish a message to *RecipientList* with its “routing key” being RecipientList and so on.

Bottleneck

Performance is always a huge factor in most modern systems as everyone wants a fast and smooth system. For various reasons a system might have potential negatives on performance including security, network, heavy calculations etc.

For our loan broker system, the potential performance issues will specifically be towards bottlenecks. Our system is designed in a way that allows a load balancer to start more than one version of a subsystem, allowing heavy traffic to be spread out across the several components. The only one where this is currently not possible is the Aggregator, which saves the replies for a while and compares them to a local Map. This creates an obvious bottleneck.

Another possible bottleneck is our choice to use a single exchange for all our internal messages, however we don't think this will create a problem as the exchanges can handle a lot of messages at once.

Other possible bottlenecks, that we do not have control over, include web services that aren't load balanced along with banks being flooded with messages. Two of the banks are out of our control to balance, but do use messaging and can possibly spin up several instances to clear a queue faster. Our two implemented banks are implemented as a web service and the other through messaging, our messaging bank can as well be spun up to create several instances.

One of our bottlenecks seems to be our Aggregator, which we can solve by applying a load balancer to the system.

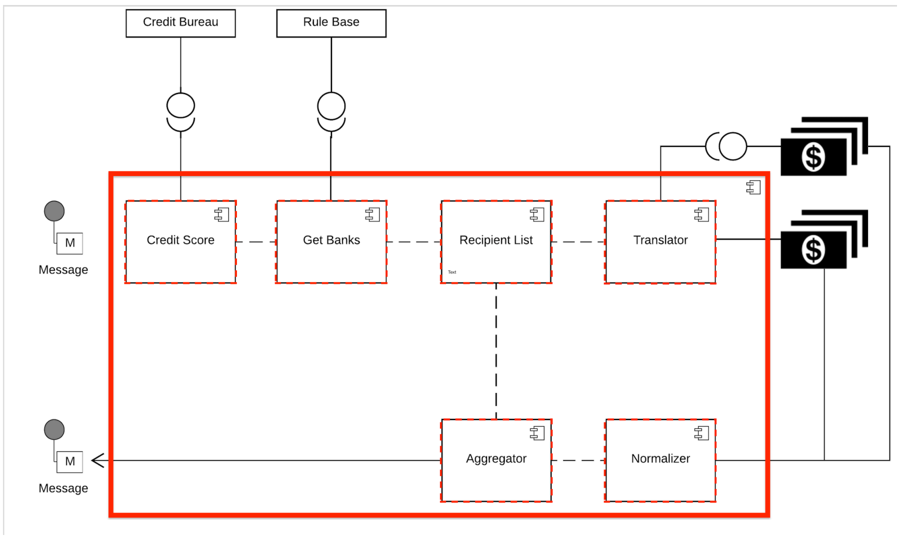
1. Based on date of birth, i.e. 30 servers running at once, each taking the date of birth in the month.
   (1st server takes every person who has a birthdate on the 01-01-2016,
   2nd server takes every person who has a birthdate on the 02-01-2016 and so on)
   This doesn't necessarily balance the load as in theory everyone born on the 1st could all send a request on the same day maximizing the single server with the remaining being empty.
2. A similar system based on month, requires less servers, meaning less possibility of all but one being idle, but still allows everyone from a single month to make requests at once.
3. Web servers of popular websites, throws the request to the idlest one, so that other server do not get overloaded. This however gives us a similar problem as spinning up a new one does, in which we might send a response to the "wrong" server that doesn't store the data for that single customer.

Essentially it is hard to create multiple instances of our aggregator, since our references between request are not taken into account.

The best solution seems to be either a combination of the above or the load balancer doing the third option, but has to know where previous SSN where sent. This however is also a problem as the banks are sending different lengths of social security numbers back (json sending 10 and xml sending 8 as per description). This seems to be an implementation error from the BANK.

