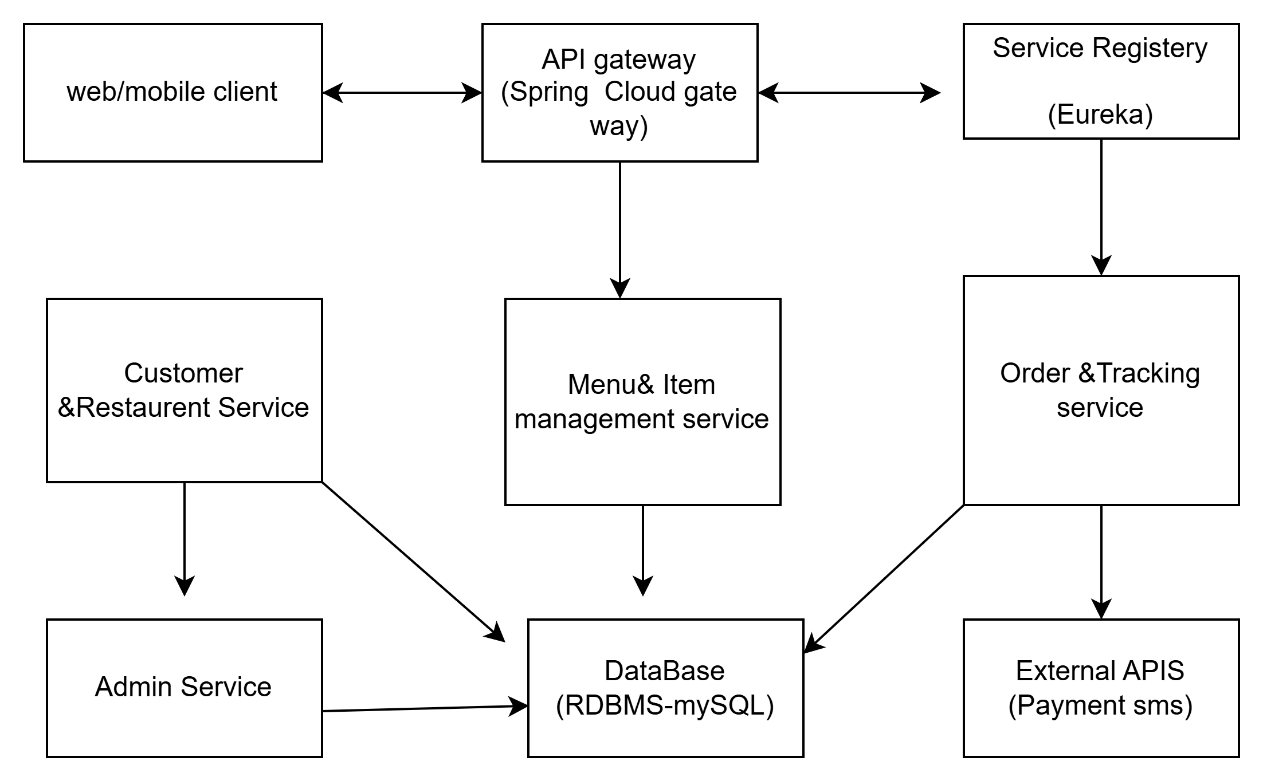
Project Name: Online Food Ordering System    
Technology Stack: Java, Spring Boot, Microservices Architecture   
  
  
 Introduction:-  
"The Online Food Ordering System is a microservices-based application designed to provide a seamless platform for customers and restaurants. Customers can browse restaurant menus, place orders, and track deliveries, while restaurants can manage their menus, and orders. Admins oversee all the operations in the system. The system is highly scalable, using Spring Boot microservices, **Spring Security** and **JWT Authentication** for secure access."

HLD Diagram for online food ordering system:



the main components in the **HLD Diagram**

#### ****1. Client Application (Web/Mobile)****

"The **Client App** is the user interface where both customers and restaurants interact with the system. Customers use it to browse menus, place orders, and track their deliveries. Restaurants use it to manage their profiles and menus. This application can be either a web app or a mobile app. It communicates with the backend services via APIs exposed by the API Gateway."

