The primary objective of the Café Order Management System is to maximize the operational efficiency of the café while simultaneously providing a modern, digitized customer experience. These goals are achieved through the collaboration of three main modules:

1.1 Customer-Focused Objectives (Customer Module)

- **Accelerate the Ordering Process:** To enable customers to instantly access the menu by scanning the **QR codes** placed on their tables and transmit their orders to the system without waiting.
- **Offer Flexible Payment Options:** To provide customers with the convenience of making **online payments** directly through their mobile devices while placing or after placing an order, thereby eliminating cashier waiting times.
- **Ease of Use:** To allow seamless menu navigation, adding customized notes, and tracking order status through a clean and fast user interface.

Operational Efficiency Objectives (POS/Manager and Kitchen Modules)

- **Efficient Kitchen Management (Kitchen Module):** To optimize the preparation and delivery process by ensuring that orders from the Customer and POS modules instantly drop into the KDS (Kitchen Display System) interface.
- **Eliminate Error Rate:** To eliminate errors in taking or communicating orders (miscommunication between kitchen/waitstaff) by ensuring orders are taken digitally and transmitted directly to the kitchen.
- **Enhance Staff Focus:** To ensure staff focus on value-added tasks such as order preparation, delivery, and customer service, rather than spending time on order taking and payment processing.

1.2 Management and Control Objectives (POS/Manager Module)

- **Flexible POS Integration:** To offer customers the option to pay **at the counter** (cash/card) alongside online payment, while managing all financial transactions consistently within a single POS module.
- **Comprehensive Reporting:** To provide detailed, real-time reports on sales, best-selling products, and inventory consumption, enabling data-driven business decisions.
- **Menu and Pricing Control:** To provide the ability to instantly update products, pricing, and stock status on the menu via the Manager module.

Backend Architectural Overview

The system is designed as a Modular Monolith (or a bounded microservices approach). This structure ensures that the business logic is separated into domains under the (internal/ directory, but operates within a single main application (cmd/api).

Directory	**Purpose and Responsibility**
cmd/api	**Main Entry Point:** Contains the application's entry point (main.go). It is responsible for initializing the HTTP server, defining the routers, and injecting dependencies between the *internal/* modules.
internal/	**Business Logic:** Contains the core business logic, the data access layer (repository), and the services. The use of the internal/ directory ensures that this code cannot be imported by external projects, aligning with Go standards.
pkg/	**Reusable Packages:** Stores general-purpose helper code (utilities, helpers, custom error handling) that might be used by multiple *internal/* modules or potentially by different future services.

2.1. Business Logic Modules (*internal/*)

Each subdirectory under *internal/* represents a distinct business module (domain) within the application. Every module is responsible for its own data models, services, and HTTP handlers.

Module	Functionality	Dependencies and Responsibilities
auth	Identity Management	Provides user/employee login, generates JWT (JSON Web Tokens), handles session termination, and provides authorization middleware.
inventory	Stock Management	Tracks stock levels for raw materials and final products. Manages stock deductions based on sales orders received from the order module.
menu	Menu Management	Manages CRUD (Create, Read, Update, Delete) operations for product information (price, description, category). Determines pricing and out-of-stock statuses.
order	Order Processing	Manages creating new orders (from Customer or POS), listing order items, and updating the order status (in coordination with payment and inventory) such as submitted, preparing, ready.
payment	Payment Transactions	Manages integration with online payment gateways. Records payment status, updates the payment status (success/failure), and notifies the order module.
report	Reporting and Analytics	Fetches data from the order, payment, and inventory modules to generate daily/monthly sales reports, inventory analyses, and Z-Report data for the Manager Module.
review	Customer Feedback	Provides functionality for recording and managing customer reviews and ratings for products or service.

2.2. *API_CONTRACT.MD* Versioning

Maintaining API contracts in separate Markdown files ensures clear communication between the frontend (React) and backend (Golang) teams.

- **API_CONTRACT.MD:** Represents the current (Production) API version.
- **API_CONTRACTV2.MD:** Indicates upcoming API changes, new endpoints, or data structure updates that are currently under development. This practice ensures future changes are documented proactively.

