VIETNAM GENERAL CONFEDERATION OF LABOUR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**



**LÊ MINH ĐỨC – 518H0335**

**VÕ TUẤN LỘC – 518H0390**

**HUGIO – INVENTORY MANAGEMENT SYSTEM**

**INFORMATION TECHNOLOGY PROJECT 2**

**KHOA HỌC MÁY TÍNH**

**HO CHI MINH CITY, 2023**

VIETNAM GENERAL CONFEDERATION OF LABOUR

**TON DUC THANG UNIVERSITY**

**FACULTY OF INFORMATION TECHNOLOGY**



**LÊ MINH ĐỨC – 518H0335**

**VÕ TUẤN LỘC – 518H0390**

**HUGIO – INVENTORY MANAGEMENT SYSTEM**

**INFORMATION TECHNOLOGY PROJECT 2**

**KHOA HỌC MÁY TÍNH**

Supervisor

**Dr. Doãn Xuân Thanh**

**HO CHI MINH CITY, 2023**

**THANKS**

I would like to extend my sincere gratitude to Mr. Doãn Xuân Thanh, the guiding instructor, for dedicating his time and expertise to mentor us through the completion of our Information Technology Project 2. Your guidance not only helped us successfully complete the project but also allowed us to develop valuable research skills and learn many invaluable lessons.

We greatly appreciate your dedication and tireless support throughout this journey. Your advice, feedback, and encouragement have helped us overcome challenges and complete the Information Technology Project 2 with confidence.

This expression of gratitude may not be enough to convey our respect and appreciation, but we genuinely thank you, Mr. Doãn Xuân Thanh, and look forward to continuing to learn and grow under your guidance in the future.

*TP. Hồ Chí Minh, ngày ... tháng … năm 20..*

*Tác giả*

*(Ký tên và ghi rõ họ tên)*

**THE PROJECT WAS COMPLETED**

**AT TON DUC THANG UNIVERSITY**

I hereby certify that this is my own research work and was conducted under the scientific guidance of Dr. Doãn Xuân Thanh. The research content and results in this study are truthful and have not been published in any form prior to this. The data presented in the tables, which serve for analysis, comments, and evaluations, were collected by the author from various sources, as clearly indicated in the reference section.

Furthermore, in this project, some comments, evaluations, and data from other authors and different organizations were used, all of which have been appropriately cited and referenced.

**If any form of misconduct is discovered, I take full responsibility for the content of my project**. Ton Duc Thang University is not liable for any copyright violations or infringements caused by me during the implementation process (if any).

*TP. Hồ Chí Minh, ngày … tháng … năm 20..*

*Tác giả*

*(Ký tên và ghi rõ họ tên)*

**HUGIO – INVENTORY MANAGEMENT SYSTEM**

**SUMMARY**

The application is a warehouse management solution designed for a store. This application is designed to help the store become more efficient in managing and controlling inventory. Below is a summary of the application:

1. Add and Update Products: Allows users to add new products to the warehouse and update information such as name, description, price, and quantity.
2. Inventory Control: Displays the remaining inventory quantity for each product. Users can easily check the status of inventory to ensure there is enough stock to supply customers.
3. Inbound and Outbound Tracking: Allows users to record inbound and outbound transactions. This helps accurately track transaction history and know when to restock or propose promotional programs to move excess inventory.
4. Order Management: Users can create purchase orders for products that need to be restocked. The application will help track the status of purchase orders and provide alerts when stock arrives.
5. Statistics and Reports: Provides statistical and reporting functions regarding inventory status, items sold, revenue, and other key indicators. This helps the store make informed decisions based on statistical data.

* Benefits:
  + Increase warehouse management efficiency and reduce the risk of stockouts or overstocking.
  + Easily generate reports and statistics to formulate more effective business strategies.
  + Save time and effort in inventory control and transaction recording.
  + Enhance customer experience by ensuring the availability of products they need.

**TABLE OF CONTENT**

[TABLE OF FIGURE 7](#_Toc144458162)

[LIST OF ABBREVIATIONS 9](#_Toc144458163)

[CHAPTER 1. INTRODUCTION AND TOPIC OVERVIEW 1](#_Toc144458164)

[1.1 Reasons for choosing the topic 1](#_Toc144458165)

[1.2 Objectives of the project 1](#_Toc144458166)

[CHAPTER 2. APPLICATION ARCHITECTURE 3](#_Toc144458167)

[2.1 Frontend (FE) 3](#_Toc144458168)

[2.1.1 MicroFrontend 3](#_Toc144458169)

[2.1.2 Angular Framework 4](#_Toc144458170)

[2.1.3 Nx Workspace 6](#_Toc144458171)

[2.1.4 Primeng UI Framework 7](#_Toc144458172)

[2.1.5 TailwindCSS 9](#_Toc144458173)

[2.1.6 RxJS 10](#_Toc144458174)

[2.2 Backend (BE) 11](#_Toc144458175)

[2.2.1 MicroService 12](#_Toc144458176)

[2.2.2 Java Spring Boot 14](#_Toc144458177)

[2.2.3 gRPC Communication 18](#_Toc144458178)

[2.2.4 Kafka for Data Streaming 19](#_Toc144458179)

[2.2.5 Redis for Temporary Storage 21](#_Toc144458180)

[2.2.6 MySQL for Data Storage 22](#_Toc144458181)

[CHAPTER 3. APPLICATION DETAIL 23](#_Toc144458182)

[3.1 Frontend 23](#_Toc144458183)

[3.1.1 Overview 23](#_Toc144458184)

[3.1.2 Shell app 25](#_Toc144458185)

[3.1.3 Remote Apps 25](#_Toc144458186)

[3.1.4 Libraries 27](#_Toc144458187)

[3.2 Back end service 27](#_Toc144458188)

[3.2.1 Overview 28](#_Toc144458189)

[3.2.2 Detail 29](#_Toc144458190)

[3.2.3 How service communicate? 57](#_Toc144458191)

[3.2.4 Applying ChatGPT's api to product sales statistics 59](#_Toc144458192)

[CHAPTER 4. DEPLOYMENT, MANAGEMENT AND MONITORING 59](#_Toc144458193)

[4.1 Introduction 59](#_Toc144458194)

[4.1.1 Importance of efficient deployment and management processes 59](#_Toc144458195)

[4.1.2 Tool Selection 60](#_Toc144458196)

[4.2 Version Control and Collaboration with GitHub 60](#_Toc144458197)

[4.3 Continuous Integration and Continuous Deployment (CI/CD) with Jenkins 62](#_Toc144458198)

[4.3.1 Jenkins in Hugio project 62](#_Toc144458199)

[4.3.2 CI/CD Pipeline Stages 63](#_Toc144458200)

[4.3.3 Benefits of CI/CD with Jenkins 73](#_Toc144458201)

[4.4 Containerization with Docker 74](#_Toc144458202)

[4.4.1 Docker in Hugio 74](#_Toc144458203)

[4.4.2 Benefits of Docker 77](#_Toc144458204)

[4.5 Orchestration with Kubernetes and Helm 77](#_Toc144458205)

[4.5.1 Introduction to Kubernetes 77](#_Toc144458206)

[4.5.2 Key Kubernetes Concepts 78](#_Toc144458207)

[4.5.3 Benefits of Kubernetes 78](#_Toc144458208)

[4.5.4 Introduction to Helm 78](#_Toc144458209)

[4.5.5 Helm Charts 78](#_Toc144458210)

[4.5.6 Benefits of Helm 79](#_Toc144458211)

[4.5.7 Helm and Kubernetes in Hugio 79](#_Toc144458212)

[4.6 Web Server Configuration with Nginx 82](#_Toc144458213)

[4.6.1 Introduction to Nginx 82](#_Toc144458214)

[4.6.2 Serving Static Content 82](#_Toc144458215)

[4.6.3 Reverse Proxy Configuration 83](#_Toc144458216)

[4.6.4 Load Balancing 83](#_Toc144458217)

[4.6.5 Benefits of Nginx 83](#_Toc144458218)

[4.7 Deployment Process in Hugio 83](#_Toc144458219)

[4.7.1 Code Commit and Pull Request 83](#_Toc144458220)

[4.7.2 Continuous Integration with Jenkins 84](#_Toc144458221)

[4.7.3 Continuous Deployment to Kubernetes with Helm 84](#_Toc144458222)

[4.7.4 Nginx Configuration and Load Balancing 84](#_Toc144458223)

[4.8 Benefits and Impact 84](#_Toc144458224)

[4.8.1 Streamlined Software Delivery 85](#_Toc144458225)

[4.8.2 Consistency and Reproducibility 85](#_Toc144458226)

[4.8.3 Scalability and Performance 85](#_Toc144458227)

[4.8.4 Resource Efficiency 85](#_Toc144458228)

[4.8.5 Continuous Feedback and Improvement 85](#_Toc144458229)

[4.8.6 Deployment Confidence 86](#_Toc144458230)

[4.8.7 Collaboration and Visibility 86](#_Toc144458231)

[CHAPTER 5. SOFTWARE OVERVIEW 86](#_Toc144458232)

[5.1 Summary page 86](#_Toc144458233)

[5.1.2 Total Orders 87](#_Toc144458234)

[5.1.3 Total Sales 87](#_Toc144458235)

[5.1.4 Cancelled Orders 88](#_Toc144458236)

[5.2 Order page 88](#_Toc144458237)

[5.2.1 Order list 89](#_Toc144458238)

[5.2.2 Create order 90](#_Toc144458239)

[5.3 Client page 92](#_Toc144458240)

[5.3.1 List of Clients 93](#_Toc144458241)

[5.3.2 Creating New Clients 94](#_Toc144458242)

[5.3.3 Editing Client Details 94](#_Toc144458243)

[5.3.4 Deleting Clients 95](#_Toc144458244)

[5.4 Product page 96](#_Toc144458245)

[5.4.1 List of Products 97](#_Toc144458246)

[5.4.2 Creating New Products 97](#_Toc144458247)

[5.4.3 Editing Product Details 98](#_Toc144458248)

[5.4.4 Deleting Products 99](#_Toc144458249)

[5.4.5 Viewing QR Codes 100](#_Toc144458250)

[5.5 Chat page 100](#_Toc144458251)

[5.5.1 Bot-Assisted Communication 101](#_Toc144458252)

[5.5.2 Future Expansion 102](#_Toc144458253)

[CHAPTER 6. CONCLUSION 102](#_Toc144458254)

[6.1 Conclusion 102](#_Toc144458255)

[6.2 Development path 103](#_Toc144458256)

[REFERENCES 105](#_Toc144458257)

# TABLE OF FIGURE

[Figure 1 Microfront-end architecture 3](#_Toc144402400)

[Figure 2 Angular logo 5](#_Toc144402401)

[Figure 3 NX Workspace 6](#_Toc144402402)

[Figure 4 Prime NG 8](#_Toc144402403)

[Figure 5 Tailwind CSS 9](#_Toc144402404)

[Figure 6 RxJS 11](#_Toc144402405)

[Figure 7 Microservice architecture 12](#_Toc144402406)

[Figure 8 Spring framework 14](#_Toc144402407)

[Figure 9 Spring data flow 16](#_Toc144402408)

[Figure 10 Dependency Injection 17](#_Toc144402409)

[Figure 11 gRPC flow 18](#_Toc144402410)

[Figure 12 Kafka flow visualization 20](#_Toc144402411)

[Figure 13 Redis flow visualization 21](#_Toc144402412)

[Figure 14 HUGIO microfrontend flow 24](#_Toc144402413)

[Figure 15 HUGIO Microservice flow 28](#_Toc144402414)

[Figure 16 Login sequence diagram 29](#_Toc144402415)

[Figure 17 Retrieve info Sequence diagram 30](#_Toc144402416)

[Figure 18 A flow show how HUGIO using Kafka 58](#_Toc144402417)

[Figure 19 Hugio front end github page 61](#_Toc144402418)

[Figure 20 Hugio back end github page 62](#_Toc144402419)

[Figure 21 Hugio's jenkins dashboard 63](#_Toc144402420)

[Figure 22 Hugio dashboard for back end 69](#_Toc144402421)

[Figure 23 Hugio K8S Dashboard for Front end deployment 79](#_Toc144402422)

[Figure 24 Hugio K8S Dashboard for Front end pod 80](#_Toc144402423)

[Figure 25 Hugio K8S Dashboard for Front end service 80](#_Toc144402424)

[Figure 26 Hugio K8S Dashboard for Back end deployment 81](#_Toc144402425)

[Figure 27 Hugio K8S Dashboard for Back end pods 81](#_Toc144402426)

[Figure 28 Hugio K8S Dashboard for Back end services 82](#_Toc144402427)

[Figure 29 Summary page 86](#_Toc144402428)

[Figure 30 List of Order page 89](#_Toc144402429)

[Figure 31 Create Product page 91](#_Toc144402430)

[Figure 32 List of clients page 93](#_Toc144402431)

[Figure 33 Product list page 96](#_Toc144402432)

[Figure 34 Chat page 101](#_Toc144402433)

# LIST OF ABBREVIATIONS

|  |  |
| --- | --- |
| MFE | Micro Front-end |
| MSV | Micro Service |
| K8S | Kubernetes |
| FE | Frontend |
| BE | Backend |

# INTRODUCTION AND TOPIC OVERVIEW

## Reasons for choosing the topic

* Urgency: Warehouse management is a crucial aspect of every store's business operations. Understanding efficient inventory management and control can help the store conserve resources and optimize business processes.
* Real-world application: Store owners often face challenges in maintaining a balance between inventory and customer demands. Mastering inventory management techniques can assist them in maximizing profits and enhancing customer experiences.
* Learning management skills: Studying how to construct and implement a warehouse management application provides an opportunity for you to develop and enhance skills in design, software development, and project management.
* Advancements in the technology field: The popularity of mobile applications and management software has generated an increasing demand for inventory management solutions. Seizing this opportunity can provide you with a deeper understanding of the synergy between business and information technology.

## Objectives of the project

* Analysis of Needs and Requirements: Grasp the store's requirements for warehouse management. This involves identifying product types, desired inventory levels, and methods for recording inbound and outbound transactions.
* Interface and Function Design: Set goals for building a user-friendly interface and necessary functionalities in the warehouse management application. This may include adding, updating, and deleting products, tracking inventory, creating purchase orders, and generating reports.
* Application Development: Establish development targets based on the outlined design. This includes programming functionalities, building a database to store product information and transactions, and ensuring the stability and security of the application.
* Testing and Bug Fixing: Set objectives to conduct comprehensive testing of the application, identifying and fixing any bugs. This ensures that the application operates as expected and effectively meets the store's needs.
* Deployment: Propose goals for how to deploy the application into the store's real-world environment.

# APPLICATION ARCHITECTURE

## Frontend (FE)

### MicroFrontend

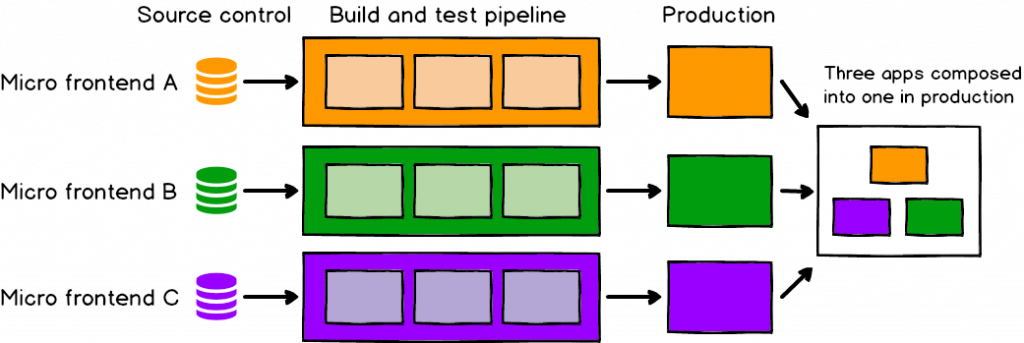


Figure 1 Microfront-end architecture

Microfrontend is an architectural approach that extends the concept of microservices to the frontend development of applications. It involves breaking down a complex user interface into smaller, self-contained modules or "microfrontends," each responsible for a specific functionality or feature. These microfrontends can be developed, deployed, and maintained independently, allowing different teams to work on different parts of the application without tightly coupling them.

In a traditional frontend development approach, a monolithic frontend codebase handles the entire user interface. This can lead to challenges in terms of scalability, maintainability, and collaboration, especially in larger applications with multiple teams working simultaneously.

Microfrontend architecture addresses these challenges by promoting modularity and separation of concerns in the frontend domain. Each microfrontend is developed using a technology stack that best fits its requirements, allowing teams to choose the tools they are most comfortable with.

Key concepts of microfrontend architecture include:

1. Independence: Microfrontends are self-contained units that can be developed, tested, and deployed independently. This enables teams to work autonomously and release changes without affecting the entire application.
2. Isolation: Microfrontends are isolated from one another, which means that issues in one microfrontend do not directly impact others. This isolation improves the overall stability of the application.
3. Integration: While microfrontends are developed separately, they must be integrated to form a coherent user experience. This integration can be achieved through various techniques, such as server-side composition or client-side routing.
4. Team Autonomy: Different teams can work on different microfrontends, enabling faster development cycles and specialization in specific areas of expertise.
5. Improved Scalability: Microfrontend architecture allows an application to scale more efficiently, as individual parts can be scaled independently based on demand.
6. Technology Diversity: Since microfrontends are independent, teams can choose the most suitable technology stack for their specific use case without being restricted by a monolithic frontend architecture.

Overall, microfrontend architecture promotes flexibility, collaboration, and rapid development in frontend application projects. However, it also introduces challenges related to cross-microfrontend communication, shared resources, and maintaining a consistent user experience across different modules. Proper planning, communication, and the use of appropriate tools can help mitigate these challenges and harness the benefits of this architectural approach.

### Angular Framework



Figure 2 Angular logo

Angular is a powerful and popular open-source front-end development framework created and maintained by Google. It's used for building single-page applications (SPAs) and dynamic web applications. Angular provides a comprehensive set of tools, libraries, and features that streamline the process of creating interactive and responsive user interfaces.

* Component-Based Architecture:
  + Angular follows a component-based architecture, where the user interface is broken down into reusable and encapsulated components. Components encapsulate the HTML, CSS, and TypeScript logic related to a specific part of the user interface.
* Reactive Data Binding:
  + Angular uses two-way data binding, allowing changes in the user interface to automatically update the underlying data model, and vice versa. This reactive approach simplifies managing data synchronization between components and the application state.
* Dependency Injection:
  + Angular's dependency injection system enables the efficient management of component dependencies. This promotes modularity, reusability, and testability by allowing components to receive required services and dependencies from external sources.
* Directives:
  + Directives in Angular are used to extend HTML with custom behaviors. They include structural directives (like \*ngIf and \*ngFor) and attribute directives (e.g., ngStyle, ngClass) that enhance the behavior and appearance of HTML elements.
* Modules:
  + Angular applications are modularized into NgModule units. Modules help organize the application by grouping components, services, and other features together. They also facilitate lazy loading, allowing parts of the application to be loaded on-demand for improved performance.
* Services:
  + Services in Angular are used to provide reusable functionality across components. They can handle data retrieval, business logic, communication with APIs, and more. Services are typically injected into components through dependency injection.

### Nx Workspace



Figure 3 NX Workspace

Nx is a set of powerful development tools and libraries designed to enhance the development process for Angular applications and monorepos. Nx Workspace, in particular, provides a suite of features that help developers manage complex projects, increase collaboration, and maintain code quality.

* Monorepo Management:
  + Nx promotes the use of a monorepo structure, where multiple projects (apps and libraries) coexist within a single repository. This approach simplifies code sharing, promotes reusability, and streamlines the development workflow.
* Workspace Scaffolding:
  + Nx Workspace offers a command-line interface (CLI) that enables developers to generate projects, apps, and libraries using predefined templates. This consistency ensures that projects adhere to best practices from the start.
* Dependency Graph:
  + One of Nx's standout features is its dependency graph. It analyzes the relationships between projects and determines the optimal order for building and testing. This improves build and test times, especially in monorepo environments.
* Code Generation:
  + Nx provides code generation capabilities for creating components, modules, services, and more. Developers can quickly scaffold out different parts of their applications, ensuring consistency in structure and naming conventions.
* Task Automation:
  + Nx Workspace includes task automation features that help streamline development tasks. These tasks can be automated using Nx's command-line interface, allowing for consistent execution of common tasks across projects.
* Testing and Continuous Integration:
  + Nx includes tools for running tests across different projects and generating code coverage reports. It also integrates well with popular continuous integration (CI) tools, enabling automated testing and deployment pipelines.

### Primeng UI Framework



Figure 4 Prime NG

PrimeNG is a popular open-source User Interface (UI) component library for Angular applications. It offers a comprehensive set of pre-designed UI components that enable developers to build modern, visually appealing, and user-friendly web applications.

* Rich Component Library:
  + PrimeNG provides a wide range of UI components that cover various aspects of user interface development. These include buttons, forms, tables, charts, dialog boxes, calendars, menus, and more.
* Material Design:
  + PrimeNG components are designed following Google's Material Design principles. This ensures a consistent and visually pleasing user experience that aligns with modern design standards.
* Customizability:
  + While PrimeNG components come with default styles adhering to Material Design, they are highly customizable. Developers can easily modify styles, themes, and appearance to match the application's branding and design.
* Accessibility:
  + PrimeNG places a strong emphasis on accessibility. Components are built with accessibility features in mind, ensuring that users with disabilities can use the application effectively.
* Responsive Design:
  + PrimeNG components are designed to be responsive, adapting seamlessly to various screen sizes and devices. This helps in creating applications that work well across desktops, tablets, and mobile devices.

### TailwindCSS



Figure 5 Tailwind CSS

Tailwind CSS is a modern, open-source CSS framework that provides a utility-first approach to building user interfaces. It emphasizes a highly efficient and modular way of styling by offering a comprehensive set of pre-designed utility classes that can be directly applied to HTML elements.

* Utility-First Philosophy:
  + Unlike traditional CSS frameworks that encourage writing custom CSS classes, TailwindCSS promotes using utility classes to apply styles directly in HTML. This approach speeds up development by reducing the need for custom CSS.
* Responsive Design:
  + Tailwind CSS offers responsive utility classes, making it easy to create responsive layouts and adapt designs to various screen sizes and devices.
* Component-Oriented Development:
  + Tailwind CSS enables developers to compose complex components by combining utility classes. This modular approach leads to reusable and maintainable code.
* Customization:
  + While Tailwind CSS provides a default set of styles, it's highly customizable. Developers can configure color schemes, typography, spacing, and other aspects of the design according to project requirements.
* Build Process:
  + Tailwind CSS is not a traditional CSS framework with predefined styles. Instead, it provides a set of utility classes that you can use. This means that you need to run a build process to generate optimized CSS files for production.

### RxJS



Figure 6 RxJS

RxJS (Reactive Extensions for JavaScript) is a library that brings reactive programming concepts to JavaScript applications. It allows developers to work with asynchronous data streams, enabling elegant and powerful ways to handle events, asynchronous operations, and data manipulation.

* Observables:
  + Observables represent asynchronous data streams. They can emit multiple values over time, including data, events, or even errors. Observables provide a unified way to handle various types of asynchronous operations.
* Operators:
  + Operators are functions that transform, filter, or combine observables. RxJS offers a rich set of operators that enable powerful data manipulation, such as mapping, filtering, merging, and more.
* Subjects:
  + Subjects are a special type of observable that allows multicasting. They serve as both an observer and an observable, making it possible to broadcast events to multiple subscribers.
* Streams and Pipelines:
  + RxJS allows developers to build data processing pipelines by chaining operators. This approach promotes a functional programming style for handling data streams.

## Backend (BE)

### MicroService

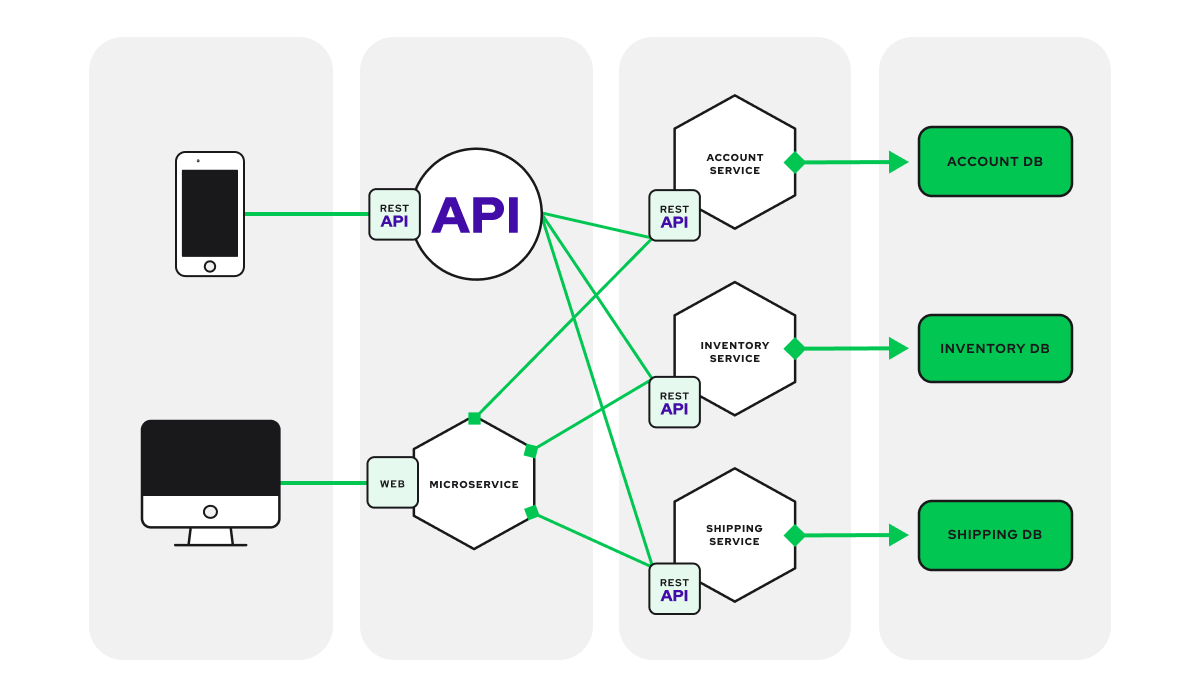


Figure 7 Microservice architecture

Microservices is an architectural style for designing and building software applications as a collection of small, loosely coupled, and independently deployable services. In this approach, an application is divided into multiple small services, each responsible for a specific business capability. These services communicate with each other through well-defined APIs, often over a network, and can be developed, deployed, and scaled independently.

The main characteristics of microservices include:

1. Decomposition: Instead of building a monolithic application with all functionalities tightly integrated, microservices advocate breaking down the application into smaller, manageable parts. Each service is responsible for a distinct business function.
2. Independence: Each microservice can be developed, deployed, and scaled independently of other services. This provides flexibility for teams to work on different services concurrently and release updates without affecting the entire application.
3. Technology Diversity: Different microservices can use different technology stacks that best suit their requirements. This allows teams to choose the most suitable programming languages, frameworks, and tools for each service.
4. Scalability: Microservices can be scaled individually based on their specific load, rather than scaling the entire application. This efficient scaling helps handle varying levels of demand.
5. Resilience: Microservices are designed to handle failures gracefully. If one service experiences an issue, it should not bring down the entire application, as other services can continue to function.
6. Ease of Maintenance: Smaller, focused services are generally easier to understand, maintain, and troubleshoot. Changes or updates to one service have minimal impact on others.
7. Autonomy: Microservices enable different teams to take ownership of specific services, leading to greater autonomy and faster development cycles.
8. API-Based Communication: Microservices interact with each other through well-defined APIs, often using lightweight protocols like HTTP/REST or gRPC. This enables efficient communication and loose coupling between services.

While microservices offer several benefits, they also introduce challenges such as managing inter-service communication, data consistency across services, and ensuring proper monitoring and observability. Implementing a microservices architecture requires careful consideration of the application's requirements, the organization's structure, and the tools and practices needed to manage the complexities introduced by distributed systems.

Overall, the microservices approach supports building applications that are more adaptable, scalable, and resilient, making it well-suited for modern, complex software development scenarios.

### Java Spring Boot

Spring Boot is a framework built on top of the Spring framework that simplifies the process of creating stand-alone, production-grade Spring-based applications. It provides a wide range of tools and libraries for building robust and scalable Java applications with minimal configuration.

