A black and white logo

Description automatically generated with medium confidence

**Prototypes document**

OFS Platform

By Mohammed Al Harbi

Contents

[Introduction 3](#_Toc118741177)

[Benefits of prototypes 3](#_Toc118741178)

[Proof of concept per prototype 4](#_Toc118741179)

[Prototype 1: 4](#_Toc118741180)

[Benefits obtained after executing first prototype: 9](#_Toc118741181)

[Prototype 2: 11](#_Toc118741182)

[Benefits obtained after executing second prototype: 15](#_Toc118741183)

[Conclusion 16](#_Toc118741184)

# Introduction

*Going through the theory of microservices, their benefits and how they solve software development issues on a large scale with great possibilities of integrating systems autonomously is great and definitely something needs to be learnt for software engineers. In this document, I will take you through 2 prototypes in which I applied microservices to investigate whether the classic approach (SpringBoot & MySQL) can be used in my case or not and to prove in case I switched to AWS Serverless that I can apply those concepts as some of the functionalities are already built-in from AWS side and I do not need to invent the wheel again.*

# Benefits of prototypes

*Trying as much as possible to meet the non-functional requirements is something I need to do and in the context of those requirements choosing an architecture that aligns with those definitely have huge impact between meeting those requirements and failing in meeting them. Therefore, such prototypes allow me to kind of test beforehand whether a proposed architecture and technology can fit certain requirements and to what extent.*

*My main goal here is in case I have to manage infrastructure of the software application can I still meet my non-functional requirements and where I can deploy my microservices and where the deployment of front-end should be. Additionally, what are the options of testing the loading of my application and where I can monitor my microservices RAM & CPU usage. Having some practical sense allows me to picture whether taking a traditional approach with a complex architecture can actually meet the non-functional requirements or not.*

# Proof of concept per prototype

*I have mainly been working on 2 prototypes which would help me invasion whether or the traditional approach (SpringBoot, Kafka, MySQL) can handle the non-functional requirements.*

# Prototype 1:

***The first prototype*** *is simple software that consist of 2 microservices first one being the WordManagerService (Java SpringBoot Maven) microservice which is connected to a MySQL database to perform CRUD (Create-Read-Update-Delete) operation via RESTful API. The second microservice is WordProcessorService(Python Flask) microservice which messages the WordManagerService to retrieve words than recalculate the size per word based on frequency of appearing in database then exposing the results. The 2 microservices are Dockerized and hosted on DigitalOcean.*

*This prototype contains two interfaces, first one is a phone interface (React application) that takes word input and pushes it to the microservices. The second interface is a big screen interface where all the resized words are obtained displayed in screen.*

*Diagram

Description automatically generated*

*Below picture is an evidence to Dockerizing the WordManagerService microservice which I called ‘yc’ this stands for Youthcenter*

*Graphical user interface, application

Description automatically generated*

*Below picture is an evidence of hosting both microservices in digitalOcean:*

*Graphical user interface, text, application, email

Description automatically generated*

*Below pictures are evidences of using an online MySQL hosted database the wordManager*

*Graphical user interface, application

Description automatically generated*

*Graphical user interface, text, application, Teams

Description automatically generated*

*Below is a snapshot of Memory usage and CPU usage in the WordManagerService where the CPU usage is really low and Memory usage is around 35% for each instance (Here I experimented with horizontal scaling as I have multiple instances for microservices.)*

*A picture containing diagram

Description automatically generated*

*The following is log of the wordProcessor microservice and the calculations results:*

*Graphical user interface, text

Description automatically generated*

*The following shows how changes pushed gets deployed automatically to host provider (in this case DigitalOcean):*

*Graphical user interface, text, application

Description automatically generated*

*The following pictures are proof of concept live (in real life):*

**

*(in website: https://akkasa.om/):*

Text

Description automatically generated

*Finally, I have searched about simulating load test as part of experimenting how the end software enterprise will be tested and most of the options are paid but I managed to find one option that offers 50 users simulation with many requests. The tool is named app.k6.io and I am planning to use JMeter to simulate that load with the final form of the individual project.*

