**Họ và tên: Bùi Quang Minh**

**MSSV: 22664411**

**Bài 1:**

**1. Cách tăng Performance (Trade-off & Logic)**

Để tối ưu hệ thống thương mại điện tử, chúng ta không chỉ dùng "best practice" mà phải dựa trên sự đánh đổi:

* **Caching (Đánh đổi tính nhất quán lấy tốc độ):** Thay vì truy vấn trực tiếp vào Database cho mỗi lượt xem sản phẩm, ta lưu vào bộ nhớ đệm.
* **Asynchronous Processing (Đánh đổi sự phức tạp lấy khả năng phản hồi):** Các tác vụ như gửi email xác nhận đơn hàng, cập nhật điểm thành viên sẽ được đẩy vào hàng đợi xử lý sau.
* **Database Optimization:** Sử dụng **Indexing** cho các cột tìm kiếm (tên sản phẩm, danh mục) và **Read Replicas** để giảm tải cho node ghi chính.

**2. Công nghệ phục vụ Performance**

* **Redis:** Dùng để cache thông tin Session người dùng, giỏ hàng và danh sách sản phẩm hot. Đây là chìa khóa để đạt tốc độ phản hồi tính bằng milisecond.
* **Message Queue (RabbitMQ/Kafka):** Xử lý luồng đặt hàng. Khi khách nhấn "Mua", hệ thống chỉ cần ghi nhận vào Queue rồi báo thành công ngay, việc xử lý kho bãi diễn ra ngầm phía sau.
* **Elasticsearch:** Thay thế tìm kiếm SQL thông thường để khách hàng có thể search sản phẩm cực nhanh với tính năng gợi ý (autocomplete).
* **CDN (Cloudflare):** Lưu trữ hình ảnh sản phẩm tại các server gần người dùng nhất để giảm tải băng thông cho server chính.

