# Technical Specification

FDM05

# Contents

# 1. Architecture Overview

## 1.1. Introduction

This document contains two technical design specifications which approach the project with different views on the chosen design goals. However, both approaches will follow the same high-level design and architecture, due to preferences outlined in the project requirements.

The focus design goals for this project are:

- Reliability
- Reusability
- Maintainability

## 1.2. High-Level Design

Figure 1 shows a use case diagram highlighting the project requirements, which will be a basis for both proposed technical solutions. Most notably the functionality is separated into two areas:

- The 'Stock System' is responsible for general interactions within the web application, such as browsing, buying, and selling shares.
- The 'Email System' holds responsibility for all actions related to sending user and admin communications through email.


Figure 1

A requirement for this project is the use of a multi-client client-server architecture (see figure 2). Both technical design approaches will follow this pattern to ensure a fast, concurrent user experience.

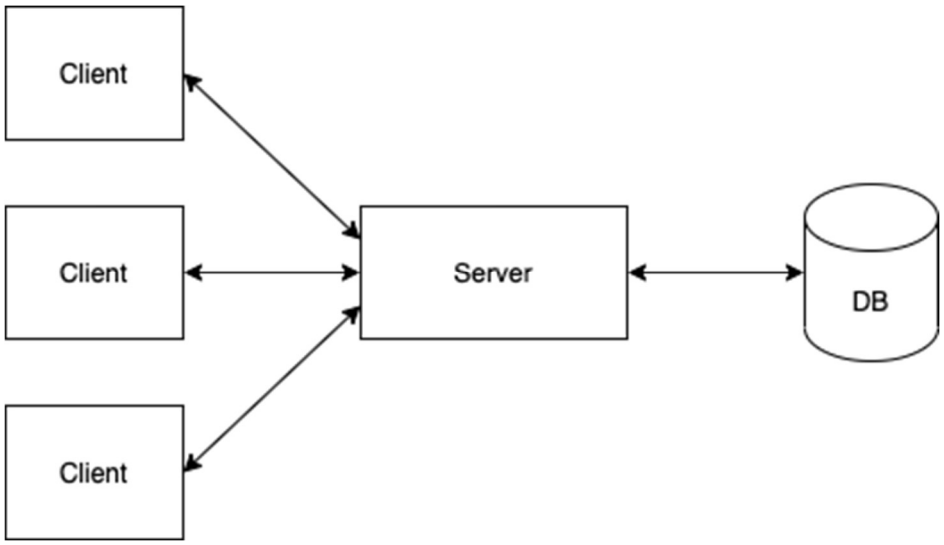


Figure 2

## 1.3. Client Architecture

The client-server approach allows the client to be completely decoupled from the back end / server code, therefore both technical solutions will focus on the differences in server and database design and follow the same design for their client. This approach also enables complete segregation of business logic from the front-end, which creates an additional layer of security for potential threats to bypass. In addition, this architecture improves maintainability for the client as the codebase would be significantly reduced through loose coupling.

# 2. Technical Design A

## 2.1. Data Model

The proposed data model for this design uses relational database technology. This approach utilizes more mature technology and tooling. The maturity of a chosen technology generally impacts the reliability and sometimes the security of the resulting software, as there is generally more time spent addressing vulnerabilities, and a greater choice in version selection to avoid potential known vulnerabilities.

Relational databases also allow for ACID (atomicity, consistency, isolation, durability) guarantees, generally recognized as *transactions*. This assures changes in data across multiple tables always remain in sync and there is no additional management required to assure this from the codebase, thus removing several maintainability headaches it may cause.

Figure 3 shows the ERD (Entity Relationship Diagram) for the proposed solution.

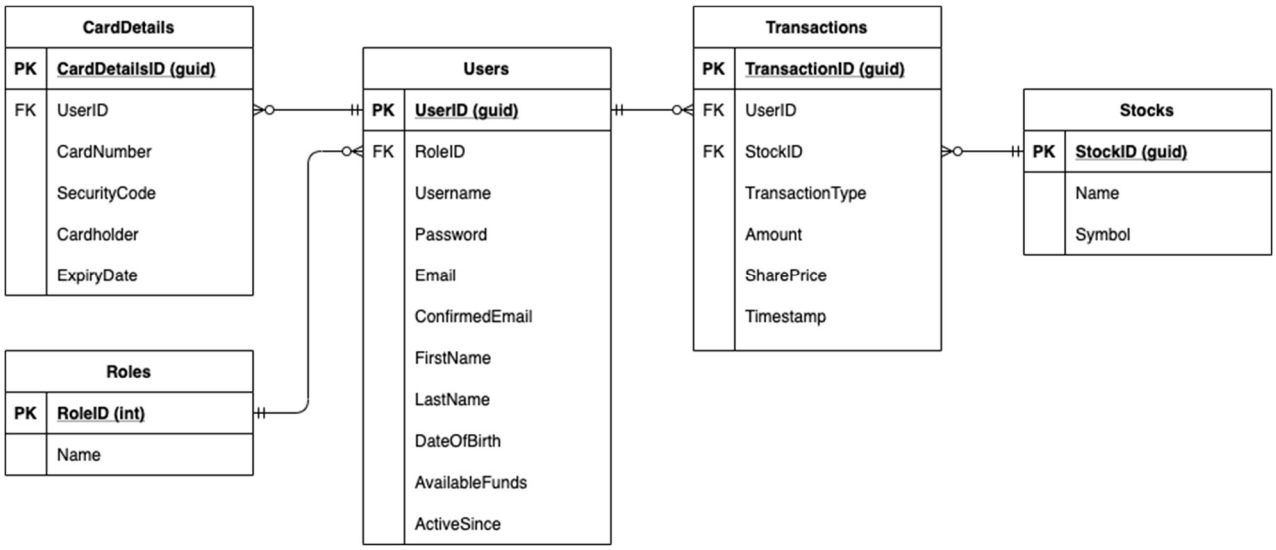


Figure 3

- The *Users* table is essential to the system for keeping track of users and their information.
- The *Stocks* table is important for storing basic details about a company. The table does not include any pricing information, as this will be generated by a separate back-end component of the application.
- The *Transactions* table is central to the buying and selling of shares, and storing a user's history and the detail of each transaction is important for the reporting functionality of the application.
- The *CardDetails* table contains a subset of user information specifically in a separate table, to decouple the basic details from more advanced information.
- The *Roles* table will contain the different roles a user can have in the application. The major difference in this table to others is the ID field, as there will only ever be a finite number of entries in the table: user and super user.

## 2.2. Server Architecture

Figure 4 provides an expanded overview of the proposed client-server architecture. The diagram outlines the several components required to make up the server functionality, and how they interact with one another.