#### ****2. API Gateway****

"The **API Gateway** (implemented using Spring Cloud Gateway) serves as the entry point for all requests coming from the client. It will routing requests to the appropriate microservices. It also provides some functionality like load balancing, request filtering, and security checks. All external API calls from the client pass through the API Gateway."

example:-if a customer places an order, the API Gateway forwards that request to the appropriate service (like the Order Service) after performing authentication and authorization checks."

#### ****3. Service Discovery (Eureka)****

"We use **Eureka** as the **Service Discovery** mechanism, which enables the dynamic discovery of services in the system. Each microservice registers itself with Eureka, and Eureka provides a registry of available services. When one service needs to communicate with another, it can query Eureka to find the location of the service, ensuring that the system can adapt to changes in service instances dynamically."

"For instance, if a service instance crashes and restarts, Eureka updates its registry to reflect the new instance, so the API Gateway or other services can always find the correct instance to send requests to."

#### ****4. Microservices****

##### ****Customer & Restaurant Service****:

"This service handles the **registration** and **management** of customer and restaurant profiles. Customers can register, update their details, and login to the system, while restaurants can manage their menus, update information, and configure service options."

"This service uses **JWT authentication** to secure access for different user roles (Customer, Restaurant, Admin). For example, a customer can only perform actions related to their profile, while restaurants have access to their menus."

##### ****Menu & Item Management Service****:

"This service is responsible for the **menu management** of restaurants. Restaurants can **add, update, and remove food items** from their menus. This service ensures that only authorized restaurant users can modify the menu, while customers can view it for placing orders."

##### ****Order & Tracking Service****:

"This service manages the **order placement** and **tracking** functionality. It allows customers to place orders by selecting items from the menu. Once an order is placed, the service tracks the order status, updating customers on the progress (e.g., processing, in delivery, delivered)."

##### ****Admin Service****:

"This service is for **administrative oversight** of the entire system. It allows the admin to view and manage users, track transactions, monitor the system’s operations, and ensure everything is running smoothly."

#### ****5. Database Layer****

"All the application data, such as user details, restaurant menus, orders, and transaction history, is stored in a **database ,** The database ensures that the data is structured and easily accessible, and each microservice interacts with it through its own database schema. For example, the Customer Service interacts with the user data, while the Order Service interacts with order records."

#### ****6. External APIs (Payment, SMS)****

"For features like **payment processing** and **SMS notifications**, the system integrates with **external APIs**. The Payment API handles transactions, while the SMS API sends delivery notifications to customers. These services are designed to be modular so that they can be replaced or extended in the future."

### ****Interaction Flow****

an example of a **customer order flow**:

1. **Customer logs in** through the Client App.
   * The request is sent to the **API Gateway**.
   * The API Gateway authenticates the customer using **JWT tokens** and forwards the request to the **Customer & Restaurant Service**.
2. **Customer browses the menu** of a restaurant and places an order.
   * The Client App calls the **Menu & Item Management Service** to get the available items.
   * The customer places an order, which is sent to the **Order & Tracking Service**.
3. **Order Processing**.
   * The **Order & Tracking Service** updates the status of the order and tracks it in the system.
   * The **Order & Tracking Service** communicates with the database (RDBMS) to store order data.
4. **Payment and Notifications**.
   * The **Payment Service** processes the payment (via external APIs) and sends a confirmation.
   * The **SMS Notification Service** sends an update to the customer about the order status (e.g., "Your food is on the way").

Service 1: Customer & Restaurant Registration and Management  
  
1. Customer Registration:  
   - Customers can register using email, phone number, or social media.  
   - Validate inputs (e.g., email format, password strength).  
2. Restaurant Registration:  
   - Restaurants can register by providing details (name, address, cuisine type, contact info).  
   - Admin approval required for restaurant registration.  
3. Profile Management:  
   - Customers and restaurants can update their profiles (e.g., name, address, phone number).  
   - Restaurants can update their menus and service options.  
  