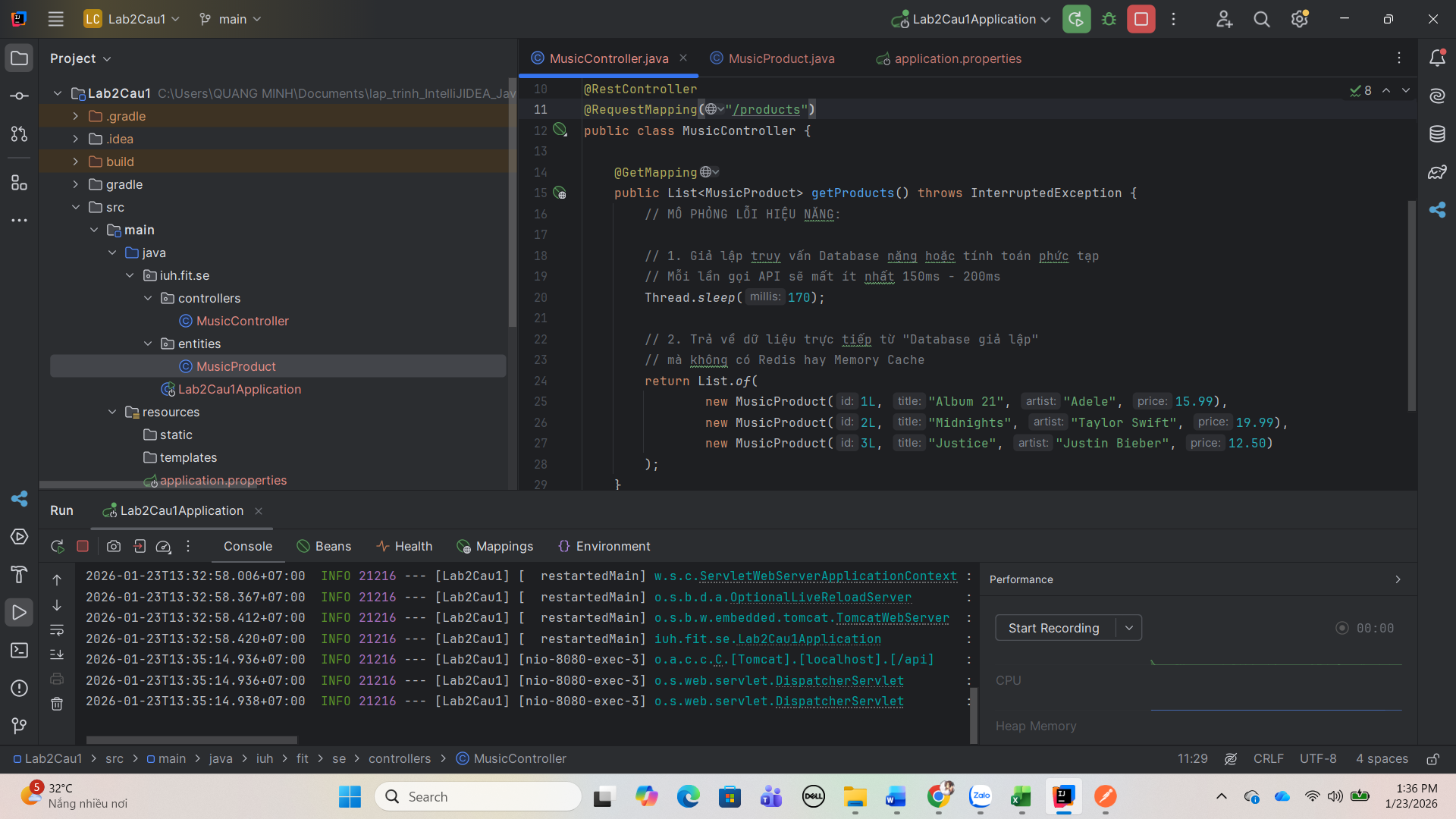
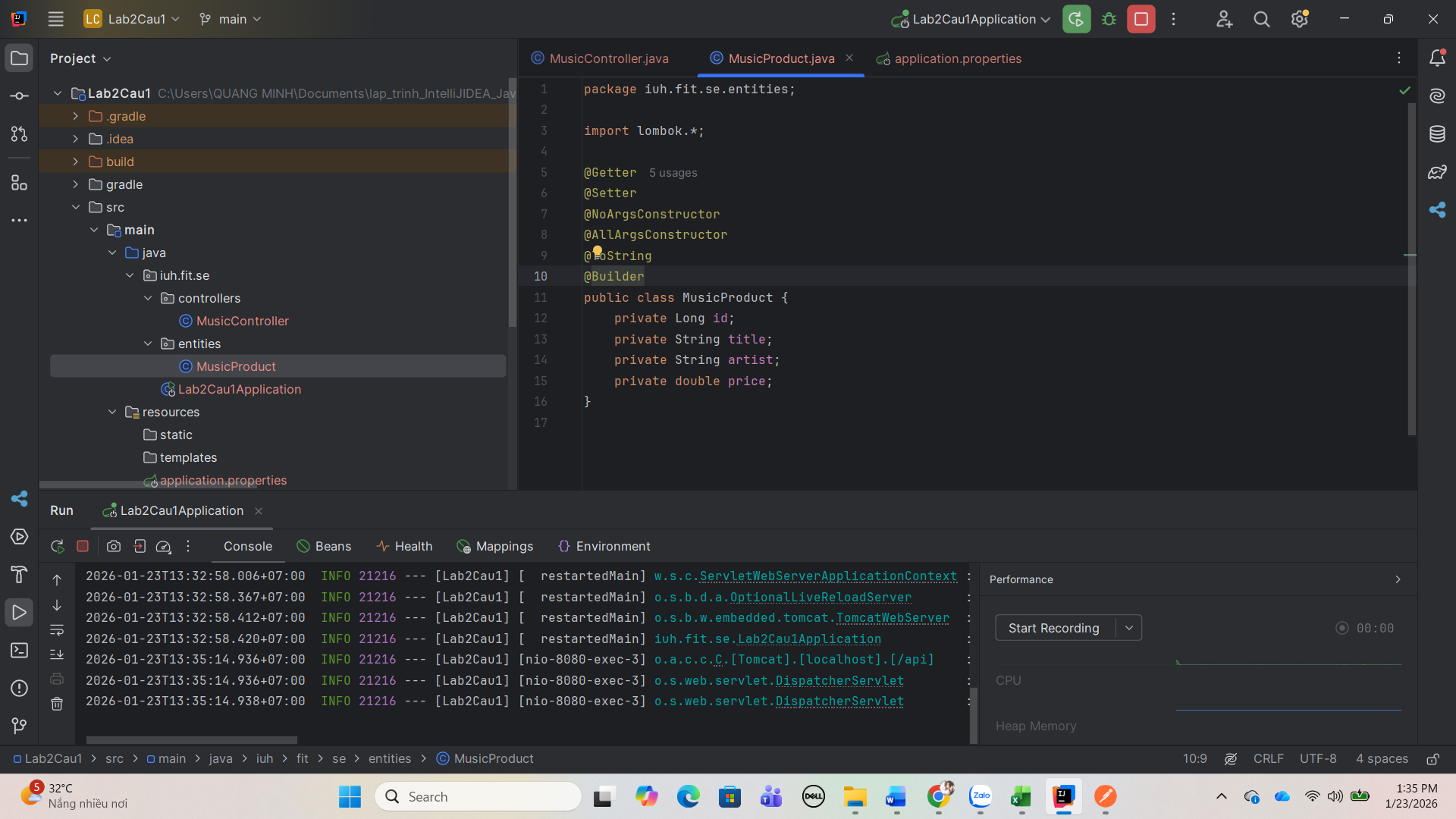
**3. Minh họa con số cụ thể (Scenario)**

