Real-time Trading Platform - Interim Report

John Costa - 100943301

06 Oct 2022

Final Year Project

Supervised By: Julien Lague
Royal Holloway, University of London

Contents

1 Introduction

1.1 The Problem

For my final year project, I am creating a Real-time Trading platform, with a web interface and a robust backend, allowing the many users of the platform to trade - in real-time - with each other, with very low latency. The reason I picked this project was due to my interest in distributed systems and desire to learn more, but also because I believe that current solutions to this problem are too highly coupled with the service they are providing. For example Ebay's trading and auctioning system is tightly coupled with Ebay itself. Furthermore systems such as the London Stock Exchange (LSE), which I took majority of my inspiration for this project from, are focused on only 1 group of stocks, as well as running on somewhat primitive technology. My goal is to provide a universal system, where users are allowed to trade any type of asset, whose ownership can be determined by a centralized system (This does mean that Cryptocurrency and NFT are not something my project will be able to do).

1.2 Aims and Goals

1.2.1 Low Latency

Latency [10] is described as '[the] time delay between the cause and the effect of some physical change in the system being observed'. This means that when the user tries to buy an asset, there should be very little time until they actually own that asset, the same is true for selling. Latency will have to be something I think about through the entire system, on one hand the UI must be responsive and active, even if it is waiting for some data, but the backend must also process requests very quickly.

1.2.2 Advanced Web Interface

My system will only be as good as the front facing application, that lets users interact with the complex system. The web interface must be featureful, and allow the complete access of the backend system to the correct users, whilst also providing a user friendly way to navigate.

1.2.3 Resilient, Performant, Distributed System

Lastly, my system must be performant for reasons mentioned above. Everything must be snappy and even very high loads should be dealt with with scaling. Furthermore the nature of the system being distributed helps with the fact that there is no single point of failure, therefore making the system resilient.

1.3 Objectives - Milestone summary

In my plan I have outlined 4 objects, I will now review and summarise how these can be achieved.

1.3.1 Scalable Backend

At the very core of my project, is a service which can scale to an arbitrary number of users without problem, therefore every decision I make has to have some architectural rational behind it as well as thoughts about performance to make sure that there are very few bottlenecks in the entire system. Because of this I will build with a microservices architecture, which I have outlined the research for below.

1.3.2 Resilient system with little downtime

Not only does the system need to have good error handling, so that even complete noise as input does not crash the system, it must also be able to load balance and spin up more systems as demand increases, furthermore it must be able to hot-swap [5] whenever there is an update that needs to go into production.

1.3.3 Interactive Web Interface

The project is only as good as the web interface which allows the user to interact with the overall system. If the UI isn't clear, responsive and functional, the users are not going to be able to interact with the system. A good chunk of time must be spent perfecting the user interface to make it as user friendly as possible, although this talk will be hard to complete before building at least a prototype of the backend, however I can starting thinking about how to efficiently fetch data from the backend and update it when needed in the frontend, and after this I can think about design decisions that enable the users to want to use the application.

1.3.4 Allows users to trade assets with low latency

This is the very basis of my project, a platform that allows users to trade assets between each other. In a way this is the only feature of my project, however in order to do this effectively and with very little latency, everything must work flawlessly and very quickly so that my users can actually perform the buy or sell operation they want. The low latency part is extremely important for the system to actually be used to trade assets, because often trading can involve split second decisions to sell at the perfect second and buy at the exact moment a certain price has gone down, without this the system cannot be considered a trading platform.

2 Research

My primary source of information is the book *Designing Data Intensive Application* [23]. This book provides an array of knowledge about building data intensive distributed applications that are reliable, scalable and usable.

2.1 Architecture

Modern systems tend to lean into a distributed architecture, meaning we have multiple services which communicate with each other, but are free to scale as load increases. It is unrealistic in the modern world to assume that all services will have the same amount of load and therefore could be packaged together, it is much more likely specific systems need to be scaled more, and perhaps other systems need to be scaled only at specific times of increased load. This means I need a more individual control over the multiple services that I wish to run. In my case, it is much more likely that looking at the top traded assets will be most of the load, therefore I need to scale the services which provides users with this information and have the information readily available on a cache, instead of fetching from permanent storage all time.

2.1.1 Microservices

Microservices are small, autonomous services that work together to form a complete system [26]. Instead of creating a single application which has a single executable and database (Often referred to as a monolith), we break the system apart into services which all perform one thing, and do it well. These system then communicate with each other via a network (Most often the internet).

There are several advantages to using microservices:

- Scalability: Each service can decide how many instances of it self it wants, this ties is really well with Docker which I talk about below.
- Technological independence: We are not bound by technologies from other services, if we decide a service is better of with a certain database instead of another, we can make that decision. This also goes for programming languages.
- Resilience: If our system contains a singular monolith, there is a clear single point of failure. With microservices, if a system goes down - we lose functionality yes, but we do not lose the entire system as other services may still perform they functions.

(The information above came from this book by Sam Newman [26])

However, this does make the entire development process more complex. We are no longer talking about a single application, but instead about developing multiple, smaller application that must all talk to each other flawlessly, with low latency and must allow for errors to occur in other services without the service itself going down.

Furthermore, it is much more important to keep logs in microservices because bugs are often very difficult to track down because they are rarely localised to a single service, and often happen across multiple services, making it harder

to track down.

Microservices architectures also create a big dependency on standards throughout the application. Every API should have the same routing convention, for example. There must also be a language independent way in which we can define how our APIs are going to work, and how we can communicate with them. The emphasis on standards and how to achieve them can be found here [2]

One of the techniques we can use to better standardise our microservices is by definition how are APIs look like (routes, body required data, etc...), outside of the service we are building. This way we can know how to communicate with our range of services, without having to have some pre-existing knowledge about the code itself. Furthermore this makes refactoring a lot easier, because we no longer need to change code, but instead we need to change some configuration file which defined what the route was going to look like in the first place.

2.1.2 Scaling

Scaling is the ability a system has to increase or decrease resources depending on demand placed on the system. Scaling is essential for a modern system which could at any time see a spike in demand, or fall completely flat at certain points throughout the day. Scaling allows us to be sure our system not only meets user demands, but also scales down when there isn't as much demand, saving money and energy on infrastructure. There are two main ways of scaling an application.

2.1.3 Vertical Scaling

Vertical scaling is the process of increasing system resources to your systems instance on demand. So if the system sees an increase in demand, we might want to increase the CPU or RAM. This is possible because of modern Cloud infrastructure, which can control resources like this.

