Preface

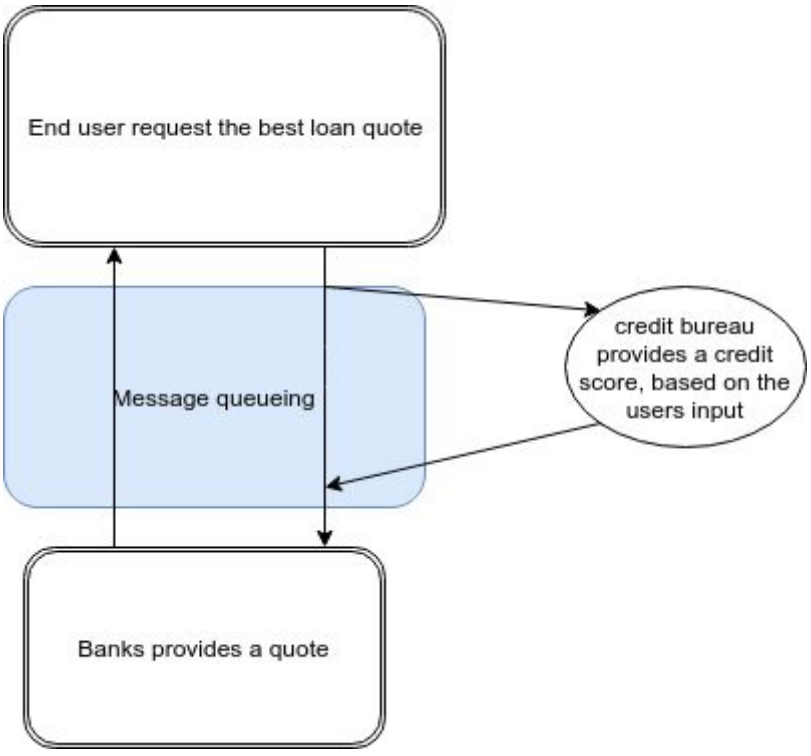
In this report we will describe our implementation of the *Loan broker* assignment.
We will explore the system in both a macroscopic perspective, where we draw the general boundaries of the system and describe it as whole, as well as on a microscopic level where we will go in depth with the individual subcomponent and subsystems that make up the entirety of our implementation.

You can try out our live system, hosted on digitalocean[1], and watch the flow of events over at http://37.139.5.194:3030/.
To make a loan request, you need to provide a social security number (on the form: xxxxxx-xxxx), and a loan amount (an integer) as well as a loan duration (also an integer).

Happy reading
- Marco Blum, Alexandar Kraunsøe, Christian Lind, David Samuelsen & Kasper Pagh.

Introduction to the system

Our *Loan broker* system, as a whole, is meant to allow for a user to request interest rates on a given loan from multiple different banks[2]. The user is then presented with the quote that has the best interest rate out of all the received replies.



[1] https://www.digitalocean.com/
[2] In our application, only banks offer loans.

Fig. 1. A very broad conceptual illustration of our system

All the different sub elements of the system pictured above (such as the credit bureau and banks) communicate their individual data in a certain format and via certain protocols. which in turn means that the system has to be able to handle a wide variety of non standardized inputs and outputs.

The system manages this feat, through the use of *message queueing*. Message queueing is a concept with a rich history, and have been utilized for many years.
These message queues act as the “glue” that tie together the various subsystems by providing the tools to handle and manage the different the various inputs and outputs.

Our implementation

We have decided to implement the entirety of our system in Javascript, including both the banks and the RabbitMQ system.

Specifically we have used Node.js, a Javascript runtime, based on google's V8 Javascript interpreter. Node allow us to easily separate the various functionalities, and keep the coupling of the system low, while the high number of individual scripts allow us to keep a high degree of cohesion.

We also used a number of libraries, we will briefly explain the function of each of our primary dependencies.

*Fig. 2. A screenshot from the **package.json** file which contains startup and dependency information.*

- **amqplib**: is a library that handles the connection to the various message queues we use in our application.
- **express**: a framework that allow us to easily create http server, without the need for trivial configuration.
- **bodyparser**: A piece of middleware that parses JSON in the body of incoming requests to the express server.

- **body-parser-xml**: Same as above, only with XML instead of JSON.
- **js2xmlparse**: Same as above, only from Javascript to XML in responses.
- **cors**: A piece of middleware that removes the limitations set in place by the Cross-Origin Resource Sharing, that would otherwise disallows browsers from interacting with the servers API.
- **pug**: Is a template engine that allow us to construct dynamic html files on the server according to various variables (which is then transmitted to the client for rendering).
- **SOAP**: A piece of middleware for creating SOAP endpoints and loading .wsdl files.
- **xml2js**: Another Javascript to XML parser.

