**Câu 1.**

**1. JWT là gì?**

**JWT (JSON Web Token)** là một phương thức nhỏ gọn, an toàn với URL để đại diện cho các thông tin (claims) được truyền giữa hai bên: **client và server**.

Các claims được mã hóa dưới dạng đối tượng JSON và dùng làm **payload** của cấu trúc **JWS (JSON Web Signature)** hoặc dưới dạng **plaintext** trong **JWE (JSON Web Encryption)**.

Các thông tin này có thể được **ký số** (digital signed) hoặc **bảo vệ toàn vẹn** bằng mã xác thực thông điệp (**MAC**) và/hoặc **mã hóa**.

**2. Cấu trúc của JWT**

Một JWT có 3 phần, ngăn cách bằng dấu chấm ., theo định dạng:

css

header.payload.signature

**3. Phần Header**

Chỉ định **loại token** (thường là "JWT") và **thuật toán** dùng để ký token (ví dụ: HMAC SHA256, RSA,...).

Đây là một đối tượng JSON đơn giản, được mã hóa dưới định dạng **Base64Url** để đảm bảo an toàn khi truyền qua URL.

**4. Phần Payload**

* Chứa các **claims** – tức là các phát biểu về một đối tượng (thường là người dùng) và các dữ liệu bổ sung.
* Các loại claims:
  + **Registered claims**: Các claims tiêu chuẩn được đề xuất sử dụng như:
    - iss – người phát hành token
    - sub – chủ thể của token (người dùng)
    - aud – người nhận token
    - exp – thời gian hết hạn
    - nbf – thời điểm không chấp nhận token trước đó
    - iat – thời điểm phát hành
    - jti – mã định danh duy nhất của token
  + **Public claims**:
    - Không bắt buộc, nhưng được khuyến nghị để hỗ trợ khả năng tương thích.
    - Ví dụ: name, email, locale,...
  + **Private claims**:
    - Tùy chỉnh theo ứng dụng.
    - Ví dụ: department, role, permissions...

**5. Phần Signature (Chữ ký)**

* Dùng để xác minh **tính toàn vẹn** và **độ tin cậy** của token.
* Luôn cần kiểm tra chữ ký trước khi lưu hoặc sử dụng token.

**6. So sánh JWT vs Session**

| **Tiêu chí** | **JWT** | **Session** |
| --- | --- | --- |
| Trạng thái | Stateless (không lưu trạng thái) | Stateful (lưu trạng thái) |
| Lưu trữ | Phía client (thường là localStorage/cookie) | Phía server |
| Khả năng mở rộng | Tốt, dễ mở rộng hệ thống | Khó mở rộng nếu dùng memory |

**7. JWT với OAuth2 trong Spring Security**

**Các vai trò trong luồng OAuth2:**

* **Resource Owner**: Chủ sở hữu tài nguyên (người dùng)
* **Client**: Ứng dụng cần truy cập tài nguyên
* **Resource Server**: API chứa tài nguyên
* **Authorization Server**: Server xác thực người dùng và cấp access token

Nguồn tham khảo: <https://auth0.com/docs/authenticate/protocols/oauth>

**8. Chìa khóa RSA cố định cho Authorization & Resource Server**

* Authorization Server sử dụng JwtEncoder để ký access token bằng **khóa riêng RSA**.
* Resource Server sử dụng JwtDecoder để xác minh access token với **khóa công khai RSA**.
* **Mặc định**, Spring Security tạo khóa RSA trong bộ nhớ mỗi khi khởi động lại server → token cũ sẽ không còn hợp lệ.
* → **Khuyến nghị**: Sử dụng **khóa RSA cố định (lưu file)** thay vì sinh động trong bộ nhớ.

**9. Các vai trò (Roles) và quyền hạn (Permissions)**

**Vai trò (Roles):**

USER

ADMIN

SUPER\_ADMIN

**Quyền hạn (Permissions):**

READ

WRITE

DELETE

UPDATE

**Cách hoạt động JWT**

**🔐 1. Đăng ký người dùng (User Registration)**

**➤ Bước 1 (Client → Server):**

Client gửi yêu cầu POST đến endpoint /api/auth/signup

json

{

"username": "...",

"email": "...",

"role": "...",

"password": "..."