Vertical scaling is useful for applications which cannot be broken up into multiple instances, however they tend to be limited as the instance has to stay in the same physical location (Not useful if your traffic comes from another area of the world). Also, some problems cannot be solved with vertical scaling, maybe a process is busy waiting, and therefore increase CPU won't increase performance.

2.1.4 Horizontal Scaling

Horizontal scaling is the process of creating more instances of your running application/service. This could mean starting up another computer with the same program running in another region or it could be as simple as spinning up another docker container (more on that below).

Although horizontal scaling is limited when the application is all contained within one package (often referred to as a monolith), we do not have the same problem when our application has been broken up into microservices. With microservices we can take the part of our application that is experiencing increased load and spin up more instances, often these instances could be in another physical location or on the same machine.

Throughout my project I will mostly focus on horizontal scaling, because the technologies I use facilitate them and it tends to be the most flexible with modern architectures.

2.1.5 Docker

Docker is an application which uses OS-level virtualization to deliver application in packages called containers. Docker containers do not include a Guest OS, and only contain the binaries and system libraries necessary to run the required application, leading to an increased performance, decrease container size and more portability across machines. Other virtualization technologies such as KVM [9], require a host operating system in order to run the isolated environment, this means docker beats KVM in terms of performance in almost every regard [27], and it even competes with native performance. To run containers, the host machine requires the docker engine, which acts as a sort of hypervisor. This makes docker extremely portable across machines, as the docker engine is not a full hypervisor and allows the host machine to run normally, as well as hosting the docker engine.

2.1.6 Docker Containers

A docker container is the packaged application, running on top of the docker engine. Containers are "Span up", from docker images which are made with Dockerfiles, these Dockerfiles describe the base image you are using (Ubuntu, Nodejs, Nginx, etc...), and allows you to copy, compile or do anything inside of this isolated environment. After these images are build, they can be ran and turned into containers. As mentioned before these containers only include the resources needed for the application to run, hence the performance improvements over full isolated, virtualized environments.

2.1.7 Docker Swarm and Microservices

Docker swarm [3] is a native way to scale docker containers into thousand of notes. Docker swarm allows for a very easy way to create microservices. We can run our many services in a swarm which will deal with replication for us, therefore we have a better system resilience, but docker swarm also includes load balancing across these replicated instances, giving us better system performance and scalability. It also handles container restarts if any of them fail, and because of the nature of docker, these restarts are often less than a second (obviously depending on the size of the application).

2.1.8 Docker Swarm vs Kubernetes

Another very popular solution to running docker containers is by using Kubernetes [8]. Kubernetes is a container orchestrator, meaning it manages the docker images in what it calls Pods and Nodes. These can individually scale horizontally and vertically, all automatically by Kubernetes depending on load. It can also handle replication and hot-swaps of different containers, across multiple machines, making it the perfect solution for a truly scalable, complex system.

So why am I not using it? It's too complex, my system is complex yes but it is not Facebook level complex, and therefore does not need a container orchestrator as complex as Kubernetes, which would be another technology I would have to learn and manage. Docker Swarm is native to docker and uses the same tools and CLI (Command Line Interface) whilst doing 90% of what I need it to do, which is: Maintain replica instances of containers, and scale up when needed either manually or automatically.

2.2 Advanced Web Development

Over the years, we have put an increased amount of responsibility on websites. Today, websites are the default options for building applications which users interact with everyday. Because of this increased responsibility, developers have had to create frameworks for building web applications, which are reactive and responsive to user input, but also provide a performant experience for the end user.

2.2.1 Web framework

A web framework is a library that allows developers to build web application, by automagically performing certain tasks such as updating parts of the websites UI. There are various web frameworks, ranging from server-side rendering frameworks (NextJS, Laravel), to client-side rendering applications (React, SolidJS), and this is what I will be using.

Most modern frameworks are written in JavaScript, which is an advantage because that is the only language the browser can read (Except for WASM). This advantage often makes the application faster, and much closer to the browser than other web frameworks such as Ruby on Rails, which is written (as the name suggests), in Ruby.

In order to allow developers to quickly build application, and for these applications to be scalable, many modern frameworks use a special syntax called JSX, which is an extension to JavaScript that allows the developer to write HTML-like code inside of their JavaScript files, this means that the markdown for a component lives together with the state of the application, making the

component an entire stand-alone part of the application, which contains its own state, markdown and most often, styling too.

There are a few features that a web framework must have in order to be adequate for my use:

- Reactivity: When new data comes into the application, or the user updates something in the UI, the application must reactively respond to these changes, and show them to the user.
- Component Based: Building a website from scratch without a framework runs quickly into a modularity problem, where you cannot easily separate different parts of the UI from another. Because of this the framework I use will need to separate components into different Classes/Functions, which I can in turn split into multiple files and organise them in a file structure which works for me. These components must be able to control their own state as well as the way they appear in the application.
- Performance: My application's main concern is speed, the user must be able to see, act on and finish trades with other users with very little latency, this means the framework must be fast and have as little overhead as possible.

2.2.2 React

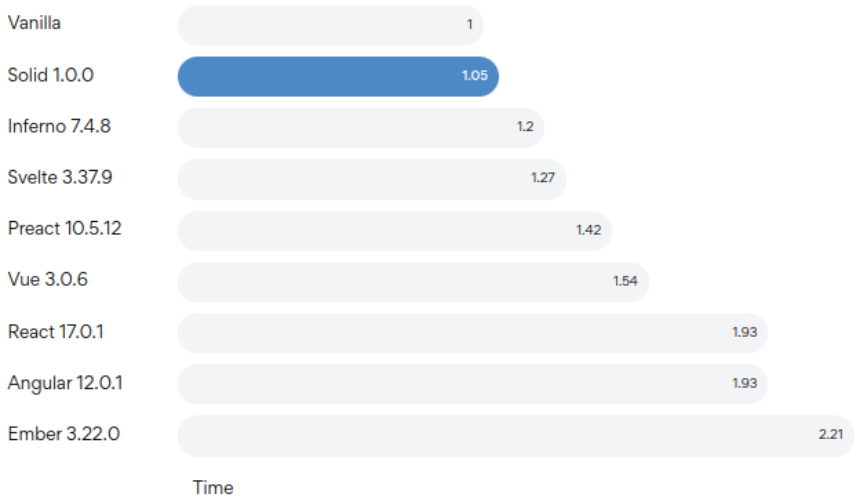
React [13] is a library for building client side application using JavaScript, by separating different parts of the UI into declarative components which manage their own state and life-cycle. It updates the UI by using a virtual DOM (Document Object Model), which updated the real DOM when there are changes in any component, meaning that the entire DOM does not need to re-render when there is a singular change.

