Big D - Loan Broker

### Github link

<https://github.com/bigDUnited/big-d-loan-broker>

# Bottlenecks

The loan broker system depends on some external systems and various technologies that could each prove to be a bottleneck at some point.

Most of the communication inside the system (and even outside the system) happens on RabbitMQ. RabbitMQ is known to be a fast and reliable service. However, the specifications of the machine on which it runs and the network speed/reliability could be limits for the RabbitMQ server. If the RabbitMQ server is under too much stress and it becomes a bottleneck the kind of behaviour we could expect is longer communication time between each of the modules of the system that communicate with RabbitMQ. However, RabbitMQ is very optimised and it is extremely unlikely to turn into a bottleneck for this system.

Another potential bottleneck are the outside systems that we have no control over such as the credit bureau and the rulebase and all the external banks. Since we do not have any control over them, if our system is under a lot of pressure and it’s handling it well, the outside services will be what is slowing us down. If the credit bureau or the rule base are not able to keep up with the amount of messages we’re processing, the overall throughput of our system will go down.

The main bottleneck is the tee that we have to make between the recipient list and the aggregator. The aggregator needs to know which banks to wait for. In order to achieve that we created a tee between the recipient list and the aggregator. Using this we will be able to handle waiting for banks to respond. If they do not respond before the timeout runs out the message will be send on to the customer informing them of the failure of the bank. If only a percentage of the banks that should respond actually do, the customer will know which bank responded and which didn’t. In the case of non responding banks the system will start piling messages in the aggregator, which, if the timeout is too long the system will get flooded and severely hindered. This is our biggest potential bottleneck. The way we handle it is by making the timeout short.

# Testability

The testing procedure is important and it is part of every reasonable and working project nowadays. As we can see from the class diagram, there is increasing complexity in our code, therefore a reasonable troubleshooting mechanism is always a good strategy. Following this statement, there are few testing advices, or checkpoints, which we should all take in consideration before starting the testing procedure. We will state all of them one by one and we will describe what we did to improve the quality of our code and find possible flaws in it:

1. *Isolate the application from the messaging implementation by using interfaces and implementation classes:*

At the beginning of the development process, our team discovered that there will be a lot of repetitive code. Therefore we decided to structure everything and to introduce several classes such as rmqConsume, rmqPublish, queue\_names, translators in Python and RuleBaseInterface in Java. By stacking the frequently used code in logical classes we isolate specific implementations and we reduce the chances or human mistakes or errors in our program.

1. *Test the business logic with unit test cases before plugging it into the messaging environment:*

Our current solution is mainly written in Python, therefore unit testing is not a possibility. However, what we do it so execute our code with different entry parameters to check for any edge cases, similar to what we will do in the classic java unit testing.

1. *Provide a 'Mock' implementation of the messaging layer that allows you to test synchronously:*

We introduced “dummy” python classes which trigger specific components, or services in our code. In this way we make sure that every component will work as expected. A good example of a class like this one is : credit.py, which can be found in our git repository. It makes sure that the credit score server will run and will return a specific value. This was a must in the early stages of our project.

# Description of the web service

The web service will take a request which should contain the SSN of the customer, which is in the format XXXXXX-XXXX, and this parameter, although a string, should contain only numbers. Next we will need the loan amount, which will be a float, greater than 0. Lastly, we will need a loan duration, which will be a NUMBER?????. These parameters are sufficient information for the request, because all the other parameters are not dependent on the client. The response will contain the SSN of the customer, with the same format as the request, just to make sure that he gets the correct information back. It will also have the interest rate, which will be a float. In case of an exception, we will return a message, which will contain a description of the error, in a human readable format that could be displayed to the user.

# Implementation from the SOA perspective

Our business domain, or the consumer interface is the web service that accepts the requests and creates the initial message in the chain. It communicates the message directly to the services, which transform the message, utilizing various service components. These components can also communicate with operational systems, that provide more information, like the credit score or the banks. This means that the customer does not need to know the bottom layers in order to get an answer. All he needs to know is the initial business domain, and then the chain will be triggered automatically.