Component Diagram

The general idea behind the Loan Broker(LB) System, aside from the fact it needs to return a "Loan Quote" to the Customer, is that the LB should be divided into different components where neither of them are related or dependent on each other. Meaning that a functionality in our case "Loan Quote", should be divided into separate projects as well as obeying the Service Oriented Architecture(SOA), also defined a bit later. The component diagram can be seen below:



(Picture 2: Loan Broker Component Diagram)

The diagram above has some similarities from Picture 1, although this diagram shows how we are going to split our projects into separate components. The small components within the big Red rectangle is the Sub-systems of the "Loan Broker" which is the big solid Red rectangle. This also clarifies that the Loan Broker itself is a Component and 6 other components are running within it.

Loan Broker flow process

Now that the general picture has been described, a step by step definition is needed to fully understand the whole process (the order followed is the whole process):

Message: Is the Customer/client that inputs the necessary info, could be from a website or application.

Credit Score: is implemented with the enricher pattern in mind, meaning that the information gathered from the Customer/Client, which in our case holds SSN, LoanAmount and LoanDuration needs to be formatted into a Class Object, that then gets enriched with the Credit Score from the

Credit Bureau web service, which is displayed as a lollipop notation. Meaning that *Credit Bureau* is used by Credit Score*.*

Get Banks: is also implemented as an enricher, but in this case the information needed first is the Date of the Loan Request, so that we currently have SSN, LoanAmount, (int)LoanDuration, CreditScore and a LoanDuration as an Actual date. Based on that information Get Banks uses the Rule Base Web service which decides, with specific rules, which banks to direct the Loan request to. The information retrieved back from *Rule base* is also enriched in the object.

RecipientList: is implemented with the Recipient List pattern in mind, meaning that incoming messages from Get Banks, needs to hold values on which Banks to route the Loan Request to. And based on that list, the RecipientList determines the desired recipients and forwards the messages to all channels associated with it. There is no modification of the message contents.

Translator: is implemented as a Message Translator pattern. It provides the systems (banks) using different data formats to be able to communicate with each other. This component is responsible for translating the data from RecipientList to the Bank (Depending on which one JSON/XML), so that the bank can understand the data format and compute and return a Loan Quote. E.g. In the case of our Web Service Bank (lollipop notation from Translator to Bank in picture 2) needing information, the format it needs to understand is of format XML.

Normalizer: is implemented as a Normalizer. This component, needs to detect the type of the incoming message provided by the Banks. The information provided from whichever as a XML or JSON format, our Normalizer needs to format the data received to a common format.

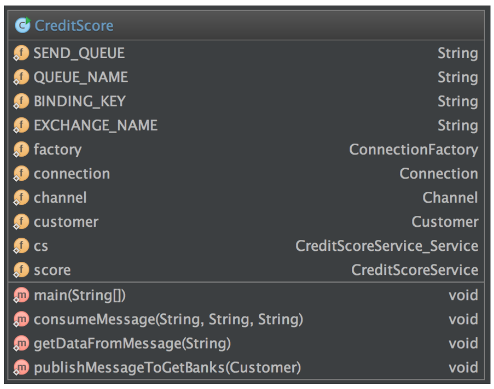
Aggregator: is implemented as an Aggregator, and is responsible for collecting and storing individual messages until a set of related messages has been received. This means that the information received from the Normalizer does not guarantee a message to be a whole message, but rather multiple individual messages that could, together, be a whole message.

Object transferal

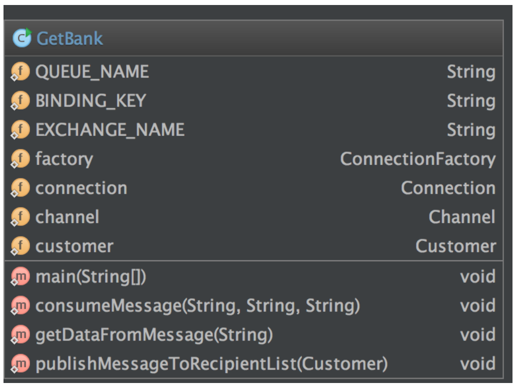
Within our internal system, we send the info around as JSON Strings with many similarities in most components i.e. ID, SSN, LoanAmount, LoanDuration. However, from normalizer to aggregator the above-mentioned values does not hold, since they banks does not guarantee the same values. In our case the banks return both SSN and a InterestRate, hence Normalizer sends SSN and InterestRate further to Aggregator. These strings gets enriched with data throughout most of the early components in our component diagram (CreditScore, GetBanks). Some other subsystems have special requirements of the JSON input, such as the recipient list component, which also takes 4 boolean parameters for each of our currently implemented banks. These booleans essentially decide whether to send to what bank and would be essential to lowering any excess messages to a bank specializing in high-end customers. For information about input restrictions check Input restrictions on page.