The abovementioned tools allow us to structure our system as a series of distinct self contained subsystems, working together to achieve the goal of the system.

```
call pm2 start loanRequest.js -x -- Dev
call pm2 start getCreditScore.js -x -- Dev
call pm2 start rulebaseServer.js
call pm2 start getBanks.js -x -- Dev
call pm2 start soapBank1.js
call pm2 start rabbitMqBank.js -x -- Dev
call pm2 start normalizer.js -x -- Dev
call pm2 start Aggregator.js -x -- Dev
call pm2 start translatorJSONBank.js -x -- poor Dev
call pm2 start translatorXMLBank.js -x -- rich Dev
call pm2 start translatorSoapBank.js -x -- all Dev
call pm2 start translatorRabbitBank.js -x -- all Dev
```

*Fig 3. A screenshot of our **runeverything.bat** script, that starts all the module with pm2[3]. Note. this screenshot is from the development branch,in production we run the above scripts without the -dev argument*

This design philosophy means it is easy to change the various elements, without affecting the system as a whole.

[3] pm2 is a **production process manager** that includes a lot of tools including: load balancing, very fast reloading/restarting and various other features.

Brief description of subsystems

loanRequest.js:
This is the initial script, that sets up the express server. It has 2 api requests one for getting logs and results and one for posting a loan request, which gets queued to getCreditScore
getCreditScore.js:
This component sends the user's request to the credit score server and puts it with the prior request and sends it to getBanks.
getBanks.js:
This component takes the credit score and sends it to rulebaseserver, from which it gets the topic and publishes prior request to said topics. It also sends the aggregator the topics.
rulebaseServer.js:
Takes a credit score and returns which topics are matching.
translatorJSONBank.js:
This translator listens on the ***"poor"*** topic and converts the ssn to a 10 digit int instead of a string, it then sends it to Tine's JSON Bank.
translatorXMLBank.js:
This translator listens on the ***"rich"*** topic and converts the ssn to a 10 digit int instead of a string and the request to XML instead of JSON, it then sends it to Tine's XML Bank.
translatorSoapBank.js:
This translator listens on the ***"all"*** topic and converts the ssn to a 10 digit int instead of a string, it then uses SOAP to request the soapBank1 for a interest rate and forwards it to the normalizer
translatorRabbitBank.js:
This translator listens on the ***"all"*** topic and converts the ssn to a 10 digit int instead of a string, it then sends it to the Rabbit Bank.
soapBank1.js:
This is a bank that is created from SOAP, it receives a msg: {ssn, loanDuration, loanAmount, creditScore} and replies with a msg: {ssn, interestRate}
rabbitMqBanks.js:
This is a bank that is created with RabbitMQ, it receives a msg: {ssn, loanDuration, loanAmount, creditScore} and replies with a msg: {ssn, interestRate}
normalizer.js:
This component takes all the replies from the banks and converts them into a standadized msg form: {banq, ssn, interestRate}, where the banq is what bank it received it from.
Aggregator.js:
This component gets the topics from the getBanks, and calculates the assumed amount of responses it then then takes all the messages from the normalizer, and forwards the best one to the loanRequst server as the result.

Message Queue

For this system we use a various amounts of messaging modules, but what do these modules exactly do? In order to know how each module work first one must know what a message is. A message is just a way to package data so that you can send information from system to system that use messaging. To send these messages from system to system they need to be sent to a channel. A channel is a buffer zone for messages. Via messaging systems can read and write messages to channels. In order to not have messages mixed up, each message must be sent down a queue to the channel. It is from the queue that each system can read and write information to a channel. Now that we know how messages can be sent from system to system it's time to talk about how we can manipulate these messages.

Taking this project into account, we have a user who wants to make a loan. All this user needs to input is their ssn as well as the amount they want to loan and how long of a time they need to pay the money back. From this information alone you can not get a loan, as most banks want to know of a person's credit score. For this we use an enricher. An enricher takes information sent down the system, and then it appends information that it takes from an external source to the information. In other words it adds more data to the message from different source. Our enricher connects to the given credit bureau and receives the credit score for the given ssn. With the credit score we can then categorize whether or not the user is rich, poor, or middle class(all). We use another enricher to get the category based on the person's credit score from the SOAP rulebase service.

Now our message has ssn, loan, credit score, and a category. Now it's time to send the information to the banks. But wait each bank are dependent on the user's credit score so first we have to send our message through a recipient list. This recipient list takes the message and sends it to all banks based on the category of the message. Before the banks can read the message however we must send each of the messages through a translator. A translator formats the message so that the bank can read the message. We need a total of four translators as we need the message to be in XML, JSON, SOAP and RabbitMQ format.

Two of the banks are given to us. Those banks are set to be fan-out exchanges. A fan-out exchange is when an application receives a message and forwards the message further down the line to any and all applications looking for a fan-out exchange. The two banks we created ourselves are slightly different. The Soap bank is a SOAP service similar to the creditscore bureau given to us for use. The RabbitMQ bank takes the message and sends a response down to a replyto queue. This replyto queue sends the information further to the next step in the messaging process.

Now that every bank has been given a message and has sent a response. Each response however is in a different format and rather than sending each response through their own translator we are using a normalizer. A normalizer is similar to a translator. Instead of taking one message and formatting it to a specified format, a normalizer takes all types of

messages in all formats and creates a standardized format for each message. Our normalizer however also acts like an enricher as we wanted the user to know from which bank they would get the best offer from. The normalizer then sends each message further down a queue.

