Doy
Architecture Notebook

1. Purpose

This document describes the philosophy, decisions, constraints, justifications, significant elements, and any other overarching aspects of the system that shape the design and implementation.

2. Architectural goals and philosophy

The **philosophy** of the architecture is centered on:

- To make system **easier to understand**, **maintain** and **extend** over time by leveraging well-known three-tier model.
- This layered approach allows the **separation of responsibilities**. Each layer is responsible for a distinct part of the application's behavior, allowing for independent development, testing and deployment.
- The system is designed to be **easily extendable** with additional features like customer preferences (vegan mode, allergens etc.) and a recommendation system tailored for the customer utilizing these preferences.

Goals:

- **Maintainability:** The architecture should support long-term maintenance and iterative development allowing bugfixes and new features without introducing big system-wide changes.
- **Performance:** The backend should process requests quickly, especially requests related to order like placing, modifying and cancelling the order.
- **Responsiveness:** The frontend should provide the users with responsive and seamless experience by communicating with the backend via REST APIs in a consistent manner.
- **Security:** User data especially sensitive ones like location and payment information should be securely handled and isolated in the database from other parts of the system.

3. Assumptions and dependencies

Assumptions:

- **Stable internet connection:** The application assumes users has reliable internet connection for interacting with system services.
- **Device Usage:** The users are expected to access the system via modern web browsers and use the devices that support these web browsers.
- **No legacy systems:** There is no need to integrate with legacy platforms, reducing coupling and simplifying the design.
- **Team skill set:** The development team is proficient with OOP (Object Oriented Programming), ORM (Object to Relational Mapping) and RESTful API design which drives the choice of frameworks and technologies.

Dependencies:

- **Spring Boot Framework:** Core to the backend application, used for REST APIs, dependency injection, security, and database integration.

- **JPA (Java Persistance API):** Allows the system to persist objects in the relational database utilizing ORM (Object Relational Mapping).
- **Relational Database (PostgreSQL):** Used to store and manage persistent data such as users, orders, restaurants etc.
- **Frontend Technologies (React):** The application depends on a compatible frontend to communicate with backend via HTTP.
- **Authentication and authorization services:** The backend depends on Spring Security to manage user sessions and access control.

4. Architecturally significant requirements

5. Decisions, constraints, and justifications

- **Decision: Use of Spring Boot for backend development**
  - **Justification:** Spring Boot provides production-ready framework with built-in support for REST APIs, data access, security and dependency injection. These properties accelerates the development process for the team.
- **Decision: RESTful API-based communication between backend and frontend**
  - **Justification:** REST is stateless, simple to implement and widely supported by frontend frameworks. It allows for clear separation between frontend backend concerns.
- **Decision: Use of layered architecture (Controller – Service – Repository)**
  - **Justification:** This pattern supports modularity, clear responsibility delegation and ease of testing. It also allows swapping or upgrading individual layers without affecting the entire system.
- **Constraint: Use of relational database (PostgreSQL)**
  - **Justification:** The nature of the data (users, orders, restaurants, menu items) is highly structured and relational. ACID properties are essential to ensure data integrity during concurrent operations like placing and cancelling orders.
- **DON'T: Embed business logic in the controller layer**
  - **Justification:** Controllers should only handle HTTP request/response logic. Business logic must reside in service classes to maintain separation of concerns and improve testability and maintainability.
- **DON'T: Hardcode configuration values**
  - **Justification:** All sensitive or environment-specific values must be placed in *application.properties* or loaded from environment variables to support deployment flexibility and security best practices.

6. Architectural Mechanisms

Security Mechanism

- **State:** Implementation
- **Description:** Ensures that users are authenticated and only authorized roles can access specific operations.
- **Attributes:**

- Supports JWT-based authentication.
- Role-based access control (*CUSTOMER*, *RESTAURANT_OWNER*, *COURIER*, *ADMIN*).
- Secure password storage using hashing.
- Endpoint protection via method-level and URL-level authorization.

Persistence Mechanism

- **State:** Implementation
- **Description:** Handles storage and retrieval of application data in a structured, reliable way.
- **Attributes:**
  - Uses relational database (PostgreSQL).
  - Entities managed via Spring Data JPA.
  - Supports CRUD operations and custom queries.
  - ACID compliance required for order and payment transactions.

