**Project’s main goals**

**Concepts of Kubernetes used in the solution**

**Collaboration and project management tools**

**Identification of actors**

**Functional requirements**

**Services Breakthrough**

**Communication protocols between Services**

**Stock Service Class Diagram**

**ProbabilityService Class Diagram**

**UseCase Diagram**

**RecommandationService Class Diagram**

**Project’s main goals:**

* **.NET/C#** *Continue growing in the Microsoft .NET stack, coding in C#, for the 5th time in a year will hopefully get me to take a good grasp of the environment and how to get the most out of it.*
* **Docker/ Kubernetes** *Introducing Docker and Kubernetes into my development profile. These tools were learned but never put into practice. Since scalability is now a big thing in development, thinking of containerization and orchestration is a must for any developer to maintain high availability. This very exact point can make way for me to learn about administration namely DevOps.*
* **Microservices** *Exploring a compact Microservices solution. This mainly summarizes in the communication protocols I’ll be using to complete my goals, precisely* ***Http****,* ***GRPC*** *and* ***RabbitMQ****.*
* **Portfolio Management** *To illustrate all the above, I have chosen to work on portfolio management, illustrating some of the concepts I have learned throughout the last year. This is only here to give functional sense to the development architecture and is not serious or meant for production.*
* **Unit testing** *Learning the different concepts of unit testing is essential to ensure a good architecture and functional components.*

**Concepts of Kubernetes used in the solution:**

* **Resource Type: Deployment:** After processing an application into an image using a DockerFile and pushed into DockerHub, a **Deployment** pulls the image from DockerHub, takes care of its deployment and ensures that a specific number of replica pods are always running containing it.

*In this solution, each microservice needs to have its own pod, therefore its own deployment. Some of the images used are self-developed namely StockService, SimulatorService, ProbabilityService, RecommandationService and ProbabilitySubService. Some other images have been pulled directly from DockerHub such as Microsoft SQL Server 2017 and RabbitMQ-Management.*

* **Service: ClusterIP:** provides internal network connectivity to a set of pods within a cluster. It assigns a stable IP address to the service, and the requests to this IP address are automatically load balanced to the pods that are part of that service.

*Essentially, I’ve given each microservice in my solution a clusterip so it can be then managed by the ingress controller.*

* **Resource: Ingress Controller:** manages external access to services within a cluster. It acts as a layer between the external network and the services running inside the cluster.

*Makes the cluster IPs come to life. Registering each component’s Cluster IP in the ingress controller will enable communication between the services in the cluster. That’s what is done in the solution.*

* **Component: Ingress Load Balancer:** an ingress load balancer is a component in Kubernetes that works with an ingress controller to manage external access to services within the cluster.

*In a scenario where this solution has a front-end application, an ingress load balancer will provide an external port for the client to access the server. Can also be used for testing but that’s not the main purpose.*

* **Service: NodePort:** allows to expose a service externally by assigning a port on each node in the cluster.

*This made the stockservice accessible from outside the cluster on that port. A NodePort was perfect for testing the StockService’s API endpoints.*

* **Resource: PersistantVolumeClaim:** allows you to request storage resources from cluster’s underlying storage system.

*In my situation, I’ve used it to get storage from my disk drive to ensure persistence for the database.*

**Collaboration and project management tools:**

* **Git:** modern version control. Keeps the source code safe and easy to update. Easy to use when it comes to merging. Having a portal inside Visual Studio is a plus and makes the experience even smoother.
* **Microsoft Teams:** a workspace for collaboration and real-time communication.
* **Microsoft Outlook:** a workspace for emailing services.

**Identification of actors:**

An actor represents a user role that interacts with the system under study.

*In this situation, since the solution doesn’t contain any front-end interface, any actor will be making direct api calls.*

* **Client/Investor:**  a moral or a physical person that is willing to invest his money in the Stock Market.

**Functional requirements:**

Functional requirements are essential statements that define the necessary actions or behaviors a system or software must exhibit to satisfy user requirements.