Finally we have everything we need. We have asked for a loan, we have gotten our credit score, we sent the loan information to the banks. The banks sent a response with the interest rate that they would give for a loan of that amount. We have normalized the messages from the banks. All there is left to do is pick the best bank for our user. The best bank is the one offering the lowest interest rate for the loan. To find which is the best response we use an Aggregator. An aggregator takes a lot of messages and finds the best message depending on a given detail, interest rate in this case, and sends that message down a queue. From here we just show the message to the user and the loan request is complete.

Detailed description of subsystems

loanRequest.js:

This component starts an express server on port 3030 and serves the HTML request form. Here a user can input Social Security Number, Loan Amount and Loan Duration in months.

```
<form action='/loanRequest' method='post'>
    Social Security Number(ssn):
    <input type="text" name="ssn" id="ssn">
    <br> Loan Amount($):
    <input type="number" name="loanAmount" id="loanAmount">
    <br> Loan Duration:
    <input type="number" name="loanDuration" id="loanDuration">
    <input type="submit" value="Find Loan" />
</form>
```

The form does a HTTP post request on /loanRequest with a JSON with the info in the body.

After the loanRequest server receives the REST call, it starts a connection to RabbitMQ and asserts the queue “group7GetCredit” and sends the sent information to the queue, so that the next module can fetch it.

```
//Send to credit score
ch.sendToQueue(q, Buffer.from(JSON.stringify(req.body)));
```

The loanRequest server also binds to the “group7LogExchange” which is a topic exchange that we will go into further detail later.

Lastly it also creates a binding to the “group7AggregatorFrontend” where we get our response.

After entering the details of the loan request the user is sent to a response page, where the log and response can be viewed.

```
res.render('response', {
    title: 'LoanRequest',
    message: 'LoanRequest',
    ssn: cpr,
    log: loanLogs[cpr],
    response: loanResponses[cpr]
});
```

getCreditScore.js:

The next component in the series is getCreditScore, this one has to retrieve the credit score based on a social security number.

We start by consuming messages sent to the “group7GetCredit” queue, from here we have the request that was sent.

When consuming a message the component connects to the credit bureau SOAP server, supplied by the school, then sending the social security number to the SOAP server’s function called creditScore.

```
soap.createClient(creditBureau, function (err, client) {
    if (err) {
        console.log(err)
    } else {
        client.creditScore({ssn: ssn}, function (err, result) {
            if (err) {
                console.log(err);
            } else {

                callback(result.return);

            }
        });
    }

});
```

The credit score supplied by the SOAP service is then added to the JSON object we have been sending. The next part in the chain connects to a rulebase SOAP service that tells which banks are interested in the given request.

To send the current JSON object now containing all the information we need, the component sends the JSON as a message on “group7GetBanks” queue where the next module receives it.

```
var q = 'group7GetBanks';
ch.assertQueue(q, {
    durable: true
});

ch.sendToQueue(q, Buffer.from(JSON.stringify(request)));
```

rulebaseServer.js:

The rulebaseServer component is the SOAP[4] service we made ourselves. There is one method defined called getBanks, which takes the credit score and request and determines which category the request is.

```
var service = {
    Banks_Service : {
        Banks_Port :{
            getBanks:function(args){…
            }
        }
    }
};
```

For simplicity we decided on three categories: ALL, RICH and POOR. This is to be understood as which customers a specific bank takes.
ALL being a bank that takes any customers.
RICH being banks that only take rich customers with high credit scores.
POOR being banks who help the poor and take customers with low credit scores.

The rulebase SOAP service is hosted on http://localhost:3031/getbanks?wsdl

[4] https://www.npmjs.com/package/soap

```
var xml = require('fs').readFileSync('./rulebase.wsdl','utf8');
var server = app.listen(3031,function(){
    var host = "127.0.0.1";
    var port = server.address().port;
    console.log("server running at http://%s:%s\n", host, port);
});
soap.listen(server,'/getbanks', service, xml);
```

```
 <message name="getBanksRequest">
    <part name="ssn" type="xs:string"/>
    <part name="loanAmount" type="xs:string"/>
    <part name="loanDuration" type="xs:string"/>
    <part name="creditScore" type="xs:string"/>
 </message>
//message tags define data elements for each operation
 <message name="getBanksResponse">
    <part name="banks" type="xs:string"/>
 </message>
```

getBanks.js:

The getBanks module consumes messages sent from getCreditScore on “group7GetBanks” queue.

```
soap.createClient(rulebase,function(err, client){
    if(err)
        console.error(err);
    else {
        client.getBanks(request, function(err, response){
            if(err)
                console.error(err);
            else{
                var arr = Object.keys(response).map(function(key){ return response[key] });
                console.log(arr)
                callback(arr);
            }
        });
    }
});
```

As the getCreditScore component getBanks also connects to a SOAP service (rulebaseServer) the response is just used differently.