Class & Sequence Diagrams

A Design Class Diagram, is typically a redefined domain model, that shows real-world classes (conceptual classes), in our case we used a Component Diagram(CD) to help visualize our composite project design, check Component Diagram Chapter.



CreditScore	
SEND_QUEUE	String
QUEUE_NAME	String
BINDING_KEY	String
EXCHANGE_NAME	String
factory	ConnectionFactory
connection	Connection
channel	Channel
customer	Customer
cs	CreditScoreService_Service
score	CreditScoreService
main(String[])	void
consumeMessage(String, String, String)	void
getDataFromMessage(String)	void
publishMessageToGetBanks(Customer)	void



GetBank	
QUEUE_NAME	String
BINDING_KEY	String
EXCHANGE_NAME	String
factory	ConnectionFactory
connection	Connection
channel	Channel
customer	Customer
main(String[])	void
consumeMessage(String, String, String)	void
getDataFromMessage(String)	void
publishMessageToRecipientList(Customer)	void

(CreditScore and GetBank: Design Class Diagrams)

Now with the above diagrams in mind, please be aware that our Projects and our Classes are named the same. The class's seen above have a main class that runs everything. Most of our classes consists of the same variables and methods:

- Variables
  - SEND_QUEUE
  - QUEUE_NAME
  - BINDING_KEY
  - EXCHANGE_NAME
  - Customer (class customer that holds the value)
- Methods
  - consumeMessage(String exchangeName, String queueName, String routingKey)
  - getDateFromMessage(String message)
  - publishMessageToGetBank(Customer customer)

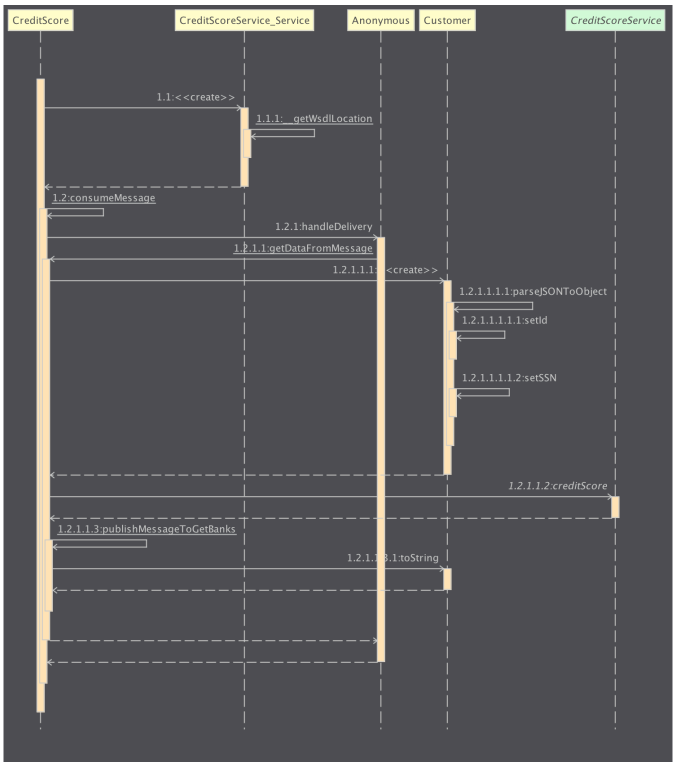
This could have been done better in different ways:

- Create an interface as a contract that requires every class to implement those methods in their own way. This is not the best solution, but there are some cases where our Normalizer, Aggregator needs to listen to more than 1 queue making them behave differently according to the data type received.

- Create an abstract class that can be inherited from each class that needs to handle the messaging part. This is a really good solution since the code shared between every project is closely related, but some needs something extra.