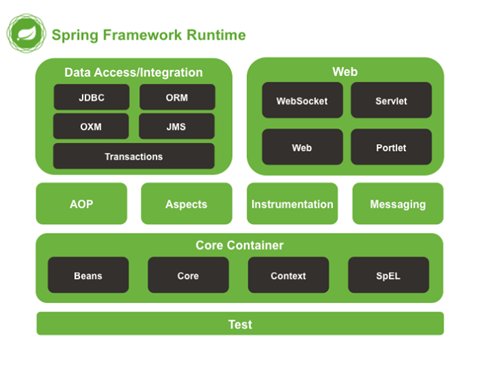


Figure 8 Spring framework

Some key aspects and features of the Spring framework:

* *Inversion of Control (IoC)*: One of the fundamentals of Spring is IoC. It allows objects to be loosely coupled and managed by the framework itself. Instead of directly creating and managing objects, the framework controls the initialization and inclusion of dependencies, promoting modularity and easier unit testing.
* *Dependency Injection (DI)*: DI is a core concept in Spring that complements IoC. It allows you to define the dependencies of an external object, usually through configuration files or annotations. Spring takes care of injecting runtime dependencies, reducing tight coupling and increasing flexibility.
* *Aspect-Oriented Programming (AOP)*: AOP is a powerful feature in Spring that allows you to modularize cross-cutting concerns, such as logging, transaction management, and security, into separate modules called aspects. This approach improves code modularity and maintainability by decoupling business logic from system-level services.
* *Spring MVC*: Spring provides a web framework called Spring MVC (Model-View-Controller), which offers a structured approach to building web applications. It follows the MVC pattern and provides features like request handling, view resolution, data binding, and validation. Spring MVC integrates seamlessly with other Spring components, making it a popular choice for building web applications.
* *Spring Boot*: Spring Boot is a project within the Spring ecosystem that aims to simplify the setup and configuration of Spring applications. It provides a convention-over-configuration approach, eliminating the need for boilerplate code and XML configuration. Spring Boot includes an embedded server, auto-configuration of dependencies, and production-ready features, allowing developers to quickly build standalone Spring applications.
* *Data Access*: Spring offers robust support for database integration and data access. It provides the Spring Data module, which abstracts the underlying data store and simplifies database operations through repositories and query methods. Spring also integrates with popular Object-Relational Mapping (ORM) frameworks like Hibernate, providing seamless persistence support.
* *Testing*: Spring provides excellent support for unit testing and integration testing. It offers various testing utilities and annotations to write clean and comprehensive tests. With Spring's testing framework, you can easily mock dependencies, test Spring MVC controllers, and perform integration testing against different layers of your application.
* *Security*: Spring Security is a powerful module that provides comprehensive security features for web applications. It supports authentication, authorization, and various security mechanisms like form-based authentication, OAuth, and JWT (JSON Web Tokens). Spring Security integrates seamlessly with other Spring modules and allows fine-grained control over application security.

#### ***Data flow in a single request of client***

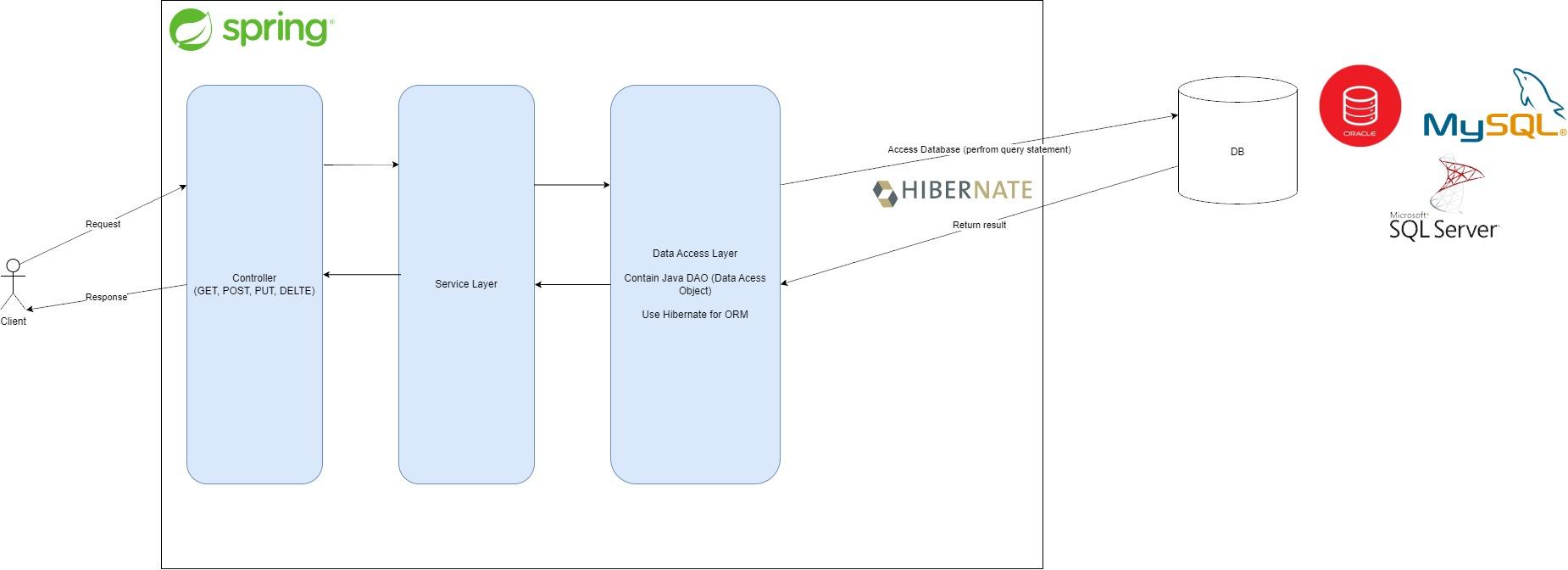


Figure 9 Spring data flow

*Client/Request*: A data stream initiated with a client or request from an external source. It could be a user making an HTTP request to a Spring MVC controller, a message from a message queue, or any other form of data entry.

*Controller/Endpoint*: In web application, client request is handled by Spring MVC controller or endpoint. The controller receives the request, processes it, and prepares the data to pass on to the next layer.

*Service Layer*: The controller delegates business logic and data processing to the service layer. The service layer contains the core application logic, which may involve data manipulation, authentication, calling external services, or interacting with repositories.

*Data access layer*: The data access layer, commonly known as the repository layer, is responsible for interacting with the database or any other data storage mechanism. It uses Spring's data access technologies, such as Spring Data JPA or Spring JDBC, to perform CRUD (Create, Read, Update, Delete) operations on the data.

*Data source*: Data source refers to the actual storage where the application persists or retrieves data. It can be a relational database, a NoSQL database, a file system or any other external system.

*ORM/Mapper*: If an Object Relational Mapping (ORM) framework, such as Hibernate, is used, it acts as a bridge between the application and the database. It maps Java objects to database tables and handles data translation between the application and the database.

*Response*: Once the data is retrieved or modified, the data goes back through the layers in reverse order. The service layer processes the data, prepares the response, and passes it back to the controller.

Show view/response: In a web application, the controller forwards the response to the view or template engine for rendering. The view generates the appropriate HTML, XML, JSON, or whatever other format is required to send the response back to the client.

*Client/Response*: Finally, the processed data or response is sent back to the client, completing the data flow cycle.

#### ***Dependency Injection***

First, to depend or depend means to depend on the support of something, something. For example, if we spend a lot of time using smartphones, it can be understood that we have become dependent on smartphones and smartphones are one of our dependencies.

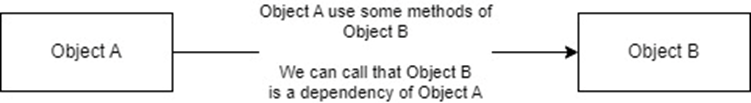


Figure 10 Dependency Injection

In java, before we can use methods of another class, we must instantiate an object of that class (or A needs to create an instance of B). So, we can understand, when Object A is initialized in the JVM, then Object B **must be initialized before** Object A and then **put into** Object A, or in other words, when Object A is created, our inclusion of Object B Object A is called **injection.**

In Spring, there are three options for how dependencies can be injected into a Object:

1. Through a constructor: the dependencies are provided through the constructor (class constructor).
2. Through setters or other methods: dependencies will be passed into a class through setter methods (setter functions).
3. Through reflection, directly into fields: Using a annotation provide by Spring (@Autowired).

### gRPC Communication

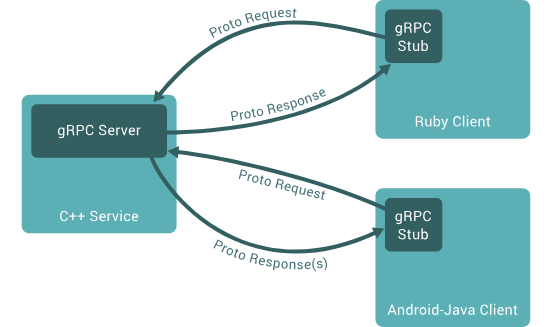


Figure 11 gRPC flow

gRPC is a high-performance, open-source framework for remote procedure calls (RPC). It enables services to communicate with each other over a network using a binary protocol, making it more efficient compared to traditional protocols like JSON over HTTP. gRPC is often used in microservices architectures.

#### ***Why we need gRPC?***

During the heyday of REST APIs, the basic interaction between clients and servers was reasonably effective. However, in the era of Microservices, it has become clear that there is a need for a superior approach to enhance scalability and boost the flow of data between services.

This might not seem like a significant concern, especially when dealing with systems featuring only a few services and a limited number of servers or nodes. However, the context here involves numerous services and a substantial load, such as hundreds of services and concurrent users exceeding 100k CCU (Concurrent Users - the number of users actively engaged simultaneously).

In such a scenario, when a request necessitates the aggregation of data from multiple services, each receiving service is burdened with the continuous encoding and decoding of data, typically in JSON format. This can unnecessarily strain the CPUs. Ideally, CPUs should be allocated to more critical tasks, rather than being tied up in the encoding and decoding of intermediary data.

The underlying concept of how to facilitate high-speed communication between services while alleviating the burden of data encoding and decoding is what drives the adoption of gRPC.

### Kafka for Data Streaming

Apache Kafka is a distributed event streaming platform that allows you to publish and subscribe to streams of records (events). It's commonly used for building real-time data pipelines and streaming applications, enabling reliable communication and data processing between different parts of an application.

Kafka is built on a publish/subscribe model, similar to any other messaging system. Applications acting as producers send messages (records) to a Kafka node (broker) and indicate that these messages will be processed by other applications known as consumers. The messages sent to the Kafka node are stored in a location called a topic, and then consumers can subscribe to that topic and listen to these messages. Messages can be any type of information, such as sensor readings, user actions, and more.

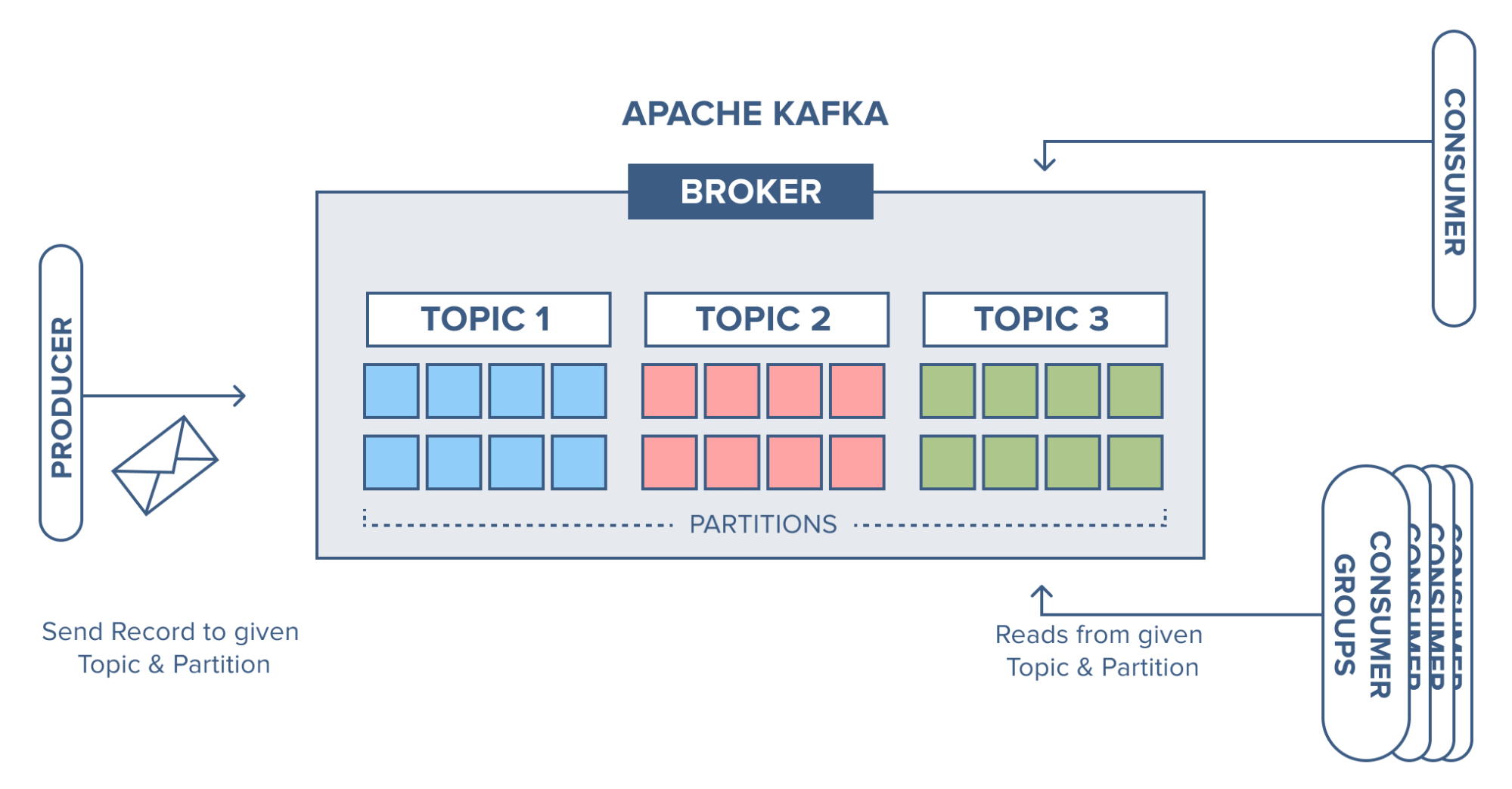


Figure 12 Kafka flow visualization

The flow of sending and listening of a message can be describe as following:  
 *Sending (Producer)*: An application acts as a producer, creating messages and sending them to a Kafka broker. The producer decides which topic to place the messages in.

*Topic*: A topic is a virtual category in Kafka where messages are stored. The producer sends messages to a specific topic.

Storage: Messages are stored within the Kafka broker. Kafka uses a log-based data structure to maintain these messages.

*Listening (Consumer)*: A consumer is another application acting as a listener. It subscribes to one or more topics of interest. When new messages arrive in a topic it has subscribed to, the consumer receives and processes them.

Processing: The consumer processes the received messages according to the requirements of the application. Processing can involve actions such as storing in a database, performing computations, or even forwarding messages to another system.

*Iteration (Repeat)*: This cycle continues as the producer keeps sending new messages, and the consumer keeps listening and processing them.

### Redis for Temporary Storage

Redis is an in-memory data store that can be used as a cache or temporary storage solution. It's known for its high-speed data retrieval and ability to handle complex data structures. Redis is often used to store frequently accessed data to reduce the load on other parts of the application.

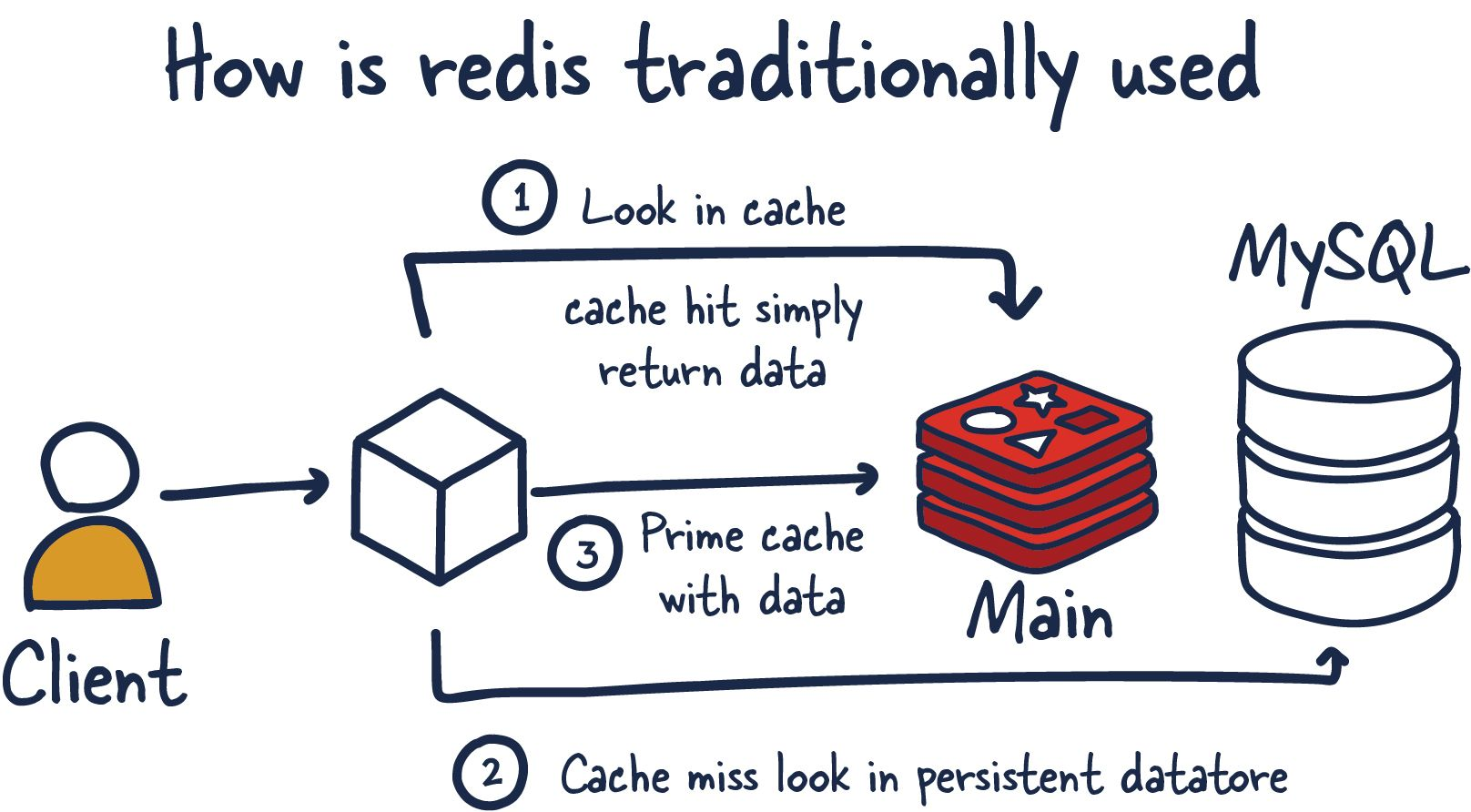


Figure 13 Redis flow visualization

The benefits of using Redis:

*In-Memory Data Store*: All Redis data resides in the server's main memory, in contrast to databases like PostgreSQL, Cassandra, MongoDB, and others that store most of their data on disk or SSD. Unlike traditional disk-based databases, most operations with Redis do not require roundtrips to disk since it stores data in-memory. Therefore, it can handle a large number of executions and provide faster response times. As a result, Redis offers faster performance for read or write operations, typically taking less than a millisecond, and supports millions of operations per second.

*Flexible Data Structures*: Unlike other simple key-value stores that provide limited data structures, Redis offers a wide range of data structures to meet your application's needs, including strings, lists, sets, sorted sets, and hashes.

*Simplicity and Ease of Use*: Redis simplifies your code by allowing you to write fewer lines of code to store, access, and use data in your applications. For example, if your application has data stored in a hashmap and you want to store that data in a database, you can simply use the hash data structure to store it. Performing a similar operation on a database without hash data structures would require more lines of code to convert from one format to another. For each data type, Redis provides numerous options for manipulating and interacting with your data.

*Replication and Persistence*: Redis uses a primary-replica architecture and supports asynchronous replication, allowing data to be copied to multiple servers. This improves read performance (as requests can be distributed among servers) and provides faster recovery when the primary server encounters a failure. To maintain data integrity, Redis supports point-in-time backups, where Redis data is copied to disk.

### MySQL for Data Storage

MySQL is a popular open-source relational database management system. It's widely used for storing structured data and offers features like data integrity, ACID compliance, and powerful querying capabilities. MySQL is suitable for various applications, from small projects to large-scale systems.

# APPLICATION DETAIL

## Frontend

### Overview

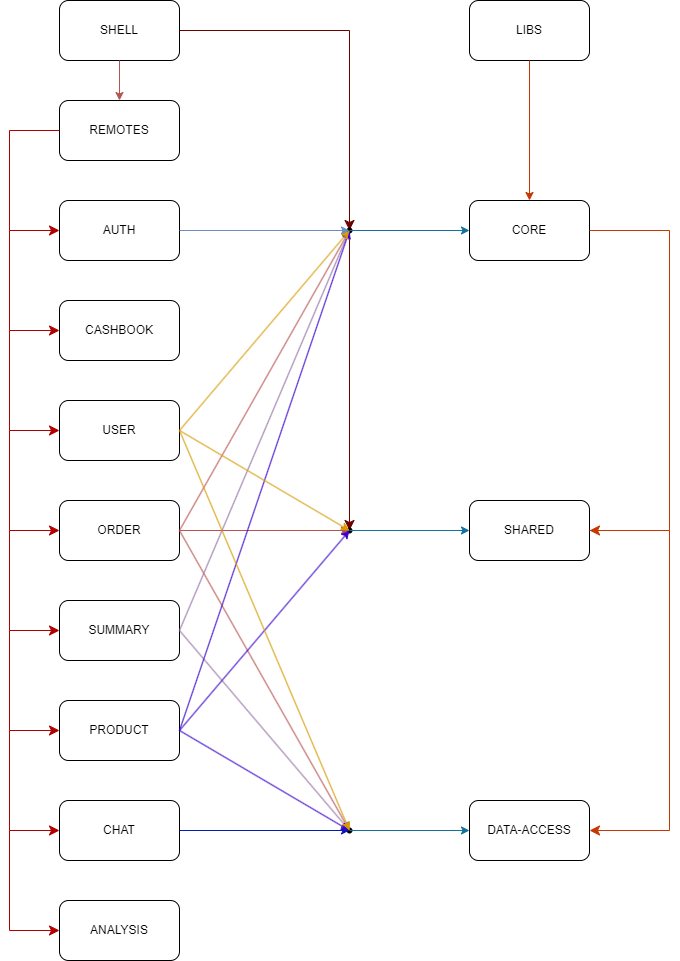


Figure 14 HUGIO microfrontend flow

* Detail the structure of the application:
  + Shell App: Central application responsible for rendering and integrating remote apps.
  + Remote Apps: Individual microfrontends responsible for specific functionalities (analysis, cashbook, chat, order, summary, product, user, auth).
  + Libraries: Data-access, core, and shared libraries used across the application.

### Shell app

* Explain the role of the shell app:
  + Integration of remote apps using routing or iframes.
  + Navigation and layout management.
  + Handling global state and communication between remote apps.

### Remote Apps

* Detail each remote app:
  + Purpose and functionality of each app (analysis, cashbook, chat, etc.).
  + How each app is developed, including technologies used.
  + Interaction with the shell app and other remote apps.
* Analysis App:
  + Purpose: Provides data analysis and visualization tools for business insights.
  + Functionality: Displays charts, graphs, and reports based on data retrieved from backend services.
  + Technologies: Angular, Nx Workspace, Data-access library, Charting libraries.
* Cashbook App:
  + Purpose: Manages financial transactions and records.
  + Functionality: Allows users to input, view, and manage transactions and balances.
  + Technologies: Angular, Nx Workspace, Data-access library, Form components.
* Chat App:
  + Purpose: Enables real-time communication among users.
  + Functionality: Supports text-based chat and message exchange.
  + Technologies: Angular, Nx Workspace, Real-time communication libraries.
* Order App:
  + Purpose: Handles order management and processing.
  + Functionality: Allows users to place, track, and manage orders.
  + Technologies: Angular, Nx Workspace, Data-access library, Order processing logic.
* Summary App:
  + Purpose: Provides an overview of key performance indicators (KPIs) and summaries.
  + Functionality: Displays aggregated data and performance metrics in visual formats.
  + Technologies: Angular, Nx Workspace, Data-access library, Visualization components.
* Product App:
  + Purpose: Manages product information and inventory.
  + Functionality: Enables users to view, search, and update product details and quantities.
  + Technologies: Angular, Nx Workspace, Data-access library, Product management logic.
* User App:
  + Purpose: Manages user accounts and profiles.
  + Functionality: Allows users to create accounts, update profile information, and manage preferences.
  + Technologies: Angular, Nx Workspace, Data-access library, User account management.
* Auth App:
  + Purpose: Handles user authentication and authorization.
  + Functionality: Provides login, registration, and session management features.
  + Technologies: Angular, Nx Workspace, Data-access library, Authentication logic.

### Libraries

* Data-access Library:
  + Purpose: Provides a centralized way to interact with backend APIs.
  + Functionality: Handles API calls, data retrieval, and data transformation.
  + Technologies: Angular Services, HttpClientModule.
* Core Library:
  + Purpose: Houses core business logic and shared services.
  + Functionality: Contains services for authentication, logging, error handling, etc.
  + Technologies: Angular Services, RxJS for state management.
* Shared Library:
  + Purpose: Offers reusable UI components and styles across apps.
  + Functionality: Provides consistent styling, shared components, and utilities.
  + Technologies: Angular Components, SCSS for styling.

## Back end service

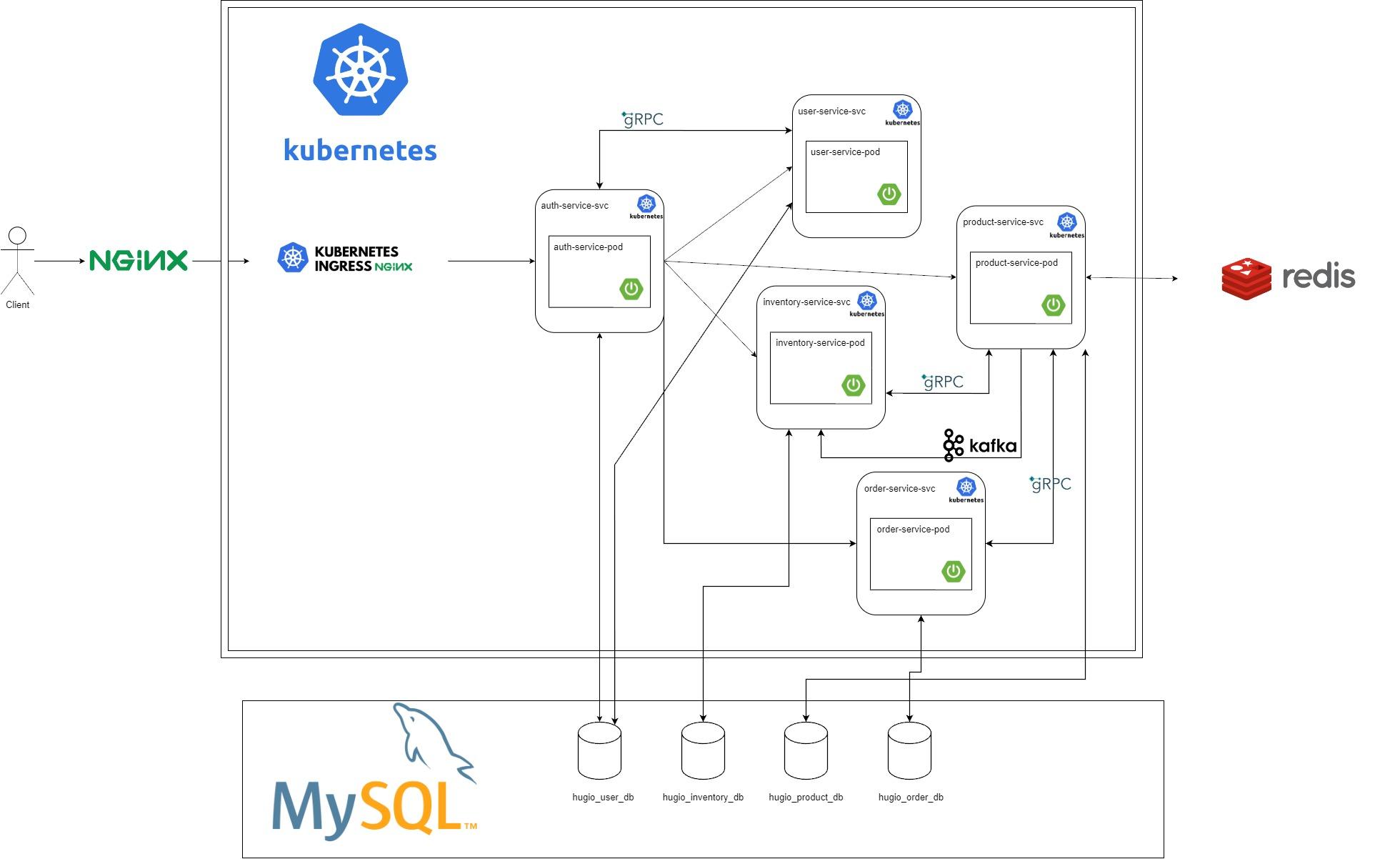


Figure 15 HUGIO Microservice flow

### Overview

There are will be 5 service:

*Auth Service*: The Authentication Service, or Auth Service, is the cornerstone of your microservices architecture, ensuring secure access to your system. It handles user authentication and authorization, safeguarding your application's data and functionalities.

