Tasks

The project will be developed incrementally. Key implementation tasks include:

1. **Project Setup**

    - Initialize Git repositories and Gradle projects: one for each microservice (`user-service`, `restaurant-service`, `order-service`) and one for the React frontend.

    - Add dependencies: Spring Boot Starter Web, Spring Data JPA, Spring Security, `springdoc-openapi`, Lombok, MapStruct, Kafka, Liquibase, PostgreSQL driver, and testing libraries (JUnit). For React, use Create React App and install `react-router-dom`, `redux`, `react-redux`, `axios`, and a UI library.

    - Configure application properties (database URLs, JWT secret, etc.) with support for different environments (dev, prod).

    - Begin drafting a `docker-compose.yml` to orchestrate PostgreSQL instances.

2. **User Service Implementation**

    - **Entities & JPA**: Define `User`, `Address`, and `Role` entities (with `@Entity` annotations). Use `@ManyToMany` for users↔roles and `@OneToMany` for user→addresses. Create `UserRepository`, `AddressRepository`, `RoleRepository` extending `JpaRepository`.

    - **DTOs & Mappers**: Create DTO classes for user data (e.g. `UserDto`, `AddressDto`) and MapStruct mappers to convert between entities and DTOs.

    - **Services**: Implement a `UserService` with methods for registration (hash passwords), login (verify credentials), and profile management. Use `PasswordEncoder` for security.

    - **Controllers**: Build `AuthController` with `/register` and `/login` endpoints (POST). On login, generate a JWT token. Build `UserController` with `/users/me` (GET/PUT for profile). Apply `@PreAuthorize` annotations to secure endpoints (e.g. only authenticated users can view their profile).

- **Security Config**: Configure Spring Security to use a JWT filter. Define `WebSecurityConfigurerAdapter` (or newer component-based config) to permit `/auth/**` to all and protect other endpoints. Implement `JwtTokenProvider` to create/validate tokens. Ensure the server does not maintain session state, relying on the token.

3. **Restaurant Service Implementation**

   - **Entities & JPA**: Define `Restaurant` and `Dish` entities. A `Dish` has a foreign key to `Restaurant`. Create corresponding repositories (`RestaurantRepository`, `DishRepository`).

   - **DTOs & Mappers**: Create DTOs like `RestaurantDto` and `DishDto` and MapStruct mappers.

   - **Services**: Implement `RestaurantService` with methods to create, update, delete, and list restaurants, and to add/update/delete dishes for a restaurant.

   - **Controllers**:

     - Public endpoints (no auth required or only USER): `GET /restaurants` (with optional filters by cuisine or rating), `GET /restaurants/{id}`, `GET /restaurants/{id}/dishes`.

     - Admin endpoints (require `ADMIN` role): `POST /restaurants`, `PUT /restaurants/{id}`, `DELETE /restaurants/{id}`, and similarly `POST/PUT/DELETE /restaurants/{id}/dishes`.

   - **Image Handling**: Support an `imageUrl` field for dishes. This could be a URL stored as a string

4. **Order Service Implementation**

   - **Entities & JPA**: Define `Order`, `OrderItem`, and `Payment` entities (see schema above). Each `OrderItem` has foreign keys to `Order` and (optionally) `Dish` (or just store dish details). Create `OrderRepository`, `OrderItemRepository`, `PaymentRepository`.

   - **DTOs & Mappers**: Use DTOs (e.g. `OrderRequestDto`, `OrderResponseDto`) and MapStruct to map them.

- **Services**:
    - **Order Creation**: Implement `OrderService.placeOrder(userId, restaurantId, items)` that computes the total, saves the `Order` and `OrderItems`, and sets `status = "Placed"`. It should also create a `Payment` record (for the mock payment) and set status to "Paid" (simulated).
    - **Event Publishing**: After saving the order, publish an “OrderCreated” event to Kafka using a `KafkaTemplate`. The event payload could include order ID and user info.
    - **Order Update**: Methods to change order status (e.g. to Confirmed, Out for Delivery, Delivered) and record the timestamp.
- **Controllers**:
    - `POST /orders`: for users to place a new order. Requires USER role.
    - `GET /orders`: if USER role, return orders for the authenticated user; if ADMIN, return all orders.
    - `GET /orders/{id}`: returns details of one order (only if user owns it or is ADMIN).
    - `PUT /orders/{id}/status`: ADMIN only, to update the order status.
- **Payment Simulation**: The API can accept a payment method (e.g. "CreditCard") in the order request. The service simulates payment success and records it. For simplicity, assume all mock payments succeed.

