PHENIKAA UNIVERSITY

PHENIKAA SCHOOL OF COMPUTING

ONLINE FOOD ORDERING MANAGEMENT

Course: Software Architecture

Course Class: CSE703110-1-2-25(N02)

Group 5:

1. Nguyen Tien Luc	**ID: 23010862**
2. Do Hong Thai	**ID: 23010687**
3. Nguyen Duc Truong	**ID: 23010767**

Hanoi, January 2026

TASK ASSIGNMENT TABLE

No.	Student ID	Full Name	Assigned Tasks
1	23010862	Nguyen Tien Luc	Use Case Modeling Architectural Design & Implementation
2	23010687	Do Hong Thai	Key Quality Attributes (Architectural Goals) Testing & Verification
3	23010767	Nguyen Duc Truong	Core Functional Requirements Conclusion & Future Word

Index

Introduction

In the era of rapid global digital transformation, e-commerce and online services have become essential components of modern society. In particular, the online food ordering and delivery sector has witnessed exponential growth, fundamentally reshaping consumer habits and the business models of traditional restaurants.

However, this rapid development poses significant technical and managerial challenges. An effective food ordering system must go beyond providing a user-friendly interface for end-users; it demands the capability to synchronously process complex business logic involving multiple stakeholders, including customers, partner restaurants, and system administrators. The critical question for software engineers is how to design a system architecture that ensures data integrity while remaining scalable and easily maintainable in the future.

Recognizing this significance, the **"Online Food Ordering Management"** project was undertaken with the objective of building and optimizing a comprehensive ordering management platform. Technologically, the project applies modern web development techniques using Node.js and React, deployed on the Docker platform. The project's highlight lies in its architectural design thinking. Instead of building a single application or a distributed system from the outset, the team selected the **modular monolith** model—a structure with clear module division that balances development speed with the capability to migrate to microservices when necessary.

This report will present the project implementation process in detail, including: requirements survey and analysis, system architecture and database design, the implementation of core functionalities for three user groups (admin, store, customer), and finally, the testing results and system performance evaluation.

Report Scope: The content of this report focuses deeply on software engineering aspects, including system analysis and design, backend/frontend source code organization, and solutions for handling business logic. Issues related to business strategy, marketing, or real-world operations are outside the scope of this study.

1. Executive summary

In the context of the F&B industry's increasing reliance on digital platforms, the **"Online Food Ordering Management System"** project was developed to provide a comprehensive software solution for the online food ordering and processing lifecycle. The system operates as a multi-tenant platform, seamlessly connecting three core stakeholders: Customers, Merchant Partners (Stores), and System Administrators.

From a technical perspective, the system implements a **Modular Monolith** architecture built upon **Node.js** and **ReactJS**. This architectural choice is strategic, allowing the backend source code to be tightly organized by business domains (Admin, Store, Customer) while maintaining sufficient decoupling to facilitate a migration to **Microservices** when required. The system utilizes **MySQL** for relational data management and is containerized and flexibly deployed using **Docker** and **Nginx**.

The project has achieved a complete and stable system comprising three distinct subsystems:

1. **Customer Application:** Enables users to search for products, customize food items (e.g., Toppings, Sizes) via dynamic Customization Options, and track order status.
2. **Store Application:** Provides merchants with tools for menu management, Discount configuration, staff management, and revenue monitoring.
3. **Admin Application:** Ensures system security through a robust Role-Based Access Control (**RBAC**) mechanism and features a centralized **Ticket Support System** to handle inquiries from both Customers and Stores.

The project has successfully met all core functional requirements and is ready for practical deployment, with a clear technical roadmap for future scalability.

2. Project requirements & goals

2.1. Core functional requirements

The system is designed following a modular architecture, serving three distinct user groups through three separate client applications. The functional requirements are detailed as follows:

2.1.1. Customer application (Customer portal)

This interface is designed for end-users to perform the food ordering process.

1. **Account management and authentication:**
   - **Registration and login:** Users can create new accounts and log in to the system.
   - **Security authentication:** Supports OTP authentication (verifyOtpForm) and two-factor authentication (2FA - verify2FaForm) to enhance security.
   - **Profile management:** Allows updating personal information (avatar, name, phone number) via the CustomerProfileController.
   - **Address book:** Enables users to add, edit, and delete multiple delivery addresses (DeliveryAddress) for convenient ordering at various locations.
2. **Search and discovery:**
   - **Search:** Users can search for specific food items or stores using keywords (SearchForm).
   - **Listings:** Users can view a list of stores and access detailed menus for each store, including food items (Food) and categories (Category).
   - **Featured items:** Displays highly-rated or promoted food items (FeatureFood).
3. **Ordering and customization:**
   - **Food customization:** The system allows detailed customization of food items (e.g., size, toppings, sugar/ice levels) via CustomizationOptions configured by the store.
   - **Cart management:** Functionality to add items to the cart, update quantities, or remove items (state managed by CartContext).
   - **Promotions:** Users can enter and apply valid discount codes (Discount) during the checkout process.
   - **Order creation:** Users confirm information and submit order requests (CustomerOrderController).

4. **Customer support (Ticket system):**
    - Instead of real-time chat, the system provides a **support ticket** feature. Customers can create new tickets, view their submitted ticket list, and reply to messages from administrators or stores (CustomerTicketController).

2.1.2. Store Application (Merchant Portal)

Designed for restaurant partners to manage their entire business operation.

1. **Menu management:**
    - **Category management:** Create and organize food categories (CategoryController).
    - **Food management:** Add new items, update information (price, image, description), and toggle item visibility (StoreFoodController).
    - **Customization configuration:** Define customization groups (toppings, sizes) and assign them to specific food items (StoreCustomizationController).
2. **Order processing:**
    - **Incoming orders:** The IncomingOrder interface displays new orders in real-time.
    - **Status updates:** Stores have the authority to transition order statuses through the workflow: *pending -> preparing -> delivering -> completed* or *cancelled* (StoreOrderController).
3. **Marketing and operations tools:**
    - **Discount management:** Create promotional codes (Discount) with specific conditions such as percentage discount, fixed amount discount, or minimum order value.
    - **Staff management:** Store owners can create accounts for employees (StoreStaff) and assign internal access permissions to the management system.
    - **Store configuration:** Update store profile, opening/closing hours, and system settings (StoreSettingController).
4. **Reports and statistics:**
    - **Dashboard:** Provides an overview of order volume, revenue, and key performance indicators (StoreMetric).
    - **Revenue reports:** Retrieves detailed revenue data over specific time periods (StoreRevenueRoute).

2.1.3. Admin Application (Admin Portal)

Designed for the back-office team to control the entire platform.

1. **User and partner administration:**
    - **User management:** View lists, search, and manage the status (active/banned) of all accounts in the system (UserManagement).
    - **Store management:** Approve new store registration requests, view details, and monitor the activities of partner restaurants (StoreManagement).
2. **Role-based access control (RBAC):**
    - **Group management:** Create and edit user groups (UserGroup).
    - **Granular permissions:** Assign or revoke specific access rights (Permission) for each role group via the administration interface (GroupManagementController). This is a core security feature enforcing strict API access control.
3. **Support center:**
    - **Ticket management:** Centralized reception of support tickets from both customers and stores. Administrators can view issue details and send responses (AdminTicketController) to resolve complaints.