When getBanks receives the responce from the rulebase SOAP service, it connects to the “group7RecipientList” topic exchange, using the category from the rulebase and the topic for this exchange.

```
var ex = 'group7RecipientList';

//var key = (topic.length > 0) ? topic[0] : 'all';

ch.assertExchange(ex, 'topic', {durable: true});

ch.publish(ex, key, Buffer.from(JSON.stringify(request)));
```

getBanks also sends the category and social security number to the Aggregator so it knows how many responses to expect on “group7AggregatorTopic” queue.

```
conn.createChannel(function (err, ch) {
    var q = 'group7AggregatorTopic';
    ch.assertQueue(q, {
        durable: true
    });

    ch.sendToQueue(q, Buffer.from(JSON.stringify(request)));
    console.log(" [x] Send request to Aggregator");
});
```

Translators:

For each bank that needs to be connected to the system, we need a translater to translate the data into the format in which the bank takes requests.

In our case we have the two fanout exchange RabbitMQ banks supplied by Tine, one need the request in JSON format the other needs XML.

Then we have our own RabbitMQ bank which works as a simple working queue that takes JSON formatted request.

Lastly we have a SOAP bank which is a SOAP service.

Each of these banks except for the SOAP banks, binds a queue to the “group7RecipientList” exchange with the key category we have chosen the banks to have.

```
ch.bindQueue(q, ex, key);
```

As for the SOAP bank, when sending the request we receive the response in the translater and send it to the replyQueue.

translatorJSONBank.js: Poor

The JSON bank is set to receive all requests that have the key “poor”, as in they accept loan request from people with low creditscores.

For Tine’s JSON bank we can just continue using the JSON object we have used until now, to send to the bank, as we have used the same formatting as the standard for the whole system.

The translator then published the JSON to the 'cphbusiness.bankJSON' fanout exchange with a replyTo queue aswell for the exchange to send its reply to.

```
var ex = 'cphbusiness.bankJSON';

ch.assertExchange(ex, 'fanout', {
    durable: false
});

ch.publish(ex, '', Buffer.from(JSON.stringify(request)), {
    replyTo: 'group7JSONReply'
});
```

translatorXMLBank.js: Rich

The XMLbank is set to receive all requests that have the key “rich”, as in they accept loan request from people with high creditscores.

In here we convert the JSON object to an XML object called LoanRequest, using NPM package js2xmlparser[5].

```
var XML = js2xmlparser.parse("LoanRequest", request);
```

Now with the object as XML we can send it to the XML bank as we did with the JSON bank.

```
conn.createChannel(function (err, ch) {
    var ex = 'cphbusiness.bankXML';
    ch.assertExchange(ex, 'fanout', {
        durable: false
    });
    console.log(" [x] sent: %s", XML);
    ch.publish(ex, '', Buffer.from(XML), {
        replyTo: 'group7XMLReply'
    });

});
```

translatorSoapBank.js: All

The SOAP bank is set to receive all requests that have the key “all”, as in they accept loan request from people with any creditscore.

As we receive the object, we create a connection to the SOAP bank using using the SOAP[6] NPM package.

```
soap.createClient(soapBank, function (err, client) {
    if (err) {
        console.log(err)
    } else {
        client.calculateInterest(request, function (err, result) {
            if (err) {
                console.log(err);
            } else {
                sendToNormalizer(result);
```

[5] https://www.npmjs.com/package/js2xmlparser
[6] https://www.npmjs.com/package/soap

translatorRabbitBank.js:

Our Rabbit bank is set to receive all requests that have the key “all”, as in they accept loan request from people with any creditscore.

```
var q = 'group7RabbitBank';

ch.assertQueue(q, {
    durable: true
});

console.log(" [x] rabbit sent: %s", JSON.stringify(request));

ch.sendToQueue(q, Buffer.from(JSON.stringify(request)), {replyTo: 'group7RabbitReply'})
```

The translator sends the JSON object to the group7RabbitBank queue and a replyTo queue for the reply to go to.

soapBank1.js:

This is one of our “homemade” banks. It consists of a SOAP service that listens for incoming loan requests and responds with an interest rate.

```
// xml data is extracted from wsdl file created
var xml = require('fs').readFileSync('./soapbank.wsdl', 'utf8');
var server = app.listen(3032, function () {
    var host = "localhost";
    var port = server.address().port;
});
+
    soap.listen(server, '/calculateInterest', service, xml);
```

The service is made in Javascript and functions with the help of the SOAP middleware, that allows for seamless creation of SOAP services.

The banks calculates interest rate, by first dividing incoming requests into 8 different “tiers” according to their credit score. Then a switch case assigns a random number within a certain range as interest rate.

rabbitMqBanks.js:

Another homemade bank that works by consuming incoming requests on the "*group7RabbitBank"* queue. The bank then formulates an interest rate, in the same way as soapBank1.js (see above), and pass the new loan quote to the queue defined in the *replyTo* fields of the original request.

```
var creditScore = JSON.parse(msg.content).creditScore;
var interestRate = 0;

switch (true) {
    case (creditScore <= 100):
        interestRate = (Math.random() * 20);
        console.log("1");
        break;
    case (creditScore >= 200 && creditScore < 300):
        interestRate = (Math.random() * 16);
        console.log("2");
        break;
```

normalizer.js:

The normalizer listens on all the reply queues.

```
var qs = [
    'group7XMLReply',
    'group7JSONReply',
    'group7RabbitReply',
    'group7SoapReply'];
```