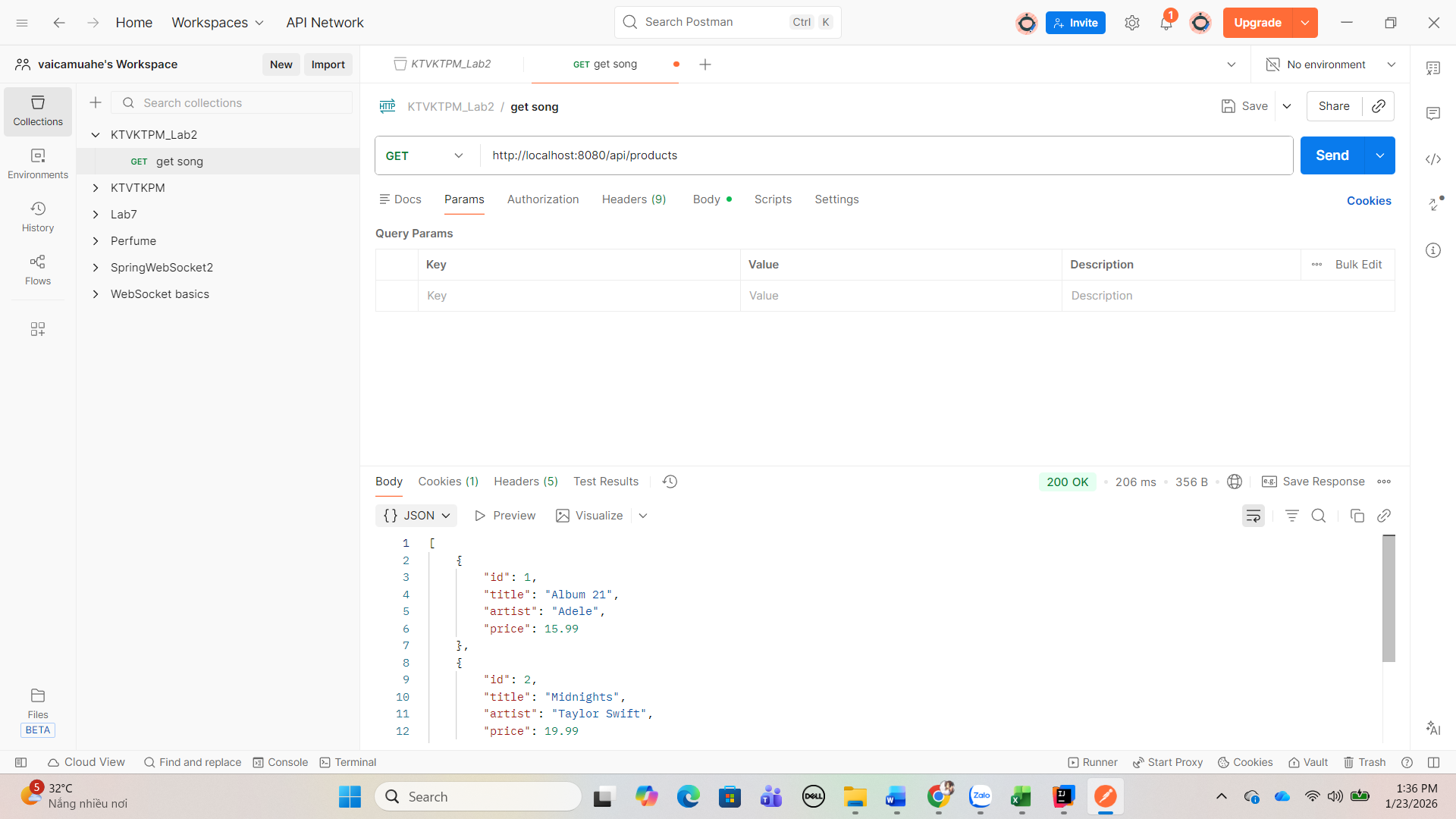
Đây là mục tiêu cụ thể dựa trên tư duy **Utility Tree** (ưu tiên kịch bản High priority):