(Check rest of the projects Class Diagrams in picture 'Class Diagrams' provided in the zip)

Sequence Diagram



(Sequence diagram: CreditScore)

The Sequence diagram above, describes the interaction between different objects of a process, whenever the CreditScore class gets started, a process that flows through the above steps is done every time. This component CreditScore, gets in touch with a CreditScoreService which is the *Credit Bureau* mentioned earlier in this report (see picture 1), the reason it 1.2.1.2 overlaps 1.2.1 is because that the method handle delivery is asynchronous.

Most of the classes and projects follow the above diagram, first with a Connection to the messaging system(RabbitMQ) that handles the received message from previous component. Sequence diagrams for the other classes can be found in 'UML Sequence Diagrams' in the provided ZIP.

Testability

With testability in mind we tried to not deviate to much from our existing conceptual design and it proved to be unsuccessful, testability in these kinds of system essentially requires you to isolate the subsystem to allow testing it without the possibility of errors out of your control, such as errors in regards to delivery on the internet or error caused by other systems you use. For this kind of testing it’s important that you make an object very similar to your expected design to, and possibly in this case a Message object which would have similar properties as the AMQP messages allowing the sending of that instead. This allows you to test the body of that message along with any headers or properties. Along with this kind of no network test, it is always beneficial to do unit tests in which “the usual rules” apply. Test everything that is not auto generated and test on limits in particular.

- Test just below a limit
- Test the limit
- Test just above the limit

As a possible test, we could have tested our incoming JSON message, along with the data the keys hold, and if they match our business protocols. In that way, we make sure that all the keys for the information we need in a subsystem holds.

Input restrictions

Loan broker description

For our project, we have implemented our loan broker as a web service. This web service has a single method called LoanRequest with three parameters

1. Social security number as a string (DDMMYY-XXXX)
2. loanAmount as a positive integer
3. loanDuration in month as a positive integer.

If the above requirements aren't met, the command line will simply answer that something went wrong.

Since the current implementation is purely as a quick small program sending the request, all changes to the request has to be made through that, this should also restrict the use of values over int.max in most modern IDEs.

Discussion of implementation

Coupling

Our implementation is very loosely coupled as every subsystem essentially knows nothing about the other. All they know about is an exchange which acts as message broker for message between our subsystems, this means could be visualized in a way that every system is connected to one thing in the middle of them all, which handles everything sent between them with a couple of them knowing more than one to receive from our internal system and send messages externally. The low coupling is ensuring by every system knowing nothing about the workings of other parts of the system, as they are entirely focused on their own part of the full systems entire operations.

Cohesion

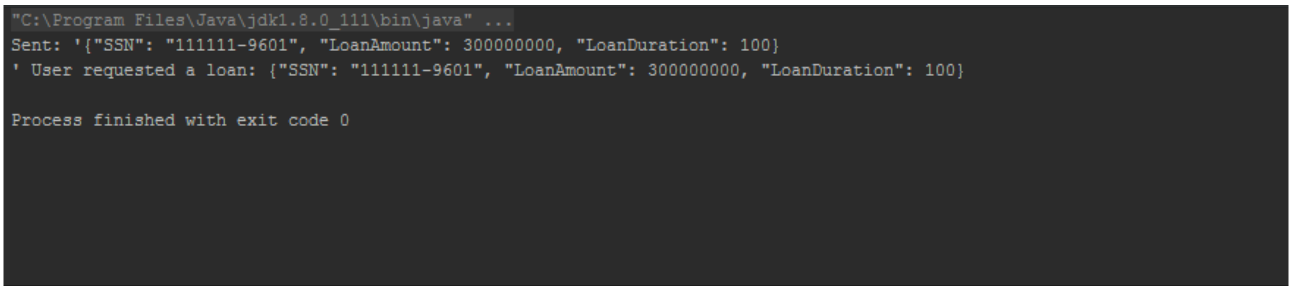
While arguably higher cohesion could be achieved, we feel like we have relatively high cohesion, which could have possibly been improved by having all our very similar methods put into a class and inherited from it, we however felt like we wouldn't be able to get console output like we are now, had we kept it in one super class

Granularity

We have use different levels granularity. Internally we haven't rated it too highly, and in our web services we are using a fine-grained granularity. And considering this we probably should have used a more coarse-grained granularity for this, as we could have done it easier for ourselves this way. Having coarse granularity possibly adds a bit more work to the entire process of creating our services, as we then should define our objects to transfer etc. On our bank, we have somewhat of a combination as we take a single xml string, which however contains every piece of info on our customer, this would have been possible to do in our other services as well and would likely have been better as opposed to our current solution with our rule base web service taking 4 parameters.