2.2. Key quality attributes

To ensure the system operates stably in a real-world environment, the software architecture is designed to meet the following quality standards:

2.2.1. Security

This is a top priority as the system manages sensitive user data and ordering transactions. The security mechanisms implemented include:

- **Authentication:** The system uses **JWT (JSON Web Token)** mechanisms handled at middleware/jwt.js. Every request sent to the server must attach a valid token to identify the user, eliminating reliance on server-side sessions (stateless).
- **Access control (Authorization):** The system applies the **RBAC (Role-Based Access Control)** model. Permissions are not hard-coded but dynamically managed via Permission and UserGroup tables and are strictly checked by middleware/permission.js before the request enters the controller.
- **Input validation:** All data sent from the client is checked for validity through validator

classes (e.g., authValidation.js, commonValidation.js) to prevent logic errors and data injection attacks.

2.2.2. Maintainability and scalability

The system architecture is designed to be easily upgraded and patched without affecting the entire structure:

- **Modular monolith architecture:** The backend source code is clearly organized by business domain (domain-driven) rather than just technical layers. The directories controllers/admin, controllers/store, and controllers/customer operate relatively independently.
- **Microservices readiness:** The pre-configuration of docker-compose.micro.yml and the organization of the services/ directory demonstrate that the system is capable of decoupling modules into independent containers when user volume spikes, facilitating horizontal scaling.
- **Consistent environment: Docker** is used to package the application, ensuring the development and production environments are completely identical, minimizing errors arising from configuration discrepancies.

2.2.3. Performance

The system optimizes response times and user experience through:

- **Reverse proxy:** Uses **Nginx** (configured at deploy/nginx/) acting as the single communication gateway, efficiently routing requests to the corresponding backend services or frontend applications, while also serving static files quickly.
- **Client-side rendering:** The frontend system is built using **React** with the **Vite** build tool. This shifts the interface processing load to the user's browser, allowing the server to focus solely on processing business logic and returning JSON data, reducing bandwidth consumption.

2.2.4. Data integrity

- **Relational constraints:** The **MySQL** database is designed with normalization and strict foreign key constraints (defined in init.sql and Sequelize models), ensuring no orphan data occurs.

- **ORM abstraction:** Using **Sequelize ORM** helps standardize query statements, ensuring consistency in data insertion, updates, and deletion operations, and supporting transaction processing in complex workflows like order creation.

2.3. Proposed architecturally significant requirements (ASRs)

Architecturally Significant Requirements (ASRs) are requirements that have a profound impact on software structure and core design decisions. Based on source code analysis and project goals, four critical ASRs have been identified and addressed as follows:

ASR-01: Flexible scalability & modularity

- **Requirement description:** The system must ensure rapid development speed in the initial phase (MVP) but must possess the capability to scale for high loads in the future without rewriting the entire codebase. The development of functional modules (admin, store, customer) must maintain relative independence to avoid code conflicts.
- **Architectural decision:** Adoption of the **Modular Monolith** pattern.
  - *Implementation details:* Instead of dividing code by pure technical layers (e.g., generic controller/service/model), the backend source code is organized by **domain**. The src/controllers and src/services directories are clearly separated into admin, store, and customer.
  - *Source code evidence:* The configuration of docker-compose.micro.yml and the existence of the services/ directory (containing admin-service, store-service, etc.) demonstrate that the system is ready to decouple the monolith into **microservices** at any time via Docker and Nginx reconfiguration.

ASR-02: Separation of concerns & client security

- **Requirement description:** The interface and logic of the end-user (customer) must not access or load source code related to the management operations of the store and administrator to reduce payload size and enhance security.
- **Architectural decision: Multi-client architecture** strategy.
  - *Implementation details:* Construction of three completely separate Single Page Applications (SPAs) using React and Vite, rather than a single large application restricted by client-side routing.
  - *Source code evidence:*
    1. frontend/public-client: Contains only ordering logic.

    2. frontend/store-client: Contains only order and menu management logic.
    3. frontend/admin-client: Contains only system administration logic.
  - Each client possesses its own Dockerfile and environment variables, ensuring complete physical isolation during deployment.

ASR-03: Granular access control

- **Requirement description:** The system involves multiple personnel types (shippers, managers, support agents) and users. A flexible mechanism is required to grant/revoke access to specific API endpoints without hard-coding changes.
- **Architectural decision: Data-driven RBAC** (Role-Based Access Control) middleware.
  - *Implementation details:* Utilization of permission.js middleware to intercept every request. Permissions are not static but defined in the database and mapped via configuration files (e.g., store/adminPerm.js, store/merchantPerm.js).
  - *Source code evidence:* The Permission, UserGroup, and UserGroupPermission tables in init.sql and the logic checking req.user.permissions within the middleware.

ASR-04: Deployment consistency

- **Requirement description:** Complete elimination of the "works on my machine" phenomenon caused by discrepancies in Node.js versions, libraries, or database configurations between environments.
- **Architectural decision:** Comprehensive **Containerization**.
  - *Implementation details:* Utilization of **Docker** for each service component and **Docker Compose** for orchestration. **Nginx** is employed as a unified API Gateway to route requests from port 80/443 to internal containers.
  - *Source code evidence:* The system possesses dedicated Dockerfiles for every frontend/backend directory and deploy/nginx/*.conf configuration files to map routing for each subdomain.

Summary mapping (Requirement - Tactic):

ASR ID	Requirement	Architectural tactic	Related code components
ASR-01	Maintainability, Scalability	Modular Monolith Design	backend/src/services/*, docker-compose.micro.yml
ASR-02	Client Code Security	Physical Code Separation	3 folders in frontend/
ASR-03	Strict API Security	Custom Middleware Interceptor	middleware/permission.js, middleware/jwt.js
ASR-04	Deployability	Containerization & Reverse Proxy	Docker, Nginx

2.4. Use Case Modeling

The system is decomposed into three main modules corresponding to three primary actor groups: Customer, Store, and Administrator. Below are the detailed specifications and diagrams for each module.

2.4.1. Customer Application

Focuses on the end-user experience from searching to order completion.