*User Service*: The User Service is responsible for managing user profiles and accounts. It facilitates user registration, profile updates, and user-related operations, offering a personalized experience within your ecosystem.

*Product Service*: The Product Service manages the heart of your business - the products. It handles product information, catalog management, and ensures that your users can browse and interact with your offerings seamlessly.

*Inventory Service*: The Inventory Service is crucial for tracking and managing product availability. It helps you maintain accurate stock levels, ensuring that your customers can purchase products without any hiccups.

*Order Service*: The Order Service orchestrates the complete order lifecycle. It allows users to place orders, handles order processing, and monitors order status, providing transparency and efficiency in the order fulfillment process.

### Detail

#### ***3.2.1.1 Auth Service***

##### ***3.2.1.1.1 Flow***

This service contain the following function:

* Login:

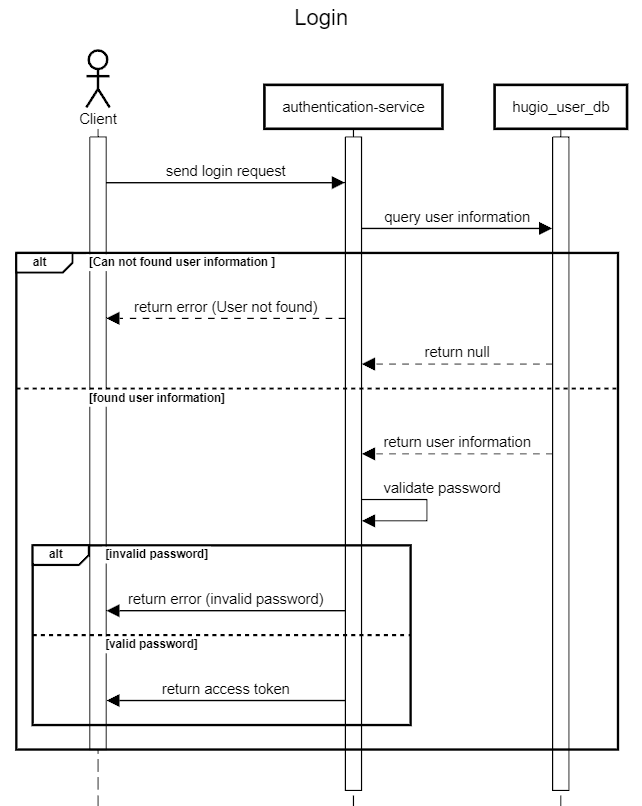


Figure 16 Login sequence diagram

* Retrieve info

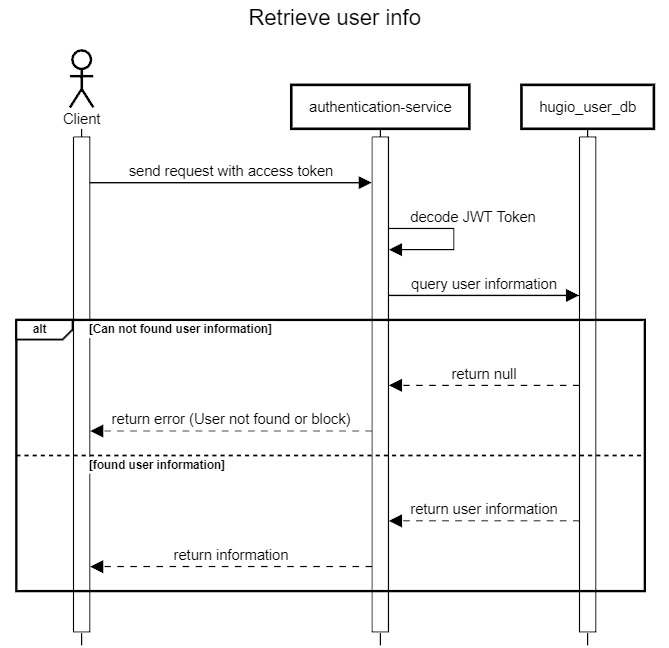


Figure 17 Retrieve info Sequence diagram

##### ***3.2.1.1.2 Database***

The database structure of this service is designed as follows:

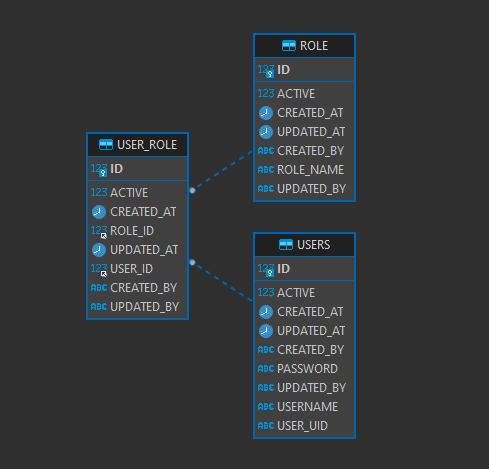


Figure 18 Hugio auth service database

The **USERS** is used for storing user account information, including usernames, passwords, creation/update timestamps, and related user details. This table contain the following information:

* *ACTIVE*: This is a boolean column (BIT data type) that stores whether the user account is active (1) or not (0).
* *CREATED\_AT*: This column of type datetime(6) records the timestamp when the user account was created.
* *ID*: This is the primary key of the table, of type bigint, and it auto-increments with each new user record.
* *UPDATED\_AT*: Similar to CREATED\_AT, this column of type datetime(6) records the timestamp when the user account was last updated.
* *CREATED\_BY*: This is a varchar(255) column that might store information about the user who created the account. It has a default value of NULL.
* *PASSWORD*: This column of type varchar(255) stores the user's password. It is marked as NOT NULL, meaning a password must be provided.
* *UPDATED\_BY*: Similar to CREATED\_BY, this column might store information about the user who last updated the account. It also has a default value of NULL.
* *USERNAME*: This varchar(255) column stores the username of the user and is marked as NOT NULL.
* *USER\_UID*: This varchar(255) column stores a user UID (user identifier) and is marked as NOT NULL.
* The ID column serves as the primary key for the table.

The **ROLE** is used for storing role information, including role names, creation/update timestamps, and related details. It might be associated with user roles and permissions within the application's user management system. This table contain the following information:

* *ACTIVE*: This is a boolean column (BIT data type) that stores whether the role is active (1) or not (0).
* *CREATED\_AT*: This column of type datetime(6) records the timestamp when the role was created.
* *ID*: This is the primary key of the table, of type bigint, and it auto-increments with each new role record.
* *UPDATED\_AT*: Similar to CREATED\_AT, this column of type datetime(6) records the timestamp when the role was last updated.
* *CREATED\_BY*: This is a varchar(255) column that might store information about the user who created the role. It has a default value of NULL.
* *ROLE\_NAME*: This varchar(255) column stores the name of the role and is marked as NOT NULL.
* *UPDATED\_BY*: Similar to CREATED\_BY, this column might store information about the user who last updated the role. It also has a default value of NULL.
* The ID column serves as the primary key for the table.

The **USER\_ROLE** is used for establishing associations between users and roles, possibly for managing user permissions and access control within the application. This table contain the following information:

* *ACTIVE*: This is a boolean column (BIT data type) that stores whether the user-role association is active (1) or not (0).
* *CREATED\_AT*: This column of type datetime(6) records the timestamp when the user-role association was created.
* *ID*: This is the primary key of the table, of type bigint, and it auto-increments with each new user-role association record.
* *ROLE\_ID*: This column of type bigint stores the ID of the role associated with the user.
* *UPDATED\_AT*: Similar to CREATED\_AT, this column of type datetime(6) records the timestamp when the user-role association was last updated.
* *USER\_ID*: This column of type bigint stores the ID of the user associated with the role.
* *CREATED\_BY*: This is a varchar(255) column that might store information about the user who created the user-role association. It has a default value of NULL.
* *UPDATED\_BY*: Similar to CREATED\_BY, this column might store information about the user who last updated the user-role association. It also has a default value of NULL.
* The ID column serves as the primary key for the table.

#### ***3.2.1.2 User Service***

##### ***3.2.1.2.1 Flow***

This service contain the following function:

* Get all user

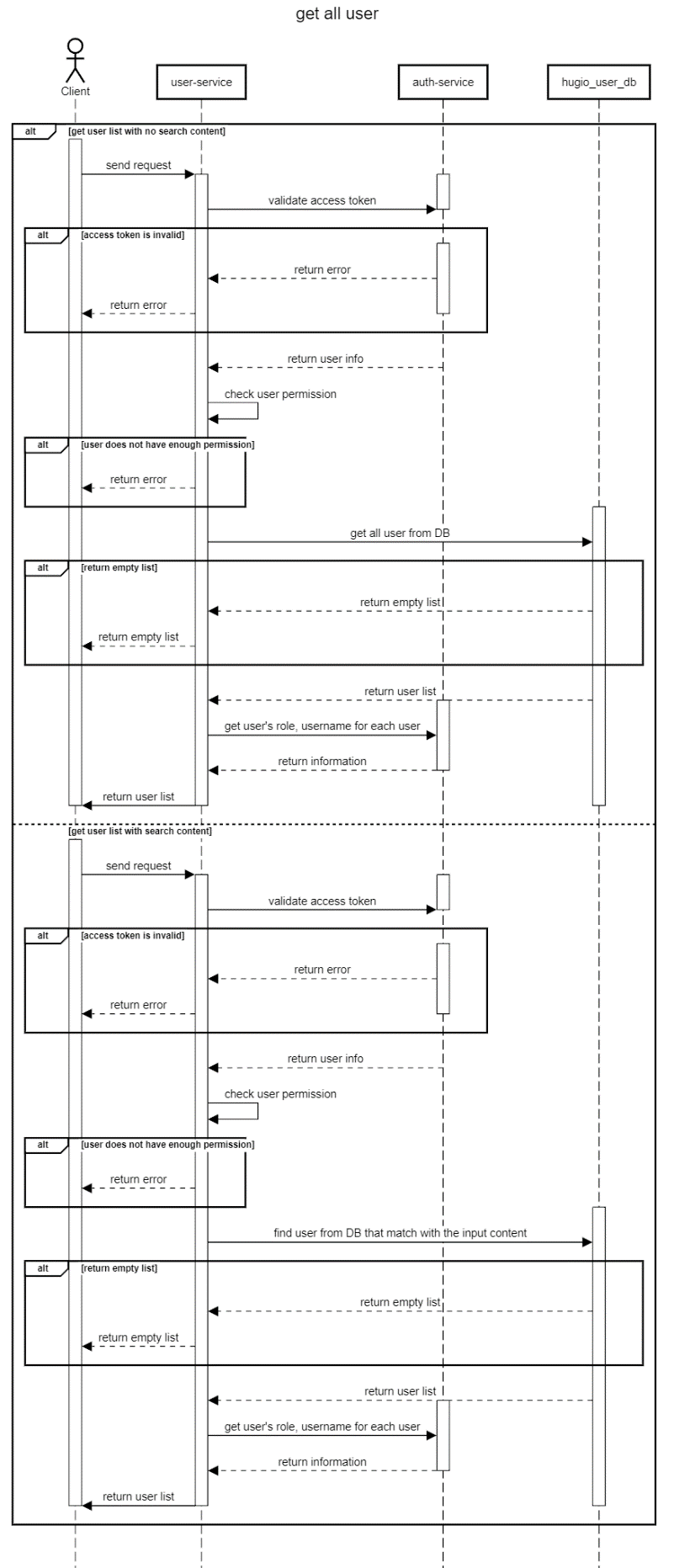


Figure 19 Get all user sequence diagram

* Create user

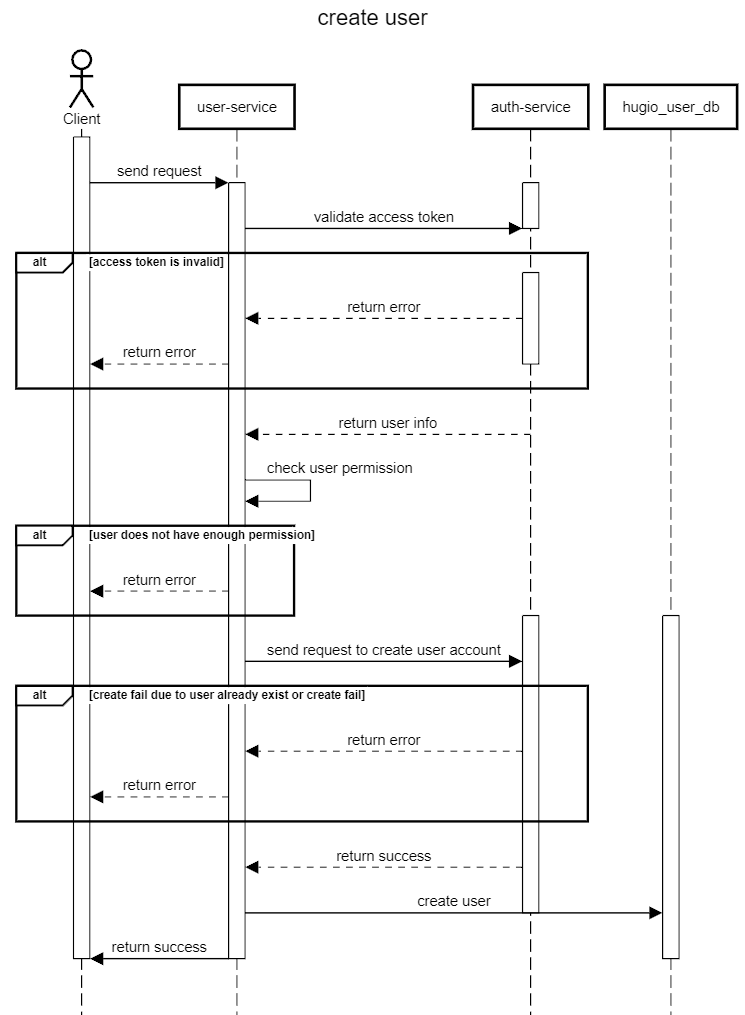


Figure 20 Create user sequence diagram

* Delete/active user

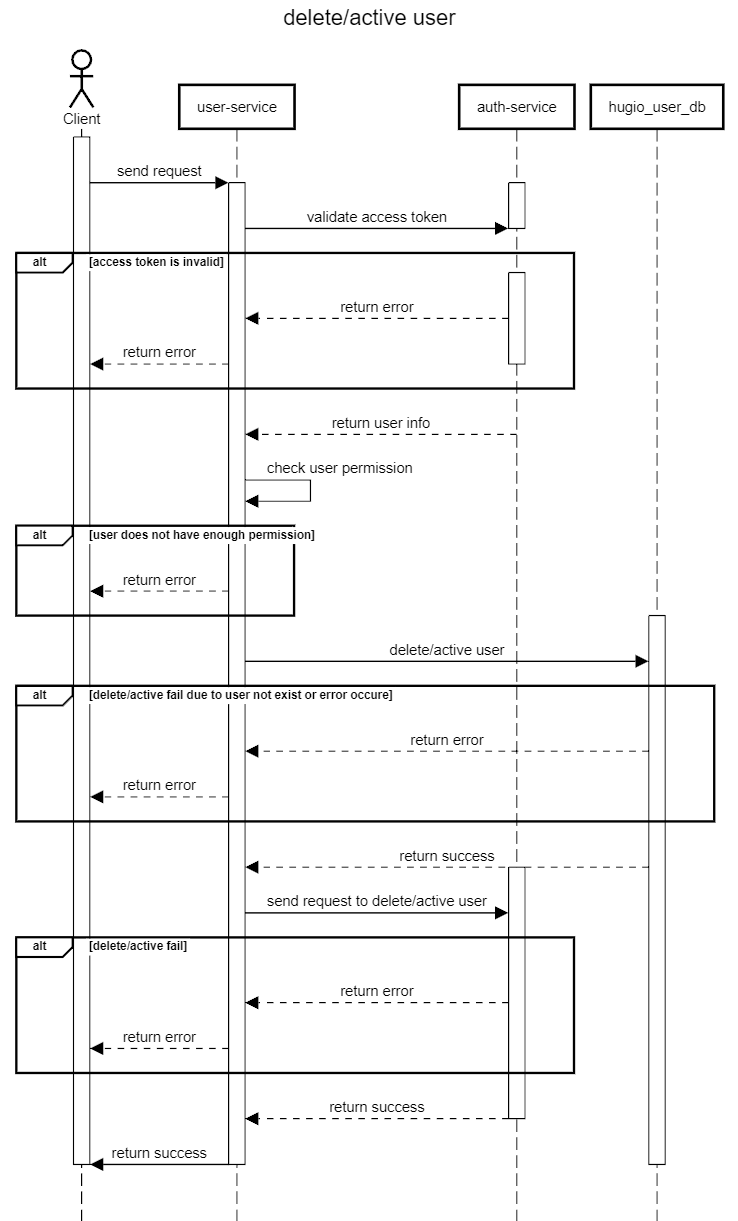


Figure 21 Delete/active user sequence diagram

* Update user

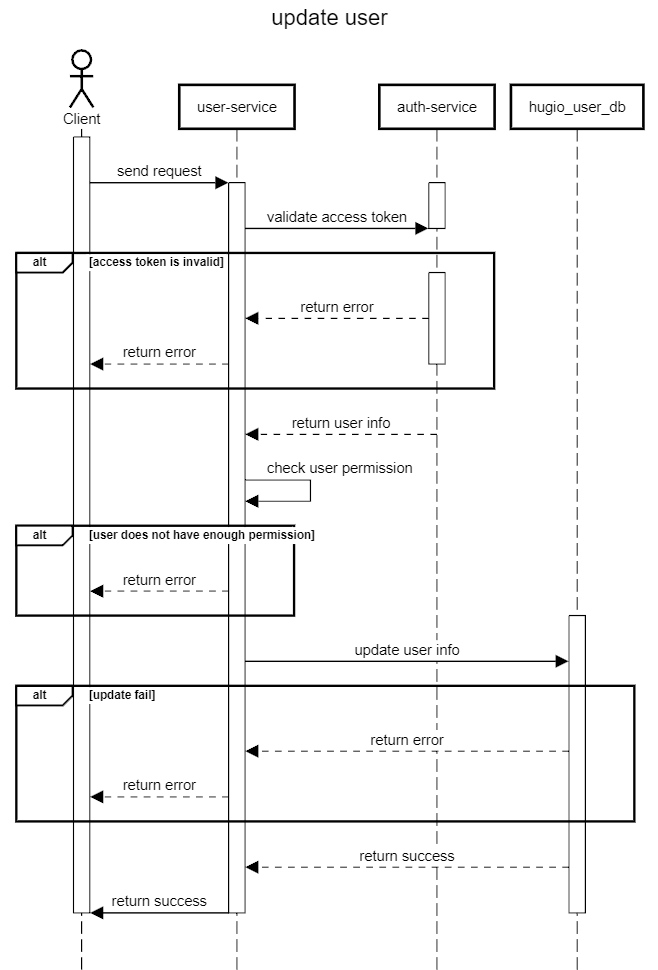


Figure 22 Update user sequence diagram

##### ***3.2.1.2.2 Database***

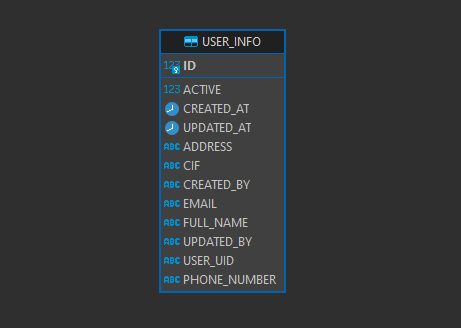


Figure 23 User serivce Database

The **USER\_INFO** is used for storing user information, including addresses, CIF values, email addresses, full names, phone numbers, and related user details. It might be associated with user profiles or customer information within the application. This table contain the following information:

* *ACTIVE*: This is a boolean column (BIT data type) that stores whether the user's information is active (1) or not (0).
* *CREATED\_AT*: This column of type datetime(6) records the timestamp when the user's information was created.
* *ID*: This is the primary key of the table, of type bigint, and it auto-increments with each new user information record.
* *UPDATED\_AT*: Similar to CREATED\_AT, this column of type datetime(6) records the timestamp when the user's information was last updated.
* *ADDRESS*: This varchar(255) column stores the user's address and is marked as NOT NULL.
* *CIF*: This varchar(255) column stores a CIF (Customer Information File) value and is marked as NOT NULL.
* *CREATED\_BY*: This is a varchar(255) column that might store information about the user who created the user's information. It has a default value of NULL.
* *EMAIL*: This varchar(255) column stores the user's email address and has a default value of NULL.
* *FULL\_NAME*: This varchar(255) column stores the user's full name and is marked as NOT NULL.
* *UPDATED\_BY*: Similar to CREATED\_BY, this column might store information about the user who last updated the user's information. It also has a default value of NULL.
* *USER\_UID*: This varchar(255) column stores a user UID (user identifier) and is marked as NOT NULL.
* *PHONE\_NUMBER*: This varchar(255) column stores the user's phone number and is marked as NOT NULL.
* The ID column serves as the primary key for the table.

***3.2.1.3 Product Service***

##### ***3.2.1.3.1 Flow***

This service contain the following function:

* Get all product
* Create product
* Update product
* Delete product
* Get all product