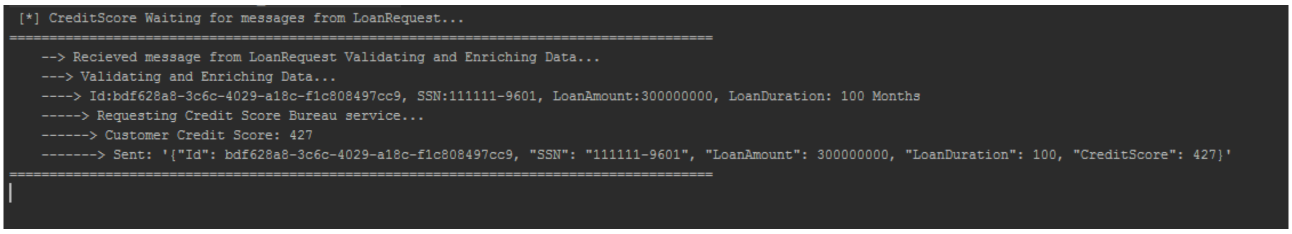
Application flow

1. Credit score



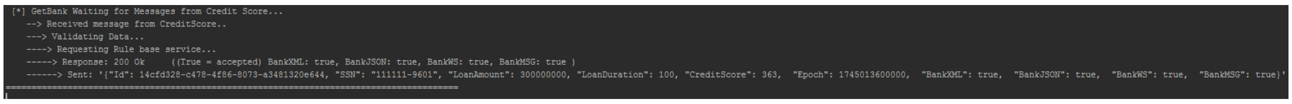
```
"C:\Program Files\Java\jdk1.8.0_111\bin\java" ...
Sent: '{"SSN": "111111-9601", "LoanAmount": 300000000, "LoanDuration": 100}
' User requested a loan: {"SSN": "111111-9601", "LoanAmount": 300000000, "LoanDuration": 100}

Process finished with exit code 0
```

2. Credit score



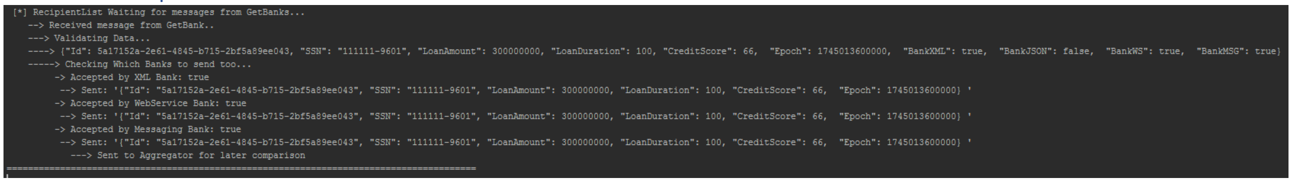
```
 [*] CreditScore Waiting for messages from LoanRequest...
=========================================================================================
    --> Recieved message from LoanRequest Validating and Enriching Data...
    ---> Validating and Enriching Data...
    ----> Id:bdf628a8-3c6c-4029-a18c-f1c808497cc9, SSN:111111-9601, LoanAmount:300000000, LoanDuration: 100 Months
    -----> Requesting Credit Score Bureau service...
    ------> Customer Credit Score: 427
    -------> Sent: '{"Id": bdf628a8-3c6c-4029-a18c-f1c808497cc9, "SSN": "111111-9601", "LoanAmount": 300000000, "LoanDuration": 100, "CreditScore": 427}'
=========================================================================================
|
```

3. Get Bank



```
 [*] GetBank Waiting for Messages from Credit Score...
    --> Received message from CreditScore..
    ---> Validating Data...
    ----> Requesting Rule base service...
    -----> Response: 200 Ok     ((True = accepted) BankXML: true, BankJSON: true, BankWS: true, BankMSG: true )
    ------> Sent: '{"Id": 14cfd328-c478-4f86-8073-a3481320e644, "SSN": "111111-9601", "LoanAmount": 300000000, "LoanDuration": 100, "CreditScore": 363,  "Epoch": 1745013600000,  "BankXML": true,  "BankJSON": true,  "BankWS": true,  "BankMSG": true}'
=========================================================================================
|
```

