Smart E-commerce

🚀 Project Idea: "Smart eCommerce Platform with Real-Time Order Processing & Inventory Management"

Service	Purpose
1. Product Service	Manages product catalog: CRUD operations for products, categories, images, and descriptions.
2. Order Service	Handles order placement, tracking, updates, cancellations. Manages order lifecycle and status.
3. Cart Service	Allows users to add/remove products to their shopping cart before placing an order.
4. User/Customer Service	Handles user registration, authentication, profile management, addresses.
5. Payment Service	Manages payment transactions via gateways like Stripe/PayPal. Confirms success/failure.
6. Inventory Service	Tracks product stock. Decreases stock on order placement. Sends low stock alerts.
7. Shipping/Delivery Service	Manages delivery, shipping status, tracking numbers, courier integration.
8. Notification Service	Sends order confirmations, shipping updates via email, SMS, or push notifications.

Service	Purpose
9. API Gateway (Spring Cloud Gateway)	Single entry point for all microservices. Handles routing, authentication, throttling.
10. Authentication & Authorization Service (OAuth2 / JWT)	Issues and verifies login tokens. Manages user roles and permissions.
11. Service Registry (Eureka/Consul)	Enables dynamic discovery of services. Supports load balancing and failover.

12. Configuration Service (Spring Cloud Config)	Centralized configuration management for all services (DB URLs, keys, ports, etc.).
13. Message Broker (RabbitMQ / Kafka)	Enables asynchronous communication between services. Useful for event-driven patterns.
14. API Documentation Service (Swagger/OpenAPI)	Provides interactive API docs for all services (can be centralized via API Gateway).

Tool	Type	Purpose
15. Spring Boot Actuator	Library	Adds health, metrics, environment info to each Spring Boot service.
16. Prometheus	Monitoring	Collects metrics (CPU, memory, request latency, custom metrics) from services.
17. Grafana	Dashboard	Displays dashboards and graphs from Prometheus metrics.
18. ELK Stack	Logging	Consists of:

Here's a complete list of **Product Service APIs** tailored for a **microservices-based eCommerce system**.

✅ Product Service – Full API Design (Microservice Architecture)

📍 **Base URL:** `/api/products`

This service handles product catalog operations — creation, update, retrieval, categorization, etc.

📌 1. Create Product

🔒 **Admin only**

POST `/api/products`

```
Request:
{
```

```
  "name": "iPhone 15 Pro",
  "description": "Latest iPhone with A17 chip",
  "price": 1299.00,
  "category": "Smartphones",
  "brand": "Apple",
  "imageUrls": ["https://cdn.example.com/iphone15.jpg"]
}
```

```
Response:
{
  "productId": 1,
  "message": "Product created successfully"
}
```

📌 2. Update Product by ID

🔒 **Admin only**

PUT `/api/products/{productId}`

```
Request:
{
  "price": 1249.00,
  "description": "Updated iPhone 15 with new specs"
}
```

```
Response:
{
  "message": "Product updated successfully"
}
```

📌 3. Delete Product by ID

🔒 **Admin only**

DELETE `/api/products/{productId}`

```
Response:
{
  "message": "Product deleted successfully"
}
```

📌 4. Get Product by ID

GET `/api/products/{productId}`

```
Response:
{
  "productId": 1,
  "name": "iPhone 15 Pro",
  "description": "Latest iPhone with A17 chip",
  "price": 1299.00,
  "category": "Smartphones",
  "brand": "Apple",
  "imageUrls": ["https://cdn.example.com/iphone15.jpg"],
  "createdAt": "2025-06-29T12:00:00Z"
}
```

📌 5. Get All Products

GET `/api/products`

✅ Supports optional query params:

- `category`, `brand`, `minPrice`, `maxPrice`, `search`, `sort`, `page`, `size`

```
GET /api/products?category=Smartphones&minPrice=500&maxPrice=2000&page=0&size=10
```

```
Response:
{
  "content": [
    {
      "productId": 1,
```

```
      "name": "iPhone 15 Pro",
      "price": 1299.00,
      "category": "Smartphones"
    },
    ...
  ],
  "totalPages": 3,
  "totalItems": 24
}
```

📌 6. Get All Categories

GET `/api/products/categories`

```
Response:
[
  "Smartphones",
  "Laptops",
  "Accessories",
  "Wearables"
]
```

📌 7. Check Product Availability (for Order/Cart)

📦 Called internally by Cart/Order/Inventory services

POST `/api/products/check-availability`

```
Request:
[
  { "productId": 1, "quantity": 2 },
  { "productId": 5, "quantity": 1 }
]
```

```
Response:
[
  { "productId": 1, "available": true, "price": 1299.00 },
```

```
  { "productId": 5, "available": false, "reason": "Out of stock" }
]
```

📌 8. Bulk Get Products by IDs (for internal microservice use)

POST `/api/products/bulk`

```
Request:
[1, 2, 5]
```

```
Response:
[
  { "productId": 1, "name": "iPhone 15", "price": 1299.00 },
  { "productId": 2, "name": "MacBook Pro", "price": 2399.00 },
  { "productId": 5, "name": "AirPods Pro", "price": 249.00 }
]
```

📌 9. Upload Product Images

🔒 Admin

POST `/api/products/{productId}/images`

(multipart file upload)

📌 10. Health Check / Info

For Spring Actuator (Eureka, monitoring)

GET `/api/products/actuator/health`

GET `/api/products/actuator/info`

🧠 Microservice Integration

Service	Purpose	Mode
Cart	Validate price, stock	REST / Feign

Order	Fetch product details	REST / Feign
Inventory	Sync stock after creation	Messaging (event)

✅ Optional: On product create/update/delete, you can emit events like:

```
Event: product.updated
{
  "productId": 1,
  "name": "iPhone 15 Pro",
  "price": 1299.00
}
```

Would you like a **Spring Boot implementation** next (entity + controller + service + repository)? Or setup Swagger/OpenAPI for this?

Let me know what part you want to dive into next!

Absolutely! Below is the **full list of APIs for the Order Service**, fully adapted for a **microservices-based eCommerce system**.

✅ Order Service – Full API List (Microservice Architecture)

📍 **Base URL:** `/api/orders`

📌 1. Place Order

POST `/api/orders`

Creates a new order for the current user.

```
Request:
{
  "shippingAddress": "123 Main St, New York, NY"
}
```

- ✅ Authenticated user ID is extracted from JWT.
- ✅ Cart Service is called internally to get items.
- ✅ Product and Inventory Services are queried for validation.