*Graphical user interface, text, application

Description automatically generated*

# Benefits obtained after executing first prototype:

*By the time I have executed this prototype I understood that the architecture technology choose can be optimized from the feedback I received from Bartosz,, in this context of such prototype simple CRUD API (firebase) along with serverless lambda function that retrieves words and apply the calculation logic then spitting it to big screen interface would have been a much more efficient architecture then deploying all that in either google could probably lead to super scalable application as there is less code write and thus more reliable and in terms of cost this can be free to deployed it all live.*

*Compared to my traditional architecture I actually had sometime where the database went down and the microservices went down see the picture below:*A computer screen capture

Description automatically generated with low confidence

*The cost of deployment for services to be able to handle around 1k of load is about 48$ (each instance cost 12$, 2 microservices each with 2 instances online in digital ocean is a total of 48$) and if you count the database hosting services which first I used Cpanel database then SiteGround which I paid too around 15$ a month this application can be deployed as a whole bundle at cost of around 63$.*

*I believe I have learnt a lot during the development of this prototype as it can prevent me from just taking the traditional approach which could be unreliable in some cases then ending up paying more than what the software actually worth investing on.*

# Prototype 2:

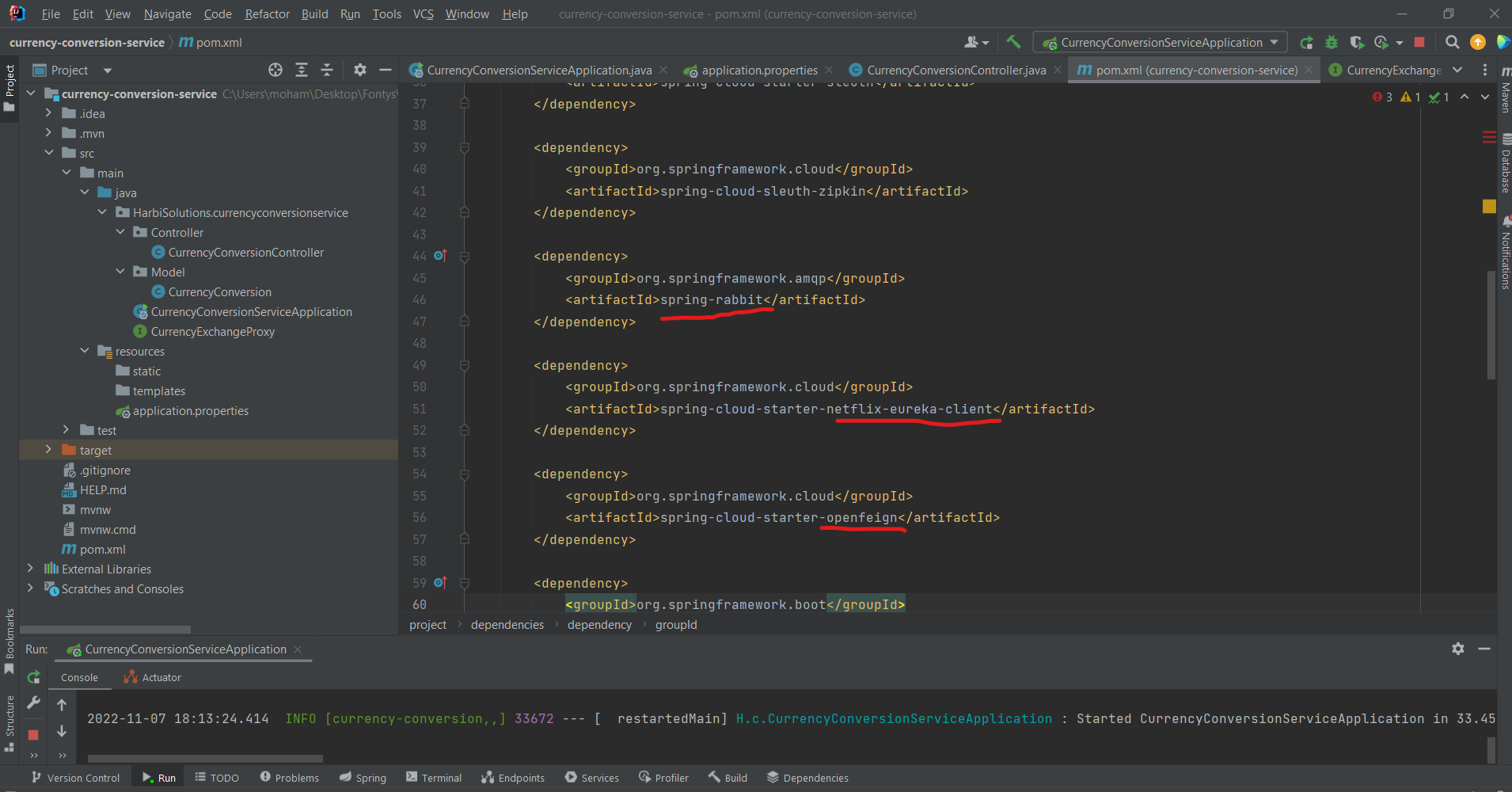
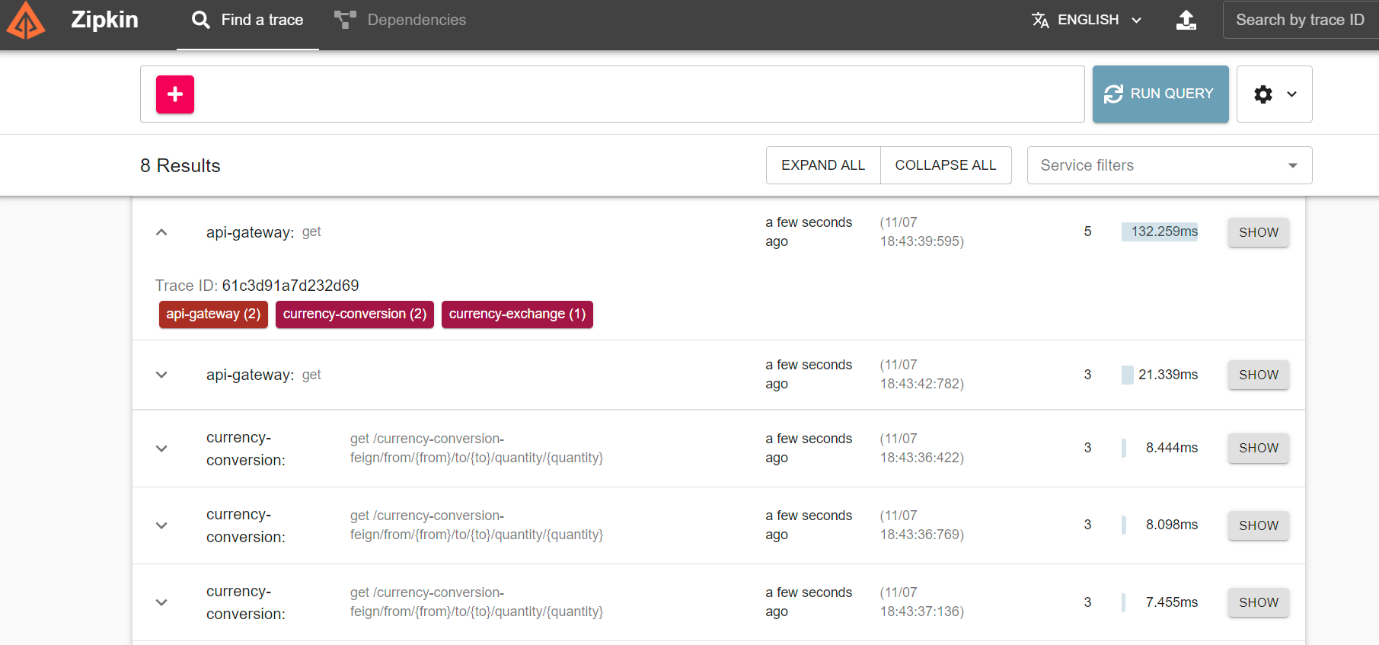
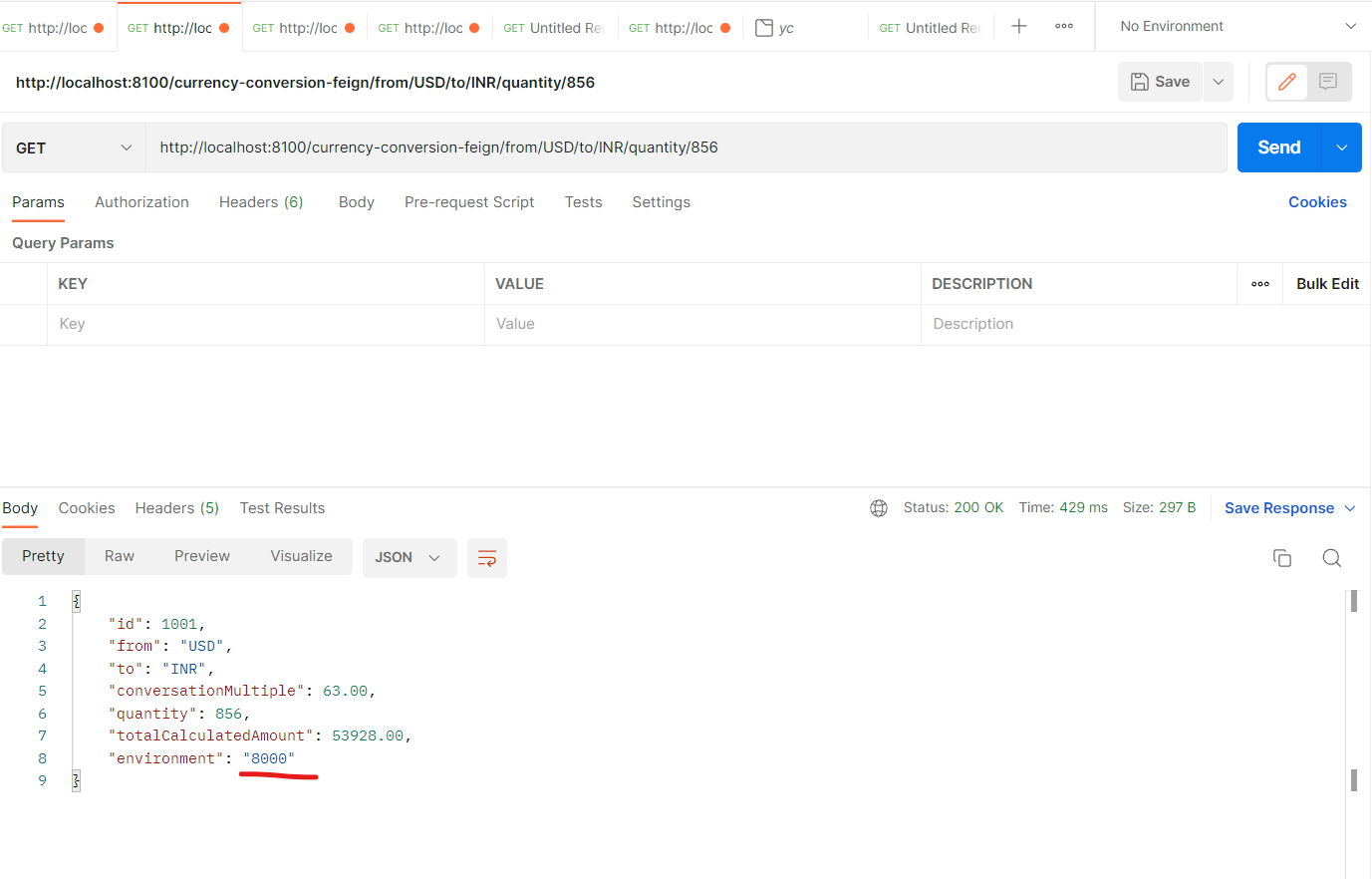
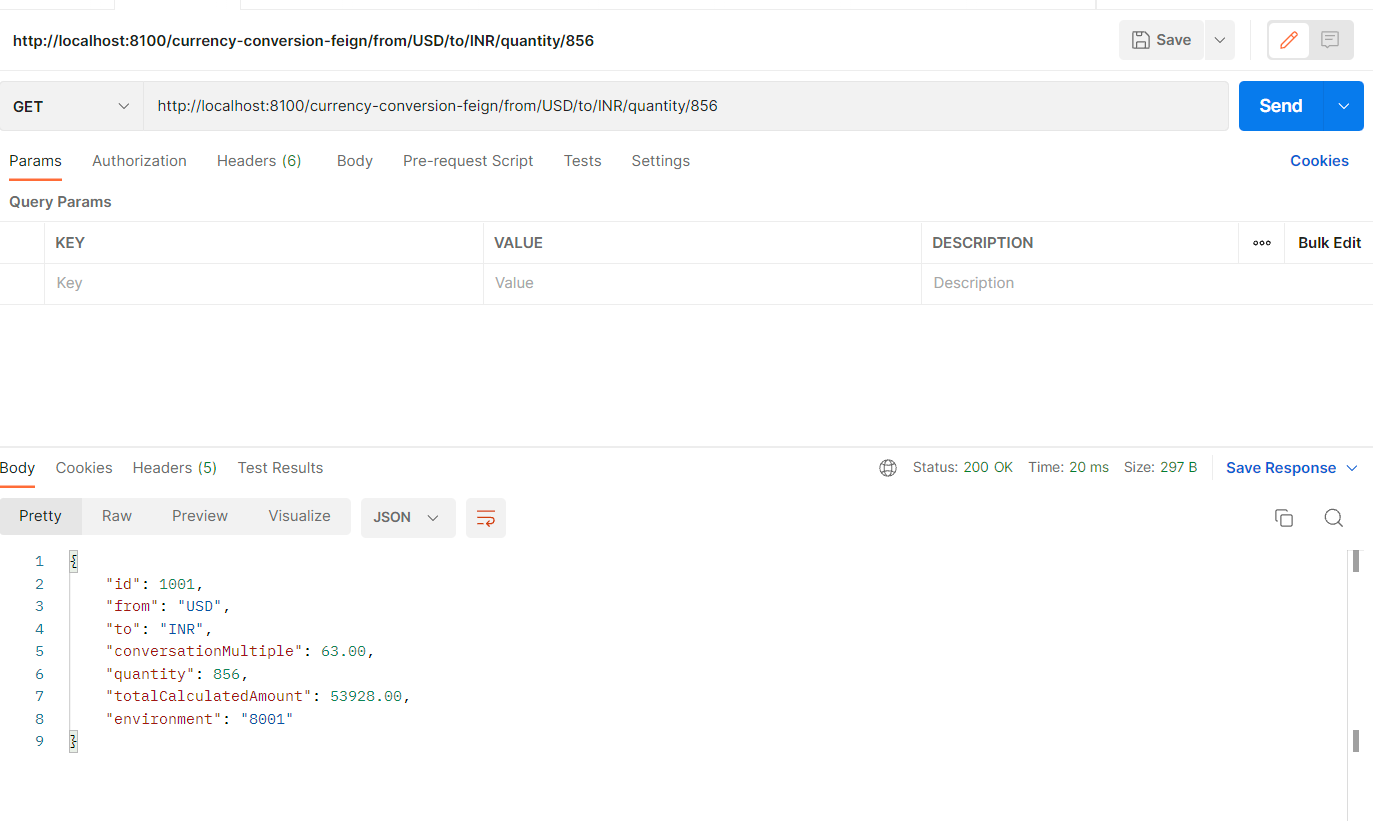