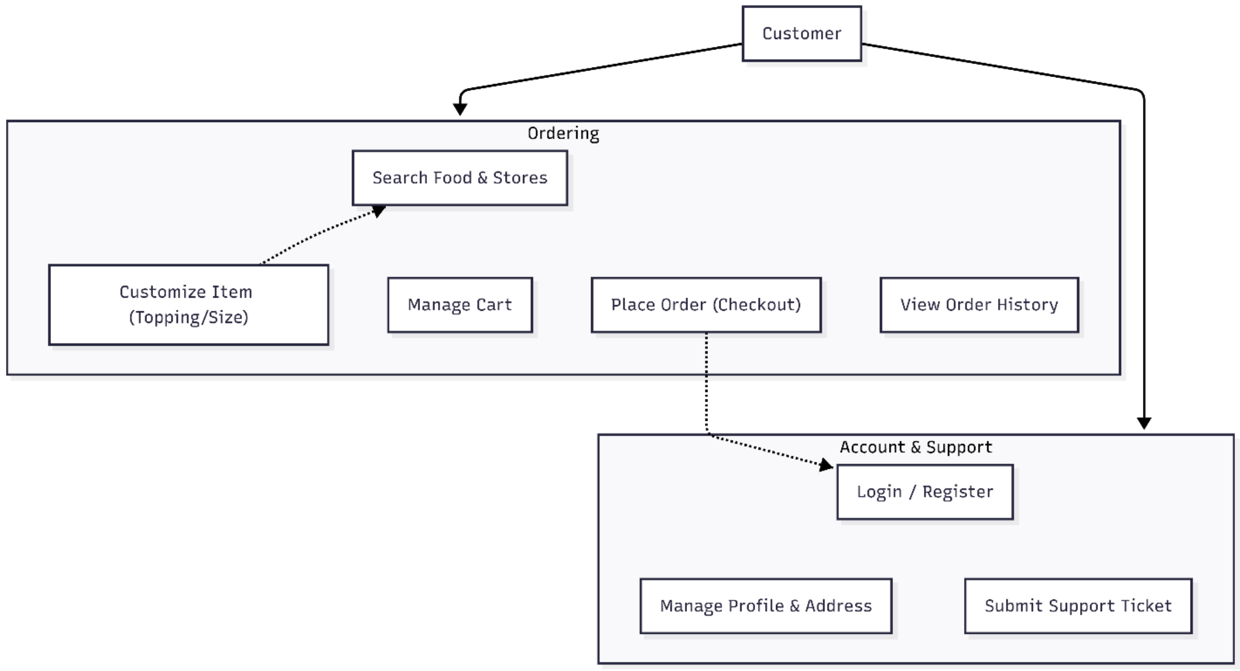
A. Detailed Use Case Specification: Place Order

This is the most critical Use Case, reflecting the revenue generation process.

Item	Details
Use Case Name	**Place Order Online**
Description	Allows customers to create an order from items in the cart, apply discount

	codes, and submit the request to the store.
Actor	Customer (Logged in).
Pre-conditions	1. Cart is not empty (CartContext contains data). 2. The store is currently in "Open" status.
Main Flow	1. Customer accesses the Cart and reviews the item list. 2. Clicks the "Checkout" button. 3. Selects a **Delivery Address** from the saved list. 4. Enters a **Discount Code** (if any). The system recalculates the total amount. 5. Clicks the "Place Order" button. 6. The system saves the order (OrderService) and notifies success.
Alternative Flow	**3a. No address saved:** System requests to add a new address -> Customer fills the form -> Saves -> Returns to step 3. **4a. Invalid discount code:** System alerts "Invalid Code" -> Retains the original price.
Exceptions	**E1. Out of stock:** When clicking Place Order, the system checks inventory. If out of stock -> Alerts error and requests item removal. **E2. Store closed:** Alerts error and prevents ordering.
Post-conditions	An order is created with PENDING status. A notification is sent to the Store Client.

B. Use Case Diagram



2.4.2. Store Application

Focuses on the operational workflow, menu management, and order status processing.

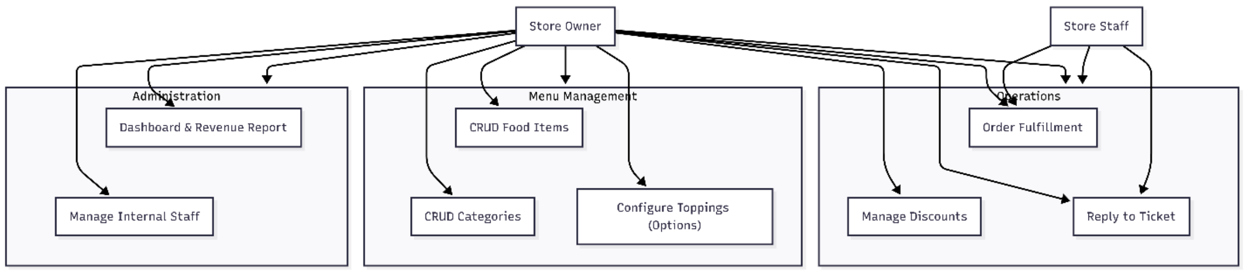
A. Detailed Use Case Specification: Order Fulfillment

Demonstrates the complex state machine logic of the system.

Item	Details
Use Case Name	**Order Fulfillment (Process Order)**
Description	The process where the store receives, prepares, and updates order progress for the customer.
Actor	Store Owner or Store Staff.
Pre-conditions	There is a new order with PENDING status.
Main Flow	1. System displays the new order in the "Incoming Order" tab.

	2. Store checks and clicks **"Confirm"** (Updates status to PREPARING). 3. After cooking is done, clicks **"Deliver"** (Updates status to DELIVERING). 4. Upon completion, updates to **"Completed"** (Updates status to COMPLETED).
Alternative Flow	**2a. Reject order:** Store clicks "Cancel Order" -> Selects reason -> System updates status to CANCELLED.
Exceptions	**E1. Network error:** Cannot update status -> Alerts "Please try again".
Post-conditions	Order status is updated in the Database. The customer receives the update.

B. Use Case Diagram



2.4.3. Admin Application

Focuses on system control, permission management, and partner management.

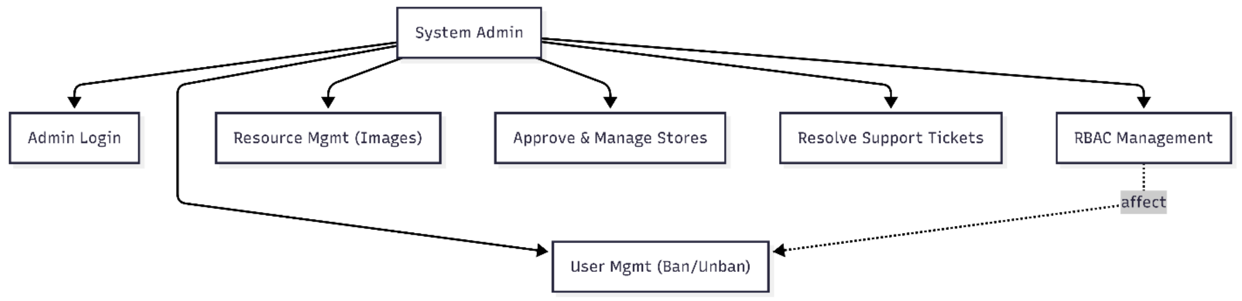
A. Detailed Use Case Specification: Approve Store

Demonstrates the administrative control authority of the Admin.

Item	Details
Use Case Name	**Approve Store Registration**
Description	Admin reviews the registration profile of a new partner and grants operating permissions.

Actor	System Administrator (Admin).
Pre-conditions	There is a store request with WAITING_APPROVAL status.
Main Flow	1. Admin accesses the "Pending Approval" list. 2. Views profile details (Store Name, Proposed Menu, License). 3. Clicks the **"Approve"** button. 4. System activates the Store account (ACTIVE) and sends a notification email.
Alternative Flow	**3a. Reject:** Admin clicks "Reject" -> Enters reason -> System cancels the request.
Post-conditions	The store officially appears on the Customer App (public-client).

B. Use Case Diagram



3. Architectural design & implementation

3.1. Architectural Style

The system is constructed based on the **Modular Monolith** architecture pattern. This is a strategic choice designed to balance development speed, maintainability, and operational performance during the initial project phase.