Implementation Steps:  
1. Database Design:  
   - Create tables for `Customers`, `Restaurants`, and `Admins`.  
   - Example:  
     ```sql  
     CREATE TABLE Customers (  
         id INT PRIMARY KEY AUTO\_INCREMENT,  
         name VARCHAR(100),  
         email VARCHAR(100) UNIQUE,  
         password VARCHAR(100),  
         phone VARCHAR(15),  
         address VARCHAR(255)  
     );  
  
     CREATE TABLE Restaurants (  
         id INT PRIMARY KEY AUTO\_INCREMENT,  
         name VARCHAR(100),  
         address VARCHAR(255),  
         cuisine\_type VARCHAR(100),  
         contact\_info VARCHAR(100),  
         approved BOOLEAN DEFAULT FALSE  
     );  
     ```  
2. \*\*APIs\*\*:  
   - `POST /api/customers/register`: Register a new customer.  
   - `POST /api/restaurants/register`: Register a new restaurant.  
   - `PUT /api/customers/{id}`: Update customer profile.  
   - `PUT /api/restaurants/{id}`: Update restaurant profile.  
  
  
---  
  
 \*\*Service 2: Menu and Item Management:  
\*\*Functionality\*\*:  
1. Add Menu Items:  
   - Restaurants can add food items (name, description, price, category, image).  
2. Modify Menu Items:  
   - Restaurants can update or delete food items.  
  
\*\*Implementation Steps\*\*:  
1. \*\*Database Design\*\*:  
   - Create a `MenuItems` table.  
   - Example:  
     ```sql  
     CREATE TABLE MenuItems (  
         id INT PRIMARY KEY AUTO\_INCREMENT,  
         restaurant\_id INT,  
         name VARCHAR(100),  
         description TEXT,  
         price DECIMAL(10, 2),  
         category VARCHAR(50),  
         image\_url VARCHAR(255),  
         FOREIGN KEY (restaurant\_id) REFERENCES Restaurants(id)  
     );  
     ```  
2. APIs:  
   - `POST /api/restaurants/{restaurantId}/menu`: Add a new menu item.  
   - `PUT /api/restaurants/{restaurantId}/menu/{itemId}`: Update a menu item.  
   - `DELETE /api/restaurants/{restaurantId}/menu/{itemId}`: Delete a menu item.  
  
3. \*\*Spring Boot Implementation\*\*:  
   - Use \*\*Spring Data JPA\*\* for CRUD operations.  
   - Validate inputs (e.g., price must be positive).  
  
---  
  
Service 3: Customer Ordering and Tracking:  
\*\*Functionality\*\*:  
1. Browse Menus:  
   - Customers can view menus of registered restaurants.  
2. Place Orders:  
   - Customers can add items to a cart and place an order.  
3. Track Orders:  
   - Customers can track the status of their orders (e.g., preparing, out for delivery, delivered).  
  
#### \*\*Implementation Steps\*\*:  
1. \*\*Database Design\*\*:  
   - Create tables for `Orders` and `OrderItems`.  
   - Example:  
     ```sql  
     CREATE TABLE Orders (  
         id INT PRIMARY,  
         customer\_id INT,  
         restaurant\_id INT,  
         status VARCHAR(50),  
         total\_price DECIMAL(10, 2),  
         created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP,  
         FOREIGN KEY (customer\_id) REFERENCES Customers(id),  
         FOREIGN KEY (restaurant\_id) REFERENCES Restaurants(id)  
     );  
  
     CREATE TABLE OrderItems (  
         id INT PRIMARY KEY,  
         order\_id INT,  
         menu\_item\_id INT,  
         quantity INT,  
         FOREIGN KEY (order\_id) REFERENCES Orders(id),  
         FOREIGN KEY (menu\_item\_id) REFERENCES MenuItems(id)  
     );  
     ```  
2. \*\*APIs\*\*:  
   - `GET /api/restaurants/{restaurantId}/menu`: Get menu items for a restaurant.  
   - `POST /api/orders`: Place a new order.  
   - `GET /api/orders/{orderId}`: Track order status.  
  
