HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA ĐÀO TẠO SAU ĐẠI HỌC

BÁO CÁO

CÁC HỆ THỐNG PHÂN TÁN

Đề Tài: Phân tích và thiết kế hệ thống thương mại điện tử dựa trên kiến trúc Microservices

Giảng viên hướng dẫn : TS. Kim Ngọc Bách
Lớp : M25CQIS02
Nhóm 12
Trần Ngọc Minh : B25CHHT040
Lê Trung Anh : B25CHHT001
Bùi Thảo Ly : B25CHHT038

Hà Nội – 2025

MỤC LỤC

LỜI MỞ ĐẦU

Trong bối cảnh công nghệ thông tin ngày càng phát triển, các hệ thống phần mềm hiện đại đòi hỏi khả năng mở rộng, tính linh hoạt và độ tin cậy cao để đáp ứng nhu cầu ngày càng lớn của người dùng. Đặc biệt, trong lĩnh vực thương mại điện tử, hệ thống không chỉ cần xử lý số lượng lớn giao dịch mà còn phải đảm bảo tính ổn định, dễ bảo trì và khả năng phát triển lâu dài.

Kiến trúc Microservices đã và đang trở thành một xu hướng phổ biến trong việc xây dựng các hệ thống phần mềm quy mô lớn. Thay vì xây dựng một ứng dụng nguyên khối (Monolithic), kiến trúc Microservices chia hệ thống thành nhiều dịch vụ nhỏ, độc lập, mỗi dịch vụ đảm nhiệm một chức năng riêng biệt và có thể được phát triển, triển khai cũng như mở rộng một cách linh hoạt.

Xuất phát từ những yêu cầu thực tế đó, nhóm chúng em lựa chọn đề tài “Phân tích và thiết kế hệ thống thương mại điện tử dựa trên kiến trúc Microservices”. Mục tiêu của đề tài là nghiên cứu cách phân tích yêu cầu, thiết kế kiến trúc hệ thống và xây dựng các dịch vụ cơ bản trong một hệ thống thương mại điện tử theo mô hình Microservices, từ đó giúp sinh viên hiểu rõ hơn về cách áp dụng kiến trúc này vào các bài toán thực tế.

Thông qua đề tài, nhóm mong muốn vận dụng các kiến thức đã học về phân tích thiết kế hệ thống, kiến trúc phần mềm và công nghệ hiện đại, đồng thời rèn luyện kỹ năng làm việc nhóm và tư duy thiết kế hệ thống.

I. Tổng quan

1. Bối cảnh chuyển đổi từ Monolithic sang Microservices

Trong giai đoạn đầu, kiến trúc **Monolithic** được sử dụng phổ biến nhờ tính đơn giản, toàn bộ chức năng và cơ sở dữ liệu được triển khai trong một khối duy nhất. Tuy nhiên, đối với các hệ thống thương mại điện tử quy mô lớn, mô hình này bộc lộ nhiều hạn chế như khó mở rộng, phụ thuộc chặt chẽ giữa các module và khó bảo trì.

Kiến trúc **Microservices** được áp dụng trong dự án nhằm khắc phục các nhược điểm trên. Hệ thống được chia thành các dịch vụ độc lập như *user-service, product-service, order-service*, mỗi dịch vụ có vòng đời phát triển và triển khai riêng, giúp tăng tính linh hoạt và ổn định.

2. Đặc điểm kỹ thuật nổi bật của hệ thống

Dựa trên cấu trúc dự án, kiến trúc Microservices mang lại các lợi ích chính sau:

- **Tính độc lập và chịu lỗi:** Mỗi service hoạt động độc lập, lỗi của một dịch vụ không làm gián đoạn toàn hệ thống.
- **Khả năng mở rộng linh hoạt:** Có thể mở rộng riêng từng service khi xảy ra tải cao (ví dụ inventory-service trong đợt khuyến mãi).
- **Đa dạng công nghệ:** Mỗi dịch vụ có thể sử dụng công nghệ và cơ sở dữ liệu phù hợp với nghiệp vụ của mình.

3. Kiến trúc Microservices

Microservices là phong cách kiến trúc trong đó một ứng dụng lớn được chia thành nhiều dịch vụ nhỏ, độc lập, giao tiếp với nhau thông qua các giao thức nhẹ như HTTP hoặc Message Queue. Mỗi dịch vụ đảm nhiệm một chức năng nghiệp vụ riêng và có thể phát triển, triển khai độc lập.

Kiến trúc này mang lại các đặc trưng nổi bật như: liên kết lỏng lẻo, triển khai độc lập, khả năng mở rộng cao, cô lập lỗi, quản lý dữ liệu phi tập trung và khả năng chịu lỗi tốt.

Các đặc trưng của kiến trúc Microservice:

- Dựa trên dịch vụ (Service-based)

Microservices được thiết kế như các dịch vụ riêng lẻ, mỗi dịch vụ chịu trách nhiệm cho một khả năng kinh doanh hoặc chức năng cụ thể trong ứng dụng. Mỗi dịch vụ có thể được phát triển, triển khai và mở rộng độc lập.

- Tính phi tập trung (Decentralized)

Microservices khuyến khích sự quản trị và ra quyết định phi tập trung. Mỗi dịch vụ có mã nguồn riêng, cơ sở dữ liệu (nếu cần), và đội ngũ chịu trách nhiệm phát triển và duy trì dịch vụ đó.

- Tính lỏng lẻo (Loosely Coupled)

Microservices giao tiếp với nhau thông qua các giao thức nhẹ như REST (Representational State Transfer) hoặc hàng đợi thông điệp. Các dịch vụ được tách rời nhau, cho phép phát triển, triển khai và mở rộng một cách độc lập.

- Phát triển độc lập (Independent Deployment)

Microservices có thể được triển khai độc lập với nhau. Điều này có nghĩa là thay đổi hoặc cập nhật một dịch vụ không yêu cầu phải triển khai lại toàn bộ ứng dụng, giảm thiểu thời gian ngừng hoạt động và sự gián đoạn.

“Product not project” đây là khái niệm của Amazon “bạn xây dựng, bạn vận hành”, nơi một nhóm phát triển chịu trách nhiệm toàn bộ cho phần mềm trong quá trình hoạt động.

- Khả năng mở rộng (Scalability)

Vì mỗi dịch vụ hoạt động độc lập, có thể mở rộng hoặc giảm quy mô từng dịch vụ theo nhu cầu mà không ảnh hưởng đến phần còn lại của ứng dụng. Điều này cho phép các đội ngũ phân bổ tài nguyên hiệu quả hơn và đảm bảo ứng dụng có thể xử lý lưu lượng truy cập hoặc nhu cầu tăng cao.

- Khả năng cô lập lỗi (Fault Isolation)

Nếu một dịch vụ gặp sự cố, nó không ảnh hưởng đến toàn bộ ứng dụng. Sự cố được giới hạn trong dịch vụ gặp lỗi, và các dịch vụ khác vẫn có thể hoạt động bình thường.

- Đa dạng công nghệ (Technology Diversity)

Các dịch vụ khác nhau có thể được viết bằng các ngôn ngữ lập trình khác nhau hoặc sử dụng các công nghệ khác nhau. Điều này giúp dễ dàng chọn công cụ phù hợp cho từng công việc, thay vì bị ràng buộc vào một nền tảng công nghệ cụ thể.

- Thiết kế để chịu lỗi và khả năng hồi phục (Resilience)

Microservices được thiết kế để đối phó với các lỗi và quản lý lỗi với các hành động phù hợp. Microservices cũng được thiết kế để phục hồi, nghĩa là chúng có thể tiếp tục hoạt động ngay cả khi một hoặc nhiều dịch vụ bị lỗi. Vì mỗi dịch vụ hoạt động độc lập, sự cố trong một dịch vụ không ảnh hưởng đến toàn bộ ứng dụng.

- Quản lý dữ liệu phi tập trung (Decentralized Data Management)

Microservices cũng phân cấp quyết định lưu trữ dữ liệu. Chúng ta có thể gọi cách tiếp cận này là Polyglot Persistence hoặc Polyglot Databases. Điều này có nghĩa là Microservices cho phép mỗi dịch vụ quản lý cơ sở dữ liệu của riêng mình, có thể là

các phiên bản khác nhau của cùng một công nghệ cơ sở dữ liệu hoặc các hệ thống cơ sở dữ liệu hoàn toàn khác nhau.

4. Docker và Công nghệ Container hóa (Containerization)

Trong bối cảnh hệ thống Microservices bao gồm hơn 15 dịch vụ riêng lẻ (order-service, payment-service, inventory-service,...), việc triển khai thủ công từng dịch vụ là bất khả thi. Docker đóng vai trò là "tiêu chuẩn hóa" đơn vị phần mềm, giúp đóng gói ứng dụng và tất cả các thành phần phụ thuộc vào một môi trường cô lập.