We have pretty loose coupling, because all our services run in their own process and they are independent of one another, since all they do is consume and publish messages. Only things which have a tighter coupling are the components communicating with the other services, like the credit bureau or the rulebase, because they have to know the server and the method. Although this is not such a problem because they don’t hold the actual server object, just the reference to the WSDL file. Our cohesion is high, because we keep the classes as simple as possible, yet still reusable, like the publishing and the consuming classes. They are reused in all of the services which are communicating through RabbitMQ. Our high cohesion means that the services although very simple can do everything that they need to, and changes in one, will have no effect on the others. Our services are coarsely grained, because we are using complex structures, and they also communicate with other services. But this is not a large coarse graining, because we are using a simple request/reply mode, the complexity of the structure is not very complex, i.e. we are not using many nested properties.

# Source code:

## SOAP Rule-base Server:

### RuleBasePublisher.java

package webservice.endpoint;

import javax.xml.ws.Endpoint;

import webservice.soap.RuleBaseImplementation;

//Endpoint publisher

public class RuleBasePublisher {

/\*

\* Can be accessed via "http://localhost:9999/ws/hello?wsdl"

\* Tutorial : http://www.mkyong.com/webservices/jax-ws/jax-ws-hello-world-example/

\* Export command : wsimport -keep http://localhost:9999/ws/rule-base?wsdl

\* Must have : The server should be runing before exporting

\* Your current directory should not be src/ ,because then you will overwrite stuff

\*/

public static void main(String[] args) {

Endpoint.publish("http://localhost:9999/ws/rule-base?wsdl", new RuleBaseImplementation());

}

}

### RuleBaseInterface.java

package webservice.soap;  
  
import java.util.ArrayList;  
import javax.jws.WebMethod;  
import javax.jws.WebService;  
import javax.jws.soap.SOAPBinding;  
import javax.jws.soap.SOAPBinding.Style;  
import webservice.logic.entity.Bank;  
  
//Service Endpoint Interface  
//Remote procedure call -its calling a method from another system  
//You have to tell the system what method to run and what to set as params  
@WebService  
@SOAPBinding(style = Style.RPC)  
public interface RuleBaseInterface {  
  
 @WebMethod  
 ArrayList<Bank> getBanksByCreditScore(int creditScore);  
  
 @WebMethod  
 String getBanksByCreditScoreJson(int creditScore);  
  
}

### RuleBaseImplementation.java

package webservice.soap;  
  
import java.io.StringWriter;  
import java.util.ArrayList;  
import javax.jws.WebService;  
import javax.xml.bind.JAXBContext;  
import javax.xml.bind.JAXBElement;  
import javax.xml.bind.JAXBException;  
import javax.xml.bind.Marshaller;  
import javax.xml.namespace.QName;  
import webservice.logic.Controller;  
import webservice.logic.entity.Bank;  
import webservice.logic.entity.RuleBaseResponse;  
  
//Service Implementation  
@WebService(endpointInterface = "webservice.soap.RuleBaseInterface")  
public class RuleBaseImplementation implements RuleBaseInterface {  
  
 @Override  
 public ArrayList<Bank> getBanksByCreditScore(int creditScore) {  
 Controller control = new Controller();  
 return control.getBanksByCrediScore(creditScore);  
 }  
  
 @Override  
 public String getBanksByCreditScoreJson(int creditScore) {  
 String xmlString = null;  
  
 Controller control = new Controller();  
 ArrayList<Bank> banks = control.getBanksByCrediScore(creditScore);  
  
 RuleBaseResponse rbr = new RuleBaseResponse(banks);  
  
 try {  
 JAXBContext jaxbContext = JAXBContext.newInstance(RuleBaseResponse.class);  
 Marshaller jaxbMarshaller = jaxbContext.createMarshaller();  
  
 // output pretty printed  
 jaxbMarshaller.setProperty(Marshaller.JAXB\_FORMATTED\_OUTPUT, true);  
  