3. \*\*Spring Boot Implementation\*\*:  
   - Use \*\*Spring Data JPA\*\* for order management.  
   - Use \*\*WebSocket\*\* or \*\*Polling\*\* for real-time order tracking.  
  
---  
  
Service 4: Administrative Oversight:  
##Functionality:  
1. \*\*User Activity Monitoring\*\*:  
   - Admins can view all user activities (e.g., registrations, orders).  
2. \*\*Transaction Monitoring\*\*:  
   - Admins can view all transactions and revenue reports.  
  
#### \*\*Implementation Steps\*\*:  
1. \*\*Database Design\*\*:  
   - Use existing tables (`Customers`, `Restaurants`, `Orders`) for reporting.  
2. \*\*APIs\*\*:  
   - `GET /api/admin/users`: Get all users.  
   - `GET /api/admin/orders`: Get all orders.  
   - `GET /api/admin/transactions`: Get all transactions.  
  
3. \*\*Spring Boot Implementation\*\*:  
   - Use \*\*Spring Data JPA\*\* for querying data.  
   - Use \*\*Spring Security\*\* to restrict access to admin-only endpoints.  
  
---  
  
## \*\*Microservices Architecture\*\*  
1. \*\*Service Breakdown\*\*:  
   - \*\*User Service\*\*: Handles customer and restaurant registration.  
   - \*\*Menu Service\*\*: Manages menu items.  
   - \*\*Order Service\*\*: Handles order placement and tracking.  
   - \*\*Admin Service\*\*: Provides admin functionalities.  
  
2. \*\*API Gateway\*\*:  
   - Use \*\*Spring Cloud Gateway\*\* as a single entry point for all microservices.  
  
3. \*\*Service Discovery\*\*: Eureka  
   
5. \*\*Database Per Service\*\*:  
   - Each microservice should have its own database.  
  
---  
  
\*\*Tools   
1. \*\*GitHub\*\*:  
   - Use Git for version control. Create separate branches for each feature.  
2. \*\*Postman\*\*:  
   - Test all APIs using Postman.  
3. \*\*RDBMS\*\*:  
   - Use \*\*MySQL\*\* or \*\*PostgreSQL\*\* for database management.  
4. \*\*IDE\*\*:  
   - Using \*\*IntelliJ IDEA\*\* for development.  
5. \*\*Logging\*\*:  
   - Use \*\*Logback\*\* or \*\*SLF4J\*\* for logging.  
6. \*\*Testing\*\*:  
   - Write unit tests using \*\*JUnit\*\* and \*\*Mockito\*\*.  
  
---

### Dev plan and efforts estimation:-

### Sprint Plan for Online Food Ordering System (1-Week Sprint)

down the development work for the **Online Food Ordering System** into tasks and assign **story points** This sprint plan assumes we're focusing on the initial set of core features that would be a good starting point for the project.

**Sprint Duration:** 1 Week

### ****User Stories Breakdown for Sprint****

#### ****Service 1: Customer & Restaurant Registration and Management****

##### ****User Story 1.1: Customer Registration and Login****

**Story**: As a customer, I want to register and log in to my account so that I can place orders and manage my profile.

**Tasks**:

* Develop REST API for customer registration and login using Spring Boot.
* Implement database schema for storing customer data (MySQL).
* Implement session management using JWT (JSON Web Token) for authentication.
* Frontend integration for login and registration forms.
* Test APIs with Postman.

**Story Points**: 5 SP

##### ****User Story 1.2: Restaurant Registration and Profile Management****

**Story**: As a restaurant owner, I want to register my restaurant and manage my profile so that I can provide services on the platform.

**Tasks**:

* Develop REST API for restaurant registration (Spring Boot).
* Create database schema for restaurant data.
* Implement profile management for restaurant owners to update their details (name, address, etc.).
* Admin validation for restaurant profile.
* Test APIs with Postman.

