KØBENHAVNS ERHVERVSAKADEMI

Final Project Report

Student: Grzegorz Goraj
Date of Birth: 27-10-1982
Class Id: SD18I

9 March 2020 Copenhagen, Denmark

Table of Contents

Introduction

About Yostocks

Yostocks[1] is a startup company located currently at Copenhagen Fintech Lab . 1 Yostock allow its customers purchase of fraction of certain companies stock, which normally, on the stock market is available as a whole expensive stock ie. ‘Tesla’ costs 315.45 USD at the time of writing this report . Yostocks on the contrary can sell its user fraction worth 100 DKK, 200 DKK or custom amount - this way making the stock market available to a wider range of investors. The idea challenges the competition and is innovative in the whole Europe. Yostocks is hoping to attract a large number of people starting from Denmark, people that watch investments only in the movies and do not really associate with investing in any way. These common people stand now before the unique opportunity of investing the smallest amounts ever and experience economic growth of their small home budget thanks to Yostocks. In order to enter the market the company needs first to attract the attention of potential investors willing to take a risk of bringing up small inexperienced in the market company and pay for a very costly licence, which grants Yostocks entrance to the local stock market in Denmark. Yostocks is of course willing to expand to other countries of European Union, but in each country they are willing to open a business branch, they have to pay for the expensive process of business evaluation and certification again. Thus here we are in Denmark waiting for potential investors and the question is how do we attract investors, how do we convince them that we can eventually deliver potentially valuable product?

According to Eric Ries, startup should produce a MVP[2] - that is a minimum viable product which would serve as a proof of concept for potential investors, a product that does not cost anything but proves the interest of the clients. Yostocks goes a step further and develops a game for android and ios mobile devices, a game that lets the customer experience first hand how it is to wait for change in price, observe the stock market first hand and imagine how the virtual money could be easily substituted with their own real cash. What brings people on board is also what produces actual numbers and bases for usage statistics of purchases and sales frequencies. Most of all, the company can now be proud of releasing actual

1https://www.yostocks.com/

2Minimum Viable Product - Eric Ries, “Lean Startup” proof of it’s reliability - a stock market game for mobile phone. Real proof that it is in fact really worth it to believe in Yostocks and pay a small price of introducing their project to the real life market.

Inception

Since Yostocks acts currently mainly as a proof of concept in form of android and ios application, based on Yostocks business idea I decided to implement it by applying service oriented architecture[3] using Spring[4] framework that will efficiently support android and ios applications playing part as its backend.

I talked directly with Yostocks CEO Anoop Nair during the time of my internship last year[5] and I have access to the visual prototype of the application made with InVision[6]. Based on that prototype and multiple meetings with him and his business partnerI predicted what endpoints and logic is necessary in order to bring this product to life. We all agreed that this product is a good starting point to build the future real life project and experience that I gain working on the game will pay off in the long run.

I decided to use the Spring framework as I have good experience working with it during my education in KEA. Although concurrency, asynchrony, thread pools and caching as well as resource locking were the terms I only knew from theory I knew that applying it in this application will be crucial for it to work properly and so it was in fact.

Plan was to build this application according to SOA creating most responsive, reliable, durable, scalable, secured, interoperable and most possibly loosely coupled since software oriented architecture is more scalable than monolithic, more modern and it is definitely a good step towards further maturity of Yostocks IT system[7].

Scaling out won't be an issue for a couple of years, because for any company dealing with stocks a licence is required and in each country where a branch is being opened a new licence has to be paid for. Since Yostocks is a young startup and located in Copenhagen we are only taking Denmark under consideration as a location for the next 5 to 10 years will not change.

3https://www.ibm.com/cloud/learn/soa
4https://spring.io/why-spring
5Autumn 2019
6https://www.invisionapp.com
7https://www.opengroup.org/soa/source-book/osimmv2/p2.htm

Project Requirements

Functional Requirements:

- User can not invest less then 100 Dkk
- Authentication is required and should be implemented with consideration of mobile application for Android device which means issuing a limited ttl[8] authorization token[9] issued by a proxy.
- Authorization levels: user, pm, admin - mainly ‘user’ level used in the project
- Remote Apis :
  - Quotes Api For Yahoo Finance[10] - it’s free and does not require a token to communicate with. Offers: historical stock data in form of ‘closing price’[11]
    and current stock price which is moving very dynamically for some stocks.
  - Currency Converter ‘Fixer’[12] - cost: 768$/year for 500k calls or 10$ for 10k/month
    Although it is costly, it could be used for free if application was cashing the response every hour since it’s only a game and Dollar and Danish Krone do not fluctuate that much.

8https://www.oauth.com/oauth2-servers/access-tokens/access-token-lifetime/
9https://auth0.com/learn/token-based-authentication-made-easy/
10https://financequotes-api.com/
11https://www.investopedia.com/terms/c/closingprice.asp
12https://fixer.io/product

Additionally I also confirmed following user stories according to witch the application is currently being developed:

Id	**User Stories**
1	User wants to successfully create an account, in order to become an active client of Yostocks.
2	User wants to log in, so he is able to securely use the application.
3	User wants to be able to see all possible client activities in order to start building portfolio, update profile, access latest stock related news.
4	User wants to buy a fraction of particular stock in order to hold percentage of that stock as an investment
5	User wants to sell a stock percentage to gain a return on his investment.
6	User wants to see brands of stocks available in Yostocks application, in order to choose whether to invest in it.
7	User wants to verify the history of chosen stock and read its description in order to choose whether to invest in it.
8	User wants to invest a certain amount of money in a particular stock, so he may owe a percentage of that stock which becomes part of his portfolio.
9	User wants to see all his investments in his portfolio, so he can be certain which stocks hi currently owns.
10	User wants to see the current market value of the particular stock in his portfolio, in order to make further decisions regarding it.
11	User wants to see how much cash he invested in order to decide on future investments.
12	User wants to see his return on investment at a particular stock in a portfolio, so he can decide whether to sell the percentage he owns or not.
13	User wants to see the current market value of a particular stock in the portfolio which contributes to his investment decision.
14	User wants to see the date of the purchase of stock in his portfolio, which contributes to his investment decision.

