Product – Service

Use Cases

1. **Creating a Product**: When adding new products to the catalog.
2. **Updating Product Details**: To reflect any changes in the product information.
3. **Deleting Products**: When products are no longer available.
4. **Listing All Products**: For customers to browse the product catalog.
5. **Searching Products**: To provide a better user experience with search and filters.

```
@Entity
public class Product {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long productId;
   private String name;
   private String category;
   private String description;
   private Double basePrice; // Base price can act as a default price
   // No references to Vendor or Inventory here
}
```

Methods need to implement :

1. public ProductResponse createProduct(ProductRequest productRequest);
2. public ProductResponse updateProduct(Long productId, ProductRequest productRequest);
3. public void deleteProduct(Long productId);
4. public ProductResponse getProductById(Long productId);
5. public List<ProductResponse> getAllProducts()
6. public List<ProductResponse> searchProducts(String name);

```
public class ProductRequest {
   private String name;
   private String category;
   private String description;
   private Double basePrice;
}
```

```
public class ProductResponse {
   private Long productId;
   private String name;
   private String category;
   private String description;
   private Double basePrice;  }
```

Vendor – Service :

Use Cases

- **Creating a Vendor**: To onboard new vendors to the system.
- **Updating Vendor Details**: To update their contact, location, or other information.
- **Deleting Vendors**: When a vendor discontinues operations or partnerships.
- **Listing All Vendors**: For administrative purposes or integration with Inventory-Service.
- **Getting Vendors for a Product**: Supports customer-facing features like showing which vendors offer a product.

```
@Entity
public class Vendor {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long vendorId;
   private String name;
   private String contactEmail;
   private String phone;
   private String location;}
```

Methods :

1. public VendorResponse createVendor(VendorRequest vendorRequest);
2. public VendorResponse getVendorById(Long vendorId);
3. public VendorResponse updateVendor(Long vendorId, VendorRequest vendorRequest);
4. public void deleteVendor(Long vendorId);
5. public List<VendorResponse> getVendorsByProduct(Long productId);

Note :

- Vendor-Service does not have a direct relationship with products.
- The relationship between products and vendors is handled in the **Inventory-Service**.

```
public class VendorRequest {
   private String name;
   private String contactEmail;
   private String phone;
   private String location;
}
```

```
public class VendorResponse {
   private Long vendorId;
   private String name;
   private String contactEmail;
   private String phone;
   private String location;
}
```

Inventory – Service

Use Cases

1. **Adding Inventory**:
    - Vendors add product-specific inventory records for stock and pricing.
    - Example: Vendor A adds ProductId=101 with stock=20, price=499
2. **Updating Inventory**:
    - Vendors update stock for a product
    - Example: Update stock from 20 to 15 after a sale.
3. **Checking Product Availability**:
    - Used during order placement to ensure requested quantity is in stock.
    - Example: Check if 5 units of ProductId=101 are available with VendorId=1.
4. **Fetching Inventory by Product**:
    - Customers or admins retrieve all vendor-specific inventory for a product.
    - Example: Retrieve all vendors offering ProductId=101.
5. **Fetching Inventory by Vendor**:
    - Admins retrieve all products managed by a vendor.
    - Example: Retrieve all inventory records for VendorId=2.

```
@Entity
public class Inventory {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long inventoryId;
   private Long productId; // Reference to Product (managed in Product-Service)
   private Long vendorId;  // Reference to Vendor (managed in Vendor-Service)
   private Integer stock;
   private Double price;
   private String deliveryTime;
   // Optional: Add timestamps or other tracking fields
}
```

Methods :

1. public InventoryResponse addInventory(InventoryRequest inventoryRequest);
   - **Description**: Adds a new inventory record for a product by a specific vendor.
   - **Input**: InventoryRequest (details like productId, vendorId, stock, price, deliveryTime, etc.)
2. public InventoryResponse updateInventory(Long inventoryId, InventoryRequest inventoryRequest);
3. public void deleteInventory(Long inventoryId);
4. public List<InventoryResponse> getInventoryByProduct(Long productId);
   - **Description**: Lists all inventory records for a given product across vendors.
5. public List<InventoryResponse> getInventoryByVendor(Long vendorId);
6. public AvailabilityResponse checkAvailability(Long productId, Long vendorId, Integer quantity);
   - **Description**: Checks if a product is available in stock for a given quantity and vendor

public class InventoryRequest {
private Long productId;
private Long vendorId;
private Integer stock;
private Double price;
private String deliveryTime;
}
	public class InventoryResponse {
	private Long inventoryId;
	private Long productId;
	private Long vendorId;
	private Integer stock;
	private Double price;
	private String deliveryTime;
	}
		public class AvailabilityResponse {
		private boolean isAvailable;
		private Integer availableStock;
		}

Scenarios and Workflow

1. **Placing an Order**:
   - **Step 1**: Order-Service calls Inventory-Service to check stock availability.
   - **Step 2**: If available, Inventory-Service reduces stock accordingly.
2. **Vendor Adds/Updates Inventory**:
   - Vendor-Service calls Inventory-Service to add/update records.