4.1. Nguyên lý vận hành của Docker và Containerization

Docker hoạt động dựa trên cơ chế chia sẻ nhân hệ điều hành (Host OS Kernel) nhưng vẫn đảm bảo sự cô lập hoàn toàn giữa các tiến trình.

- **Tính đóng gói (Encapsulation):** Mỗi Microservice (ví dụ: api-gateway) được định nghĩa trong một Dockerfile. File này chứa các chỉ dẫn để xây dựng **Docker Image** – một snapshot tĩnh chứa từ mã nguồn Java, JRE, các biến môi trường cho đến các thư viện hệ thống cần thiết.
- **Tính bất biến (Immutability):** Một khi Image đã được tạo ra, nó sẽ không thay đổi. Điều này loại bỏ hoàn toàn lỗi "it works on my machine" (chạy tốt trên máy tôi nhưng lỗi trên server) do sai lệch phiên bản thư viện hoặc môi trường.
- **Tiết kiệm tài nguyên:** Vì không cần khởi chạy một hệ điều hành khách (Guest OS) như máy ảo (VM), các Docker Container khởi động chỉ trong vài giây và tiêu tốn rất ít RAM/CPU, cho phép chạy hàng chục service trên cùng một máy chủ vật lý.

4.2. Docker Compose

Đóng vai trò điều phối toàn bộ hệ thống, quản lý sự phụ thuộc giữa các dịch vụ, thiết lập mạng nội bộ, cấu hình biến môi trường và lưu trữ dữ liệu bền vững.

Các chức năng cốt lõi của Docker Compose trong dự án:

- **Quản lý sự phụ thuộc (Dependency Management):** Thông qua chỉ thị depends_on, Docker Compose đảm bảo các dịch vụ hạ tầng như discovery-service (Eureka) hoặc Kafka phải khởi động thành công trước khi các business service như product-service bắt đầu chạy. Điều này ngăn ngừa lỗi kết nối khi hệ thống vừa khởi động.
- **Hệ thống mạng nội bộ (Internal Networking):** Docker Compose tự động khởi tạo một mạng ảo (Virtual Bridge Network). Trong mạng này, các container có thể giao tiếp với nhau bằng **Service Name** thay vì địa chỉ IP.
    - *Ví dụ:* order-service có thể gọi đến inventory-service thông qua URL http://inventory-service:8080 thay vì phải biết địa chỉ IP động của

container đó. Docker tích hợp một cơ chế DNS nội bộ để phân giải tên service này.

- **Quản lý biến môi trường tập trung:** Thay vì cấu hình cứng (hard-code) các tham số như thông tin kết nối Database hay địa chỉ Kafka Broker vào mã nguồn, Docker Compose cho phép định nghĩa chúng trong phần environment. Điều này giúp hệ thống linh hoạt khi chuyển đổi giữa môi trường Development, Staging và Production.
- **Quản lý dữ liệu bền vững (Volumes):** Vì Container có đặc tính "tạm thời" (dữ liệu biến mất khi container bị xóa), Docker Compose sử dụng cơ chế volumes để gắn kết (map) dữ liệu từ bên trong container (như dữ liệu MySQL của user-service) ra đĩa cứng của máy chủ. Điều này đảm bảo dữ liệu đơn hàng và người dùng luôn được bảo toàn ngay cả khi nâng cấp hoặc khởi động lại hệ thống.

4.3. Lợi ích thực tiễn đối với hệ thống E-commerce

1. **Mở rộng nhanh chóng (Scaling):** Với Docker, ta có thể dễ dàng tăng số lượng instance cho search-service chỉ bằng lệnh docker-compose up --scale search-service=3 để đáp ứng lượng truy cập lớn mà không cần cấu hình lại mạng.
2. **Khôi phục thảm họa (Disaster Recovery):** Nếu một container bị sập, Docker có thể được cấu hình để tự động khởi động lại (restart policy), giúp hệ thống tự phục hồi mà không cần can thiệp thủ công.
3. **Môi trường phát triển đồng nhất:** Thành viên mới gia nhập dự án chỉ cần cài đặt Docker và chạy một lệnh duy nhất để có toàn bộ môi trường hoạt động (gồm DB, Message Broker, các Service), giảm thời gian thiết lập từ vài ngày xuống vài phút.

5. Kafka và Mô hình Giao tiếp hướng sự kiện (Event-Driven)

Trong một hệ thống phân tán phức tạp như Thương mại điện tử, việc các dịch vụ gọi trực tiếp lẫn nhau (Point-to-point) sẽ tạo ra một mạng lưới kết nối chằng chịt và dễ đổ vỡ. Apache Kafka xuất hiện như một hệ thống trung tâm giúp tách rời các dịch vụ (Decoupling).

5.1 Cơ chế vận hành chi tiết: Publish/Subscribe & Data Streaming

Apache Kafka được sử dụng để hỗ trợ giao tiếp bất đồng bộ giữa các dịch vụ, giúp giảm sự phụ thuộc trực tiếp và tăng khả năng mở rộng hệ thống. Thông qua mô hình **Publish/Subscribe**, các dịch vụ có thể gửi và nhận sự kiện mà không cần biết trực tiếp nhau.

Kafka giúp hệ thống xử lý song song hiệu quả, đảm bảo khả năng chịu tải cao, phục hồi dữ liệu và nâng cao trải nghiệm người dùng trong các nghiệp vụ phức tạp như xử lý đơn hàng.

5.2 Vai trò chiến lược trong hệ thống phân tán

- **Xử lý bất đồng bộ (Asynchronous Processing) & Nâng cao trải nghiệm người dùng:**
  - Trong luồng đặt hàng, nếu thực hiện đồng bộ, người dùng phải đợi: *Lưu đơn -> Trừ kho -> Tính thuế -> Gửi Email -> Phản hồi*. Chỉ cần một bước chậm, người dùng sẽ thấy ứng dụng bị "treo".
  - Với Kafka, order-service chỉ làm nhiệm vụ chính là lưu đơn và bắn một Event vào Kafka, sau đó phản hồi ngay cho người dùng: "Đơn hàng của bạn đang được xử lý". Các tác vụ nặng như gửi Email hay tính toán vận chuyển sẽ được thực hiện "ngầm" bởi các dịch vụ khác ở phía sau.
- **Khả năng chịu tải và Mở rộng (Scalability qua Partitions):**
  - Kafka chia một Topic thành nhiều **Partitions** (Phân vùng). Các phân vùng này được phân tán trên nhiều máy chủ (Brokers) khác nhau.
  - Cơ chế này cho phép hệ thống thực hiện song song hóa: 10 instance của notification-service có thể đọc đồng thời từ 10 partitions khác nhau của cùng một Topic, giúp tốc độ xử lý tăng gấp 10 lần mà không bị thắt nút cổ chai.
- **Khả năng chịu lỗi (Fault Tolerance):**
  - Kafka tự động sao lưu (Replication) mỗi Partition sang nhiều Broker khác nhau. Nếu một máy chủ chứa Kafka bị sập, hệ thống vẫn hoạt động bình thường vì dữ liệu đã có bản sao ở máy chủ khác.

II. Phân tích yêu cầu hệ thống

1. Yêu cầu chức năng nghiệp vụ

a) Bộ phần người dùng chưa đăng ký tài khoản

- **Bảng yêu cầu chức năng**

STT	Tên yêu cầu	Loại nghiệp vụ	Ghi chú
1	Đăng ký tài khoản	Lưu trữ	Tạo tài khoản mới
2	Xem danh sách sản phẩm	Tra cứu	Lấy từ product service
3	Xem chi tiết sản phẩm	Tra cứu	Xem thông tin sản phẩm
4	Tìm kiếm sản phẩm	Tra cứu	Tìm theo tên

- ***Quy định:***

STT	Mô tả chi tiết	Ghi chú
1	Người dùng chưa đăng ký chỉ được phép xem thông tin	Không thao tác dữ liệu
2	Không yêu cầu xác thực khi truy cập	Public API

b) Người dùng đã đăng ký tài khoản

STT	Tên yêu cầu	Loại nghiệp vụ	Ghi chú
1	Đăng nhập hệ thống	Xác thực	Xác thực người dùng
2	Thêm sản phẩm vào giỏ	Xử lý	Chuẩn bị tạo đơn
3	Tạo đơn hàng	Lưu trữ	Gọi Order Service
4	Thanh toán đơn hàng	Xử lý	Gọi Payment Service
5	Xem trạng thái đơn hàng	Tra cứu	Theo dõi đơn