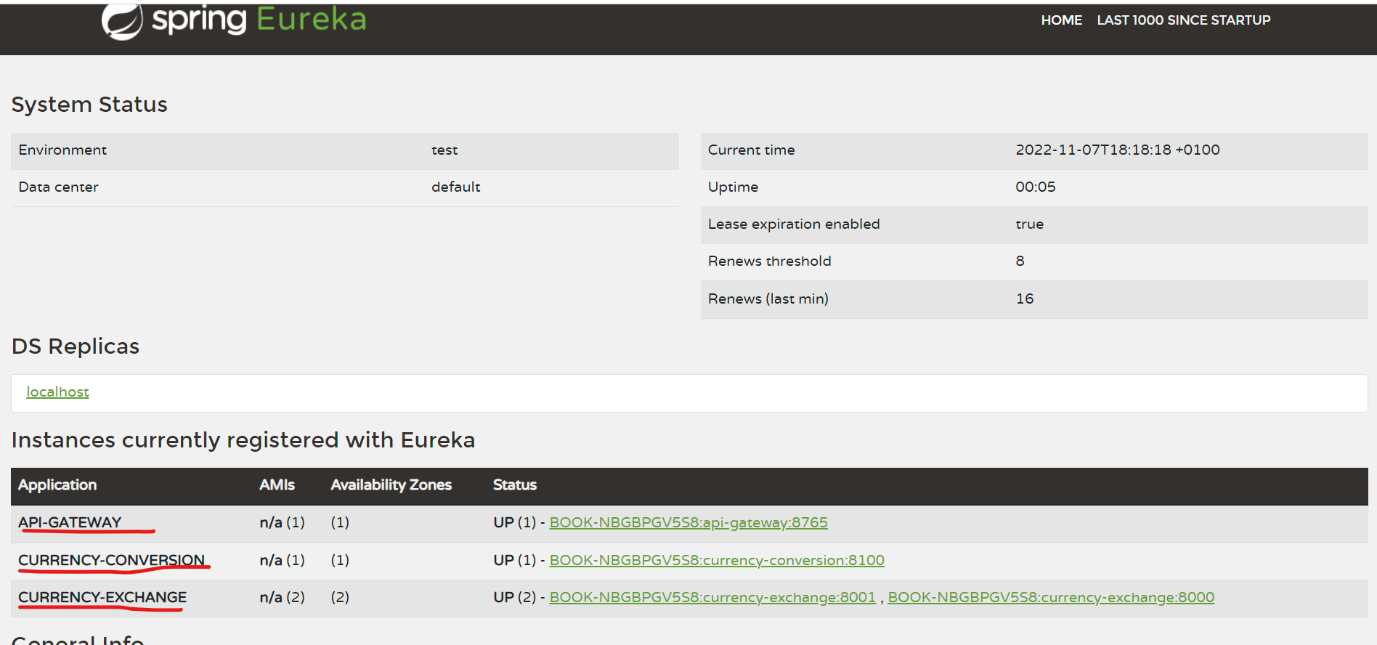
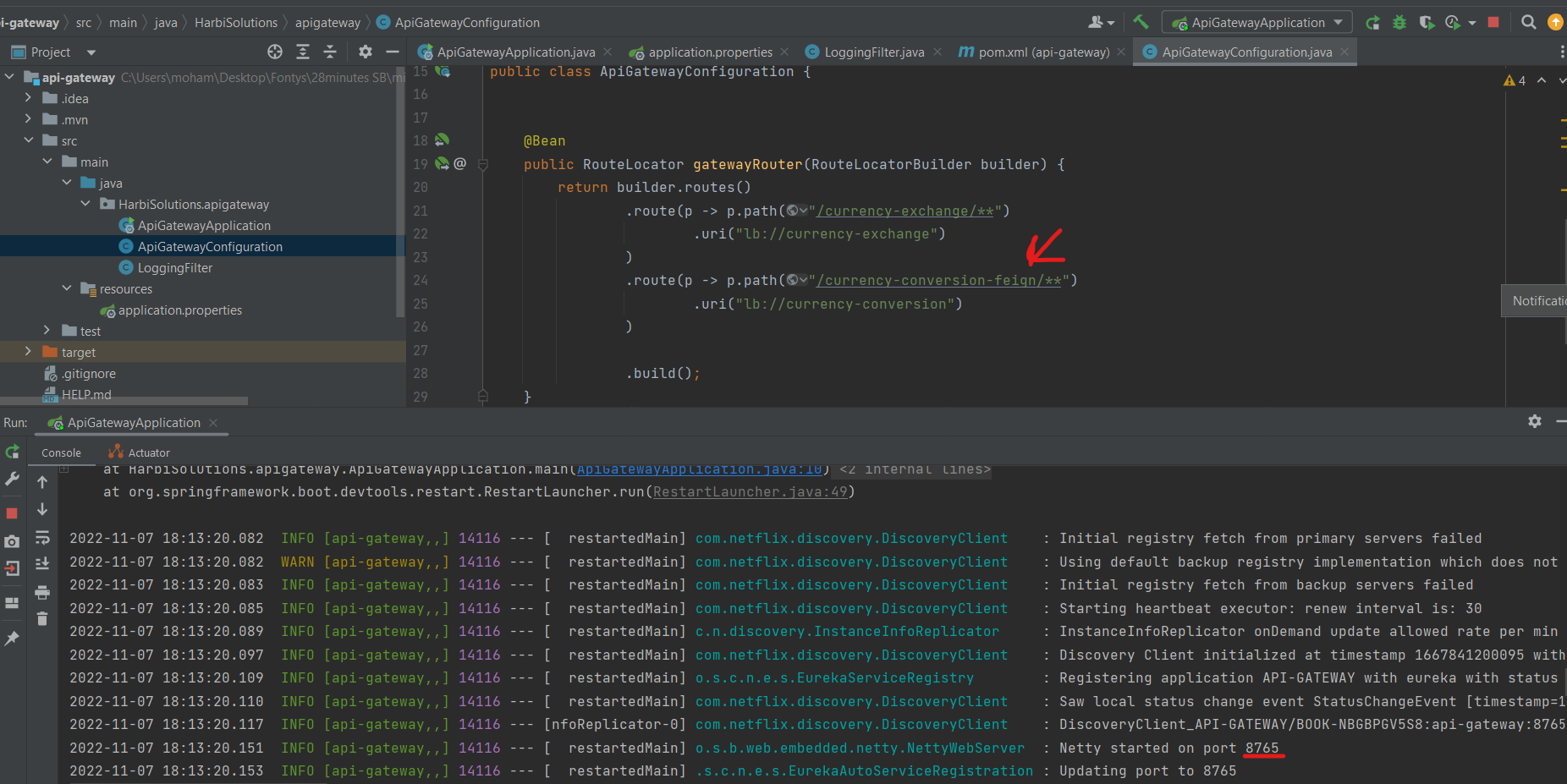
***The second prototype*** *is about applying the asynchronous message idea and apply best practises of microservices. I focused in this prototype in having multiple microservices where the communication between them is done via message broker and I have kept the database consistent for the sake of focusing on applying other microservices ideas.*