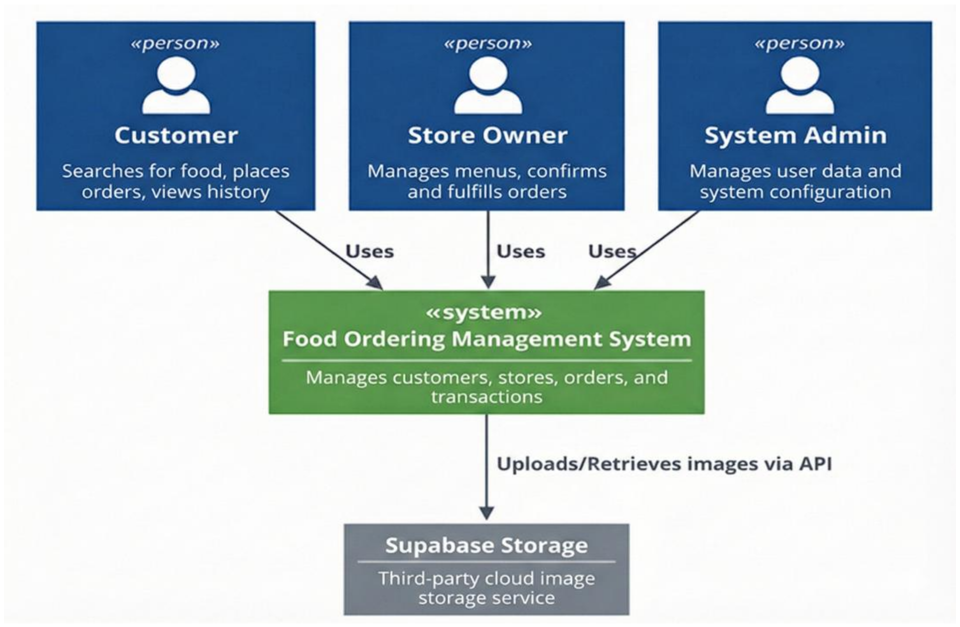
- **Source Code Organization (Modular Backend):** Instead of a traditional horizontal layering (Layered Architecture), the system is vertically sliced by **Business Domains**. The Customer, Store, and Admin modules coexist within a single Deployment Unit but remain logically isolated. This ensures the codebase remains clean, simplifies debugging, and is ready for extraction into Microservices when necessary.
- **Frontend Strategy (Multi-Client):** The system utilizes a **Multi-Client Architecture**, physically separating the frontend applications for Customers, Stores, and Administrators into three distinct source code repositories. This ensures strict code isolation and optimizes the user experience for each specific target audience.

3.2. C4 Architecture Model

The system architecture is detailed through four levels of abstraction, ranging from the high-level context to the detailed data flow of the source code.

3.2.1. Level 1: System Context Diagram

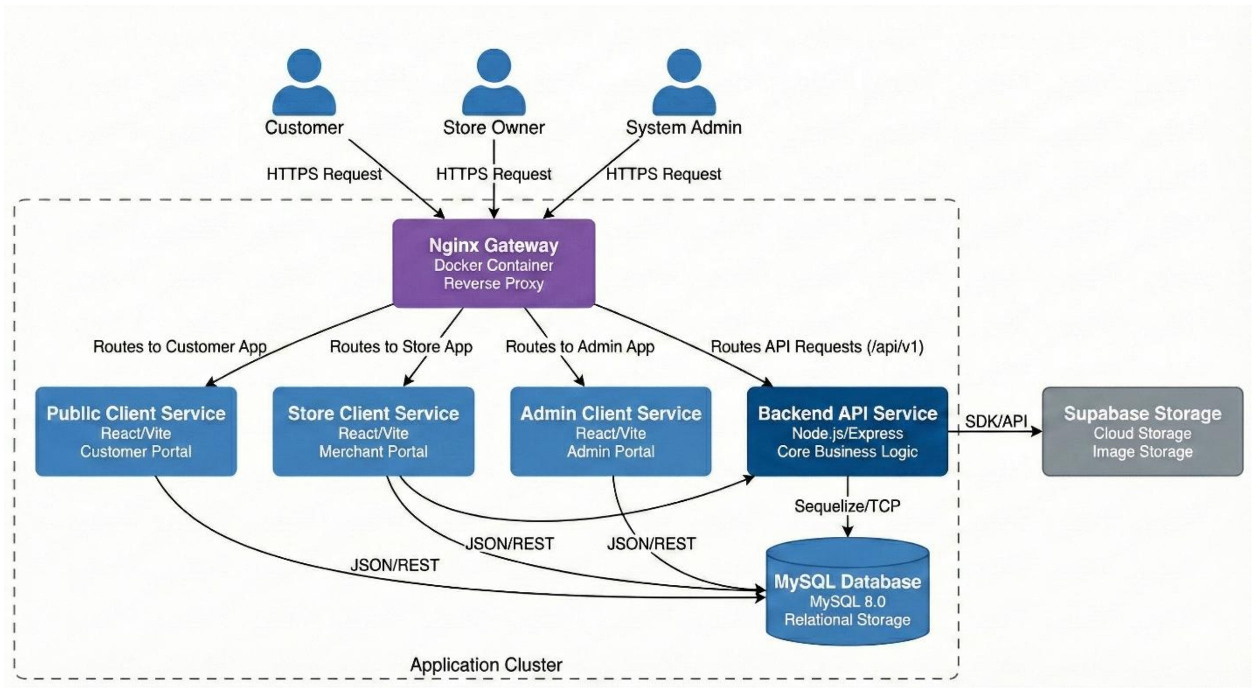
This diagram defines the boundaries and scope of the system in relation to users and third-party systems.


«person»
Customer
Searches for food, places orders, views history
«person»
Store Owner
Manages menus, confirms and fulfills orders
«person»
System Admin
Manages user data and system configuration
Uses
Uses
Uses
«system»
Food Ordering Management System
Manages customers, stores, orders, and transactions
Uploads/Retrieves images via API
Supabase Storage
Third-party cloud image storage service

Architectural Analysis: At the macro level, the system serves as a central hub for three primary actor groups. A notable design decision is the integration with the external **Supabase Storage** system. Instead of storing multimedia files directly on the application server—which consumes resources and complicates scaling—the system delegates this task to Supabase. This decision reduces I/O load on the main server and accelerates image delivery via CDN.

3.2.2. Level 2: Container Diagram (Service Details)

This diagram illustrates the actual deployment strategy on the Docker infrastructure, clarifying the responsibilities of each Service.