 //Create array of objects based on RuleBase structure  
 JAXBElement<RuleBaseResponse> je2 = new JAXBElement(  
 new QName("RuleBaseList"), RuleBaseResponse.class, rbr);  
  
 //jaxbMarshaller.marshal(customer, file);  
 StringWriter sw = new StringWriter();  
 jaxbMarshaller.marshal(je2, sw);  
 xmlString = sw.toString();  
  
 } catch (JAXBException e) {  
 return "Internal server error : " + e;  
 }  
  
 return xmlString;  
 }  
}

### Controller.java

package webservice.logic;  
  
import java.util.ArrayList;  
import webservice.logic.entity.Bank;  
import webservice.logic.entity.Rank;  
import webservice.logic.utils.RanksLevels;  
  
public class Controller {  
  
 private ArrayList<Bank> banksList;  
 private ArrayList<Bank> shareableList;  
 private ArrayList<Rank> ranksList;  
  
 public Controller() {  
 banksList = new ArrayList();  
 ranksList = new ArrayList();  
 loadBanks();  
 loadRanks();  
 }  
  
 private void loadBanks() {  
 banksList.add(new Bank("danskebank\_translator\_queue", new int[]{RanksLevels.EXCELLENT\_CS,  
 RanksLevels.GOOD\_CS}));  
 banksList.add(new Bank("nordea\_translator\_queue", new int[]{RanksLevels.GOOD\_CS, RanksLevels.AVERAGE\_CS,  
 RanksLevels.POOR\_CS}));  
 banksList.add(new Bank("nytkredit\_translator\_queue", new int[]{RanksLevels.AVERAGE\_CS,  
 RanksLevels.POOR\_CS, RanksLevels.BAD\_CS}));  
 banksList.add(new Bank("bdo\_translator\_queue", new int[]{RanksLevels.POOR\_CS,  
 RanksLevels.BAD\_CS, RanksLevels.MISERABLE\_CS}));  
  
 }  
  
 private void loadRanks() {  
 ranksList.add(new Rank(RanksLevels.EXCELLENT\_CS, 720, 800));  
 ranksList.add(new Rank(RanksLevels.GOOD\_CS, 680, 719));  
 ranksList.add(new Rank(RanksLevels.AVERAGE\_CS, 620, 679));  
 ranksList.add(new Rank(RanksLevels.POOR\_CS, 580, 619));  
 ranksList.add(new Rank(RanksLevels.BAD\_CS, 500, 579));  
 ranksList.add(new Rank(RanksLevels.MISERABLE\_CS, 0, 500));  
 }  
  
 public ArrayList<Bank> getBanksByCrediScore(int creditScore) {  
 shareableList = new ArrayList();  
 int chosenRank = 0;  
  
 for (int i = 0; i < ranksList.size(); i++) {  
 if (ranksList.get(i).getLowerLimit() < creditScore && ranksList.get(i).getUpperLimit() > creditScore) {  
 chosenRank = ranksList.get(i).getRankLevel();  
 break;  
 }  
 }  
  
 if (chosenRank > 0) {  
 for (int x = 0; x < banksList.size(); x++) {  
 int[] currRanksList = banksList.get(x).getRankElem();  
 for (int y = 0; y < currRanksList.length; y++) {  
 if (currRanksList[y] == chosenRank) {  
 shareableList.add(banksList.get(x));  
 }  
 }  
 }  
 }  
 return shareableList;  
 }  
}

### RanksLevels.java

package webservice.logic.utils;  
  
public class RanksLevels {  
  
 public final static int EXCELLENT\_CS = 1;  
  
 public final static int GOOD\_CS = 2;  
  
 public final static int AVERAGE\_CS = 3;  
  
 public final static int POOR\_CS = 4;  
  
 public final static int BAD\_CS = 5;  
  
 public final static int MISERABLE\_CS = 6;  
}

### Bank.java

package webservice.logic.entity;  
  
import java.util.Arrays;  
import javax.xml.bind.annotation.XmlRootElement;  
  