}

**➤ Bước 2 (Server):**

* Server kiểm tra xem username/email đã tồn tại chưa.
* Nếu chưa có, thì **lưu thông tin người dùng** vào cơ sở dữ liệu.

**➤ Bước 3 (Server → Client):**

Trả về thông báo "Registered successfully!"

**🔑 2. Đăng nhập người dùng (User Login)**

**➤ Bước 1 (Client → Server):**

Client gửi yêu cầu POST đến endpoint /api/auth/signin

json

{

"username": "...",

"password": "..."

}

**➤ Bước 2 (Server):**

* Xác thực thông tin đăng nhập.
* Nếu hợp lệ, **tạo chuỗi JWT** bằng cách ký thông tin người dùng với một **mã bí mật (secret key)**.

**➤ Bước 3 (Server → Client):**

Server trả về:

json

{

"token": "JWT...",

"type": "Bearer",

"user info": { ... },

"authorities": [ ... ]

}

**🔓 3. Truy cập tài nguyên (Access Resource)**

**➤ Bước 4 (Client → Server):**

Client gửi yêu cầu đến API khác, kèm theo **JWT trong phần Authorization Header**:

makefile

Authorization: Bearer <JWT>

**➤ Bước 5 (Server):**

* Server kiểm tra **chữ ký JWT** để xác thực token.
* Nếu hợp lệ, **trích xuất thông tin người dùng** từ JWT.
* Xác thực quyền truy cập dựa trên **authorities** (vai trò/quyền) của người dùng.

**➤ Bước 6 (Server → Client):**

Trả về **dữ liệu/tài nguyên** tương ứng với quyền của người dùng.

👉 **Tóm lại**, JWT giúp **xác thực không trạng thái**: sau khi đăng nhập một lần và nhận token, client có thể gửi token đó trong các yêu cầu tiếp theo để truy cập dữ liệu, **không cần đăng nhập lại** – tất cả nhờ thông tin được mã hóa sẵn trong token.

**Đề 2**

**Câu 1a: Cấu trúc của JWT có mấy thành phần? Mô tả chi tiết từng thành phần.**

**JWT gồm 3 thành phần chính**, được phân tách bởi dấu chấm (.):

1. **Header** (Phần đầu):
   * Chứa thông tin về loại token (luôn là JWT) và thuật toán dùng để ký (ví dụ: HS256, RS256).
   * Ví dụ:

json

Sao chépChỉnh sửa

{

"alg": "HS256",

"typ": "JWT"

}

1. **Payload** (Phần thân):
   * Chứa các dữ liệu (claims) được mã hóa, gồm 3 loại:
     + **Registered claims**: Các trường chuẩn như iss (issuer), exp (expiration), sub (subject), aud (audience).
     + **Public claims**: Các trường do cộng đồng định nghĩa chung.
     + **Private claims**: Các trường do ứng dụng tự định nghĩa.
   * Đây là nơi chứa thông tin người dùng (user info), vai trò (roles), quyền hạn (permissions),...
2. **Signature** (Chữ ký):
   * Dùng để xác minh token có bị chỉnh sửa không.
   * Được tạo bằng cách sử dụng thuật toán ký (như HMAC SHA256) trên header và payload cùng với một **secret key**.

✅ **Kết luận**: JWT gồm **3 phần**: Header, Payload, Signature, định dạng như sau:

xxxxx.yyyyy.zzzzz

**Câu 1b: Token có mấy loại phổ biến? Giải thích mục đích sử dụng của từng loại và nên lưu ở đâu.**

**Có 2 loại token phổ biến nhất:**

1. **Access Token**:
   * Mục đích: Xác thực và ủy quyền các yêu cầu truy cập tài nguyên (API).
   * Tuổi thọ ngắn (vài phút đến vài giờ).
   * **Nơi lưu trữ đề xuất**:
     + Nếu là ứng dụng mobile/native: lưu trong **Secure Storage**.
     + Nếu là Web App: lưu trong **memory** (RAM), không nên lưu trong localStorage hoặc sessionStorage để tránh bị tấn công XSS.
2. **Refresh Token**:
   * Mục đích: Dùng để lấy access token mới khi token cũ hết hạn mà không cần đăng nhập lại.
   * Tuổi thọ dài hơn access token.
   * **Nơi lưu trữ đề xuất**:
     + Với web app: nên lưu trong **HttpOnly Cookies** để giảm thiểu nguy cơ bị đánh cắp qua JavaScript.
     + Với mobile app: lưu trong **Secure Storage**.