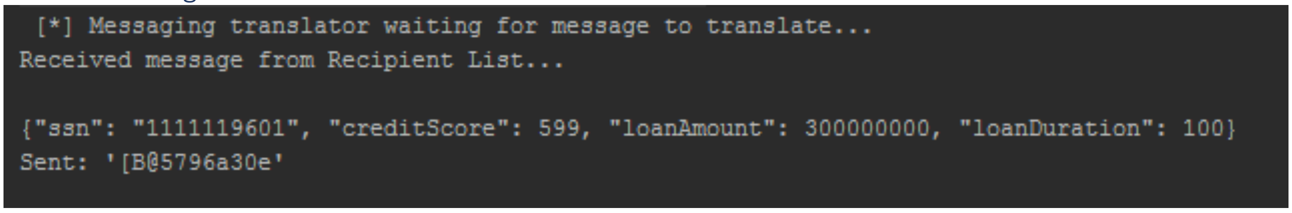
4. Recipient List



```
 [*] RecipientList Waiting for messages from GetBanks...
    --> Received message from GetBank..
    ---> Validating Data...
    ----> {"Id": 5a17152a-2e61-4845-b715-2bf5a89ee043, "SSN": "111111-9601", "LoanAmount": 300000000, "LoanDuration": 100, "CreditScore": 66,  "Epoch": 1745013600000,  "BankXML": true,  "BankJSON": false,  "BankWS": true,  "BankMSG": true}
    -----> Checking Which Banks to send too...
         -> Accepted by XML Bank: true
          --> Sent: '{"Id": "5a17152a-2e61-4845-b715-2bf5a89ee043", "SSN": "111111-9601", "LoanAmount": 300000000, "LoanDuration": 100, "CreditScore": 66,  "Epoch": 1745013600000} '
         -> Accepted by WebService Bank: true
          --> Sent: '{"Id": "5a17152a-2e61-4845-b715-2bf5a89ee043", "SSN": "111111-9601", "LoanAmount": 300000000, "LoanDuration": 100, "CreditScore": 66,  "Epoch": 1745013600000} '
         -> Accepted by Messaging Bank: true
          --> Sent: '{"Id": "5a17152a-2e61-4845-b715-2bf5a89ee043", "SSN": "111111-9601", "LoanAmount": 300000000, "LoanDuration": 100, "CreditScore": 66,  "Epoch": 1745013600000} '
            ---> Sent to Aggregator for later comparison
=========================================================================================
|
```

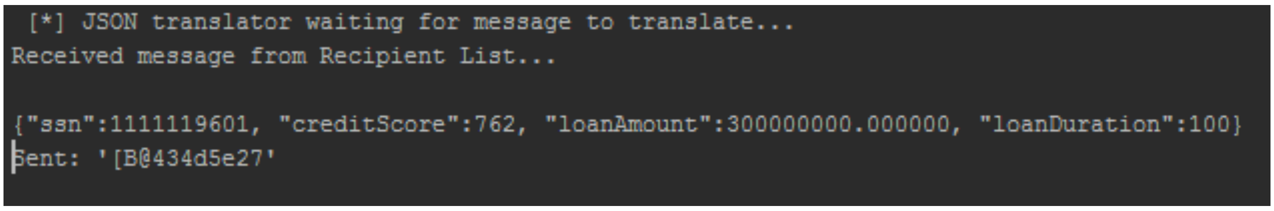
5. Translators

a. Message translator



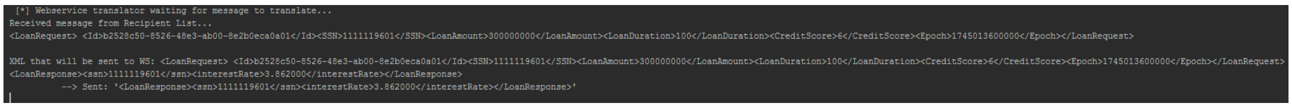
```
  [*] Messaging translator waiting for message to translate...
 Received message from Recipient List...

 {"ssn": "1111119601", "creditScore": 599, "loanAmount": 300000000, "loanDuration": 100}
 Sent: '[B@5796a30e'
```