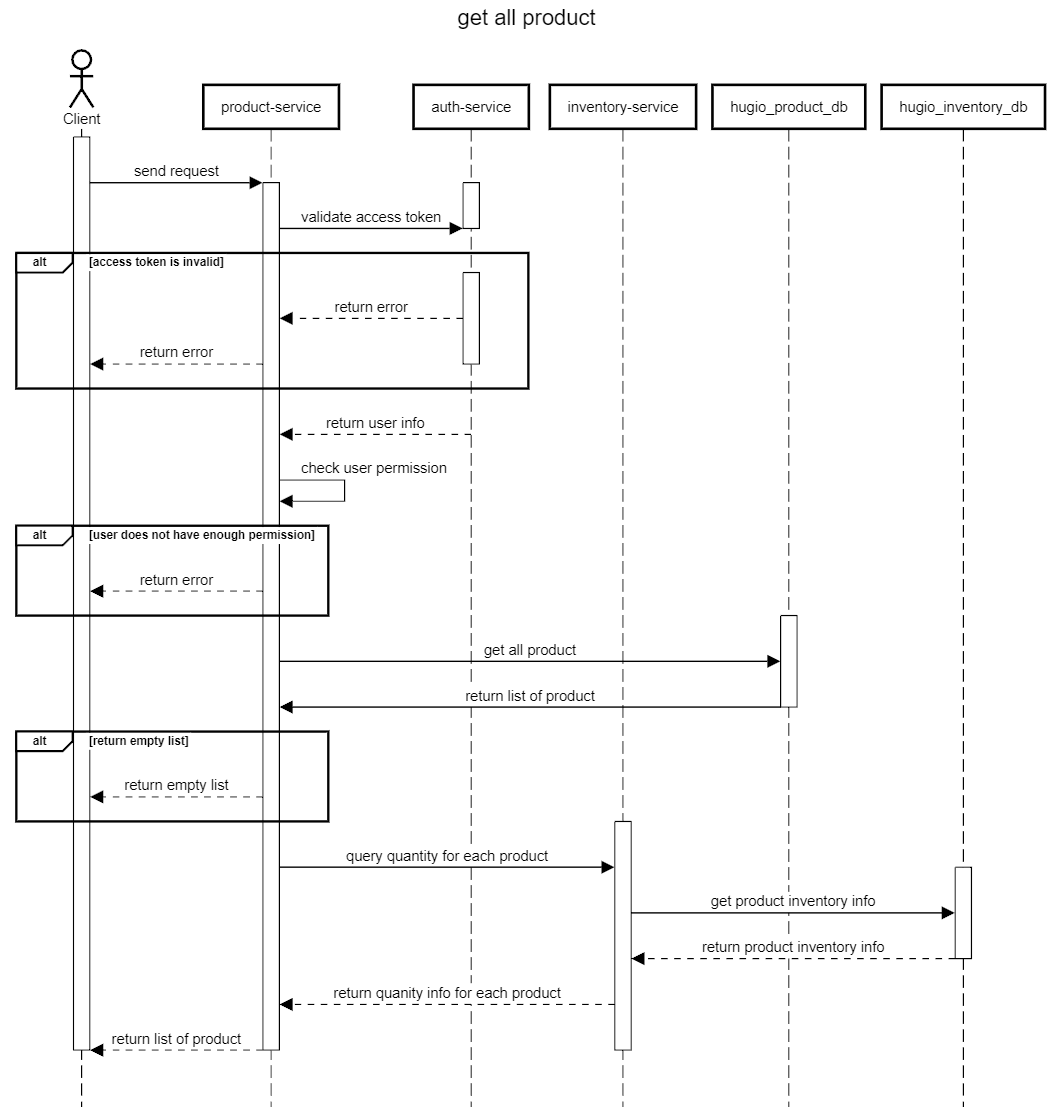


Figure 24 Get all product sequence diagram

* Create product

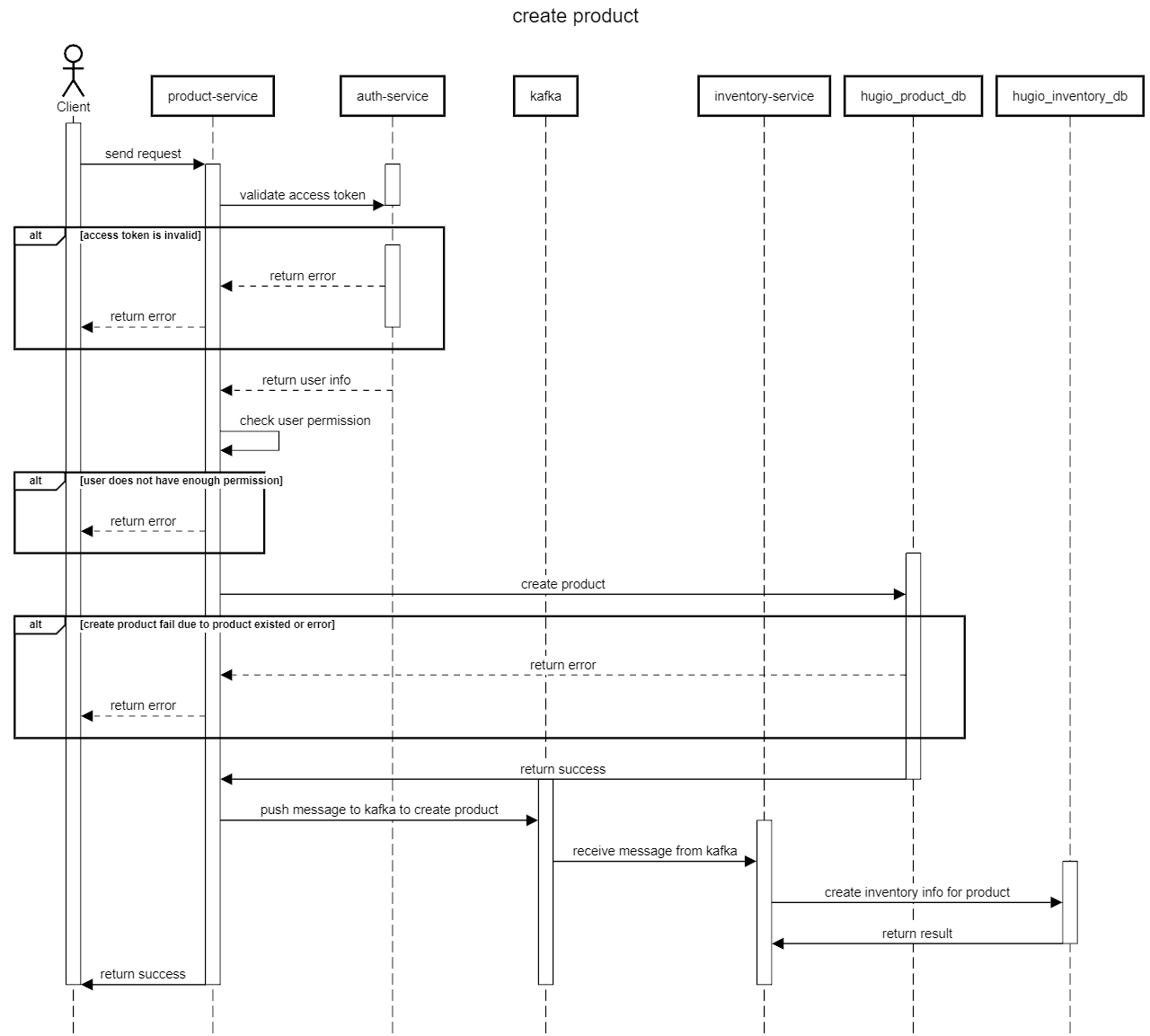


Figure 25 Create product sequence diagram

* Update product

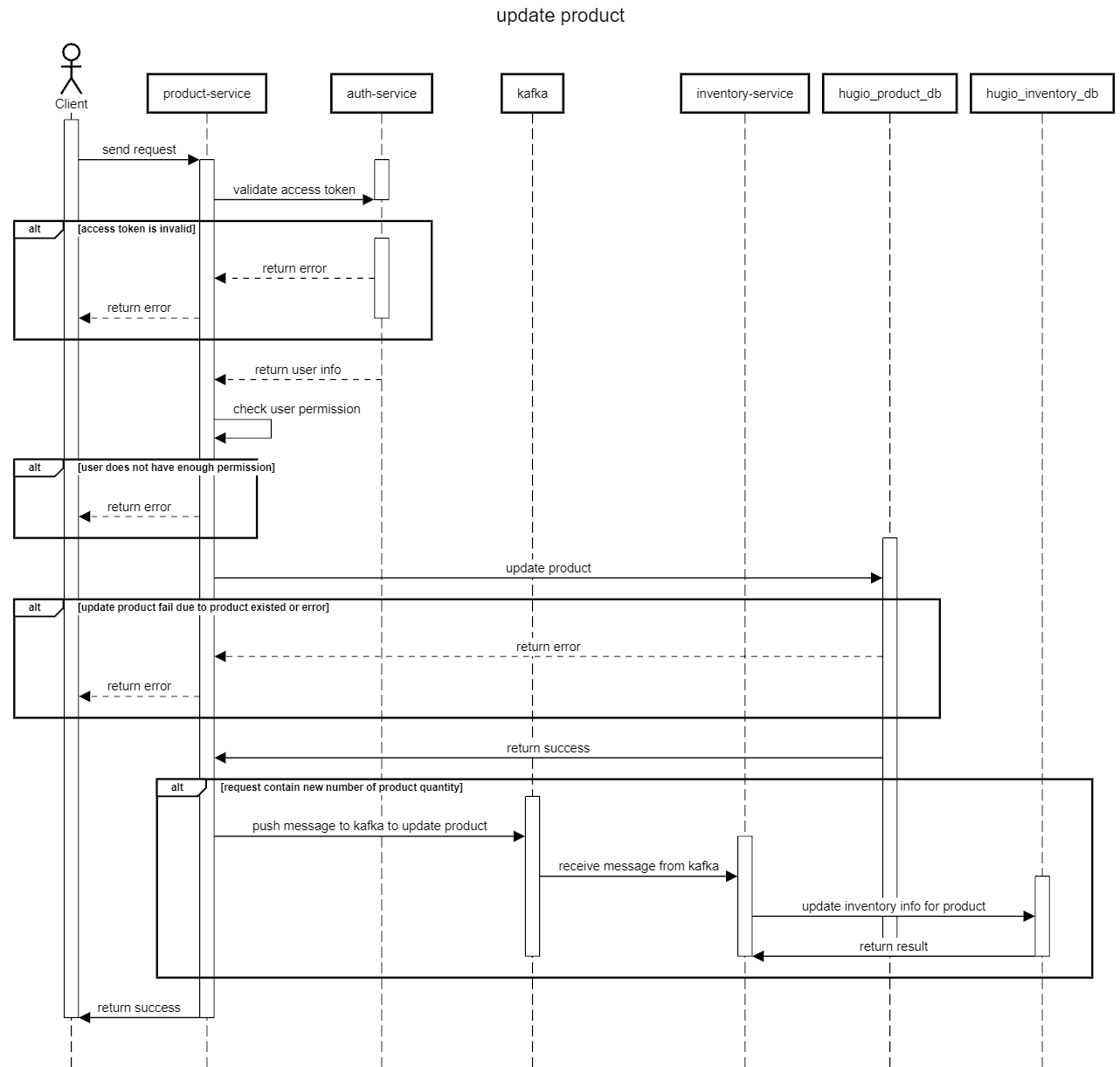


Figure 26 Update product sequence diagram

* Delete product

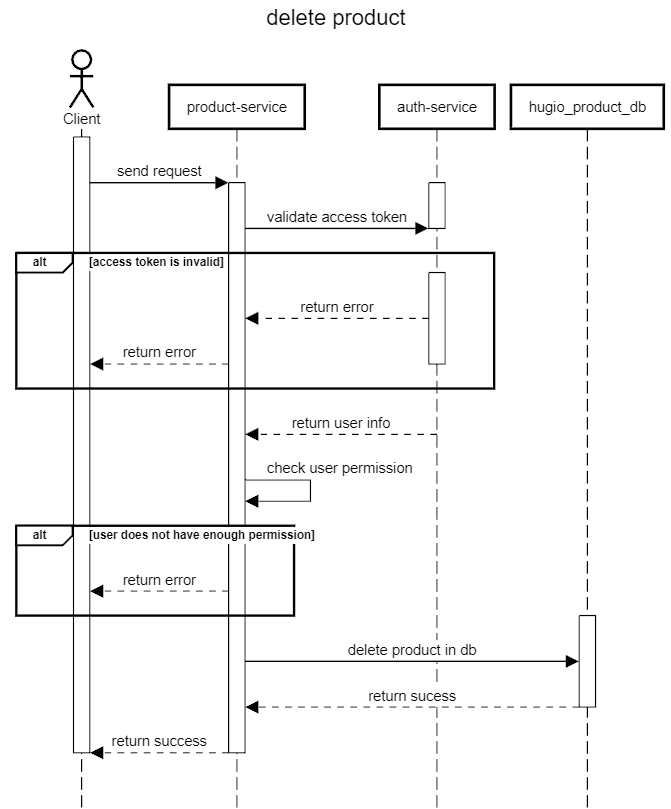


Figure 27 Delete product sequence diagram

##### ***3.2.1.3.2 Database***

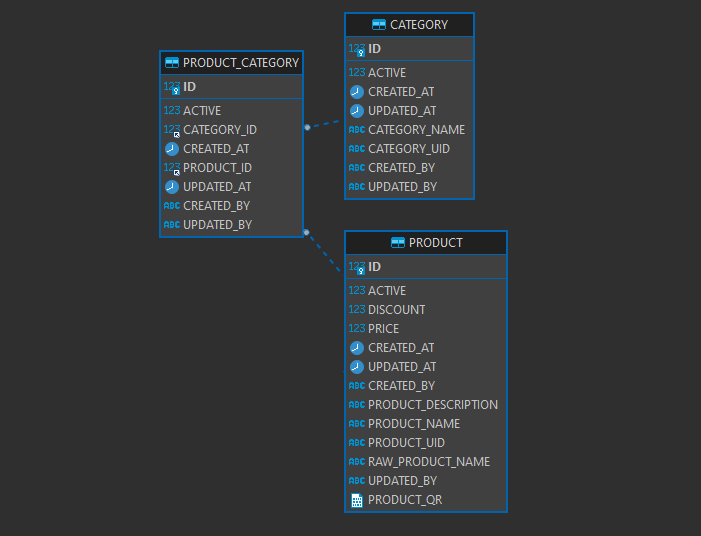


Figure 28 Product Database

The **PRODUCT** table is used for storing product information, including product names, descriptions, prices, discounts, and related details. The presence of a QR code column suggests that this table may be related to product labeling or identification within the application. This table contain the following information:

* *ACTIVE*: This is a boolean column (BIT data type) that stores whether the product is active (1) or not (0).
* *DISCOUNT*: This double-precision column stores the discount value for the product and has a default value of NULL.
* *PRICE*: This double-precision column stores the price of the product and has a default value of NULL.
* *CREATED\_AT*: This column of type datetime(6) records the timestamp when the product was created.
* *ID*: This is the primary key of the table, of type bigint, and it auto-increments with each new product record.
* *UPDATED\_AT*: Similar to CREATED\_AT, this column of type datetime(6) records the timestamp when the product was last updated.
* *CREATED\_BY*: This is a varchar(255) column that might store information about the user who created the product. It has a default value of NULL.
* *PRODUCT\_DESCRIPTION*: This varchar(255) column stores the description of the product and has a default value of NULL.
* *PRODUCT\_NAME*: This varchar(255) column stores the name of the product and has a default value of NULL.
* *PRODUCT\_UID*: This varchar(255) column stores a unique identifier for the product and is marked as NOT NULL.
* *RAW\_PRODUCT\_NAME*: This varchar(255) column stores the raw name of the product and has a default value of NULL.
* *UPDATED\_BY*: Similar to CREATED\_BY, this column might store information about the user who last updated the product. It also has a default value of NULL.
* *PRODUCT\_QR*: This longblob column likely stores binary data related to the product, possibly a QR code image.
* The ID column serves as the primary key for the table.

The **CATEGORY** is used to store information about product categories. The auto-incrementing ID field serves as a unique identifier for each category, and additional columns store details about the category, including its name, UID, and timestamps for creation and updates. This table contain the following information:

* *ACTIVE*: This is a boolean column (BIT data type) that indicates whether a category is active (1) or not (0).
* *CREATED\_AT*: This datetime(6) column records the timestamp when the category was created. It is marked as NOT NULL.
* *ID*: This is the primary key of the table, of type bigint, and it auto-increments with each new category.
* *UPDATED\_AT*: Similar to CREATED\_AT, this datetime(6) column records the timestamp when the category was last updated. It is marked as NOT NULL.
* *CATEGORY\_NAME*: This varchar(255) column stores the name of the category. It is marked as NOT NULL.
* *CATEGORY\_UID*: This varchar(255) column stores a unique identifier for the category. It is marked as NOT NULL.
* *CREATED\_BY*: This is a varchar(255) column that might store information about the user who created the category. It has a default value of NULL.
* *UPDATED\_BY*: Similar to CREATED\_BY, this column might store information about the user who last updated the category. It also has a default value of NULL.
* The ID column serves as the primary key for the table.

The **PRODUCT\_CATEGORY** is used to establish relationships between products and categories in a product catalog system. The table stores information about which products belong to which categories, and it enforces referential integrity through foreign key constraints. This table contain the following information:

* *ACTIVE*: This is a boolean column (BIT data type) that indicates whether a product-category relationship is active (1) or not (0).
* *CATEGORY\_ID*: This bigint column stores the ID of the associated product category. It is marked as NOT NULL.
* *CREATED\_AT*: This datetime(6) column records the timestamp when the product-category relationship was created. It is marked as NOT NULL.
* *ID*: This is the primary key of the table, of type bigint, and it auto-increments with each new relationship.
* *PRODUCT\_ID*: This bigint column stores the ID of the associated product. It is marked as NOT NULL.
* *UPDATED\_AT*: Similar to CREATED\_AT, this datetime(6) column records the timestamp when the product-category relationship was last updated. It is marked as NOT NULL.
* *CREATED\_BY*: This is a varchar(255) column that might store information about the user who created the relationship. It has a default value of NULL.
* *UPDATED\_BY*: Similar to CREATED\_BY, this column might store information about the user who last updated the relationship. It also has a default value of NULL.
* The ID column serves as the primary key for the table.

#### ***3.2.1.4 Inventory Service***

##### ***3.2.1.4.1 Database***

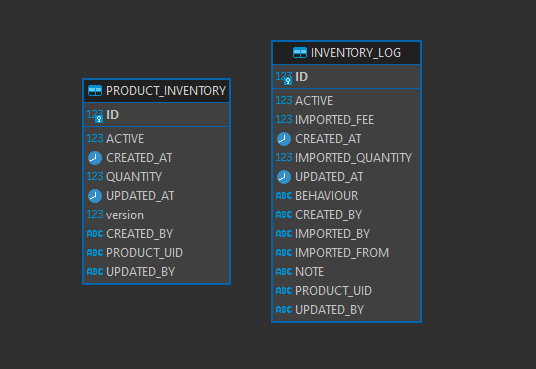


Figure 29 Inventory Database

The **PRODUCT\_INVENTORY** is used to store information about the quantity of products available in inventory. The table includes timestamps for when records were created and updated, as well as optional fields for tracking the creator and updater of records, the version of the record, and a unique identifier for the associated product. This table contain the following information:

* *ACTIVE*: This is a boolean column (BIT data type) that indicates whether a product inventory record is active (1) or not (0).
* *CREATED\_AT*: This datetime(6) column records the timestamp when the product inventory record was created. It is marked as NOT NULL.
* *ID*: This is the primary key of the table, of type bigint, and it auto-increments with each new record.
* *QUANTITY*: This bigint column stores the quantity of the product in the inventory. It has a default value of NULL, indicating that the quantity may not always be known.
* *UPDATED\_AT*: Similar to CREATED\_AT, this datetime(6) column records the timestamp when the product inventory record was last updated. It is marked as NOT NULL.
* *VERSION*: This bigint column is used to track the version of the product inventory record. It has a default value of NULL.
* *CREATED\_BY*: This is a varchar(255) column that might store information about the user or process that created the inventory record. It has a default value of NULL.
* *PRODUCT\_UID*: This is a varchar(255) column that might store a unique identifier for the product associated with this inventory record. It has a default value of NULL.
* *UPDATED\_BY*: Similar to CREATED\_BY, this column might store information about the user or process that last updated the inventory record. It also has a default value of NULL.
* The ID column serves as the primary key for the table.

The **INVENTORY\_LOG** is used to log various inventory-related activities, including creations, imports, and updates. It captures details such as fees, quantities, behavior, and associated product identifiers. This table contain the following information:

* *ACTIVE*: This is a boolean column (BIT data type) that indicates whether an inventory log record is active (1) or not (0).
* *IMPORTED\_FEE*: This double column stores a fee associated with the inventory log. It has a default value of NULL, indicating that the fee may not always be known.
* *CREATED\_AT*: This datetime(6) column records the timestamp when the inventory log record was created. It is marked as NOT NULL.
* *ID*: This is the primary key of the table, of type bigint, and it auto-increments with each new record.
* *IMPORTED\_QUANTITY*: This bigint column stores the quantity imported as part of the inventory log. It has a default value of NULL, indicating that the quantity may not always be known.
* *UPDATED\_AT*: Similar to CREATED\_AT, this datetime(6) column records the timestamp when the inventory log record was last updated. It is marked as NOT NULL.
* *BEHAVIOUR*: This enum column stores one of three values ('CREATE', 'IMPORT', or 'UPDATE') to describe the behavior associated with the inventory log entry. It is marked as NOT NULL.
* *CREATED\_BY*: This is a varchar(255) column that might store information about the user or process that created the inventory log record. It has a default value of NULL.
* *IMPORTED\_BY*: This is a varchar(255) column that might store information about the user or process that imported the inventory. It has a default value of NULL.
* *IMPORTED\_FROM*: This is a varchar(255) column that might store information about the source from which the inventory was imported. It has a default value of NULL.
* *NOTE*: This is a varchar(255) column that can store additional notes or comments related to the inventory log entry. It has a default value of NULL.
* *PRODUCT\_UID*: This is a varchar(255) column that might store a unique identifier for the product associated with this inventory log entry. It has a default value of NULL.
* *UPDATED\_BY*: Similar to CREATED\_BY, this column might store information about the user or process that last updated the inventory log record. It also has a default value of NULL.
* The ID column serves as the primary key for the table.

#### ***3.2.1.5 Order Service***

##### ***3.2.1.5.1 Flow***

This service contain the following function:

* Place order

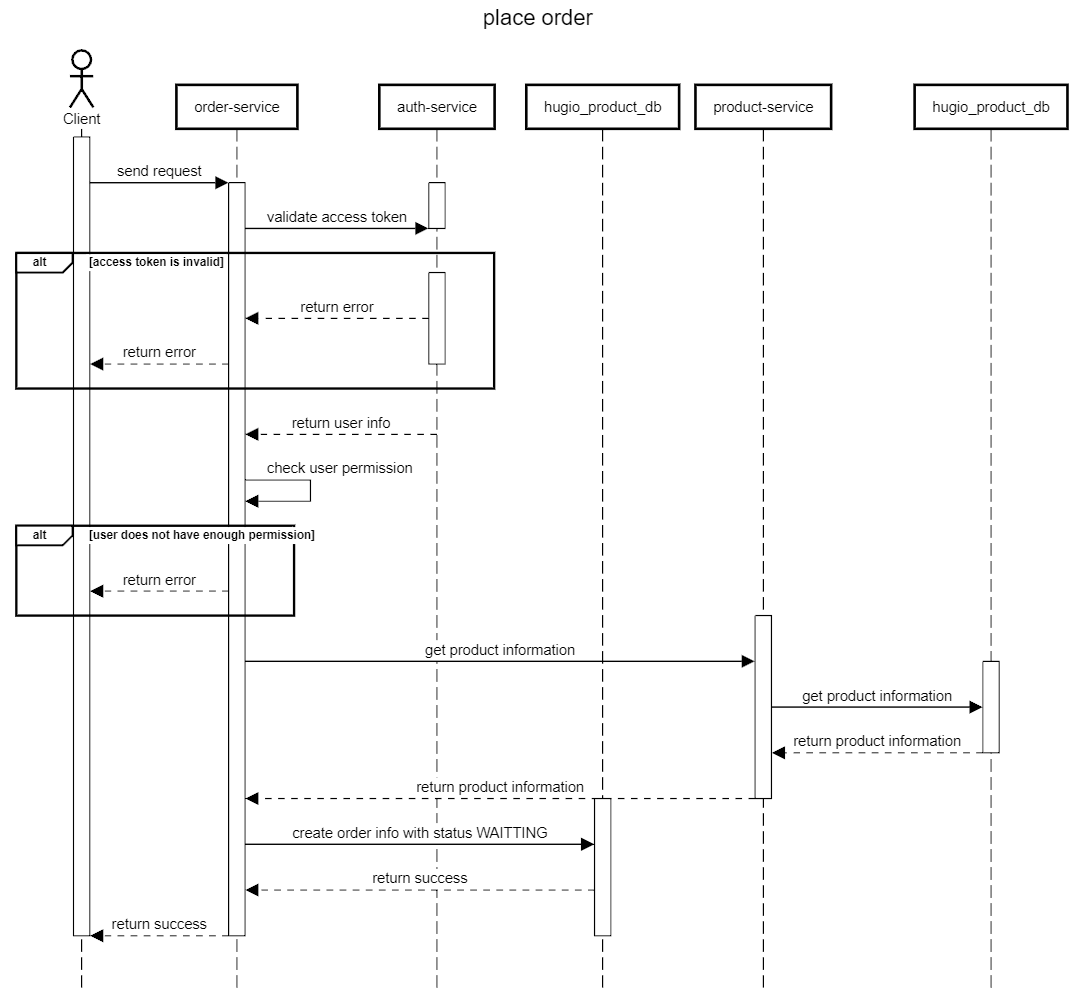


Figure 30 Place order sequence diagram

* Confirm order

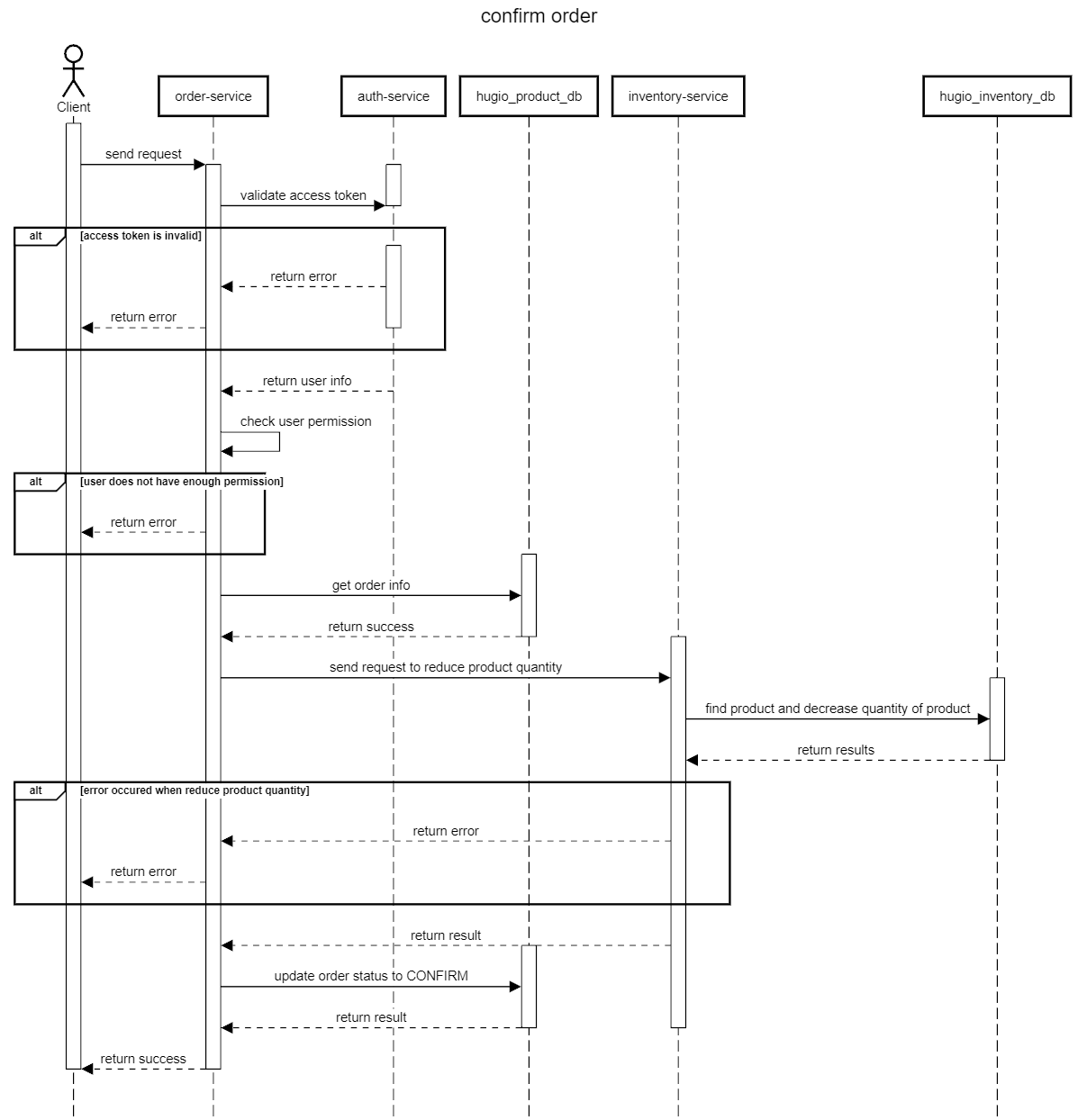


Figure 31 Confirm order sequence diagram

* Cancel order

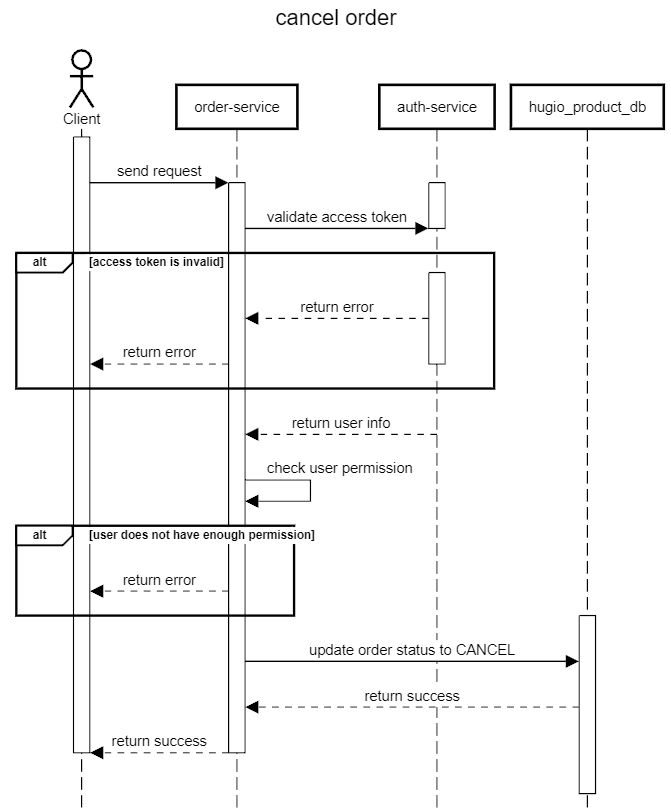


Figure 32 Cancel order sequence diagram

##### ***3.2.1.5.2 Database***

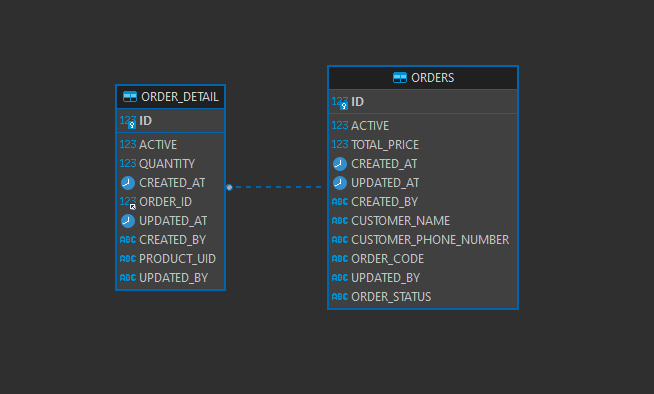


Figure 33 Order Database

The **ORDERS** is used to store information about orders, including details such as total price, customer information, order code, and status. The *ORDER\_STATUS* column indicates the current status of each order. This table contain the following information:

* *ACTIVE*: This is a boolean column (BIT data type) that indicates whether an order is active (1) or not (0).
* *TOTAL\_PRICE*: This double column stores the total price associated with the order. It has a default value of NULL, indicating that the total price may not always be known.
* *CREATED\_AT*: This datetime(6) column records the timestamp when the order record was created. It is marked as NOT NULL.
* *ID*: This is the primary key of the table, of type bigint, and it auto-increments with each new record.
* *UPDATED\_AT*: Similar to CREATED\_AT, this datetime(6) column records the timestamp when the order record was last updated. It is marked as NOT NULL.
* *CREATED\_BY*: This is a varchar(255) column that might store information about the user or process that created the order record. It has a default value of NULL.
* *CUSTOMER\_NAME*: This is a varchar(255) column that might store the name of the customer associated with the order. It has a default value of NULL.
* *CUSTOMER\_PHONE\_NUMBER*: This is a varchar(255) column that might store the phone number of the customer. It has a default value of NULL.
* *ORDER\_CODE*: This is a varchar(255) column that might store a code or identifier associated with the order. It has a default value of NULL.
* *UPDATED\_BY*: Similar to CREATED\_BY, this column might store information about the user or process that last updated the order record. It also has a default value of NULL.
* *ORDER\_STATUS*: This enum column stores one of three values ('CANCELED', 'DONE', or 'PENDING') to indicate the status of the order. It has a default value of NULL.
* The ID column serves as the primary key for the table.