15	User wants to browse stock market news in order to actualize his knowledge of current market changes.
16	User wants to access and see his profile data in order to keep it updated and accurate.
17	User wants to change his password because he decided to update it.
18	User wants to recover his account because he forgot his password in order to regain account access.
19	User wants to change his email address in order to link his yostocks account to a new one.
20	User wants to edit his profile in order to update his phone number.

NonFunctional Requirements

- Performance
  It should take no more than 30 seconds for the system to create a profile for the user. Database capacity should be increased before ever hitting 90%.
  The system should be able to support a vast amount of users. 10 000 per hour
  90% of the requests should have an average response time of under 5 seconds.
  Caching should be used as many times as possible to increase responsiveness of the system, it is especially crucial in case of usage of external services with stress on these once that application has to identify itself using private access token to limit the data traffic and costs of exploration.

- Scalability
  Application should be designed in such a manner that could be scaled horizontally and up. System should implement service oriented architecture principles and each service should be dockerized for dispatch with Kubernetes under the linux operating

system.

- Reliability
  The system’s user-specific functionality should have a failure frequency of less than 1%.

- Security
  Users’ passwords are stored coded in the database.
  There must be a proxy controlling each request and filtering out the ones that are not whitelisted. Proxy should route appropriate requests to appropriate services performing filtering and validation based on the user role. System should implement a solution that offers user registrar and issues JWT token which would be added as an authorization header in each request that has to be validated.

- Supportability
  Should be able to further develop, maintain and stop individual system features without needing to close down the entire system.The service should be available in English

- Usability
  System should be made with mobile applications in mind, mobile operating system agnostic.

- Interoperability
  Subsequent services should be dockerized for linux system deployment.

Risk Analysis

Evaluation risks by impact and probability. Risk Factor is the product. Sort by RF with highest at
the top. The impact and probability ranges from 1-10. I of course feel that there must be more risks involved but due to lack of experience it wasn’t currently possible to identify them. I consider a big disadvantage a lack of opportunity to involve more people in this project, but my efforts of interest my colleagues have not brought desired effect. I mention that because it is not only my opinion that a single brain is not able to grasp all the remaining perspectives and values that group discussion brings to the table. Additionally it is proven that writing test cases for one’s own code is often biased and may lead to decreased testing efficiency and detection of defects. I leave that issue out of the risk analysis scope since I will not find any remedy to that.

Id	**Area**	**Description**	**Rank (RF=i*p)**	**Mitigation**	**Solution**
#1	SW	Concurrency of the requests causing inaccurate data updates or loss that leads to User dissatisfaction.	80=10*8	Check if all requests are handled asynchronously in proper way	Apply Locking strategy to crucial resources, assign task executors to each service and make controllers run asynchronously returning Completable Future
#2	SW	High Latency that Unables Trustworthy and Pleasant User Experience	64 = 8*8	Assign more resources, increase thread pools in each service, apply cashing strategy	Scale services, add horizontal instances
#3	HW	Denial of service attacks leading to Customer	50=10*5	Check Logs	Implement Logging strategy and consider applying

		dissatisfaction			account lockout
#4	SW	Inadequate request validation.	24=8*3	Double Check and correct validation criteria	Revise junit tests again and apply corrections
#5	HW	Database server breaks down	16 = 4*4	Have high availability model that uses replica sets	Get a new server and migrate all the data onto it.
#6	HW	Development equipment breaks down	12 = 6*2	Have proper version control and back-up systems available.	Have replacement hardware ready to pick up the project and continue.

Technical Dictionary

- API
  Application programming interface, used to facilitate the integration between any applications. In our case, it will be used to expose functionality, as well as facilitate the connection between the different applications

- REST
  An architectural style, which we will use to design our API. It imposes a set of rules, which an API should meet, in order to call itself RESTful.

- PostgreSQL
  Open source object-relational database, with a strong reputation for reliability, feature robustness, and performance. That's where all of our data will be stored.

- Redis[13]
  Open source in-memory data structure store which this application is using for cashing values such as:
  - Value of 1 USD converted to DKK - updated each hour

13https://redis.io/topics/introduction

    - History of single stock
    - History of All Specified stocks

- JSON
  A lightweight format for storing and transporting data used here to send data from and to the server.

- FRACTION
  Stock is usually sold as a whole asset. Prices vary from 500 to 10000 DKK.
  Fraction represents percentage owned by investor - User

- LOCKING
  BuyFraction request is routed by Zuul Gateway to Stock-Service and a new thread is added to the service's thread pool. One thread updating entries in the database can override a particular row that is being updated by another thread. Locking is a mechanism that ensures that a resource obtained during a database session will not be modified by another thread. I will use **Optimistic Locking**[14] to make sure that none Fraction is updated during another update.

- RestTemplate
  The rest template is the central Spring class for client-side HTTP access. Used as a template that allows configuration of the headers, content type and finally sending Http requests and receiving the response. It is also thread safe[15]. Services use it to communicate between each other.

- Hibernate ORM framework and Spring Data JPA[16]
  "Hibernate is a JPA implementation, while Spring Data JPA is a JPA Data Access Abstraction." "Spring Data JPA is not an implementation or JPA provider, it's just an abstraction used to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores."