Frontend Architecture and Components (React / Vite)

3.1. The Frontend application is built using React (Vite-based) to ensure fast development and a modern user interface. The application is designed to handle three distinct user experiences—Customer, POS/Manager, and Kitchen Display System—within a single codebase.

Directory/File	Purpose	Description
src/	Source Code	Contains all React components, styling files, and application logic.
src/assets	Assets	Stores static media files such as logos, images, and fonts.
src/component	Shared Components	Small, independent, and often presentational (dumb) components reused across the application (e.g., Button, Modal, Input).
src/hooks	Custom Hooks	Encapsulates complex logic such as API call management, state management integration, or form validation using custom React Hooks.
src/types	TypeScript Types	Type definitions for data received from the Backend API (product, order, user) and for internal application state (if using TypeScript).
public/	Static Files	Static assets that are copied directly to the build output without being processed by Vite (e.g., $favicon.ico$).
nginx.conf	Deployment Configuration	Defines how Nginx, running inside the production Docker container, serves the static React files and how it proxies API requests (/api) to the backend service.
vite.config.js	Build Configuration	Configuration for the development server and settings for the production build output.

3.2. Application Modules (Routes / Pages)

The Frontend is segregated into separate page groups (routing) to serve different user roles effectively:

User Role	Key Pages (Routing)	Backend Dependency (Module)
Customer	/table/:qr_id, /menu, /checkout, /order/:id/track	menu, order, payment, review
POS/Manager	/login, /pos/home, /pos/reports, /pos/menu-editor	auth, order, payment, report, menu
Kitchen (KDS)	/kitchen/orders, /kitchen/archive	order (Relies on WebSocket or Polling for real-time status updates)

3.3. State Management and API Communication

State Management: Complex application state (e.g., shopping cart contents, active order list) is likely managed using a library like Redux, Zustand, or React Context.

API Communication: All communication is handled via custom hooks defined in $src/hooks (e.g., useFetchMenu, useSubmitOrder). These hooks ensure strict adherence to the **Golang Backend API Contracts** (API_CONTRACT.MD).

3.4. Containerization (Frontend)

The Frontend is containerized using Docker:

- **Dockerfile (Frontend):** In the development environment, it runs the development server using **Node.js**. In production, it builds the static React files and serves them using a high-performance web server like **Nginx**.
- **nginx.conf:** Plays a critical role in production by proxying API requests (typically prefixed with /api) received from the customer's browser to the correct Go backend service.

Docker Integration & Local Deployment Guide

4.1. Overview

The Cafe Management System utilizes **Docker** to ensure a consistent, portable, and isolated development environment. By containerizing the application, we eliminate "works on my machine" issues, ensuring that the Frontend (React), Backend (Go), and Database services interact seamlessly regardless of the host operating system.

The architecture follows a microservices approach where each component runs in its own container, orchestrated via **Docker Compose**.

4.2. Containerization Architecture

The system consists of three primary services defined within the container ecosystem:

4.2.1 Backend Service (Golang)

The backend is built using a **Multi-Stage Build** strategy to optimize the final image size and security.

- **Stage 1: The Builder**
  - **Base Image:** golang:1.23-alpine (or the specific version defined in go.mod).
  - **Operation:** The source code is copied, dependencies are downloaded via go mod download, and the application is compiled into a static binary.
- **Stage 2: The Runner**
  - **Base Image:** alpine:latest (A minimal Linux distribution).
  - **Operation:** Only the compiled binary from Stage 1 is copied here. This results in a lightweight, production-ready image containing only the necessary runtime artifacts, significantly reducing the attack surface and storage requirements.

4.2.2 Frontend Service (React/Vite)

The frontend utilizes a Node.js environment for development and serving the application.

- **Base Image:** node:18-alpine (or 20-alpine).
- **Operation:**
  - Dependencies are installed via npm install.
  - The development server (Vite) is started, exposing the application on port 5173.
  - *Note:* Environment variables (such as API endpoints) are injected at runtime or build time to ensure the frontend communicates correctly with the backend container.