**Story Points**: 5 SP

#### ****Service 2: Menu and Item Management****

##### ****User Story 2.1: Menu Item Management for Restaurants****

**Story**: As a restaurant owner, I want to add, edit, and delete items on my menu to keep my offerings up to date.

**Tasks**:

* Develop API for adding, updating, and removing menu items (Spring Boot).
* Create the database schema to store menu items (name, description, price, etc.).
* Create validation to ensure menu items are correctly entered (required fields, price checks).
* Basic UI for restaurant owners to manage the menu.
* Test the APIs with Postman.

**Story Points**: 6 SP

##### ****User Story 2.2: Display Menu for Customers****

**Story**: As a customer, I want to view the restaurant’s menu so that I can choose items to order.

**Tasks**:

* Develop API to fetch menu items (Spring Boot).
* Display menu items in the frontend (web or mobile).
* Implement pagination and filtering options (e.g., categories, price range).
* Basic UI for displaying the menu.

**Story Points**: 5 SP

#### ****Service 3: Customer Ordering and Tracking****

##### ****User Story 3.1: Customer Order Placement****

**Story**: As a customer, I want to place an order for food from the menu, specifying items and delivery details.

**Tasks**:

* Develop API for order creation (Spring Boot).
* Implement database schema for storing order details (order items, delivery address, etc.).
* Implement basic validation (check stock availability, address validation).
* Integrate frontend interface for order placement.

**Story Points**: 6 SP

#### ****Service 4: Administrative Oversight****

##### ****User Story 4.1: Admin Access for Order and User Monitoring****

**Story**: As an admin, I want to view orders, customer activity, and transaction data to ensure smooth operations and performance monitoring.

**Tasks**:

* Develop API for admin to view orders, user activity, and transactions (Spring Boot).
* Implement admin authentication (role-based access control).
* Create a basic admin dashboard to display order statistics and user activity.
* Test APIs and frontend integration.

**Story Points**: 6 SP

We will focus on the following core features:

1. **Customer Registration and Login** (5 SP)
2. **Restaurant Registration and Profile Management** (5 SP)
3. **Menu Item Management for Restaurants** (6 SP)
4. **Display Menu for Customers** (5 SP)
5. **Customer Order Placement** (6 SP)

**Total Story Points for Adjusted Sprint**  **25 SP**  
This is a manageable scope for the sprint.

### ****Sprint Tasks and Goals for Week 1****

#### ****Day 1-2:****

* Set up project structure (Spring Boot microservices).
* Implement Customer Registration & Login API.
* Implement Restaurant Registration API and profile management.

#### ****Day 3-4:****

* Develop Menu Item Management API for restaurants.
* Implement database schemas for customers, restaurants, and menu items.
* Develop frontend interface for customer login/registration and restaurant profile management.

#### ****Day 5:****

* Develop API for displaying menu to customers.
* Begin implementation of Customer Order Placement API.
* Basic UI for displaying the restaurant’s menu.

#### ****Day 6-7:****

* Test the APIs with Postman and ensure integration.
* Begin frontend integration for placing orders (connect API).
* Review the progress, debug, and conduct QA testing.

### Establish Coding standards:- ****Coding Standards:****

* **Naming Conventions**: Follow Java naming conventions:
  + **Classes**: Use CamelCase, start with an uppercase letter (e.g., CustomerService).
  + **Methods**: Use camelCase, start with a lowercase letter (e.g., createOrder()).
  + **Variables**: Use meaningful names, use camelCase (e.g., orderDetails, customerId).
  + **Constants**: All uppercase letters with underscores separating words (e.g., MAX\_RETRIES).

#### ****2 Documentation****

* **JavaDocs**: Every public method and class should have JavaDoc comments explaining its purpose and parameters.
* /\*\*
* \* Registers a new customer.
* \* @param customerDetails the details of the customer
* \* @return the created customer object
* \*/
* public Customer registerCustomer(CustomerDetails customerDetails) {
* // implementation
* }
* **Inline Comments**: Use comments to explain complex code logic but avoid redundant comments. Focus on explaining why rather than what the code does.

