**Order Service**

The **Order Service** is responsible for managing shopping cart functionalities and processing orders. It should maintain cart and order states, support line item management, and integrate with the Price Service to fetch prices dynamically. Below are detailed instructions for implementing this service.

**Step 1: Define the Cart Entity**

1. **Purpose**:
   * The Cart entity represents a temporary storage for items a customer intends to purchase.
   * It should track the items, customer details, and the cart's state.
2. **Fields to Include**:
   * id: Unique identifier for the cart.
   * key: A user-defined unique identifier for the cart.
   * customerId: Identifier of the customer (optional if the cart is anonymous).
   * customerEmail: Email address of the customer.
   * anonymousId: Identifier for anonymous carts. Suggestion:
     + Generate a UUID when the cart is created for unauthenticated users.
   * lineItems: A list of LineItem objects representing items in the cart.
   * cartState: Tracks the cart’s current state (e.g., ACTIVE, ORDERED, CANCELLED).
3. **Unique Constraints**:
   * Ensure the key field is unique.

**Step 2: Define the LineItem Entity**

1. **Purpose**:
   * Represents an individual item added to the cart.
2. **Fields to Include**:
   * id: Unique identifier for the line item.
   * key: A unique identifier for the line item (user-defined or auto-generated).
   * productId: Identifier for the product.
   * sku: Stock Keeping Unit (specific variant of the product).
   * quantity: Quantity of the item.
   * price: Price of the item (retrieved from the Price Service).
   * totalPrice: Total price calculated as price × quantity.
3. **Behavior**:
   * Dynamically calculate totalPrice based on the quantity and price.

**Step 3: Define the Order Entity**

1. **Purpose**:
   * Represents a confirmed purchase derived from a cart.
2. **Fields to Include**:
   * id: Unique identifier for the order.
   * orderNumber: A user-defined unique identifier for the order.
   * cartId: Identifier of the cart from which the order is created.
   * paymentState: Tracks payment status (PENDING, PAID, FAILED).
   * orderState: Tracks the order's lifecycle (OPEN, CONFIRMED, CANCELLED).
3. **Constraints**:
   * Ensure orderNumber is unique across the project.

**Step 4: Create Repositories**

1. **CartRepository**:
   * Interface for interacting with the Cart entity.
   * Required Methods:
     + Save or update a cart.
     + Retrieve a cart by id or key.
     + Delete a cart.
   * Example Query:
     + Optional<Cart> findByKey(String key);
2. **OrderRepository**:
   * Interface for interacting with the Order entity.
   * Required Methods:
     + Save or update an order.
     + Retrieve orders by customer ID or order number.
     + Fetch order history for a customer.
   * Example Query:
     + List<Order> findByCustomerId(Long customerId);

**Step 5: Implement Service Layer**

1. **CartService**:
   * Core responsibilities:
     + Create a new cart.
     + Add a line item to the cart:
       - Fetch the price using the Price Service.
       - Validate the product ID and SKU.
       - Update the line item list and calculate the total price.
     + Remove a line item from the cart.
     + Change the quantity of a line item:
       - Recalculate totalPrice for the item.
     + Update the cart state (ACTIVE, ORDERED, CANCELLED).
   * Logic:
     + **Adding Line Items**:
       - Check if the line item already exists in the cart (match by productId and sku).
       - If it exists, increment the quantity.
       - If not, fetch the price and create a new line item.
     + **Cart State Transitions**:
       - Ensure a cart cannot be modified once it's in the ORDERED or CANCELLED state.
2. **OrderService**:
   * Core responsibilities:
     + Create an order from a cart:
       - Validate the cart is in ACTIVE state.
       - Copy line items and customer details to the order.
       - Mark the cart as ORDERED.
     + Update payment state (PENDING, PAID, FAILED).
     + Update order state (OPEN, CONFIRMED, CANCELLED).
     + Retrieve order history for a customer.

**Step 6: Design the Controller**

1. **CartController**:
   * Expose endpoints for managing carts.
   * Endpoints to implement:
     + POST /carts: Create a new cart.
     + POST /carts/{cartId}/line-items: Add a line item to the cart.
     + DELETE /carts/{cartId}/line-items/{lineItemId}: Remove a line item.
     + PUT /carts/{cartId}/line-items/{lineItemId}: Update the quantity of a line item.
     + PUT /carts/{cartId}/state: Update the cart state.
2. **OrderController**:
   * Expose endpoints for managing orders.
   * Endpoints to implement:
     + POST /orders: Create an order from a cart.
     + PUT /orders/{orderId}/payment-state: Update the payment state.
     + PUT /orders/{orderId}/order-state: Update the order state.
     + GET /orders/customer/{customerId}: Fetch order history for a customer.

**Step 7: Integrate with the Price Service**

1. **Fetching Prices**:
   * Use a REST client or Feign to communicate with the Price Service.
   * Fetch the price for a product by productId and sku.
   * Use the price to calculate the totalPrice of the line item.
2. **Error Handling**:
   * If the price is not available, return a meaningful error to the client.

**Step 8: Validation and Error Handling**

1. **Cart Validation**:
   * Ensure the cart exists and is in an ACTIVE state for modifications.
   * Prevent duplicate line items in the same cart.
2. **Order Validation**:
   * Ensure the cart is in the correct state before creating an order.
   * Validate that the order number is unique.
3. **Error Responses**:
   * Return appropriate HTTP status codes for errors:
     + 404 Not Found: Cart or order not found.
     + 400 Bad Request: Invalid input or state transition.

**Step 9: Testing**

1. **Unit Testing**:
   * Test the service methods for cart and order operations.
   * Validate price retrieval from the Price Service.
2. **Integration Testing**:
   * Test the complete flow: cart creation → line item addition → order creation.
3. **Edge Case Testing**:
   * Ensure invalid state transitions are rejected.
   * Test handling of unavailable prices.

**Suggestions for Anonymous Carts**

* Use a randomly generated UUID as the anonymousId for anonymous carts.
* Store the anonymousId in the client-side session or cookies to associate it with the same user across requests.