- ***Quy định:***

STT	Mô tả chi tiết	Ghi chú
1	Người dùng phải đăng nhập để tạo đơn và thanh toán	Bắt buộc xác thực
2	Mỗi đơn hàng gắn với một người dùng	Không dùng chung
3	Thanh toán chỉ áp dụng cho đơn hợp lệ	Đơn chưa hủy

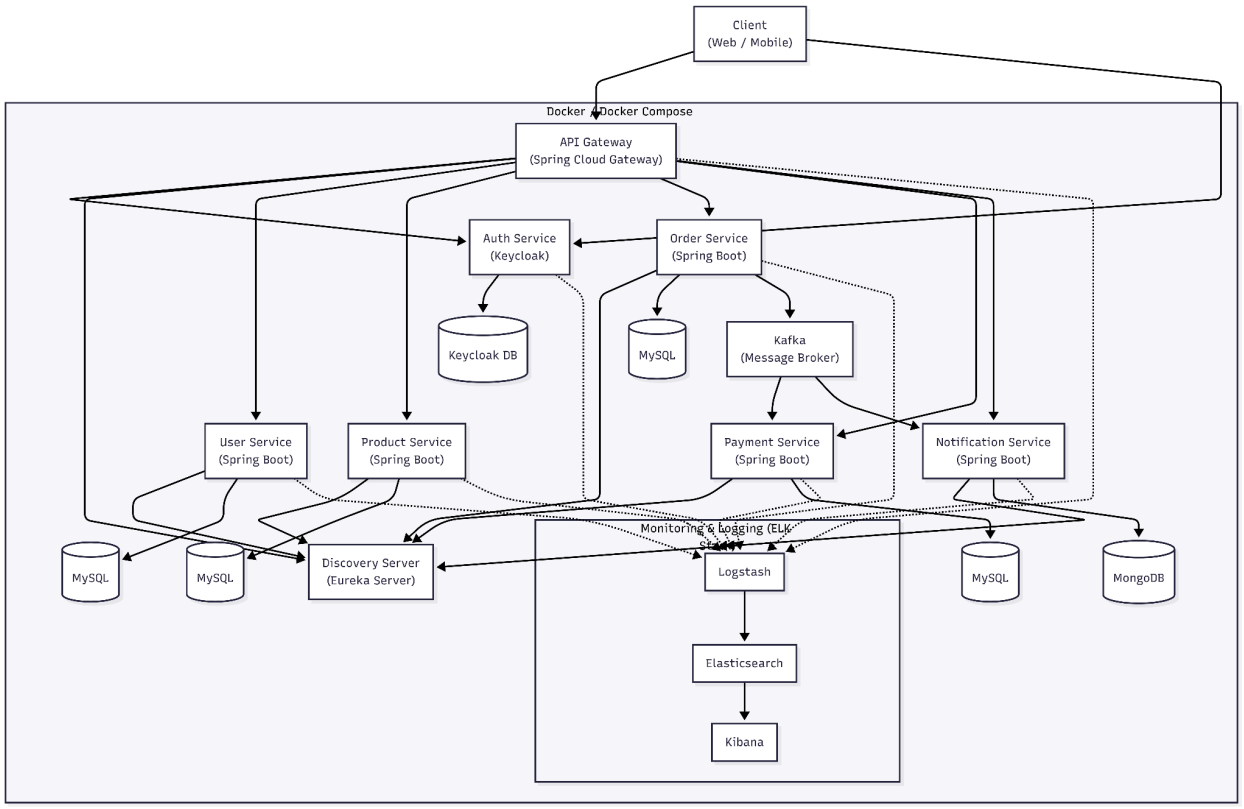
c) Bộ phận quản trị hệ thống (Admin)

STT	Tên yêu cầu	Loại nghiệp vụ	Ghi chú
1	Quản lý người dùng	Quản lý	User Service
2	Phân quyền người dùng	Quản lý	User/Admin
3	Quản lý sản phẩm	Quản lý	Product Service
4	Quản lý tồn kho	Quản lý	Inventory Service
5	Quản lý đơn hàng	Quản lý	Order Service

- ***Quy định:***

STT	Mô tả chi tiết	Ghi chú
1	Chỉ tài khoản ADMIN được truy cập chức năng quản trị	Phân quyền
2	Admin có quyền chỉnh sửa dữ liệu hệ thống	Toàn quyền
3	Mọi thao tác quản trị phải được ghi nhận	Phục vụ kiểm soát

2. Sơ đồ kiến trúc hệ thống



3. Đặc tả Use-case hệ thống

3.1 Sơ đồ lớp

- API GATEWAY

- **Lớp RouteConfig:** Định nghĩa ánh xạ giữa các Endpoint và Service tương ứng.
- **Lớp JwtAuthenticationFilter:** Tiền xử lý mọi yêu cầu, xác thực tính hợp lệ của mã Token.
- **Phương thức:** filter(), routeRequest(), handleUnauthorized().

-USER SERVICE

- **Thực thể:** User, Role, UserRole (Mối quan hệ n-n giữa người dùng và quyền hạn).
- **Thuộc tính:** id, username, password, email, status.
- **Phương thức:** authenticate(), register(), assignRole().

-PRODUCT SERVICE

- **Thực thể:** Product, Category.
- **Thuộc tính:** id, name, price, description, categoryId.
- **Phương thức:** getAllProducts(), getProductById(), saveProduct().

-ORDER SERVICE

- **Thực thể:** Order, OrderItem.
- **Thuộc tính:** id, userId, totalAmount, status (PENDING, PAID, CANCELLED).
- **Phương thức:** placeOrder(), updateOrderStatus(), cancelOrder().

-PAYMENT SERVICE

- **Thực thể:** Payment.
- **Thuộc tính:** id, orderId, transactionId, paymentMethod, status.
- **Phương thức:** processPayment(), validateTransaction().