#### ****1.3 Code Readability****

* **Avoid Long Methods**: Methods should do one thing and be as short as possible (preferably under 50 lines).
* **Refactor Long Classes**: Classes should have a single responsibility. If a class grows too large, break it down into smaller, cohesive components.
* **Logical Grouping**: Group related methods together and use regions or blank lines to separate logical blocks in classes.

### ****2. Java Specific Coding Standards:****

#### ****2.1 Exception Handling****

* Use **custom exceptions** to handle domain-specific errors (e.g., RestaurantNotFoundException).
* Use **try-catch blocks** appropriately, but avoid overusing them. Propagate exceptions when necessary to centralize error handling.
* Provide detailed error messages with meaningful information.
* try {
* // Logic
* } catch (RestaurantNotFoundException ex) {
* throw new RestaurantNotFoundException("Restaurant not found with ID: " + restaurantId);
* }

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

* Use **Spring's Dependency Injection** to manage your services and repositories instead of creating instances manually. This promotes loose coupling and easier testing.
* @Autowired
* private OrderService orderService;

#### ****2.3 Use of Spring Annotations****

* Use the right **Spring annotations** for service classes, controllers, and components:
  + @Service for business logic classes.
  + @Repository for database interaction classes.
  + @RestController or @Controller for web controllers.
  + @Autowired for automatic dependency injection.

#### ****2.4 Avoid Static Methods****

* Avoid using static methods or variables unless necessary, as they are difficult to mock for testing and can create tight coupling.
* Use **instance-based methods** whenever possible, relying on dependency injection.

### ****3. Microservices and API Design Standards:****

#### ****3.1 RESTful API Design****

* **Endpoint Naming**: Follow REST conventions for API endpoints:
  + Use **noun-based** endpoints (e.g., /customers, /orders/{orderId}, /menu-items).
  + Use **HTTP methods** correctly:
    - GET for reading data.
    - POST for creating data.
    - PUT/PATCH for updating data.
    - DELETE for deleting data.
* **Response Status Codes**:
  + 200 OK for successful requests.
  + 201 Created for successful creation.
  + 400 Bad Request for invalid client requests.
  + 404 Not Found for non-existent resources.
  + 500 Internal Server Error for server issues.

#### ****3.2 Input Validation****

* Validate inputs at the controller layer using **Spring’s validation annotations** (@NotNull, @Size, @Email, etc.).
* @PostMapping("/customers")
* public ResponseEntity<Customer> createCustomer(@Valid @RequestBody Customer customer) {
* // implementation
* }

#### ****3.3 Consistent Response Format****

* Use a standard response wrapper for consistency in API responses, e.g.:
* {
* "status": "success",
* "data": { ... },
* "message": "Request completed successfully."
* }

#### ****3.4 Pagination and Sorting****

* Implement **pagination** for endpoints that return lists, e.g., /menu-items?page=1&size=10.
* Use **sorting** options for endpoints that support ordering (e.g., /orders?sortBy=date&direction=asc).

### ****4. Database and SQL Standards:****

#### ****4.1 Entity Naming and Database Schema****

* Use singular names for **entities** (e.g., Customer, Order, Restaurant).
* Maintain **consistency** between class names and database table names (e.g., orders table for the Order entity).
* Use **proper indexing** for frequently queried columns (e.g., foreign keys, primary keys, and frequently filtered columns).
* Avoid storing **business logic** in the database (e.g., no complex calculations in SQL queries).