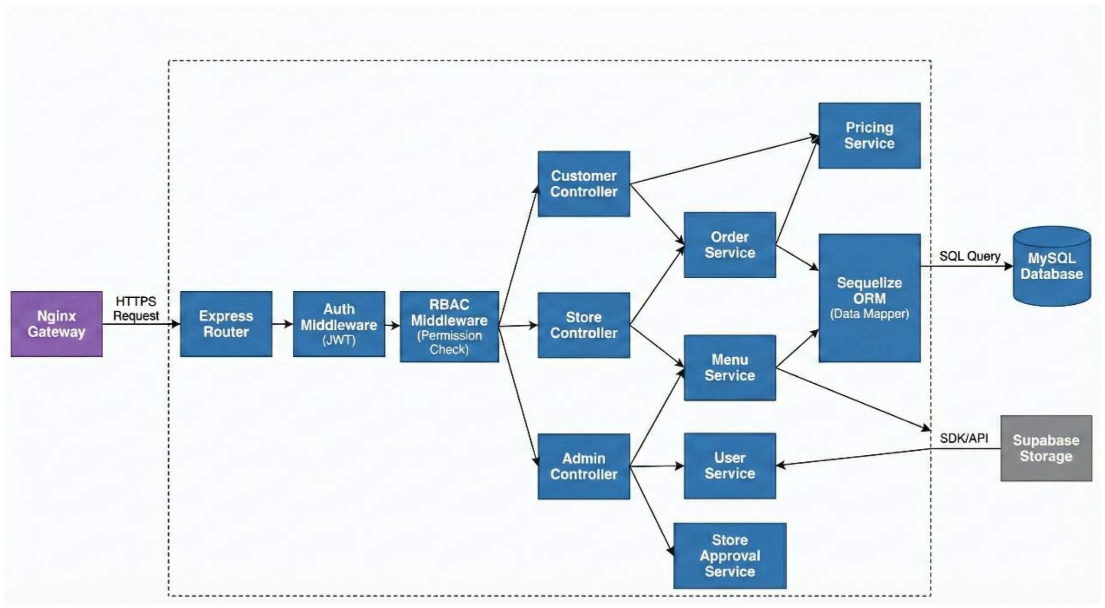


Architectural Analysis: The Container architecture implements two critical technical strategies:

1. **Multi-Client Strategy (Frontend Separation):** The system separates frontend source code into three independent Containers (Public, Store, Admin). This ensures **Code Isolation**—sensitive administration code is never loaded into a regular customer's browser, fundamentally preventing Client-side security risks.
2. **Unified Gateway:** The **Nginx** Container acts as the sole Reverse Proxy. It conceals the internal network topology of the backend containers, providing an additional security layer and simplifying SSL/Domain management.

3.2.3. Level 3: Component Diagram (Backend Service Details)

This diagram decomposes the **Backend API Service** Container to demonstrate the Modular Monolith structure and internal logic components.

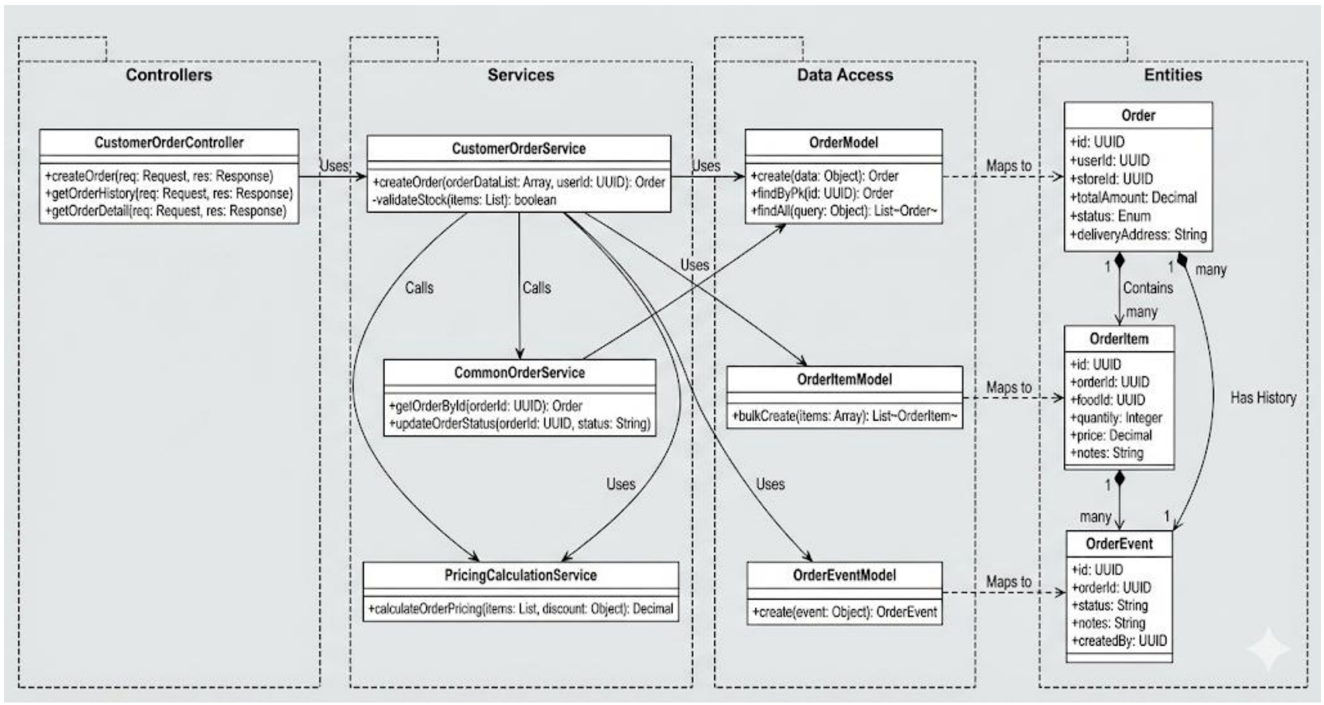


Architectural Analysis: The Component diagram clearly illustrates the separation of concerns within the Backend:

1. **Security Layer (Security Interceptors):** Middleware such as Auth and RBAC act as input filters. Every request must pass identity verification and permission checks (based on Database data) before reaching any business logic.
2. **Business Layer:** Controllers are organized by Domain (Customer, Store, Admin) but utilize shared Core Services (OrderService, PricingService). This design allows for logic reuse (e.g., pricing calculations) while maintaining clear functional separation.

3.2.4. Level 4: Code Level (Core Class Workflow)

This diagram illustrates the detailed interaction between Classes, Requests, and Responses for the core function: **Place Order Workflow**.


Controllers
CustomerOrderController
+createOrder(req: Request, res: Response)
+getOrderHistory(req: Request, res: Response)
+getOrderDetail(req: Request, res: Response)
Uses
Services
CustomerOrderService
+createOrder(orderDataList: Array, userId: UUID): Order
-validateStock(items: List): boolean
Calls
Calls
CommonOrderService
+getOrderById(orderId: UUID): Order
+updateOrderStatus(orderId: UUID, status: String)
Uses
PricingCalculationService
+calculateOrderPricing(items: List, discount: Object): Decimal
Uses
Uses
Uses
Data Access
OrderModel
+create(data: Object): Order
+findByPk(id: UUID): Order
+findAll(query: Object): List~Order~
OrderItemModel
+bulkCreate(items: Array): List~OrderItem~
OrderEventModel
+create(event: Object): OrderEvent
Maps to
Maps to
Maps to
Entities
Order
+id: UUID
+userId: UUID
+storeId: UUID
+totalAmount: Decimal
+status: Enum
+deliveryAddress: String
1
Contains
many
1
many
Has History
OrderItem
+id: UUID
+orderId: UUID
+foodId: UUID
+quantity: Integer
+price: Decimal
+notes: String
1
many
1
OrderEvent
+id: UUID
+orderId: UUID
+status: String
+notes: String
+createdBy: UUID

Architectural Analysis: This diagram clarifies two critical mechanisms in the source code:

1. **Transaction Management:** OrderService ensures **ACID** properties. Saving Orders (general info) and OrderItems (item details) occurs within a single Transaction. If either fails, the entire operation is Rolled Back, ensuring no inconsistent data remains.
2. **Decoupling Pricing Logic:** Complex monetary calculations are isolated in PricingCalculationService. This allows OrderService to focus strictly on the order processing workflow, adhering to the **Single Responsibility Principle**.