The **ORDER\_DETAIL** is used to store details of products within orders, including the quantity ordered, associated order, and product identifier. The ACTIVE column can be used to mark order details as active or inactive. The foreign key ORDER\_ID links each order detail to a specific order using the ORDERS table's primary key (ID). This table contain the following information:

* *ACTIVE*: This is a boolean column (BIT data type) that indicates whether an order detail is active (1) or not (0).
* *QUANTITY*: This int column stores the quantity of a product ordered. It has a default value of NULL, indicating that the quantity may not always be known.
* *CREATED\_AT*: This datetime(6) column records the timestamp when the order detail record was created. It is marked as NOT NULL.
* *ID*: This is the primary key of the table, of type bigint, and it auto-increments with each new record.
* *ORDER\_ID*: This bigint column is a foreign key that references the ID column in the ORDERS table. It is used to associate an order detail with a specific order.
* *UPDATED\_AT*: Similar to CREATED\_AT, this datetime(6) column records the timestamp when the order detail record was last updated. It is marked as NOT NULL.
* *CREATED\_BY*: This is a varchar(255) column that might store information about the user or process that created the order detail record. It has a default value of NULL.
* *PRODUCT\_UID*: This is a varchar(255) column that might store a unique identifier associated with the product in the order detail. It has a default value of NULL.
* *UPDATED\_BY*: Similar to CREATED\_BY, this column might store information about the user or process that last updated the order detail record. It also has a default value of NULL.
* The ID column serves as the primary key for the table.

### How service communicate?

The services will communicate with each other via gRPC using the HTTP 2.0 protocol.

Using gRPC for communication offers several benefits:

* *High Performance*: gRPC utilizes HTTP/2, optimizing data transmission and reducing latency. It enables multiplexing over a single connection, minimizing the overhead of sending requests and responses. This accelerates data transfer speed and enhances system performance.
* *Language Agnostic*: gRPC supports multiple programming languages. You can write gRPC services in various languages and interconnect them seamlessly. This simplifies development and integration of services.
* *Interface Definition*: gRPC uses Protocol Buffers (protobufs) to define service interfaces. This results in automatic documentation and code generation for clients and servers, ensuring consistency and minimizing interface definition errors.
* *Bidirectional Streaming*: gRPC supports bidirectional streaming, allowing both clients and servers to continuously send and receive data over the same connection. This is suitable for real-time data transmission and interactive scenarios.
* *Built-in Security*: gRPC provides built-in security mechanisms, including TLS/SSL, to ensure data security during transmission.
* *Service Integration and Health Checking*: gRPC offers built-in service integration and health checking (gRPC Health Check), facilitating service monitoring and easy troubleshooting.

When a product is added or modified within the system, the Product Service will send a message to the Inventory Service via Kafka using a specific topic to perform the functionality of adding or modifying the product quantity.

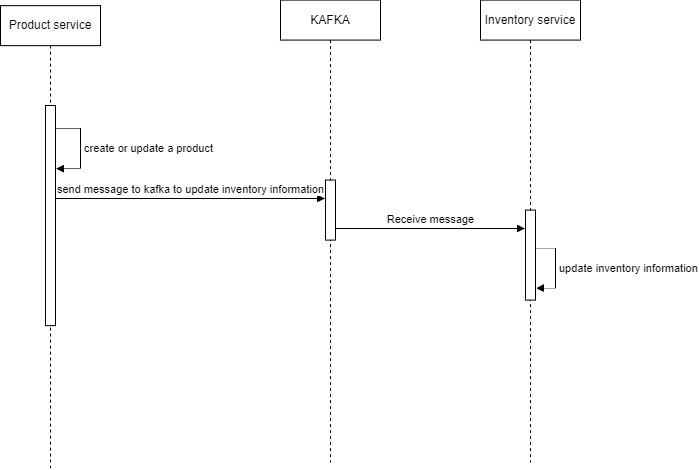


Figure 34 A flow show how HUGIO using Kafka

Redis is also employed within the system to store the entire product list, serving the purpose of enhancing API performance.

### Applying ChatGPT's api to product sales statistics

In order to facilitate accurate product statistics, our project has integrated the API of ChatGPT, an artificial intelligence language model developed by OpenAI. Built on the GPT-3.5 platform, ChatGPT has the capability to answer questions and provide support in delivering information and guidance on various topics.

At the end of each day, at 10 PM, the software compiles sales information for that day, such as the number of orders and the quantity of products sold. It then composes and sends questions to ChatGPT requesting it to summarize the sales data for the day. Subsequently, the answers are stored in the database.

Similarly, at the end of each month, at 10 PM, a question is sent to request the assessment of sales prospects for the upcoming month, aiding the store owner in making decisions based on the monthly sales data.

# DEPLOYMENT, MANAGEMENT AND MONITORING

## 4.1 Introduction

In the rapidly evolving landscape of software development, efficient deployment and management processes have become crucial components of a successful project. As the demand for faster and more reliable software delivery grows, the adoption of robust tools and methodologies becomes paramount. This report focuses on the deployment and management strategy employed in the project, leveraging a combination of GitHub for version control, Jenkins for Continuous Integration and Continuous Deployment (CI/CD), Docker for containerization, Kubernetes with Helm for container orchestration, and Nginx as a web server and reverse proxy.

### 4.1.1 Importance of efficient deployment and management processes

In today's fast-paced software development landscape, where rapid iterations and frequent updates are the norm, efficient deployment and management practices are imperative. Effective deployment ensures that software changes are delivered consistently and reliably to the production environment, minimizing downtimes and user disruptions. Efficient management guarantees that the deployed application performs optimally, scales seamlessly, and remains secure.

### 4.1.2 Tool Selection

1. GitHub: A widely used version control platform that enables collaborative development, version tracking, and streamlined code reviews.
2. Jenkins: A robust CI/CD tool that automates the build, testing, and deployment processes, ensuring code quality and accelerating software delivery.
3. Docker: A containerization platform that encapsulates applications and their dependencies in isolated containers, ensuring consistent behavior across various environments.
4. Kubernetes: A powerful container orchestration platform that automates deployment, scaling, and management of containerized applications.
5. Helm: A package manager for Kubernetes that simplifies the deployment and management of applications by providing reusable and customizable charts.
6. Nginx: A high-performance web server and reverse proxy that enhances application performance, load balancing, and security.
7. Splunk: Splunk is a powerful and widely used software platform that specializes in data analytics, particularly in the realm of log and machine data. It is designed to help organizations gain insights, monitor, and analyze a wide range of data sources in real-time. In this project, we use Splunk for manage system logs.

## 4.2 Version Control and Collaboration with GitHub

GitHub serves as a distributed version control platform that facilitates efficient code management, collaboration, and tracking changes over time. Its integration with Git, a distributed version control system, empowers development teams to work seamlessly together, regardless of geographic locations.

In the Hugio project, we will have two projects regularly saved and updated on GitHub.

The first one is for the Front End:

Link: <https://github.com/ducminhle1904/hugio-front-end>

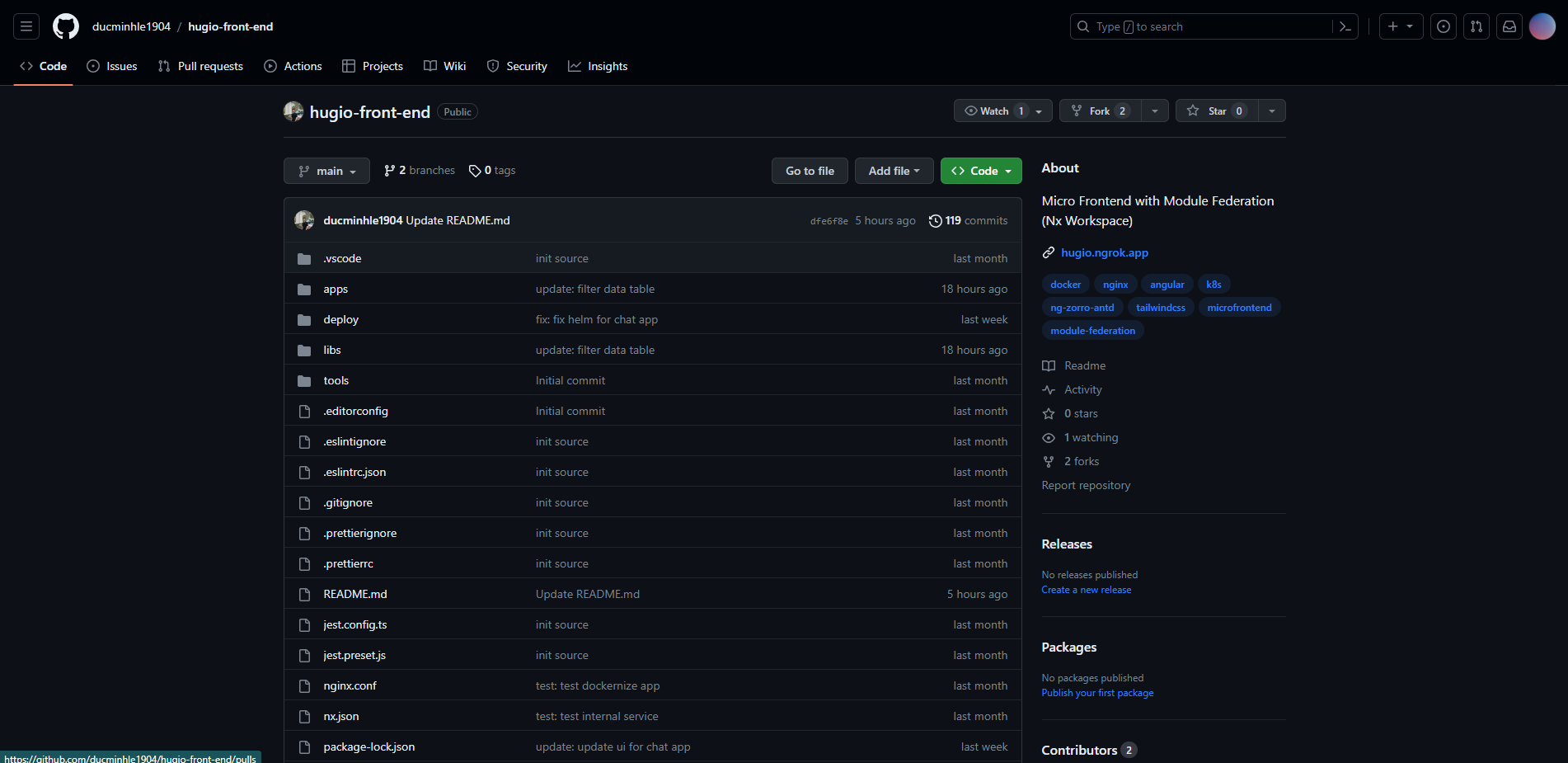


Figure 35 Hugio front end github page

The second one is for the Back End:

Link: <https://github.com/tuanloc1105/hugio-back-end>

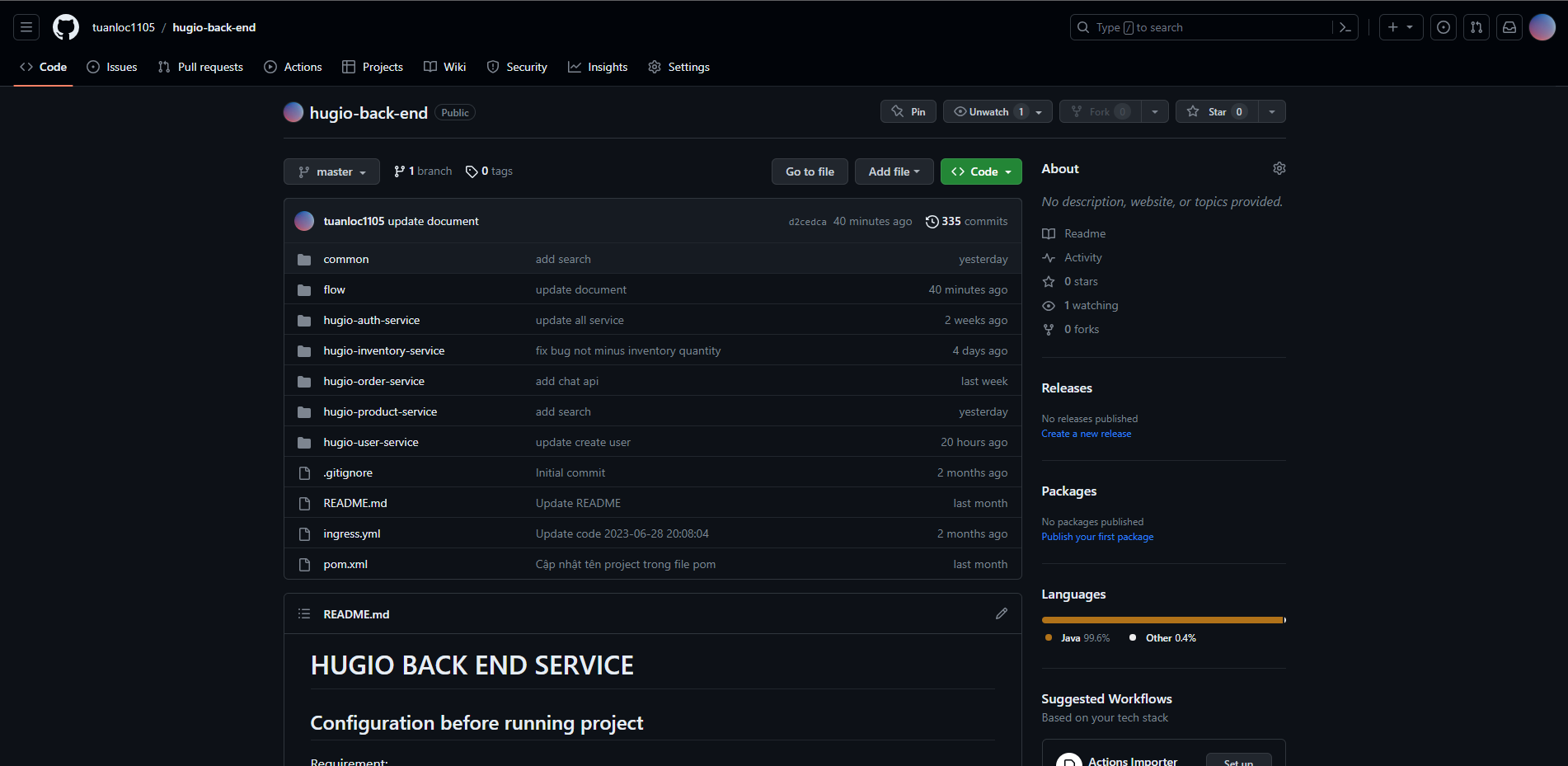


Figure 36 Hugio back end github page

GitHub allows developers to track changes at a granular level, providing a historical record of code modifications, additions, and deletions. This version tracking is crucial for understanding the evolution of the codebase and reverting to previous versions if needed.

## 4.3 Continuous Integration and Continuous Deployment (CI/CD) with Jenkins

In the realm of modern software development, the adoption of Continuous Integration (CI) and Continuous Deployment (CD) practices has become pivotal for delivering high-quality software efficiently and reliably. In this section, we delve into the utilization of Jenkins as the CI/CD tool to automate build, testing, and deployment processes, ensuring code integrity and expediting the delivery pipeline.

### 4.3.1 Jenkins in Hugio project

In the Hugio project, we utilize Jenkins to automate the deployment of both our backend and frontend services.

Jenkins plays a pivotal role in our continuous integration and continuous deployment (CI/CD) pipeline. When code changes are pushed to our respective GitHub repositories for the backend and frontend, Jenkins is configured to automatically trigger a series of predefined steps:

1. *Build*: Jenkins initiates the build process for the backend and frontend codebases. This step involves compiling, bundling, and preparing the application for deployment.
2. *Deployment*: Depending on the branch and criteria specified in our Jenkins pipeline, the application is deployed to the appropriate environment, be it a staging environment for testing or directly to production if the changes pass all required checks.
3. *Monitoring*: Jenkins continuously monitors the health and performance of our deployed services, providing real-time feedback and notifications in case of issues.

### 4.3.2 CI/CD Pipeline Stages

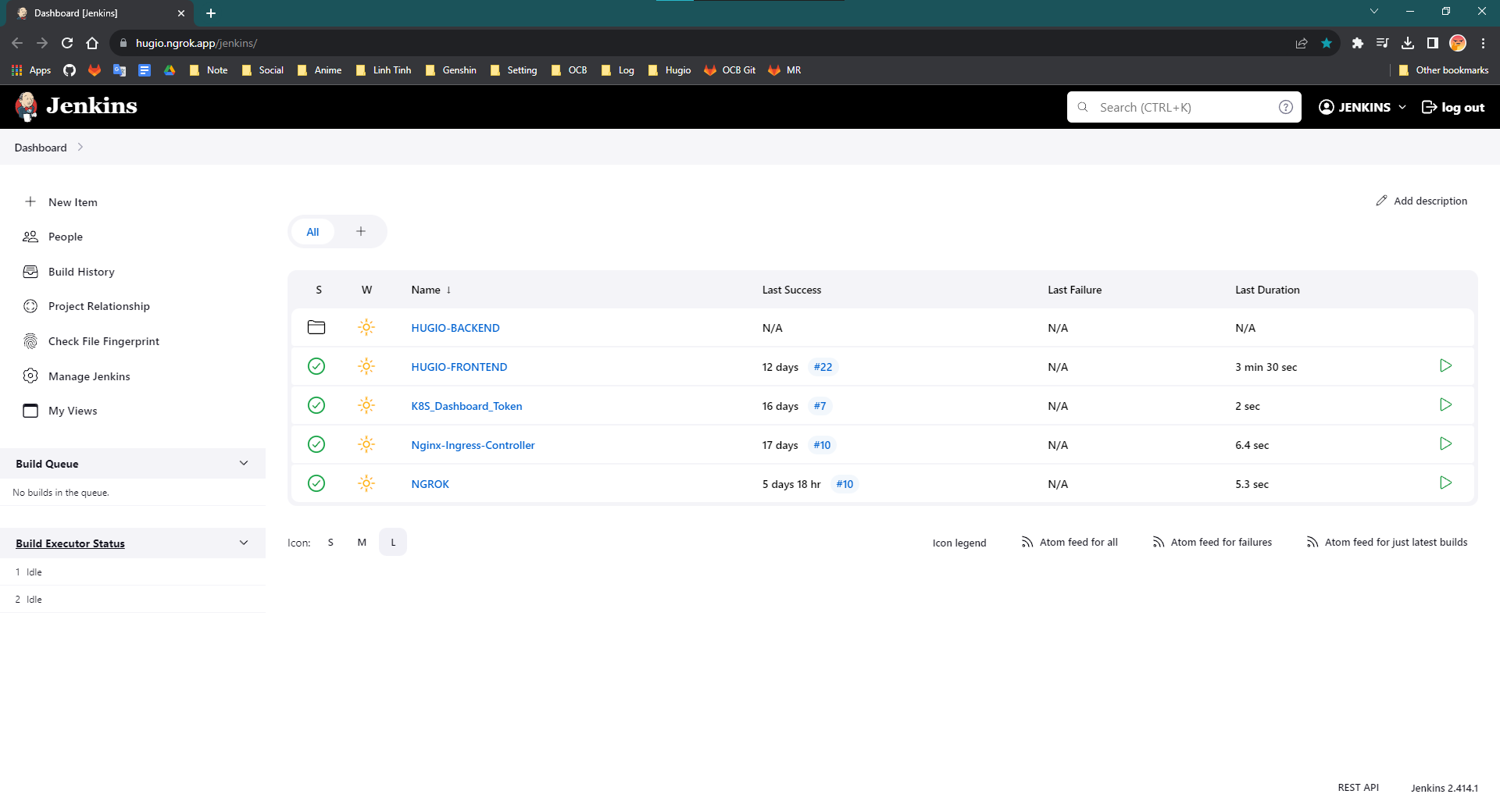


Figure 37 Hugio's jenkins dashboard

Here is the Jenkins project homepage of our project. Where:

* + The **HUGIO-BACKEND** folder is used to build 5 backend services.
  + The **HUGIO-FRONTEND** pipeline is used to build the frontend service.

Starting with the frontend pipeline, it will be as follows:

def userIdCause = currentBuild.getBuildCauses('hudson.model.Cause$UserIdCause')

def now = (new Date()).format("yyMMddHHmmss", TimeZone.getTimeZone('Asia/Ho\_Chi\_Minh'))

def dockerTag='ol'

def appShellDockerImage='hugio/app\_shell'

def appAnalysisDockerImage='hugio/app\_analysis'

def appAuthDockerImage='hugio/app\_auth'

def appCashbookDockerImage='hugio/app\_cashbook'

def appProductDockerImage='hugio/app\_product'

def appSummaryDockerImage='hugio/app\_summary'

def appUserDockerImage='hugio/app\_user'

def appOrderDockerImage='hugio/app\_order'

def appChatDockerImage='hugio/app\_chat'

def k8sNamespace='frontend'

def k8sReplica=1

pipeline {

    agent any

    tools {

        nodejs "node16"

    }

    environment {

        GIT\_CREDENTIALS\_ID = 'tuanloc-github'

        APPLICATION\_GIT\_BRANCH = 'main'

        APPLICATION\_GIT\_REPOSITRY = 'https://github.com/ducminhle1904/hugio-front-end.git'

        TARGET\_DIR = '/home/ae403/hugio/front-end'

    }

    stages {

        stage('Checkout source') {

            steps {

                script {

                    def checkoutDetails = checkout([

                        $class: 'GitSCM',

                        branches: [[ name: ('\*/' + APPLICATION\_GIT\_BRANCH) ]],

                        doGenerateSubmoduleConfigurations: false,

                        submoduleCfg: [],

                        userRemoteConfigs: [[ credentialsId: GIT\_CREDENTIALS\_ID, url: APPLICATION\_GIT\_REPOSITRY ]]

                    ])

                }

            }

        }

        stage('Build service') {

            steps {

                script {

                    CURRENT\_DATE = new java.text.SimpleDateFormat('yyyyMMdd').format(new Date())

                    CURRENT\_TIME = new java.text.SimpleDateFormat('HHmmss').format(new Date())

                    dockerTag = "$CURRENT\_DATE" + "$CURRENT\_TIME"

                    sh "npm install"

                    sh "npm run build"

                }

            }

        }

        stage('Copy source') {

            steps {

                script {

                    sh "scp -i /var/jenkins\_home/jenkins -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -r ./dist ae403@192.168.1.22:$TARGET\_DIR"

                    sh "scp -i /var/jenkins\_home/jenkins -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -r ./deploy ae403@192.168.1.22:$TARGET\_DIR"

                    sh "scp -i /var/jenkins\_home/jenkins -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no ./nginx.conf ae403@192.168.1.22:$TARGET\_DIR"

                }

            }

        }

        stage('Deploy source') {

            steps {

                script {

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker build . -t $appShellDockerImage:$dockerTag -f ./deploy/shell/Dockerfile\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker build . -t $appAnalysisDockerImage:$dockerTag -f ./deploy/analysis/Dockerfile\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker build . -t $appAuthDockerImage:$dockerTag -f ./deploy/auth/Dockerfile\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker build . -t $appCashbookDockerImage:$dockerTag -f ./deploy/cashbook/Dockerfile\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker build . -t $appProductDockerImage:$dockerTag -f ./deploy/product/Dockerfile\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker build . -t $appSummaryDockerImage:$dockerTag -f ./deploy/summary/Dockerfile\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker build . -t $appUserDockerImage:$dockerTag -f ./deploy/user/Dockerfile\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker build . -t $appOrderDockerImage:$dockerTag -f ./deploy/order/Dockerfile\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker build . -t $appChatDockerImage:$dockerTag -f ./deploy/chat/Dockerfile\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; kind load docker-image $appShellDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; kind load docker-image $appAnalysisDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; kind load docker-image $appAuthDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; kind load docker-image $appCashbookDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; kind load docker-image $appProductDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; kind load docker-image $appSummaryDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; kind load docker-image $appUserDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; kind load docker-image $appOrderDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; kind load docker-image $appChatDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; helm upgrade -i --set image.name=$appShellDockerImage,image.tag=$dockerTag,replica=$k8sReplica,tcp\_port=4200 -n $k8sNamespace app-shell ./deploy/shell/helm\_chart\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; helm upgrade -i --set image.name=$appAnalysisDockerImage,image.tag=$dockerTag,replica=$k8sReplica,tcp\_port=4202 -n $k8sNamespace app-analysis ./deploy/analysis/helm\_chart\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; helm upgrade -i --set image.name=$appAuthDockerImage,image.tag=$dockerTag,replica=$k8sReplica,tcp\_port=4206 -n $k8sNamespace app-auth ./deploy/auth/helm\_chart\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; helm upgrade -i --set image.name=$appCashbookDockerImage,image.tag=$dockerTag,replica=$k8sReplica,tcp\_port=4203 -n $k8sNamespace app-cashbook ./deploy/cashbook/helm\_chart\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; helm upgrade -i --set image.name=$appProductDockerImage,image.tag=$dockerTag,replica=$k8sReplica,tcp\_port=4205 -n $k8sNamespace app-product ./deploy/product/helm\_chart\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; helm upgrade -i --set image.name=$appSummaryDockerImage,image.tag=$dockerTag,replica=$k8sReplica,tcp\_port=4201 -n $k8sNamespace app-summary ./deploy/summary/helm\_chart\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; helm upgrade -i --set image.name=$appUserDockerImage,image.tag=$dockerTag,replica=$k8sReplica,tcp\_port=4204 -n $k8sNamespace app-user ./deploy/user/helm\_chart\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; helm upgrade -i --set image.name=$appOrderDockerImage,image.tag=$dockerTag,replica=$k8sReplica,tcp\_port=4207 -n $k8sNamespace app-order ./deploy/order/helm\_chart\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; helm upgrade -i --set image.name=$appChatDockerImage,image.tag=$dockerTag,replica=$k8sReplica,tcp\_port=4208 -n $k8sNamespace app-chat ./deploy/chat/helm\_chart\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker rmi $appShellDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker rmi $appAnalysisDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker rmi $appAuthDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker rmi $appCashbookDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker rmi $appProductDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker rmi $appSummaryDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker rmi $appUserDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker rmi $appOrderDockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker rmi $appChatDockerImage:$dockerTag\""

                }

            }

        }

    }

}

Let's look at each part of this pipeline.

1. Initialize Variables and Format Time:
   1. In this section, initialize several variables:
      1. *userIdCause*: This variable is used to store information about the user who triggered the build.
      2. *now*: This variable stores the current date and time in the format "yyMMddHHmmss" in the Asia/Ho\_Chi\_Minh time zone.
      3. Various other variables are declared for Docker image names and environment configurations.
2. Define Docker Images and Environment Variables:
   1. This part is where you define variables for Docker image names, such as appShellDockerImage, appAnalysisDockerImage, etc., and set the initial value for the dockerTag variable.
3. Start Pipeline:
   1. The pipeline block initiates the Jenkins Pipeline. It specifies that this pipeline can run on any available agent (agent any).
4. Set Up Node.js Tool:
   1. This stage ensures that the Node.js tool with version 16 is available for the pipeline. This is necessary for building the project.
5. Environment Variable Setup:
   1. Here, define environment variables used throughout the pipeline, including credentials for accessing a GitHub repository, the Git branch to be deployed (APPLICATION\_GIT\_BRANCH), and the target directory on the server (TARGET\_DIR).
6. Specific Stages:
   1. This pipeline is divided into several stages:
      1. Checkout source: This stage checks out the source code from a GitHub repository.
      2. Build service: It runs npm commands to build the service.
      3. Copy source: This stage copies the built application files and configuration to a target directory on a remote server using Secure Copy Protocol (SCP).
      4. Deploy source: This is the core deployment stage. It involves building Docker images for various components of the application, loading them into a local Kubernetes cluster using kind, and upgrading the Kubernetes deployments using Helm.
7. Remove Unused Docker Images:
   1. After the Docker images have been deployed, this section removes the Docker images that were created during the deployment process.

For the HUGIO-BACKEND folder, within this folder, there will be 5 pipelines to build 5 services as shown below.