Service Access (API) Mechanism

- **State:** Implementation
- **Description:** Enables communication between the frontend and backend through RESTful endpoints.
- **Attributes:**
  - REST over HTTP with JSON payloads.
  - Stateless communication.
  - Endpoint naming follows REST conventions.

Exception Handling Mechanism

- **State:** Implementation
- **Description:** Provides centralized error capturing and formatted HTTP responses for API clients.
- **Attributes:**
  - Returns consistent error structures (status, message, timestamp).
  - Categorized client vs. server errors (e.g., 400, 404, 500).
  - Supports custom exception types.

Validation Mechanism

- **State:** Design
- **Description:** Ensures user inputs (e.g., signup forms, menu items) are valid before processing.
- **Attributes:**
  - Uses annotations like *@NotNull, @Email, @Size.*
  - Supports both DTO and entity-level validation.

- Custom validator support planned for complex rules.

Order Lifecycle Management Mechanism

- **State:** Analysis
- **Description:** Manages the state of an order from placement to delivery, supporting transitions like "placed" → "accepted" → "delivered".
- **Attributes**
    - Order status must support transitions and be trackable.
    - Business rules define valid status changes.

User Role Management Mechanism

- **State:** Design
- **Description**: Handles assigning and enforcing different permissions for user roles.
- **Attributes**:
    - Roles persist in the database.
    - Role mapping to endpoints and services via Spring Security.

7. Key abstractions

- **App User:**
    - **Description:** Abstract base representing all users of the platform. Includes customers, couriers and restaurant owners.
    - **Role:** Centralized abstraction for login, profile and permissions management. Specialized roles extend this abstraction.

- **Customer:**
    - **Description:** Represents a user who places orders in the system. Inherits from *App User* and adds customer-specific behavior and relationships.
    - **Role:** A specialized user responsible for browsing menus, placing orders and managing personal preferences.

- **Courier:**
    - **Description:** Represents a delivery personnel responsible for transporting orders from restaurants to customers. Inherits from *App User* and contains properties specific to delivery operations.
    - **Role:** A specialized user that interacts with the order logistics system.

- **Restaurant Owner:**
    - **Description:** Represents a user who owns and manages a restaurant. Inherits from *App User* and provides business-side control over restaurant profile, menus and orders.
    - **Role:** A specialized user that enables restaurant-side interaction with the system.

- **Restaurant:**
    - **Description:** Represents a vendor in the system that offers products. Includes metadata such as name, address, phone number etc.
    - **Role:** Aggregates and owns *MenuItems*, can manage its own menu and receive orders.

- **Menu Item:**
    - **Description:** Represents an item available for purchase, such as food or beverages in a restaurant menu. It holds common attributes such as name, price, availability status and description.
    - **Role:** Base abstraction for menu items regardless of category.

- **Cart:**
    - **Description:** Represents a temporary container for *Menu Item*s a customer intends to order. Each customer has exactly one cart, linked via a shared primary key, which stores *Cart Item*s and their quantities.
    - **Role:** A central abstraction that supports the ordering flow by holding selected items before finalizing a purchase.

- **Cart Item:**
    - **Description:** Represents an individual item in a customer's cart, linking a *Menu Item* with a specified quantity. Serves as a detail record in the cart that contributes to the overall order preparation.
    - **Role:** Acts as a bridge between the *Cart* and *Menu Item*, allowing tracking of selected *Menu Item*s and their quantities, support for itemized subtotal calculations and simplified transformation into *Order Item*s upon checkout.

- **Customer Order:**
    - **Description:** Represents a transaction where a customer purchases one or more products. Includes information about order items, status (e.g., placed, accepted, delivered), pricing and timestamps.
    - **Role:** Key abstraction connecting customers, order items and payment flow.

- **Order Item:**
    - **Description:** Represents a *Menu Item* with quantity within a customer's order, linking a specific *Menu Item* to overall *Customer Order*. Each order may contain multiple *Order Item*s one for each *Menu Item* the customer selects.
    - **Role:** Serves as a bridge between the *Customer Order* and *Menu Item* entities, enabling the system to track which items were ordered and in what quantity.