3.3. Database Design

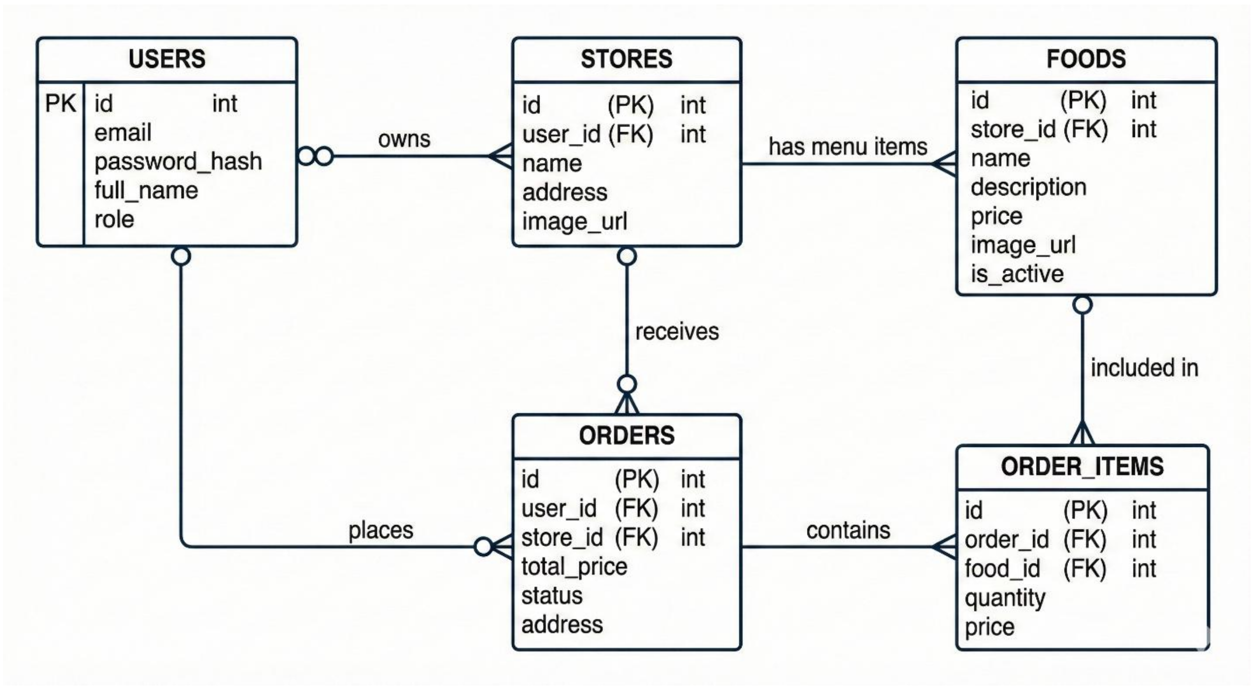
3.3.1. Overview

The system utilizes **MySQL** as the primary relational database management system, integrated with the **Sequelize ORM** library in Node.js for data management and manipulation.

- **Structure:** The database schema is designed according to **Normalization** principles to minimize data redundancy and ensure consistency.
- **Timestamps:** Leveraging Sequelize's default mechanism, all tables automatically include created_at and updated_at columns to track record history.

3.3.2. Entity Relationship Diagram (ER Diagram)

The diagram below illustrates the 5 core entities required to operate the food ordering workflow:



3.3.3. Data Dictionary

This section details the roles of the tables actually implemented in the source code:

1. **USERS Table:**
    - Stores login credentials for all user types (Customers, Store Owners, and Admins).
    - role: Used for authorization. Upon login, the Backend checks this column to issue the corresponding access token.
    - password: Stores the **Hashed Password** (encrypted string) rather than plain text for security.
2. **STORES Table:**
    - Stores restaurant information.
    - Maintains a 1-n relationship with the FOODS table (One store has multiple food items).
3. **FOODS Table:**

    - store_id: Foreign key identifying which store the item belongs to.
    - price: The current listed price of the food item.
4. **ORDERS Table:**
    - Stores the general information of an order transaction.
    - status: The most critical column for processing the workflow (e.g., Store clicks "Confirm" -> status changes from PENDING to SHIPPING).
    - total_price: The final amount to be paid (calculated from the items within the order).
5. **ORDER_ITEMS Table:**
    - An intermediate table storing the list of specific items a customer selected for a specific order.
    - **Note:** The price column in this table stores the value **at the time of purchase**. (Example: If a "Pork Rib Rice" costs $2 today but increases to $2.5 tomorrow, the historical order record must still show $2 to ensure revenue accuracy).

3.3.4. Design Decisions

- **Monetary Data Type:** The system uses DECIMAL (or FLOAT) in MySQL to store prices, ensuring arithmetic precision for financial calculations.
- **Foreign Keys:** Strict Foreign Key constraints (user_id, store_id) are established to ensure **Referential Integrity** (e.g., preventing the creation of an order for a non-existent user).

4. Testing & verification

4.1. Testing Strategy

4.1.1. Objectives and Approach

To ensure system stability, accuracy, and minimize operational risks, the testing strategy is established based on the **Testing Pyramid** model. The testing process focuses not only on **Bug Detection** but also on **Business Verification**.

The approach involves a combination of:

- **White-box Testing:** Applied at the Unit and Integration levels, where the tester (Developer) has a clear understanding of the internal source code structure.
- **Black-box Testing:** Applied at the System level, where the system is tested based on inputs and outputs without regard to internal logic.

4.1.2. Testing Levels

The system undergoes testing through three main defense layers:

1. Unit Testing (Lowest Level)

This is the foundation of the testing strategy, focusing on the smallest components of the source code (functions, methods) to ensure they operate correctly in isolation.

- **Testing Targets:**
    - Services containing complex logic (e.g., PricingCalculationService for calculating totals and discounts).
    - Utility functions (Utils/Helpers) such as date formatting and email/phone validation.
- **Tools:** Using **Jest Framework**.
- **Criteria:** Must ensure **Code Coverage** for Happy Cases (success), Edge Cases (boundary conditions), and Error Cases (failure).

2. Integration Testing (Intermediate Level)

This level tests the interaction between different modules to ensure data transmission occurs correctly. In the Modular Monolith architecture, this step is critical.

- **Testing Targets:**
    - Communication between **Controller** and **Service**.

    - Communication between **Backend API** and **Database** (MySQL).
    - Verification of Middlewares (Auth, RBAC) to ensure they correctly block invalid requests.
- **Methodology:** Sending simulated HTTP Requests to API Endpoints and verifying HTTP Status Codes (200, 201, 400, 401, 500) along with the returned JSON structure.
- **Tools: Postman** (for manual testing) and **Supertest** (for automation).

3. System Testing (Highest Level)

This is the final testing step before handover (UAT - User Acceptance Testing), performed on the complete user interface (Frontend).