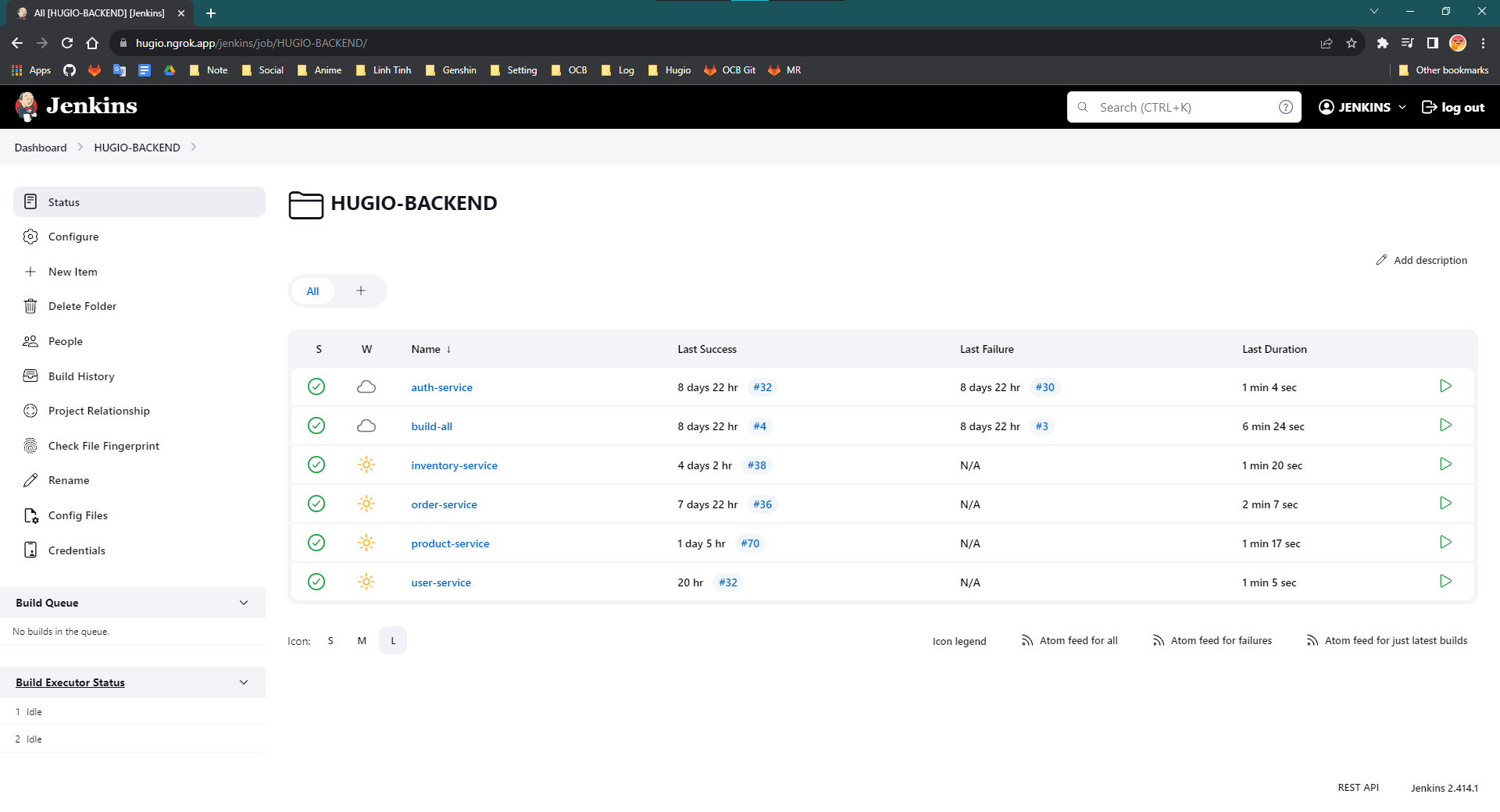


Figure 38 Hugio dashboard for back end

The pipeline scripts for the 5 services will be identical. I will use the pipeline for the *auth-service* as an example. The pipeline is written as follows:

def dockerImage = "tuanloc/auth-service"

def dockerTag = ""

def k8sReplica = 1

def k8sNamespace = 'hugio'

def userIdCause = currentBuild.getBuildCauses('hudson.model.Cause$UserIdCause')

def now = (new Date()).format("yyMMddHHmmss", TimeZone.getTimeZone('Asia/Ho\_Chi\_Minh'))

pipeline {

    agent any

    tools {

        maven "3.9.3"

        jdk "17"

    }

    environment {

        GIT\_CREDENTIALS\_ID = 'tuanloc-github'

        APPLICATION\_GIT\_BRANCH = 'master'

        APPLICATION\_GIT\_REPOSITRY = 'https://github.com/tuanloc1105/hugio-back-end.git'

        TARGET\_DIR = '/home/ae403/hugio/auth\_service'

    }

    stages {

        stage('Checkout source') {

            steps {

                script {

                    def checkoutDetails = checkout([

                        $class: 'GitSCM',

                        branches: [[ name: ('\*/' + APPLICATION\_GIT\_BRANCH) ]],

                        doGenerateSubmoduleConfigurations: false,

                        submoduleCfg: [],

                        userRemoteConfigs: [[ credentialsId: GIT\_CREDENTIALS\_ID, url: APPLICATION\_GIT\_REPOSITRY ]]

                    ])

                }

            }

        }

        stage('Build authenticate service') {

            steps {

                script {

                    CURRENT\_DATE = new java.text.SimpleDateFormat('yyyyMMdd').format(new Date())

                    CURRENT\_TIME = new java.text.SimpleDateFormat('HHmmss').format(new Date())

                    dir('common') {

                        dockerTag = "$CURRENT\_DATE" + "$CURRENT\_TIME"

                        sh 'mvn clean install -DskipTests=true -Dfile.encoding=UTF8 -f pom.xml'

                    }

                    dir('hugio-auth-service') {

                        sh 'mvn clean install -DskipTests=true -Dfile.encoding=UTF8 -f pom.xml'

                    }

                }

            }

        }

        stage('Copy source') {

            steps {

                script {

                    dir('hugio-auth-service') {

                        sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 mkdir -p $TARGET\_DIR"

                        sh "scp -i /var/jenkins\_home/jenkins -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -r ./helm\_chart ae403@192.168.1.22:$TARGET\_DIR"

                        sh "scp -i /var/jenkins\_home/jenkins -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no -r ./target ae403@192.168.1.22:$TARGET\_DIR"

                        sh "scp -i /var/jenkins\_home/jenkins -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no ./Dockerfile ae403@192.168.1.22:$TARGET\_DIR"

                        sh "scp -i /var/jenkins\_home/jenkins -o UserKnownHostsFile=/dev/null -o StrictHostKeyChecking=no ./logback.xml ae403@192.168.1.22:$TARGET\_DIR"

                    }

                }

            }

        }

        stage('Deploy source') {

            steps {

                script {

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker build . -t $dockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; kind load docker-image $dockerImage:$dockerTag\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; helm upgrade -i --set image.name=$dockerImage,image.tag=$dockerTag,replica=$k8sReplica -n $k8sNamespace auth-service ./helm\_chart\""

                    sh "ssh -i /var/jenkins\_home/jenkins ae403@192.168.1.22 \"cd $TARGET\_DIR ; docker rmi $dockerImage:$dockerTag\""

                }

            }

        }

    }

    post {

        always {

            cleanWs()

        }

    }

}

Let's look at each part of this pipeline.

1. Variable Declarations:
   1. *dockerImage*: Variable to store the Docker image name.
   2. *dockerTag*: Variable to store the Docker image tag.
   3. *k8sReplica*: Number of replicas for the Kubernetes deployment.
   4. *k8sNamespace*: Kubernetes namespace where the service will be deployed.
   5. *userIdCause*: Stores information about the user who triggered the build.
   6. *now*: Current timestamp in the Asia/Ho\_Chi\_Minh time zone.
2. Pipeline Configuration:
   1. The pipeline is defined with the pipeline block. It specifies that the pipeline can run on any available agent.
3. Tool Setup:
   1. Maven and JDK tools are configured using the tools section. This ensures that the required tools are available for the pipeline.
4. Environment Variables:
   1. Environment variables are defined, including credentials for accessing a GitHub repository, the Git branch to be deployed, the Git repository URL, and the target directory on the server.
5. Stages:
   1. The pipeline is divided into stages:
      1. *Checkout source*: This stage checks out the source code from a GitHub repository.
      2. *Build authenticate service*: In this stage, the source code is built using Maven. Two separate directories, common and hugio-auth-service, are used to build the project.
      3. *Copy source*: Files and artifacts needed for deployment, including the Helm chart, target build artifacts, Dockerfile, and configuration file, are copied to the target directory on the remote server.
      4. *Deploy source*: This stage involves building the Docker image, loading it into a local Kubernetes cluster using kind, and upgrading the Kubernetes deployment using Helm. After deployment, the Docker image is removed.
6. Post-Build Cleanup:
   1. The post section contains a cleanup step that ensures workspace cleanliness after the build.

### 4.3.3 Benefits of CI/CD with Jenkins

* Automation: Jenkins automates the entire process from code integration to deployment, reducing manual intervention and minimizing errors.
* Rapid Feedback: Automated tests provide rapid feedback on code quality, enabling early detection of defects.
* Consistency: Consistent build and deployment processes ensure that all environments are identical, reducing deployment-related issues.
* Quick Iterations: CI/CD promotes frequent and smaller code changes, facilitating faster iterations and reducing the risk of large-scale failures.

## 4.4 Containerization with Docker

Containerization has emerged as a game-changing technology in the world of software development, addressing challenges related to application portability, environment consistency, and deployment efficiency. In this section, we explore the role of Docker in the project's deployment strategy and how it enables seamless packaging, distribution, and execution of applications.

### 4.4.1 Docker in Hugio

Docker is a leading platform for containerization, allowing developers to encapsulate applications and their dependencies in isolated environments called containers. Containers offer a consistent runtime environment, ensuring that applications behave predictably across various stages of the software development lifecycle.

All services in the Hugio project are running on the K8S platform. Therefore, it is mandatory to have a Docker image containing the application.

A Dockerfile script for a frontend application is as follows, with similar content for other apps.

FROM nginx

COPY ./nginx.conf /etc/nginx/conf.d/default.conf

COPY ./dist/apps/shell /build

ENTRYPOINT ["nginx", "-g", "daemon off;"]

Let’s take a look of each part of this Dockerfile script.

1. Base Image:
   1. FROM nginx: This line specifies the base image for the Docker image. In this case, it uses the official Nginx image as the base image. This image provides a basic Nginx web server environment.
2. Configuration and Content Copy:
   1. COPY ./nginx.conf /etc/nginx/conf.d/default.conf: This line copies a custom Nginx configuration file (usually named nginx.conf) to the /etc/nginx/conf.d/ directory inside the Docker container. This configuration file is used to configure how Nginx serves content and handles requests.
   2. COPY ./dist/apps/shell /build: This line copies the built frontend application files from the local directory ./dist/apps/shell into the /build directory within the Docker container. This step ensures that the application's static assets are available for Nginx to serve.
3. Entry Point:
   1. ENTRYPOINT ["nginx", "-g", "daemon off;"]: This line specifies the command that will be executed when the Docker container is started. In this case, it runs Nginx with the arguments -g and "daemon off;". The "daemon off;" argument prevents Nginx from running in the background and keeps it running in the foreground, which is suitable for containerized environments.

A Dockerfile script for a backend application is as follows, with similar content for other apps.

FROM tuanloc/ubuntu:java17.0.8

COPY ./target/app.jar app.jar

COPY ./logback.xml logback.xml

ENTRYPOINT ["java", "-Dfile.encoding=UTF8", "-Dlogging.config=/logback.xml", "-Dlogging.level.org.hibernate.SQL=OFF", "-Dlogging.level.org.hibernate.type.descriptor.sql.BasicBinder=OFF", "-jar", "app.jar"]

Let’s take a look of each part of this Dockerfile script.

1. Base Image:
   1. FROM tuanloc/ubuntu:java17.0.8: This line specifies the base image for the Docker image. It appears to be a custom Ubuntu-based image tagged with java17.0.8. This suggests that it's a specific Ubuntu image with Java 17 installed.
2. Copying Files:
   1. COPY ./target/app.jar app.jar: This line copies a file named app.jar from the local directory ./target/ to the root directory of the Docker image. This file is typically the executable Java application JAR file.
   2. COPY ./logback.xml logback.xml: This line copies a logback.xml configuration file from the local directory to the root directory of the Docker image. This file is used to configure logging for the Java application.
3. Entry Point:
   1. ENTRYPOINT ["java", "-Dfile.encoding=UTF8", "-Dlogging.config=/logback.xml", "-Dlogging.level.org.hibernate.SQL=OFF", "-Dlogging.level.org.hibernate.type.descriptor.sql.BasicBinder=OFF", "-jar", "app.jar"]: This line specifies the command that will be executed when the Docker container is started. It runs the Java application using the java command.
      1. -Dfile.encoding=UTF8: Sets the file encoding for the Java application to UTF-8.
      2. -Dlogging.config=/logback.xml: Specifies the location of the logback.xml configuration file for logging.
      3. -Dlogging.level.org.hibernate.SQL=OFF: Configures the logging level for Hibernate SQL statements to be turned off.
      4. Dlogging.level.org.hibernate.type.descriptor.sql.BasicBinder=OFF: Configures the logging level for Hibernate SQL type descriptors to be turned off.
      5. -jar: Indicates that the JAR file specified after this flag should be executed.
      6. app.jar: The name of the JAR file to be executed. This is the application's entry point.

### 4.4.2 Benefits of Docker

* Portability: Docker containers encapsulate the application and its dependencies, making it easy to move and run the application consistently across different environments, from development to production.
* Environment Consistency: Containers eliminate the "it works on my machine" problem by ensuring that the development environment matches the production environment.
* Isolation: Containers provide process-level isolation, preventing conflicts between applications and ensuring that they don't interfere with each other.
* Efficiency: Docker's lightweight nature and efficient resource utilization enable the deployment of multiple containers on a single host without the overhead of traditional virtual machines.

## 4.5 Orchestration with Kubernetes and Helm

Containerization with Docker brings significant advantages in terms of consistency and portability, but managing and scaling a large number of containers manually can be complex. Enter Kubernetes and Helm, which together provide a comprehensive solution for orchestrating, deploying, and managing containerized applications efficiently.

### 4.5.1 Introduction to Kubernetes

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It abstracts the underlying infrastructure and provides a declarative approach to application deployment.

### 4.5.2 Key Kubernetes Concepts

* *Pods*: The basic deployment unit in Kubernetes, containing one or more containers that share network and storage resources.
* *Services*: Provide a stable IP address and DNS name for accessing a set of pods, enabling load balancing and service discovery.
* *ReplicaSets*: Ensure that a specified number of pod replicas are running at all times, providing high availability and fault tolerance.
* *Deployments*: Enable declarative updates to applications, automating scaling and rolling updates to ensure application availability.

### 4.5.3 Benefits of Kubernetes

* *Scalability*: Kubernetes enables horizontal scaling of applications by adding or removing pods based on demand.
* *Automated Rollouts and Rollbacks*: Kubernetes automates the process of deploying and updating applications while ensuring zero-downtime rollouts.
* *Self-Healing*: If a pod or node fails, Kubernetes automatically replaces or reschedules the affected containers to maintain application availability.
* *Declarative Configuration*: Developers describe the desired state of the application in configuration files, and Kubernetes ensures that the actual state matches the desired state.

### 4.5.4 Introduction to Helm

Helm is a package manager for Kubernetes that simplifies the deployment and management of applications. It uses charts, which are packages that contain pre-configured Kubernetes resources.

### 4.5.5 Helm Charts

* *Chart Structure*: Helm charts consist of templates (Kubernetes resource definitions), values files (configuration options), and a Chart.yaml file (metadata).
* *Reusability*: Charts can be shared and reused, enabling consistent application deployments across different environments.

### 4.5.6 Benefits of Helm

* *Simplified Deployment*: Helm abstracts complex Kubernetes resource configurations into reusable and customizable charts.
* *Versioning and Rollbacks*: Helm enables versioned releases and easy rollbacks, ensuring consistency in application deployments.

### 4.5.7 Helm and Kubernetes in Hugio

In Hugio, all service is running with Kubernetes (K8s).

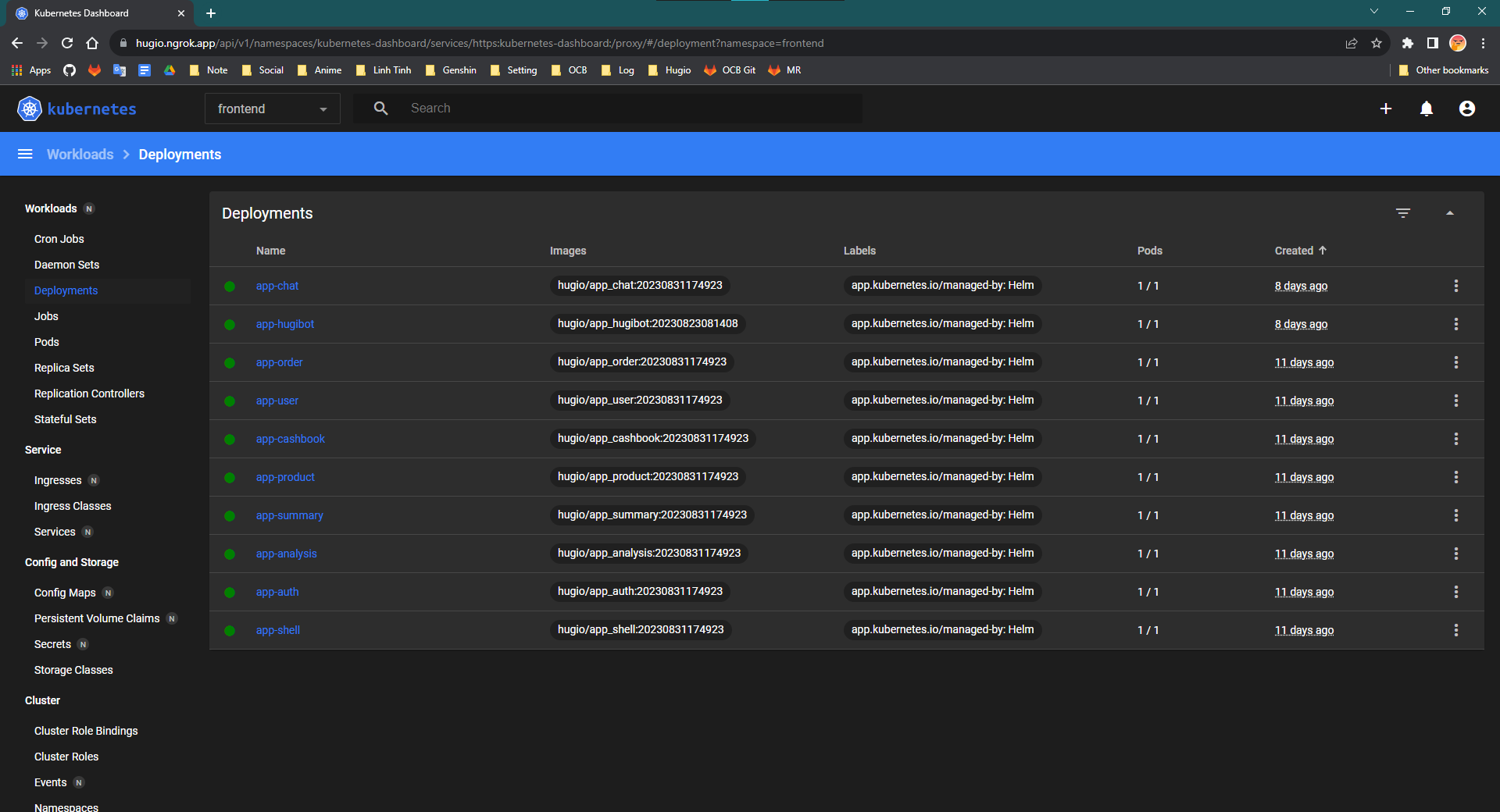


Figure 39 Hugio K8S Dashboard for Front end deployment

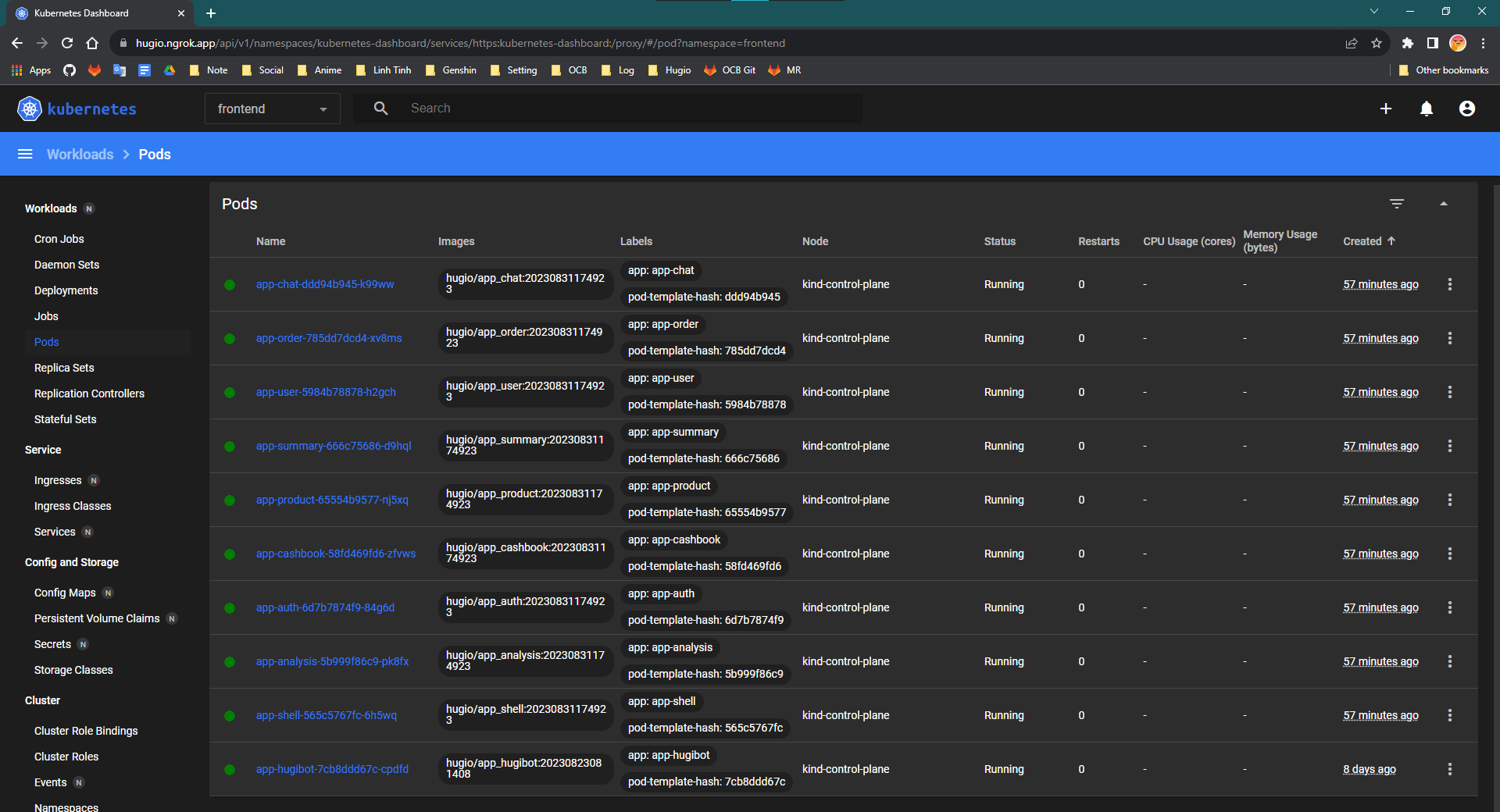


Figure 40 Hugio K8S Dashboard for Front end pod

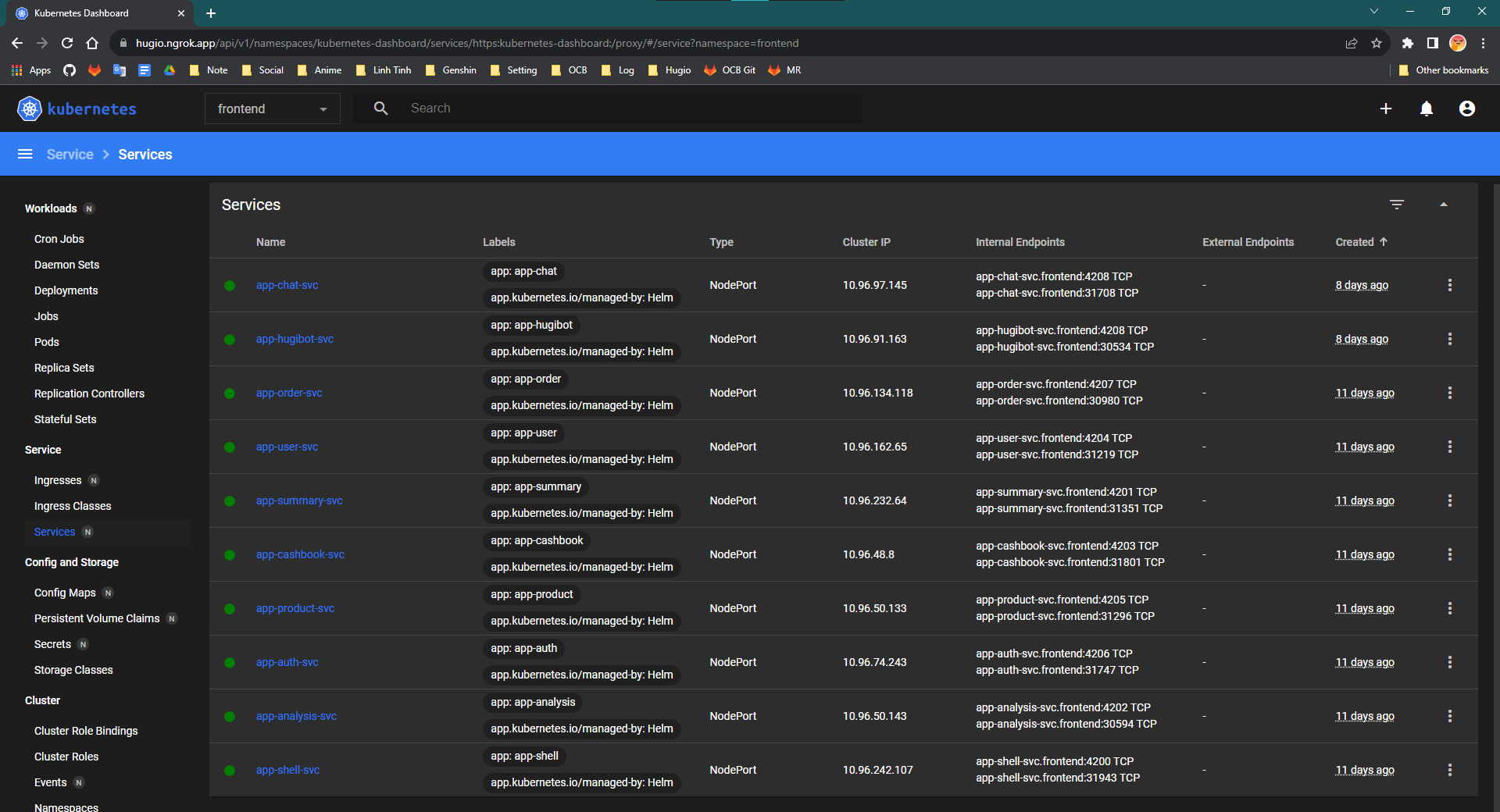


Figure 41 Hugio K8S Dashboard for Front end service

The images above provided appear to be screenshots of Kubernetes (K8s) resources associated with frontend services.