- ✅ Emits `order.placed` event to Kafka/RabbitMQ.

```
Response:
{
  "orderId": 1001,
  "status": "PLACED",
  "totalAmount": 1999.00,
  "createdAt": "2025-06-29T14:05:00Z"
}
```

📌 2. Get Order by ID

GET `/api/orders/{orderId}`

```
Response:
{
  "orderId": 1001,
  "userId": 10,
  "status": "PLACED",
  "items": [
    { "productId": 1, "quantity": 1, "price": 999.00 }
  ],
  "totalAmount": 999.00,
  "shippingAddress": "123 Main St, NY",
  "createdAt": "2025-06-29T14:05:00Z"
}
```

📌 3. Get Orders for Authenticated User

GET `/api/orders/user`

Returns all orders for the logged-in user.

```
Response:
[
  {
    "orderId": 1001,
    "status": "PLACED",
```

```
    "totalAmount": 999.00,
    "createdAt": "2025-06-29T14:05:00Z"
  },
  {
    "orderId": 1002,
    "status": "DELIVERED",
    "totalAmount": 199.00,
    "createdAt": "2025-05-01T12:00:00Z"
  }
]
```

📌 4. Admin: Get All Orders

GET `/api/orders`

🔒 Requires Admin Role

```
Response:
[
  {
    "orderId": 1001,
    "userId": 10,
    "status": "PLACED",
    "totalAmount": 999.00
  },
  ...
]
```

📌 5. Update Order Status (Admin/Internal Use)

PATCH `/api/orders/{orderId}/status`

```
Request:
{
  "status": "SHIPPED"
}
```

```
Response:
{
  "message": "Order status updated to SHIPPED"
}
```

✅ Emits event `order.status.updated`

📌 6. Cancel Order

DELETE `/api/orders/{orderId}`

- Only allowed for user if status is `PLACED` or `PROCESSING`
- Internally:
  - Releases reserved stock via Inventory Service
  - Sends cancellation notification
  - Emits `order.cancelled` event

```
Response:
{
  "message": "Order cancelled"
}
```

📌 7. Get Order Status History (Optional)

GET `/api/orders/{orderId}/status-history`

```
Response:
[
  { "status": "PLACED", "timestamp": "2025-06-29T14:05:00Z" },
  { "status": "PROCESSING", "timestamp": "2025-06-29T14:10:00Z" }
]
```

📌 8. Calculate Order Total (Optional Preview)

POST `/api/orders/preview`

Used by frontend to preview total without placing order.

```
Request:
{
  "items": [
    { "productId": 1, "quantity": 2 },
    { "productId": 3, "quantity": 1 }
  ]
}
```

```
Response:
{
  "totalAmount": 2798.99
}
```

📌 9. Webhook Endpoint (Kafka Event Consumers)

(Internal only – not REST, but handled via Kafka)

- `order.placed` → triggers Payment, Shipping
- `payment.success` → update status to `PAID`
- `payment.failed` → update status to `CANCELLED`
- `shipment.delivered` → update to `DELIVERED`

📌 10. Health Check / Info

GET `/api/orders/actuator/health`

GET `/api/orders/actuator/info`

🧱 Internal Responsibilities (Microservice Only)

Task	Mechanism
Get cart items	Call Cart Service (Feign/REST)
Fetch product/price	Call Product Service
Reserve inventory	Call Inventory Service
Trigger payment	Publish `order.placed` event
Notify user	Publish to Notification topic

Auth	Gateway + JWT

Would you like the **Spring Boot code (controllers/services)** for this next? Or a **Kafka event publisher/subscriber** for `order.placed` and `payment.success`?

Let me know — we can build it out next!

Here's a complete list of **Cart Service APIs** tailored for a microservices-based eCommerce system.

✅ Cart Service – Full API List (Microservice Architecture)

📍 **Base URL:** `/api/cart`

This service manages the customer's shopping cart — adding/removing items, updating quantity, clearing, and syncing with product service.

📌 1. Add Item to Cart

POST `/api/cart/items`

```
Request:
{
  "productId": 1,
  "quantity": 2
}
```

✅ Authenticated user ID is derived from JWT via API Gateway.

```
Response:
{
  "message": "Item added to cart"
}
```

📌 2. Update Quantity of a Cart Item

PUT `/api/cart/items/{productId}`

```
Request:
{
  "quantity": 5
}
```

```
Response:
{
  "message": "Cart item quantity updated"
}
```

📌 3. Remove Item from Cart

DELETE `/api/cart/items/{productId}`

```
Response:
{
  "message": "Item removed from cart"
}
```

📌 4. Get Cart for Current User

GET `/api/cart`

```
Response:
{
  "userId": 101,
  "items": [
    {
      "productId": 1,
      "productName": "iPhone 15 Pro",
      "price": 1299.00,
      "quantity": 2,
      "subtotal": 2598.00
    },
    {
      "productId": 3,
```

```
      "productName": "AirPods",
      "price": 199.00,
      "quantity": 1,
      "subtotal": 199.00
    }
  ],
  "totalAmount": 2797.00
}
```

📌 5. Clear Cart

DELETE `/api/cart/clear`

```
Response:
{
  "message": "Cart cleared"
}
```

📌 6. Sync Product Info (for internal use)

POST `/api/cart/sync-products`

> Triggered by Product Service (e.g., price or name change)

```
Request:
{
  "productId": 1,
  "price": 1299.00,
  "name": "iPhone 15 Pro Max"
}
```

```
Response:
{
  "message": "Product info synced in carts"
}
```

📌 7. Check if Cart is Empty (for Order Service)

GET `/api/cart/is-empty`

Used internally before order placement.

```
Response:
{
  "isEmpty": false
}
```

📌 8. Get Cart Items in Bulk (for Order Service)

GET `/api/cart/items`

➡️ Returns only raw productId, quantity (used during checkout)

```
Response:
[
  { "productId": 1, "quantity": 2 },
  { "productId": 3, "quantity": 1 }
]
```

📌 9. Health Check & Info

GET `/api/cart/actuator/health`

GET `/api/cart/actuator/info`

🧠 Microservice Integration

Service	Purpose	Interaction
Product	Get product info (price, name)	REST/Feign
Order	Fetch cart items for order placement	REST (internal)
Auth Service	Get user identity from JWT	API Gateway

🛠️ Optional Features

- ✅ Redis support for high-performance carts

- ✅ Auto-clear cart after successful order (consume `order.placed`)
- ✅ Event-based product info sync (`product.updated`)

Would you like the **Spring Boot controller + service + entity class** for this Cart Service?

Or should we scaffold Kafka integration next? Let me know where you want to go from here!

Here is a complete list of **User/Customer Service APIs** designed for a **microservices-based eCommerce platform**.

✅ User/Customer Service – Full API List

📍 **Base URL:** `/api/users`

Handles user registration, profile management, role assignments, and account status.

📌 1. Register New User (Sign Up)

POST `/api/users/register`

```
Request:
{
  "firstName": "John",
  "lastName": "Doe",
  "email": "john.doe@example.com",
  "password": "StrongPassword123",
  "phone": "+1234567890"
}
```

```
Response:
{
  "message": "User registered successfully",
  "userId": 101
}
```

> 🔐 Password is hashed internally before storing.

📌 2. Authenticate User (Delegated to Auth Service)

Handled via OAuth2 / JWT in a separate **Authentication Service**

Not directly exposed here.