14https://www.baeldung.com/jpa-optimistic-locking
15https://spring.io/blog/2009/03/27/rest-in-spring-3-resttemplate
16https://dzone.com/articles/what-is-the-difference-between-hibernate-and-sprin-1

Architectural Overview

Yostocks architecture (Figure 1) is built with Spring services which serve as a backend for android application. Auth service implemented with Zuul Gateway built by Netflix acts as a proxy server for the whole infrastructure. Spring security was implemented to provide user authentication and token authorization services. Zuul Gateway registers as a client of Eureka server, fetches service registry which allows it to recognize the proper and actual addresses and is configured to route whitelisted requests to the target services behind the gateway. Each service has to be registered as an Eureka client. ‘Stocks’ and ‘Accounting’ services both fetch service registry from Eureka in order to communicate between themselves successfully. Each service is connected with its own PostgreSQL database increasing scalability, reliability and loose coupling.

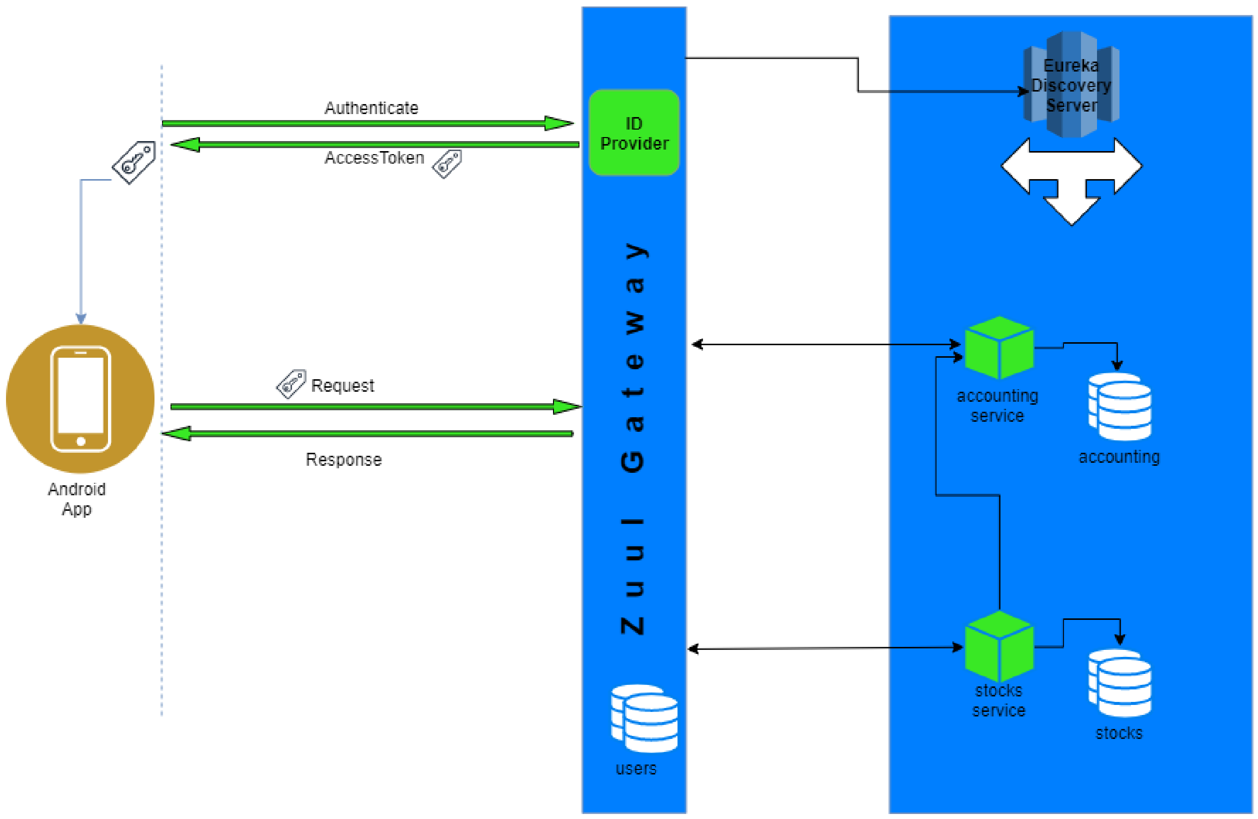


Figure 1

Detailed description of the services:

Eureka Server

Eureka Server also known as Discovery Server[17] is a standalone service registry which automatically detects its clients and enables their discovery by other services:

- 'Auth' service uses it for routing whitelisted requests.
- 'Accounting' and 'Stocks' services use Eureka's registry to communicate with each other.

Configuration

Eureka Server needs to be configured with @EnableEurekaServer annotation added for Application class.
Additional configuration has to be made in 'application.properties' resource file stating the name of Eureka's instance as well as port number of the server. Additionally we add that Eureka should not fetch the registry and it should not register with itself which is only common sense.

In order for Eureka Server to detect particular services, each service that we wish to be acknowledged in the registry should be annotated with @EnableEurekaClient annotation added for Application class. Additionally 'application.properties' should declare:

- server port number
- Service name
- Eureka's client default zone
- Fetch registry: true - really important for RestTemplate to know where the message should be sent within service internal network
- Register with eureka: true

Once we established the name of the service we can now communicate with

17https://spring.io/guides/gs/service-registration-and-discovery/

it using RestTemplate ins following manner:

```
HttpHeaders headers = configureHeaders();
Double response =
restTemplate.getForObject("http://stocks-service/fraction/gain/single/
" + fraction_id, Double.class, headers);
```

Instead of network address and port service name is used which makes auto scaling possible if needed since we would configure load balancer to redirect to the next available service by name.