Figure 42 Hugio K8S Dashboard for Back end deployment

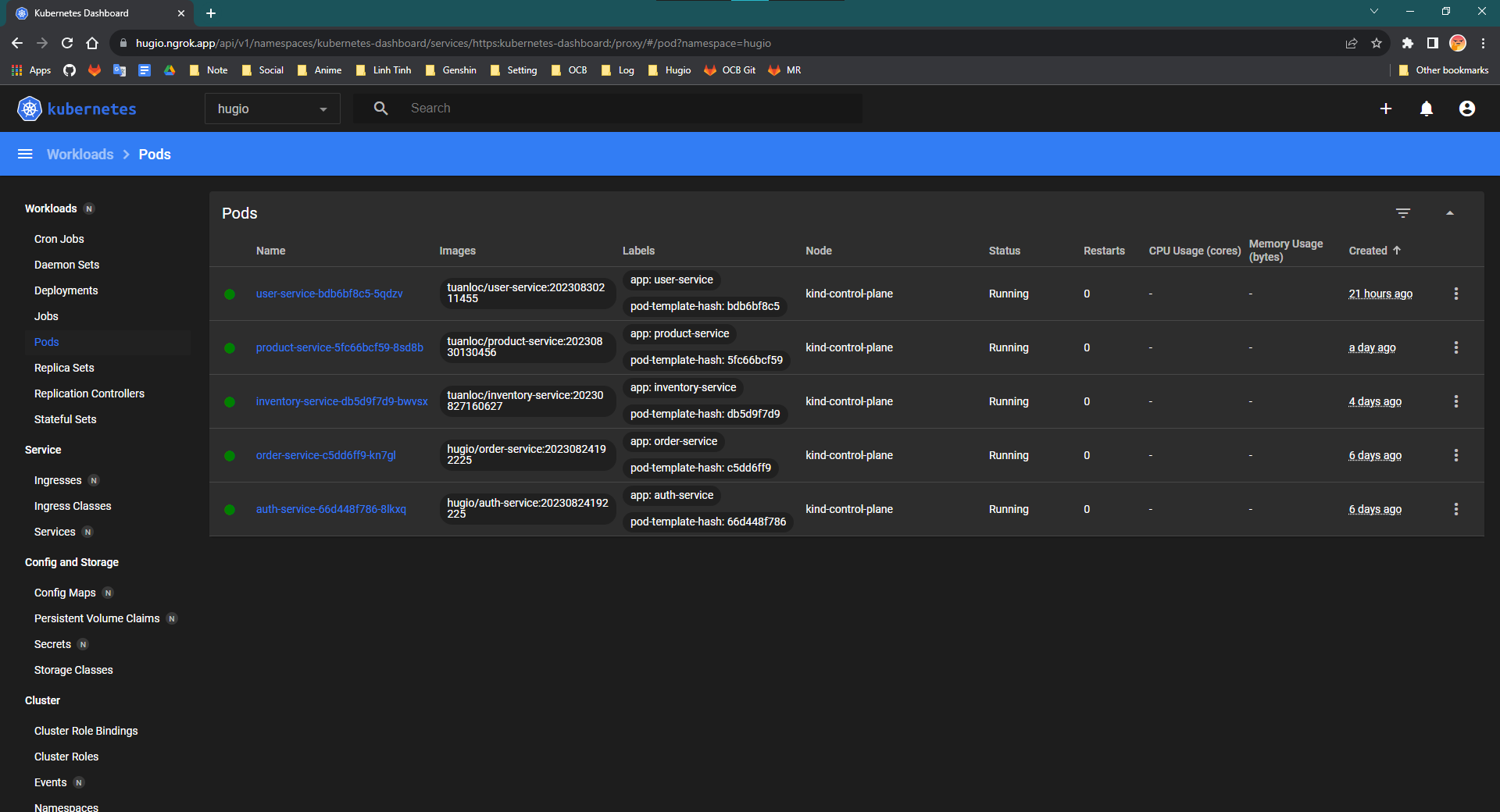


Figure 43 Hugio K8S Dashboard for Back end pods

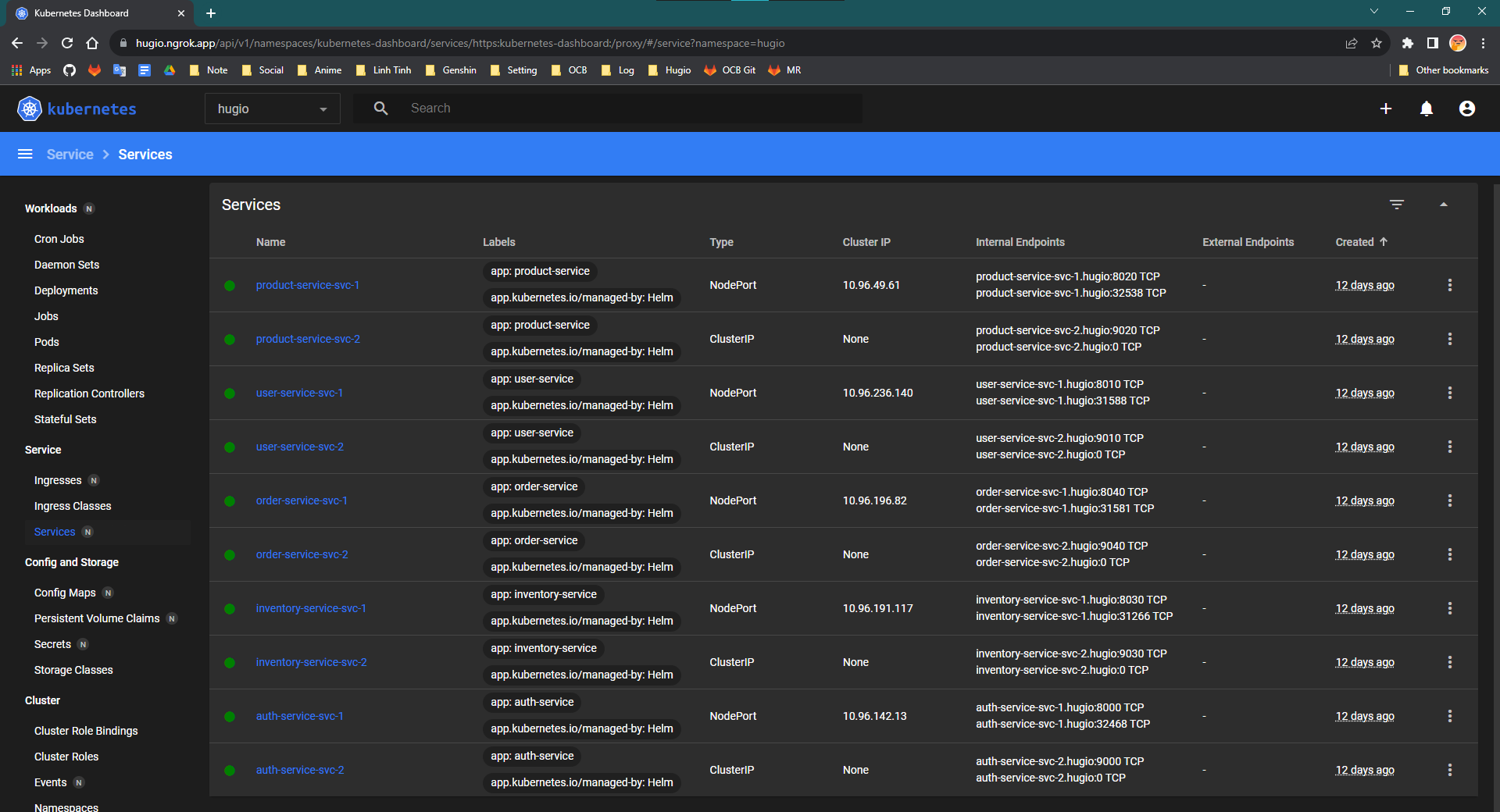


Figure 44 Hugio K8S Dashboard for Back end services

The images above provided appear to be screenshots of Kubernetes (K8s) resources associated with backend services.

## 4.6 Web Server Configuration with Nginx

A crucial component of any application deployment strategy is the configuration of a reliable and high-performance web server. In this section, we explore the integration of Nginx as a web server and reverse proxy in the project, enhancing application delivery, load balancing, and security.

### 4.6.1 Introduction to Nginx

Nginx is an open-source web server and reverse proxy server renowned for its high performance, scalability, and versatility. It excels in efficiently handling concurrent connections, distributing traffic, and serving static and dynamic content.

### 4.6.2 Serving Static Content

Nginx is particularly efficient in serving static assets like HTML, CSS, JavaScript, and images. Its lightweight architecture allows it to serve static content rapidly, reducing load times and enhancing user experience.

### 4.6.3 Reverse Proxy Configuration

Nginx can act as a reverse proxy, forwarding requests from clients to backend application servers. This arrangement provides an extra layer of security, as the application servers are not directly exposed to the internet.

### 4.6.4 Load Balancing

Nginx's load balancing capabilities distribute incoming requests across multiple backend servers, improving application performance and ensuring high availability. This is especially beneficial for applications with high traffic loads.

### 4.6.5 Benefits of Nginx

* Performance: Nginx's event-driven architecture enables it to handle a large number of concurrent connections efficiently.
* Scalability: Load balancing and reverse proxy capabilities ensure that the application scales seamlessly to accommodate increased traffic.
* Security: Nginx enhances security by acting as a barrier between clients and backend servers and providing protection against common web attacks.
* Caching: Nginx can cache static content, reducing the load on backend servers and improving response times.

## 4.7 Deployment Process in Hugio

The deployment process is a critical phase that ensures the smooth transition of code changes from development to production environments. In this section, we outline the step-by-step deployment process using the tools and technologies discussed earlier, including GitHub, Jenkins, Docker, Kubernetes, Helm, and Nginx.

### 4.7.1 Code Commit and Pull Request

* Developers create feature branches in GitHub for new features or bug fixes.
* Code changes are developed, tested, and committed to the feature branches.
* Developers submit pull requests to merge their changes into the main branch.
* Code reviews are conducted to ensure quality and correctness.

### 4.7.2 Continuous Integration with Jenkins

* Upon pull request creation or code push, GitHub's webhooks trigger a Jenkins job.
* Jenkins checks out the latest code from the repository.
* Jenkins builds the Docker image using the Dockerfile and pushes it to a Docker registry.
* Automated tests, including unit tests and integration tests, are executed.

### 4.7.3 Continuous Deployment to Kubernetes with Helm

* Jenkins triggers a deployment pipeline.
* Helm charts, which define application configurations, are pulled from the repository.
* Jenkins deploys the Helm charts to the Kubernetes cluster using Helm commands.
* Kubernetes orchestrates the deployment, scaling, and management of containerized applications.

### 4.7.4 Nginx Configuration and Load Balancing

* Nginx configurations, such as reverse proxy settings and SSL termination, are defined.
* Nginx is deployed within the Kubernetes cluster to serve as the web server and reverse proxy.
* Nginx load balancing distributes incoming traffic to the deployed application instances.

## 4.8 Benefits and Impact

The deployment strategy that encompasses the utilization of GitHub, Jenkins, Docker, Kubernetes, Helm, and Nginx brings forth a multitude of benefits and has a significant impact on the project's success, development velocity, and overall efficiency. In this section, we delve into the tangible advantages and the broader impact of adopting this deployment approach.

### 4.8.1 Streamlined Software Delivery

The integration of CI/CD practices through Jenkins accelerates the software delivery process. Automated testing and continuous deployment ensure that code changes are thoroughly validated and deployed quickly, reducing the time required to deliver new features and bug fixes.

### 4.8.2 Consistency and Reproducibility

Docker's containerization ensures that the application runs consistently across various environments, from development to production. This consistency reduces the chances of "it works on my machine" discrepancies and streamlines the debugging and testing process.

### 4.8.3 Scalability and Performance

Kubernetes orchestrates the deployment of containerized applications, enabling dynamic scaling and load balancing. This scalability ensures that the application can handle increased traffic demands without compromising performance.

### 4.8.4 Resource Efficiency

Docker's lightweight nature allows efficient utilization of resources, enabling multiple containers to run on a single host. Kubernetes optimizes resource allocation, leading to improved infrastructure utilization.

### 4.8.5 Continuous Feedback and Improvement

The CI/CD pipeline, combined with automated testing, provides continuous feedback on code quality. Rapid iterations and quick feedback loops foster a culture of continuous improvement.

### 4.8.6 Deployment Confidence

Automated testing, versioned releases, and Helm's rollbacks ensure that only thoroughly tested and validated code changes reach the production environment, boosting deployment confidence.

### 4.8.7 Collaboration and Visibility

GitHub's collaboration features, pull requests, and code reviews enhance collaboration among team members. The visibility into code changes, discussions, and reviews promotes shared ownership of the codebase.

# SOFTWARE OVERVIEW

## 5.1 Summary page

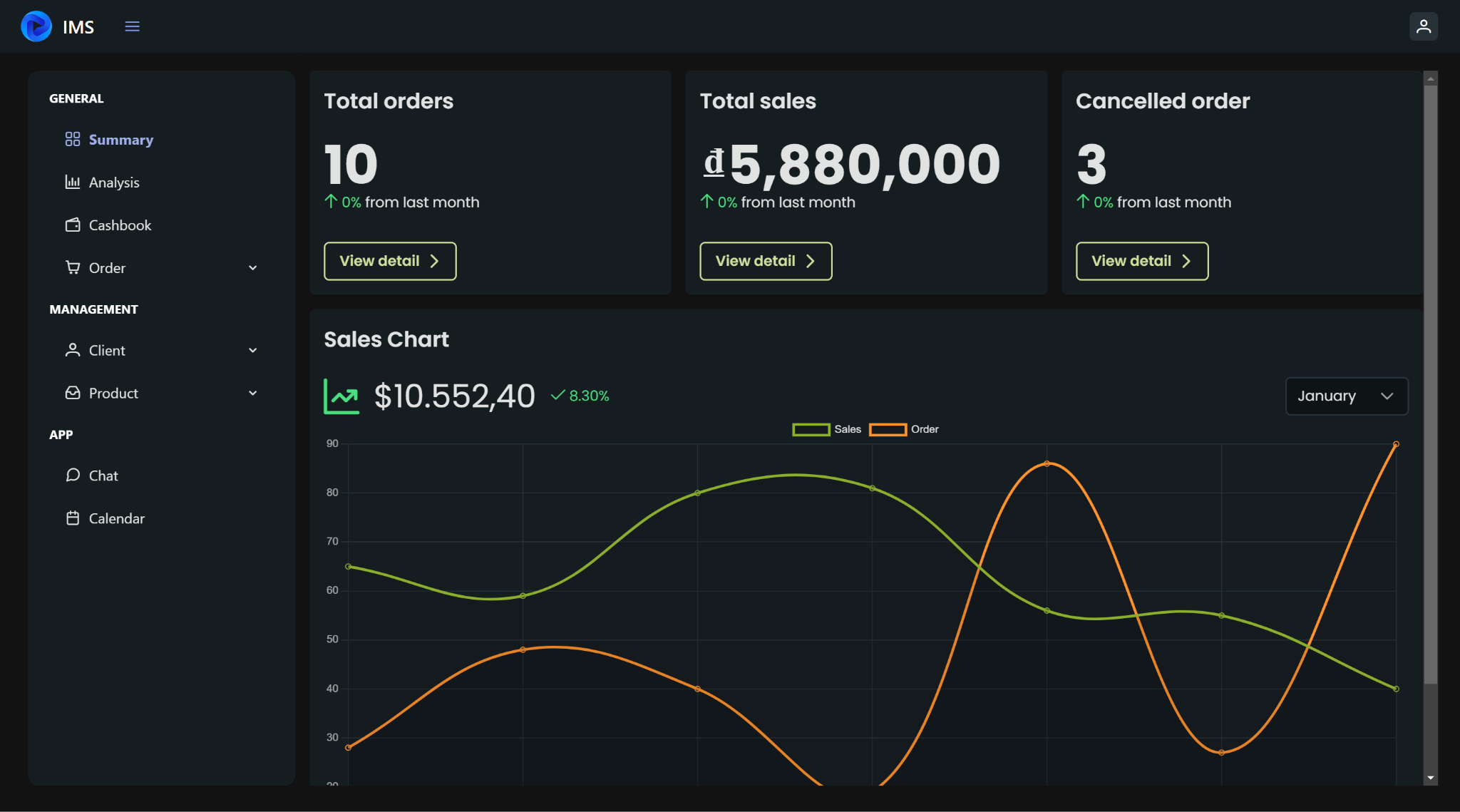


Figure 45 Summary page

The Summary Page serves as a concise yet insightful dashboard that provides a snapshot of key metrics related to the performance and health of the application. Displaying data on Total Orders, Total Sales, and Cancelled Orders, this page offers stakeholders a quick overview of the application's engagement, revenue generation, and user experience. This report delves into the significance of each metric and how they contribute to informed decision-making and strategic planning.

### 5.1.2 Total Orders

The "Total Orders" metric is a fundamental indicator of customer engagement and the application's utilization. It represents the cumulative count of successful orders placed by users. This metric offers insights into customer behavior, demand trends, and the popularity of products or services. An upward trend in Total Orders can signify a growing customer base, increased market demand, or successful marketing initiatives.

Importance:

* Customer Engagement: Total Orders reveal how actively users are interacting with the application, indicating the application's value proposition.
* Sales Revenue: An increase in Total Orders directly impacts sales revenue, contributing to business growth and profitability.
* Market Demand: Monitoring Total Orders helps identify high-demand periods, enabling proactive inventory management and capacity planning.

### 5.1.3 Total Sales

The "Total Sales" metric provides a monetary representation of the application's revenue generated from successful transactions. This metric is pivotal for assessing the application's financial performance and gauging its contribution to the organization's revenue stream. Total Sales metrics help in measuring the effectiveness of marketing campaigns, pricing strategies, and user engagement.

Importance:

* Financial Performance: Total Sales offer a clear view of the application's revenue generation, aiding in financial analysis and forecasting.
* ROI Assessment: It enables tracking the return on investment for marketing efforts and operational expenses related to sales.
* Revenue Growth: Monitoring Total Sales over time helps to identify growth trends and opportunities for revenue enhancement.

### 5.1.4 Cancelled Orders

The "Cancelled Orders" metric sheds light on the instances where users initiate an order but eventually abandon the process before completion. These cancelled orders can occur due to various reasons, including user experience issues, pricing concerns, or technical glitches. Understanding the frequency and reasons for cancelled orders is crucial for optimizing the checkout process and reducing revenue loss.

Importance:

* User Experience: High cancelled order rates might indicate friction in the user experience, prompting the need for improvements.
* Conversion Rate Optimization: Analyzing cancelled orders helps identify pain points in the purchasing journey and optimizes conversion rates.
* Revenue Preservation: By addressing the root causes of cancelled orders, revenue leakage can be minimized and potential sales realized.

## 5.2 Order page

The Order Page serves as a comprehensive hub that provides an overview of all orders processed through the application. Displaying critical details of each order, such as product listings, client information, order code, total price, status, and creator information, this page offers stakeholders valuable insights into the transaction history and the application's effectiveness in managing customer interactions. This report delves into the significance of each element and how they collectively contribute to streamlined operations and customer satisfaction.

### 5.2.1 Order list

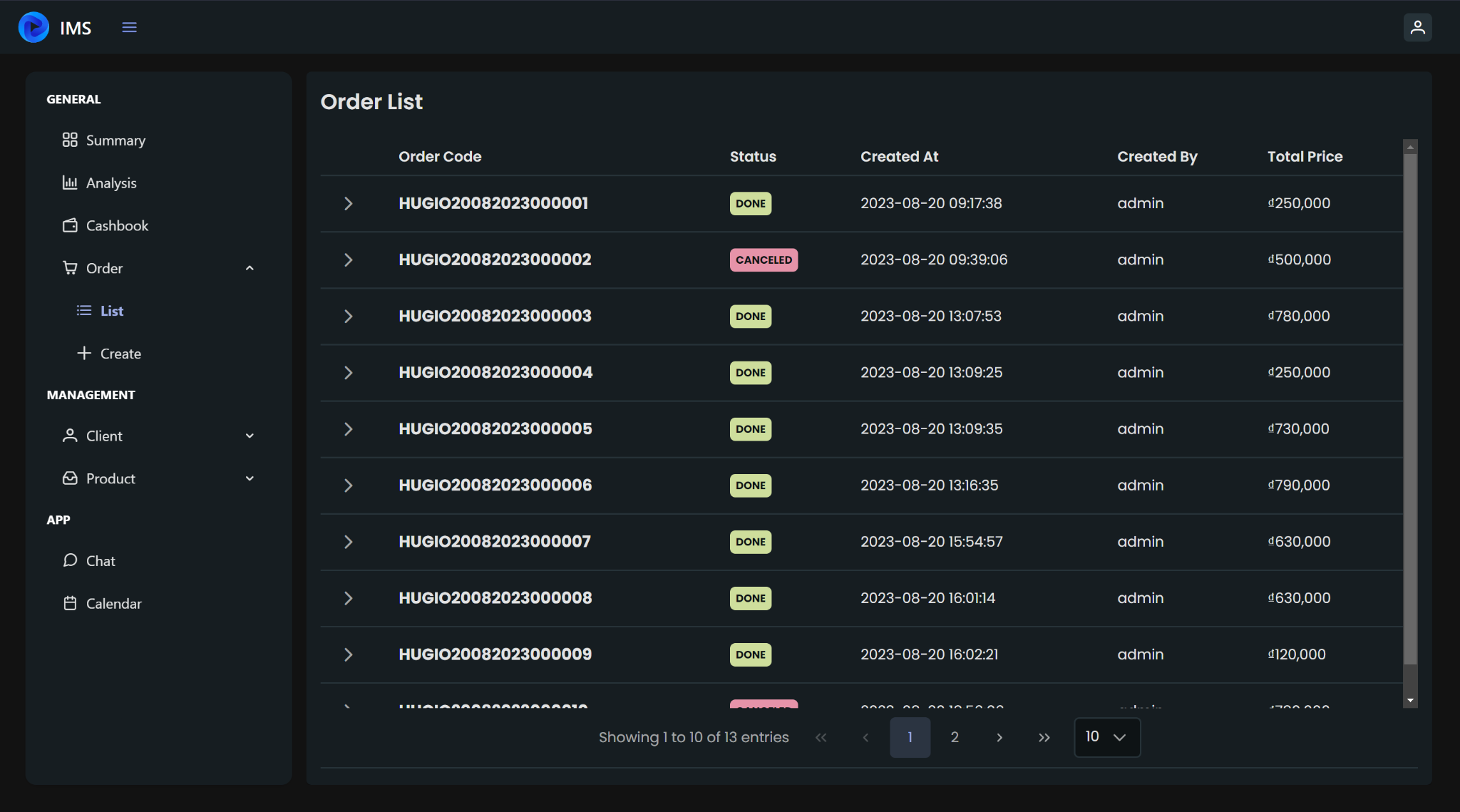


Figure 46 List of Order page

The core feature of the Order Page is the presentation of a list of orders. Each entry in the list represents an individual order placed by a customer. The list provides a high-level overview of all transactions, making it easier to manage and track the order history. The ability to search, filter, and sort orders based on various criteria enhances the user experience and facilitates efficient order management.

Importance:

* Order History: The order list offers a centralized view of all past transactions, enabling customer support and finance teams to quickly access order-related information.
* Tracking and Verification: Users can easily track the status and progress of each order, providing transparency and reducing customer inquiries.
* Operational Efficiency: The order list allows for efficient processing, management, and reporting of orders, optimizing internal operations.

Order Details:

* Product List: A breakdown of the products included in the order, providing insights into the composition of the purchase.
* Client Information: Details about the client who placed the order, including contact information and billing/shipping addresses.
* Order Code: A unique identifier assigned to each order, facilitating tracking and reference.
* Total Price: The cumulative cost of the products in the order, aiding financial reporting and analysis.
* Status: Indication of whether the order is marked as "Done" or "Canceled," offering a clear overview of order fulfillment.
* Created By: Information about the user or staff member who initiated the order.

### 5.2.2 Create order

The Create Order Page serves as a versatile tool that enables users to initiate new orders seamlessly. By offering options to select products from a list, scan products through QR codes using the camera, and assign client information or proceed as a guest, this page enhances the user experience and provides flexibility in creating orders. This report explores the significance of each functionality and how they contribute to efficient order initiation and client engagement.

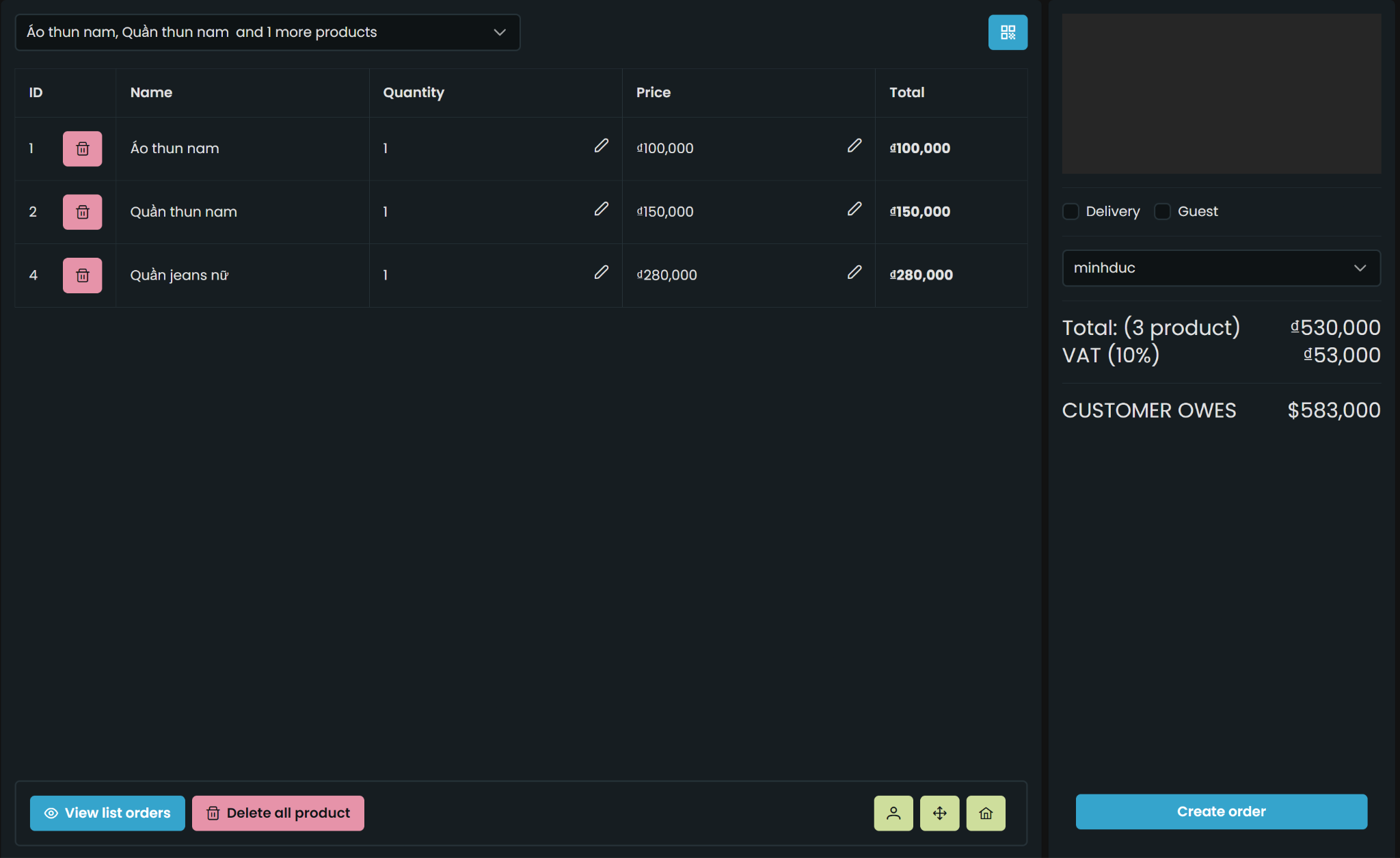


Figure 47 Create Product page

Product Selection:

The Product Selection feature allows users to choose products from a list presented on the Create Order Page. This list includes available products, enabling users to browse and select items they wish to purchase. Each selected product is added to the user's cart, preparing them for checkout.

* User Convenience: Product selection from a list provides a familiar and user-friendly way to build an order.
* Visual Representation: Users can view product details and make informed decisions based on the product information.
* Bulk Ordering: Users can easily add multiple products to the cart, streamlining the order creation process.

QR Code Scanning:

QR Code Scanning is an innovative feature that allows users to scan product QR codes using their device's camera. Scanning a QR code instantly adds the corresponding product to the cart. This method offers a quick and efficient way to add products without manual searching.