Even though React is trusted by many thousands of companies, and loved across the world I have decided to not use it in my project, mostly because of performance. React re-renders whole component when state updates, not only that but it has to re-render child components which can create a snow-ball effect where the entire application is re-rendered. It is hard to avoid this, to do so we have to employ memorization which react supports, but this means we need to write more code simply because React re-rendered every time state is updated.

2.2.3 SolidJs

SolidJS [15] is a client-side web framework, highly inspired by React, in fact programming with Solid is extremely similar to programming with React, which is a very big advantage for me because I have a lot of experience programming with React. However SolidJS beats most other web frameworks in terms of performance. INSERT REF HERE. This is mostly because it lacks a virtual DOM,

Figure 1: SolidJS vs Other frameworks performance: Taken from [15]



therefore component are rendered only once, however it maintains reactivity by making direct changes to the real DOM, when state changes, making massive improvements on performance because the whole component does NOT need to re-render, as it would in React.

There are some disadvantages of SolidJs, most of these come from the very reason that makes it good, the lack of a virtual DOM. Because Solid, needs to update the real DOM programmatically, you need to carefully understand how solid components actually re-render, otherwise the application might not update in some scenarios.

Even though Solid is generally harder than React because it does not re-render as often, its very developer friendly APIs and brutal performance, make it the obvious pick for a performance based project such as this.

2.3 Building and bundling

As mentioned above, most modern web frameworks (React and Solid included), use JSX. This is great for the developer as it leads to quicker development and more maintainable code, however browsers do not natively support this syntax, and therefore the entire application must go through a stage of transpilation (Taking the JSX into regular JavaScript), and then bundling the application into a few JavaScript files which contain all of the components of the application. Therefore, I must use a tool which both transpiles the code and allows me to build it into a bundled JavaScript file which my end users can then interact

with.

2.3.1 Webpack

Webpack [20] is the industry standard tool to perform these functions, it is written in JavaScript and takes your code and outputs bundled, transpiled JavaScript files. Webpack has extensive configuration which allows the developer to really specify what settings they want (What version of JavaScript do you need, what poly fills are necessary, etc...). Webpack also provides a development server, allowing me to quickly view my application without having to completely build it.

Webpack is great, but there are a few drawbacks. First of the webpack configuration is notoriously difficult to setup, there are many options to choose from and a lot of pre-exiting knowledge is needed is order to create a good webpack configuration.

Another drawback is that it is written in JavaScript. JavaScript is great for web application but for intense workloads such as bundling and transpilation, it has weaker performance than compiled languages, making the developer experience here, slower and often less pleasing.

2.3.2 Vite

Vite [19] is a newer build tool, which provides transpilation, bundling and a development server. Vite uses various other tools to complete these tasks and brings them to the developer in a very easily configured tool.

Vite also uses ES Modules, which is a newer browser feature which allows JavaScript files to import and export from other JavaScript files natively, without needing to go through a bundling stage. This means that the Vite Development Server is the fastest amongst tools like it, because it uses ES Modules directly, meaning it only needs to transpile the JavaScript files, but does not need to bundle them. It also allows for Hot-module replacement, which means that when a component changes, you replace that singular component and leave the others unchanged, increasing development speed drastically, because you don't need to wait for a long build time each time you make a change.

Vite also integrates very easily with React, Solid, or most other JavaScript frameworks. Making it often a 5 second step to get started with Vite with any of these JavaScript frameworks.

2.4 Typescript

JavaScript is the language that browsers use to make websites reactive, paving the foundation for web applications, and most user-facing applications. How-

ever, as web applications grew in size and responsibility, it became harder to develop applications which interact with complex data, because JavaScript does not natively support types, as it is a dynamic language, you often don't know the type of something and must perform extra validation in order to make sure data is of the format you expect.

This is where Typescript [18] comes in. Typescript is JavaScript with types, it extends Javascript by including interfaces, type declarations and even classes, each with specific fields. This makes working with complex data much easier as the developer is able to know the format of the data, as well as when they have made a mistake with specific objects.

Typescript compilers down to regular Javascript, but the compiler is strict and will not compile if it detects type errors, avoiding many of the mistakes that web developers face when building web applications. It also makes the code base much more maintainable, due to other developers knowing exactly the shape of the data they are interacting with.

Vite uses ESBuild [4], which is a build tool written in Golang, in order to compile the Typescript down to Javascript. This results in extremely fast developing time, even though your code has to go a compilation stage from Typescript down to regular Javascript

3 Designed Architecture

One of my projects main goals is to have a scalable architecture, which can automatically scale depending on load from the end users, I needed to create a robust architecture, by splitting the project into smaller systems (microservices) which all work and communicate together.

My project is split into the following (micro) services:

- Authentication - User authentication.
- Hub - Initial point of contact for users.
- Brain - Permanent storage and data management.

Each of these services has their own database, as to reduce the dependency on a shared database, therefore increasing the autonomy of each service and increasing scalability as it decouples each system from the others. There is also the downside that, if systems start to share database we begin to lose the source of truth for various events, so if the Hub directly modified the Brain database, the Brain service cannot know how this was changed, and in what context. Below I go into further detail on each of these systems, I have also included a system diagram, in the figure below.