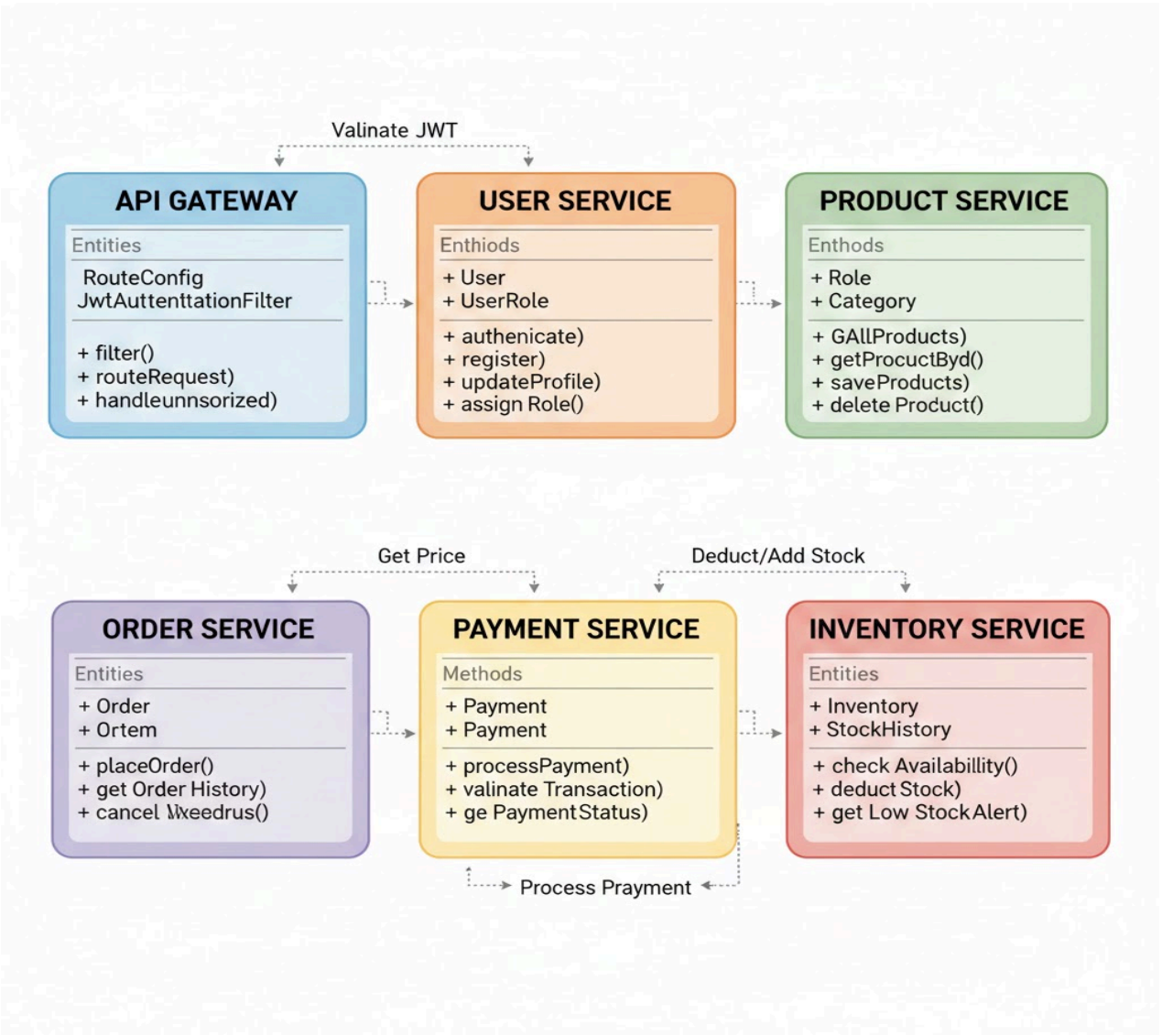
-INVENTORY SERVICE

- **Thực thể:** Inventory, StockHistory.
- **Thuộc tính:** id, productId, stockQuantity, type (IMPORT/EXPORT).
- **Phương thức:** checkAvailability(), deductStock(), addStock().

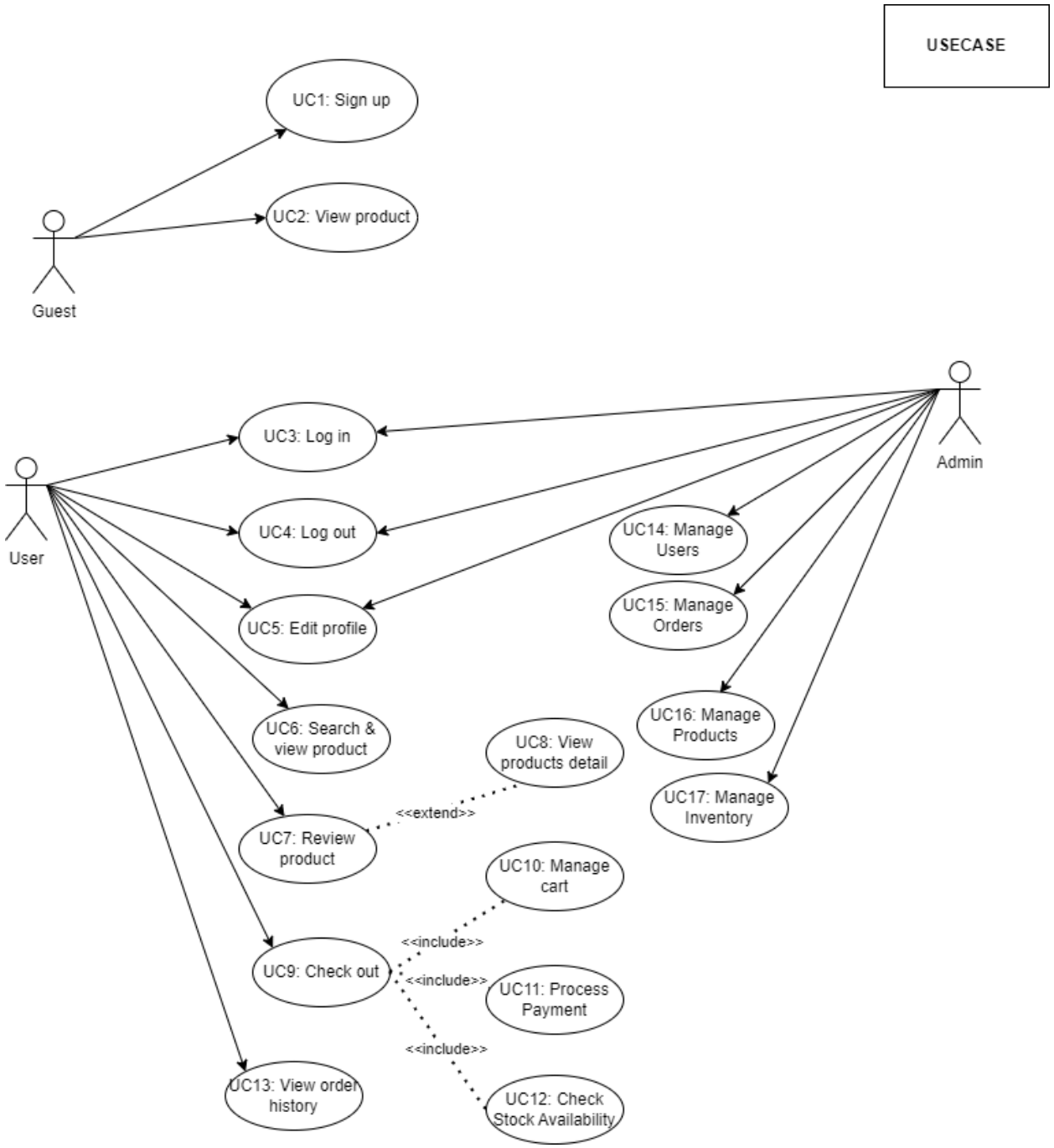
MỐI QUAN HỆ GIỮA CÁC SERVICE (Logic Diagram)

Để đảm bảo tính toàn vẹn dữ liệu và luồng nghiệp vụ thông suốt, hệ thống thiết lập các cơ chế tương tác (Inter-service Communication) như sau:

- Xác thực tập trung qua API Gateway: * API Gateway đóng vai trò là "Entry Point". Khi nhận request từ Client, Gateway sẽ thực hiện gọi đồng bộ (Synchronous Call) sang User Service để xác thực JWT Token và kiểm tra quyền hạn (Authorization) trước khi điều hướng request tới các service nghiệp vụ.
- Đảm bảo thông tin giá trị thực tế (Order ↔ Product): * Khi khách hàng tạo đơn, Order Service truy vấn dữ liệu từ Product Service thông qua productId. Việc này giúp hệ thống lấy được thông tin giá gốc (unit price) tại thời điểm đặt hàng, tránh sai lệch nếu sản phẩm thay đổi giá sau đó.
- Quản lý tồn kho nghiêm ngặt (Order ↔ Inventory): * Order Service gửi yêu cầu tới Inventory Service thông qua phương thức deductStock() để thực hiện cơ chế "giữ chỗ" hàng hóa (Reserve stock). Điều này đảm bảo không xảy ra tình trạng bán quá số lượng tồn kho (Over-selling).
- Hoàn tất giao dịch thanh toán (Order ↔ Payment): * Sau khi kho đã được giữ, Order Service điều hướng luồng xử lý tới Payment Service.
    - Cơ chế bù trừ (Compensating Transaction): Đây là điểm mấu chốt trong Microservices. Nếu Payment Service phản hồi trạng thái thất bại, Order Service sẽ kích hoạt một lệnh gọi ngược lại tới Inventory Service (phương thức addStock()) để hoàn kho, đồng thời cập nhật trạng thái đơn hàng thành "Cancelled".



3.2. Use-case tổng quan



3.3 Đặc tả Use-case và lược đồ sequence, collaboration

Use-case đăng ký tài khoản

Thành phần	Nội dung chi tiết
Mã Use Case	UC-01
Tên Use Case	Guest

Actor(s)	Guest
Mô tả ngắn	Cho phép actor tạo tài khoản mới để tham gia vào hệ thống.
Tiền điều kiện	Actor chưa đăng nhập vào hệ thống.
Hậu điều kiện	Tài khoản người dùng được tạo và kích hoạt thành công trong User Service.
Luồng sự kiện chính (Main flow)	1. Actor truy cập trang chủ và nhấn biểu tượng đăng nhập để chuyển sang trang đăng nhập. 2. Tại màn hình đăng nhập, actor ấn vào liên kết “Chưa có tài khoản? Đăng ký”. 3. Actor điền đầy đủ thông tin: Họ, tên, email, password và nhấn nút Đăng ký. 4. Hệ thống (User Service) tạo tài khoản mới (trạng thái chờ), gửi mã xác nhận đến email và chuyển đến trang nhập mã kích hoạt. 5. Actor nhập mã xác nhận từ email. 6. Hệ thống kiểm tra và kích hoạt tài khoản. 7. Hệ thống chuyển Actor về trang chủ với trạng thái đã đăng nhập.

Luồng thay thế (Alternative flow)	4.1. Actor chọn liên kết “Đã có tài khoản? Đăng nhập”: Hệ thống chuyển Actor đến trang đăng nhập (thực hiện tiếp UC-02).
Luồng ngoại lệ (Exception flow)	3.1. Thông tin nhập không đúng định dạng email hoặc đã tồn tại: Hệ thống thông báo lỗi, quay lại bước 3. 5.1. Nhập mã xác nhận không chính xác: Hệ thống hiển thị thông báo, quay lại bước 5.