Auth service

Auth service exposes both authentication and authorization functionalities as ‘sign up’ and ‘sign in’ respectively. Each user is assigned a user role in ‘sign up’ request and service returns the access token on ‘sign in’ request which is then used as authorization method by the android application. Auth service is based on “Zull Gateway” created and maintained by Netflix. The gateway has a built in load balancing mechanism called “Netflix Ribbon” as well as routing and filtering mechanism[18]. Auth service is registered as Eureka Client. Zull in its configuration file ‘application.properties’ makes use of Eureka Server referring to ‘service-id’ property which each Eureka registered service issues.

```
# Map paths to services
#stock service
zuul.routes.stocks-service.path=/api/stock/**
zuul.routes.stocks-service.service-id=stocks-service
```

SignUp Request

It is worth mentioning that in order for the system to work Auth service initializes a new limited balance of 10000 DKK for each new user that is signing up. Auth service is then dependent on Accounting service to which balance initialization request must be first sent in order for the user to be registered.
Request uses RestTemplate in order to send the message and the message is evaluated in Accounting service. If the balance was previously initialized

18https://cloud.spring.io/spring-cloud-netflix/multi/multi__router_and_filter_zuul.html

the
If initialization fails - user won’t be registered and he has to try again.

Part of signUp method:

```
.
.
String response = initialiseUserCashBalance(user.getId());
if (!response.equals("initialized")) {
  userRepository.delete(user);
  return new ResponseEntity<>("sign up failed - cash balance not
initialized",  HttpStatus.INTERNAL_SERVER_ERROR);
} else {

  return ResponseEntity.ok().body("user registered");
}




public String initialiseUserCashBalance(Long user_id) {

  HttpHeaders headers = new HttpHeaders();
  headers.setContentType(MediaType.APPLICATION_JSON);
  InitialiseUserCashBalanceRequest requestModel = new
InitialiseUserCashBalanceRequest(user_id, 10000);
  HttpEntity<InitialiseUserCashBalanceRequest> entity = new
HttpEntity<>(requestModel, headers);
  InitializeCashBalanceResponse response =
restTemplate.postForObject("http://accounting-service/balance/init",
                                   entity, InitializeCashBalanceResponse.class);
  return response.getResponse();
}
```

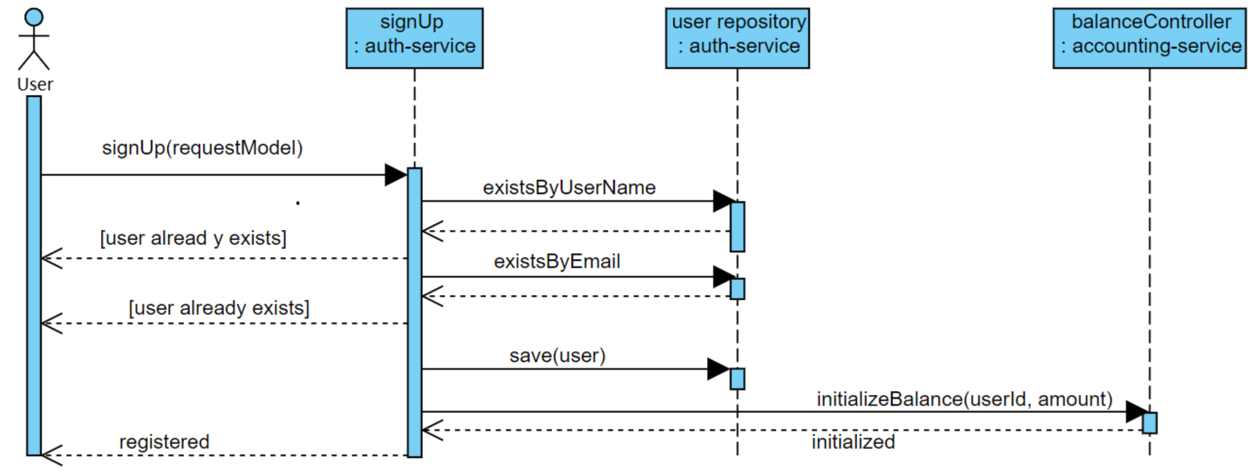
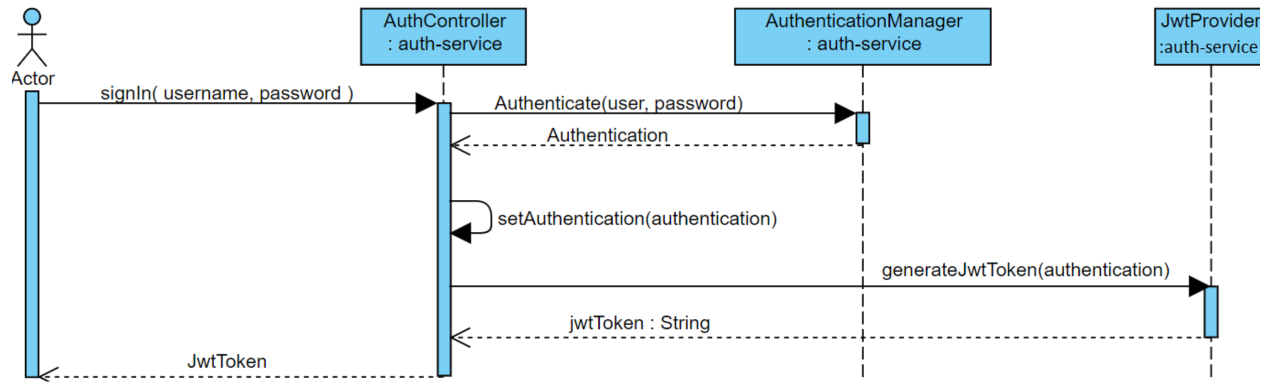


Figure 2

SignIn Request

Second interesting endpoint allows the user to authorize his application so that he may from now on begin his investments.
User acquires an authorization token while providing the correct username and password.