@XmlRootElement  
public class Bank {  
  
 private String name;  
 private int[] rankElem;  
  
 public Bank() {  
 this.name = null;  
 this.rankElem = null;  
 }  
  
 public Bank(String name, int[] rankElem) {  
 this.name = name;  
 this.rankElem = rankElem;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int[] getRankElem() {  
 return rankElem;  
 }  
  
 public void setRankElem(int[] rankElem) {  
 this.rankElem = rankElem;  
 }  
   
 @Override  
 public String toString() {  
 return "Bank{" + "name=" + name + ", ranks=" + Arrays.toString(rankElem) + '}';  
 }  
  
}

### Rank.java

package webservice.logic.entity;  
  
public class Rank {  
  
 private final int rankLevel;  
 private final int lowerLimit;  
 private final int upperLimit;  
  
 public Rank(int rankLevel, int lowerLimit, int upperLimit) {  
 this.rankLevel = rankLevel;  
 this.lowerLimit = lowerLimit;  
 this.upperLimit = upperLimit;  
 }  
  
 public int getLowerLimit() {  
 return lowerLimit;  
 }  
  
 public int getRankLevel() {  
 return rankLevel;  
 }  
  
 public int getUpperLimit() {  
 return upperLimit;  
 }  
  
}

### RuleBaseResponse.java

package webservice.logic.entity;  
  
import java.util.ArrayList;  
  
public class RuleBaseResponse {  
 private ArrayList<Bank> bankElem;  
  
 public RuleBaseResponse( ArrayList<Bank> bankElem ) {  
 this.bankElem = bankElem;  
 }  
  
 public ArrayList<Bank> getBankElem() {  
 return bankElem;  
 }  
  
 public void setBankElem(ArrayList<Bank> bankElem) {  
 this.bankElem = bankElem;  
 }  
}

## Broker:

"""

Handles results from banks and produces a result message

listing results from qualified banks.

"""

from rmqConsume import Consumer

from rmqPublish import publish\_to\_bank

from queue\_names import \*

from json import loads

import translators as tr

awaiting = {}

def callback(ch, method, properties, body):

print properties

print body

consumer = Consumer("localhost", AGGREGATOR\_QUEUE)

consumer.on\_receive = callback

consumer.consume()

AGGREGATOR\_QUEUE

"""

Enriches a message with available banks by calling

a rule base service.

"""

from rulebase import RuleBase

from rmqConsume import Consumer

from rmqPublish import publish\_to\_q

from queue\_names import \*

from json import loads, dumps

import requests

def callback(ch, method, properties, body):

m = loads(body)

req = "http://127.0.0.1:3001/score/{0}".format(m['score'])

response = requests.get(req)

banks = response.json()

m['banks'] = banks

publish\_to\_q("localhost", RECIPIENT\_LIST\_QUEUE, dumps(m), properties)

consumer = Consumer("localhost", BANK\_ENRICHER\_QUEUE)

consumer.on\_receive = callback

consumer.consume()

from rulebase import RuleBase

from rmqConsume import Consumer

from rmqPublish import publish\_to\_q

from queue\_names import \*

from json import loads, dumps

import requests

def callback(ch, method, properties, body):

m = loads(body)

print "BDO: ", m

consumer = Consumer("localhost", BDO\_TRANSLATOR\_QUEUE)

consumer.on\_receive = callback

consumer.consume()

"""

Entry point to project. Sends a single message to be processed.

"""

from rmqPublish import publish\_to\_q

from queue\_names import CREDIT\_ENRICHER\_QUEUE

import json

import pika

message = {

"ssn": "123456-3212", #social security number

"amount": "300", #crowns

"duration": "60", #days

"timeout" : "900" #seconds

}

publish\_to\_q(

'localhost',

CREDIT\_ENRICHER\_QUEUE,

json.dumps(message),

pika.BasicProperties(

correlation\_id="12399942",

reply\_to="result",))

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

Dummy service that provides a random credit score.