#### ****4.2 SQL Queries and Transactions****

* Use **Spring Data JPA** or **Spring JDBC** for database interactions. Avoid writing custom SQL unless necessary.
* For complex operations that require multiple steps, wrap them in a **transaction** to ensure data consistency.
* @Transactional
* public void createOrder(Order order) {
* // database operations
* }

### ****5. Testing Standards:****

#### ****5.1 Unit Testing****

* Use **JUnit 5** and **Mockito** for unit testing and mocking external dependencies (like database calls).
* Test service logic, ensuring correctness of business rules.
* @Test
* void testRegisterCustomer() {
* Customer customer = new Customer("John", "[john@example.com](mailto:john@example.com)");
* when(customerRepository.save(any(Customer.class))).thenReturn(customer);
* Customer registeredCustomer = customerService.registerCustomer(customer);
* assertNotNull(registeredCustomer);
* }

#### ****5.2 Integration Testing****

* Write integration tests to validate the interaction between components (e.g., database, services, controllers).
* Use **Spring Boot Test** annotations (@SpringBootTest, @WebMvcTest) for end-to-end testing.
* @SpringBootTest
* class CustomerControllerTest {
* @Autowired
* private MockMvc mockMvc;
* @Test
* void testCreateCustomer() throws Exception {
* mockMvc.perform(post("/customers")
* .contentType(MediaType.APPLICATION\_JSON)
* .content("{\"name\":\"John\", \"email\":\"[john@example.com](mailto:john@example.com)\"}"))
* .andExpect(status().isCreated());
* }
* }

#### ****5.3 Test Coverage****

* Aim for **high test coverage** (ideally 80%+) for both unit and integration tests.
* Ensure coverage of all key paths, including positive, negative, and edge cases.

### ****6. Code Review and Collaboration Standards:****

#### ****6.1 Pull Requests and Git****

* **Commit messages**: Use descriptive commit messages that explain the "why" behind the change (e.g., Fix bug in customer registration validation).
* **Branching Strategy**: Use a branching strategy such as **Git Flow** or **GitHub Flow**:
  + main for production-ready code.
  + develop for integration testing.
  + Feature branches for new features/bug fixes.

#### ****6.2 Code Reviews****

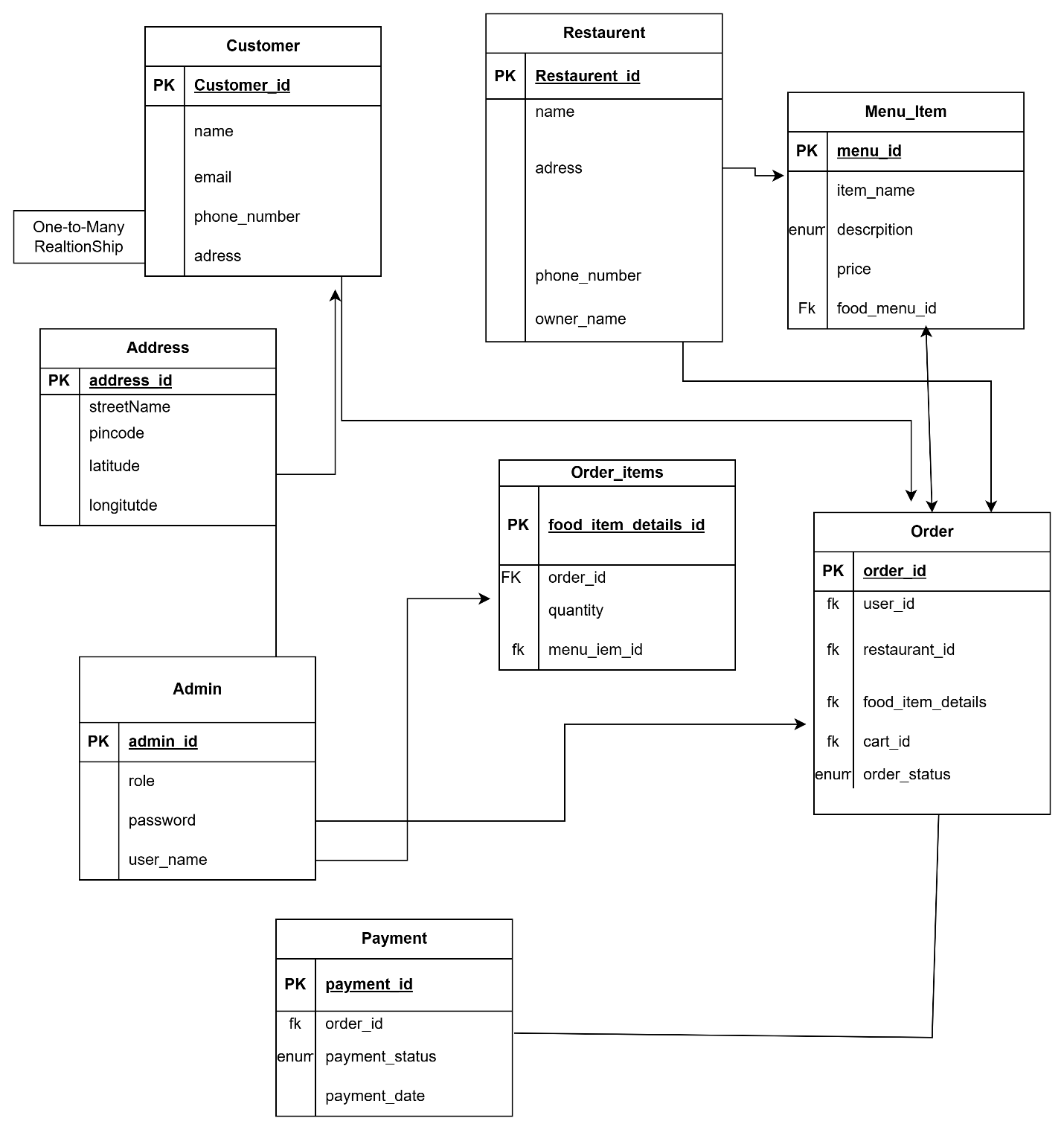
* Ensure **peer reviews** for all pull requests. Focus on:
  + Code quality and readability.
  + Consistency with project conventions.
  + Test coverage.
  + Proper error handling and edge cases.