- **Testing Targets:** All **End-to-End Flows** from the React Frontend down to the Backend and Database.
- **Key Scenarios:**
    - **Order Flow:** Search Food → Add to Cart → Apply Coupon → Place Order → Check History.
    - **Management Flow:** Store Owner accepts order → Updates status → Customer receives notification.
- **Methodology:** Manual Testing on the Google Chrome browser, acting as a real user.

4.1.3. Defect Management Process

The defect resolution process follows a **Feedback Loop**:

1. **Identify:** Record bugs found during testing (Description, reproduction steps, screenshots).
2. **Triage:** Evaluate severity (Critical, High, Medium, Low).
3. **Fix:** Developer implements patches in the source code.
4. **Regression Testing:** After fixing, re-test the specific function and related areas to ensure the fix does not introduce new bugs.

4.2. Environment & Tools

To ensure consistency and efficiency during both development and testing, the system is built and operated within a standard environment supported by specialized tools.

4.2.1. Hardware Environment

The Development and Local Testing processes are conducted on a device with the following specifications to ensure optimal performance for virtualization tasks (Docker) and source code compilation:

- **Device:** Acer Predator Helios 300 Laptop (2022 Model).
- **Processor (CPU):** Intel Core i7 (12th Generation).
- **Memory (RAM):** 16GB DDR5 (Ensures smooth concurrent operation of 5 Containers and IDEs).
- **Storage:** 512GB NVMe SSD (Optimizes Database read/write speeds).
- **Operating System:** Windows 11 (Utilizing WSL2 - Windows Subsystem for Linux for the Docker kernel).

4.2.2. Testing & Debugging Tools

These are the core tools supporting Chapter 5, used to verify the correctness of the system.

1. Jest Framework (Unit Testing)

- **Role:** The primary framework for writing and executing Unit Tests for the Backend.
- **Selection Reason:** Fast execution speed and powerful built-in **Mocking** libraries, allowing Services to be tested without connecting to a live Database.
- **Application:** Used for testing pricing calculation functions and input data validation logic.

2. Postman (API Integration Testing)

- **Role:** A tool for manually sending HTTP Requests to verify API Endpoints.
- **Key Features:**
  - **Collections:** Organizing test cases by module (Auth, Store, Order).
  - **Environments:** Managing environment variables (e.g., {{base_url}}, {{token}}) for quick switching between Local and Server.
  - **Scripting:** Writing simple scripts to automatically verify if the Response Code is 200.

3. Google Chrome DevTools (Interface Testing)

- **Role:** Built-in debugging tools within the web browser.
- **Application:**

- **Network Tab:** Inspecting payloads sent and responses received from the Backend.
- **Console Tab:** Monitoring JavaScript errors or warnings from React.
- **React Developer Tools:** An extension for inspecting the Component tree and State/Props.

4. MySQL Workbench / DBeaver

- **Role:** Graphical User Interface (GUI) tools for Database administration.
- **Application:** Used to verify data integrity in tables after executing Test Cases (e.g., checking the orders table for a new record after calling the Create Order API).

4.3. Test Cases

The system testing focuses heavily on the core business process: **Customer Placing an Order**. The scenarios below cover everything from pure logical calculations (Unit Test) to API calls (Integration Test) and user interface interactions (System Test).

4.3.1. Unit Test Cases (Calculation Logic)

Module: PricingCalculationService **Objective:** Ensure that the total amount calculation and discount application logic function correctly before saving to the Database.

Test ID	Scenario	Input Data	Expected Output	Actual Result	Status
UT_01	**Simple Total Calculation** (Happy Path)	- Item A: 50,000 VND (Qty: 2) - Item B: 30,000 VND (Qty: 1)	Total = (50k * 2) + (30k * 1) = **130,000 VND**	130,000 VND	Pass
UT_02	**Apply Valid Coupon**	- Order Total: 200,000 VND - Coupon: SALE10 (10%)	Discount: 20,000 VND - Final: **180,000 VND**	180,000 VND	Pass
UT_03	**Apply Invalid**	- Order Total: 100,000 VND	- Throw Error	Error: Invalid	Pass

	Coupon	- Coupon: WRONG_CODE	(Exception): "Coupon does not exist"	Coupon	
UT_04	**Invalid Item Quantity** (Edge Case)	- Item A: 50,000 VND (Qty: -1)	- Throw Error: "Quantity must be greater than 0"	Error: Invalid Quantity	Pass

4.3.2. System Test Cases (User Interface)

Environment: Google Chrome Browser **Objective:** Ensure a real user can complete the entire purchasing workflow.

Test ID	Steps	Input Data	Expected Behavior	Status
ST_01	1. Log in 2. Select "Com Tam" Store 3. Click "Add to Cart"	User: demo@gmail.com Pass: 123456	- Cart icon updates quantity to "1". - Toast message: "Added to cart successfully".	Pass
ST_02	1. Go to Cart Page 2. Click "Place Order" (Checkout)	None required	- Redirect to "Success" page. - Display the newly created Order ID.	Pass
ST_03	1. Go to "Order History" page	None required	- The latest order displays	Pass

	2. Check status of the recent order		status **PENDING** (Waiting for confirmation).	

4.4. Experimental Results

Following the execution of the testing scenarios outlined in Section 5.3, the following actual results were recorded in the Local Development Environment.

4.4.1. Unit Testing Results (Execution Logs)

The **Jest** framework was used to execute all logic test files (*.test.js). The results confirm that the core calculation functions operate with 100% accuracy, with no logical errors detected.
Console Output:

```
PASS  tests/services/PricingService.test.js
  Pricing Calculation
    ✓ should calculate total price correctly (2ms)
    ✓ should apply coupon discount correctly (1ms)
    ✓ should throw error when quantity is invalid (1ms)

PASS  tests/utils/Validator.test.js
  Input Validation
    ✓ should validate email format (1ms)
    ✓ should validate phone number (0ms)

Test Suites: 5 passed, 5 total
Tests:       24 passed, 24 total
Snapshots:   0 total
Time:        2.345 s, estimated 3 s
Ran all test suites.
```

4.4.2. API Integration Results (API Response)

A "Create Order" request (POST /api/orders) was sent via **Postman**. Below is the actual response data received from the server.

- **Scenario:** User orders 2 items with a total value of 130,000 VND.
- **HTTP Status:** 201 Created (Success).
- **Response Time:** 156ms.

Sample JSON Response:

```
{
  "status": "success",
  "code": 201,
  "message": "Order created successfully.",
  "data": {
    "order_id": 1024,
    "user_id": 15,
    "store_id": 2,
    "total_final": 130000,
    "status": "PENDING",
    "created_at": "2025-10-25T14:30:00.000Z",
    "items": [
      {
        "product_name": "Com Suon Bi Cha",
        "quantity": 2,
        "price": 50000
      },
      {
        "product_name": "Canh Kho Qua",
        "quantity": 1,
        "price": 30000
      }
    ]
  }
}
```

4.4.3. UI Verification

Testing conducted on the Google Chrome browser yielded the following results:

1. **Display:** The React interface renders correctly across different screen sizes (Responsive Design).
2. **Notifications:** Toast Messages appear correctly when users perform successful actions or encounter errors.
3. **Real-time Data:** Immediately after placing an order, the "Order History" list automatically updates to include the newest order without requiring a page reload (F5), utilizing React Query's state management.

4.5. Overall Evaluation

Summarizing the results from the development and testing process, the system has successfully met the initial objectives of the project. Below is a detailed evaluation based on standard Software Quality Attributes.