b. Json translator



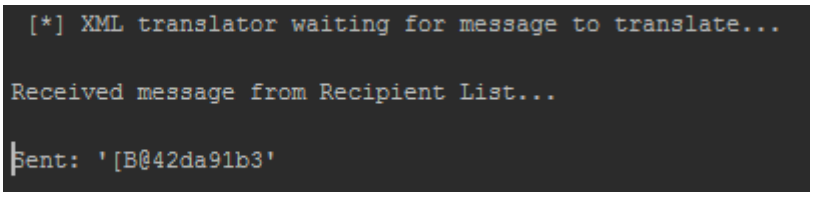
```
 [*] JSON translator waiting for message to translate...
Received message from Recipient List...

{"ssn":1111119601, "creditScore":762, "loanAmount":300000000.000000, "loanDuration":100}
Sent: '[B@434d5e27'
```

c. WS translator



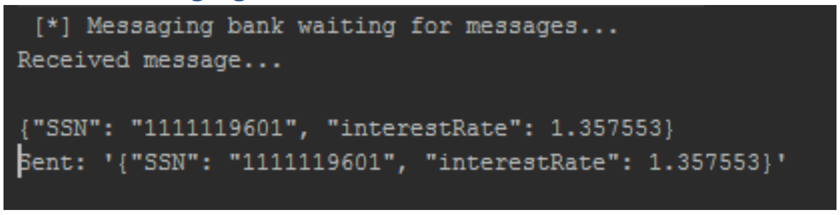
```
 [*] Webservice translator waiting for message to translate...
Received message from Recipient List...
<LoanRequest> <Id>b2528c50-8526-48e3-ab00-8e2b0eca0a01</Id><SSN>1111119601</SSN><LoanAmount>300000000</LoanAmount><LoanDuration>100</LoanDuration><CreditScore>6</CreditScore><Epoch>1745013600000</Epoch></LoanRequest>

XML that will be sent to WS: <LoanRequest> <Id>b2528c50-8526-48e3-ab00-8e2b0eca0a01</Id><SSN>1111119601</SSN><LoanAmount>300000000</LoanAmount><LoanDuration>100</LoanDuration><CreditScore>6</CreditScore><Epoch>1745013600000</Epoch></LoanRequest>
<LoanResponse><ssn>1111119601</ssn><interestRate>3.862000</interestRate></LoanResponse>
         --> Sent: '<LoanResponse><ssn>1111119601</ssn><interestRate>3.862000</interestRate></LoanResponse>'
```

d. Xml translator



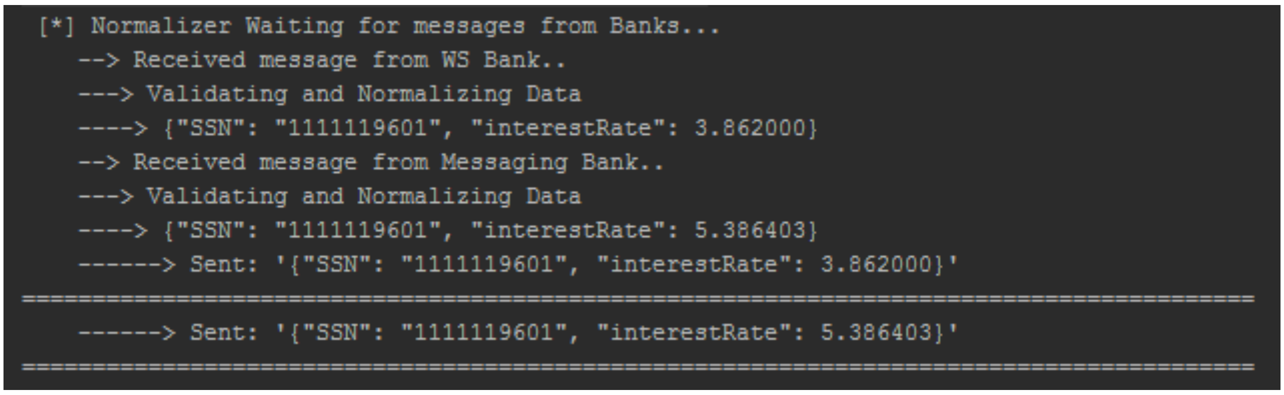
```
 [*] XML translator waiting for message to translate...

Received message from Recipient List...

Sent: '[B@42da91b3'
```