Figure 2: My entire system architecture

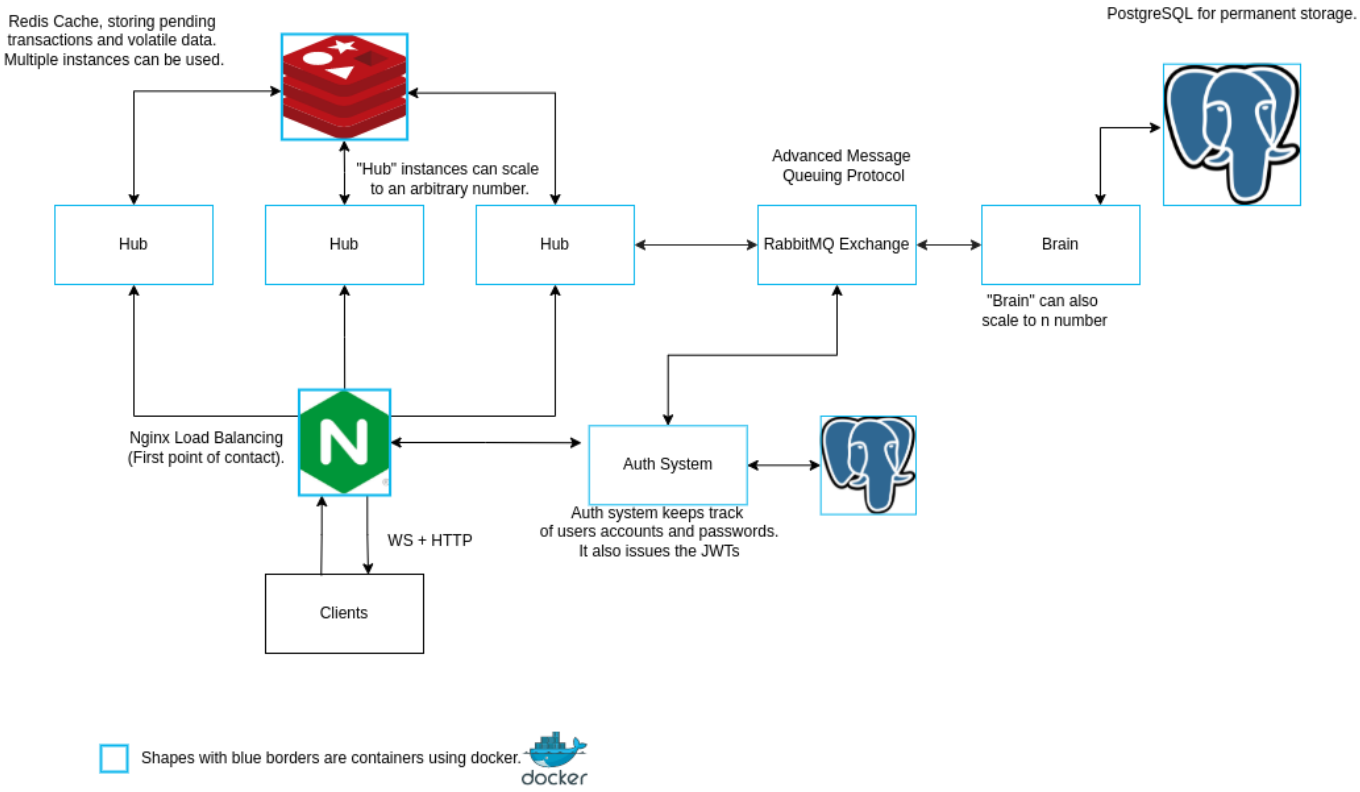
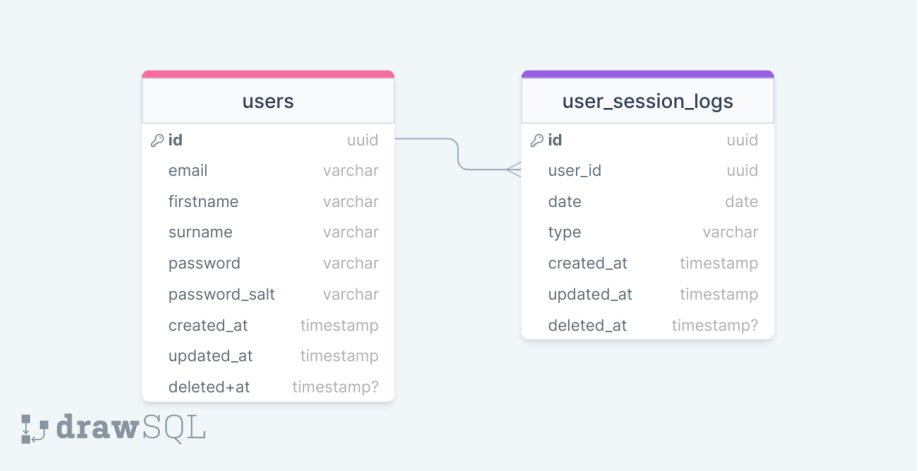


Figure 3: Auth system database tables



3.1 Authentication

This system is the most isolated from the others. As you can see on the diagram is only communicates with the load balancer which comes straight from the user. This is because communication to and from the auth system contains very secure information such as passwords and access tokens, which if intercepted can allow users to access another users account. Therefore I don't want to handle this information for longer than I need to and definitely through the least number of systems possible - like this we go from the user to Nginx (A load balancer), straight to my authentication system.

The authentication system has its own database with a few tables that store user information. The database I choose is PostgreSQL, which as outlined in my plan, is one of the fastest, most robust and safest permanent storage databases in the world, often outperforming competition by magnitudes or performance. The tables for this database are as follows:

3.2 Hub

The hub is the only service without any permanent storage and this is very intentional. Its job is to act as a router for other services, but also a cache. My aim is to have this service serve from cache around 50 percent of the time. The hub communicates with a redis database which is a key pair in-memory database, which acts as a very fast caching system, that stores frequently requested data, as to avoid requesting this data from other systems.

The Hub can be vertically scaled very easily, together with the Redis database

that it users, as Redis supports vertical scaling in a cluster mode. It is likely the system that will be scaled the most, as it will handle all of the users request (except for authentication).

3.3 Brain

A terrible name, but the Brain performs CRUD operations on the assets and transactions for the entire system, it acts as an API which the Hub can communicate with to fetch information from. The Brain like all other services has a database of its own, another PostgreSQL database. The Brain can also scale vertically just like all services but it's difficult to scale the PostgreSQL database vertically because it is difficult to split the data, instead a horizontal scaling approach will be required for the database portion of this service.

The brain also communicates with the Redis cluster which the Hub primarily interacts with. This is to update the cache so that the Hub can in future requests return data from this cache database instead of requesting the data from the Brain. The reason I do this step in the Brain service instead of on the Hub on return of the data, is because I want the Hub to only be conserned with handling user requests and either returning the data from cache, or handing it off to another service, as to not "Hold up the line".

The main database for the system is in the same service as the Brain. This is a PostgreSQL database which stores the permanent information required through the rest of the application. The schema for this is the following diagram.