**Đề 1**

**a. JWT có thể được sử dụng cho những mục đích gì?**

JWT được sử dụng trong các mục đích chính sau:

* **Xác thực người dùng (Authentication)**: Sau khi người dùng đăng nhập thành công, server sẽ tạo JWT và gửi về client. Các request sau đó sẽ kèm JWT để chứng minh danh tính.
* **Ủy quyền (Authorization)**: Dựa vào thông tin và quyền hạn (roles/permissions) trong JWT, server sẽ cho phép hoặc từ chối truy cập tài nguyên.
* **Trao đổi thông tin an toàn** giữa hai bên: Do JWT được ký (sign), nên đảm bảo dữ liệu không bị thay đổi trong quá trình truyền.

**b. Cấu trúc của JWT gồm những gì?**

JWT có 3 phần chính, phân tách bởi dấu chấm (.):

css

Sao chépChỉnh sửa

Header.Payload.Signature

* **Header**: chứa thông tin về loại token (JWT) và thuật toán ký (ví dụ: HS256, RS256).
* **Payload**: chứa các claims – tức là thông tin như ID người dùng, email, vai trò, thời gian hết hạn,...
* **Signature (Chữ ký)**: đảm bảo token không bị sửa đổi và được tạo bởi server hợp lệ.

**c. Hãy mô tả việc xác thực bằng JWT theo cách bạn hiểu**

1. **Người dùng đăng nhập** bằng username và password.
2. Nếu hợp lệ, server tạo một **token JWT** chứa thông tin người dùng và ký bằng secret key.
3. Client lưu token (thường trong localStorage hoặc cookie).
4. Khi gửi yêu cầu đến server, client sẽ đính kèm JWT trong **Authorization header**.
5. Server nhận JWT, **kiểm tra chữ ký** để xác thực tính hợp lệ và trích xuất thông tin người dùng.
6. Nếu token hợp lệ và chưa hết hạn, người dùng sẽ được phép truy cập tài nguyên.

**Câu 2**

**Kiến trúc**

**🧱 1. Monolithic Architecture**

**✅ Ưu điểm:**

* **Dễ phát triển ban đầu, đơn giản để triển khai.**
* **Debug và kiểm thử đơn giản do mọi thành phần nằm trong cùng 1 ứng dụng.**
* **Hiệu suất cao hơn khi chạy nội bộ không cần giao tiếp qua mạng.**

**❌ Nhược điểm:**

* **Khó mở rộng theo module → dễ trở thành “ứng dụng khổng lồ”.**
* **Dễ gây ra lỗi nếu thay đổi ở một module nhỏ.**
* **Khó áp dụng CI/CD cho từng phần riêng.**
* **Không linh hoạt khi muốn thay đổi công nghệ.**

**📌 Khi nào sử dụng:**

* **Dự án nhỏ, ít thay đổi, thời gian phát triển ngắn.**
* **Nhóm phát triển ít người.**

**⚙️ Công nghệ áp dụng:**

* **Spring Boot (Java), ASP.NET MVC (C#), Laravel (PHP), Express.js (Node.js)**

**📐 UML Tổng Quan:**

**pgsql**

**Sao chépChỉnh sửa**

**+-------------------------+**

**| Monolithic App |**

**|-------------------------|**

**| Controller |**

**| Service |**

**| Repository |**

**| Database (1 DB) |**

**+-------------------------+**

**🧩 2. Microservices Architecture**

**✅ Ưu điểm:**

* **Mỗi service là một thành phần độc lập → dễ triển khai, dễ scale.**
* **Dễ áp dụng CI/CD.**
* **Dễ dùng công nghệ khác nhau cho từng service (polyglot).**
* **Đảm bảo tính cô lập khi một service gặp lỗi.**

**❌ Nhược điểm:**

* **Phức tạp về giao tiếp (REST, gRPC, Message queue).**
* **Quản lý nhiều service → tốn công DevOps.**
* **Debug phức tạp hơn.**