6. Messaging bank



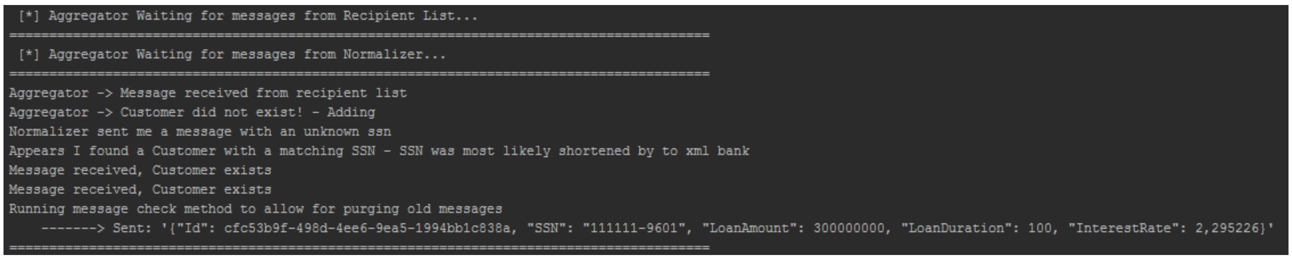
```
 [*] Messaging bank waiting for messages...
Received message...

{"SSN": "1111119601", "interestRate": 1.357553}
Sent: '{"SSN": "1111119601", "interestRate": 1.357553}'
```

7. Normalizer



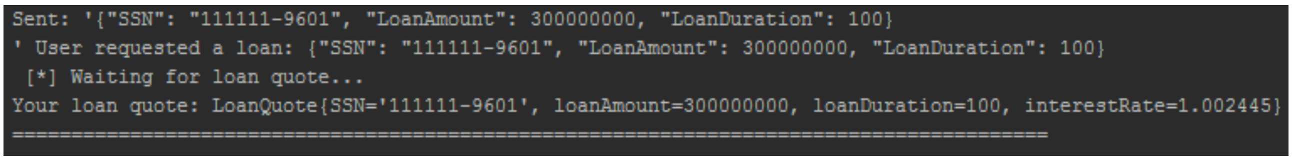
```
 [*] Normalizer Waiting for messages from Banks...
    --> Received message from WS Bank..
    ---> Validating and Normalizing Data
    ----> {"SSN": "1111119601", "interestRate": 3.862000}
    --> Received message from Messaging Bank..
    ---> Validating and Normalizing Data
    ----> {"SSN": "1111119601", "interestRate": 5.386403}
    ------> Sent: '{"SSN": "1111119601", "interestRate": 3.862000}'
==============================================================================================
    ------> Sent: '{"SSN": "1111119601", "interestRate": 5.386403}'
==============================================================================================
```

8. Aggregator



```
 [*] Aggregator Waiting for messages from Recipient List...
========================================================================================
 [*] Aggregator Waiting for messages from Normalizer...
========================================================================================
Aggregator -> Message received from recipient list
Aggregator -> Customer did not exist! - Adding
Normalizer sent me a message with an unknown ssn
Appears I found a Customer with a matching SSN - SSN was most likely shortened by to xml bank
Message received, Customer exists
Message received, Customer exists
Running message check method to allow for purging old messages
    -------> Sent: '{"Id": cfc53b9f-498d-4ee6-9ea5-1994bb1c838a, "SSN": "111111-9601", "LoanAmount": 300000000, "LoanDuration": 100, "InterestRate": 2,295226}'
========================================================================================
```

9. Client with answer



```
Sent: '{"SSN": "111111-9601", "LoanAmount": 300000000, "LoanDuration": 100}
' User requested a loan: {"SSN": "111111-9601", "LoanAmount": 300000000, "LoanDuration": 100}
 [*] Waiting for loan quote...
Your loan quote: LoanQuote{SSN='111111-9601', loanAmount=300000000, loanDuration=100, interestRate=1.002445}
========================================================================================
```