Use-case đăng nhập tài khoản

Thành phần	Nội dung chi tiết
Mã Use Case	UC-02
Tên Use Case	Login
Actor(s)	Người dùng (User), Quản trị viên (Admin)
Mô tả ngắn	Xác thực danh tính người dùng để truy cập các chức năng của thành viên.
Tiền điều kiện	Actor đã có tài khoản hợp lệ và đã được kích hoạt.
Hậu điều kiện	Hệ thống cấp mã JWT Token hợp lệ. Actor truy cập được vào các tài nguyên giới hạn.

Luồng sự kiện chính (Main flow)	1. Actor truy cập vào trang Đăng nhập. 2. Actor nhập Email và Mật khẩu, sau đó nhấn nút "Đăng nhập". 3. API Gateway điều phối yêu cầu đến User Service để thực hiện xác thực. 4. User Service kiểm tra thông tin trong cơ sở dữ liệu và tạo mã định danh JWT. 5. Hệ thống lưu Token vào trình duyệt và chuyển Actor về trang chủ.
Luồng ngoại lệ (Exception flow)	3.1. Sai Email hoặc mật khẩu: Hệ thống thông báo lỗi xác thực. 3.2. Tài khoản chưa được kích hoạt: Hệ thống chuyển Actor sang trang nhập mã kích hoạt (UC-01 bước 4).

Use-case đăng xuất tài khoản

Thành phần	Nội dung chi tiết
Mã Use Case	UC-03
Tên Use Case	Log out
Actor(s)	Người dùng (User), Quản trị viên (Admin)
Mô tả ngắn	Kết thúc phiên làm việc hiện tại của người dùng.

Tiền điều kiện	Actor đang ở trạng thái đăng nhập hợp lệ.
Hậu điều kiện	Token của Actor bị hủy/xóa. Actor trở về trạng thái Khách vãng lai.
Luồng sự kiện chính (Main flow)	1. Actor nhấn vào biểu tượng tài khoản trên thanh điều hướng và chọn "Đăng xuất". 2. Hệ thống thực hiện xóa JWT Token tại trình duyệt của Actor. 3. Hệ thống chuyển hướng Actor về trang chủ hệ thống.

Use-case: Tạo đơn hàng và thanh toán

Thành phần	Nội dung chi tiết
Mã Use Case	UC-04
Tên Use Case	Tạo đơn hàng và thanh toán
Actor(s)	Người dùng (User)
Mô tả ngắn	Cho phép người dùng tạo đơn hàng và thực hiện thanh toán cho các sản phẩm đã chọn trong giỏ hàng.
Tiền điều kiện	Actor đã đăng nhập hệ thống hợp lệ và giỏ hàng có ít nhất một sản phẩm.

Hậu điều kiện	Đơn hàng được tạo và cập nhật trạng thái **PAID** nếu thanh toán thành công, hoặc **CANCELLED** nếu thanh toán thất bại.
Luồng sự kiện chính (Main flow)	Actor gửi yêu cầu tạo đơn hàng từ giỏ hàng. Hệ thống tiếp nhận yêu cầu thông qua API Gateway và kiểm tra JWT Token. API Gateway chuyển yêu cầu hợp lệ đến Order Service. Order Service truy vấn Product Service để lấy thông tin và giá sản phẩm tại thời điểm đặt hàng. Order Service gọi Inventory Service để kiểm tra và giữ chỗ tồn kho. Order Service tạo đơn hàng với trạng thái **PENDING**. Order Service gọi Payment Service để thực hiện thanh toán. Payment Service xử lý thanh toán và phản hồi kết quả.

	Order Service cập nhật trạng thái đơn hàng thành **PAID**. Hệ thống thông báo kết quả thanh toán thành công cho Actor.
Luồng thay thế (Alternative flow)	5.1. Nếu Inventory Service phản hồi không đủ tồn kho: → Hệ thống hủy thao tác tạo đơn và thông báo sản phẩm đã hết hàng cho Actor.
Luồng ngoại lệ (Exception flow)	8.1. Nếu thanh toán thất bại: → Order Service cập nhật trạng thái đơn hàng thành **CANCELLED**. → Order Service gọi Inventory Service để hoàn kho. → Hệ thống thông báo thanh toán không thành công cho Actor.

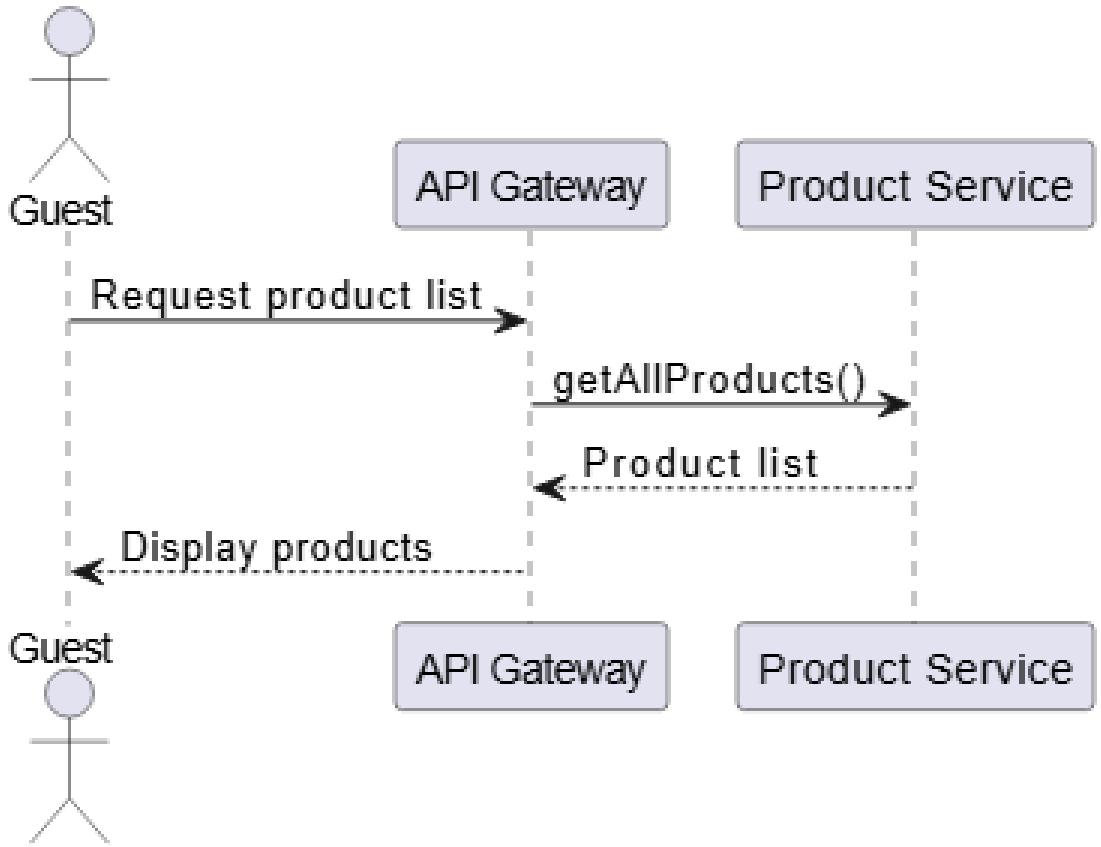
Use-case cho **quản trị viên hệ thống (Admin)**