4.2.3 Database Service (PostgreSQL)

For local development, a dedicated PostgreSQL container is utilized to simulate the production database environment.

- **Image:** postgres:15-alpine.
- **Persistence:** A named Docker volume (e.g., postgres_data) is attached to ensure data persists even if the container is stopped or removed.

4.3. Orchestration (Docker Compose)

The docker-compose.yml file serves as the central configuration to manage the lifecycle of the entire application stack. It defines:

1. **Service Dependencies:** The backend service is configured with depends_on conditions to ensure it attempts to start only after the db (database) service is healthy.
2. **Network Bridge:** A custom Docker network is created to allow seamless communication between containers using service names (e.g., the backend connects to db instead of localhost).
3. **Port Mapping:**
   - **Frontend:** Maps host port 5173 to container port 5173.
   - **Backend:** Maps host port 5000 to container port 5000.
   - **Database:** Maps host port 5432 to container port 5432 (optional, for local GUI access).
4. **Environment Management:** Credentials and configuration flags are injected via a .env file, ensuring sensitive data is never hardcoded in the image.

4.4. Installation & Setup Guide

To deploy the Cafe Management System locally using Docker, follow these steps:

Prerequisites

- **Docker Desktop** (or Docker Engine + Compose plugin) installed and running.
- **Git** installed for version control.

Step 1: Clone the Repository

Clone the project source code to your local machine:

```
git clone https://github.com/aomtrks/Order-System-Soyle.git
cd cafe-management-system
```

Step 2: Environment Configuration

Create a .env file in the root directory. This file will house the environment variables required by the Docker containers.

Example .env content:

```
# Database Configuration
DB_HOST=db
DB_USER=postgres
DB_PASSWORD=your_secure_password
DB_NAME=cafe_db
DB_PORT=5432

# Backend Configuration
PORT=5000
# OAuth Credentials (Google/Facebook)
GOOGLE_CLIENT_ID=your_google_client_id
GOOGLE_CLIENT_SECRET=your_google_client_secret
FACEBOOK_APP_ID=your_facebook_app_id
FACEBOOK_APP_SECRET=your_facebook_app_secret

# Frontend Configuration
VITE_API_URL=http://localhost:5000/api
```

Step 3: Build and Run

Execute the following command to build the images and start the services in detached mode:

```
docker-compose up --build
```

--build: Forces a rebuild of the images to ensure the latest code changes are included.

Step 4: Verification

Once the containers are running, the application will be accessible at:

- **Frontend (UI):** http://localhost:5173
- **Backend (API):** http://localhost:5000
- **Database:** Accessible via port 5432 using any PostgreSQL client (e.g., pgAdmin, DBeaver).

Step 5: Stopping the Application

To stop the services and remove the containers (preserving data volumes):

```
docker-compose down
```

4.5. Troubleshooting Common Issues

- **Port Conflicts:** If ports 5000 or 5432 are already in use on your host machine, modify the ports section in docker-compose.yml (e.g., "5001:5000").
- **Database Connection Refused:** Ensure the backend connection string uses the service name db as the hostname, not localhost, as containers communicate over the internal Docker network.
- **CORS Issues:** If API requests fail, ensure the backend main.go file includes http://localhost:5173 in the AllowOrigins CORS configuration.

4.6. Summary of Data Flow

In this containerized setup, the data flow operates as follows:

1. Incoming Request: The user accesses the application via the browser on the host machine. The request is forwarded through the host's port to the Frontend Container.
2. API Call: When the user performs an action (e.g., logging in via Google), the frontend sends a request to the Backend Container.
3. Data Query: The backend processes the request and communicates with the Database Container over the internal Docker network to retrieve or store data.
4. Response: The data travels back through the chain, ensuring a cohesive user experience despite the distributed nature of the underlying infrastructure.

5. Cloud Deployment Architecture & Production Infrastructure

5.1. Executive Summary

The deployment strategy for the Cafe Management System transitions from a local containerized environment to a **Distributed Cloud Architecture**. Unlike traditional monolithic hosting, this project adopts a "Decoupled" approach, where the Database, Backend, and Frontend layers are hosted on specialized, best-in-class cloud platforms. This strategy ensures high availability, independent scaling, and optimized performance for end-users.