5. **React Frontend Implementation**
    - **Project Structure**: Organize into `src/components`, `src/pages`, `src/redux` (actions/reducers), and `src/api` (axios instance).
    - **Authentication**:
        - Create **Login** and **Register** pages. On submit, send credentials to `/auth/login` or `/auth/register` and handle responses. Store the received JWT in localStorage and Redux state.

        - Protect routes: if not logged in, redirect to Login. Check user role (decoded from token) to allow access to admin routes.

    - **Customer UI**:

        - **RestaurantListPage**: Fetch and display restaurants. Include filters (cuisine, rating, delivery time). Display each with name and a “View Menu” button.

        - **MenuPage**: When a restaurant is selected, fetch its dishes and display them. Each dish shows image, name, description, price, and an “Add to Cart” button.

        - **Cart**: Implement a cart state (with Redux slice). Allow users to add dishes (with quantity). Provide a Cart page to review items, change quantities, and see total price.

        - **Checkout**: On the Cart page, add a “Place Order” button. On click, send `POST /orders` with user ID, restaurant ID, and item list. Show success and saved order ID.

        - **Order Tracking**: After placing an order, show an Order Status page that periodically polls `GET /orders/{id}` for status updates. Display the progression through statuses (Placed → Confirmed → Out for Delivery → Delivered) in a timeline or status bar.

    - **Admin UI** (under `/admin` route):

        - **AdminLoginPage**: Possibly reuse the same login but show admin link. Only users with ADMIN role should reach the admin dashboard.

        - **RestaurantManager**: Form and list: allow admin to add new restaurants (name, cuisine, address), and edit/delete existing ones.

        - **DishManager**: For each restaurant, list its dishes and allow add/edit/delete of dishes (fields: name, description, price, image URL).

        - **OrderManager**: Show all orders in a table (user, restaurant, total, status). Include controls to filter by status or date. Admin can click an order to update its status (e.g. mark as Confirmed or Delivered). This calls the `PUT /orders/{id}/status` API.

    - **State Management (Redux)**:

        - Store user auth (token, user data) and cart contents in Redux. Use Redux Thunk or Saga for async actions.

        - Create actions and reducers for login/logout, adding/removing items from cart, placing orders, and listing orders.

    - **API Calls**: Use Axios to call backend APIs. Configure an Axios interceptor or middleware to automatically attach the `Authorization: Bearer <token>` header when the user is authenticated.

    - **UI/UX**: Use Material-UI (MUI) or Bootstrap to build responsive layouts and forms. Validate forms on client side (e.g. required fields). Provide visual feedback (loading spinners, success/error messages).

6. **Documentation**

    - **API Documentation**: Ensure Swagger UI is available at each service (e.g. `http://localhost:8080/swagger-ui/index.html`) showing endpoints and models. Include example requests in Swagger.

    - **Postman Collection**: Create a Postman (or similar) collection that includes sample requests for all key APIs (user registration/login, restaurant CRUD, place order, update order, etc.) with example responses.

    - **README**: Write a comprehensive README explaining the system architecture, technologies used, and step-by-step setup instructions. Include how to run via Docker Compose and how to build/deploy services. Document any environment variables or prerequisites.

7. **Final Review**

    - **Code Review**: Review all code for best practices (error handling, logging, exception messages).

    - **Refinement**: Address any performance issues (e.g. database indexing, lazy loading), UI bugs, or missing validation.

    - **Polish**: Ensure UI is user-friendly, messages are clear, and errors (e.g. failed logins) are handled gracefully.