When receiving a message on one of the queues, it will know what bank it came from by the queue name.
The normalizer will then build a JSON object reply, with the bank it is from, the social security number used, and the interest rate given from the bank.

```
//Tine's JSON Bank
ch.consume(qs[1], function (msg) {

    var temp = JSON.parse(msg.content);

    var request = {
        bankq: "TineJSONBank",
        ssn: temp.ssn.toString(),
        interestRate: temp.interestRate
    }
    var logtemp = "["+request.bankq+"] sent to [Normalizer]: "+msg.content.toString();
    logm.sendLog(request.ssn,logtemp)
    sendToAggregator(request);
}, {
        noAck: true
    });
```

Finally it sends the built JSON to the aggregator for it to collect all the responses.

```
var q = 'group7Aggregator';
ch.assertQueue(q , {
    durable: true
});
ch.sendToQueue(q, Buffer.from(JSON.stringify(request)));
```

Aggregator.js:

The aggregator collects all the responses sent from the normalizer, and groups them based on the social security number of the request.

While collecting responses it picks the best interest rate to respond with, but incase the banks do not respond within a timeout function from the first responses with that social security number, then the best interest rate so far is sent back to the user through the frontend.

```
setTimeout(function () {
    if(sent != res.ssn){
        console.log("bank not responding");
        sendToServer(res.ssn, ssnRes.response)
        sent = res.ssn;
    }
}, 3000);
```

logModule.js:

Throughout all components we use the logModule which gives us the ability to log a message concerning the loanrequest.

The sendLog function in the logModule takes two arguments; social security number and the message you want to send.

The log is implemented as a topic exchange that the loanRequest.js component binds a queue to to listen for log messages with topic key being the social security number.

```
var ex = 'group7LogExchange';
if(ssn.indexOf("-") != -1){
ssn = ssn.slice(0, ssn.indexOf("-"))+ssn.slice(ssn.indexOf("-")+1);
}

ch.assertExchange(ex, 'topic', { durable: true });

ch.publish(ex, ssn, Buffer.from(msg));
```

To use the logModule we simply import it into the application

```
var logm = require('./logModule.js')
```

And log the message as follows:

```
var logtemp = "[group7GetBanks] published to ["+key+"] ["+ex+"]: "+JSON.stringify(request);
logm.sendLog(request.ssn, logtemp, dev)
```

Sequence Diagram

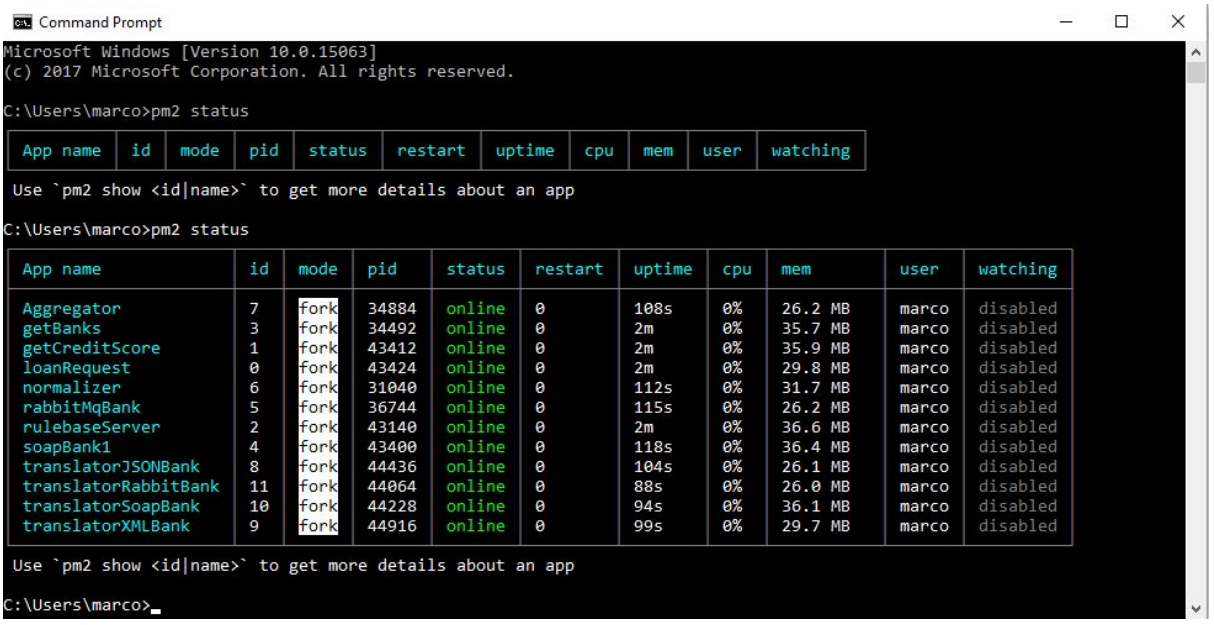
The sequence diagram is quite big for us to share, click on the following link which allows you to zoom in and maneuver around yourself to see each individual part.

https://go.gliffy.com/go/share/stmxaobfstiu6gqdnipx

The sequence diagram tracks the data flow that happens when an actor makes a request to the loan broker site.
On a side note, the sequence diagram can be found on github as well. It's in the folder with the pdf.