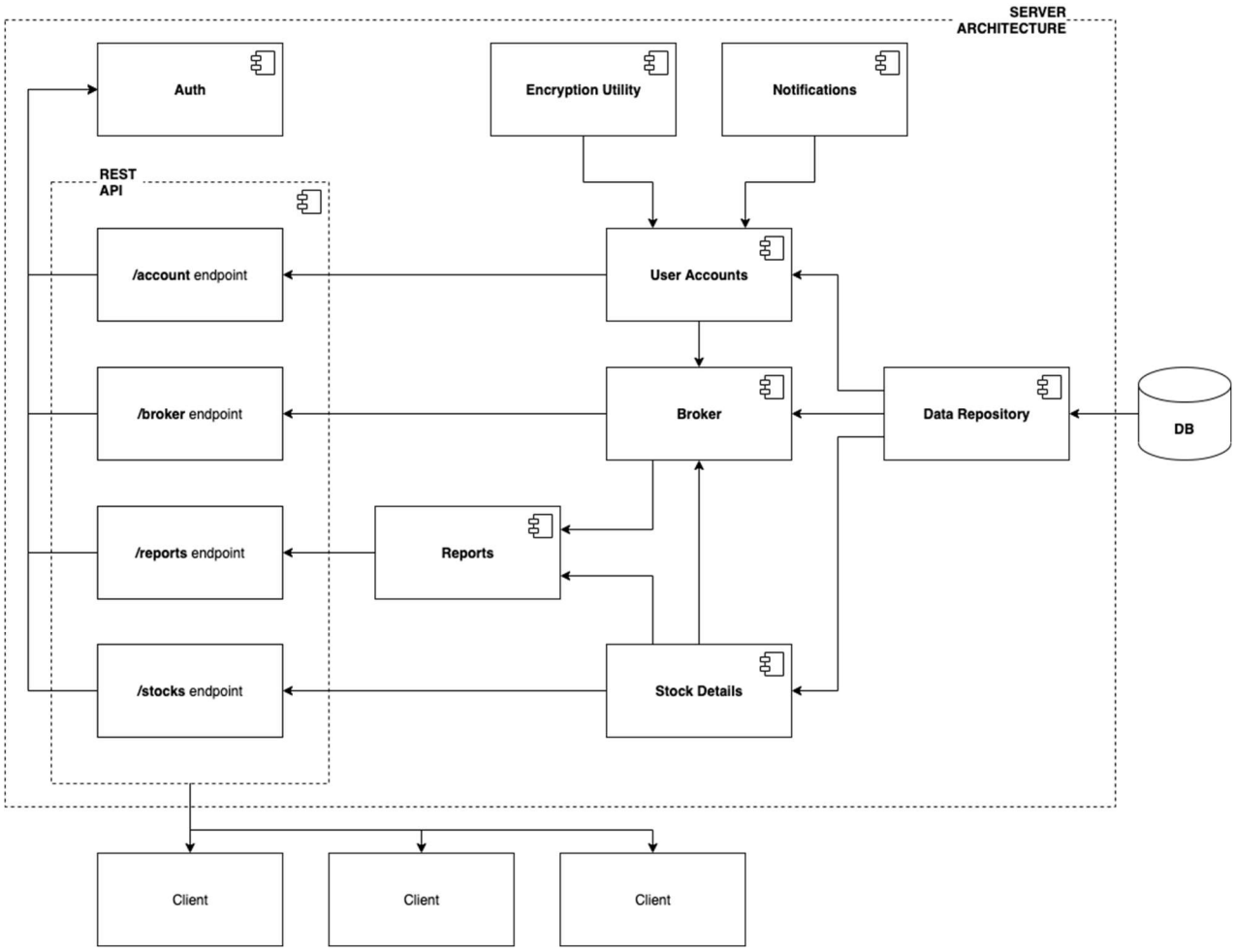


Figure 4

The segregation of components in such a way helps to encourage high cohesion within components and understanding the connections between one another provides a foundation for good loose coupling. Highly cohesive components with a focused purpose helps to increase the maintainability of software by being clear in their design and functionality.

### 2.2.1. Data Repository

Figure 5 shows a class diagram outlining the data access component, which controls all database interaction functions. This component employs the repository pattern. Each other component that requires access to the database will invoke the required repository.

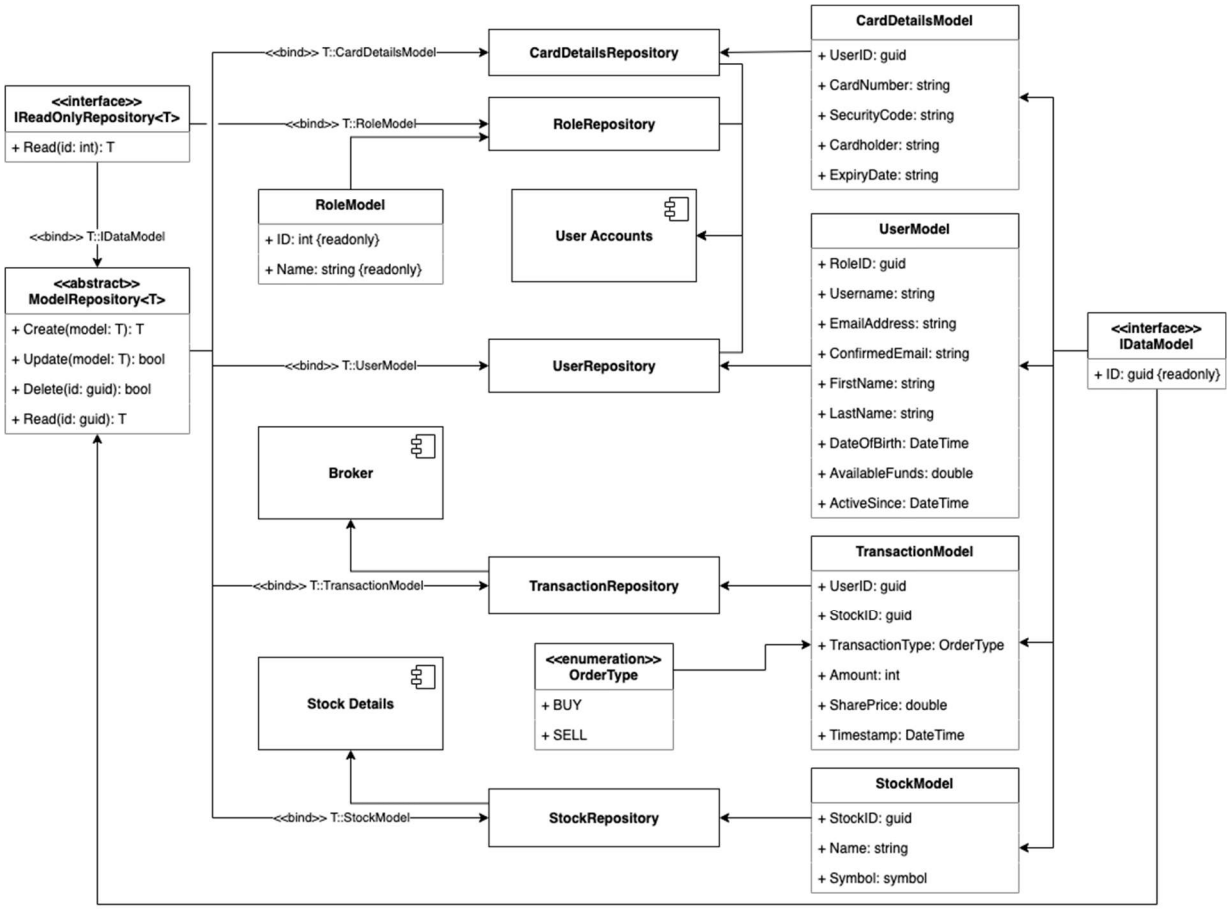


Figure 5

The diagram shows multiple children of the repository class, one per data model, where each data model is a direct equivalent of the tables shown in the ERD (figure 1). The data models exist to map data from the database into read / write objects within the codebase, and vice versa. Notably, *RoleModel* does not implement the *IDataModel* interface as the identifier is of a different type, and the corresponding repository is read only as no changes should be made to the *Roles* table once initialised.

The design of this component encourages reusability through the usage of abstract classes and interfaces.

### 2.2.2. User Accounts

Figure 6 outlines the functionality for the user account component within the application alongside their associated API endpoints. This component contains CRUD functionality for user accounts and credit card dependencies. The supporting code is quite large to account for all data models in the relational database.