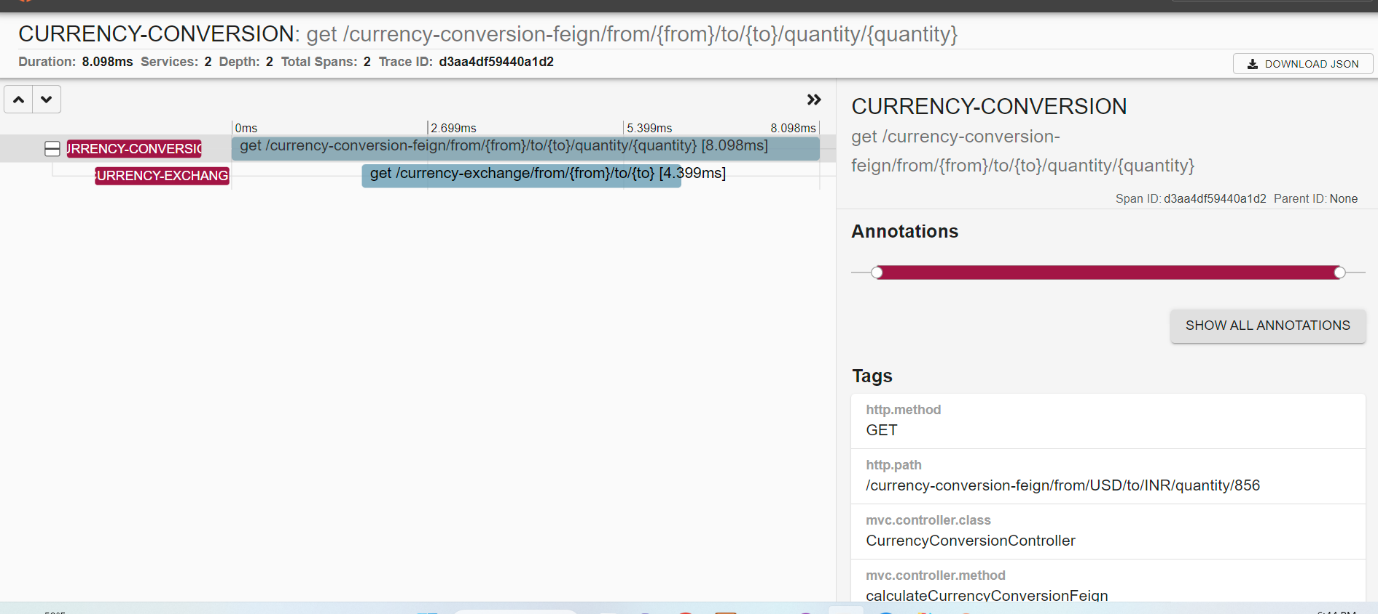
*The idea is about forwarding some information is that present in embedded database which in this case H2 and asynchronously retrieve that via a load balancers the naming server provides out of the box in SpringBoot. The currencyExchange contains the loadings of conversion and we have 2 running instances one in port 8000 and one in port 8001. The currencyConversion Microservice is subscribed to the naming server via load balancer to get the loadings via an automated discovery approach (regardless of the port we can still obtain the loading via one of the running instances of currencyExchange.) and last but not least I access the naming server via an API gateway which all the microservices subscribed to and thus can provide the entire functionality via API gateway.*