Order-Service

- **Purpose**: Handles order placement, order item details, and order management (status updates, retrieval).
- **Key Relationships**:
    - Communicates with **Customer-Service, Address-service** to validate customer details.
    - Communicates with **Inventory-Service** to verify and reserve stock.
    - Communicates with **Payment-Service** for payment processing.
    - Communicates with **Notification-Service** to notify customers.

Workflow for Placing an Order

1. **Validate Customer**:
    - Use Customer-Service, Address-service to verify if the customerId, addressId exists.
2. **Check Inventory**:
    - For each order item:
        - Call Inventory-Service to check stock availability for productId and vendorId.
3. **Reserve Stock**:
    - If stock is available, call Inventory-Service to reserve the requested quantity.
4. **Calculate Total Amount**:
    - Sum up the total price for all order items.
5. **Check Payment Method**:
    - Ensure the provided payment method is supported.
6. **Process Payment**:
    - Interact with a payment gateway to process the transaction.
7. **Update Payment Status**:
    - Based on the response from the payment gateway, set the payment status (Success, Failed, Pending).
8. **Notify Order-Service**:
    - Notify Order-Service of the payment status (e.g., to move the order to the "Confirmed" status upon successful payment).
9. **Save Order**:
    - Save the order and its associated order items in the database.
10. **Notify Customer**:
    - Use Notification-Service to notify the customer about the order placement.

Methods :

1. public OrderResponse placeOrder(OrderRequest orderRequest);
2. public OrderResponse getOrderById(Long orderId);
3. public List<OrderResponse> getOrdersByCustomer(Long customerId);
4. public OrderResponse updateOrderStatus(Long orderId, OrderStatusUpdateRequest statusUpdateRequest);
5. public void cancelOrder(Long orderId);
6. public List<OrderResponse> getOrdersBetweenDates(LocalDate startDate, LocalDate endDate);
7. public List<OrderResponse> getOrdersByCustomerIdAndDateRange(Long customerId,LocalDate startDate, LocalDate endDate);

```
@Entity
public class Order {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long orderId;
   private Long customerId; // Reference to Customer-Service
   private Long shippingAddressId;
   private LocalDateTime orderDate;
   private String status; // "Placed", "Shipped", "Delivered",
   private Double totalAmount;
   @OneToMany(cascade = CascadeType.ALL, fetch =
FetchType.LAZY)
   @JoinColumn(name = "order_id")
   private List<OrderItem> orderItems;
   }
```

```
@Entity
public class OrderItem {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long orderItemId;
private Long productId; // Reference to Product-Service
private Long vendorId;  // Reference to Inventory-Service
private Integer quantity;
private Double price; // Price at the time of order
private Double total; // price * quantity
}
```

```
public class OrderRequest {
   private Long customerId;
   private Long addressId;
   private List<OrderItemRequest> items;
}
```

```
public class OrderItemRequest {
   private Long productId;
   private Long vendorId;
   private Integer quantity;
}
```

```
public class OrderResponse {
   private Long orderId;
   private Long customerId;
   private LocalDateTime orderDate;
   private String status;
   private Double totalAmount;
   private List<OrderItemResponse> orderItems;
   private AddressResponse address;
}
```

```
public class OrderItemResponse {
   private Long productId;
   private Long vendorId;
   private Integer quantity;
   private Double price;
   private Double total;
}
```

Payment – Service :

Use Cases

1. **Initiate Payment**
   - A customer places an order, and the system sends a payment request to the **Payment-Service**.
   - The service processes the payment and updates the status as **SUCCESS**, **FAILED**, or **PENDING**.
2. **Verify Payment Status**
   - Other services (e.g., **Order-Service**) can query the **Payment-Service** to check the status of a payment for a given order.
3. **Payment History Retrieval**
   - Customers can view their payment history for all orders.
   - Admins can fetch payment details for audit or reporting purposes.
4. **Support for Multiple Payment Methods**
   - Handles credit/debit cards, UPI, net banking, wallets, etc.
5. **Payment Gateway Integration**
   - Integrates with external payment gateways for real-time transaction processing.

Methods :

1. public PaymentResponse processPayment(PaymentRequest paymentRequest);
2. public PaymentResponse getPaymentDetails(Long paymentId);
3. public PaymentResponse getPaymentByOrderId(Long paymentId);
4. public List<PaymentResponse> getPaymentsByCustomerId(Long orderId);
5. public List<PaymentResponse> getAllPayments();

```
@Entity
public class Payment {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long paymentId;
   private Long orderId; // Reference to Order-Service
   private Long customerId; // Reference to Customer-Service
   private Double amount;
   private String paymentMethod; // e.g., "Credit Card", "UPI", "Net Banking"
   private LocalDateTime paymentDate;
   private String status; // "Success", "Failed", "Pending"
}
```

```
public class PaymentRequest {
   private Long orderId;
   private Long customerId;
   private Double amount;
   private String paymentMethod; // e.g., "Credit Card",
"UPI", "Net Banking"
}
```

```
public class PaymentResponse {
   private Long paymentId;
   private Long orderId;
   private Long customerId;
   private Double amount;
   private String paymentMethod;
   private LocalDateTime paymentDate;
   private String status;
}
```