**📌 Khi nào sử dụng:**

* **Dự án lớn, có nhiều đội phát triển.**
* **Cần mở rộng theo module hoặc tính năng.**
* **Yêu cầu tính sẵn sàng, hiệu năng cao.**

**⚙️ Công nghệ áp dụng:**

* **Spring Cloud, Netflix OSS, Docker, Kubernetes, Kafka/RabbitMQ, API Gateway (Zuul, Kong), Eureka, Consul**

**📐 UML Tổng Quan:**

**pgsql**

**Sao chépChỉnh sửa**

**+------------+ +------------+ +------------+**

**| Service A | | Service B | | Service C |**

**+------------+ +------------+ +------------+**

**| | |**

**+-------> API Gateway <-------------+**

**|**

**+-------------+**

**| Database(s) |**

**+-------------+**

**🏛️ 3. Layered Architecture (Kiến trúc tầng)**

**✅ Ưu điểm:**

* **Phân tách rõ ràng giữa các tầng: UI, Business, Data.**
* **Dễ maintain, dễ test đơn vị (unit test).**
* **Thích hợp với mô hình MVC.**

**❌ Nhược điểm:**

* **Giao tiếp giữa các tầng gây chậm trễ.**
* **Thay đổi xuyên tầng khó nếu không thiết kế tốt.**
* **Có thể dẫn đến phụ thuộc chặt giữa các tầng.**

**📌 Khi nào sử dụng:**

* **Dự án vừa đến lớn có cấu trúc rõ ràng.**
* **Dễ cho sinh viên, người mới học lập trình backend.**

**⚙️ Công nghệ áp dụng:**

* **Java Spring Boot (Controller – Service – Repository), .NET, Django, Angular**

**📐 UML Tổng Quan:**

**pgsql**

**Sao chépChỉnh sửa**

**+------------------+ <-- Presentation Layer (UI)**

**| Controller |**

**+------------------+**

**↓**

**+------------------+ <-- Business Layer**

**| Service |**

**+------------------+**

**↓**

**+------------------+ <-- Data Access Layer**

**| Repository / DAO |**

**+------------------+**

**↓**

**+------------------+ <-- Database Layer**

**| Database |**

**+------------------+**

**⚡ 4. Event-Driven Architecture**

**✅ Ưu điểm:**

* **Hệ thống rời rạc và phản ứng linh hoạt theo sự kiện.**
* **Dễ mở rộng hoặc thêm chức năng mới mà không ảnh hưởng hệ thống cũ.**
* **Giảm phụ thuộc giữa các thành phần (loose coupling).**

**❌ Nhược điểm:**

* **Khó debug, tracking flow (vì phi tuần tự).**
* **Cần hệ thống xử lý sự kiện đáng tin cậy.**
* **Phức tạp hơn để đảm bảo thứ tự và trạng thái hệ thống.**

**📌 Khi nào sử dụng:**

* **Hệ thống nhiều nghiệp vụ độc lập, cần phản hồi nhanh (ngân hàng, IoT, thương mại điện tử).**
* **Hệ thống xử lý dữ liệu lớn theo luồng.**

**⚙️ Công nghệ áp dụng:**

* **Kafka, RabbitMQ, Redis Pub/Sub, AWS SNS/SQS, Azure Event Grid**

**📐 UML Tổng Quan:**

**lua**

**Sao chépChỉnh sửa**

**+-----------+ +------------+ +-----------+**

**| Producer | ---> | Event Bus | ---> | Consumer |**

**+-----------+ +------------+ +-----------+**

**| |**

**+------------> Event Handler ----------+**

**Đề 1**

**🧠 Đề bài yêu cầu:**

**Hệ thống có các chức năng:**

* **Xử lý đơn hàng (tạo, thanh toán, theo dõi).**
* **Quản lý sản phẩm (thêm, sửa, xóa).**
* **Quản lý khách hàng (đăng ký, đăng nhập, lịch sử mua hàng).**
* **Thống kê – báo cáo doanh thu, bán hàng.**

**✅ a. Liệt kê 3 mô hình kiến trúc phù hợp để xây dựng hệ thống trên**

1. **Kiến trúc Microservices**
2. **Kiến trúc Layered (tầng)**
3. **Kiến trúc Monolithic**