Thành phần	Nội dung chi tiết
Mã Use Case	UC-06
Tên Use Case	Quản lý sản phẩm

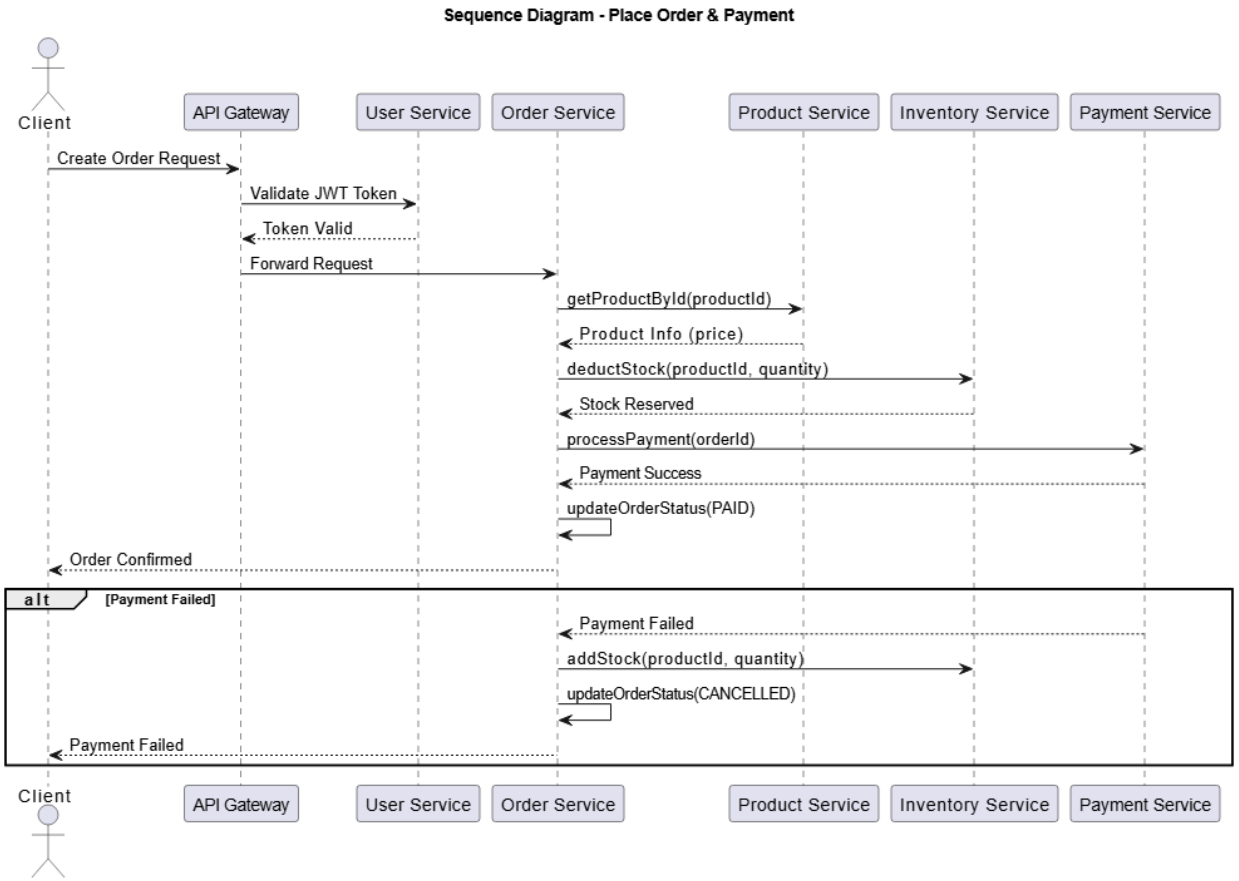
Actor(s)	Quản trị viên (Admin)
Mô tả ngắn	Cho phép Admin thêm, sửa, xóa và cập nhật thông tin sản phẩm.
Tiền điều kiện	**Actor đã đăng nhập với quyền ADMIN.**
Hậu điều kiện	Thông tin sản phẩm được cập nhật trong hệ thống.
Luồng sự kiện chính (Main flow)	Admin đăng nhập hệ thống. Admin chọn chức năng quản lý sản phẩm. Hệ thống xác thực quyền ADMIN qua API Gateway. API Gateway chuyển yêu cầu đến Product Service. Product Service cập nhật dữ liệu sản phẩm. Hệ thống thông báo thao tác thành công.

Lược đồ Sequence

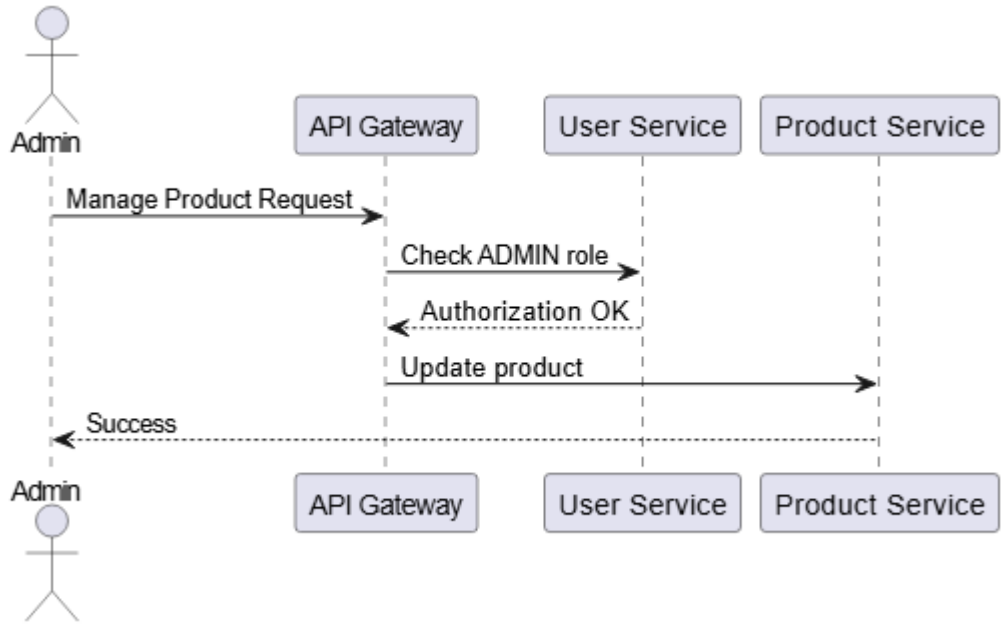
Sequence – Xem danh sách sản phẩm (Guest)



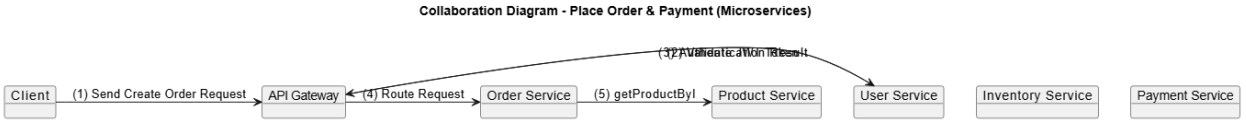
2. Sequence – Tạo đơn hàng & thanh toán (User)



Sequence – Quản lý sản phẩm (Admin)



Lược đồ Collaboration tổng quát theo nghiệp vụ



Mô tả sự phối hợp giữa các Microservices trong nghiệp vụ tạo đơn hàng và thanh toán.

III. NHIỆM VỤ CỦA CÁC DỊCH VỤ MICROSERVICES

1. Xác định và mô tả các dịch vụ cụ thể

Hệ thống được xây dựng theo kiến trúc microservices, trong đó mỗi dịch vụ đảm nhiệm một chức năng nghiệp vụ riêng biệt. Việc tách các chức năng thành các dịch vụ độc lập giúp hệ thống dễ mở rộng, dễ bảo trì và tăng tính linh hoạt trong quá trình phát triển.

1.1. Dịch vụ API Gateway

Tên gọi: API Gateway Service

Mục tiêu:
API Gateway đóng vai trò là điểm truy cập duy nhất của hệ thống microservices. Mục tiêu chính của dịch vụ này là tiếp nhận các yêu cầu từ phía client, sau đó định tuyến các yêu cầu này đến đúng dịch vụ backend tương ứng. Ngoài ra, API Gateway còn hỗ trợ các chức năng kiểm tra và xử lý ban đầu như xác thực cơ bản và lọc request.

Phạm vi:
Dịch vụ này xử lý tất cả các request từ client, đảm bảo việc giao tiếp giữa client và các microservices diễn ra thống nhất, an toàn và hiệu quả. API Gateway không xử lý logic nghiệp vụ phức tạp mà chỉ đóng vai trò trung gian.

1.2. Dịch vụ Quản lý Người dùng

Tên gọi: User Service

Mục tiêu:
Dịch vụ Quản lý Người dùng chịu trách nhiệm quản lý thông tin tài khoản và xác thực người dùng trong hệ thống. Mục tiêu chính của dịch vụ là cung cấp các chức năng liên quan đến người dùng như đăng ký, đăng nhập, quản lý thông tin tài khoản và phân quyền người dùng.

Phạm vi:
Dịch vụ này xử lý các yêu cầu liên quan đến người dùng, bao gồm quản lý danh sách người dùng, gán vai trò (USER, ADMIN) và đảm bảo việc xác thực người dùng được thực hiện đúng và an toàn.

1.3. Dịch vụ Quản lý Sản phẩm

Tên gọi: Product Service

Mục tiêu:
Dịch vụ Quản lý Sản phẩm chịu trách nhiệm quản lý thông tin sản phẩm trong hệ thống. Mục tiêu của dịch vụ là cung cấp các chức năng CRUD (Create, Read, Update, Delete) cho sản phẩm, giúp hệ thống lưu trữ và truy xuất thông tin sản phẩm một cách hiệu quả.