*As mentioned before, the functional requirements have been hand picked to be perfect use cases for the technologies I had in mind. For example, the order placing is complicated and what is showcased in the solution is a simplified fraction of the process.*

* **Order Placing:** an investor needs to be able to sell and buy his stock units.
* **Recommendation:**an investor can have intel on how to spend his money based on his appetite for risk.

**Non-Functional Requirements:**

Define implicit limitations that the system must address.

* **Realism in user experience:** the solution/system mimics real-world interactions and behavior, leading to a more authentic and immersive user experience.

*This is one of the main goals that the author has tried to achieve. The problem was: Since there won’t be no other users for the stock market, how can the stock market come to life and simulate real-life life transactions? This question will be answered in the SimulatorService description.*

**Services Breakthrough:**

* C#/.NET: **StockService:** the main service of the solution. Each other service will be pointed at this.

*Other than taking care of authentication and authorization, StockService will be responsible for storing data for each specific user and each stock. Not only that but this service will be able to take in order requests, validate them and execute them whether they are buying or selling* ***limit orders.*** *Implicitly, StockService will make sure that the appropriate updates will be made to the data (user goods and stock status).*

* C#/.NET**: SimulatorService:** the service that will give life to the stock market.

*This service will try and compensate for the lack of actual real investors in the stock market. Its main component is a recurring job that uses the users with the ids 1,2,3 and 4 (dummy users), authenticates them and decides for them which orders they want to execute. This is done to simulate a real exchange in the real stock market. It’s worth noting that some stocks are purposefully made more probable to go higher than other stocks, as they should be.*

* Python/FastAPI: **ProbabilitySubService:** the smallest unit of the solution

*Given a stock price history, this service, using the normal distribution, will provide a prediction of the probability distribution of the stock’s prices.*

* C#/.NET: **ProbabilityService:** the service responsible for calculating and storing probability distribution.

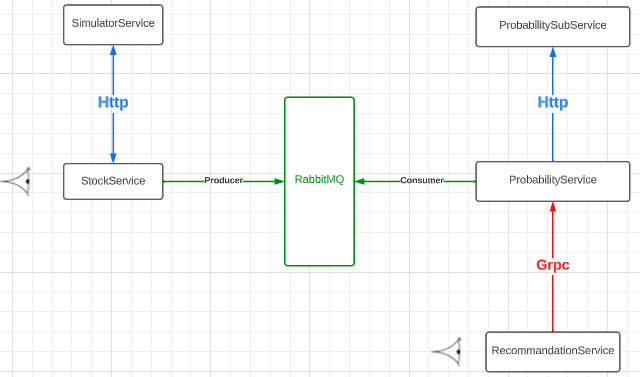
*This service will listen actively on RabbitMQ to follow the changes in the prices of the stocks. Will consume the ProbabilitySubService to determine the probability distribution. A recurring job will be activated each X time to update the data.*

* C#/.NET**: RecommandationService:** the kernel for the recommendation process

*When called upon, this service will take the utility function (a function representing his risk appetite), consume the data he needs from the probability function and will propose a list of orders for the user to execute.*

**Communication protocols between Services:**

*This section will describe the data workflow in the solution and how services are linked up together. Keeping in mind that each service has a pod where it’s deployed, a clusterip has been reserved for each one of them and the Ingress-Controller will ensure that each service each reachable within the cluster/node.*

**

* **SimulatorService 🡨🡪 StockService: Http**

*The SimulatorService will consume data from StockService, gathering information about the* ***dummy users.***

*Then, SimulatorService, in the name of a* ***random*** *user, will post an order concerning a* ***random stock.***

*Since the two operations: the gathering and the insertion of data, the SimulatorService will be considered as a physical person making order creations, making Http a suitable synchronous data transfer option.*

* **ProbabilitySubService🡨 ProbabilityService: Http**

*The ProbabilityService will make use of the logic implemented in the ProbabilitySubService. Since the ProbabilitySubService won’t be exposed for a high number of requests, I decided to make it simple and make Http API calls directly.*