**✅ b. Chọn 1 mô hình kiến trúc phù hợp & giải thích lý do**

**👉 Chọn: Microservices Architecture**

**Giải thích lựa chọn:**

* **Hệ thống có nhiều chức năng riêng biệt (đơn hàng, khách hàng, sản phẩm, thống kê).**
* **Cần phân chia theo module để dễ phát triển độc lập.**
* **Hệ thống có thể mở rộng nhanh, dễ dàng bảo trì theo từng service.**
* **Hỗ trợ triển khai CI/CD, DevOps dễ dàng.**

**Phân tích ưu nhược điểm so với 2 mô hình còn lại:**

| **Kiến trúc** | **Ưu điểm** | **Nhược điểm** |
| --- | --- | --- |
| **Microservices** | **Dễ mở rộng, bảo trì, CI/CD, triển khai độc lập** | **Phức tạp, cần nhiều kỹ thuật DevOps** |
| **Layered** | **Cấu trúc rõ ràng, dễ học, dễ kiểm thử** | **Khó scale, các tầng phụ thuộc nhau** |
| **Monolithic** | **Dễ triển khai nhanh ban đầu, phù hợp dự án nhỏ** | **Khó bảo trì, khó mở rộng khi hệ thống lớn dần** |

**✅ c. Vẽ mô hình khái quát hệ thống với kiến trúc đã chọn (Microservices)**

**📐 Sơ đồ minh họa Microservices UML (dạng đơn giản):**

**+---------------------+**

**| API Gateway |**

**+---------------------+**

**/ | \**

**/ | \**

**v v v**

**+----------------+ +----------------+ +----------------+**

**| Order Service | | Product Service| | Customer Service|**

**+----------------+ +----------------+ +----------------+**

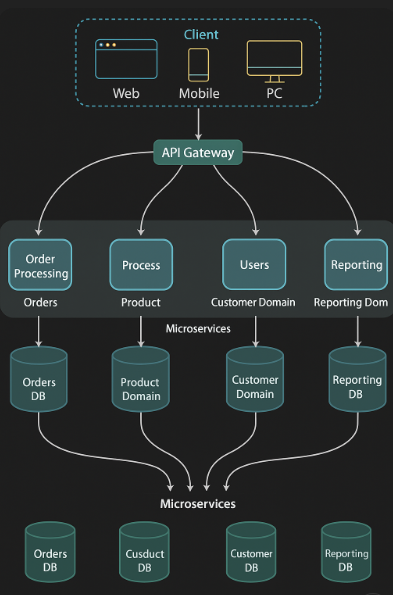
**\**

**\**

**+------------------+**

**| Reporting Service|**

**+------------------+**

****

**Các service có thể sử dụng cơ sở dữ liệu riêng hoặc chia sẻ tùy theo thiết kế.**

**✅ d. Liệt kê 3 công nghệ phù hợp với kiến trúc Microservices**

1. **Spring Boot + Spring Cloud – để xây dựng các service.**
2. **Docker & Kubernetes – để triển khai, quản lý, scale các service.**
3. **API Gateway (Zuul, Kong, Nginx) – định tuyến và kiểm soát truy cập.**

**(Có thể bổ sung thêm Kafka/RabbitMQ nếu hệ thống dùng Event-driven)**

**Đề 2**

**a. (CLO3 - 0.5đ) Chọn kiến trúc cho bài toán trên**

→ **Kiến trúc được chọn: Microservices kết hợp với Event-driven (hướng sự kiện)**

**b. (CLO3 - 1.0đ) Lý do chọn kiến trúc trên**

* Hệ thống thư viện có **nhiều chức năng độc lập**: đăng ký, làm thẻ thư viện, mượn sách, trả sách, ghi log hành động,...
* Việc tách thành **microservices** giúp dễ dàng mở rộng hoặc chỉnh sửa từng chức năng mà không ảnh hưởng toàn hệ thống.
* Kết hợp với kiến trúc **Event-driven** để xử lý các sự kiện như:
  + Ghi log khi mượn/trả sách dù thành công hay thất bại.
  + Gửi thông báo sau khi đăng ký thành công, tạo thẻ,...
* Cho phép hệ thống **linh hoạt, mở rộng dễ dàng**, hỗ trợ cả **Web và Mobile App**.