8. Layers or architectural framework

The backend architecture for our food delivery project is built using a layered (3-tier) architectural pattern. This design organizes the system into distinct layers that separate concerns, making the code easier to maintain, test and extend. The architecture is structured as:

1. **Presentation Layer (Controllers):** The controllers package handles incoming HTTP requests and orchestrates responses. It serves as the interface for clients to interact with the system. Internally, it contains components that map HTTP requests to business operations, ensuring that the request processing (like input validation, formatting and routing) remains separate from the core logic of the application.

2. **Business Logic Layer (Services):** Acting as the core of the system, this layer encapsulates all the essential business rules and workflows. It processes data received from the presentation layer, coordinates with repositories for data persistence and applies the necessary business decisions.

3. **Data Access Layer (Repositories):** This layer is tasked with managing all interactions with the data storage. It provides a clear separation between the business logic and the underlying database mechanisms. The abstraction offered here means that changes to the database system or data access strategies can be implemented without disturbing the other layers.

Overall, this structured approach not only simplified maintenance and testing but also provides scalability, ensuring that changes in one area can be performed independently without unintended side effects.

9. Architectural views

Logical View

The application follows a layered architecture pattern with clear separation of concerns. The key architectural elements can be identified as:

- **Presentation Layer**
- **Business Logic Layer**
- **Data Access Layer**

Key Components:

- **Authentication & Security:** JWT-based authentication framework with user registration and login flows.
- **DTO Pattern:** Data Transfer Objects facilitate clean data exchange between layers.
- **Enumerations:** Type-safe representations of domain concepts.
- **Interfaces:** Clear contracts between components promoting loose coupling.

Domain Areas:
The application has these main domain concepts:

- Restaurant management
- Menu item management
- Customer ordering system
- User authentication and authorization

Operational View

The application is structures as a Spring Boot application, which operates as follows:

Runtime Components:

- **Application Server:** Spring Boot embedded application server (Tomcat).
- **Database Connection:** Persistence layer using JPA/Hibernate.,
- **Security Filter Chain:** JWT authentication filter integrated into request processing.
- **Email Service:** External communication capability for user confirmation

Execution Flow:

1. HTTP requests are received by controllers.
2. Requests pass through security filters for authentication.
3. Controllers delegate to appropriate services.
4. Services implement business logic, interacting with repositories.
5. Repositories handle database operations.
6. Response flows back through the service and controller layers.

Thread Interactions:

- **Request Processing Flow:** When a request arrives, a thread from the servlet container's pool handles the complete request-response cycle through the controller, service and repository layers.
- **Database Connection Pool:** Database operations use connection pooling, where a separate set of connections is maintained and used by request-handling threads when needed.
- **Transaction Management:** Spring's transaction management ensures that database operations within a single logical operation occur atomically, with thread local transaction contexts.

- **Use case:** A list or diagram of the use cases that contain architecturally significant requirements.

Use Case View

Customer-Focused Use Cases

- **User Registration and Authentication:** With use of registration controllers, login services and confirmation tokens a complete user onboarding flow with email verification is ensured.
- **Restaurant Browsing and Searching:** With search functionality customers can discover restaurants with filtering capabilities.
- **Menu Exploration:** Item controllers and repositories enable customers to view available food items from restaurants.
- **Order Management:** With the use of order services and related entities *(Cart, Cart Item, Customer Order)* customers can create and manage food orders.

Restaurant-Focused Use Cases

- **Restaurant Profile Management:** *Restaurant* services and repositories allow restaurant owners to create and update their profiles and construct their menus with *Menu Item*s.
- **Menu Item Management:** With the use of *Menu Item* services and repositories restaurant owners can persist the *Menu Item*s they created in the database.