📌 3. Get Current User Profile

GET `/api/users/me`

🔒 Requires JWT (handled via API Gateway)

```
Response:
{
  "userId": 101,
  "firstName": "John",
  "lastName": "Doe",
  "email": "john.doe@example.com",
  "phone": "+1234567890",
  "roles": ["CUSTOMER"],
  "status": "ACTIVE"
}
```

📌 4. Update Current User Profile

PUT `/api/users/me`

```
Request:
{
  "firstName": "Johnny",
  "phone": "+1987654321"
}
```

```
Response:
{
  "message": "Profile updated"
}
```

📌 5. Change Password

POST `/api/users/change-password`

```
Request:
{
  "oldPassword": "OldPass123",
  "newPassword": "NewPass456"
}
```

```
Response:
{
  "message": "Password changed successfully"
}
```

📌 6. Get User by ID (Admin/Internal Use)

GET `/api/users/{userId}`

🔒 Admin only or internal services (like Order or Notification)

```
Response:
{
  "userId": 101,
  "firstName": "John",
  "email": "john.doe@example.com",
  "status": "ACTIVE"
}
```

📌 7. Get All Users (Admin Only)

GET `/api/users`

Supports pagination and filtering.

```
GET /api/users?page=0&size=10&status=ACTIVE
```

```
Response:
{
  "content": [
    {
      "userId": 101,
      "firstName": "John",
      "email": "john.doe@example.com",
      "roles": ["CUSTOMER"],
      "status": "ACTIVE"
    }
  ],
  "totalPages": 2,
  "totalItems": 15
}
```

📌 8. Deactivate User Account (Soft Delete)

PATCH `/api/users/{userId}/deactivate`

🔒 Admin or Self

```
Response:
{
  "message": "User deactivated"
}
```

📌 9. Assign Roles to User

🔒 Admin only

PATCH `/api/users/{userId}/roles`

```
Request:
{
  "roles": ["CUSTOMER", "SELLER"]
}
```

```
Response:
{
  "message": "Roles updated"
}
```

📌 10. Internal API: Get Basic Info by User ID

GET `/api/users/basic/{userId}`

➡️ Used by Order, Payment, Notification services

```
Response:
{
  "userId": 101,
  "fullName": "John Doe",
  "email": "john.doe@example.com",
  "phone": "+1234567890"
}
```

📌 11. Health Check & Info

GET `/api/users/actuator/health`

GET `/api/users/actuator/info`

🧠 Microservice Integrations

Service	Purpose	Interaction
Order	Get user info (name/email)	REST / Feign
Auth Service	Handle login, token, roles	OAuth2/JWT
Notification	Get email/phone for messages	REST or event
Shipping	Address management (if included here)	Optional add-on

---

✅ Optional Features

- ✅ Email/phone verification APIs

- ✅ Address management (or separate Address Service)
- ✅ GDPR delete/export APIs

Would you like me to generate the **Spring Boot code structure** (entity, controller, service) for this User Service next?

Or build the **OAuth2 Auth Service** that works with this user data?

Let me know how you'd like to continue!

Here is a complete list of **Payment Service APIs** designed for a **microservices-based eCommerce architecture**.

✅ Payment Service – Full API Design

📍 **Base URL:** `/api/payments`

This service handles payment initialization, status tracking, webhook handling, refunds, and integration with third-party payment gateways like Stripe, Razorpay, or PayPal.

📌 1. Initiate Payment for an Order

POST `/api/payments/initiate`

```
Request:
{
  "orderId": "ORD-1001",
  "amount": 2499.00,
  "currency": "USD",
  "paymentMethod": "CARD", // or "UPI", "NET_BANKING"
  "returnUrl": "https://example.com/order-confirmation"
}
```

```
Response:
{
  "paymentId": "PAY-8901",
  "paymentGatewayUrl": "https://stripe.com/checkout/session/xyz",
  "status": "PENDING"
}
```

> 🔁 The Payment Service stores orderId, status, and waits for success/failure notification.

📌 2. Confirm Payment (Webhook / Polling)

POST `/api/payments/confirm`

✅ Triggered by external gateway (webhook) or internal polling

```
Request:
{
  "paymentId": "PAY-8901",
  "status": "SUCCESS",
  "transactionId": "TXN12345678",
  "paidAt": "2025-06-29T12:30:00Z"
}
```

```
Response:
{
  "message": "Payment confirmed"
}
```

> This will publish an event like payment.success to RabbitMQ/Kafka.

📌 3. Get Payment Status by Order ID

GET `/api/payments/status/{orderId}`

```
Response:
{
  "paymentId": "PAY-8901",
  "orderId": "ORD-1001",
  "status": "SUCCESS",
  "amount": 2499.00,
  "method": "CARD",
```

```
  "transactionId": "TXN12345678"
}
```

📌 4. Cancel Payment / Timeout Expiry

POST `/api/payments/cancel/{paymentId}`

Used when user closes the payment page or time runs out.

```
Response:
{
  "message": "Payment canceled"
}
```

📌 5. Issue Refund

POST `/api/payments/refund`

```
Request:
{
  "paymentId": "PAY-8901",
  "reason": "Product returned",
  "amount": 2499.00
}
```

```
Response:
{
  "refundId": "RF-5001",
  "status": "INITIATED"
}
```

📌 6. Get Refund Status

GET `/api/payments/refund/{refundId}`

```
Response:
{
```

```
  "refundId": "RF-5001",
  "paymentId": "PAY-8901",
  "amount": 2499.00,
  "status": "COMPLETED",
  "refundedAt": "2025-06-30T08:10:00Z"
}
```

📌 7. List Payment History for User

GET `/api/payments/history?userId=101&page=0&size=10`

```
Response:
{
  "content": [
    {
      "paymentId": "PAY-001",
      "orderId": "ORD-001",
      "status": "SUCCESS",
      "amount": 100.0
    },
    ...
  ],
  "totalPages": 5
}
```

📌 8. Health Check & Info

GET `/api/payments/actuator/health`

GET `/api/payments/actuator/info`

🧠 Microservice Integrations

Service	Purpose	Integration Type
Order Service	Get payment status to confirm order	REST or Event-based
Notification	Send payment success/failure messages	Kafka / RabbitMQ

Auth Service	Authenticated payment request (JWT check)	Gateway pass-through
Inventory	Trigger reservation only if payment succeeds	Kafka Event (`payment.success`)

🛠️ Optional/Advanced APIs

- `/webhook/stripe`, `/webhook/razorpay` → external gateway callbacks
- `/simulate` → for test/sandbox payments
- `/fees/estimate` → if dynamic transaction fees exist

Would you like to see:

- A **Spring Boot starter template** for this Payment Service?
- Or the **event/message format** for integration with Kafka/RabbitMQ?

Let me know where you'd like to continue!

Here's a complete list of **Inventory Service APIs** for a microservices-based eCommerce architecture. This service tracks and manages stock levels for products and works closely with the **Product**, **Order**, **Cart**, and **Payment** services.

✅ Inventory Service – Full API Design

📍 **Base URL:** `/api/inventory`

📌 1. Check Inventory for a Product

GET `/api/inventory/{productId}`

```
Response:
{
  "productId": 101,
  "availableQuantity": 45,
  "reservedQuantity": 5,
  "status": "IN_STOCK"
}
```

📌 2. Bulk Inventory Check (Used by Cart/Order)

POST `/api/inventory/check`

```
Request:
[
  { "productId": 101, "quantity": 2 },
  { "productId": 102, "quantity": 5 }
]
```

```
Response:
[
  { "productId": 101, "available": true },
  { "productId": 102, "available": false, "availableQuantity": 3 }
]
```

📌 3. Reserve Inventory (Before Payment)

POST `/api/inventory/reserve`

Used by the **Order Service** after cart validation.

```
Request:
{
  "orderId": "ORD-1001",
  "items": [
    { "productId": 101, "quantity": 2 },
    { "productId": 102, "quantity": 1 }
  ]
}
```

```
Response:
{
  "message": "Inventory reserved successfully",
  "status": "RESERVED"
}
```