Actor
AuthController : auth-service
AuthenticationManager : auth-service
JwtProvider :auth-service
signIn(username, password)
Authenticate(user, password)
Authentication
setAuthentication(authentication)
generateJwtToken(authentication)
jwtToken : String
JwtToken

Figure 3

List of All endpoints exposed by Auth service:

Reque	Path	Function	Request Body

st Type			
POST	/api/auth/signup	Registers new User and initializes User’s balance in Accounting service	{ "name":**"full name"**, "email":**"bob@gmail.com"**, "username":**"bob@gmail.com"**, "role":[**"user"**], "password":**"password"** }
POST	/api/auth/signin	Returns JWT used to communicate with endpoints that require authorization.	{ "username":**"user@gmail.com"**, "password":**"password"** }
GET	/api/auth/details	Returns Additional User details based on the authorization token passed in request header	
POST	/api/auth/change/password	Updates User’s password	{ "currentPassword":**"dupajasio"**, "newPassword":**"password"** }
GET	/api/auth/balance/{id}	Returns Balanced from Accounting Service	

Accounting service

Accounting is responsible for registering all the purchases and sales transactions for the purpose of providing real time statistics such as Return On Investment in cash and in percentage.

Last but not least, service exposes endpoints used by Auth and Stock services for the purpose of checking on and updating users' cash balance. Accounting service owns its own postgres database named ‘accounting’ and consists of two tables ‘balances’ and ‘transactions’.

I calculated Return on Investment based on Marty Shmidt’s article describing various aspects of investment metrics in “Return on Investment ROI metric measures profitability[19] which proposes following methods of calculation:

Percent ROI = [Current value of Investment - Cost of Investment] / Cost of Investment
Cash ROI = [Current value of Investment - Cost of Investment]

List of exposed endpoints:

Request Type	**Path**	**Function**
GET	/statistics/roi/cash/single/{fraction_id}	Returns cash ROI of single investment
GET	/statistics/roi/percent/single/{fraction_id}	Returns percent ROI of single investment
GET	/statistics/roi/cash/all/{user_id}	Returns cash ROI of all investments owned by the User
GET	/statistics/roi/percent/all/{user_id}	Returns percent ROI of all investments owned by the User

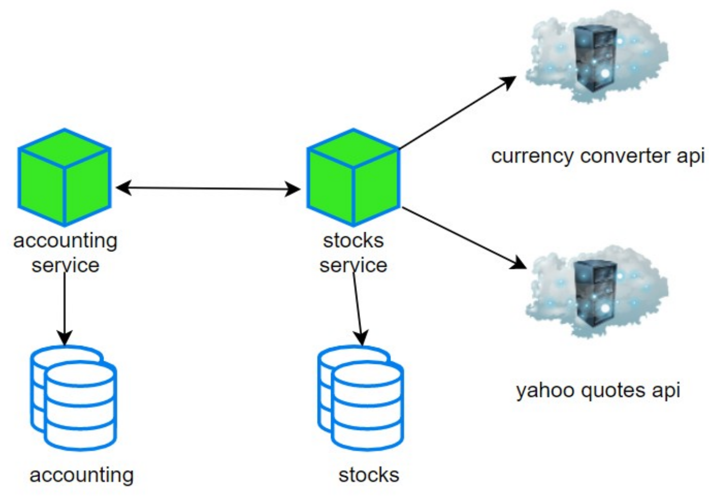
19https://www.business-case-analysis.com/return-on-investment.html

Request Type	**Path**	**Function**	**Request Body**
POST	/transaction/register	Registers both PURCHASE or SALE transaction and updates User’s balance based on the ‘transaction_type’ parameter	{ "user_id":1, "fraction_id":1, "transaction_type":**"PURCHASE"**, "amount":3000, "Time_stamp": **"2020-0 1-0100:00:10.0"** }
POST	/balance/init	Initializes User’s balance by value 10,000 DKK	{ "user_id":1, "amount":100000 }
POST	/balance/update	Updates User’s balance	{ "user_id": 1, "transaction_type":**"SALE"**, "amount": 500 }
GET	/balance/{user_id}	Returns current Users balance	{ "user_id":1, "amount":100000 }

Stocks service

This service is the actual heart of the whole application. Related to current

Yostocks choice of brands as well as related historical data regarding past fluctuations of stocks closing prices[20]. Android application consumes this api and displays it via ‘Stocks’ bottom navigation tab. Users may scroll and choose each row with tap, which results in purchasing a percentage of a particular stock which user finds in Yostocks offer. Service is discovered by Eureka Discovery Service and registered in Authorization Gateway for routing.
Stocks service relies upon 3 other services, two of them are 3rd party services (Figure nr 4). Please refer to the Currency Converter chapter for more details regarding its integration issues.



One of the objectives of integrating all of the services was that at any time if stocks service can not retrieve some response from services it depends on, appropriate response is returned to the source of request.
That would of course mean that ie when OptimisticLockingException occurs, User is informed about the exception in a friendly manner to kindly repeat the buying attempt due to an error.

Figure 4

Buy Fraction Method

Although at the end of the project it seems pretty compact and frugal, it presents itself this way after a long way of design and implementation difficulties that I have not actually foreseen at the very beginning. During the design phase I assumed the application to be closely related to the real world stock selling case. I would imagine that each stock should have it’s

20 https://www.investopedia.com/terms/c/closingprice.asp

own 100% property which would decrease while selling out and while stock percentage equaled zero, some mechanism would create new stock in the database simulating company broker's activity on the stock market. During all of that Stock entity would have to be updated in the loop and Fractions created one by one depending on the request.
That led me to following database design (figure nr 5) and very complicated code functions.