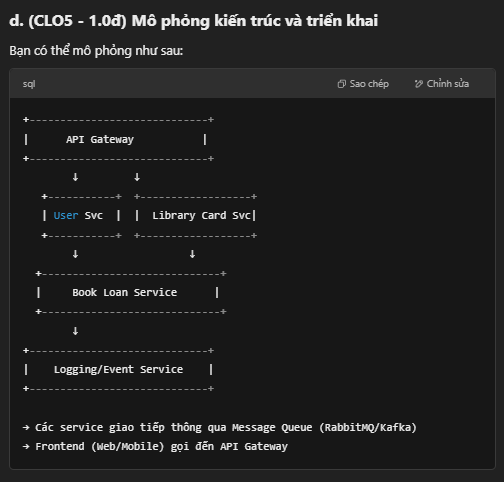
**c. (CLO2 - 1.0đ) Ưu/Nhược điểm của kiến trúc đã chọn**

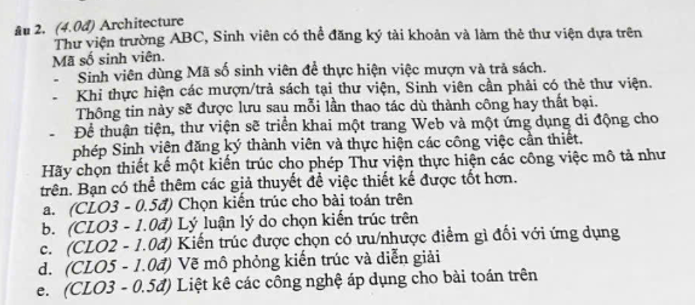
**Ưu điểm:**

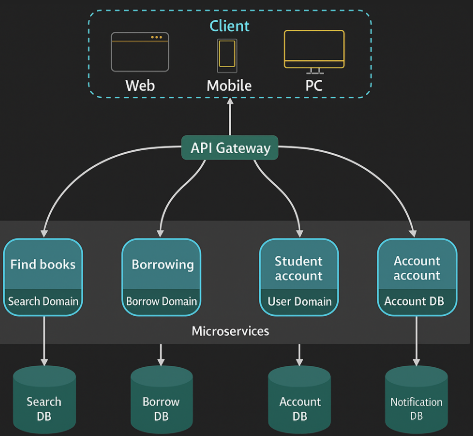
* **Tăng tính module hóa**: Dễ bảo trì và triển khai riêng từng service.
* **Khả năng mở rộng**: Mỗi service có thể scale độc lập.
* **Tăng tính phản hồi** với Event-driven: Ghi log, gửi thông báo... diễn ra không đồng bộ.
* **Tương thích đa nền tảng** (Web, mobile).

**Nhược điểm:**

* **Phức tạp hơn monolithic**: Cần triển khai hệ thống communication (REST, message queue,...).
* **Khó debug** hơn do nhiều service chạy riêng biệt.
* Yêu cầu kiến thức vững về kiến trúc và công cụ hỗ trợ (API Gateway, message broker,...).







**e. (CLO3 - 0.5đ) Các công nghệ có thể áp dụng**

* **Ngôn ngữ & Backend**: Node.js / Java / .NET
* **Frontend**: ReactJS (Web), React Native/Flutter (Mobile)
* **Giao tiếp**: REST API, gRPC
* **Message Queue**: Kafka / RabbitMQ (cho event-driven)
* **Database**: PostgreSQL / MongoDB
* **API Gateway**: NGINX / Kong / Spring Cloud Gateway
* **Auth**: JWT, OAuth2
* **Triển khai**: Docker, Kubernetes

**Câu 3**

**1. 🧍 Singleton Pattern**

**✅ Ưu điểm:**

* **Đảm bảo chỉ có một thể hiện duy nhất trong toàn bộ hệ thống.**
* **Cung cấp truy cập toàn cục đến đối tượng đó.**

**❌ Nhược điểm:**

* **Khó kiểm thử (testing) do trạng thái toàn cục.**
* **Nếu dùng sai cách, dễ vi phạm nguyên lý SRP (Single Responsibility).**

**📌 Áp dụng khi:**

* **Cần quản lý tài nguyên chung: Connection pool, logging, cấu hình hệ thống.**

**2. 🏭 Factory Method Pattern**

**✅ Ưu điểm:**

* **Tách logic khởi tạo đối tượng khỏi lớp sử dụng nó.**
* **Dễ mở rộng khi thêm loại đối tượng mới.**