📌 4. Release Inventory (Payment Failed or Order Cancelled)

POST `/api/inventory/release`

```
Request:
{
  "orderId": "ORD-1001"
}
```

```
Response:
{
  "message": "Inventory released for order ORD-1001"
}
```

📌 5. Deduct Inventory After Payment Success

POST `/api/inventory/deduct`

Called once payment is confirmed.

```
Request:
{
  "orderId": "ORD-1001"
}
```

```
Response:
{
  "message": "Inventory deducted"
}
```

📌 6. Add/Update Stock (Admin Use)

POST `/api/inventory/update-stock`

```
Request:
{
  "productId": 101,
```

```
  "quantity": 100
}
```

```
Response:
{
  "message": "Stock updated successfully"
}
```

📌 7. Get All Low Stock Products (For Alerting)

GET `/api/inventory/low-stock?threshold=10`

```
Response:
[
  {
    "productId": 105,
    "productName": "Wireless Mouse",
    "availableQuantity": 8
  },
  ...
]
```

📌 8. Health Check & Info

GET `/api/inventory/actuator/health`

GET `/api/inventory/actuator/info`

🧠 Microservice Integrations

Service	Purpose	Type
Product	Sync product creation for stock	REST / Event
Order	Reserve, release, and deduct stock	REST / Kafka/RMQ
Cart	Stock validation before checkout	REST
Payment	Only deduct stock on payment success	Event-based
Notification	Notify when stock is low	Kafka/RMQ Event

✅ Optional Events (Kafka / RabbitMQ)

- `stock.low` → Notification/alert
- `stock.reserved` → Order Service
- `stock.released` → Order Cancelled
- `stock.deducted` → Payment Success

📦 Bonus: Suggested Entity Model (Simplified)

```
@Entity
public class Inventory {
    private Long productId;
    private Integer availableQuantity;
    private Integer reservedQuantity;
    private LocalDateTime lastUpdated;
}
```

Would you like the **Spring Boot code template (Controller, Service, Entity)** for this Inventory Service next?

Or should we move to **Shipping/Delivery Service** APIs?

Let me know your next step!

Here is a complete list of **Shipping/Delivery Service APIs** tailored for a **microservices-based eCommerce architecture**.

✅ Shipping/Delivery Service – Full API Design

📍 **Base URL:** `/api/shipping`

The Shipping Service manages shipping address, delivery assignment, tracking, and status updates. It works with Order, Notification, and User Services.

📌 1. Create Delivery Request

POST `/api/shipping/create`

Triggered by the **Order Service** once the payment is confirmed.

```
Request:
{
  "orderId": "ORD-1001",
  "userId": 101,
  "deliveryAddress": {
    "name": "John Doe",
    "phone": "+1234567890",
    "line1": "123 Main Street",
    "line2": "Apt 4B",
    "city": "New York",
    "state": "NY",
    "postalCode": "10001",
    "country": "USA"
  },
  "items": [
    { "productId": 101, "quantity": 2 },
    { "productId": 102, "quantity": 1 }
  ]
}
```

```
Response:
{
  "deliveryId": "DEL-5011",
  "status": "CREATED",
  "estimatedDeliveryDate": "2025-07-03"
}
```

📌 2. Get Delivery Info by Order ID

GET `/api/shipping/order/{orderId}`

```
Response:
{
  "deliveryId": "DEL-5011",
  "orderId": "ORD-1001",
```

```
  "status": "SHIPPED",
  "courier": "DHL",
  "trackingNumber": "DHL-TRK-202501",
  "estimatedDeliveryDate": "2025-07-03",
  "deliveredAt": null
}
```

📌 3. Get Delivery Info by Delivery ID

GET `/api/shipping/{deliveryId}`

(Same structure as above)

📌 4. Update Delivery Status (Webhook or Internal)

PATCH `/api/shipping/update-status`

```
Request:
{
  "deliveryId": "DEL-5011",
  "status": "DELIVERED", // or SHIPPED, IN_TRANSIT, DELAYED
  "deliveredAt": "2025-07-03T14:30:00Z",
  "trackingInfo": {
    "location": "New York Distribution Center",
    "timestamp": "2025-07-01T10:00:00Z"
  }
}
```

```
Response:
{
  "message": "Delivery status updated"
}
```

📌 5. Track Delivery (Customer-Facing)

GET `/api/shipping/track/{trackingNumber}`

```
Response:
{
  "trackingNumber": "DHL-TRK-202501",
  "status": "IN_TRANSIT",
  "lastLocation": "Philadelphia Hub",
  "history": [
    {
      "status": "SHIPPED",
      "location": "NY Warehouse",
      "timestamp": "2025-06-30T09:00:00Z"
    },
    {
      "status": "IN_TRANSIT",
      "location": "Philadelphia Hub",
      "timestamp": "2025-07-01T12:00:00Z"
    }
  ]
}
```

📌 6. Cancel Delivery (Optional)

POST `/api/shipping/cancel`

```
Request:
{
  "orderId": "ORD-1001",
  "reason": "Customer cancelled order"
}
```

```
Response:
{
  "message": "Delivery canceled"
}
```

📌 7. Delivery Partner Assignment (Optional)

POST `/api/shipping/assign-partner`

```
Request:
{
  "deliveryId": "DEL-5011",
  "partner": "Delhivery",
  "trackingNumber": "DLV123456"
}
```

```
Response:
{
  "message": "Partner assigned"
}
```

📌 8. Delivery Metrics / Analytics (Admin Use)

GET `/api/shipping/metrics`

```
Response:
{
  "totalDeliveries": 1020,
  "delivered": 950,
  "inTransit": 60,
  "failed": 10,
  "averageDeliveryTimeInDays": 3.2
}
```

📌 9. Health Check & Info

GET `/api/shipping/actuator/health`

GET `/api/shipping/actuator/info`

🧠 Microservice Integrations

Service	Purpose	Method
Order Service	Initiate shipping post-payment	REST / Kafka
User Service	Get delivery address or phone	REST or embedded

Notification	Send delivery updates	Kafka / RabbitMQ
Inventory Service	Trigger warehouse fulfillment	Event / REST

✅ Optional Events via Kafka/RabbitMQ

- `delivery.created`
- `delivery.shipped`
- `delivery.delivered`
- `delivery.failed`

Would you like the **Spring Boot template (entity, controller, service)** for this Shipping Service next?

Or shall we continue with **Notification Service APIs**?

Let me know your next step!

Here's a comprehensive list of **Notification Service APIs** for your **microservices-based eCommerce architecture**.

✅ Notification Service – API Design

📍 **Base URL:** `/api/notifications`

This service handles sending emails, SMS, push notifications, and in-app messages. It's event-driven and often uses **Kafka or RabbitMQ** for async processing.