Screen Dumps

We are using pm2 to keep everything running, and below is all the files that are running to make the system work.



```
Command Prompt
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

C:\Users\marco>pm2 status
┌──────────┬────┬──────┬─────┬────────┬─────────┬────────┬─────┬──────┬──────┬──────────┐
│ App name │ id │ mode │ pid │ status │ restart │ uptime │ cpu │ mem  │ user │ watching │
└──────────┴────┴──────┴─────┴────────┴─────────┴────────┴─────┴──────┴──────┴──────────┘
 Use `pm2 show <id|name>` to get more details about an app

C:\Users\marco>pm2 status
┌─────────────────────┬────┬──────┬───────┬────────┬─────────┬────────┬─────┬──────────┬──────┬──────────┐
│ App name            │ id │ mode │ pid   │ status │ restart │ uptime │ cpu │ mem      │ user │ watching │
├─────────────────────┼────┼──────┼───────┼────────┼─────────┼────────┼─────┼──────────┼──────┼──────────┤
│ Aggregator          │ 7  │ fork │ 34884 │ online │ 0       │ 108s   │ 0%  │ 26.2 MB  │ marco │ disabled │
│ getBanks            │ 3  │ fork │ 34492 │ online │ 0       │ 2m     │ 0%  │ 35.7 MB  │ marco │ disabled │
│ getCreditScore      │ 1  │ fork │ 43412 │ online │ 0       │ 2m     │ 0%  │ 35.9 MB  │ marco │ disabled │
│ loanRequest         │ 0  │ fork │ 43424 │ online │ 0       │ 2m     │ 0%  │ 29.8 MB  │ marco │ disabled │
│ normalizer          │ 6  │ fork │ 31040 │ online │ 0       │ 112s   │ 0%  │ 31.7 MB  │ marco │ disabled │
│ rabbitMqBank        │ 5  │ fork │ 36744 │ online │ 0       │ 115s   │ 0%  │ 26.2 MB  │ marco │ disabled │
│ rulebaseServer      │ 2  │ fork │ 43140 │ online │ 0       │ 2m     │ 0%  │ 36.6 MB  │ marco │ disabled │
│ soapBank1           │ 4  │ fork │ 43400 │ online │ 0       │ 118s   │ 0%  │ 36.4 MB  │ marco │ disabled │
│ translatorJSONBank  │ 8  │ fork │ 44436 │ online │ 0       │ 104s   │ 0%  │ 26.1 MB  │ marco │ disabled │
│ translatorRabbitBank│ 11 │ fork │ 44064 │ online │ 0       │ 88s    │ 0%  │ 26.0 MB  │ marco │ disabled │
│ translatorSoapBank  │ 10 │ fork │ 44228 │ online │ 0       │ 94s    │ 0%  │ 36.1 MB  │ marco │ disabled │
│ translatorXMLBank   │ 9  │ fork │ 44916 │ online │ 0       │ 99s    │ 0%  │ 29.7 MB  │ marco │ disabled │
└─────────────────────┴────┴──────┴───────┴────────┴─────────┴────────┴─────┴──────────┴──────┴──────────┘
 Use `pm2 show <id|name>` to get more details about an app

C:\Users\marco>_
```