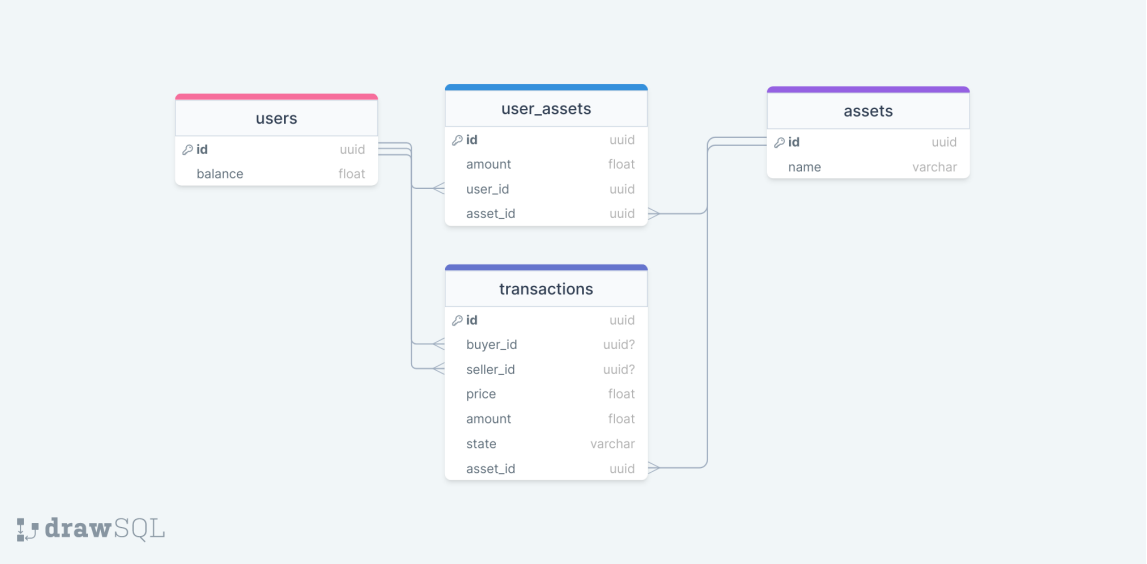
All of the tables in the brain database also contain the fields: created_at, updated_at and deleted_at, the same as the auth system.

3.4 Communication and Load Balancing

I have not yet described how these systems communicate with either one another or the user. To start with the user makes request to Nginx which can act as a very high performance web server, load balancer or reverse proxy. These requests are HTTP(s) requests (and also WS), and therefore this is what the Hub receives from the user.

However, for the hub to communicate with the Brain service it does not use HTTP requests as these would be somewhat inefficient and not complex enough to handle some of the required distributed behaviour of the systems, therefore I am using RabbitMQ to communicate between the Brain service and the Hub. RabbitMQ acts as a message exchange which supports AMQP (Advanced Messaging Queue Protocol), and with this I can create a sort of RPC (Remote Procedural Call), from the Brain service. All whilst allowing many

Figure 4: Brain system database tables



messages from multiple instances of both services.

Microservices architectures generally have the problem of transmitting information across each other, efficiently. To mitigate this problem I have created a library which is shared across my multiple services, that includes the types for commonly used objects. This makes it much easier to share data across system because each one knows the shape of the data, and can easily encode it and decode it without many problems.

4 Technologies used

I have a variety of technologies for my project, these include frontend technologies to create a state of the art web interface, to backend technologies that allow the project to be scaled to as many users as it needs to be scaled to.

4.1 Databases

4.1.1 Permanent Storage

One of the main technical challenges of my project, is choosing the correct database systems. My project requires performance first and foremost and therefore the main consideration when choosing a database is speed. My system also contains highly relational data spread across various areas. Users, assets which user owns which assets and what amount, all this data would be

best representation in various table, and therefore SQL Relational Databases seem to be the way forward, the two most popular being: MySQL and PostgreSQL. After researching their features and performance [22], I have seen that PostgreSQL reliably beats MySQL in most performance tests, with an even more functionalily, making this an obvious choice, for my permanent storage solutionion.

There is another paradigm of databases that have in recent years gotten a lot of transaction, these are document databases. Document databases have much more flexibility than relational databases because they do not (often) contain a set schema, and allows the programmer to easily add/remove parts of data without many constraint. After reading this chapter [24], it is clear to me that applications often have a disconnect from the data on the database and the data they actually require in the application, hence why many solutions turn to Document Databases, as these tend to be flexible and denormalisation is common practice. However these databases have poor (or often no), support for joins across tables/collections making one-to-many or many-to-many relationships more difficult.

Furthermore, one of the most popular and advances document databases available is MongoDb [11], which uses JSON as a method of storage, and has support for data sharding (Splitting into multiple nodes), and (as most document databases do), does not require a schema, giving a lot of flexibility to the developer. However, as shown in this paper [25], MongoDb performs worst than PostgreSQL in almost all performance tests, often by some great magnitude.

This is one of the easier decisions in the project, I am using PostgreSQL as my permanent storage database(s).

4.1.2 Caching

Caching is a common technique to store commonly used items of data, in a faster, readily available way. For this I am using an in-memory database called Redis [14].

Redis is an unstructured key-value pair in-memory database. Because it is in-memory, it is magnitudes faster than any persistent storage database, therefore I have used it as caching in my system, storing popular assets and data that is used frequently.
However, it is important that all the data stored in Redis, can be lost. This means I first have to store it in permanent storage, and only then store it for use in Redis.

4.2 Backend Technologies

These are the technologies which I used to build the backend system, which consists of the Hub, Brain and Authentication.

4.2.1 Golang

As a backend programming language I have used Golang , it is a statically typed compiled language, known for its simplicity and ability to write extremely scalable systems, doing my own research I have found that a lot of modern architectures use Golang Real-time Trading Platform

4.2.2 RabbitMQ

RabbitMq [12] is a message broker. It allows various systems to communicate with each other through a reliable and scalable broker, which can be accessed from various system. RabbitMq is capable of broadcasting messages across systems or simple declaring queues which can be queued and dequeued from the various services accessing it. I am using RabbitMq to organise the communication between the Hub and the Brain service in my project, mostly due to RabbitMq supporting multiple clients accessing the queue at a time, meaning that I can have multiple instances of the Hub and Brain both pushing and pulling from the defined queues without much problem.

4.2.3 JWT

JSON Web Tockets or JWTs for short, are cryptographically safe methods of client side authentication. A JWT is generated in the Authentication system, and signed with a private key, every time the user wishes to authenticate themselves, they serve the JWT to any system, and if the signature on the JWT is correct then the user is authenticated, if not, we know the user either tampered with the signature or it isn't a valid JWT. JWTs can only encode a small amount of data in them, ranging from user information and ids, however this information is assessible to anyone who views the JWT, so it is important that the data is public knowledge.