📌 1. Send Notification (Generic)

POST `/api/notifications/send`

```
Request:
{
  "to": "john@example.com",
  "type": "EMAIL",        // EMAIL, SMS, PUSH
  "subject": "Order Shipped",
  "message": "Your order #ORD123 has been shipped.",
  "meta": {
```

```
    "orderId": "ORD123",
    "trackingId": "TRK456"
  }
}
```

```
Response:
{
  "status": "SENT",
  "notificationId": "NOTIF-1001"
}
```

📌 2. Trigger Notification by Event

POST `/api/notifications/event`

Used when triggered by events like `order.placed`, `delivery.shipped`, etc.

```
Request:
{
  "event": "order.placed",
  "userId": 101,
  "data": {
    "orderId": "ORD123",
    "amount": 500.00
  }
}
```

```
Response:
{
  "status": "QUEUED"
}
```

> 🔁 Internally maps event → message template → delivery channel

---

📌 3. Get Notifications by User

GET `/api/notifications/user/{userId}`

```
Response:
[
  {
    "notificationId": "NOTIF-1001",
    "type": "EMAIL",
    "message": "Your order #ORD123 has been shipped.",
    "status": "SENT",
    "timestamp": "2025-06-29T14:35:00Z"
  },
  ...
]
```

📌 4. Get Notification by ID

GET `/api/notifications/{notificationId}`

```
Response:
{
  "notificationId": "NOTIF-1001",
  "type": "EMAIL",
  "to": "john@example.com",
  "subject": "Order Shipped",
  "message": "Your order has been shipped",
  "status": "SENT",
  "timestamp": "2025-06-29T14:35:00Z"
}
```

📌 5. Resend Notification

POST `/api/notifications/resend/{notificationId}`

```
Response:
{
  "status": "RESENT"
}
```

📌 6. Add Notification Template (Admin)

POST `/api/notifications/templates`

```
Request:
{
  "event": "order.placed",
  "type": "EMAIL",
  "subject": "Order #{orderId} Placed",
  "template": "Hello {userName}, your order #{orderId} has been placed."
}
```

```
Response:
{
  "message": "Template saved successfully"
}
```

📌 7. List Templates (Admin)

GET `/api/notifications/templates`

```
Response:
[
  {
    "event": "order.placed",
    "type": "EMAIL",
    "subject": "Order #{orderId} Placed"
  },
  ...
]
```

📌 8. Delete Template (Admin)

DELETE `/api/notifications/templates/{event}/{type}`

```
Response:
{
```

```
  "message": "Template deleted"
}
```

📌 9. Health Check

GET `/api/notifications/actuator/health`

GET `/api/notifications/actuator/info`

🔗 Event-Based Triggers

Triggered by Kafka/RabbitMQ messages like:

Event	Triggered By	Notification Sent
`order.placed`	Order Service	Email/SMS
`payment.success`	Payment Service	Email
`delivery.shipped`	Shipping Service	Email/SMS
`delivery.delivered`	Shipping Service	Push notification
`user.signup`	User Service	Welcome email
`password.reset`	Auth Service	Email/SMS

🧠 Microservice Integrations

Service	Purpose	Integration
Order Service	Notify when order is placed	Kafka/Event
Shipping Service	Notify shipment status	Kafka/Event
User Service	Welcome or verification messages	REST/Event
Auth Service	Password reset or OTP messages	REST/Event

Would you like a **Spring Boot implementation** for this Notification Service (Kafka listener, controller, templates, etc.)?

Or continue to **API Gateway or Auth Service** next?

Let me know how you'd like to proceed!

In a microservices architecture, the **API Gateway Service** is the single entry point for all client requests. It routes requests to appropriate services, handles authentication, rate limiting, load balancing, and may provide logging and response transformation.

You're using **Spring Cloud Gateway**, so here's a full guide including API structure, use cases, and configurations for the **API Gateway Service**.

✅ API Gateway Service – Spring Cloud Gateway

📍 **Base URL:** Client-facing → `https://api.myecommerce.com/**`

📍 Internally proxies to microservices like `/product/**`, `/order/**`, etc.

🔧 Main Responsibilities

Feature	Handled by Gateway?	Notes
Routing requests	✅	Based on path or host
Authentication & JWT parsing	✅	With filter
Rate limiting	✅	Redis-based or in-memory
Circuit breaking	✅	Resilience4j/Hystrix
Load balancing	✅	Eureka + Ribbon
CORS	✅	Global config
Centralized logging/tracing	✅	Sleuth / Zipkin
Request/response transformation	✅	Optional pre/post filters

🚀 Sample Routes Configuration (`application.yml`)

```
spring:
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
          lower-case-service-id: true
      routes:
        - id: product-service
```

```
      uri: lb://PRODUCT-SERVICE
      predicates:
        - Path=/product/**
      filters:
        - AuthenticationFilter

    - id: order-service
      uri: lb://ORDER-SERVICE
      predicates:
        - Path=/order/**
      filters:
        - AuthenticationFilter

    - id: auth-service
      uri: lb://AUTH-SERVICE
      predicates:
        - Path=/auth/**

    - id: cart-service
      uri: lb://CART-SERVICE
      predicates:
        - Path=/cart/**
      filters:
        - AuthenticationFilter
```

> lb:// indicates load-balanced discovery via Eureka

🛡️ Example: Custom Authentication Filter

JWT token is validated before forwarding request:

```
@Component
public class AuthenticationFilter extends AbstractGatewayFilterFactory<AuthenticationFilter.Config> {

    public Mono<Void> apply(ServerWebExchange exchange, GatewayFilterChain chain) {
```

```
        // Check Authorization Header
        if (!exchange.getRequest().getHeaders().containsKey(HttpHeaders.AU
THORIZATION)) {
            throw new RuntimeException("Missing Authorization Header");
        }

        String token = exchange.getRequest().getHeaders().get(HttpHeaders.
AUTHORIZATION).get(0).substring(7);

        // Validate JWT (decode/verify with public key or secret)
        Claims claims = jwtUtil.validateToken(token);

        // Add user info to request
        exchange.getRequest().mutate()
            .header("X-User-Id", claims.getSubject())
            .build();

        return chain.filter(exchange);
    }

    public static class Config { }
}
```

📄 Exposed Endpoints (for DevOps/Admin)

These are **not** routed to downstream services, but for monitoring the gateway itself.

Method	Endpoint	Description
GET	`/actuator/gateway/routes`	Lists all routes
GET	`/actuator/health`	Health status of gateway
GET	`/actuator/metrics`	Performance metrics
GET	`/actuator/loggers`	Logging configuration
POST	`/actuator/gateway/refresh`	Refresh routes dynamically

🧠 Optional Enhancements

Feature	How
Swagger Aggregation	Combine all microservices' Swagger docs via SpringDoc Gateway Aggregation
Rate Limiting	Redis-based using `RequestRateLimiter`
Tracing	Spring Cloud Sleuth + Zipkin
Circuit Breaker	Add `- CircuitBreaker` filter with fallback URI

💡 API Gateway Has No "Business APIs"

Unlike other services, the **API Gateway doesn't have REST APIs like `/create`, `/get`**, etc. Instead, it:

- Exposes microservices' APIs under a unified domain

  e.g. `https://api.myecommerce.com/order/create`
- Adds filters (auth, logging, limits)
- Forwards to internal services via Eureka or static URLs

🧪 Example Request Flow via Gateway

Request:

```
GET https://api.myecommerce.com/order/ORD123
Authorization: Bearer eyJhbGciOi...
```

Gateway:

1. Auth filter validates JWT
2. Injects `X-User-Id` into header
3. Forwards to `ORDER-SERVICE` via Eureka

Final Internal Request:

```
GET http://ORDER-SERVICE/order/ORD123
X-User-Id: 102
```

Would you like help setting up a **Spring Cloud Gateway project**, or move to the next service like **Authentication (JWT + OAuth2)**?

Let me know how you'd like to proceed!