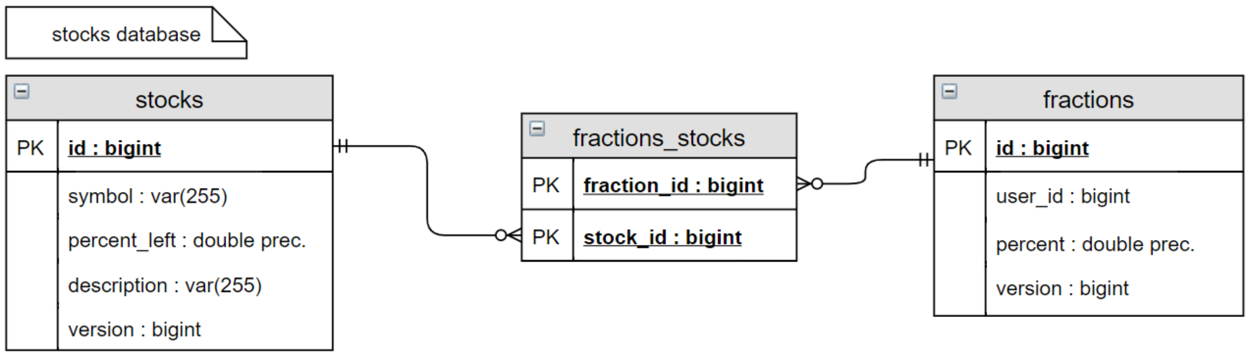


Figure 5

While joined-table is not very problematic in the database, it occurs very troublesome while using Spring Data and Optimistic Locking strategy. Please refer to the repository[21] for details regarding the older version of the 'BuyFraction' method I implemented previously.

Hibernate allows to configure relation owner[22] while implementing Many-To-Many relationships and define join column relation as "fraction_id" and inverse join column "stock_id". Normally the owner of the relation is responsible for updating the whole dependency, but that occured to cause problems with optimistic locking on Stock entity's side because I used only crude repository for simplicity and there was no defined session hence the Stock object I added to Fraction was detached from the database and because of the 'version' property differences couldn't be updated so I would have to call the object again (find it in the database with the actual version) and try saving it.
Each update to 'Stocks' brought the nightmare of optimistic locking possibility so I tested the application using JMeter to see the results. While 30 Users bought the 'GOOGL', 74% of the 'Buy Request' returned 'OptimisticLockingFailedException'. I saw 3 possible solutions: use

21https://github.com/GGoraj/stocks-service/blob/master/src/main/java/com/yostocks/stocksservice/fraction/FractionService.java
22https://www.baeldung.com/hibernate-many-to-many

Activemq[23] for quing messages, wrap whole method in transaction[24] or simplify the code. If there is a easy solution why should I not take it?
I got rid of the idea of keeping track of many-to-many relationship and updating the Stock's percentage on every 'buy' request. After all there is no stock broker anyway, and the sole purpose of the application is to prove that there are people interested in investing small amounts on the stock market with Yostocks and Accounting service transactions registration and the number of active users is enough to prove it.
Final version of the method successfully uses an optimistic locking system and thanks to its simplicity turns out well in tests - only 2 in 50 'buy fraction' requests turn out to trigger the resource lock.

Currency Converter

Currency converter is a service responsible for converting USD to DKK and DKK to USD.
While looking for an appropriate converter api I came across several available solutions. One of them was Foreign Exchange API[25] which is a free api offering exchange rates information. Api endpoint I found most interesting was comparing USD and DKK in regards to Euro:

```
GET
https://api.exchangeratesapi.io/latest?symbols=USD,DKK HTTP/1.1
```

Using this endpoint would provide following output:

```
{"rates":{"USD":1.1052,"DKK":7.4731},"base":"EUR","date":"2020-01-31"}
```

Exchange rates were the only solution provided by this api and as I realized conversion is possible only to Euro ("base":"EUR"), so then 1 USD = 1.1052 EURO, other calculations (from EUR to DKK) is not possible[26].
'Fixer'[27] was another candidate for my exchange service and so I decided to use it because of the following endpoint, which provides ready to implement solution:

```
GET http://data.fixer.io/api/convert?
access_key=d87cefa218dd238422923aa6bc1c0b53&from=DKK&to
```

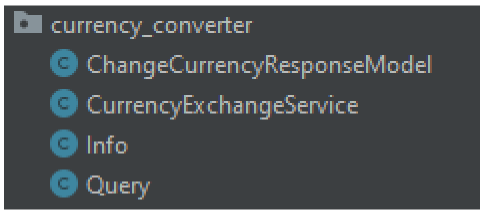
23https://activemq.apache.org/
24https://www.baeldung.com/transaction-configuration-with-jpa-and-spring
25https://exchangeratesapi.io/
26https://www.thebalance.com/how-to-read-and-calculate-exchange-rates-1978919
27https://fixer.io/

=USD&amount=100

Output:

```
{
   "success": true,
   "query": {
      "from": "DKK",
      "to": "USD",
      "amount": 100
   },
   "info": {
      "timestamp": 1580481485,
      "rate": 0.147968
   },
   "date": "2020-01-31",
   "result": 14.7968
}
```

The property to look at is the “result”:14.7968 which is a result of currency conversion between two currencies: DKK and USD.
Provided solution seems too good to be true and so it happens that the “convert” api endpoint is not free of charge[28] and the one I chose has a limit of 10.000 calls per month. Yostocks game would rather need much more than that, therefore I came up with the following solution.
Since prices are updated every hour, the service will schedule a cache update every hour checking how much 1 US dollar costs in DKK and inversely. This way planning api usage was boiled down to 1488[29] calls per month, not to mention that service latency is vastly reduced.
To create Currency Converter Service i used ‘curl’ command to get the return message, JSON formatter[30] and JSON to POJO[31] to easily build required classes. ‘CurrencyExchangeService’ is then but a class annotated @Service and providing the conversion methods used by the ‘StockController’ class.



```
currency_converter
    ChangeCurrencyResponseModel
    CurrencyExchangeService
    Info
    Query
```

28 https://fixer.io/product
29 24h * 31 days * 2 for both conversions USD->DKK and DKK-USD
30 https://jsonformatter.curiousconcept.com/
31http://www.jsonschema2pojo.org/

Adding cashing in Spring application is as simple as adding @EnableCaching annotation to Spring Boot Application file, configuring caching provider in /resources/application.properties class:

```
# Cache - Redis
spring.cache.cache-names=usd-to-dkk, dkk-to-usd
```

then simply annotating required methods: with @Cashable(value = **"usd-to-dkk"**).
Redis provides simple solution caching the value method returns to key-value pair registry (in this case: **"usd-to-dkk"**: 6.743604)[32].