System-Level Use Cases

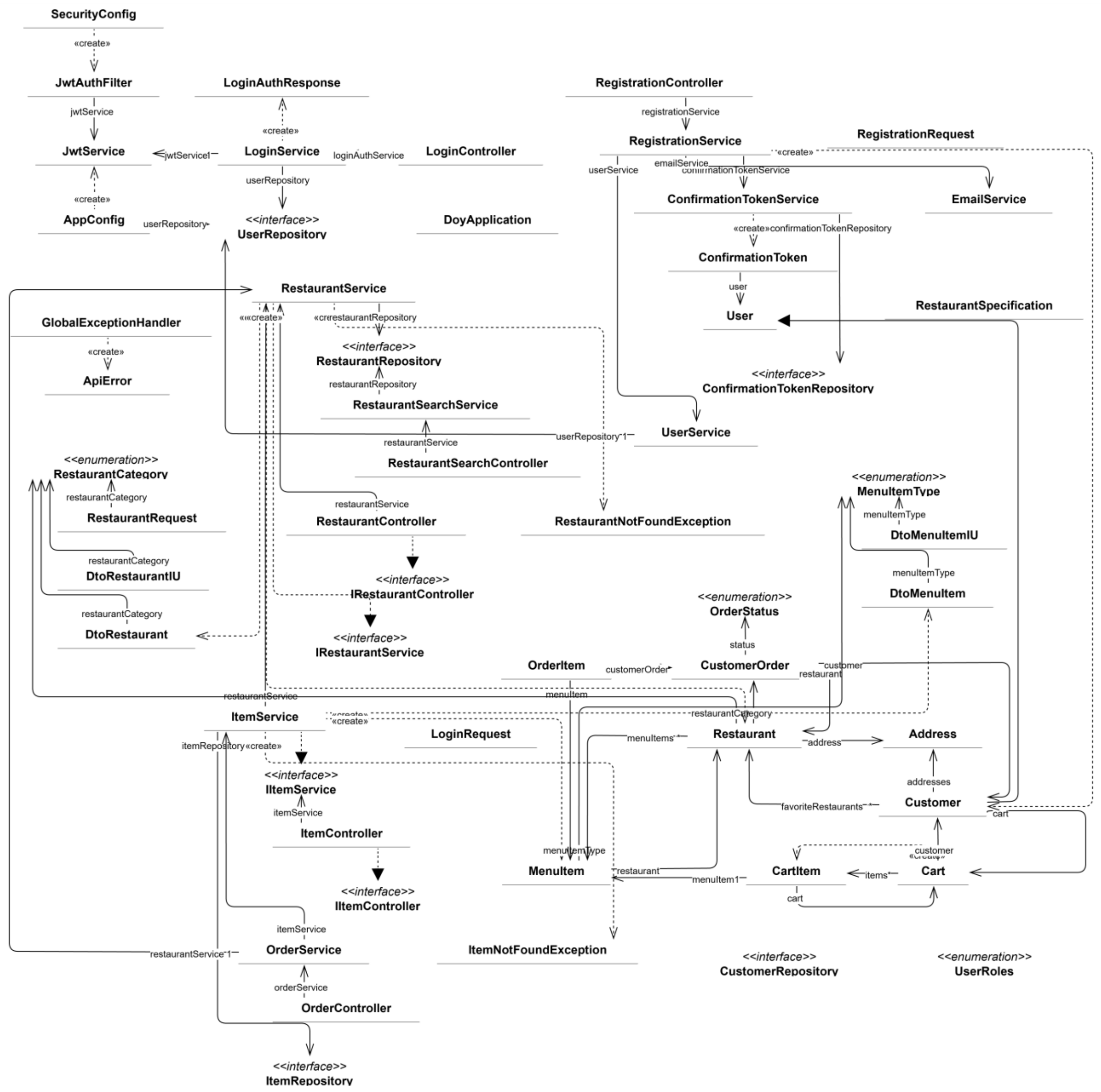
- **Security and Authorization:** JWT implementation and security configurations allow for role-based access control for different user types.
- **Order Processing Workflow:** With use of order status enums and order services a complete workflow from order creation to delivery is realized.

