**Order Service**

* **Simple APIs for Order Management:**
  + Example endpoints:
    - POST /orders to place an order.
    - GET /orders/{id} to fetch order details.
    - GET /users/{userId}/orders to fetch a user's orders.
* **Entity Design:**
  + Order entity with fields like:

java

Copy code

public class Order {

private Long id;

private Long userId;

private List<OrderItem> items;

private String status; // e.g., "Pending", "Completed"

private Double totalPrice;

private LocalDateTime createdAt;

}

* + OrderItem entity for individual products within an order:

java

Copy code

public class OrderItem {

private Long id;

private Long orderId;

private Long productId;

private Integer quantity;

private Double price;

}

* **Integration with User Service:**
  + Verify user existence and retrieve user details during order placement.
  + Use secure communication via the resource server.

**Later: Payment Service**

* Once the order and notification services are complete, the payment service can handle payment-related workflows.
* Design it to:
  + Accept payment requests tied to orders.
  + Integrate with payment gateways.
  + Update the status of the associated order after payment confirmation.

**Advantages of This Approach**

1. **Decoupled Services:** Each service focuses on its domain logic, improving scalability and maintainability.
2. **Scalability:** Using cron jobs for notifications ensures timely processing without overloading the system.
3. **Incremental Development:** By implementing the order service before the payment service, you ensure the base workflow is functional first.

Would you like help with specific cron job configurations, API design, or database schema setup for these services?

4o