**❌ Nhược điểm:**

* **Dễ tạo nhiều lớp con → phức tạp hóa cấu trúc.**
* **Code có thể trở nên khó đọc nếu dùng quá mức.**

**📌 Áp dụng khi:**

* **Hệ thống cần tạo nhiều loại đối tượng có cùng giao diện nhưng khác hành vi.**

**3. 🏭🏗️ Abstract Factory Pattern**

**✅ Ưu điểm:**

* **Tạo nhóm đối tượng liên quan mà không phụ thuộc lớp cụ thể.**
* **Đảm bảo tính đồng bộ giữa các đối tượng được tạo.**

**❌ Nhược điểm:**

* **Tăng độ phức tạp code.**
* **Mỗi nhóm sản phẩm cần 1 factory mới.**

**📌 Áp dụng khi:**

* **Hệ thống cần chuyển đổi môi trường (GUI, Database, OS) mà không ảnh hưởng code chính.**
* **Ví dụ: Giao diện hệ thống hỗ trợ cả Dark/Light Theme.**

**4. 🔁 State Pattern**

**✅ Ưu điểm:**

* **Loại bỏ if-else phức tạp khi xử lý nhiều trạng thái.**
* **Mỗi trạng thái là 1 class riêng → dễ quản lý.**

**❌ Nhược điểm:**

* **Tăng số lượng class.**
* **Có thể quá phức tạp với hệ thống nhỏ.**

**📌 Áp dụng khi:**

* **Đối tượng có nhiều trạng thái và hành vi thay đổi theo trạng thái đó.**
* **Ví dụ: Máy bán hàng tự động, game state, trạng thái đơn hàng.**

**5. 🧠 Strategy Pattern**

**✅ Ưu điểm:**

* **Tách biệt logic thuật toán khỏi lớp chính.**
* **Dễ thay đổi/hoán đổi thuật toán tại runtime.**

**❌ Nhược điểm:**

* **Tăng số lượng class.**
* **Client cần hiểu cách chọn Strategy phù hợp.**

**📌 Áp dụng khi:**

* **Cần thay đổi thuật toán xử lý động, ví dụ: thuật toán sắp xếp, tính phí giao hàng,...**

**6. 👀 Observer Pattern**

**✅ Ưu điểm:**

* **Cập nhật tự động khi có thay đổi → phù hợp mô hình Pub/Sub.**
* **Tăng khả năng mở rộng và giảm kết nối chặt giữa các lớp.**

**❌ Nhược điểm:**

* **Khó debug khi có nhiều observer.**
* **Có thể gây lỗi hiệu năng nếu không quản lý tốt.**

**📌 Áp dụng khi:**

* **Cần phản ứng theo sự kiện thay đổi, ví dụ: hệ thống thông báo, UI bindings, socket server,...**

**7. 🎨 Decorator Pattern**

**✅ Ưu điểm:**

* **Mở rộng chức năng mà không thay đổi lớp gốc.**
* **Dễ kết hợp nhiều chức năng động.**

**❌ Nhược điểm:**

* **Dễ dẫn đến nhiều lớp decorater lồng nhau → khó đọc.**
* **Cấu trúc class phức tạp.**

**📌 Áp dụng khi:**

* **Cần thêm chức năng động cho đối tượng mà không kế thừa.**
* **Ví dụ: thêm border, shadow cho thành phần UI.**

**Đề 1**

**🧠 Đề bài tóm tắt:**

**Bạn muốn gửi thông báo cho các người dùng đã đăng ký nhận khi có sự kiện xảy ra (ví dụ: có tin nhắn mới, cập nhật mới,...).**

**✅ a. Chọn 1 Design Pattern phù hợp**

**👉 Chọn: Observer Pattern**

**✅ b. Giải thích lý do chọn Observer Pattern**

* **Observer Pattern phù hợp với các hệ thống mà một đối tượng (Subject) cần thông báo thay đổi cho nhiều đối tượng khác (Observers) mà không cần biết chính xác chúng là ai.**
* **Trong bài toán này:**
  + **Người dùng đăng ký nhận thông báo là các Observer.**
  + **Hệ thống hoặc sự kiện (ví dụ: tin nhắn mới, cập nhật mới) là Subject.**
* **Khi sự kiện xảy ra, Subject sẽ notify() toàn bộ Observer, đảm bảo phản ứng theo thời gian thực và dễ mở rộng thêm loại thông báo mới (email, SMS, push,...).**

