Real-time Trading Platform

John Costa - 100943301

06 Oct 2022

CS3821 - BSc Final Year Project

Supervised By: Julien Lague

Royal Holloway, University of London

1 Project Description and Motivation

My project will be a trading platform, that allows users to trade - in real time, with others users on the platform, on various assets, for a certain currency. The price of buying and selling is entirely determined by the amount at which users decide to buy and sell assets, very similar to a stock exchange. The user facing application should allow users to access charts of historical asset data, as well as view their current assets and buy/sell other assets. By the term assets, I don't mean real stocks, I mean some hypothetical asset the platform could be used to trade, it can be anything that can be bought or sold overtime, over the internet.

My primary inspirations is taken from the great stock exchanges in the world: London Stock Exchange (LSE), and New York Stock Exchange (NYSE), inspiration not because they are innovative ideas, but because of the cheer amount of data that these institutions needs to process every second, they also cannot fail, because there is massive amounts of money at stake so their technological platforms must be of the most resilliant kind, and every since engineering decision they made has to surround these principles.
I am aiming to provide a solution similar to that of a stock exchange, but not restricted to stocks, my platform could trade any asset. Furthermore, I am aiming to deliver a product that is able to be used 24 hours of the day, all year around. To do this, scalability and resilience must be at the core of every single decision I make. In order to help me with this goal I will be reading through the book: Designing Data-Intensive Distributed Applications [**?**]. In this book, Klapmann describes various aspects of building distributed applications and how to deal with the various challenges that come with that as well as how to quickly handle all the data that flows through them. Furthermore, an objective of my project is to provide real-time, blazingly fast service to the users exchanging assets. This is also the hardest part of the entire project, because everything will have to be built to not just handle hundreds of requests a second, but also scale to meet thousands of requests a second, providing a great challenge on my software architecture, as well as my backend design. The way I aim to solve these problems is by having a micro-services architecture [**?**], which splits the entire application into smaller services that can each scale individually, as well as each service having their own database. However this presents a challenge of its own, how do the micro-services interact with each other efficiently? I plan on using RabbitMQ [**?**] as a messaging service between each service, it enables for easy communication between services, supporting various protocols, the one I will attempt to use is AMQP 0-9-1 [**?**], allowing me to create queues between services that are efficiently managed by RabbitMQ [**?**].

In order to test that my solution does actually scale when various users are using it, I will create BOTS, which will trade in a random(ish) way. The point of this is to have accounts trade with each other in real-time and to be able to precisely control how many accounts and trades take place on the system at once. This

specific testing will be very useful to really stretch my system to the max, and also see how much hardware my system will require to run a certain amount of accounts or trades. To do this I will not only need to create the BOTS, but also run them across various machines, because running them all from one machine would invalidate testing because they would come from the same network, I plan on using various machines across the internet (Virtual Machines (VMs) from AWS [**?**]), as well as the machines I have available to me at home and colleagues. Apart from testing with BOTS, I will also include human trades to the mix by using the platform myself and having my friends try it out at the same time. What I will measure on these tests will be the average time a trade took to complete, as well as other requests that the users make to the server, and wether the system is able to maintain a consistent average time or if it spikes.

I have not mentioned the specifications of the trades - does the user need a certain amount of money? How would I create an eco-system of this sort? I cannot allow the users to use actual money in the product, and giving them infinite credits would discredit all trades because there would be no value in any asset. So I will instead a limited credit system, where each account gets a certain amount of credits to start (£50 000, or any other value), the important part is that the supply has to be limited in some way. Of course that people could however just create new accounts and get another set amount of credits, but for the purpose of my application this system will be sufficient, I could furthermore restrict the amount of accounts that I allow onto my system, precisely to reduce this problem.