"""

import json

from random import randint

from flask import Flask

from flask\_restful import Resource, Api

def randit():

if 1 == 1:

print "DSDS"

return randint(500, 1000)

app = Flask(\_\_name\_\_)

api = Api(app)

class HelloWorld(Resource):

def get(self):

return {'creditScore': str(randit())}

api.add\_resource(HelloWorld, '/')

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True, port=3000)

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

from suds.client import Client

import logging

def getCreditScore(ssn='123456-1234'):

"""

Get credit score from third-party web service.

Takes a social security number.

"""

client = Client('http://139.59.154.97:8080/CreditScoreService/CreditScoreService?wsdl')

result = client.service.creditScore(ssn)

logging.info("Answer from API:{0}".format(result))

return result

"""

Node that receives initial message and enriches it with a credit from

the credit score service.

"""

from rmqConsume import Consumer

from rmqPublish import publish\_to\_q

from creditapi import getCreditScore

from queue\_names import \*

from json import loads, dumps

def callback(ch, method, properties, body):

m = loads(body)

score = getCreditScore(m['ssn'])

m['score'] = str(score)

publish\_to\_q("localhost",

BANK\_ENRICHER\_QUEUE,

dumps(m), properties)

consumer = Consumer("localhost", CREDIT\_ENRICHER\_QUEUE)

consumer.on\_receive = callback

consumer.consume()

"""

Translator node for Danske Bank(bankJSON).

"""

from rmqConsume import Consumer

from rmqPublish import publish\_to\_bank

from queue\_names import \*

from json import loads

import translators as tr

def callback(ch, method, properties, body):

m = loads(body)

result = tr.dumps(m, "danskebank")

properties.correlation\_id += "danskebank"

print "danske bank translator: ", result

publish\_to\_bank(

"datdb.cphbusiness.dk",

'cphbusiness.bankJSON',

result,

properties)

consumer = Consumer("localhost", DANSKEBANK\_TRANSLATOR\_QUEUE)

consumer.on\_receive = callback

consumer.consume()

"""

Translator node for Nordea(bankXML).

"""

from rmqConsume import Consumer

from rmqPublish import publish\_to\_bank

from queue\_names import \*

from json import loads

import translators as tr

def callback(ch, method, properties, body):

m = loads(body)

result = tr.dumps(m, "nordea")

properties.correlation\_id += "nordea"

print "nordea bank translator: ", result

publish\_to\_bank(

"datdb.cphbusiness.dk",

'cphbusiness.bankXML',

result,

properties)

consumer = Consumer("localhost", NORDEA\_TRANSLATOR\_QUEUE)

consumer.on\_receive = callback

consumer.consume()

"""

Handles results from banks passing a message in

internal message format to aggregation service.

"""

from rmqConsume import Consumer

from rmqPublish import publish\_to\_bank

from queue\_names import \*

from json import loads

import translators as tr

def callback(ch, method, properties, body):

print properties

print body

mtype = properties.correlation\_id[8:]

message = tr.loads(body, mtype)

consumer = Consumer("datdb.cphbusiness.dk", NORMALIZER\_QUEUE)

consumer.on\_receive = callback

consumer.consume()

from rulebase import RuleBase

from rmqConsume import Consumer

from rmqPublish import publish\_to\_q

from queue\_names import \*

from json import loads, dumps

import requests

def callback(ch, method, properties, body):

m = loads(body)

print "Nytkredit: ", m

consumer = Consumer("localhost", NYTKREDIT\_TRANSLATOR\_QUEUE)

consumer.on\_receive = callback

consumer.consume()

"""

Constants for all queue names used in the project.

"""

CREDIT\_ENRICHER\_QUEUE="credit\_enricher\_queue"

BANK\_ENRICHER\_QUEUE="bank\_enricher\_queue"

RECIPIENT\_LIST\_QUEUE="recipient\_list\_queue"

DANSKEBANK\_TRANSLATOR\_QUEUE="danskebank\_translator\_queue"

NORDEA\_TRANSLATOR\_QUEUE="nordea\_translator\_queue"

NYTKREDIT\_TRANSLATOR\_QUEUE="nytkredit\_translator\_queue"

BDO\_TRANSLATOR\_QUEUE="bdo\_translator\_queue"

DANSKEBANK\_REQUEST\_QUEUE="danskebank\_request\_queue"

NORDEA\_REQUEST\_QUEUE="nordea\_request\_queue"

NYTKREDIT\_REQUEST\_QUEUE="nytkredit\_request\_queue"

BDO\_REQUEST\_QUEUE="bdo\_request\_queue"

NORMALIZER\_QUEUE="normalizer\_queue"

AGGREGATOR\_QUEUE="aggregator\_queue"

RESPONSE\_QUEUE="response\_queue"

"""

Node that ensures that a message is sent to all relevant translators.