Here's a **complete guide** for designing the **Authentication & Authorization Service** for your **Spring Boot Microservices architecture** using **OAuth2 + JWT**.

✅ Authentication & Authorization Service

This service handles **login**, **registration**, **JWT token generation**, **validation**, **role-based access**, and optionally **refresh tokens**.

📍 **Base URL:** `/api/auth`

📌 Key Components

Component	Purpose
`JWT`	Token-based stateless auth
`OAuth2`	Support for Google, Facebook, etc. (optional)
`Spring Security`	Auth, role/permission enforcement
`Password Encoder`	BCrypt encoding
`User Roles`	`ROLE_USER`, `ROLE_ADMIN`, etc.
`Token Filter`	Validates token for incoming requests

📄 Authentication API Endpoints

1. 🔐 Register a New User

POST `/api/auth/register`

```
Request:
{
  "username": "john_doe",
  "email": "john@example.com",
  "password": "secret123",
```

```
  "roles": ["USER"]  // Optional, default is USER
}
```

```
Response:
{
  "message": "User registered successfully"
}
```

---

2. 🔑 Login (Generate JWT Token)

POST `/api/auth/login`

```
Request:
{
  "username": "john_doe",
  "password": "secret123"
}
```

```
Response:
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "refreshToken": "d9fa8d2d-eabc-4d8f-b2b3-99a...",
  "expiresIn": 3600
}
```

---

3. 🔄 Refresh Token

POST `/api/auth/refresh-token`

```
Request:
{
  "refreshToken": "d9fa8d2d-eabc-4d8f-b2b3-99a..."
}
```

```
Response:
{
```

```
  "token": "new-jwt-token",
  "expiresIn": 3600
}
```

---

4. 👱‍♀️ Get Current User Info

GET `/api/auth/me`

Header: `Authorization: Bearer <token>`

```
Response:
{
  "id": 1,
  "username": "john_doe",
  "email": "john@example.com",
  "roles": ["ROLE_USER"]
}
```

---

5. 🔒 Validate Token (Internal)

POST `/api/auth/validate`

```
Request:
{
  "token": "eyJhbGciOiJIUzI1NiIs..."
}
```

```
Response:
{
  "valid": true,
  "userId": 101,
  "roles": ["ROLE_USER"]
}
```

> Used by other microservices for token validation.

---

6. 🛑 Logout (Invalidate Refresh Token)

POST `/api/auth/logout`

```
Request:
{
  "refreshToken": "..."
}
```

```
Response:
{
  "message": "Logged out successfully"
}
```

🛠️ JWT Token Claims Example

```
{
  "sub": "101",
  "username": "john_doe",
  "roles": ["ROLE_USER"],
  "iat": 1719667311,
  "exp": 1719670911
}
```

🔐 Role-Based Access Control

Role	Access To
`ROLE_USER`	Orders, cart, product viewing
`ROLE_ADMIN`	Product management, user management

You can secure endpoints like this in other services:

```
@PreAuthorize("hasRole('ADMIN')")
@GetMapping("/products/admin")
public List<Product> adminOnlyEndpoint() { ... }
```

Or via `SecurityConfig`:

```
.antMatchers("/api/products/**").hasAnyRole("USER", "ADMIN")
.antMatchers("/api/admin/**").hasRole("ADMIN")
```

🧠 Integration With Other Microservices

Other services (Product, Order, etc.) do **not** handle login. They:

- Use Spring Security
- Intercept JWT from `Authorization: Bearer <token>`
- Decode it using shared secret/public key
- Authorize based on roles or user ID

🟢 With Spring Cloud Gateway, this is done via **pre-filter**.

📦 Important Files in Spring Boot

File / Class	Purpose
`SecurityConfig.java`	Configure HTTP security, filters
`JwtUtils.java`	Generate, parse, validate JWTs
`AuthController.java`	REST APIs for login, register
`UserDetailsServiceImpl.java`	Load user by username
`JWTFilter.java`	Intercepts every request
`RefreshTokenService.java`	Handle refresh tokens (optional)

🔐 Security Best Practices

- Use **BCrypt** for password hashing
- Use **short-lived access tokens** (e.g. 15–30 mins)
- Use **refresh tokens** stored securely (e.g. HttpOnly cookie)
- Add **rate limiting** for login/register APIs
- Protect Gateway endpoints with security filters
- Use **HTTPS only** for secure transport

💡 Optional Enhancements

- ✅ 2FA / OTP login
- ✅ Social login (Google, Facebook)
- ✅ Blacklist tokens on logout (via DB or cache)
- ✅ Multi-tenant user handling

Would you like a **Spring Boot starter project template** for this Auth service?

Or continue with the **next service**, like **Service Registry (Eureka)**?

Let me know your next step!

Great! Let's go over the **Service Registry** in a microservices architecture, using **Spring Cloud Netflix Eureka** (most common choice) or **Consul** as alternatives.

✅ Service Registry Service

The **Service Registry** is responsible for:

- Keeping track of all running microservices and their locations.
- Allowing services to **discover each other dynamically** (client-side or server-side discovery).
- Supporting **load balancing** and **resilience** via integration with Ribbon/Feign.

📦 **Tools Used**:

- **Spring Cloud Netflix Eureka** (preferred in Spring ecosystem)
- Alternative: **HashiCorp Consul**

🌐 Eureka Architecture Overview

```
[Microservice A] ↔ Eureka Server ↔ [Microservice B]
      ↑                  ↓
   Registers          Discovers
```

🔧 Eureka Server Setup

Dependencies (Maven):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Application class:

```
@EnableEurekaServer
@SpringBootApplication
public class EurekaServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }
}
```

application.yml:

```
server:
  port: 8761

eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
  server:
    wait-time-in-ms-when-sync-empty: 0
```

✅ APIs Exposed by Eureka Server

These are automatically exposed when Eureka is running.

Method	Endpoint	Description
GET	`/eureka/apps`	List all registered services
GET	`/eureka/apps/{SERVICE-ID}`	Details of a specific service

GET	`/eureka/apps/{SERVICE-ID}/{ID}`	Specific instance of a service
DELETE	`/eureka/apps/{SERVICE-ID}/{ID}`	Unregister a service
GET	`/eureka/status`	Health/status of Eureka server

> 💡 These are for admin use or programmatic interaction.

🧩 Client Microservice Setup (e.g. Product Service)

Dependencies:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

Main class:

```
@EnableDiscoveryClient
@SpringBootApplication
public class ProductServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProductServiceApplication.class, args);
    }
}
```

application.yml:

```
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true

spring:
```

```
  application:
    name: product-service
```

🎯 Important Configs

Config Option	Description
`register-with-eureka`	Whether the service registers itself
`fetch-registry`	Whether it fetches the registry info
`prefer-ip-address`	Useful in Docker/Kubernetes
`lease-renewal-interval-in-seconds`	How often heartbeat is sent
`lease-expiration-duration-in-seconds`	Timeout for unresponsive services

🔄 Discovery in Other Services

Using **Spring Cloud LoadBalancer** or **OpenFeign**:

```
@FeignClient(name = "product-service")
public interface ProductClient {
    @GetMapping("/products/{id}")
    Product getProduct(@PathVariable Long id);
}
```

With this, no need to hardcode hostnames—just use service name from Eureka.