How does this then relate to Advanced Web Development. The only way that users will be able to interact with the platform is through a web interface, and to allow for such a complicated task, I need to use advanced concepts of web development, such as Single Page Applications (SPA), and multi-threaded processing on the client side using Web Workers or Service works. Furthermore, because the connection between client and server must be real-time I will need to use Web Sockets (WS), which is an application layer protocol for exchanging data through a persistent TCP connection.

Summarising, my project goals are as follows:

- Have a backend which can scale to handle hundreds of requests per second.
- Build a system that has very close to 0 downtime, and that is resilience and doesn't fail.
- Interactive web interface that allows users to perform trades and view information about assets.
- Allows users to trade assets in real time with very low latency.

1.1 Technology Stack

Which technologies are best suited for the problem at hand? All of them must be able to perform quickly, as this is the core feature of my application. These are only the core technologies I know I will end up using in my application, but there will be various others as the project gets underway.

- Golang [**?**]: A backend language, notorious for building scalable and reliable software. Many technology companies use Golang as their backend technology of choice, and indeed many open source projects such as Docker and Kubernetes.
- SolidJS [**?**]: A library for building user interfaces, whos main feature is it's performance. It is very similar to React which I have a lot of experience with.
- Docker [**?**]: A containarized environment on which to run software. These containers can very quickly be ran (spin-up) to meet incoming demain, making it very useful when scaling my application.
- RabbitMQ [**?**]: A message broker that allows distributed system to communicate with each other, implementing many messaging protocols such as AMQP [**?**].
- Postgresql [**?**]: A relational database, well known for its stability and performance. In this paper [**?**], you can see how Postgres outperforms competition such as MongoDB.

1.2 Features

To formalise the features that I want in my project, I have created two lists, one for core feature which must be present in the final product, and a few optional features which would be nice to have but are not mission critical. These features do not mention technical implementations and requirements for the solution, and instead talk about what a user of my product should be able to do.

1.2.1 Core Features

- Buying and selling assets in real-time with other users.
- Allow all users to view all on-going and previous trades.
- View historical data on all assets (This includes typical candle stick, and line charts)
- View assets owned by all users on the platform.

1.2.2 Optional Features

- Queue trades, allow users to buy trades a specific times.
- Auto buying/selling. Depending on the price of an asset, automatically buy or automatically sell an asset. In investment terms, an optional feature would be to implement Stop-Loss orders [**?**] and take profit orders [**?**]
- Trading bots to test the scalability of the entire system.

2 Timeline

Most of the implementation of the project should be done within term one, with some final testing going over to term two. Like this I can dedicate the majority of my time in the second term to finishing the report and doing final testing on the project. I will further split my development time into two iterations, and attempt to use an agile methodoly with my development.

2.1 Term 1

This will be where the majority of development work is done. Because a lot of these technologies I am only somewhat comfortable with, there will be a degree of prototyping in early days to make sure that I fully understand the technologies I need to use before making major commitments.

- Week 1: Complete project plan and finalise core features, research technologies that would be suited for the problem.
- Week 2: Prototype with the choose technologies to create a very small MVP which enables users to trade across a Web Socket (WS) connection, from a basic web interface onto a golang backend.
- Week 3-5: Splitting backend into multiple services which each scale independently with their own Docker container. The priority for these two weeks is to get the system to work just as well as if it was a monolith, and to enable for many users at the same time.
- Week 6-8: Frontend application that allows users to see the price of assets and perform sell/buy options at specific prices, these trades must be executed very quickly and it very little latency.
- Week 9-10: Polishing both frontend and backend, this will include various bug fixes, UI improvements on the frontend and some minor optimisations on the backend.
- Week 11: Prepare for presentation.

2.2 Term 2

Term 2 will be more dedicated to finalising the solution and making sure that my solution scales. It will also be about further testing with real users, report writing and fixing the bugs that are bound to come up.

- Week 1: Testing the solution from Term 1 to identify bugs and todos.
- Week 2: Fixing bugs that I discovered in Week 1.
- Week 3: Implement trading BOTS.