* ***ProbabilityService* 🡨🡪 StockService: RabbitMQ**

*This is where it gets interesting, the ProbabilityService will need to be updated immediately of recent changes in the StockService. So, it’s a matter of a* ***real-time exchange****. The problem lies in the heavy load put on the StockService from external clients. This makes the instant update via a* ***synchronous client (like http****) a hurdle to cross and makes the application significantly slower. In conclusion, an* ***asynchronous client*** *was mandatory. That’s where RabbitMQ comes in.*

**RabbitMQ: an open-source message broker software, enables applications or components to exchange data messages. It follows the “publish-subscribe” or “producer-consumer” model, allowing various software components to communicate asynchronously by sending and receiving messages through a central server (the message broker). This help in decoupling different parts of a system and improving scalability and fault tolerance.**

*To put it into simpler words, StockService won’t need to receive any response from the ProbabilityService when trying to notify about the latest changes. Instead, the StockService will post data into the message bus which is basically a queue. In the meantime, a background service running in the ProbabilityService will detect an event in the Message Bus, decompose it and make the appropriate adjustments to its database.*

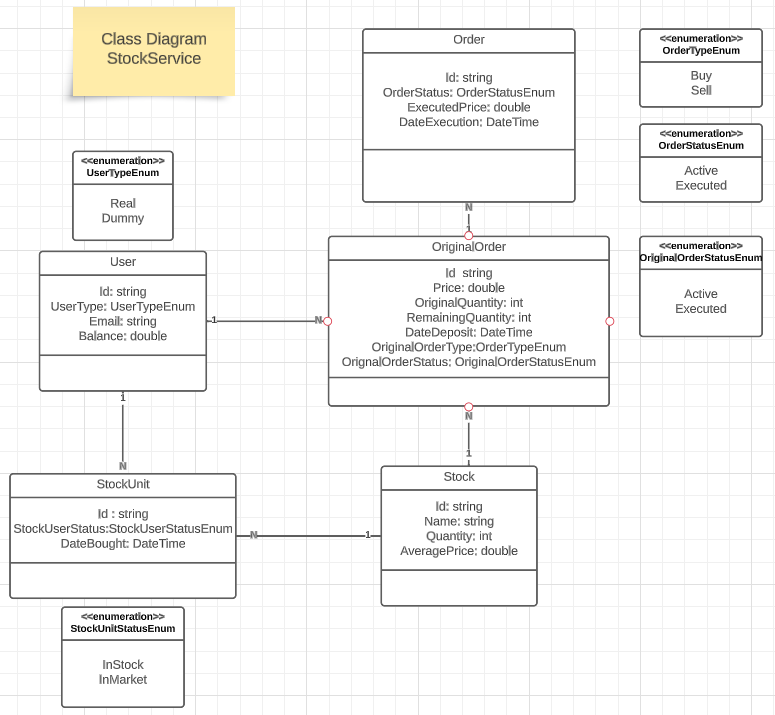
* ***ProbabilityService* 🡨 RecommandationService: GRPC**

*The RecommandationService, when faced with a recommendation request, needs to access data rapidly and will need to ensure that it’s coherent data. That’s where GRPC comes in.*