First call to the testing endpoint took 530ms, after that response time equals to 16ms on average. Alternatively @CashPut can be used which actually runs the method every time, but caches the returned value for next use. Since this method returns the result in 200ms on average, I chose to use @Cashable and schedule the eviction of the cache.

Remaining hourly schedule is added as follows[33]:
There are 2 ways to clean the cache:

- Programmatically
- Time Interval

I added @EnableScheduling and set the time interval 1 hour (3600000 ms).
Following code is added to Spring Boot Application class:

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableCaching
@EnableScheduling
@EnableAsync
public class StocksServiceApplication {


@Autowired
private CacheManager cacheManager;

@Scheduled(fixedrate = 3600000)            // execute after every 60 min
public void clearCacheSchedule(){
```

32https://medium.com/@MatthewFTech/spring-boot-cache-with-redis-56026f7da83a
33https://javadeveloperzone.com/spring-boot/spring-cache-clear-all-cache/#Was_this_post_helpful

```
    for(String name:cacheManager.getCacheNames()){
       cacheManager.getCache(name).clear();          // clear cache by name
    }
 }
 .
 .
 }
```

@Scheduled annotation can be used in any situation when we need to schedule a task in Spring, not only while operating on cache. It is possible to configure additional parameters such as condition which will decide whether operation should be run or not[34].

Performance Testing

Performance testing is a process of checking the speed, stability, resilience and responsiveness of the application under test under a workload. It is often used to locate bottlenecks within system boundaries - single point of errors that affect the overall performance of the whole infrastructure. Strength of a chain is measured by the strength of it's single link. Performance testing can help to detect these weaknesses and check if they have been removed properly. It can also help to determine if the system meets it's declared efficiency and resilience ie. helping detecting data leaks or database operations inconsistency.
I will mainly focus on load and stress testing in this paper.

I used JMeter[35] for performance testing. It offers a very simple interface that enables testing api with multiple parallel threads. Apis I tested require authentication and authorization at the Zuul gate(**make annotation)** so I started with creating 500 user credentials with Jupyter Notebook. (**creating_user_credentials_with_python.jpg).** Although it is possible to use csv file with credentials So the request will upload one for each running thread, I found much easier solution which I used from then on:
First I added a JsonExtractor to the sign in request and set up 'names of created variables' and 'JSON Path expressions: '$.accessToken' (response body contains pair { 'accessToken':'"token"'},
Then I added 'BeanShell Assertion' to the same request, which has the capability of setting up global parameter: '*${__setProperty(token, ${accessToken})};*' which can be used in between different thread groups -

34https://www.baeldung.com/spring-scheduled-tasks
35https://gist.github.com/GGoraj/b13f4ded7ed04e5af5c5701d662e82f5

helpful feature.

Testing

AdHoc stress testing ‘Buy Fraction Request’ brought new light on performance of the Zuul Gateway. With 3000 users sending requests following error was returned - not from Stocks service but from Zuul: “REJECTED_SEMAPHORE_EXECUTION”. Solutions[36] I found on Stack Overflow pointed me towards additional configuration of Zuul’s Ribbon load balancing[37] and timeouts. According to suggestions I configured the thread isolation strategy to thread ,
hystrix.command.**default**.execution.isolation.strategy = THREAD
Which led the execution to the following error: “Hystrix circuit short-circuited and is OPEN” and HystrixRuntimeException: stocks-service short-circuited and no fallback available.

According to Ryan Baxter[38] it means that some error is causing tripping in circuit breaker while sending a request to Stocks service, but he doesn’t say what could cause it, but he mentioned that it’s not the configuration issue in Zuul.

New configuration: nr. Of threads: 200, ramp up period 5s, Loop Count 10. Results: 36.4% of BuyFraction requests returned error, pretty decent response times, **figure 6**

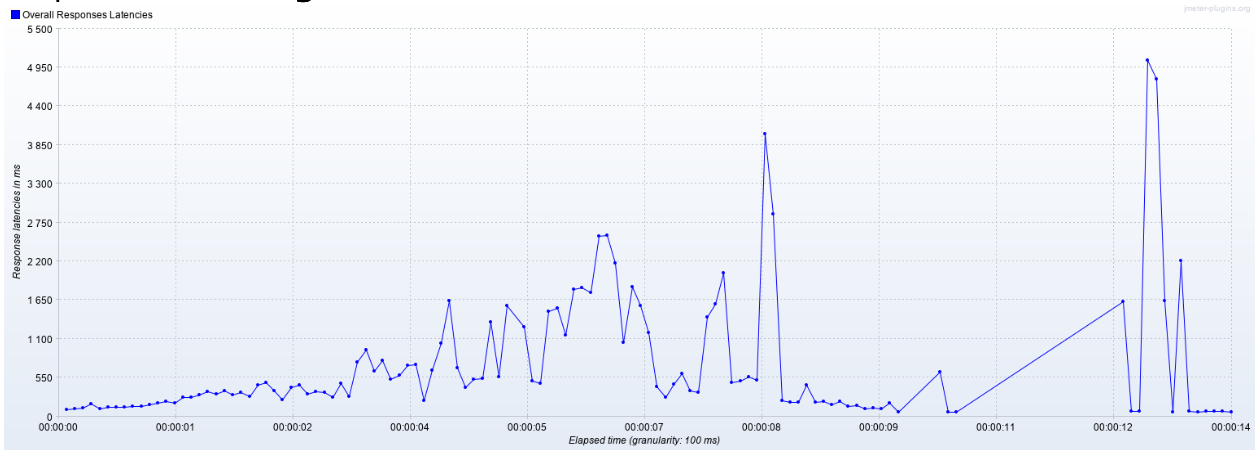


Figure 6

I noticed that yahoo finance service is throwing SocketTimeoutException: Read timed out.
That service is responsible for checking actual stock prices, but I could cache

36https://stackoverflow.com/questions/53653309/spring-boot-zuul-hystrix-short-circuited-is-open
37https://github.com/Netflix/zuul/wiki/Core-Features
38https://github.com/spring-cloud/spring-cloud-netflix/issues/2793

it for 5 seconds.
That solved the issue, but for the same setup there is 41% error: REJECTED_SEMAPHORE_EXECUTION.
I run the same stress test on the isolated Stocks service and I can see now that at some point cause of my struggles is “insufficient credit” error coming from related Accounting service. I use again my favorit reset sql command: “**truncate users** restart **identity cascade**;” and proceeded with a batch of 3000 signup requests.

Farther testing shows interesting error:

```
concurrent.CompletableFuture$AsyncSupply@676f63f6 rejected from
java.util.concurrent.ThreadPoolExecutor@5a25317a[Running, pool size =
996, active threads = 844, queued tasks = 1000, completed tasks =
25322]
```

That shows that perhaps the queue I set up for maximum 1000, in the task executor bean is not enough, and I should make it larger. Increasing it up to 10000 worked.

Another important error: REJECTED_SEMAPHORE_EXECUTION I found a solution on spring-cloud github discussion[39]. I increased the number of max semaphores to 10000 and it looks like this number has to be proportionally larger while proceeding with more users and more requests over time. Here is the code snippet from ‘application.properties’ of Auth service:
`zuul.eureka.stocks-service.semaphore.maxSemaphores=10000`

Additional issues appeared due to HikariPool default settings (default 100). Service returns SQLTransientConnectionException[40] during testing the ‘buy fraction request’. According to post I found on ‘mkyong.com’[41], the HikariPool is responsible for maximum JDBC connection timeout. During further testing I will have set up the timeout to 60000 milliseconds.

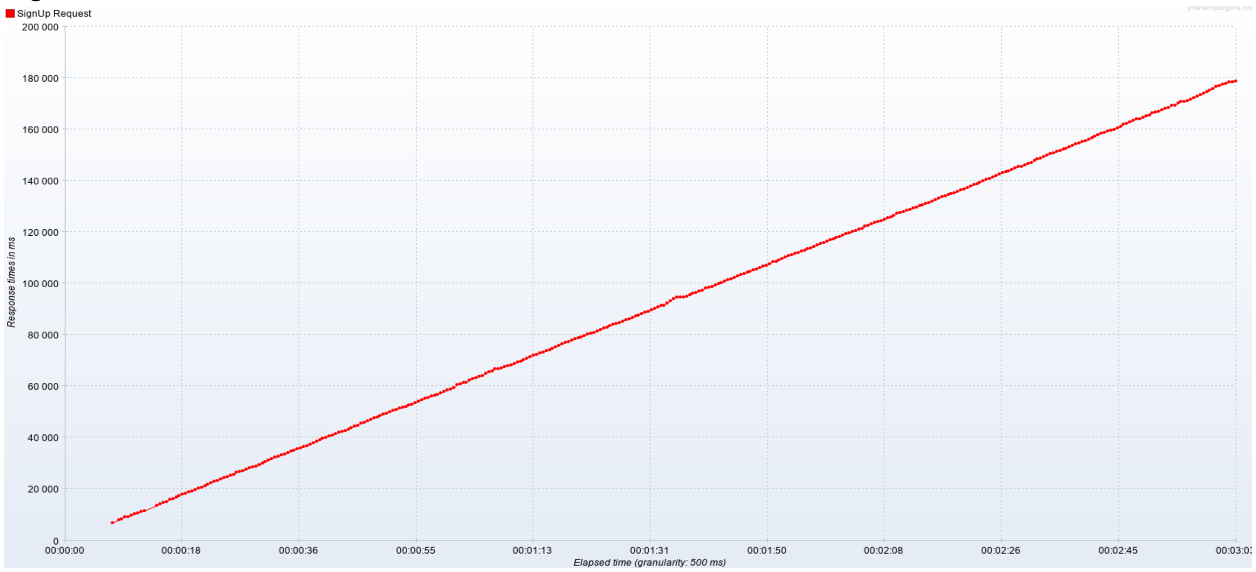
SignUp request

I tested that endpoint with 10000 samples. First errors occurred around 3441 requests and total number of successfully registered users was 9,949/10,000 which sums up to 0.51% which is

39https://github.com/spring-cloud/spring-cloud-netflix/issues/1130
40java.sql.SQLTransientConnectionException: HikariPool-1 - Connection is not available, request timed out after 30001ms.
41https://mkyong.com/jdbc/hikaripool-1-connection-is-not-available-request-timed-out-after-30002ms/

below assumed error threshold of 1%. Minimum response time was 6,7s and maximum almost 3 minutes. We are not expecting this endpoint to be that much occupied of course, but response time should be logged and monitored and additional instances of Auth service should be scaled if the response time is more than 15 seconds. Response time over time diagram (figure 7) shows how the response time increases with constant manners.

Figure 7:



SignIn request

Quite similarly turns out request for authorization token, but with much lower response time - maximum 2 minutes.

Conclusions

I feel I realized all the assumed objectives.
Part of the implementation was successful caching which reduced response time of more complex operations, and I think that api providers were also happy.
I finally could implement and test Completable Future objects together with asynchronous programming as well as Task Executors that allowed me setting up appropriate thread pools to handle large numbers of requests directed to each service.
I'm looking forward to deployment and hoping to introduce my solution during the exam.