📊 Eureka Dashboard

Visit:

```
http://localhost:8761
```

You'll see:

- Registered services
- Instance metadata (IP, port, status)
- Health
- Uptime

📦 Eureka Health Check (Optional)

You can enable automatic removal of unhealthy services:

```
eureka:
  instance:
    lease-renewal-interval-in-seconds: 10
    lease-expiration-duration-in-seconds: 30

management:
  endpoints:
    web:
      exposure:
        include: "*"
```

Each client will need Spring Boot Actuator to expose `/actuator/health`.

❗ Common Issues

Issue	Fix
Services not appearing	Ensure registration is enabled & Eureka URL is reachable
IP address shows as hostname	Set `prefer-ip-address: true`
Services not discovering others	Use Feign or RestTemplate with `@LoadBalanced`

🚀 Optional Advanced

- **High Availability**: Deploy multiple Eureka servers and use peer replication
- **Eureka Security**: Protect `/eureka/*` endpoints using Spring Security
- **Service Filtering**: Control which apps are visible to whom
- **Consul Alternative**: More advanced, supports key-value config, but heavier setup

Would you like help generating a **ready-to-run Eureka Server** project, or shall we move to the next:

👉 **Spring Cloud Config (Configuration Service)**?

Let me know!

Here's a comprehensive guide for your **Configuration Service** using **Spring Cloud Config**, an essential part of centralized configuration in a Spring Boot microservices architecture.

✅ Configuration Service Overview (Spring Cloud Config)

The **Configuration Service** provides:

- **Centralized management** of application properties
- Dynamic **reload of config without redeploying**
- Support for **profiles (dev, test, prod)**, environments, version control (Git)
- Secure storage of sensitive data via **Vault**

📦 Tools Involved

Tool	Purpose
Spring Cloud Config Server	Serves configuration properties
Git Repository	Stores the actual config files
Spring Boot Actuator + Bus	Enables live refresh via messaging (RabbitMQ/Kafka)

🔧 1. Config Server Setup

Dependencies (Maven):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Main Class:

```
@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigServerApplication.class, args);
    }
}
```

`application.yml` **for Config Server:**

```
server:
  port: 8888

spring:
  cloud:
    config:
      server:
        git:
          uri: https://github.com/your-org/your-config-repo
          clone-on-start: true
          search-paths:
            - config-files
  application:
    name: config-server
```

📂 Git Repo Structure Example

```
your-config-repo/
│
├── product-service-dev.yml
├── product-service-prod.yml
├── order-service-dev.yml
├── order-service-prod.yml
└── application.yml (shared config)
```

📄 REST APIs Exposed by Config Server

Method	Endpoint	Description
GET	`/{application}/{profile}`	Get config for app & profile
GET	`/{application}/{profile}/{label}`	Get config from Git branch/tag
GET	`/health`	Health check
POST	`/actuator/refresh`	Trigger manual refresh of config (when using Spring Cloud Bus)

Example:

```
GET http://localhost:8888/product-service/dev
```

Returns:

```
{
  "name": "product-service",
  "profiles": ["dev"],
  "propertySources": [
    {
      "name": "https://github.com/.../product-service-dev.yml",
      "source": {
        "server.port": 8081,
        "spring.datasource.url": "jdbc:mysql://..."
      }
    }
  ]
}
```

🧩 Microservice Client Setup (e.g., Product Service)

Dependencies:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

`bootstrap.yml` or `bootstrap.properties`:

```
spring:
  application:
    name: product-service
  cloud:
    config:
      uri: http://localhost:8888
      profile: dev
```

> Replace dev with prod, test, etc., depending on environment.

🔄 Dynamic Config Refresh (Optional)

If you want to **dynamically refresh config** at runtime using Spring Cloud Bus:

Add dependencies:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-bus-amqp</artifactId> <!-- For RabbitM
Q -->
</dependency>
```

Enable refresh scope:

```
@RefreshScope
@RestController
public class SomeConfigAwareController {
    @Value("${some.dynamic.config}")
    private String configValue;
}
```

Refresh config:

```
POST http://localhost:8081/actuator/refresh
```

Or broadcast with:

```
POST http://localhost:8888/actuator/bus-refresh
```

🔐 Securing Config Server (Recommended)

```yaml
spring:
  security:
    user:
      name: configuser
      password: secret123
```

Then call it like:

```
GET http://configuser:secret123@localhost:8888/product-service/dev
```

📌 Best Practices

Practice	Why?
Use Git or Vault	Secure, version-controlled, audit logs
Separate config per profile	Dev/test/prod configs differ
Avoid sensitive data in plain text	Use Vault or encrypted placeholders
Use `@RefreshScope` wisely	Only for beans that need dynamic refresh
Centralize common config	In `application.yml` (shared for all)

🧪 Health Check & Testing

- Config Server: `GET http://localhost:8888/actuator/health`
- Client service: `GET /actuator/env` or `GET /actuator/configprops`

🚀 Advanced Enhancements

- Use **HashiCorp Vault** as backend instead of Git
- Enable **encryption/decryption endpoints** for sensitive values

- Use **Consul** for config + service registry (alternative to Git+Eureka)

🔗 Real Example Git Config

```
# product-service-dev.yml

server:
  port: 8081

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/product_db
    username: product_user
    password: secure_pass
```

Would you like help **generating a Spring Boot Config Server project** or **connecting it with GitHub + RabbitMQ**?

Or should we move on to the next service:

👉 **Message Broker (RabbitMQ/Kafka)**?

Let me know!

Here's a comprehensive breakdown for **Kafka Service** in your microservices architecture:

✅ 13. Message Broker – Kafka in Microservices

Apache **Kafka** is a distributed streaming platform used in microservices for:

- **Asynchronous communication**
- **Decoupling services**
- **Event-driven architecture**
- **Guaranteed message delivery and replayability**

💡 Why Kafka in Microservices?

Benefit	Description
Scalability	Kafka handles high-throughput event streams
Loose Coupling	Services publish/subscribe without knowing about each other
Resilience	Services can continue independently even if others are down
Event-Driven Design	Great for modern reactive/event-sourcing systems

---

🧱 Architecture

```
[Product Service] ——> [Kafka Topic: product-events] ——> [Inventory Servi
ce]
                        ↑
                    Kafka Broker
                        ↓
              [Notification Service, Order Service]
```

🔧 Kafka Service Setup in Spring Boot

Dependencies (Producer/Consumer):

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
```

🔁 Common Kafka Concepts

Concept	Description
Topic	Logical channel to send/receive messages
Producer	Publishes messages to a topic
Consumer	Listens to messages from a topic
Broker	Kafka server instance
Partition	Unit of parallelism; each topic has 1+ partitions
Consumer Group	Set of consumers sharing the load

📄 Common APIs (Kafka-related service APIs)

While Kafka itself is not a REST-based service, your microservices can expose APIs **to publish or monitor Kafka**. Examples:

✅ 1. Publish Event (Producer API)

Endpoint (from Order Service):

```
POST /api/orders/publish
```

Request Body:

```
{
  "orderId": "123",
  "status": "PLACED",
  "userId": "u001",
  "total": 199.99
}
```

Spring Producer Code:

```
@Autowired
private KafkaTemplate<String, OrderEvent> kafkaTemplate;

@PostMapping("/publish")
public ResponseEntity<String> sendOrder(@RequestBody OrderEvent orde
rEvent) {
    kafkaTemplate.send("order-events", orderEvent.getOrderId(), orderEven
t);
    return ResponseEntity.ok("Order event published");
}
```

✅ 2. Listen to Event (Consumer Service – e.g., Inventory)

Kafka Consumer Listener:

```
@KafkaListener(topics = "order-events", groupId = "inventory-group")
public void handleOrderEvent(OrderEvent event) {
```

```
    // Reduce inventory, log event, etc.
}
```

✅ 3. Monitor Messages (Optional Admin API)

GET /api/kafka/messages

Returns last `n` messages from a topic using an internal storage/cache.