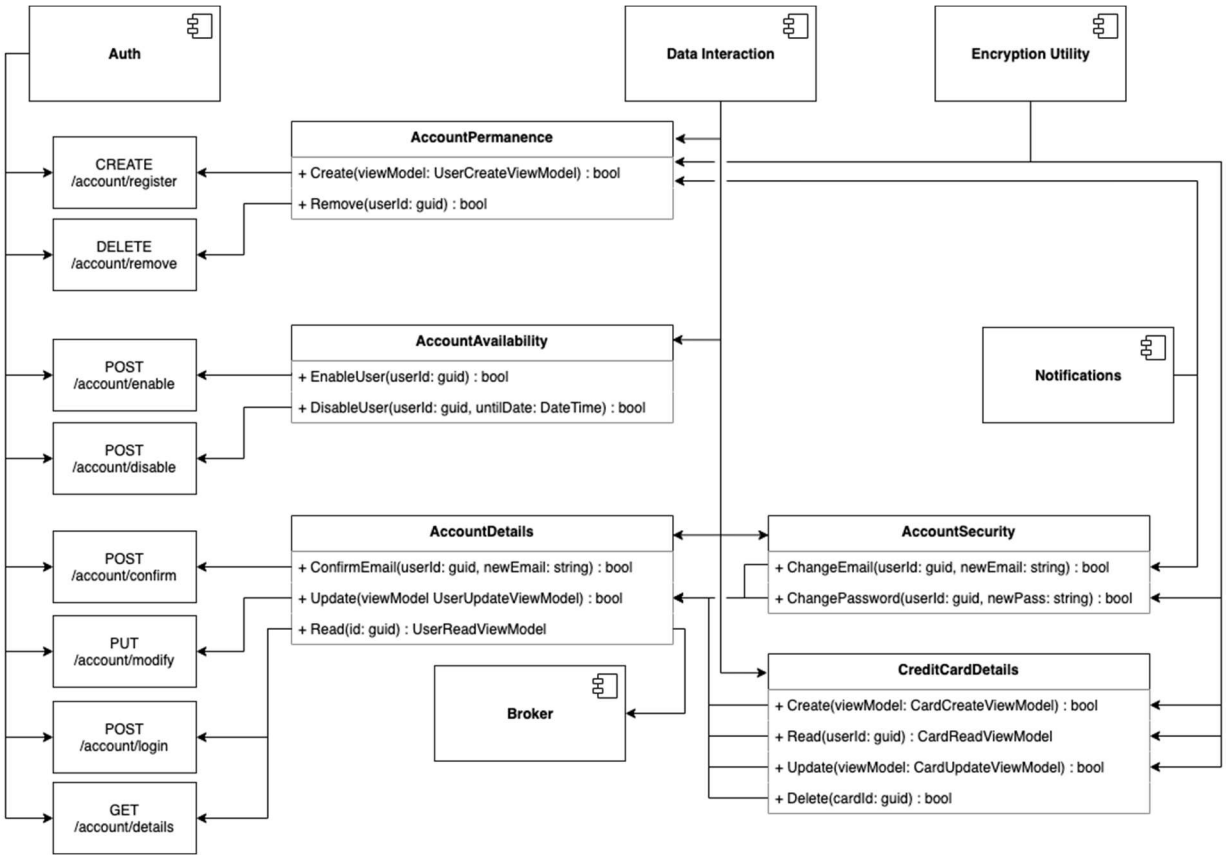


Figure 6

This component demonstrates loosely coupled code, encouraging easier maintainability through separation of concerns between classes. Many small classes also make unit testing simple, which improves reliability of a codebase.

### 2.2.3. Broker

Figure 7 outlines the classes required for the broker component of the application, which validates and performs transaction requests from a client.

The broker component aims to be simple and cohesive through the implementation of limited functionality. This allows the code to be more readable as there is much less abstraction to consider, and there is no requirement to overcomplicate the code.

### 2.2.4. Stock Details

Figure 8 shows an overview of the stock component, which contains CRUD functionality for company details, and obtains corresponding pricing information.

Similar to the broker component, the stock details component aims for simplicity, high cohesion, and maintainability. Stock pricing data may also be retrieved in given frequencies (every 5 mins, 1 hour, etc.), which enables the codebase to be easily extendible for future potential changes in functionality.

### 2.2.5. Reports

Figure 9 shows an overview of the reporting component, which produces reports for stock details or a user's previous transactions through interfacing with the stock details and broker components.