10. Traceability Table

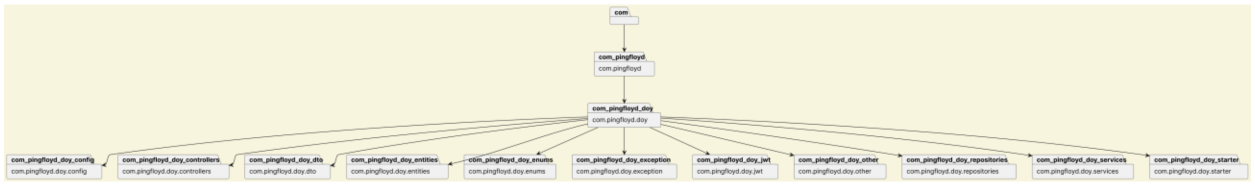
Requirement identifiers	Reqs tested	REQ1 UC-FO-001 (User Registration)	REQ1 UC-FO-002 (Food Ordering)	REQ1 UC-FO-010 (Menu Management)
Test cases		3	2	1
BR_1 (User Registration and Authentication)	2	x	x	
BR_2 (Restaurant Search and Browsing)	1		x	
BR_3 (Food Ordering Process)	1		x	
BR_4 (Restaurant Menu Management)	1			x
TC#001 (Verify user registration)	1	x		
TC#002 (Verify user login)	1	x		
TC#003 (Verify address validation)	1	x		
TC#004 (Verify profile persistence)	1	x		
TC#005 (Verify restaurant search)	1		x	
TC#006 (Verify category browsing)	1		x	
TC#007 (Verify adding items to cart)	1		x	
TC#008 (Verify cart management)	1		x	
TC#009 (Verify adding menu items)	1			x
TC#010 (Verify updating menu items)	1			x