🗃️ Typical Kafka Topics in Your Architecture

Topic Name	Description	Publisher	Consumers
`order-events`	Order placed/cancelled	Order Service	Inventory, Payment, Notify
`payment-events`	Payment success/failure	Payment Service	Order, Notification
`inventory-events`	Inventory reserved/released	Inventory Service	Order, Shipping
`shipping-events`	Shipment dispatched/delivered	Shipping Service	Order, Notification
`cart-events`	Cart updates	Cart Service	User Service (analytics)

⚙️ Sample Kafka Config in `application.yml`

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: order-group
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer
    producer:
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
```

```
    value-serializer: org.springframework.kafka.support.serializer.JsonSeri
alizer
```

🔒 Security Considerations

- Enable **SSL/TLS** for encryption
- Use **SASL authentication** with Kafka brokers
- Validate payloads to avoid injection or malformed events

🔁 Message Format Example

OrderEvent.java

```
@Data
public class OrderEvent {
    private String orderId;
    private String userId;
    private String status; // CREATED, CANCELLED, etc.
    private Double amount;
}
```

Kafka will serialize/deserialize this to/from JSON (if configured with `JsonSerializer`).

🔥 Advanced Kafka Patterns

Pattern	Use
Dead Letter Topics	Handle failed events
Retry Topics	For automatic reprocessing
Event Sourcing	Persist entire state change history
Compacted Topics	Keep only the latest value for a key

🔍 Tools for Kafka Monitoring

Tool	Purpose

Kafka UI (Confluent Control Center, AKHQ, Kafdrop)	Browse topics, partitions, messages
Prometheus + Grafana	Kafka performance metrics
Zipkin/Jaeger	Distributed tracing if integrated with Kafka

✅ Best Practices

Practice	Reason
Use separate topics per entity	Prevents event mixing
Use compact/retention policies wisely	Control data size
Don't expose Kafka directly	Wrap with APIs or message gateway
Use schemas (Avro/JSON) + validation	Prevent contract mismatch

---

Would you like a **Kafka producer/consumer Spring Boot template**, or move on to:

👉 **Swagger/OpenAPI Documentation Service**?

Let me know!

To build a robust **API Documentation Service** for your microservices using **Swagger / OpenAPI**, here's a full guide including **important concepts**, **integration strategy**, and **example API setups** for each microservice.

✅ 14. API Documentation Service – Swagger/OpenAPI

🔍 What Is It?

Swagger (now OpenAPI Specification) is a framework for **describing**, **producing**, **consuming**, and **visualizing** RESTful APIs.

With Swagger you get:

- Auto-generated API documentation
- Interactive testing UI
- Consistent and clear contract between frontend & backend

- Language-agnostic REST interface description

⚙️ Tools in Swagger Ecosystem

Tool	Purpose
`springdoc-openapi`	Java/Spring Boot Swagger integration
`Swagger UI`	Web-based UI for exploring/testing APIs
`OpenAPI JSON/YAML`	Machine-readable contract definition
`Swagger Editor`	Edit and visualize OpenAPI specs

🧩 Integration Per Microservice

Each microservice (Product, Order, Cart, etc.) should expose its **own Swagger documentation** endpoint, usually at:

```
GET /v3/api-docs          → OpenAPI JSON spec
GET /swagger-ui.html      → Interactive Swagger UI
```

📦 Dependency for Spring Boot (OpenAPI 3)

Add this in every microservice:

```xml
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.5.0</version>
</dependency>
```

🧾 Sample Configuration (Optional)

```yaml
springdoc:
  api-docs:
    path: /v3/api-docs
  swagger-ui:
    path: /swagger-ui.html
```

🧪 Sample Endpoint Documentation

Product Service

```
@Operation(summary = "Get product by ID", description = "Returns the det
ails of a product.")
@ApiResponses(value = {
  @ApiResponse(responseCode = "200", description = "Found the produc
t"),
  @ApiResponse(responseCode = "404", description = "Product not foun
d")
})
@GetMapping("/products/{id}")
public ResponseEntity<Product> getProductById(@PathVariable Long id) {
  // ...
}
```

📋 Centralized API Gateway Documentation

You **can aggregate all microservice Swagger docs** behind your API Gateway using **SpringDoc + Swagger UI with custom configuration**, or use tools like:

- **SwaggerHub** (for hosted solutions)
- **Redocly**
- Custom UI aggregating `/v3/api-docs` from all services

🌐 OpenAPI JSON Aggregation Example

Assume:

- Product Service: `http://product-service/v3/api-docs`
- Order Service: `http://order-service/v3/api-docs`

Create a **custom Swagger UI frontend** or use a **gateway route to expose multiple docs**.

🧠 Important Concepts to Document

Section	Why Important
Title, Version	Clear version tracking
Base Path	Helps UI routing
Tags	Group endpoints (e.g., Orders, Payments)
Schema definitions	Explain request/response formats
Security Schemes	JWT/OAuth2 integration for secured APIs

🔐 Example: JWT Security in Swagger

```
@SecurityScheme(
  name = "bearerAuth",
  type = SecuritySchemeType.HTTP,
  scheme = "bearer",
  bearerFormat = "JWT"
)
```

```
@Operation(
  summary = "Place an order",
  security = @SecurityRequirement(name = "bearerAuth")
)
```

📄 API Documentation per Service (Example Overview)

Service	Swagger URL	Description
Product Service	`/product-service/swagger-ui.html`	CRUD for products
Order Service	`/order-service/swagger-ui.html`	Place/view orders
Cart Service	`/cart-service/swagger-ui.html`	Cart item operations
Payment Service	`/payment-service/swagger-ui.html`	Payment processing APIs
Inventory Service	`/inventory-service/swagger-ui.html`	Stock/reservation APIs
Notification Service	`/notification-service/swagger-ui.html`	Email/SMS notifications
Auth Service	`/auth-service/swagger-ui.html`	Login/Register/OAuth2 flows
API Gateway	`/api-gateway/swagger-ui.html`	(Optionally aggregates others)

✅ Best Practices

Practice	Benefit
Keep it versioned	Avoid breaking changes for consumers
Describe all params and responses	Helps consumers understand contract
Use enums where applicable	Better validation and clarity
Secure Swagger UI in production	Prevent API misuse or leakage
Generate client code from OpenAPI	Frontend teams can autogenerate clients

🚀 Optional Enhancements

- Integrate with **SwaggerHub** for team collaboration
- Use **Redoc** for stylish and user-friendly API docs
- Auto-generate **SDKs** from OpenAPI specs using tools like:
  - OpenAPI Generator
  - Swagger Codegen

Would you like me to generate a **starter Swagger config** or sample `ProductController` with full OpenAPI annotations?

Let me know how you want to proceed!