Notification – Service :-

Use Cases

1. **Order Confirmation Notifications**
    - Sends order confirmation emails or SMS to customers after placing an order.
2. **Payment Status Notifications**
    - Notifies customers of payment success, failure, or pending status.
3. **Delivery Updates**
    - Sends real-time updates about the status of the delivery (e.g., dispatched, out for delivery, delivered).
4. **Custom Notifications**
    - Allows for sending custom messages to specific users for special cases.

Integration with Other Microservices

- Trigger notifications for events like order placement, payment success/failure, or delivery updates.
- Subscribe to events via a message broker (e.g., RabbitMQ or Kafka).

Integration Flow Example
Order Placement Notification:

1. Order-Service places an order.
2. Order-Service sends an event (e.g., OrderPlacedEvent) to a message broker.
3. Notification-Service listens to the event and sends:
    - **Email**: "Your order has been successfully placed."
    - **SMS**: "Order #12345 confirmed."

```
@Entity
public class Notification {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;

   private Long userId;  // Reference to the recipient
   private String channel;  // EMAIL, SMS, PUSH
   private String message;
   private String subject;  // For email notifications
   private String status;  // PENDING, SENT, FAILED
   private LocalDateTime createdAt;
   private LocalDateTime sentAt;
}
```

Methods :

1. NotificationResponse sendNotification(NotificationRequest notificationRequest);
2. List<NotificationResponse> getNotificationsByUserId(Long userId);
3. NotificationResponse getNotificationById(Long notificationId);
4. List<NotificationResponse> getAllNotifications();
5. void deleteNotification(Long notificationId);

```java
@Data
public class NotificationRequest {
    private Long userId;
    private String channel; // EMAIL, SMS, PUSH
    private String subject;
    private String message;
}
```

```java
@Data
public class NotificationResponse {
    private Long notificationId;
    private Long userId;
    private String channel;
    private String subject;
    private String message;
    private String status; // SENT, FAILED, PENDING
    private LocalDateTime createdAt;
    private LocalDateTime sentAt;
}
```

Customer – Service

Use Cases

1. **Customer Registration**:
   - A customer registers using createCustomer.
2. **Profile Management**:
   - Customers update their profile using updateCustomer.
3. **Customer Deletion**:
   - Admins can delete customers who no longer use the platform.
4. **Verification**:
   - Other services (e.g., Order-Service) can verify if a customer is active before processing their requests.
5. **View Details**:
   - Customers can view their profile using getCustomerById.
6. **Admin Overview**:
   - Admins can fetch all customers for reporting or monitoring

```
@Entity
@Table(name = "customers")
public class Customer {

   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long customerId;
   private String firstName;
   private String lastName;
   private String email;
   private String phone;
   private String status; // ACTIVE, INACTIVE, etc.
}
```

Methods :

1. CustomerResponse createCustomer(CustomerRequest customerRequest);
2. CustomerResponse updateCustomer(Long customerId, CustomerRequest customerRequest);
3. CustomerResponse getCustomerById(Long customerId);
4. List<CustomerResponse> getAllCustomers();
5. void deleteCustomer(Long customerId);
6. void activateCustomer(Long customerId);

```
package com.customer.dto;

public class CustomerRequest {
   private String firstName;
   private String lastName;
   private String email;
   private String phone;
   private String address;
}
```

```
package com.customer.dto;

import java.util.List;

public class CustomerResponse {
   private Long customerId;
   private String firstName;
   private String lastName;
   private String email;
   private String phone;
   private List<AddressResponse> addresses; // List of
addresses if fetched with customer details
}
```

Address- Service

Use Cases

- **Create Address**: When a customer adds a new shipping or billing address during the registration process or when updating their profile.
- **Get Customer's Addresses**: The customer can view all their saved addresses.
- **Update Address**: A customer may want to update the details of an existing address (e.g., if they move to a new house).
- **Delete Address**: A customer may choose to delete an address that is no longer needed (e.g., after moving).
- **Search Address**: When an order is placed, the address linked to the customer should be retrieved for shipping details.

```
@Entity
public class Address {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   private Long id;
   private Long customerId;  // Foreign Key to Customer
   private String line1;     // Address Line 1
   private String city;      // City
   private String state;     // State
   private String postalCode; // Postal Code
   private String country;    // Country
   private String phoneNumber; // Optional for delivery contact
}
```

Methods :

1. public AddressResponse createAddressForCustomer(AddressRequest addressRequest);
2. public AddressResponse getAddressById(Long addressId);
3. public List<AddressResponse> getAddressesByCustomer(Long customerId);
4. public AddressResponse updateAddress(Long addressId, AddressRequest addressRequest);
5. public void deleteAddress(Long addressId);