The logs of one user getting his LoanBroker response

```
13|loanReq | 1234561234
13|loanReq | { ssn: '123456-1234', loanAmount: '123', loanDuration: '123' }
1|getCredi |  [x] Received {"ssn":"123456-1234","loanAmount":"123","loanDuration":"123"}
13|loanReq | ex: amq.gen-V63FyPAB7wPPG6rxwyGMFA
1|getCredi | { ssn: '123456-1234',
1|getCredi |   loanAmount: '123',
1|getCredi |   loanDuration: '123',
1|getCredi |   creditScore: '657' }
13|loanReq | ex2: amq.gen-3AMw9KmLnl5pGzaYUJCZbw
1|getCredi |  [x] Send request to getBanks
3|getBanks |  [x] Received {"ssn":"123456-1234","loanAmount":"123","loanDuration":"123","creditScore":"657"}
2|rulebase | { ssn: '123456-1234',
2|rulebase |   loanAmount: '123',
2|rulebase |   loanDuration: '123',
2|rulebase |   creditScore: '657' }
2|rulebase | [ 'rich', 'all' ]
3|getBanks | [ 'rich', 'all' ]
3|getBanks | callback:rich,all
1|getCredi | 1234561234:[group7GetCredit] sent to [group7GetBanksDev]: {"ssn":"123456-1234","loanAmount":"123","loanDuration":"123"
"creditScore":"657"}
3|getBanks |  [x] Sent rich: '{"ssn":"123456-1234","loanAmount":"123","loanDuration":"123","creditScore":"657"}'
3|getBanks |  [x] Sent all: '{"ssn":"123456-1234","loanAmount":"123","loanDuration":"123","creditScore":"657"}'
9|translat |  [x] rich:'{"ssn":"123456-1234","loanAmount":"123","loanDuration":"123","creditScore":"657"}'
9|translat | { ssn: '1234561234',
9|translat |   loanAmount: '123',
9|translat |   loanDuration: '123',
9|translat |   creditScore: '657' }
10|transla |  [x] all:'{"ssn":"123456-1234","loanAmount":"123","loanDuration":"123","creditScore":"657"}'
11|transla |  [x] all:'{"ssn":"123456-1234","loanAmount":"123","loanDuration":"123","creditScore":"657"}'
11|transla | { ssn: '1234561234',
11|transla |   loanAmount: '123',
11|transla |   loanDuration: '123',
11|transla |   creditScore: '657' }
3|getBanks |  [x] Send request to Aggregator
7|Aggregat |  [x] Received from server   topic: rich,all       msg: {"ssn":"1234561234","topic":["rich","all"]}
4|soapBank | [ 'ssn', 'loanAmount', 'loanDuration', 'creditScore' ]
4|soapBank | { ssn: '1234561234',
4|soapBank |   loanAmount: '123',
4|soapBank |   loanDuration: '123',
4|soapBank |   creditScore: '657' }
4|soapBank | her er credit: 657
4|soapBank | 6
4|soapBank | 0.4552743467203968
```

```
3|getBanks | 1234561234:[group7GetBanks] published to [all] [group7RecipientListDev]: {"ssn":"123456-1234","loanAmount":"123","loanD
uration":"123","creditScore":"657"}
3|getBanks | 1234561234:[group7GetBanks] published to [rich] [group7RecipientListDev]: {"ssn":"123456-1234","loanAmount":"123","loan
Duration":"123","creditScore":"657"}
9|translat | 1234561234:[group7RecipientListDev] to [group7TranslatorXMLBankDev]: {"ssn":"123456-1234","loanAmount":"123","loanDurat
ion":"123","creditScore":"657"}
9|translat |  [x] sent: <?xml version='1.0'?>
9|translat | <LoanRequest>
9|translat |     <ssn>1234561234</ssn>
9|translat |     <loanAmount>123</loanAmount>
9|translat |     <loanDuration>1980-04-01 01:00:00.0 CET</loanDuration>
9|translat |     <creditScore>657</creditScore>
9|translat | </LoanRequest>
11|transla |  [x] rabbit sent: {"ssn":"1234561234","loanAmount":"123","loanDuration":"123","creditScore":"657"}
11|transla | 1234561234:[group7RecipientListDev] to [group7TranslatorRabbitBankDev]: {"ssn":"123456-1234","loanAmount":"123","loanDu
ration":"123","creditScore":"657"}
5|rabbitMq |  [x] Received {"ssn":"1234561234","loanAmount":"123","loanDuration":"123","creditScore":"657"}
5|rabbitMq | group7RabbitReplyDev
5|rabbitMq | 1234561234
5|rabbitMq | 6
5|rabbitMq | [object Object]
5|rabbitMq | group7RabbitReplyDev
5|rabbitMq | sendte reply: {"loanResponse":{"interestRate":0.9833925760775937,"ssn":"1234561234"}}
10|transla | { loanResponse: { interestRate: '0.4552743467203968', ssn: '1234561234' } }
10|transla |  [x] Send request to Normalizer
10|transla | 1234561234:[group7RecipientListDev] to [group7TranslatorSoapBankDev]: {"ssn":"123456-1234","loanAmount":"123","loanDura
tion":"123","creditScore":"657"}
6|normaliz | 1234561234:[RabbitBank] sent to [Normalizer]: {"loanResponse":{"interestRate":0.9833925760775937,"ssn":"1234561234"}}
6|normaliz |  [x] Send request RabbitBank to Aggregator: {"bankq":"RabbitBank","ssn":"1234561234","interestRate":0.983392576077593
7}

6|normaliz |  [x] Send request SoapBank to Aggregator: {"bankq":"SoapBank","ssn":"1234561234","interestRate":"0.4552743467203968"}
6|normaliz | 1234561234:[SoapBank] sent to [Normalizer]: {"loanResponse":{"interestRate":"0.4552743467203968","ssn":"1234561234"}}
6|normaliz | 1234561234:[TineXMLBank] sent to [Normalizer]: <LoanResponse>
6|normaliz |    <interestRate>2.4000000000000004</interestRate>
6|normaliz |    <ssn>1234561234</ssn>
6|normaliz | </LoanResponse>
7|Aggregat | Recieved: {"bankq":"SoapBank","ssn":"1234561234","interestRate":"0.4552743467203968"}       ResCol.length: 1
7|Aggregat | Key stored: {"numBanks":3,"response":[{"bankq":"SoapBank","ssn":"1234561234","interestRate":"0.4552743467203968"}],"sen
t":false}
7|Aggregat | Recieved: {"bankq":"RabbitBank","ssn":"1234561234","interestRate":0.9833925760775937}       ResCol.length: 1
7|Aggregat | Key stored: {"numBanks":3,"response":[{"bankq":"SoapBank","ssn":"1234561234","interestRate":"0.4552743467203968"},{"ban
kq":"RabbitBank","ssn":"1234561234","interestRate":0.9833925760775937}],"sent":false}
6|normaliz |  [x] Send request TineXMLBank to Aggregator: {"bankq":"TineXMLBank","ssn":"1234561234","interestRate":"2.40000000000000
04"}
7|Aggregat | Recieved: {"bankq":"TineXMLBank","ssn":"1234561234","interestRate":"2.4000000000000004"}       ResCol.length: 0
7|Aggregat | Key stored: {"numBanks":3,"response":[{"bankq":"SoapBank","ssn":"1234561234","interestRate":"0.4552743467203968"},{"ban
kq":"RabbitBank","ssn":"1234561234","interestRate":0.9833925760775937},{"bankq":"TineXMLBank","ssn":"1234561234","interestRate":"2.4
000000000000004"}],"sent":false}
6|normaliz | 1234561234:[Normalizer] publish to [group7AggregatorDev]: {"bankq":"RabbitBank","ssn":"1234561234","interestRate":0.983
3925760775937}
6|normaliz | 1234561234:[Normalizer] publish to [group7AggregatorDev]: {"bankq":"SoapBank","ssn":"1234561234","interestRate":"0.4552
743467203968"}
6|normaliz | 1234561234:[Normalizer] publish to [group7AggregatorDev]: {"bankq":"TineXMLBank","ssn":"1234561234","interestRate":"2.4
000000000000004"}
7|Aggregat |  [x] Send to Frontend {"bankq":"SoapBank","ssn":"1234561234","interestRate":"0.4552743467203968"}
13|loanReq | {"bankq":"SoapBank","ssn":"1234561234","interestRate":"0.4552743467203968"}
13|loanReq | logq: amq.gen-V63FyPAB7wPPG6rxwyGMFA
7|Aggregat | 1234561234:[Aggregator] sent to [group7AggregatorFrontendDev]: {"bankq":"SoapBank","ssn":"1234561234","interestRate":"0
.4552743467203968"}
6|normaliz | ----------------
6|normaliz | ----------------
6|normaliz | ----------------
```