Phạm vi:
Dịch vụ này xử lý các yêu cầu liên quan đến việc thêm mới, chỉnh sửa, xóa sản phẩm, cũng như cho phép người dùng và quản trị viên xem danh sách sản phẩm và chi tiết từng sản phẩm.

1.4. Dịch vụ Quản lý Đơn hàng

Tên gọi: Order Service

Mục tiêu:
Dịch vụ Quản lý Đơn hàng chịu trách nhiệm xử lý toàn bộ vòng đời của một đơn hàng trong hệ thống. Mục tiêu chính là tạo đơn hàng mới, lưu trữ thông tin đơn hàng và theo dõi trạng thái của đơn hàng.

Phạm vi:
Dịch vụ này xử lý các yêu cầu liên quan đến đơn hàng, bao gồm tạo đơn hàng, truy vấn trạng thái đơn hàng và lưu trữ thông tin chi tiết của từng đơn. Dịch vụ cũng thể hiện sự giao tiếp giữa các microservices khác trong hệ thống.

1.5. Dịch vụ Thanh toán

Tên gọi: Payment Service

Mục tiêu:
Dịch vụ Thanh toán chịu trách nhiệm xử lý quá trình thanh toán cho các đơn hàng. Trong phạm vi đồ án, dịch vụ này không tích hợp cổng thanh toán thực tế mà chỉ mô phỏng quá trình thanh toán và trạng thái thanh toán.

Phạm vi:
Dịch vụ này xử lý các yêu cầu thanh toán từ Order Service, cập nhật trạng thái thanh toán của đơn hàng như SUCCESS hoặc FAILED, và trả kết quả thanh toán về cho hệ thống.

1.6. Dịch vụ Quản lý Tồn kho

Tên gọi: Inventory Service

Mục tiêu:
Dịch vụ Quản lý Tồn kho chịu trách nhiệm quản lý số lượng sản phẩm trong kho. Mục tiêu chính của dịch vụ là đảm bảo số lượng tồn kho được cập nhật chính xác khi có phát sinh đơn hàng.

Phạm vi:
Dịch vụ này xử lý các yêu cầu liên quan đến tồn kho, bao gồm kiểm tra số lượng sản phẩm và trừ số lượng tồn kho khi đơn hàng được tạo thành công, giúp đảm bảo tính nhất quán dữ liệu trong hệ thống.

IV. Thiết kế phần mềm

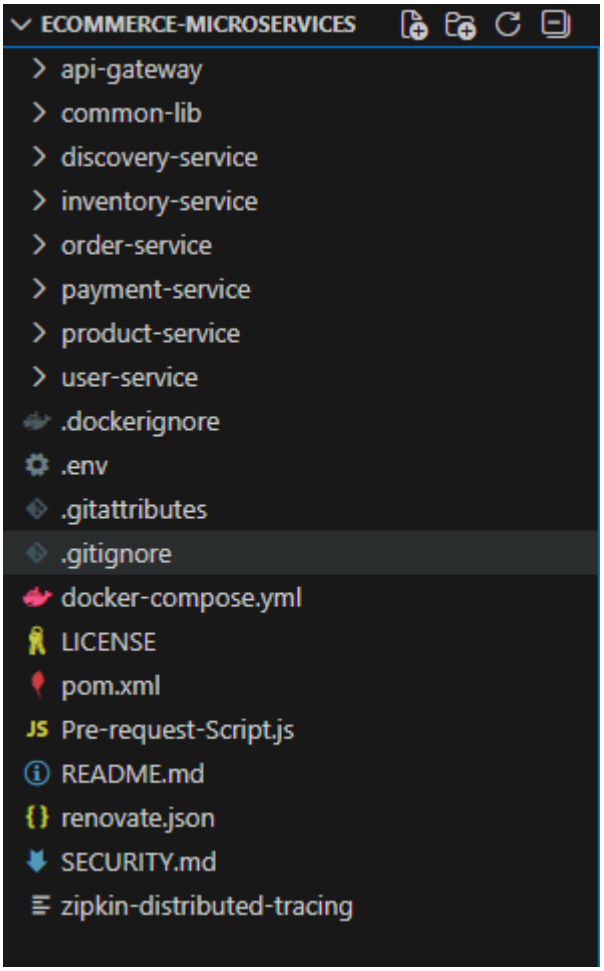
1. Công cụ và công nghệ áp dụng

- Ngôn ngữ lập trình: Java
- Nền tảng chạy ứng dụng: Java Virtual Machine (JVM)
- Framework: Spring Boot, Spring Cloud
- Kiến trúc hệ thống: Microservices
- IDE / Trình soạn thảo mã nguồn: Visual Studio Code
- Công cụ kiểm thử API: Postman
- Quản lý mã nguồn: Git
- Lưu trữ mã nguồn: GitHub

- Công nghệ đóng gói & triển khai: Docker, Docker Compose
- Message Broker: Apache Kafka
- Service Discovery: Eureka Server
- API Gateway: Spring Cloud Gateway
- Cơ sở dữ liệu:
  MySQL (dữ liệu nghiệp vụ có cấu trúc)
  MongoDB (dữ liệu phi cấu trúc, thông báo)
- Thiết kế hệ thống & sơ đồ: StarUML, Whimsical
- Thiết kế giao diện (UI/UX): Figma

2. Thiết kế kiến trúc chung của hệ thống Microservice

2.1 Cấu trúc code



Các Microservices

- api-gateway: Điểm truy cập duy nhất, định tuyến request và xác thực JWT.
- discovery-service: Đăng ký và phát hiện các service trong hệ thống.
- user-service: Quản lý người dùng, đăng nhập, phân quyền.

- product-service: Quản lý và tra cứu thông tin sản phẩm.
- order-service: Tạo và quản lý đơn hàng, điều phối nghiệp vụ.
- payment-service: Xử lý thanh toán đơn hàng.
- inventory-service: Quản lý tồn kho, giữ và hoàn hàng.
- common-lib: Thư viện dùng chung cho toàn hệ thống.

File cấu hình & quản lý

- .env: Biến môi trường.
- .dockerignore: Loại trừ file khi build Docker.
- .gitignore / .gitattributes: Cấu hình Git.
- docker-compose.yml: Triển khai toàn bộ hệ thống.
- pom.xml: Quản lý build và dependency.
- Pre-request-Script.js: Hỗ trợ test API bằng Postman.
- README.md: Mô tả và hướng dẫn dự án.
- LICENSE: Bản quyền mã nguồn.
- renovate.json: Tự động cập nhật thư viện.
- SECURITY.md: Chính sách bảo mật.
- zipkin-distributed-tracing: Theo dõi luồng request giữa các service.

V. Quản lý

Trong kiến trúc Microservices, công tác quản lý hệ thống đóng vai trò then chốt nhằm đảm bảo hệ thống vận hành ổn định, dễ mở rộng và dễ bảo trì. Do hệ thống được chia thành nhiều dịch vụ độc lập, việc quản lý không chỉ dừng lại ở mức ứng dụng mà còn bao gồm quản lý vòng đời dịch vụ, triển khai, giám sát và xử lý sự cố.

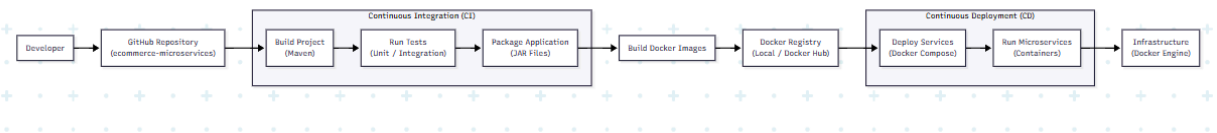
1. Cách quản lý từng dịch vụ

Mỗi Microservice trong hệ thống được thiết kế và quản lý như một đơn vị độc lập, có vòng đời phát triển riêng biệt. Các nguyên tắc quản lý chính bao gồm:

- Quản lý độc lập theo service:
  Mỗi dịch vụ (user-service, product-service, order-service,…) có mã nguồn, cấu hình và cơ sở dữ liệu riêng. Điều này giúp việc phát triển, sửa lỗi hoặc nâng cấp một dịch vụ không ảnh hưởng đến các dịch vụ còn lại.
- Đăng ký và phát hiện dịch vụ (Service Discovery):
  Discovery Service (Eureka Server) được sử dụng để quản lý danh sách các service đang hoạt động. Khi một service khởi động, nó sẽ tự động đăng ký với Eureka. Các service khác có thể truy vấn Eureka để tìm địa chỉ của service cần giao tiếp, giúp loại bỏ việc cấu hình cứng địa chỉ IP.
- Quản lý truy cập tập trung thông qua API Gateway:
  API Gateway đóng vai trò là điểm vào duy nhất của hệ thống, chịu trách nhiệm định tuyến request, kiểm tra xác thực và phân quyền. Cách tiếp cận này giúp đơn giản hóa việc quản lý bảo mật và kiểm soát truy cập cho toàn bộ hệ thống.
- Quản lý cấu hình linh hoạt:
  Các thông tin cấu hình như cổng dịch vụ, thông tin kết nối cơ sở dữ liệu, Kafka broker được tách khỏi mã nguồn và cấu hình thông qua biến môi trường hoặc file cấu hình. Điều này giúp hệ thống dễ dàng triển khai trên nhiều môi trường khác nhau.