The infrastructure utilizes **Neon.tech** for serverless database management, **Render** for backend API hosting, and **Vercel** for frontend global distribution.

5.2. Database Layer: Serverless PostgreSQL (Neon.tech)

The foundation of the system's data persistence is hosted on **Neon.tech**, a modern serverless PostgreSQL provider.

- **Architecture Separation:** Unlike a traditional database container running on the same server as the application, Neon decouples storage from compute. This allows the database to scale its computing resources down to zero when idle (saving costs) and instantly scale up during high traffic, without manual intervention.
- **Connection Security:** The database is accessed via a secure, encrypted connection string (postgres://...) over TLS (Transport Layer Security). This ensures that data in transit between the Backend API and the Database layer remains confidential and tamper-proof.
- **Branching & High Availability:** Neon provides cloud-native features such as database branching and autoscaling, ensuring that the production data remains isolated and highly available even during maintenance windows.

5.3. Backend Service Deployment (Render)

The Backend API (Golang) is deployed on **Render**, a unified Cloud Platform as a Service (PaaS). Render was selected for its native support for Go applications and Docker containers, ensuring a seamless transition from the local development environment.

- **Build Process:** The deployment pipeline is triggered automatically via Git integration. Upon a commit to the main branch, Render pulls the latest code and executes the build command. It compiles the Go application into a binary executable, similar to the multi-stage Docker build process described in the development phase.
- **Environment Management:** Sensitive configuration data—specifically the Database Connection String (from Neon), OAuth Client Secrets (Google/Facebook), and JWT keys—are managed via **Render's Environment Variable Dashboard**. These are injected into the server process at runtime, ensuring no sensitive credentials exist in the codebase.
- **Service Exposure:** Render automatically provisions an SSL certificate, exposing the API via a secure HTTPS endpoint (e.g., https://cafe-backend.onrender.com). This endpoint serves as the single source of truth for the Frontend application.

5.4. Frontend Application Delivery (Vercel)

The React-based user interface is deployed on **Vercel**, a platform specialized in frontend delivery and Edge Network distribution.

- **Static Asset Generation:** Upon deployment, Vercel executes the build script (vite build). This compiles the React code, bundles JavaScript modules, and optimizes CSS assets into static files.
- **Edge Network Distribution:** Unlike a traditional server that serves files from a single location, Vercel distributes these static assets across a Global Content Delivery Network (CDN). This ensures that a user accesses the application from the server node geographically closest to them, significantly reducing latency and load times.
- **Client-Side Routing:** The server is configured to handle Single Page Application (SPA) routing. All incoming requests are rewritten to the index.html entry point, allowing the React Router to handle navigation dynamically on the client side without refreshing the page.
- **Dynamic Configuration:** The connection to the backend is established via the VITE_API_URL environment variable configured in the Vercel dashboard, pointing directly to the Render backend endpoint.

5.5. Continuous Integration & Deployment (CI/CD) Pipeline

The entire system operates on an automated CI/CD workflow rooted in **GitHub**.

1. **Code Push:** A developer pushes changes to the GitHub repository.
2. **Parallel Triggers:**
   - **Vercel** detects changes in the frontend/ directory, triggers a new build, and deploys the updated UI.
   - **Render** detects changes in the backend/ directory, re-compiles the Go binary, and restarts the web service with zero downtime.
3. **Production Consistency:** This pipeline ensures that the live environment always reflects the latest stable codebase without requiring manual server access or file transfers (FTP/SSH).

5.6. Security & Cross-Origin Resource Sharing (CORS)

Connecting these distinct cloud services requires strict security configurations:

- **CORS Policy:** Since the Frontend (Vercel) and Backend (Render) reside on different domains, the Backend is explicitly configured to whitelist the Vercel domain. This prevents unauthorized websites from making API calls to the backend.
- **SSL/TLS Enforcement:** All communication between the User, Vercel, Render, and Neon occurs over HTTPS/TLS 1.2+, ensuring end-to-end encryption.