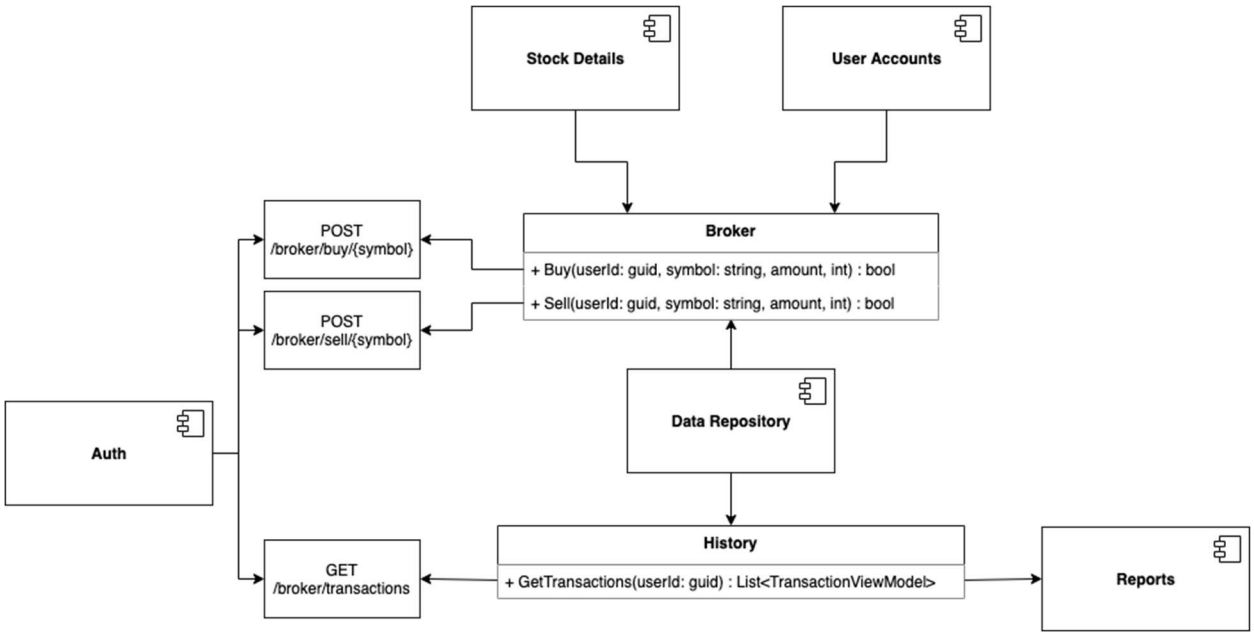


Figure 7

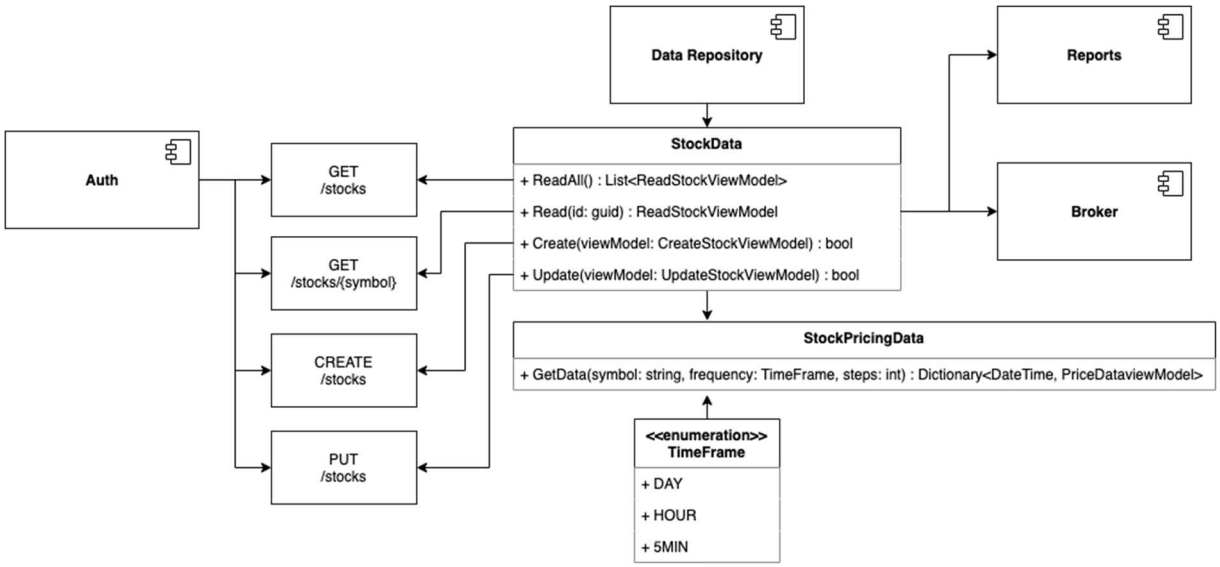


Figure 8

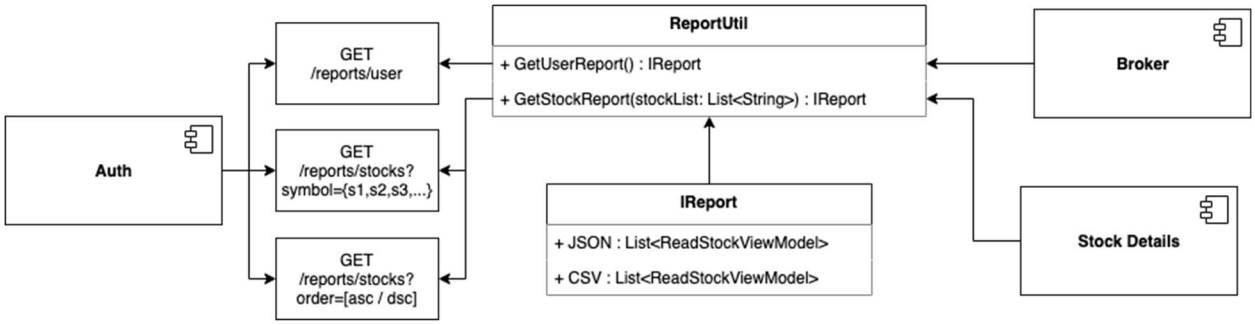


Figure 9

### 2.2.6. REST API

Figure 10 shows the REST API design for the application, which contains controllers and endpoints to expose backend functionality to the client. Each endpoint is associated with a user role to ensure correct authorization for each piece of functionality.

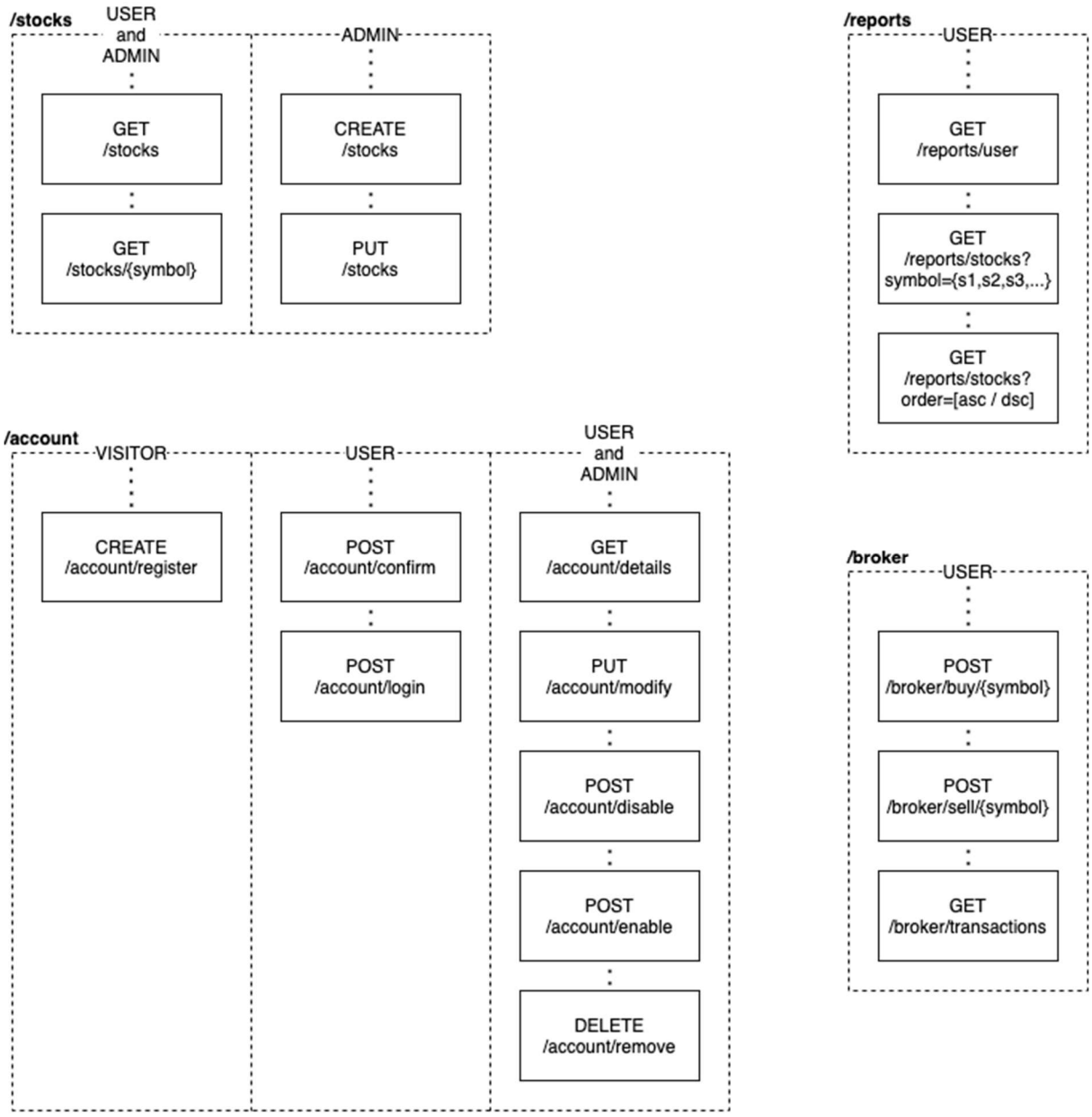


Figure 10

The REST API component is responsible for separating business logic from routing logic, this way the controllers can remain lightweight to improve future maintainability. This component shares similarities with the façade design pattern in that it is a coherent interface for the client to call more complex methods from the backend system. This pattern hides the additional complexities of making an API call, such as the inclusion of authorization / authentication middleware for each call.

### 2.2.7. Auth

Figure 11 shows details of the auth middleware, which authenticates and authorizes user requests to the REST API.

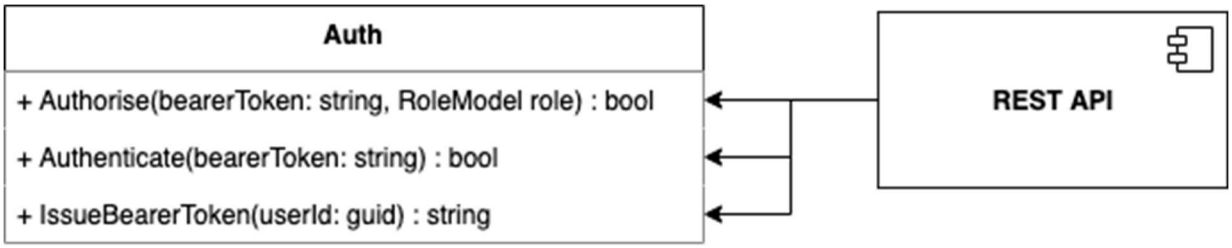


Figure 11

The auth component is a basic interface for creating and validating bearer tokens for API requests, as this process should not be overcomplicated.