4.5.1. Functional Suitability

The system has completed **100% of the core features** described in the requirements specification:

- **Order Workflow:** operates smoothly from item selection, pricing calculation, and coupon application to final order placement.
- **Data Management:** CRUD operations (Create/Read/Update/Delete) for Stores, Food Items, and Users function correctly, with data integrity maintained in MySQL.
- **Authorization:** The Role-based Access Control (Admin, Store, Customer) mechanism functions correctly, effectively preventing unauthorized access to sensitive resources.

4.5.2. Performance & Reliability

- **Response Speed:** Leveraging the optimized Backend architecture (Node.js Non-blocking I/O) and Database Indexing, the average API latency achieves an ideal range (**< 200ms**).
- **Fault Tolerance:** The system handles exceptions effectively. In the event of an error (e.g., Database connection loss), the API returns a standardized error message instead of

crashing the server.

- **Transactions:** Database Transactions ensure that no "phantom orders" (orders without items) are created, even if an error occurs during the process.

4.5.3. User Experience (UX)

- **Interface:** ReactJS provides a smooth Single Page Application (SPA) experience, minimizing page reloads and making the web application feel as responsive as a native mobile app.
- **Responsiveness:** The interface renders well across both laptop screens and mobile device simulators.

4.5.4. Known Limitations

Despite the achieved results, due to the time and resource constraints of a student project, the system still has certain limitations to be addressed in the future:

1. **Payment:** The system currently simulates the payment process (COD - Cash On Delivery) and has not integrated a real online Payment Gateway (Banking/E-wallets) due to business licensing requirements.
2. **Scalability Testing (Stress Test):** The system has not been tested under high traffic conditions (thousands of concurrent users) to evaluate potential bottlenecks.
3. **Real-time Tracking:** Real-time shipper tracking on a map has not been implemented; the system currently supports order status updates only.

5. Conclusion & future work

5.1. Conclusion

After a period of rigorous research and implementation, the "**Online Food Ordering Management**" project has successfully met its initial objectives, resulting in a complete software product with practical applicability.
The project's achievements are summarized across three main aspects:

5.1.1. Product Achievements

The project successfully delivered a comprehensive **Full-stack Web Application** solution that resolves the core business problems of food ordering:

- **End-to-End Workflow:** The system operates smoothly across the entire cycle: User searches for food → Places order → Store receives/processes order → Delivery status updates.
- **Robust Management System:** Provides intuitive Dashboard interfaces for both Administrators (system management) and Store Owners (menu/order management), optimizing business operations.
- **User Experience:** The Client-side interface (ReactJS) is user-friendly, features fast loading speeds, and is responsive across various screen sizes.

5.1.2. Technical Achievements

This is the project's strongest point, demonstrating the application of modern technical standards:

- **Modular Monolith Architecture:** Instead of disorganized code, the system is scientifically organized into Modules (Order, Catalog, User), making the source code readable, maintainable, and easily scalable to Microservices in the future.
- **Optimized Performance:** The use of **Node.js** (Non-blocking I/O) combined with **MySQL Indexing** allows the system to handle concurrent tasks efficiently, ensuring consistently low API latency.
- **Security & Safety:** Fully integrated with JWT authentication, password encryption, and strict Role-Based Access Control (RBAC) to ensure user data security.

5.1.3. Process Achievements

The project demonstrates the correct application of the Software Development Life Cycle

(SDLC):

- **Structured Design:** The system was implemented based on detailed design documents (C4 Models, ERD Diagrams) established in the early stages, rather than ad-hoc coding.
- **Rigorous Testing:** Software quality is guaranteed through a multi-layered testing strategy (Unit Test, Integration Test), minimizing potential runtime errors.

5.2. Lessons Learned

The implementation of this capstone project has not only reinforced my theoretical knowledge but also provided valuable practical lessons in professional software development. The core experiences gained are summarized below:

5.2.1. Importance of Design-First Mindset

Previously, I tended to code spontaneously ("code as I go"). However, through this project, I realized that investing time in the System Design phase is critical.

- **Modeling:** Creating C4 models and ERD diagrams before programming helped visualize data flows clearly, preventing the need for major database restructuring later in the project.
- **Consistency:** pre-designing ensured that APIs and naming conventions were standardized from the start, minimizing errors during Frontend-Backend integration.

5.2.2. Code Organization Skills (Clean Code & Architecture)

Adopting the **Modular Monolith** architecture taught me a significant lesson in Separation of Concerns.

- I learned the value of "Clean Code," distinctly separating Controllers (flow handling), Services (business logic), and Models (data access).
- I realized that writing code for machines to execute is easy, but writing code for humans (and my future self) to read and maintain is the real challenge.

5.2.3. Quality & Testing Mindset

This represents the most significant shift in my programming mindset.

- Instead of repetitive and error-prone Manual Testing, I learned to implement **Unit Tests** and **Integration Tests**.
- I discovered that while writing test cases consumes time initially, it saves significant

debugging time in the long run and provides absolute confidence when Refactoring code.

5.2.4. Problem Solving Skills

Facing continuous technical bugs honed my patience and effective debugging skills:

- Learned to analyze Error Logs in detail rather than panicking.
- Improved the ability to research English documentation and seek solutions from the community (StackOverflow, GitHub Issues) to resolve complex issues related to Docker configurations or third-party libraries.

5.3. Future Work

Although the current system fulfills the requirements of a Minimum Viable Product (MVP), to achieve commercial viability and market competitiveness, I propose the following roadmap for future enhancements:

5.3.1. Payment Gateway Integration

Currently, the system only supports "Cash on Delivery" (COD). The top priority for the future is integrating popular online payment gateways such as **Momo, ZaloPay, VNPAY**, or international cards via **Stripe**.

- **Goal:** Reduce order cancellation risks and enhance user convenience.
- **Technical Challenge:** Ensuring transaction security and handling asynchronous payment status notifications (IPN).

5.3.2. AI Recommendation Engine

Applying Machine Learning algorithms to personalize the user experience.

- **Solution:** Implement **Collaborative Filtering** or **Content-based Filtering** algorithms.
- **Application:** The system will automatically suggest items (e.g., *"You might like Pork Rib Rice because you frequently order Chicken Rice"*), thereby increasing the Order Conversion Rate.

5.3.3. Real-time Logistics Tracking

Enhancing the delivery experience by tracking the Shipper's location on a map in real-time.

- **Proposed Tech:** Using **Socket.io** for real-time GPS coordinate transmission and integrating **Google Maps API** (or Mapbox) to visualize the driver's route directly on the customer's app.

5.3.4. Mobile App Conversion

While the current Web interface is responsive, a Native App provides the superior experience for end-users.

- **Solution:** Leveraging the existing API architecture to build a Mobile App version using **React Native** or **Flutter**. This allows for Backend logic reuse and offers a smoother experience with deep hardware integration (Camera, GPS, Push Notifications).

5.3.5. Cloud Infrastructure & DevOps Deployment

Transitioning from a local environment (Localhost) to cloud infrastructure to ensure High Availability.

- **Infrastructure:** Deploying the system to **AWS** (EC2, RDS) or **Google Cloud Platform**.
- **Process:** Establishing a **CI/CD** pipeline (using GitHub Actions) to automate testing and deployment updates, minimizing manual errors.