I choose to use JWTs instead of cookie because they are very safe but still quite practice. Like this I do not need to store cookies in the database and check every time against a database that the user is logged in, reducing the amount of lookups various systems would have to perform.

5 Software Engineering

5.1 Methodology

There are various methodologies used in software engineering, but most of them focus on working with a team of various developers, as well as the client for whom you are making the software. But this is an individual project so I could not simply adopt an Agile strategy or an Extreme Programming strategy, as they aren't applicable to software being developed by one person. However there is a strategy called Rapid Application Development (RAD), which is similar to agile, where it takes an iterative approach to developing software, but excludes daily standups with other teams members, which there obviously aren't any. RAD starts off with a set of requirements which I have outlined in my plan, and it starts a loop of development, taking each feature from design to development. The reason that I chose this methodology is because it allows be to start with very little functionality and iterate over these features until they are complete, or until I run out of time. Because of this I am able to build a PoC (Proof of Concept), quickly, and later on iterate over the feature built in order to perfect them.

5.2 Project Management

My project is quite complex and has a lot of moving parts which all have different criteria and different objectives. This will be a big challenge, so I will need to employ a good project management strategy if I want to keep on top of everything, and understand the complexity of my project.

For this I will use an app called Todoist [17], which allows you to have Projects and labels, priorities and deadlines for various tasks. In Todoist (which I have used for a long time to manage almost everything in my life), I have implemented the system from Getting Things Done by David Allen [21]. This is a productivity/organisational system which involves splitting your tasks into lists and project, as well as review and process them in an appropriate manner.

5.3 Version Control

5.3.1 Monorepo

For this project, we were provided with 1 GitLab repo, however my project consists of various different services and parts that normally would not belong in the same repo. However, there is a common technique in industry of keeping all your code base in the same repository (monorepo), hence making CI (Contant Integration) and development easy, because everything is in one place and not scattered across multiple different project repositories.

5.3.2 Bare repo and Worktrees

Git has a feature called worktrees, this feature is not very well known, however I think it is a very powerful feature and one I have used through this project. Starting with cloning a bare repo, this means that we clone a Git Repository without cloning any of the files in the folder, with the main branch checked out. However, a bare clone just clones the .git file without any of the files, and we can use worktrees to edit various branches, worktrees are a way to have multiple branches in a git repository checked out at the same time, in separate folders. This feature is very powerful because I was often working on 2 things at the same time (Frontend, backend and maybe the report), and naturally these live in different branches, therefore it is extremely useful to have them all checkout out at the same time.

5.4 Testing

Testing is an important part of the software engineering process, and there are various different types of tests which need to be conducted in order for an application to be ready to go out into the wild. It is also important to test the system as a whole with all the spinning parts available.

5.4.1 Unit Testing

Unit testing is the most basic type of testing, it consists of isolating specific functions/files and testing their functionality. I find that in a distributed system these tests are the most useless out of all tests, and therefore I will give less energy to them. This is because the most important part in my system is to test the overall behaviour of all the services and not the individual behaviour. This is not to say unit tests are not important, they are, but in my case they will be the least important.

I am using Golang's built in testing library, which consists of writing test files with the _test.go postfix, and using the golang CLI to run the tests.

For the frontend I am using Jest [6], which gives you testing utilities for testing Javascript projects, and I will also use Solid Testing Library [16], which provides utilities for testing SolidJs components specifically.

5.4.2 Integration Testing

Integration testing includes the testing of an application or multiple applications running together, but falls short of the entire system. In my project, the Hub and Brain system would need integration tests because these systems work very close together.

In my project I created a subproject dedicated to integration testing with the backend system, this includes API testing down to the route level, creating

mock data for the database and expecting a certain amount of data returned at each endpoint. Below is a snipped from my integration testing, for a route which should return assets in the system.

```
it("Should give error on non-registered account", (done: jest.DoneCallback) => {
  request(AuthUrl).post(l).send({
    email: "notregister@email.com",
    password: "password",
  }).expect(400).end((err, res) => {
    expect(err).toBeNull();
    expect(res.body).toMatchInlineSnapshot(‘
      {
        "error": "this account doesn’t exist",
      }
    ‘);
    done();
  });
});
```

Just like any good test, these tests are automatic, and describe themselves. I am making a request to the AuthUrl, sending some data and expecting certain data to come back to me.

5.4.3 End-to-End Testing

End-to-end testing is the closest to the real world. These tests are mostly (but not always), done on the frontend application, because this is where the user would interact with the system from. The purpose behind these tests is the test the overall functionality of the system, without worrying about any of the details.

Testing frameworks for this type of testing often consist of opening an actual browser window and clicking various parts of the UI to see if they work as intended, to help me do this I will use Cypress [1], this framework allows me as the developer to very quickly write end-to-end tests and provides the test runner and UI to show me how the tests are running, as well as where the tests are failing.

6 End system development

I have talked extensively about decisions I made during the design stage of the project, the research that went into these decisions and why I have layed out the architecture the way I did. I have not yet talked about what the system currently does, what features and goals I have achieved, and how I will continue

to work on the project to bring it to completion.

Each system has its own Dockerfile which is used in the docker compose file that starts all the systems and databases.

6.1 Authentication System

The easier part of my project was building an authentication system, which allowing for registering and login, as well as refreshing the users authentication. To authenticate the user, the user must provide their email and password, after which they receive a JWT [7], which proves to the other systems who the user is, and allows other systems to also prove that the user is who they say they are, because JWTs are cryptographically safe.

The user is issued 2 JWTs. A "refresh" token and an "access" token. The refresh token can only be used in the /refresh API Route, all it does it allows the user to get another "access" token, because it doesn't actually allow the user to access any of the other systems, it has a long lifespan (time for which it is valid), of 2 weeks. This also means the user can be logged in for up to 2 weeks at a time without having to login again. The access token grants the user access to the entire system, and because of this it has a very short lifespan of 5 minutes, again these can only be obtained when first registering or logging in, and subsequently using the /refresh endpoint, which issues a new access token.

The auth system is implemented in Golang, just like the rest of my backend services.

There are 3 endpoints in the authentication system:

- /register - Allows users to register a new account.
- /login - Allows users to login with their pre-existing accounts.
- /refresh - Refreshes the users JWT, using a refresh token.

This system is fully complete, and testing using integration tests which I will discuss later. There shouldn't be any need to make changes to this system in the future.