It will also send the original request to the awaiting normalizer

so that it schedules a timeout and prepares to get all bank responses

in the given timeframe.

"""

from rulebase import RuleBase

from rmqConsume import Consumer

from rmqPublish import publish\_to\_q

from queue\_names import \*

from json import loads, dumps

import requests

TRANSLATORS = [

DANSKEBANK\_TRANSLATOR\_QUEUE,

NORDEA\_TRANSLATOR\_QUEUE,

NYTKREDIT\_TRANSLATOR\_QUEUE,

BDO\_TRANSLATOR\_QUEUE,

]

def callback(ch, method, properties, body):

m = loads(body)

res = loads(body)

del res["banks"]

final\_message = dumps(res)

publish\_to\_q("datdb.cphbusiness.dk",

NORMALIZER\_QUEUE,

body,

properties)

for t in TRANSLATORS:

if t in m["banks"]:

publish\_to\_q("localhost", t, final\_message, properties)

consumer = Consumer("localhost", RECIPIENT\_LIST\_QUEUE)

consumer.on\_receive = callback

consumer.consume()

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

RabbitMQ consumer utilities.

"""

import pika

def callback(ch, method, properties, body):

"""Example callback"""

print "hello!!!" + body

class Consumer(object):

"""

A simple blocking consumer that listens to a single queue.

It takes an address and a queue name to connect.

A callback on message receive can be specefied with on\_receive.

"""

def \_\_init\_\_(self, host, queue):

self.host = host

self.queue = queue

self.on\_receive = None

def consume(self):

"""

Start consuming. This method blocks.

"""

connection = pika.BlockingConnection(pika.ConnectionParameters(

host=self.host))

channel = connection.channel()

channel.queue\_declare(queue=self.queue)

channel.basic\_consume(

self.on\_receive,

queue=self.queue,

no\_ack=True)

print(' [\*] Waiting for messages. To exit press CTRL+C')

channel.start\_consuming()

if \_\_name\_\_ == '\_\_main\_\_':

consumer = Consumer("127.0.0.1", "hello")

consumer.on\_receive = callback

consumer.consume()

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

RabbitMQ publish utilities.

"""

import pika

def publish\_to\_bank(host, exchange, message, properties):

"""

Publish to a third-party bank. Uses an exchange name.

properties should be a pika properties object.

"""

exchange\_name = ""

connection = pika.BlockingConnection(pika.ConnectionParameters(

host))

channel = connection.channel()

channel.exchange\_declare(

exchange=exchange,

exchange\_type='fanout',

)

channel.basic\_publish(exchange=exchange,

routing\_key="",

body=message,

properties=properties)

connection.close()

def publish\_to\_q(host, queue, message, properties):

"""

Publishing function for the internal queues.

A host address has to be provided as well as queue name to push to.

properties should be a pika properties object.

"""

connection = pika.BlockingConnection(pika.ConnectionParameters(

host))

channel = connection.channel()

channel.queue\_declare(queue=queue)

channel.basic\_publish(exchange='',

routing\_key=queue,

body=message,

properties=properties)

connection.close()

if \_\_name\_\_ == '\_\_main\_\_':

publish\_to\_q('localhost', 'hello', 'Hello World!')

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

"""

Deprecated! Implementation of rule base.

Calculates a list of banks available for a

certain credit score wrapped in a REST service.