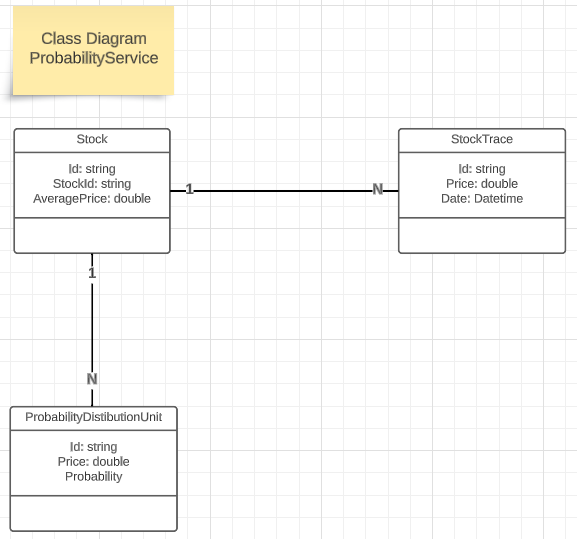
GRPC: an open-source remote procedure call (RPC) framework developed by Google, designed for efficient and high-performance communication between different applications. It facilitates the invocation of methods on remote servers as if they were local, enabling seamless interaction across various systems. One of the key features of GRPC is its utilization of binary transfer, where data is transmitted in its raw binary form rather than being converted into human-readable formats. This binary transfer approach enhances efficiency by minimizing data size and reducing the overhead associated with encoding and decoding. GRPC employs the HTTP/2 protocol for transport, allowing for multiplexing and concurrent communication streams, further improving performance. This framework supports multiple programming languages and provides advanced features like bidirectional streaming and automatic code generation from service definitions, making it a versatile choice for building robust and efficient distributed systems.

*To explain in less technical words, GRPC is suitable for high performance and reliant messaging. That’s why, as a* ***synchronous data client,*** *it was preferred compared to Http,*

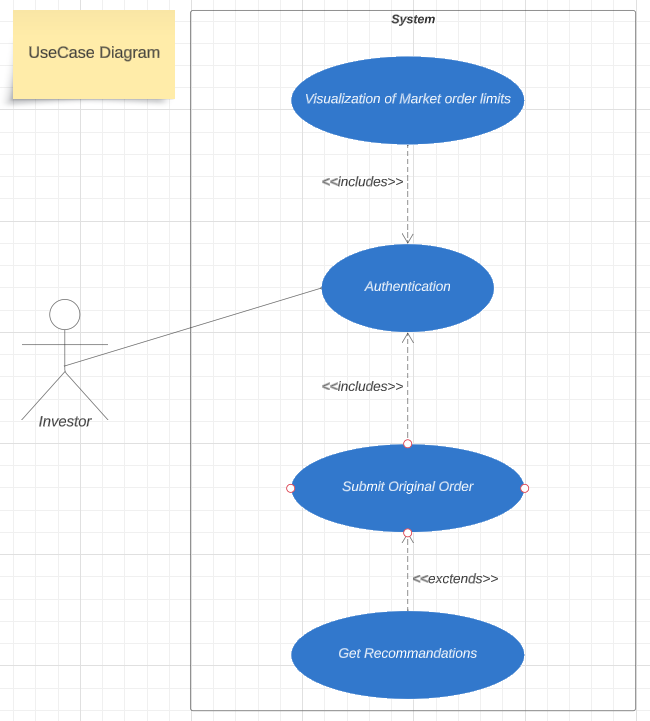
**StockService Class Diagram:**



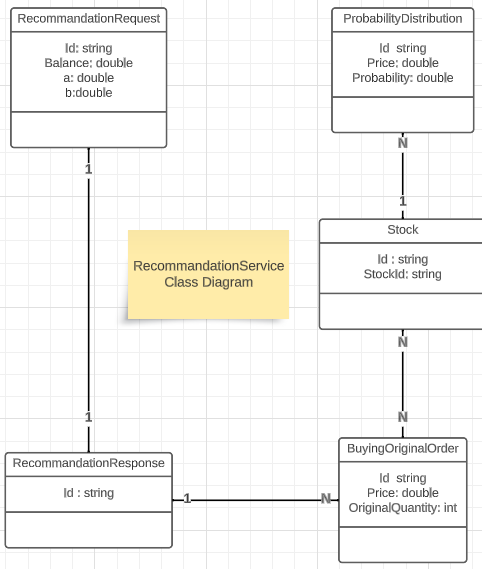
**ProbabilityService Class Diagram:**

****

**UseCase Diagram:**



**RecommandationService Class Diagram:**

****