6.1.1 Dockerfile

```
FROM golang:1.19.3-alpine3.16
WORKDIR /auth
ADD ./auth .
COPY ./shared-types ../shared-types
COPY ./utils ../utils
RUN go build -o auth
EXPOSE 4546
CMD [ "/auth/auth" ]
```

6.2 Hub System

The hub system takes in all of the user communication (except for authentication requests). As it stands it has an API, which allows the users to access assets, and create transaction requests. This means that most of the features for the system are actually implemented. However, as it stands the system currently lacks any sort of caching and always makes RPC calls to the Brain service to retrieve information, this is not efficient as is something I will work on as soon as the presentation is complete. It is also written in Golang and has the following endpoints:

- /health - Checks if the system is responsive.
- /assets - Lists all the assets.
- /trade - Lists all the currently open trades.
- /trade/create - Creates an open trade (buy/sell).
- /trade/complete - Allows the user to complete an open trade from another user and buy/sell whatever the other user was buying/selling.
- /users/assets - Returns the assets owned by the user.
- /ws - Allows the user to upgrade their connection to a WS connection.

As it stands this service also doesn't provide updates using WebSockets, which means it needs to be constantly asked about pending trades and listing assets. This is not great because it creates extra load on the service, and also a slower experience for the user. The infrastructure is there to allow WebSocket connections and updates, however I have decided to hold off on fully implementing this in order to get the core features done, which I have.

6.2.1 Dockerfile

The Dockerfiles are quite similar to one another, the only thing being different is the port. This is mostly because they are all Golang application that all include the shared libraries I created to facilitate development.

```
FROM golang:1.19.3-alpine3.16
WORKDIR /hub
ADD ./hub .
COPY ./shared-types ../shared-types
COPY ./utils ../utils
RUN go build -o hub
EXPOSE 4545
CMD [ "/hub/hub" ]
```

6.3 Brain System

The brain system handles the saving and retrieval of permanently stored data relating to users transactions and assets. It receives requests from the Hub system and responds to them with the corresponding database data. It uses RPC to do this together with RabbitMQ. I will talk about this below.

To allow for better communication between the two systems I have a shared types library which is included in both system, so that they can easily exchange data and know the shape of said data. Currently this system can perform Create and Read operations on users, users assets, assets and transactions and return them to the Hub, so they can be on their way to the user.

6.3.1 Dockerfile

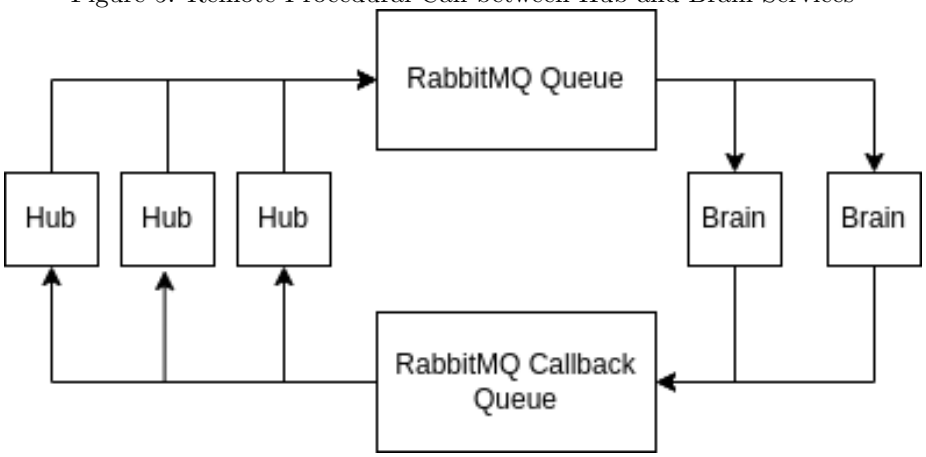
```
FROM golang:1.19.3-alpine3.16
WORKDIR /brain
ADD ./brain .
COPY ./shared-types ../shared-types
COPY ./utils ../utils
RUN go build -o brain
CMD [ "/brain/brain" ]
```

6.4 RPC Communication

RPC (Remote Procedural Call), is a mechanism to make calls to functions that do aren't on the program calling them, this is very useful in my microservices architecture because I want to retrieve data from other services in an easy and convinient way. To do this, I have made use of Golang's Go Routines and RabbitMQ as the message broker where the messages are sent to and pulled down from.

When a services requests an RPC, they publish a request to RabbitMQs RPC Queue, which has declared when the services started up. Whilst this is happening any service that should receive RPCs, waits for an event to be pushed onto the RPC Queue, this waiting occurs in a Go routine, which in this case is just another thread, this is done so checking for new messages and regular program operations can both happen concurrently. When a new event was published to the RPC queue the Go routine picks up on it and perform the required function asked by the request, after it's done it send an event to the Callback Queue with the same Id as the call received, this means in the other service that made the RPC call, we can do the same type of waiting, and check if an event with the same Id was received in the Callback Queue. A diagram that better illustrates this is below:

Figure 5: Remote Procedural Call between Hub and Brain Services



6.5 Shared Types and Utilities

As I mentioned before I have created a library which contains the types that are meant to be shared across the entire system, these create a consistent standard of communication across services, which can often be difficult to assert. This way, we have 1 source of truth for what the data should look like, all included in a package that all services have access to.

I have also created a utility package which contains functions that are used throughout the application, these include JWT verification, and middlewares used in multiple services.

6.6 Docker Compose

The docker compose file contains all the information for the entire system, it also includes the databases for each of the systems and the RabbitMQ service that is started alongside them. It also specifies information such as network mode, restarting behaviors, contexts etc...

```
version: "3"

services:
  rabbitmq:
    build: ./rabbitmq
    ports:
      - 15672:15672 #Management console
      - 5672:5672 #Actual exchange
    network_mode: host
    restart: on-failure

  # Hub service is the entry point for all thing (except authentication)
  hub:
    build:
      context: .
      dockerfile: ./Hub.Dockerfile
    ports:
      - 4545:4545
    depends_on:
      - rabbitmq
    restart: on-failure
    network_mode: host

  # Brain service consumes RabbitMQ request and deals with permanent storage
  brain:
    build:
      context: .
      dockerfile: ./Brain.Dockerfile
    ports:
      - 5672:5672
    depends_on:
      - brain_postgres
      - rabbitmq
    restart: on-failure
    network_mode: host

  brain_postgres:
    build: ./brain_postgres
    volumes:
      - ./brain_postgres/scripts/init.sql:/docker-entrypoint-initdb.d/init.sql
```

```
    - ./brain_postgres/db:/var/lib/postgresql/data
  environment:
    - POSTGRES_NAME=postgres
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=postgres
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U $$POSTGRES_USER"]
  ports:
    - 5443:5432

auth:
  build:
    context: .
    dockerfile: ./Auth.Dockerfile
  ports:
    - 4546:4546
  depends_on:
    - brain_postgres
    - auth_postgres
  restart: on-failure
  network_mode: host

auth_postgres:
  build: ./auth_postgres
  volumes:
    - ./auth_postgres/scripts/init.sql:/docker-entrypoint-initdb.d/init.sql
    - ./auth_postgres/db:/var/lib/postgresql/data
  environment:
    - POSTGRES_NAME=postgres
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=postgres
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U $$POSTGRES_USER"]
  ports:
    - 5442:5432
```