- Week 4: Test the solution with the BOTS spread across networks and humans at the same time.
- Week 5: Backend performance improvements, based on the testing done in Week 4, and other findings.
- Week 6: Frontend work to display graphs and charts to the user, as well as other nice to haves in the UI.
- Week 7: Frontend and backend development to enable auto trading with stop losses and take profit orders.
- Week 8-11: Testing the solution and putting fixing bugs that come up. I can also use this time to focus more on writing the report.

3 Risks

Throughout the duration of the project, I am bound to run into some various risks that might affect my ability to complele the project to the intended standard, outlined in this plan.

3.1 Platform Performance

The biggest risk to a real-time application might be performance. This means that the system would not be able to quickly fulfil client orders and result in an overall sluggish product. This would go directly against one of my project goals, which is to allow users to trade in real time, at very fast speeds. To mitigate this risk I will use Test Driven Development (TDD), to ensure the most robust code possible is written, I will also include performance tests (or stress tests), with my TDD process, in order to make sure the platform can handle many requests at speed.

3.2 Platform Scalability

As mentioned above, my platform must be very fast. However there is only so much I can do if I only run 1 instance of my backend. My project must automatically scale vertically (Creating more instances), as my application is not extremely resource intensitive, but there will be many connections at one time, requiring this type of scalability. To mitigate this risk I will employ an automatic scaling solution such as Kubernetes, which takes Docker containers and automatically scales them as the application grows.

3.3 Data Loss

It is needless to say that in a platform that lets users trade assets, it is absolutely crucial that my platform has no data loss, and because my platform is going to scale with multiple instances with multiple databses, it is imperative that no data is lost in transaction between the instances. It is also possible I employ some form of caching such as Redis, and the data here should never be of a permanent type, meaning it is ok to lose the data there. To mitigate this risk I will always use transaction in my databases, to make sure that a user never ends up losing anything, as well as implement some form of automatic backup solution so I can easily roll back accounts in case a data loss actually happens.

3.4 Race conditions

With a micro services architecture and a threaded backend, I am bound to run into some race conditions, specially when the system is handling hundreds of requests a second. This will be hard to mitigate or even plan, however by following TDD and using RabbitMQ I should be able to avoid most race conditions. Another approach is to not prevent race conditions amongst services

that do not deal with data persistence, but focus on the ones that interact with the Postgres database and make sure that that system, never encounters a race condition.

3.5 Poor website design

A risk with me being a Computer Science student and not a designer, is that whatever UI I come up with will not be perfect, and I run into the risk of having a very efficient backend solution, but then have a bad badckend that the user cannot properly interact with. I will take time to research various UI/UX design patterns to at least stick to a few guide lines to deliver a website that is usable, this article is about UX design principles is a good place to start [**?**].

3.6 Poor time management

This 3rd year of my university course will be the business year of my life, so I must use my time management skills to their fullest, and I cannot afford to plan my time poorly and result in me missing deadlines in the project, or moving slower through my timeline than I should. I will employ the help of organisation tools and methods such as the method outlined in the book Getting Things Done by David Allen [**?**], I will also use digital tools to help me implement this process such as: Digital Calendars, Todoist (A task management app), and Obsidean (A note taking app).

4 Glossary

Todoist: A todo list app which allows you to set deadlines, priorities, projects and labels to tasks.

Obsidean: A note taking app which lets you use markdown to write notes and create links between notes to create a graph of knowledge.

Redis: An in-memory non-persistent database, known for its very fast performance, but low capacity.

WS: Web Sockets, a standard application layer protocol that allows for a two way communication session between client and user, usually implemented with a percistent TCP connection.

Kubernetes: A container orchistrator which tells Docker containers when to spin up and spin down to maintain application scalability.

TDD: Test Driver Development, a development methodology which makes the developer write tests first and then write just enough code to make the tests pass, repeating the cycle until the solution is complete.