### 2.2.8. Encryption

Figure 12 shows the classes and interfaces within the encryption component, which performs one-way and two-way encryption for various purposes.

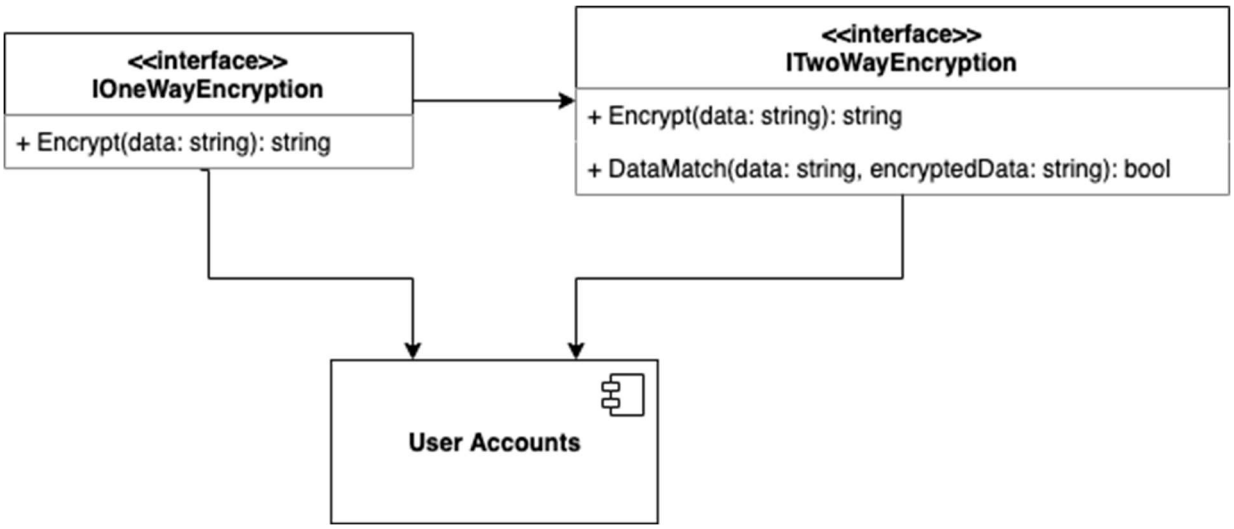


Figure 12

There are two operations needed for this component: one-way and two-way encryption. The 2 corresponding interfaces exist to allow the ease of extension, as more implementations could be added in the future.

### 2.2.9. Notifications

Figure 13 gives an overview of the user notification component, which manages communications to users, primarily through email.

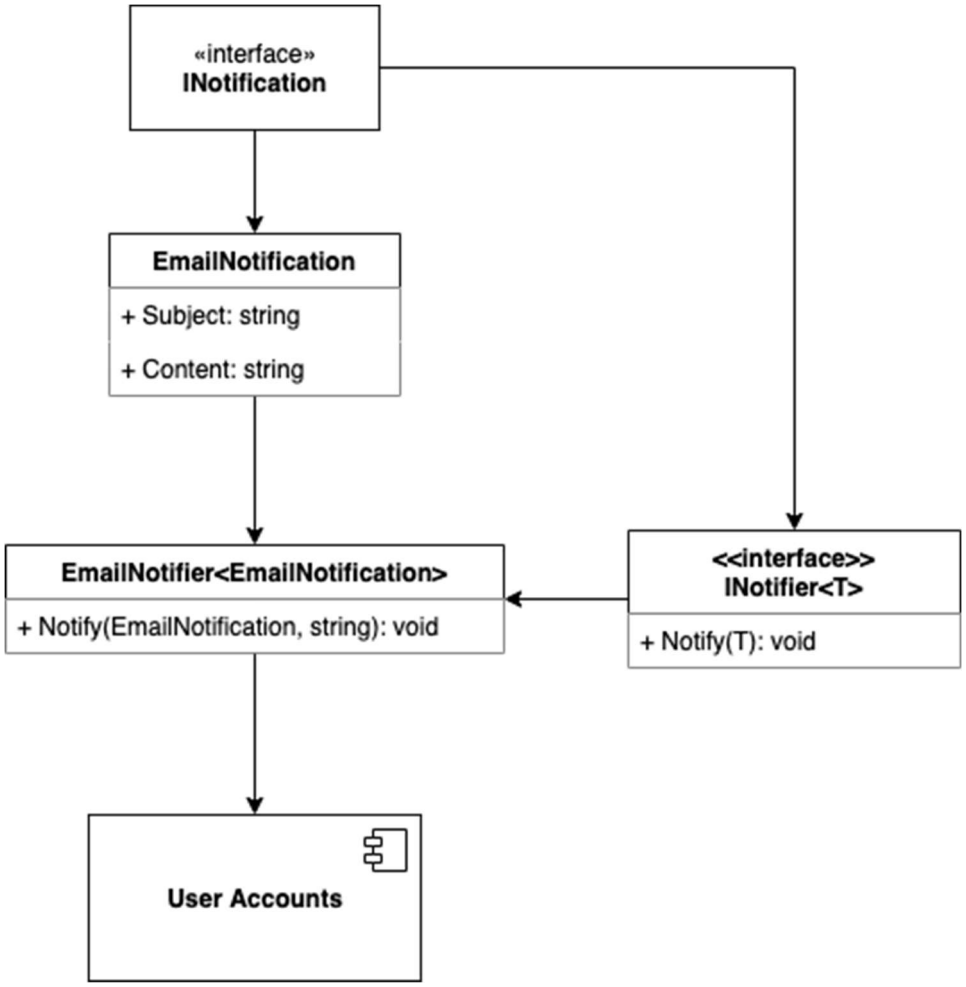


Figure 13

The notification component is built in such a way that more notification and notifier types may be added in the future. For example, the architecture would theoretically support SMS notifications with a very minor change. This highly cohesive module would increase maintainability of the solution, and the modularity of such increases the testability and thus reliability of the application.

## 2.3. Discussion

This design outlines a very traditional approach to OOP through careful consideration of SOLID principles. The design and technology choices are tried and tested, providing a reliable foundation to the development of this solution.

However, this approach is not ‘on trend’ with the industry at the time of writing, as monolithic architecture is inherently tightly coupled and causes problems for maintainability of an application due to lines being blurred between components. It can also be difficult to foresee the nuances of a software solution until the development process has started, and a tightly coupled design such as this does not allow for much freedom for change where necessary.

In addition, blindly following SOLID principles is now regarded by some as bad practice due to the unnecessary complexity it can introduce to a codebase.

# 3. Technical Design B

## 3.1. Data Model

The data model for this design will be approached with a horizontally scalable mindset, using NoSQL technology. This scalability will promote maintainability and ease of extension within the model, due to a non-rigid schema that supports iterative, agile development.

This model is inspired by the following design and its use of microservices as a back-end system, where each service connects to its own singular database. While the data model is non-relational and the schema shown is not enforced, figure 14 shows the model in an ERD format to outline the high-level data relationships between each data document within the databases.