### ****7. Security Standards:****

#### ****7.1 Authentication and Authorization****

* Use **JWT** (JSON Web Tokens) for secure session management.
* Implement **role-based access control** (RBAC) for admin and restaurant access.
* @PreAuthorize("hasRole('ROLE\_ADMIN')")
* public ResponseEntity<?> getAllOrders() {
* // implementation
* }

ER Diagram for Online \_food\_ordering system:



### ****Entities and Relationships****

1. **Customer**
   * **Attributes**:
     + customer\_id (PK)
     + name
     + email
     + phone\_number
     + address
   * **Relationships**:
     + One-to-many with **Order** (A customer can place many orders).
2. **Restaurant**
   * **Attributes**:
     + restaurant\_id (PK)
     + name
     + address
     + phone\_number
     + owner\_name
   * **Relationships**:
     + One-to-many with **Menu Item** (A restaurant has many menu items).
     + One-to-many with **Order** (A restaurant can fulfill many orders).
3. **Menu Item**
   * **Attributes**:
     + menu\_item\_id (PK)
     + name
     + description
     + price
     + restaurant\_id (FK)
   * **Relationships**:
     + Many-to-one with **Restaurant** (A menu item belongs to a single restaurant).
4. **Order**
   * **Attributes**:
     + order\_id (PK)
     + order\_date
     + status
     + customer\_id (FK)
     + restaurant\_id (FK)
   * **Relationships**:
     + Many-to-one with **Customer** (An order is placed by one customer).
     + Many-to-one with **Restaurant** (An order is fulfilled by one restaurant).
     + Many-to-many with **Menu Item** (An order can contain multiple menu items, and a menu item can be part of multiple orders). This requires a junction table like order\_items.
5. **Order\_Items (Junction Table)**
   * **Attributes**:
     + order\_id (FK)
     + menu\_item\_id (FK)
     + quantity
   * **Relationships**:
     + Many-to-one with **Order**.
     + Many-to-one with **Menu Item**.
6. **Payment**
   * **Attributes**:
     + payment\_id (PK)
     + order\_id (FK)
     + payment\_date
     + amount
     + payment\_method
     + status
   * **Relationships**:
     + One-to-one with **Order** (Each order has one payment).
7. **Admin**
   * **Attributes**:
     + admin\_id (PK)
     + username
     + password
     + role