* Efficiency: QR code scanning eliminates the need for manual product selection, saving time and effort.
* Error Reduction: QR code scanning minimizes the risk of selecting the wrong product, improving order accuracy.
* Enhanced Experience: The modern and intuitive QR code scanning enhances user engagement and satisfaction.

Client Assignment or Guest Checkout:

Users have the option to either assign an order to an existing client or proceed as a guest without assigning the order to a specific client. Assigning an order to an existing client allows for easy tracking of past orders, preferences, and client history. Proceeding as a guest is suitable for one-time purchases or users who haven't created an account.

* Client Engagement: Assigning orders to existing clients improves client relations and personalizes the experience.
* Order Tracking: Assigned orders are associated with client profiles, facilitating order history and follow-up communication.
* Accessibility: Guest checkout accommodates users who prefer not to create an account, ensuring inclusivity.

## 5.3 Client page

The Client Page serves as a central hub for managing client information, facilitating client interactions, and maintaining an organized database of clientele. With features that include displaying a list of clients, creating new clients, editing existing client details, and deleting clients, this page empowers users with the tools to manage and engage with clients effectively. This report explores the significance of each feature and how they collectively contribute to efficient client relationship management.

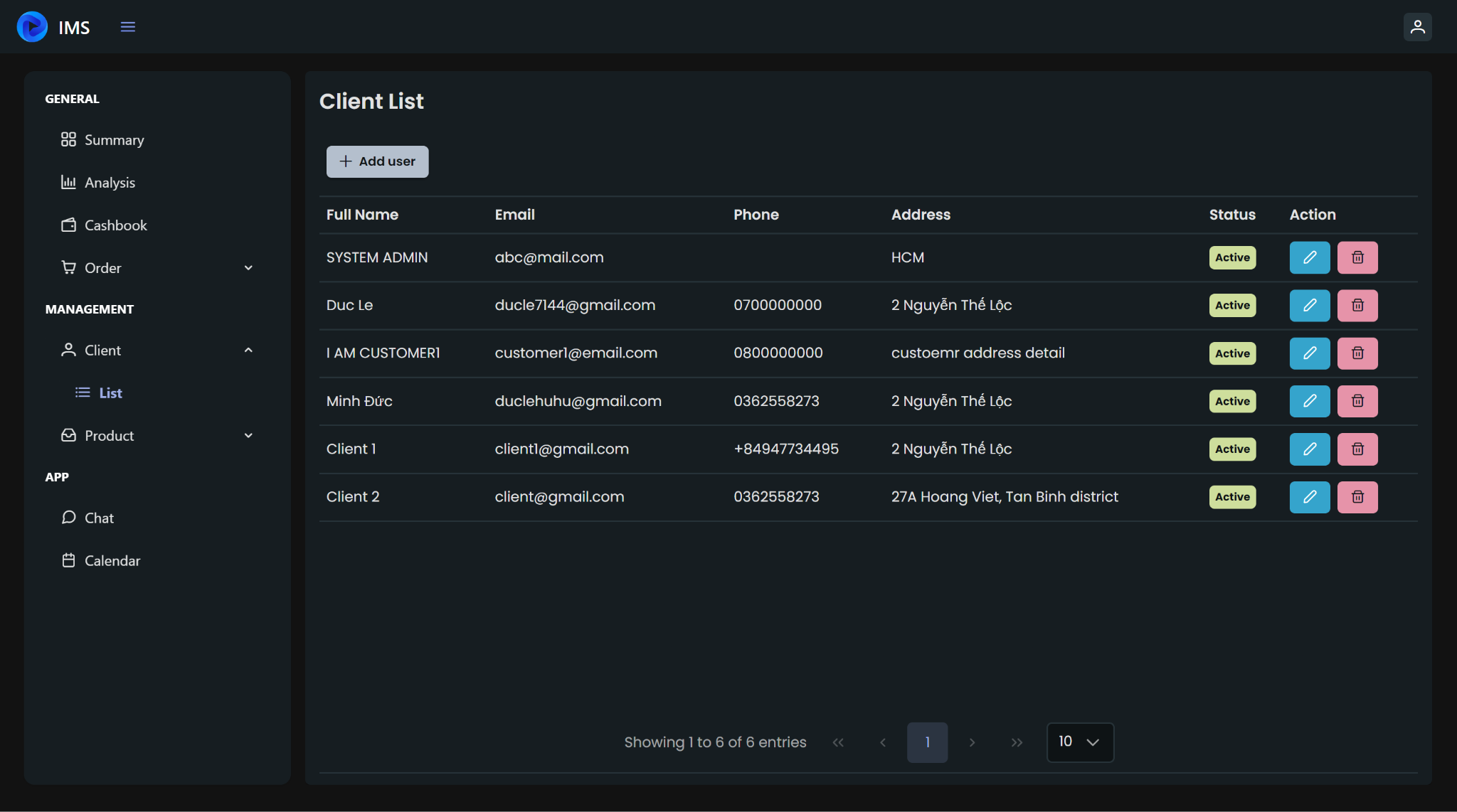


Figure 48 List of clients page

### 5.3.1 List of Clients

The core functionality of the Client Page is to present a comprehensive list of clients. Each entry in the list represents an individual client registered within the system. The list provides a convenient overview of all clients, making it easier to search for specific clients, identify key contacts, and access contact information promptly.

Importance:

* Contact Accessibility: The list provides quick access to contact details, ensuring efficient communication and engagement.
* Client Segmentation: The list helps categorize clients for targeted communication, personalized marketing, and client-specific initiatives.
* Centralized Information: A centralized list streamlines client management, eliminating the need for scattered records.

### 5.3.2 Creating New Clients

The "Create New Client" feature allows users to add new clients to the system. Users can input client information such as name, phone number, email, and address. This functionality is essential for onboarding new clients and expanding the client base.

Importance:

* Client Expansion: Creating new clients fosters business growth by expanding the customer base and potential sales.
* Data Accuracy: Direct data entry ensures accurate and up-to-date client information, enhancing client interactions.
* Onboarding Process: Streamlined client creation simplifies the onboarding of new clients, minimizing administrative tasks.

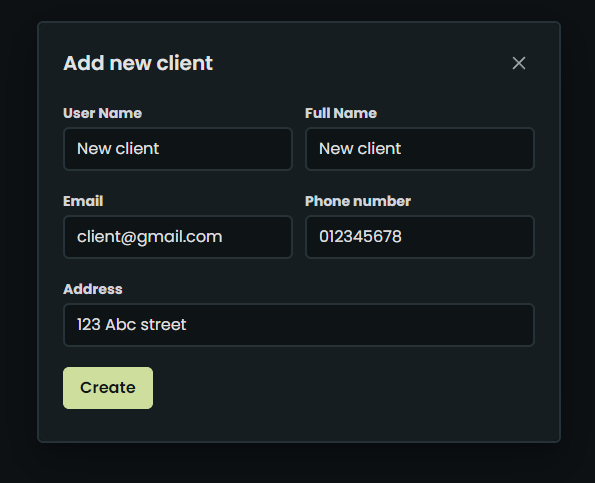


Figure 49 Add new client

### 5.3.3 Editing Client Details

Users can edit existing client details, allowing for updates and corrections to be made as needed. This feature ensures that the information in the system remains accurate and relevant, enabling effective communication and engagement.

Importance:

* Data Integrity: Editing client details maintains accurate client profiles, preventing outdated information from affecting interactions.
* Customization: The ability to update client details enables personalized communication and engagement strategies.
* Continuity: Accurate information fosters seamless client interactions, contributing to positive client experiences.

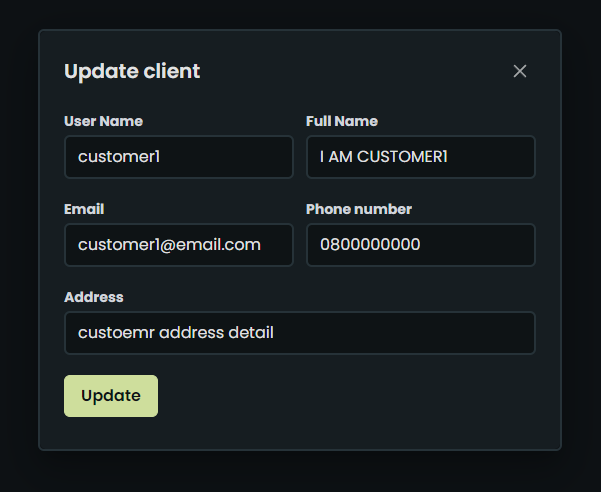


Figure 50 Update client

### 5.3.4 Deleting Clients

The "Delete Client" feature enables users to remove clients from the system. This action should be exercised with caution, as it permanently removes client records and associated data.

Importance:

* Data Management: Deleting outdated or irrelevant client records ensures data hygiene and a clean database.
* Compliance: The ability to remove client records aligns with data protection regulations and privacy standards.
* Resource Efficiency: Deleting unnecessary records prevents clutter and enhances system performance.

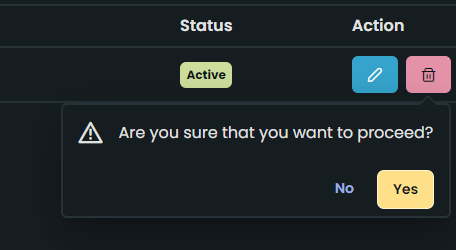


Figure 51 Delete User

## 5.4 Product page

The Product Page serves as a central hub for managing product information, facilitating inventory control, and streamlining product-related operations. With features that include displaying a list of products, creating new products, editing existing product details, deleting products, and viewing product QR codes, this page empowers users with the tools to efficiently manage the product catalog. This report delves into the significance of each feature and how they collectively contribute to efficient product management and inventory control.

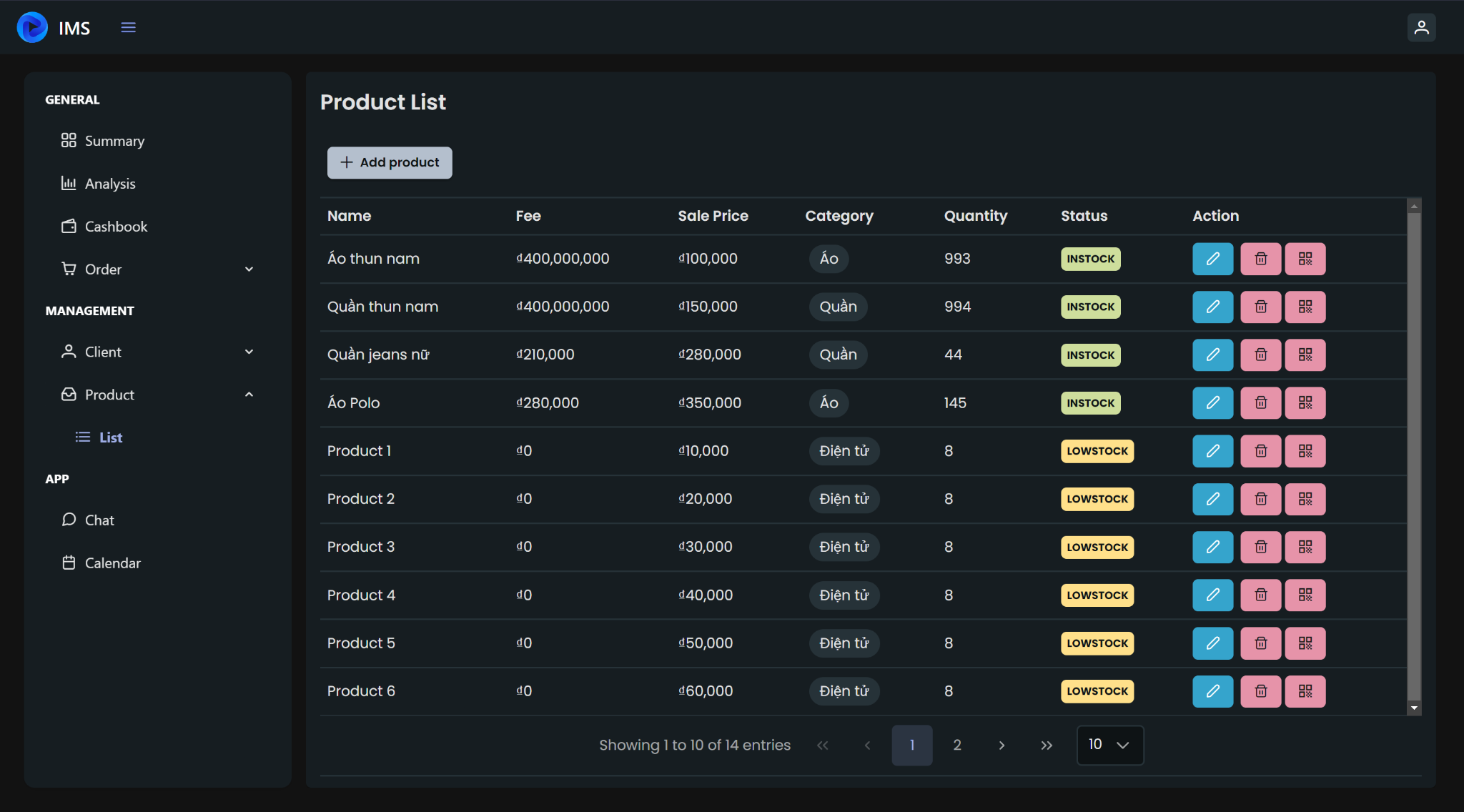


Figure 52 Product list page

### 5.4.1 List of Products

The cornerstone functionality of the Product Page is to present an organized list of products. Each entry in the list represents an individual product available within the catalog. The list provides a clear overview of the product range, making it easy to identify products, monitor quantities, and access product details promptly.

Importance:

* Inventory Oversight: The list enables users to monitor product quantities, track stock levels, and make informed restocking decisions.
* Quick Reference: The organized list facilitates quick access to product information, streamlining order processing and customer inquiries.
* Categorization: Users can categorize products based on categories or attributes, aiding product discovery and marketing strategies.

### 5.4.2 Creating New Products

The "Create New Product" feature allows users to add new products to the catalog. Users can input essential product information, including name, price, category, quantity, and status. This functionality is pivotal for expanding the product catalog and offering a diverse range of options to customers.

Importance:

* Catalog Growth: Creating new products diversifies the product range, attracting a broader customer base and catering to varied preferences.
* Stock Management: The ability to input quantity and status streamlines inventory control, facilitating accurate stock updates.
* Product Variety: A growing catalog improves customer choices, potentially boosting sales and customer satisfaction.

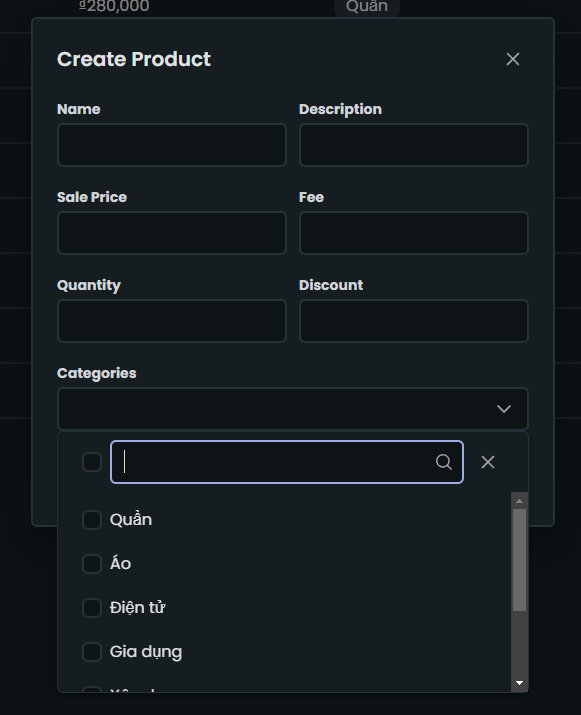


Figure 53 Create product

### 5.4.3 Editing Product Details

Users can edit existing product details, allowing for updates and corrections as required. This feature ensures that product information remains accurate and reflects any changes, such as price adjustments or specification updates.

Importance:

* Accurate Information: Editing product details ensures accurate and up-to-date product information for customers and internal teams.
* Pricing Management: The ability to edit prices and other attributes supports dynamic pricing strategies and promotional activities.
* Data Integrity: Accurate product details contribute to transparency and trust between the business and customers.

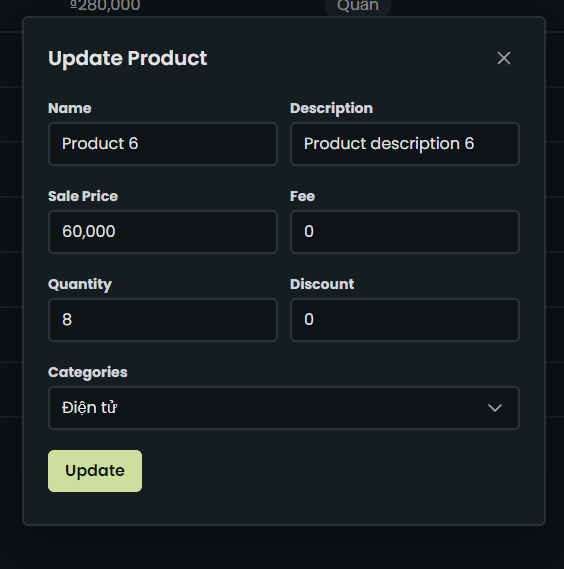


Figure 54 Update product

### 5.4.4 Deleting Products

The "Delete Product" feature enables users to remove products from the catalog. This action should be exercised with caution, as it permanently removes product records and associated data.

Importance:

* Inventory Cleanup: Deleting discontinued or outdated products maintains a clean and up-to-date product catalog, improving navigation and customer experience.
* Data Hygiene: Removing irrelevant products ensures data accuracy and streamlines the catalog for better performance.
* Regulatory Compliance: Deleting products aligns with regulatory requirements and prevents presenting unavailable items to customers.

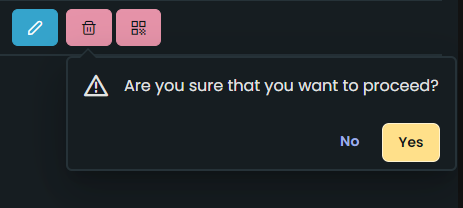


Figure 55 Delete Product

### 5.4.5 Viewing QR Codes

Users can view the QR code associated with each product. The QR code serves as a quick reference to the product's details, enabling easy access to information for scanning or sharing.

Importance:

* Quick Access: QR codes offer a convenient way for users to access product information on the go, whether for inventory tracking or customer assistance.
* Inventory Control: Scanning QR codes aids inventory management by providing instant access to product details and quantity information.
* Enhanced Efficiency: QR codes save time by directly linking users to specific product information without manual searching.

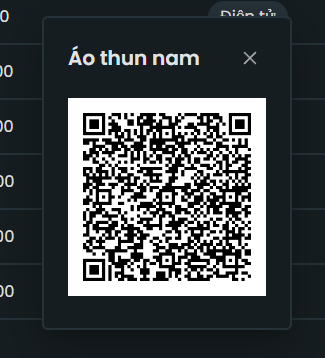


Figure 56 Example QR Code image

## 5.5 Chat page

The Chat Page serves as a versatile communication tool that enables users, specifically store owners, to interact with a bot to obtain information and answers to their queries. By integrating with the ChatGPT API, users can engage in real-time conversations to seek assistance related to products, stock, and other inquiries. Additionally, the Chat Page is designed for future scalability, allowing for communication with various individuals, such as clients and other store owners. This report explores the significance of bot-assisted communication and the potential for future expansion.

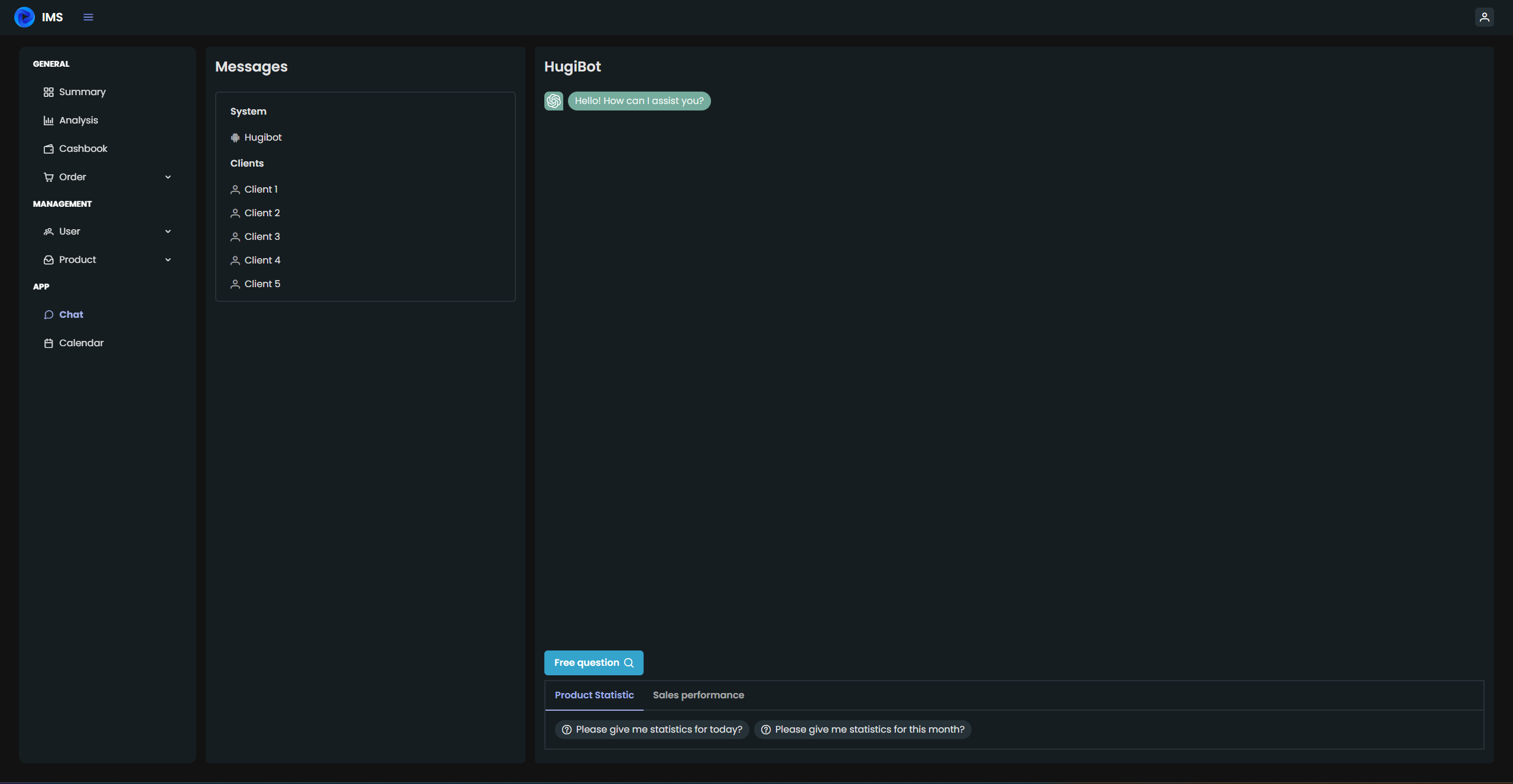


Figure 57 Chat page

### 5.5.1 Bot-Assisted Communication

The core functionality of the Chat Page involves interacting with a bot, powered by the ChatGPT API. Store owners can ask questions, seek guidance, and request information regarding products, stock availability, pricing, and other related topics. The bot processes the queries and responds with relevant and accurate information.

Importance:

* Real-Time Assistance: The bot provides instant answers to store owners' questions, eliminating the need for manual searches or waiting for support.
* Efficiency: Bot-assisted communication streamlines information retrieval, enabling store owners to make informed decisions promptly.
* Support Availability: The ChatGPT-powered bot ensures 24/7 support availability, enhancing user convenience and satisfaction.

### 5.5.2 Future Expansion

The Chat Page is designed with future expansion in mind, accommodating various communication scenarios beyond bot-assisted interactions. This includes the potential to communicate with clients or other store owners, transforming the Chat Page into a dynamic communication platform.

Importance:

* Enhanced Communication: Expanding the chat functionality to include communication with clients allows for personalized interactions and customer support.
* Business Networking: Enabling communication among store owners facilitates knowledge sharing, collaboration, and community building.
* Scalability: The adaptable framework prepares the Chat Page for accommodating new features and user groups as the business evolves.

# CONCLUSION

## Conclusion

We are very proud and pleased with the completion and implementation of this point of sale software. The development has marked an important step forward in providing efficient and convenient solutions for customers in the field of sales. The project was carried out closely with strong cooperation from development team members and customer support.

Here are some key points about our point of sale software:

*Easy to Use Interface*: We have designed a simple and friendly user interface, making it easy for users to perform shopping and payment operations.

*Efficient Product Management*: We offer a robust product management system that makes it easy for our customers to add, modify and remove products from their online store.

*Security and Data Protection*: We have made security and data protection a top priority, ensuring that our customers' personal information and business data are securely protected.

*Interaction and Support*: We have built an effective customer support and interaction system to deal with all inquiries and requests from customers.

We are committed to continuing to improve and develop this software to meet the changing needs of our customers and to help them grow their business in a sustainable way. Thank you for everyone's cooperation and support during this process.

## Development path

To develop a grocery store software and achieve the goal of increasing the user base to generate revenue and compete with industry rivals, we need to identify and implement several strategic development directions:

*Enhance User Interface (UI/UX)*: Firstly, we should focus on improving the user experience. Creating a user-friendly, easy-to-navigate, and engaging user interface can attract more users. This includes developing mobile applications for both iOS and Android to provide mobility and convenience.

*Expand Product Catalog*: Diversify the product catalog to ensure we meet a wide range of consumer needs. Provide more choices, including fresh products, packaged goods, and even doorstep delivery services.

*Multi-Channel Marketing Campaign*: Create a robust multi-channel marketing campaign to reach and attract more potential customers. Utilize social media, online advertising, and email marketing to boost awareness of our application.

*Establish a Loyalty Program*: Develop an enticing loyalty program and promotions to retain customers and encourage them to remain loyal to our application. Offer rewards, discounts, and special incentives to frequent customers.

*Data Analysis*: Utilize data analysis tools to gain a deeper understanding of user behavior and market trends. Based on this information, adjust the business strategy and enhance the user experience.

*Partnership Development*: Seek opportunities for collaboration with industry partners such as food suppliers and delivery companies to provide comprehensive and expedited services.

*Enhance Security and Safety*: Ensure that personal information and user accounts are well-protected to build trust and reliability among customers.

*Improved Customer Support*: Establish an exceptional customer support system to address all inquiries and requests from users.

By fully and meticulously implementing these development directions, we can increase our user base, generate revenue, and compete effectively in the grocery store industry.

# REFERENCES

Vietnamese

1. gRPC là gì? Vũ khí tối thượng tăng tải Microservices - <https://200lab.io/blog/grpc-la-gi/>
2. Giới thiệu tổng quan về Angular - <https://viblo.asia/p/gioi-thieu-tong-quan-ve-angular-07LKX9j2ZV4>
3. Angular là gì? Tổng quan về Angular dành cho người mới - <https://vietnix.vn/angular-la-gi/>
4. Tìm hiểu về Redis - <https://viblo.asia/p/tim-hieu-ve-redis-LzD5dXEW5jY>
5. Nx Workspace - Anh Cả trong việc xây dựng frontend architecture - <https://viblo.asia/p/nx-workspace-anh-ca-trong-viec-xay-dung-frontend-architecture-E375zAWqlGW>
6. MySQL là gì? Hướng dẫn toàn tập về MySQL - <https://wiki.matbao.net/mysql-la-gi-huong-dan-toan-tap-ve-mysql/>

English

1. What is gRPC? <https://grpc.io/docs/what-is-grpc/introduction/>
2. How apache-kafka working? <https://www.confluent.io/blog/apache-kafka-intro-how-kafka-works/>
3. What is angular? <https://angular.io/docs>
4. What is Nx Workspace? <https://nx.dev/concepts>
5. What is PrimeNG UI? <https://primeng.org/>
6. What is TailwindCSS? <https://tailwindcss.com/>
7. What are MicroFrontend? <https://micro-frontends.org/>
8. What is RxJS? <https://rxjs.dev/guide/overview>
9. What are microservices? <https://microservices.io/>
10. What is Kubernetes? - <https://kubernetes.io/vi/docs/concepts/overview/what-is-kubernetes/>