**✅ c. Vẽ Class Diagram cho bài toán**

**pgsql**

**Sao chépChỉnh sửa**

**+------------------+ observes +----------------------+**

**| Subject |<-------------------->| Observer |**

**|------------------| +----------------------+**

**| + attach() | | + update() |**

**| + detach() | +----------------------+**

**| + notify() | /\**

**+------------------+ |**

**/\ |**

**| |**

**| |**

**+------------------+ +-------------------------+**

**| NotificationSvc | | UserObserver |**

**|------------------| +-------------------------+**

**| + messages | | + update() |**

**+------------------+ +-------------------------+**

**Giải thích sơ đồ:**

* **Subject: interface hoặc lớp trừu tượng đại diện cho đối tượng phát thông báo.**
* **Observer: giao diện cho các đối tượng cần được thông báo.**
* **NotificationSvc: triển khai Subject, giữ danh sách người đăng ký.**
* **UserObserver: triển khai Observer, đại diện cho từng người dùng đăng ký nhận thông báo.**

****

**// Observer interface**

**public interface Observer {**

**void update(String message);**

**}**

**// Subject interface**

**public interface Subject {**

**void attach(Observer o);**

**void detach(Observer o);**

**void notifyObservers(String message);**

**}**

**// Concrete Subject**

**public class NotificationService implements Subject {**

**private List<Observer> observers = new ArrayList<>();**

**public void attach(Observer o) {**

**observers.add(o);**

**}**

**public void detach(Observer o) {**

**observers.remove(o);**

**}**

**public void notifyObservers(String message) {**

**for (Observer o : observers) {**

**o.update(message);**

**}**

**}**

**}**

**// Concrete Observer**

**public class User implements Observer {**

**private String name;**

**public User(String name) {**

**this.name = name;**

**}**

**public void update(String message) {**

**System.out.println(name + " nhận thông báo: " + message);**

**}**

**}**

**// Test**

**public class Main {**

**public static void main(String[] args) {**

**NotificationService service = new NotificationService();**

**User u1 = new User("Minh");**

**User u2 = new User("Hà");**

**service.attach(u1);**

**service.attach(u2);**

**service.notifyObservers("Có tin nhắn mới!");**

**}**

**}**

**Đề 2**

Câu 3

**a. Design Pattern phù hợp: State Pattern**

**b. Lý do chọn State Pattern:**

* Trạng thái của **một cuốn sách** thay đổi theo thời gian: Available, CheckedOut, OnHold, Returned, Lost.
* **State Pattern** giúp quản lý hành vi của đối tượng tùy theo trạng thái nội tại của nó.
* Khi sách ở trạng thái này, nó có hành vi khác với trạng thái kia (ví dụ: có thể mượn, không thể mượn, chỉ có thể đặt trước, v.v.).
* Tránh việc dùng quá nhiều if-else hay switch-case trong mã nguồn.



**d. Cài đặt mô phỏng bằng Java (hoặc ngôn ngữ khác)**

Ví dụ đơn giản bằng Java:

java

interface State {

void borrow(Book book);

void returnBook(Book book);

void placeHold(Book book);

void reportLost(Book book);

}

class Book {

private State state;

public Book(State state) {

this.state = state;

}

public void setState(State state) {

this.state = state;

}

public void borrow() {

state.borrow(this);

}

public void returnBook() {

state.returnBook(this);

}

public void placeHold() {

state.placeHold(this);

}

public void reportLost() {

state.reportLost(this);

}

}

// Ví dụ triển khai một trạng thái

class AvailableState implements State {

public void borrow(Book book) {

System.out.println("Borrowing book...");

book.setState(new CheckedOutState());

}

public void returnBook(Book book) {

System.out.println("Book is already available.");

}

public void placeHold(Book book) {

System.out.println("Placing a hold on the book.");

book.setState(new OnHoldState());

}

public void reportLost(Book book) {

System.out.println("Reporting lost book.");

book.setState(new LostState());

}

}