11. Appendix

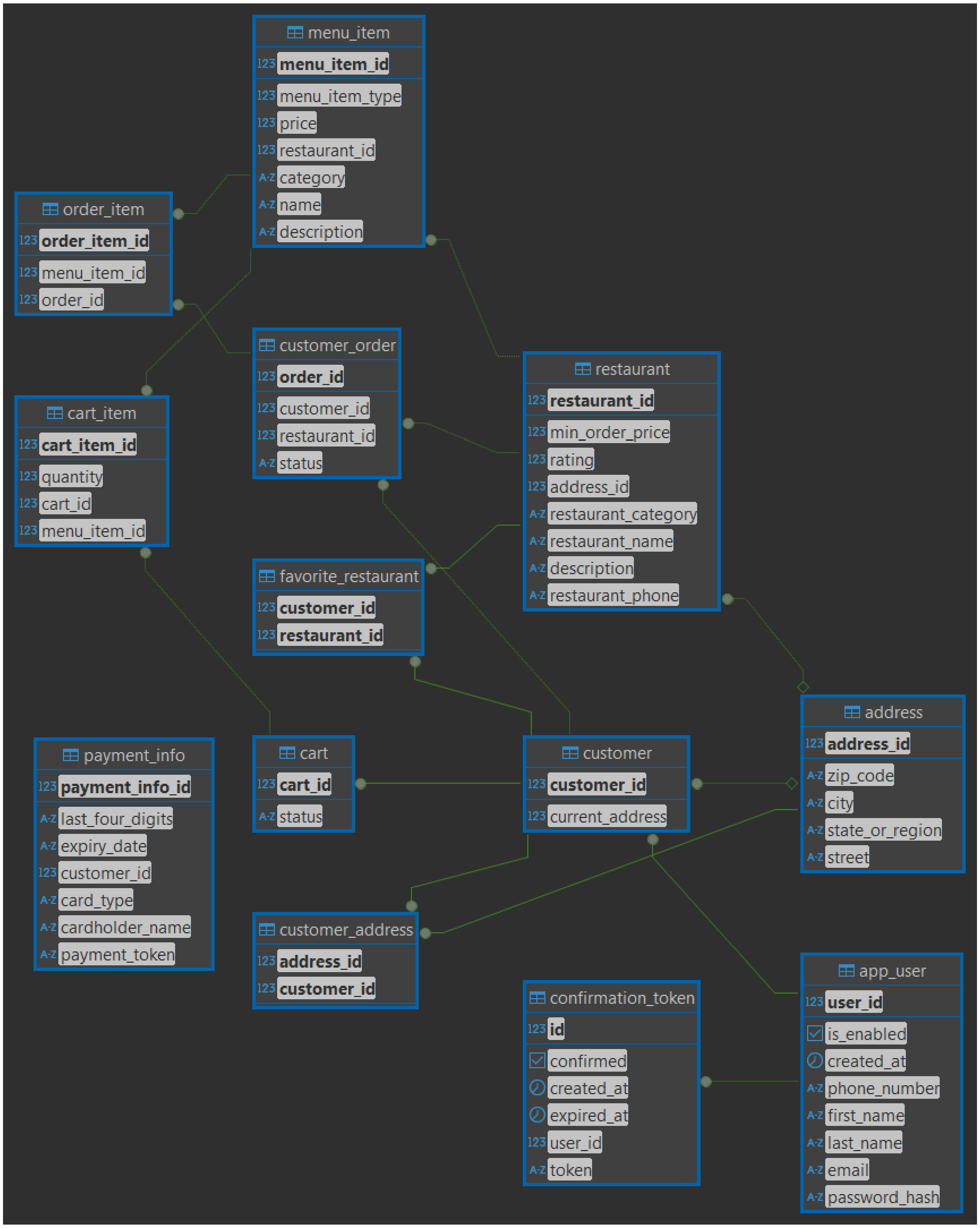
Class Diagram



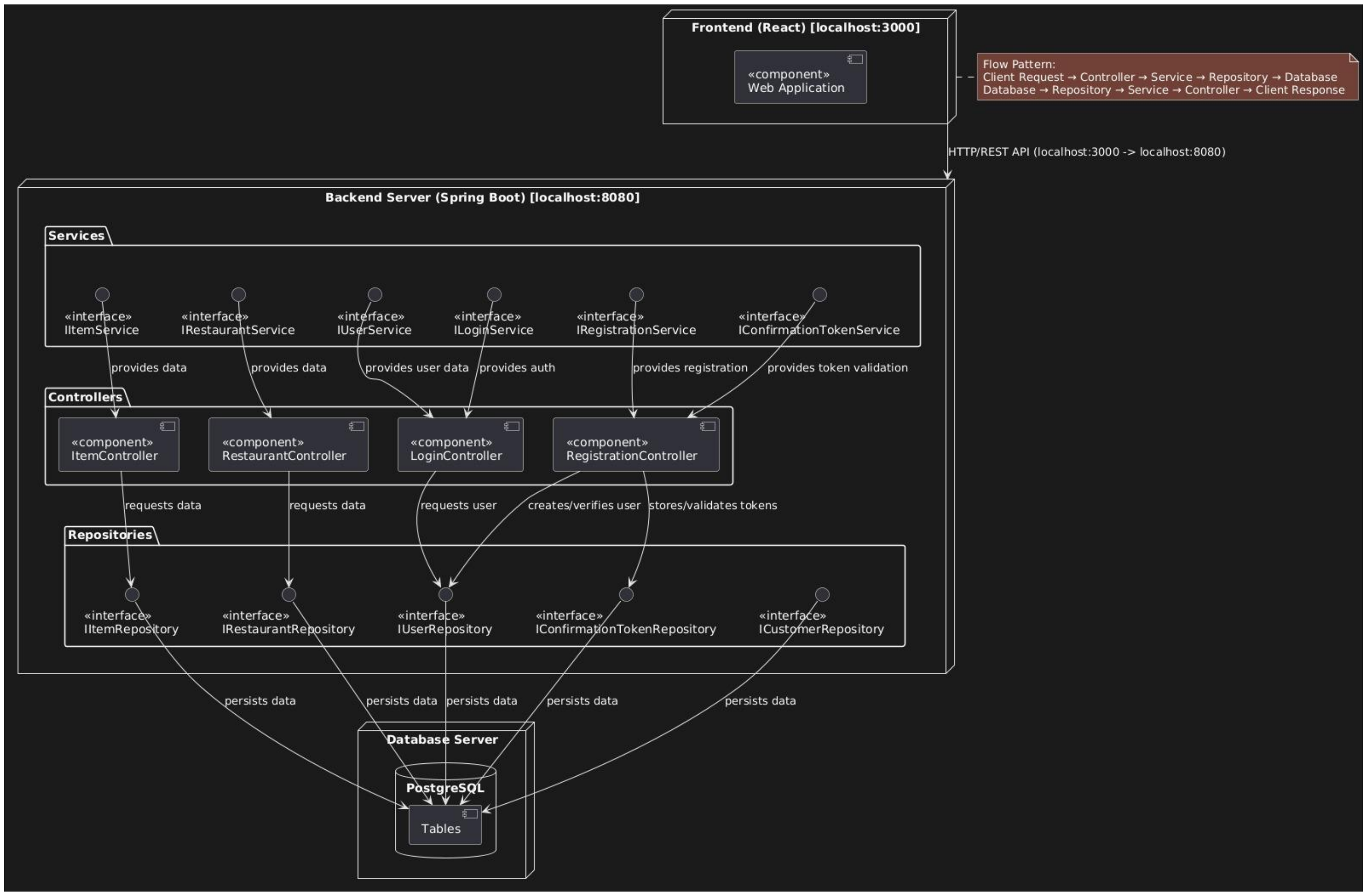
Package Diagram


com
com_pingfloyd
com.pingfloyd
com_pingfloyd_doy
com.pingfloyd.doy
com_pingfloyd_doy_config
com.pingfloyd.doy.config
com_pingfloyd_doy_controllers
com.pingfloyd.doy.controllers
com_pingfloyd_doy_dto
com.pingfloyd.doy.dto
com_pingfloyd_doy_entities
com.pingfloyd.doy.entities
com_pingfloyd_doy_enums
com.pingfloyd.doy.enums
com_pingfloyd_doy_exception
com.pingfloyd.doy.exception
com_pingfloyd_doy_jwt
com.pingfloyd.doy.jwt
com_pingfloyd_doy_other
com.pingfloyd.doy.other
com_pingfloyd_doy_repositories
com.pingfloyd.doy.repositories
com_pingfloyd_doy_services
com.pingfloyd.doy.services
com_pingfloyd_doy_starter
com.pingfloyd.doy.starter