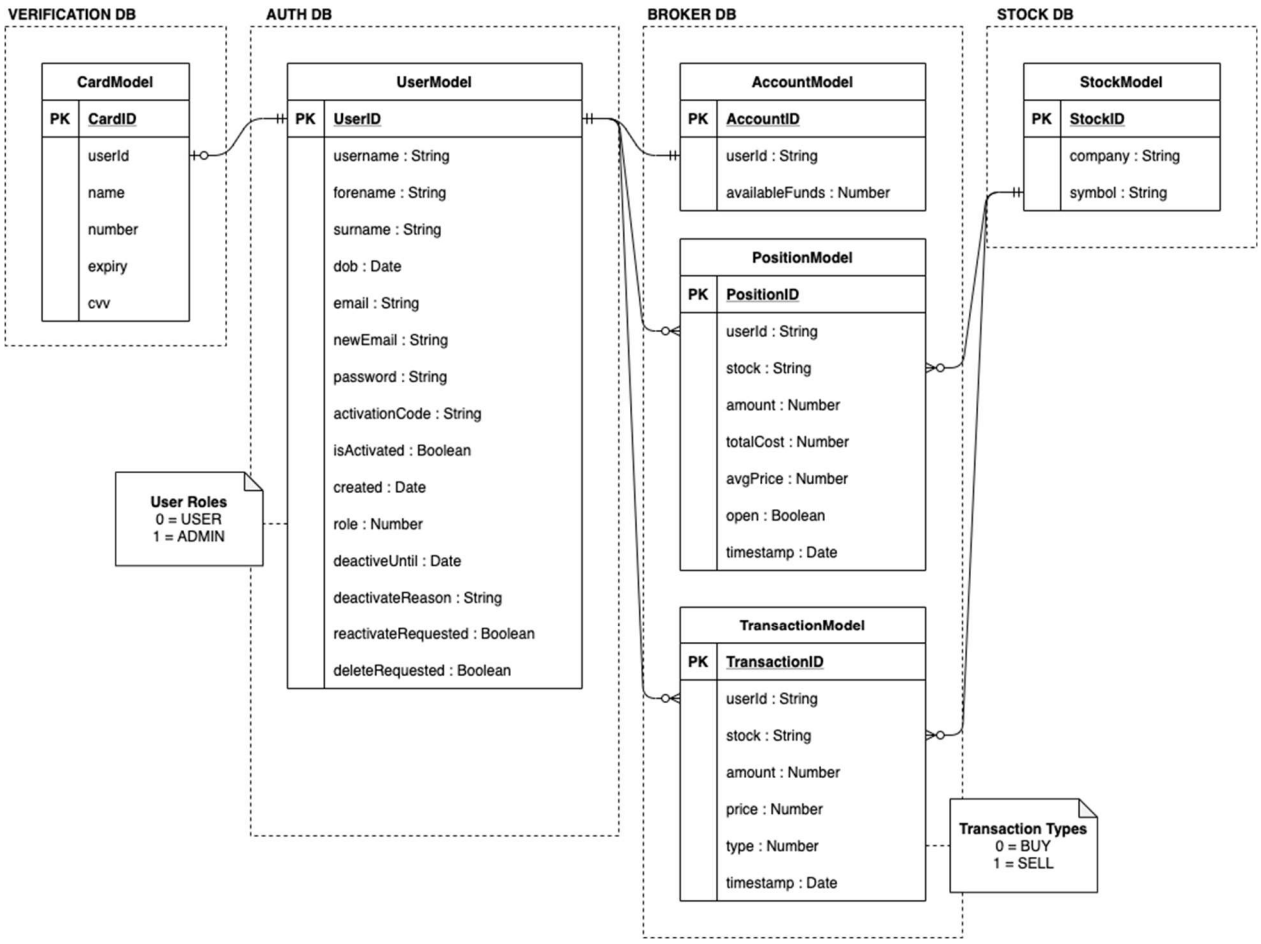


Figure 14

- *UserModel* supports all user authentication and personal details.
- *AccountModel* exists to link available funds to a user ID.
- *PositionModel* holds information to link a user with currently held shares of a particular stock. A position will remain open as long as the currently held shares remains above 0 and will be modified every time a transaction is made.
- *TransactionModel* holds read-only information for a specific transaction made by a user.
- *StockModel* holds details for a company listed on the website.
- *CardModel* exists to manage credit card information for users. Credit card details are required to verify a user, allowing them to perform transactions.

## 3.2. Server Architecture

The back-end architecture of the system in this design is separated into microservices, which coincides with the NoSQL data model as they are also horizontally scalable. This distributed services model can be more reliable, due to the absence of any single point of failure – functionality is distributed between different services which can exist as multiple instances to reduce downtime and perform load balancing. Figure 15 shows the service layout and how they connect to one another, as well as the web client.

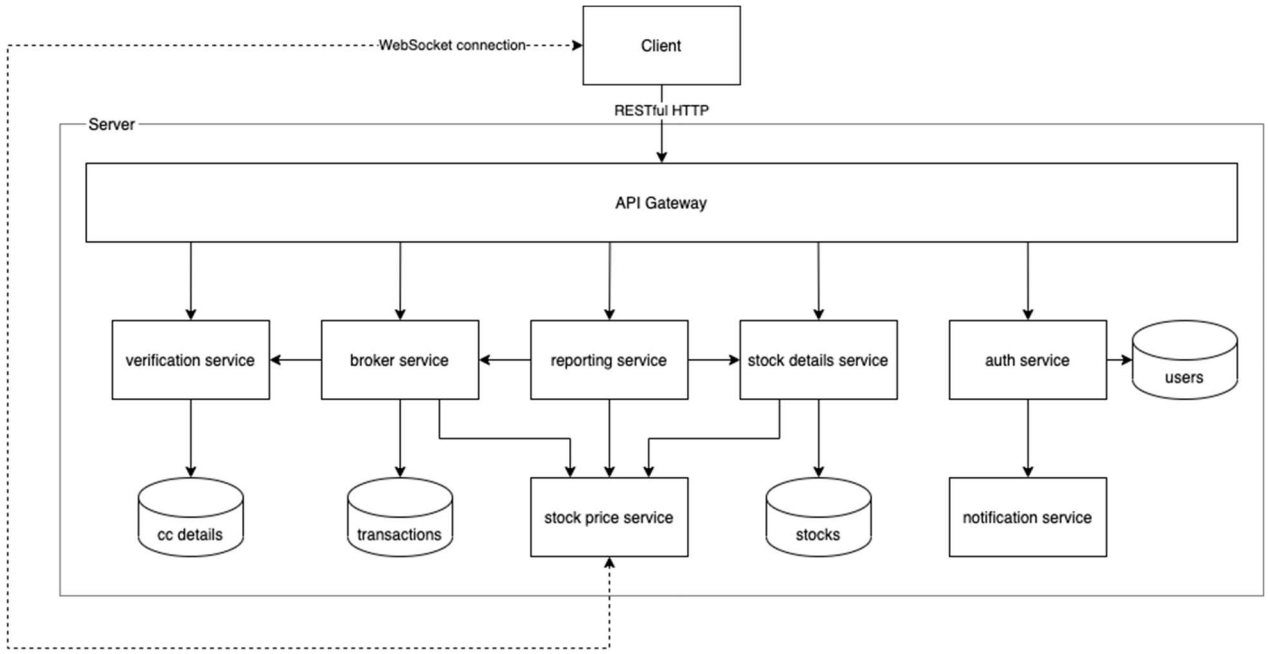


Figure 15

### 3.2.1. API Gateway

The API gateway is a singular interface for communication between clients and the back end microservices. This approach is similar to the façade design pattern in OOP but implemented at an architectural level.

Exposure of only the singular gateway to a client introduces a layer of obscurity and thus additional security to the back-end services. The client must also reference only a single location to the gateway and its endpoints, increasing maintainability by reducing the number of connections that need to be managed in client-side development. In addition to this, the gateway also allows for a singular implementation of authentication middleware into other service operations, as shown in figure 16.