You can see the log in a more simplified form on our loanbroker site:
http://37.139.5.194:3030/

LoanRequest

CPR:

1234561234

Log:

```
Request Sent!
[loanRequest] sent to [group7GetCredit]: {"ssn":"123456-1234","loanAmount":"123","loanDuration":"123"}
[group7GetCredit] sent to [group7GetBanks]: {"ssn":"123456-1234","loanAmount":"123","loanDuration":"123","creditScore":"275"}
[group7GetBanks] published to [all] [group7RecipientList]: {"ssn":"123456-1234","loanAmount":"123","loanDuration":"123","creditScore":"275"}
[group7GetBanks] published to [poor] [group7RecipientList]: {"ssn":"123456-1234","loanAmount":"123","loanDuration":"123","creditScore":"275"}
[group7RecipientList] to [group7TranslatorRabbitBank]: {"ssn":"123456-1234","loanAmount":"123","loanDuration":"123","creditScore":"275"}
[group7RecipientList] to [group7TranslatorSoapBank]: {"ssn":"123456-1234","loanAmount":"123","loanDuration":"123","creditScore":"275"}
[group7RecipientList] to [group7TranslatorJSONBank]: {"ssn":"123456-1234","loanAmount":"123","loanDuration":"123","creditScore":"275"}
[TineJSONBank] sent to [Normalizer]: {"interestRate":5.5,"ssn":1234561234}
[RabbitBank] sent to [Normalizer]: {"loanResponse":{"interestRate":12.868278123608608,"ssn":"1234561234"}}
[SoapBank] sent to [Normalizer]: {"loanResponse":{"interestRate":"13.1028804932219","ssn":"1234561234"}}
[Normalizer] publish to [group7Aggregator]: {"bankq":"TineJSONBank","ssn":"1234561234","interestRate":5.5}
[Normalizer] publish to [group7Aggregator]: {"bankq":"SoapBank","ssn":"1234561234","interestRate":"13.1028804932219"}
[Normalizer] publish to [group7Aggregator]: {"bankq":"RabbitBank","ssn":"1234561234","interestRate":12.868278123608608}
[Aggregator] sent to [group7AggregatorFrontend]: {"bankq":"TineJSONBank","ssn":"1234561234","interestRate":5.5}
```

Response:

```
{"bankq":"TineJSONBank","ssn":"1234561234","interestRate":5.5}
```

Bottlenecks and improvements.

There is one glaring bottleneck that the entire system relies on, which is the datdb:cphbusiness server. As this has to handle the entire class's queues and exchanges and we use it more than 10 times in a single run through of the loanbroker application. If performance is what needs to improved, then the logModule could be turned off or changed, as this effectively doubles the amount of of requests send to the datdb:cphbusiness server.

Testing

In regards to testability, then we have not setup any automatic tests at the current time. But this does not mean that it cannot be done.
We used manual testing during the development progress, where we used http://datdb.cphbusiness.dk:15672 to see if all the different queues responded as they should or if there was any deadlocks.
For automatic testing, we could have used a module, where the queues could be supplemented and an expected result, and it could check versus a static expression or with a standalone queue.
We could also have used a testing queue, where all the different servers could throw their msg that way and its viability could be tested.


The Caped Programmer
The Hero That RabbitMQ Deserves
wallpaperzzz.com