As you can see the docker compose file is quite extensively because it has to contain the various services used for my project, it runs the RabbitMq service, various databases and all the services which I have coded. It also configures ports and environment variables as well as docker volumes.

7 Bibliography

References

[1] Cypress: Javascript end-to-end testing framework. https://www.cypress.io/, Nov 2022.

[2] Design microservice architectures the right way. https://www.youtube.com/watch?v=j6ow-UemzBc, Nov 2022.

[3] Docker swarm. https://docs.docker.com/engine/swarm/, Nov 2022.

[4] Esbuild: An extremely fast javascript bundler. https://esbuild.github.io/, Nov 2022.

[5] Hot swap definition. https://en.wikipedia.org/wiki/Hot_swapping, Nov 2022.

[6] Jest: Javascript testing framework. https://jestjs.io/, Nov 2022.

[7] Json web tokens. https://jwt.io, Nov 2022.

[8] Kubernetes: Production-grade container orchestration. https://kubernetes.io/, Nov 2022.

[9] Kvm: Kernel virtual machine. https://www.linux-kvm.org/page/Main_Page, Nov 2022.

[10] Latency definition. https://en.wikipedia.org/wiki/Latency_(engineering), Nov 2022.

[11] Mongodb: A json document database. https://www.mongodb.com/, Nov 2022.

[12] Rabbitmq: A messaging system for distributed applications. https://www.rabbitmq.com/, Sep 2022.

[13] React: A javascript library for building user interfaces. https://reactjs.org/, Nov 2022.

[14] Redis: Open-source, in-memory database. https://redis.io/, Nov 2022.

[15] Solid: Simple and performant reactivity for building user interfaces. https://www.solidjs.com/, Nov 2022.

[16] Solidjs testing library. https://github.com/solidjs/solid-testing-library, Nov 2022.

[17] Todoist: Organise your work and life. https://todoist.com/, Nov 2022.

[18] Typescript: Javascript with syntax for types. https://www.typescriptlang.org/, Nov 2022.

[19] Vite: Next generation frontend tooling. `https://vitejs.dev/`, Nov 2022.

[20] Webpack: Bundle your code. `https://webpack.js.org`, Nov 2022.

[21] David Allen. *Getting things done: The art of stress-free productivity. A system of productivity I can use within this project.* Piatkus Books, London, England, 2002.

[22] Svetlana Andjelic, Slobodan Obradovic, and Branislav Gacesa. A performance analysis of the dbms - mysql vs postgresql. *Communications - Scientific letters of the University of Zilina*, 10:53–57, 12 2008.

[23] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* O'Reilly, 2021.

[24] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* O'Reilly, 2021.

[25] Antonios Makris, Konstantinos Tserpes, Giannis Spiliopoulos, and Dimosthenis Anagnostopoulos. Performance evaluation of mongodb and postgresql for spatio-temporal data. In *EDBT/ICDT Workshops*, 2019.

[26] Sam Newman. *1. Microservices.* O'Reilly, 2016.

[27] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

8 Logbook

29/09/2022
First entry in the log book. This week I have taken the time to get organised, finalise my project idea and talk to my supervisor about it. I finalised the list of features that I wanted to have in the final product. Also got a draft of my plan.

02/10/2022
Finish the draft of the plan, but does not yet include the bibliography. Still need to walk through the marking grid and tick all the boxes.

03/10/2022
Completed the bibliography for now. However the work is not complete.

05/10/2022
Added testing, how currency will work and core and optional features to the plan.

07/10/2022
Finished the plan, submitting it today.

11/10/2022
Started implementing the "hub" service, which is the first point of contact from the user.

12/10/2022

Beginning template for the frontend application with all the dependencies installed.

24/10/2022

- Setup docker file for the hub micro service and a docker compose file to allow for easy deployment and development. - Created a diagram for the overall system architecture.

25/10/2022

Connected frontend to the backend using websocket connection. I am still a week behind on my plan so I will have to work very hard this week to get back on schedule.

31/10/2022

Added RabbitMQ docker image, and created a test connection between hub and rabbitmq, it works.

05/11/2022

Setup the SQL schemas for the required databases.

06/11/2022

Started work on the authentication system.

07/11/2022

Finished register method in auth system, with some very nice code. I also finished the login method. Also finished the refresh method for JWTs. Basically did the entire Auth system.

08/11/2022

Connected frontend to backend login and register methods, learn about state management in SolidJs.

23/11/2022

I have a shared types library now and RPC communication working between hub and brain and sending back to the client. Need to implement the rest of the routes.

26/11/2022

There was a bug in the backend code, where some memory was declared in a different thread, and therefore was never updating, fixed it by redeclaring this memory.

27/11/2022

Fixed a lot of stuff on the backend and did some refactoring so some dependencies are shared. Also started the transactions endpoint but decided that I would like to connect the assets to the frontend first.

29/11/2022

- Model methods to get user assets, create and complete transaction. Various bug fixes and connected the frontend to view assets. Some things are very prototypy, so will need refactoring - Endpoints for the trading of assets are complete, now I need further testing.

29/11/2022

- All the backend is fully dockerised and running in containers!

22/11/2022

Started interim report, completed a first draft of the structure and included a

lot of my research.

23/11/2022

Adding sections about version control monorepos and git worktrees.

27/11/2022

Section about project management

30/11/2022

Finished the first draft of the report!

04/12/2022

Finished integration testing app, it tests the authentication system and most of the routes on the hub.

05/12/2022

Final draft of the interim report completed

02/12/2022

Integration testing setup.