2. Mô hình triển khai tự động và liên tục (CI/CD)

Để đảm bảo hệ thống có thể phát triển và triển khai nhanh chóng, mô hình CI/CD được áp dụng nhằm tự động hóa quy trình build, test và triển khai.



- Continuous Integration (CI):
  Mỗi khi có thay đổi mã nguồn, hệ thống CI sẽ tự động:
  - Kiểm tra mã nguồn
  - Build từng service độc lập
  - Chạy các bài kiểm tra cơ bản (unit test, integration test)

- Continuous Deployment (CD):
  Sau khi build thành công, các service có thể được đóng gói dưới dạng Docker Image và triển khai lên môi trường chạy. Việc sử dụng Docker giúp đảm bảo tính nhất quán giữa môi trường phát triển và môi trường triển khai.
- Triển khai độc lập từng service:
  Nhờ kiến trúc Microservices, việc cập nhật một dịch vụ (ví dụ product-service) không yêu cầu dừng toàn bộ hệ thống. Điều này giúp giảm thời gian gián đoạn và nâng cao tính sẵn sàng của hệ thống.
- Tự động hóa triển khai:
  Docker Compose (hoặc các công cụ điều phối khác như Kubernetes trong thực tế) giúp tự động hóa quá trình khởi động và liên kết giữa các service, giảm thiểu sai sót do cấu hình thủ công.

3. Giải pháp theo dõi và khắc phục sự cố trong kiến trúc Microservices

Do hệ thống Microservices có nhiều thành phần phân tán, việc theo dõi và xử lý sự cố cần được thiết kế ngay từ đầu.

- Theo dõi trạng thái dịch vụ (Health Check):
  Mỗi service cung cấp endpoint kiểm tra trạng thái (health check). Điều này cho phép phát hiện sớm các dịch vụ gặp sự cố và thực hiện các hành động khôi phục.
- Giám sát và logging tập trung:
  Log của các service được ghi lại nhằm phục vụ việc phân tích lỗi và theo dõi hành vi hệ thống. Trong hệ thống thực tế, các công cụ như ELK Stack (Elasticsearch, Logstash, Kibana) có thể được sử dụng để tập trung hóa và trực quan hóa log.
- Theo dõi luồng xử lý (Distributed Tracing):
  Việc tích hợp các công cụ theo dõi luồng request giữa các service (ví dụ Zipkin) giúp xác định chính xác service nào gây ra độ trễ hoặc lỗi trong một nghiệp vụ phức tạp như tạo đơn hàng và thanh toán.
- Cơ chế chịu lỗi và phục hồi:
  Các service được thiết kế theo hướng chịu lỗi, trong đó nếu một service gặp sự cố, các service khác vẫn có thể tiếp tục hoạt động. Docker hỗ trợ tự động khởi động lại container khi gặp lỗi, giúp hệ thống nhanh chóng phục hồi.

VI. Kết quả và nhận xét

1. Kết quả đạt được

Sau quá trình nghiên cứu, thiết kế và triển khai hệ thống theo kiến trúc Microservices, nhóm đã đạt được các kết quả chính như sau:

- Xây dựng thành công mô hình kiến trúc Microservices cho hệ thống thương mại điện tử:
  Hệ thống được chia thành nhiều dịch vụ độc lập như quản lý người dùng, sản phẩm, đơn hàng, thanh toán,… mỗi dịch vụ đảm nhiệm một chức năng riêng biệt, đúng với nguyên tắc của kiến trúc Microservices.
- Triển khai cơ chế phát hiện dịch vụ và định tuyến tập trung:
  Discovery Service giúp các dịch vụ tự động đăng ký và phát hiện lẫn nhau, trong khi API Gateway đóng vai trò là cổng giao tiếp duy nhất với phía client. Cách tiếp cận này giúp hệ thống linh hoạt hơn khi mở rộng hoặc thay đổi quy mô.
- Áp dụng giao tiếp bất đồng bộ thông qua hệ thống message broker:
  Việc sử dụng Kafka giúp các dịch vụ giao tiếp với nhau theo mô hình bất đồng bộ, giảm sự phụ thuộc trực tiếp giữa các service, từ đó nâng cao khả năng mở rộng và độ ổn định của hệ thống.
- Đóng gói và triển khai hệ thống bằng Docker:
  Các thành phần hạ tầng như cơ sở dữ liệu và message broker được triển khai bằng Docker, giúp quá trình cài đặt và chạy hệ thống trở nên nhất quán, dễ dàng và giảm phụ thuộc vào môi trường.
- Đáp ứng được các đặc trưng của hệ thống phân tán:
  Hệ thống thể hiện rõ các đặc điểm của hệ thống phân tán như phân tán dữ liệu, phân tán xử lý, khả năng mở rộng và khả năng chịu lỗi ở mức cơ bản.

2. Những khó khăn và hạn chế

Bên cạnh các kết quả đạt được, hệ thống vẫn còn tồn tại một số hạn chế nhất định:

- Độ phức tạp trong quản lý và cấu hình hệ thống:
  Do hệ thống bao gồm nhiều service độc lập, việc cấu hình và đồng bộ các thành phần đòi hỏi nhiều công sức hơn so với kiến trúc đơn khối (Monolithic).
- Khó khăn trong quá trình tích hợp và debug:
  Khi xảy ra lỗi trong một nghiệp vụ phức tạp liên quan đến nhiều service, việc xác định nguyên nhân gốc gặp nhiều khó khăn nếu thiếu các công cụ giám sát và theo dõi chuyên sâu.
- Hệ thống chưa được triển khai đầy đủ các cơ chế nâng cao:
  Các giải pháp như cân bằng tải tự động, giám sát hiệu năng nâng cao, bảo mật phân tán hoặc triển khai trên nền tảng Kubernetes mới chỉ dừng lại ở mức định hướng, chưa được triển khai toàn diện trong phạm vi bài tập.
- Giới hạn về quy mô thử nghiệm:
  Hệ thống chủ yếu được triển khai và kiểm thử trong môi trường học tập, với số lượng người dùng và dữ liệu còn hạn chế, chưa phản ánh đầy đủ các thách thức của hệ thống phân tán ở quy mô lớn.

3. Hướng phát triển trong tương lai

Dựa trên kết quả đạt được, nhóm đề xuất một số hướng phát triển và cải tiến cho hệ thống trong tương lai:

- Triển khai hệ thống điều phối container:
  Áp dụng Kubernetes để quản lý và mở rộng các Microservice một cách tự động, nâng cao khả năng chịu lỗi và cân bằng tải.
- Hoàn thiện hệ thống giám sát và logging tập trung:
  Tích hợp các công cụ như Prometheus, Grafana hoặc ELK Stack nhằm giám sát hiệu năng, theo dõi log và phát hiện sớm sự cố.
- Tăng cường bảo mật cho hệ thống:
  Áp dụng các cơ chế xác thực và phân quyền nâng cao như OAuth2, JWT nhằm đảm bảo an toàn dữ liệu và kiểm soát truy cập giữa các dịch vụ.
- Mở rộng chức năng nghiệp vụ:
  Phát triển thêm các tính năng như quản lý khuyến mãi, đánh giá sản phẩm, phân tích dữ liệu người dùng để hoàn thiện hệ thống thương mại điện tử.

4. Nhận xét chung

Thông qua đề tài này, nhóm đã có cơ hội tiếp cận và thực hành kiến trúc Microservices – một trong những kiến trúc phổ biến và quan trọng trong các hệ thống phân tán hiện đại. Quá trình triển khai giúp nhóm hiểu rõ hơn về cách tổ chức, quản lý và vận hành một hệ thống phân tán, cũng như những thách thức đi kèm. Kết quả đạt được là nền tảng quan trọng để tiếp tục nghiên cứu và phát triển các hệ thống phân tán quy mô lớn hơn trong tương lai.