*Chart, box and whisker chart

Description automatically generated*

*Below you can find proof implementation:*

**



# Benefits obtained after executing second prototype:

*I got to experience how working microservices when data is consistent looks like and I applied 3 different common ideas (Automated microservices discovery, universal tracing for monitoring and Accessing the software enterprise functionality via one API Gateway)*

*This proof of concept would not help me achieve my non-functional requirements as pay for what I use is a must in the non-functional requirements and achieving it via serverless will be most optimal choice for my architecture. Despite this, the 3 ideas which I learnt in this proof of concept and got hands on practice with in SpringBoot is applicable for every microservice application and will have to apply the same but with AWS services which I already looked into. (please check Architecture document)*

# Conclusion

*I have experienced with 2 different microservices software prototypes which I made from scratch in order to have some sense experience and take decisions wisely as I am basing my decisions on real prototypes I made. I experienced with DigitalOcean hosting using dockerized containers that I can continuously deploy to dockerhub and one downside in DigitalOcean is that you pay for idle which does not fit my non-functional requirements in OFS project. This prototype showed me how powerful and influential design choices of a software can be* ***before*** *the feedback (Initial architecture for this prototype cost around 63$ a month and not fully reliable if database is down)* ***after*** *the feedback(Free deployment & more scalable and highly available architecture). Additionally, this prototype introduced me to different testing tools which I am aware that now most of the simulation services are paid and I was recommended to use JMeter.*

*Apart from deployment pricing architectural design choices I wanted to get some experience on topic such as messaging between microservices and having an API Gateway and auto discovery. I managed to create 2 microservices which messages asynchronously and to prove that the API gateway balances the load automatically I printed the environment which changes from first running instance to second one based on load. This way I proved I am capable of applying these ideas with the traditional approach even though those would probably not be enough to meet my non-functional requirements.*