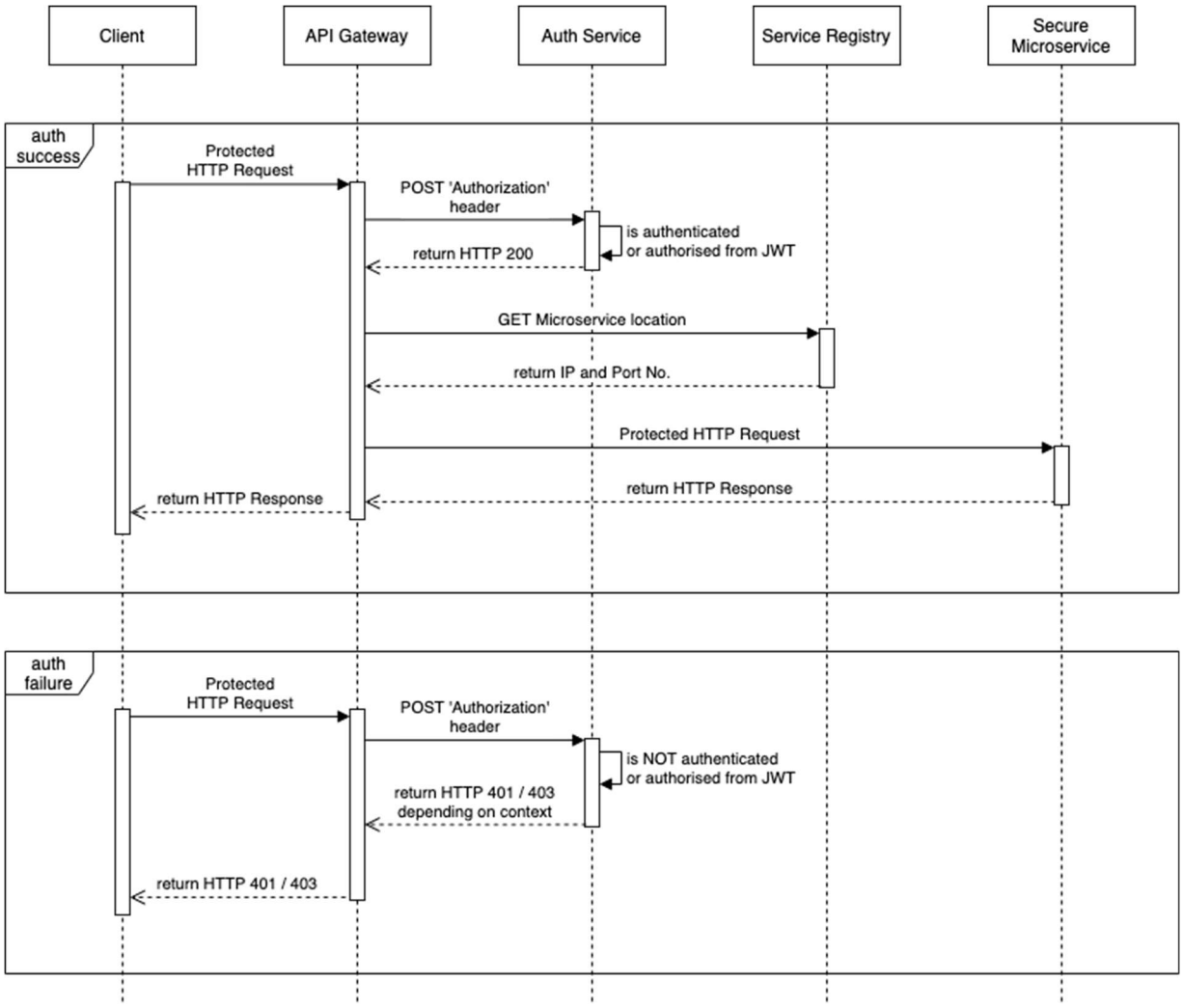


Figure 16

The sequence diagram visualizes the obscuration of a back-end system to the client, and how the segregation of such operations can simplify the client and improve its maintainability – some functionality is not yet explained but will be addressed shortly. The API gateway pattern collates the complexity of microservices into an accessible and easy to use interface; especially so if the back-end system uses a culmination of communication technologies, such as REST, GraphQL, or message queues.

### 3.2.2. Service Registry

A core benefit of microservice architecture is the ability to spin up additional instances of a particular service when demand is high, to balance the load of requests. If not managed correctly, this creates an issue whereby services must know the potential location of other services before they theoretically exist. The service registry pattern 17 is a solution to managing this problem effectively.

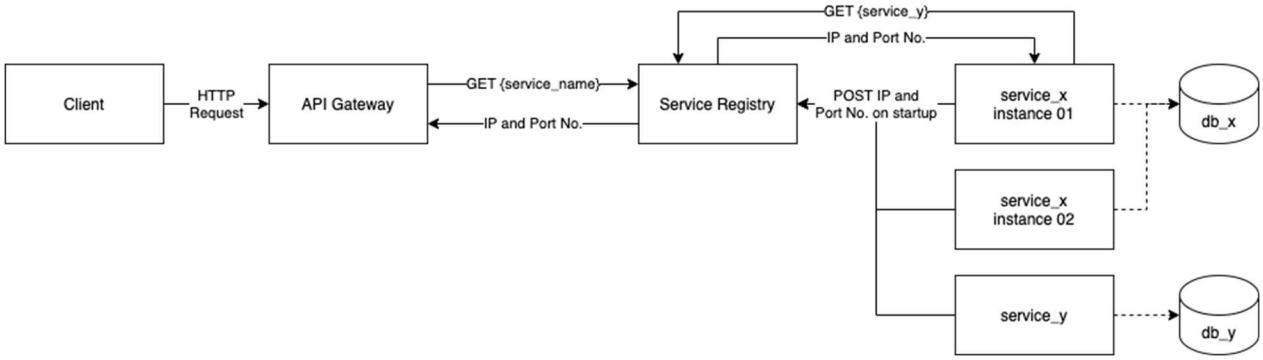


Figure 17

On the surface level, each time a service is started it should inform the registry of its IP address and port number. When queried for a service location, the registry should return the corresponding information it has stored. But what happens when a service goes down? Each service has a responsibility to send a 'heartbeat' to the registry, informing it that the service is still up. If no heartbeat is received within a given time frame, the registry can safely remove the stored location details. Figure 18 shows a sequence diagram of this process. This pattern enforces a reliable architecture for service calls.