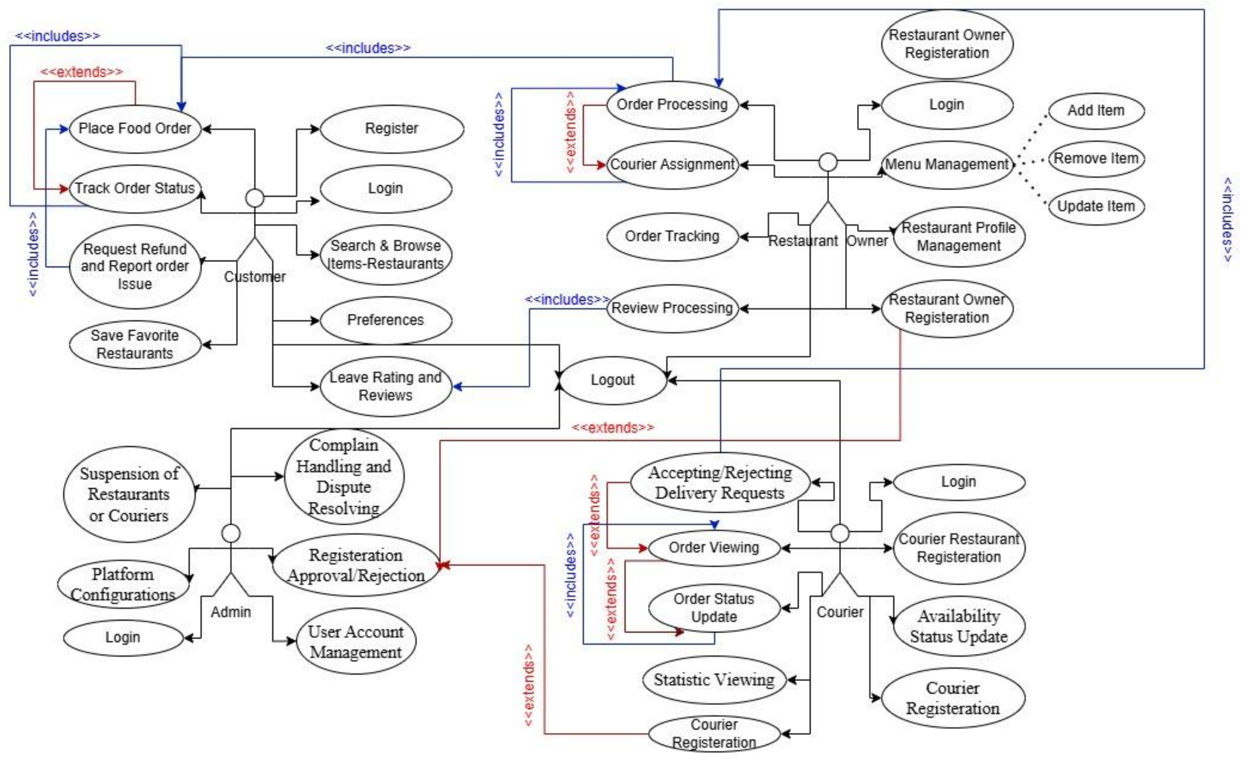
Updated ER Diagram



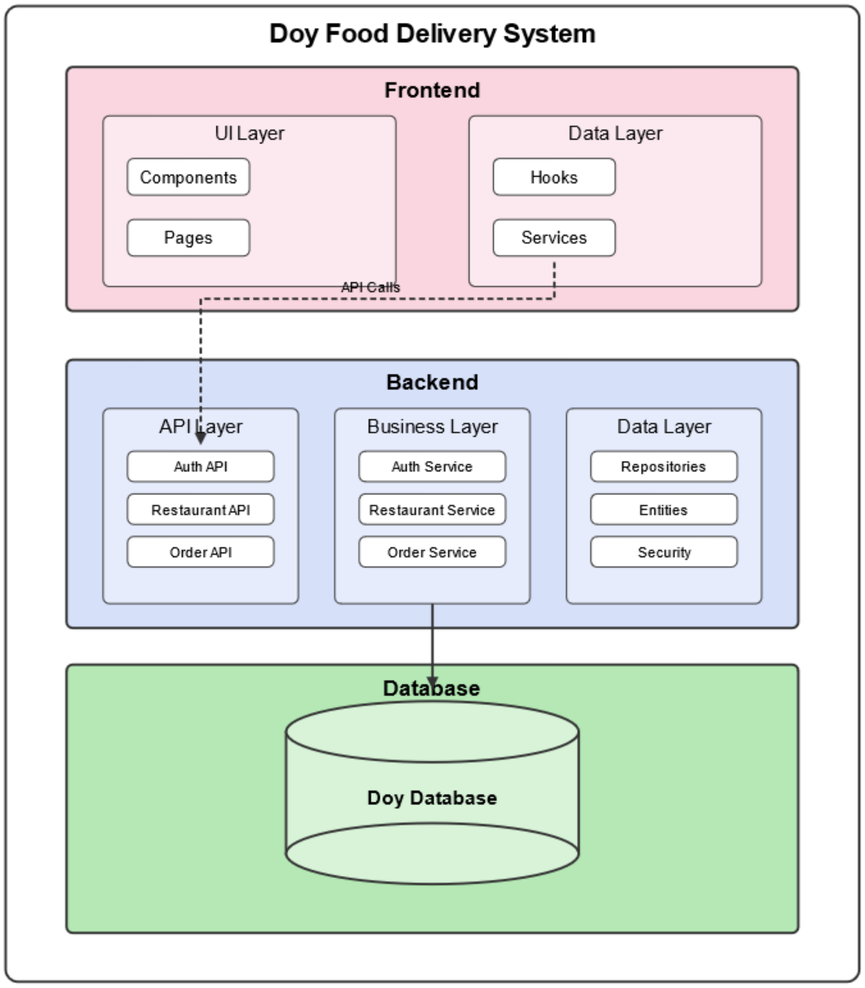
Deployment Diagram


Frontend (React) [localhost:3000]
«component» Web Application
Flow Pattern:
Client Request → Controller → Service → Repository → Database
Database → Repository → Service → Controller → Client Response
HTTP/REST API (localhost:3000 -> localhost:8080)
Backend Server (Spring Boot) [localhost:8080]
Services
«interface» IItemService
«interface» IRestaurantService
«interface» IUserService
«interface» ILoginService
«interface» IRegistrationService
«interface» IConfirmationTokenService
provides data
provides data
provides user data
provides auth
provides registration
provides token validation
Controllers
«component» ItemController
«component» RestaurantController
«component» LoginController
«component» RegistrationController
requests data
requests data
requests user
creates/verifies user
stores/validates tokens
Repositories
«interface» IItemRepository
«interface» IRestaurantRepository
«interface» IUserRepository
«interface» IConfirmationTokenRepository
«interface» ICustomerRepository
persists data
persists data
persists data
persists data
persists data
Database Server
PostgreSQL
Tables

Use Case Diagram



Component Diagram



12. Prompts

- https://claude.ai/chat/e32e5a3f-046c-4884-8c2c-46ac8d2dbd0d
- https://claude.ai/chat/d89418d5-ab33-49cb-8ff2-758212a94576
- https://you.com/search?q=Layered+Architecture+Overview
- https://claude.ai/chat/d0196163-927e-4642-8dee-31a126b20dcc