"""

import json

from random import randint

from flask import Flask

from flask\_restful import Resource, Api

from queue\_names import \*

class RuleBase(object):

"""A data object that keeps a single rule."""

def \_\_init\_\_(self, name, maxScore):

self.name = name

self.maxScore = maxScore

def \_\_str\_\_(self):

return "{0} {1}".format(self.name, self.maxScore)

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

def calculate(score, banks):

"""Calculate list of banks that are available for given score."""

res = []

for bank in banks:

if bank.maxScore <= score:

res.append(bank)

return res

banks = [

RuleBase(DANSKEBANK\_TRANSLATOR\_QUEUE, 200),

RuleBase(NORDEA\_TRANSLATOR\_QUEUE, 400),

RuleBase(NYTKREDIT\_TRANSLATOR\_QUEUE, 600),

RuleBase(BDO\_TRANSLATOR\_QUEUE, 700)

]

def bankToJSON(bank):

"""Return a bank json representation"""

return bank.name

app = Flask(\_\_name\_\_)

api = Api(app)

class ScoreService(Resource):

def get(self, score):

"""

Get banks by score service.

"""

res = map(bankToJSON, calculate(int(score), banks))

return res

api.add\_resource(ScoreService, '/score/<int:score>')

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True, port=3001)

"""

Message format translation utilities.

"""

import json

import datetime

import xml.etree.ElementTree as ET

from datetime import date

def dump\_danskebank(message):

"""

Example input: {

"ssn": "123456-3212",

"amount": "3000",

"duration": "7000",

"timeout" : "900",

"score": "3000",

}

"""

res = message

res["loanAmount"] = float(res["amount"])

del res["amount"]

res["loanDuration"] = int(res["duration"])

del res["duration"]

del res["score"]

res["ssn"] = long("".join(res["ssn"].split("-")))

del res["timeout"]

return json.dumps(res)

def load\_danskebank(message):

"""

Example input: {"interestRate":5.5,"ssn":1234563212}

"""

res = json.loads(message)

res["interest"] = float(res["interestRate"])

del res["interestRate"]

res["ssn"] = "{}-{}".format(res["ssn"][:6], res["ssn"][6:])

return res

def dump\_nordea(message):

"""

Example input: {

"ssn": "123456-3212",

"amount": "3000",

"duration": "7000",

"timeout" : "900",

"score": "3000",

}

"""

res = ET.Element('LoanRequest')

ssn = ET.SubElement(res, 'ssn')

ssn.text = "".join(message["ssn"].split("-"))

score = ET.SubElement(res, 'creditScore')

score.text = str(int(message['score']))

amount = ET.SubElement(res, 'loanAmount')

amount.text = str(float(message['amount']))

duration = ET.SubElement(res, 'loanDuration')

d = date.fromtimestamp(0)

end = d + datetime.timedelta(days=int(message['duration']))

duration.text = '{0} 00:00:00.0 CET'.format(end.strftime('%Y-%m-%d'))

return ET.tostring(res)

def load\_nordea(message):

"""

Example input:

<LoanResponse>

<interestRate>4.5600000000000005</interestRate>

<ssn>1234563212</ssn>

</LoanResponse>

"""

res = {}

root = ET.fromstring(message)

res["interest"] = float(root.find("interestRate").text)

ssn = root.find("ssn").text

res["ssn"] = "{}-{}".format(ssn[:6], ssn[6:])

return res

"""

Tables for bi-directional translation.

Each translation type should have a "dump" and a "load"

that should point to corresponding translation functions

that take a message and return a translated message.

"""

translator\_types = {

"danskebank" : {"dump": dump\_danskebank, "load": load\_danskebank},

"nordea" : {"dump": dump\_nordea, "load": load\_nordea},

}

def dumps(message, type):

"""

Transforms internal messages to external formats for usage with

third-party services.

"""

return translator\_types[type]["dump"](message)

def loads(message, type):

"""

Transform external message formats to internal message format.

"""

return translator\_types[type]["load"](message)

if \_\_name\_\_ == '\_\_main\_\_':

print load\_nordea(

"""

<LoanResponse>

<interestRate>4.5600000000000005</interestRate>

<ssn>1234563212</ssn>

</LoanResponse>

""")