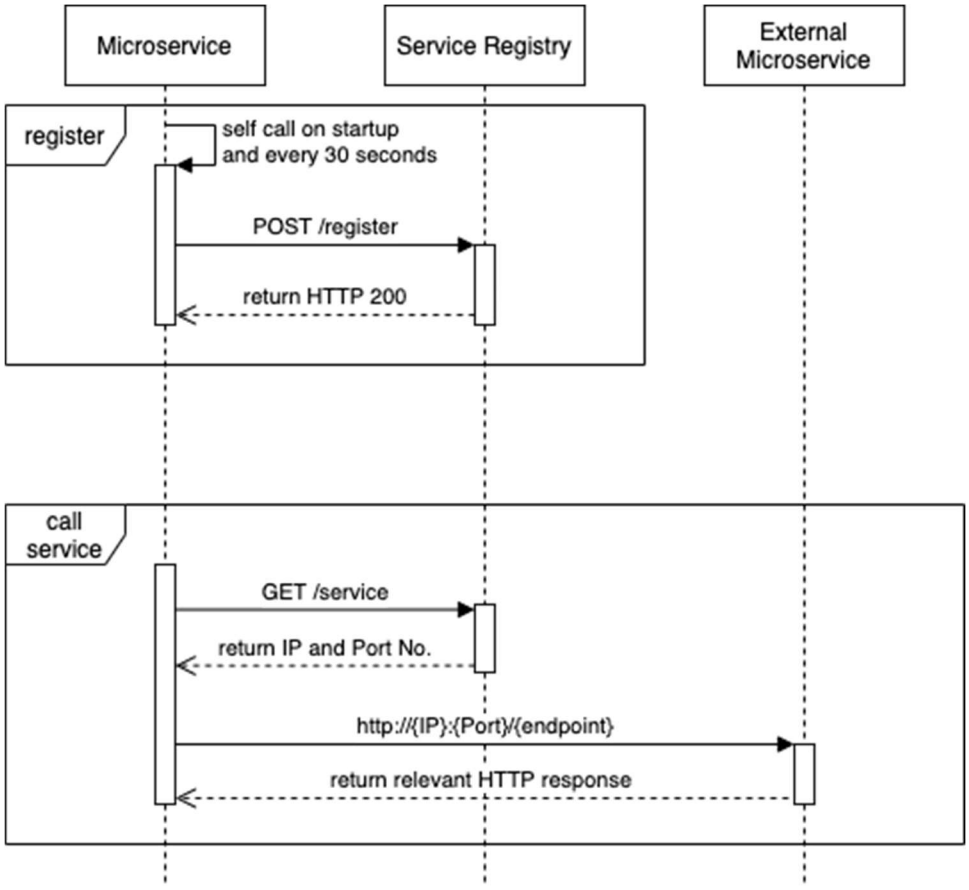


Figure 18

There are many underlying benefits to this pattern in addition to service discovery, which creates greater maintainability and readability through abstraction. The service registry has control over which instance of a service gets called, meaning there's an additional opportunity for load balancing and/or performing A/B testing on newer versions of a service.

However, the inclusion of a service registry means each request to a microservice must be preceded with a call to the registry, effectively doubling bandwidth requirements. This could be mitigated through caching the response data locally on each service for a period dictated by the registry.

### 3.3.3. Microservice Architecture

An issue in writing microservices is the boilerplate code required to build a single service. The more shared architecture that exists for a microservice approach creates additional duplicate code, and any changes down the line makes maintainability a major concern as changes must be made in each individual service. A library of common utilities and processes should be built to mitigate this.

A typical microservice in this project should consist of the following shared configuration:

- Environment variable management – will enable out of the box ability to read '.env' files for environment specific configuration (e.g., DB connection string).
- Startup processes – enables a standard process for spinning up a new service, ensuring each independent service runs the same as the next.
- A uniform database connection process – ensures all potential database connections run from the same interface, allowing easy switching to a different implementation if so desired.
- A uniform test configuration – couples with startup processes and ensures all tests run on an identical setup between services.
- Service calling support – creates a single interface for calling external services between one another, ensuring maintainability and potential for shared middleware such as contacting the service registry.
- A common HTTP package – builds a standard for HTTP responses sent by each service, so the client can expect identical formats for every request through the gateway.

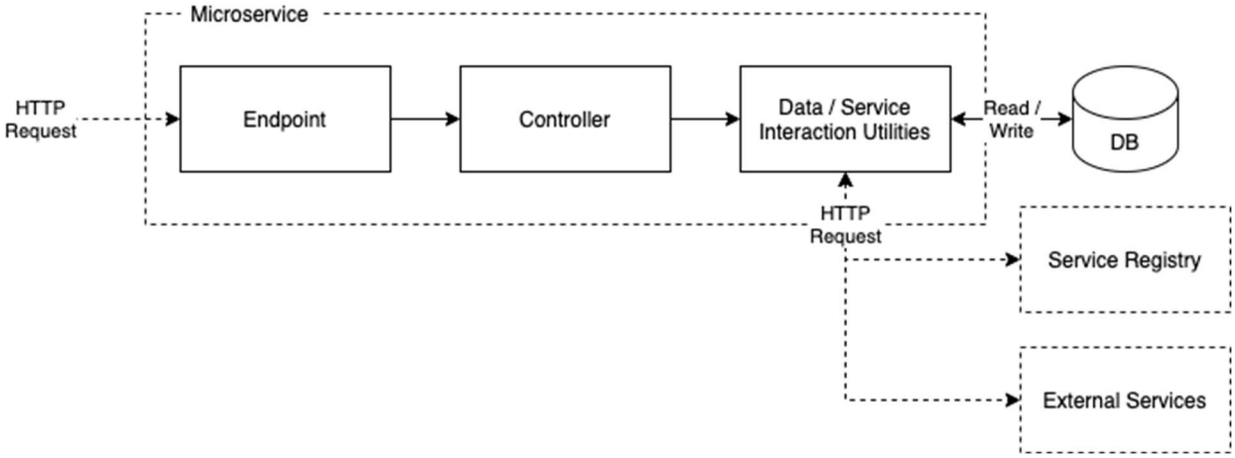


Figure 19

In addition to shared configuration, each service will follow a consistent format, as displayed in Figure 19. The major components are:

- Routes / Endpoints – entry points for HTTP requests which uses request details (body, headers, query string, etc.) to call a controller method.
- Controllers – classes containing a collection of methods which perform business logic.
- Data / Service Interaction – an optional collection of classes responsible for interacting with the service's database, or handling requests and responses to external microservices and the service registry.
- Database – each microservice should have only a single database connection at a maximum.

This utility module reduces the cost of introducing a new microservice into the solution, as it removes boilerplate code for common functionality. While being reusable, this practice is also reliable due to the ability to extensively test the shared codebase for a reduced cost.

### 3.3.4. Auth Service

The auth service is responsible for management of user details (*UserModel*) through CRUD operations into an associated database. Such operations will also include admin operations to manage permissions and activity of user accounts. As this service has access to user details, it will also provide authentication and authorisation functionality through the issuing and verification of JSON Web Tokens (JWTs).

### 3.3.5. Stock service

The stock service will manage the CRUD operations for company details *(StockModel*) within its associated database. In addition to this, it will serve as a gateway to an 'external' pricing microservice, which cannot be accessed directly from the API gateway, to allow for simple replacement of the pricing source to a third-party provider(s) in future.

The pricing service will generate and cache historic price data and provide real-time updates every 5 minutes. These updates will be sent to registered clients using WebSocket connections. This will prevent the download of full historical data each time the client requests a price and shifts responsibility for distribution of changes in data to the pricing service itself. This service could be repurposed into middleware that converts third-party data into the desired format in a production environment.

### 3.3.6. Broker service

The broker service will track account funds (*AccountModel*) and held positions *(PositonModel*), as well as authorise user transaction requests *(TransactionModel*). This service is crucial for ensuring the integrity of user interaction with the web application, through local verification of funds and integration with the verification service to ensure valid credit card details are stored.

### 3.3.7. Verification service

The verification service simply handles CRUD operations of credit card details (*CardModel*) on the associated database. The service will expose a single endpoint to verify a user through a JWT, returning successfully if valid card details exist.

### 3.3.8. Reporting service

The reporting service does not perform any database interaction and relies mostly on external services to collate information for a client. The service provides reports on current stock information or the currently held shares of a given user.

### 3.3.9. Notification service

The notification service exists as an interface to initiate communications from another microservice. Its functionality is not exposed to the client, and functions as a utility for other services to use. Its main responsibility is sending emails to customers and admins; however, it holds much more potential for other forms of communication. It may even be expanded to provide support for Multi-Factor Authentication through SMS.

## 3.3. Discussion

This design places a focus on the high-level architecture of the back-end software solution. This approach allows for microservices to be developed cheaply and effectively using reusable code. This means services can be plentiful, cohesive, and loosely coupled, meaning the design of low-level intricacies is slightly wasteful due to each service corresponding to a class in traditional OOP approaches.

The distributed microservice approach also increases stability and reliability using supporting patterns, allowing for increased uptime for other functionality if one service should fail.

However, the absence of low-level detailed design for each service removes boundaries for where functionality should begin and where it should end, which may result in microservices becoming bloated and less cohesive over time. Therefore, there must be a constant awareness of the microservice approach in the maintenance of a software solution built in this way.