* **Bài toán nhỏ:** Một request truy cập trang chi tiết sản phẩm.
* **Trước tối ưu (170ms):**
  + Server nhận request.
  + Truy vấn SQL tìm thông tin sản phẩm (mất ~100ms do bảng dữ liệu lớn).
  + Truy vấn SQL lấy danh sách bình luận/đánh giá (mất ~50ms).
  + Xử lý logic và render giao diện (~20ms).
* **Sau tối ưu (7ms):**
  + Server nhận request.
  + Kiểm tra **Redis**: Lấy toàn bộ thông tin sản phẩm đã được cache sẵn (chỉ mất ~2ms).
  + Các thành phần không quan trọng (như gợi ý sản phẩm liên quan) được load **Async** sau khi trang đã hiển thị.
  + Kết quả: Người dùng thấy trang web hiện ra tức thì.

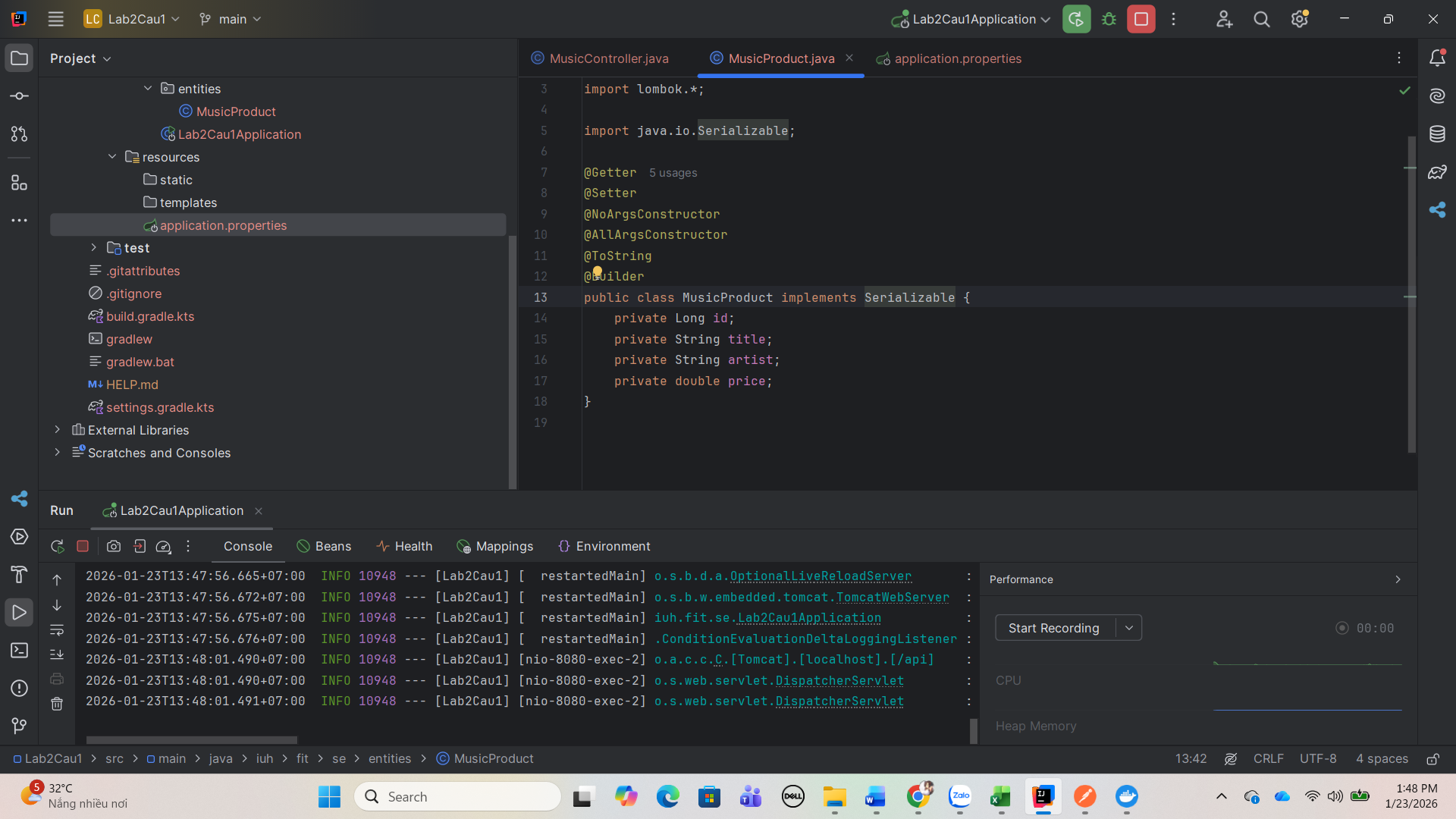
**VD:**

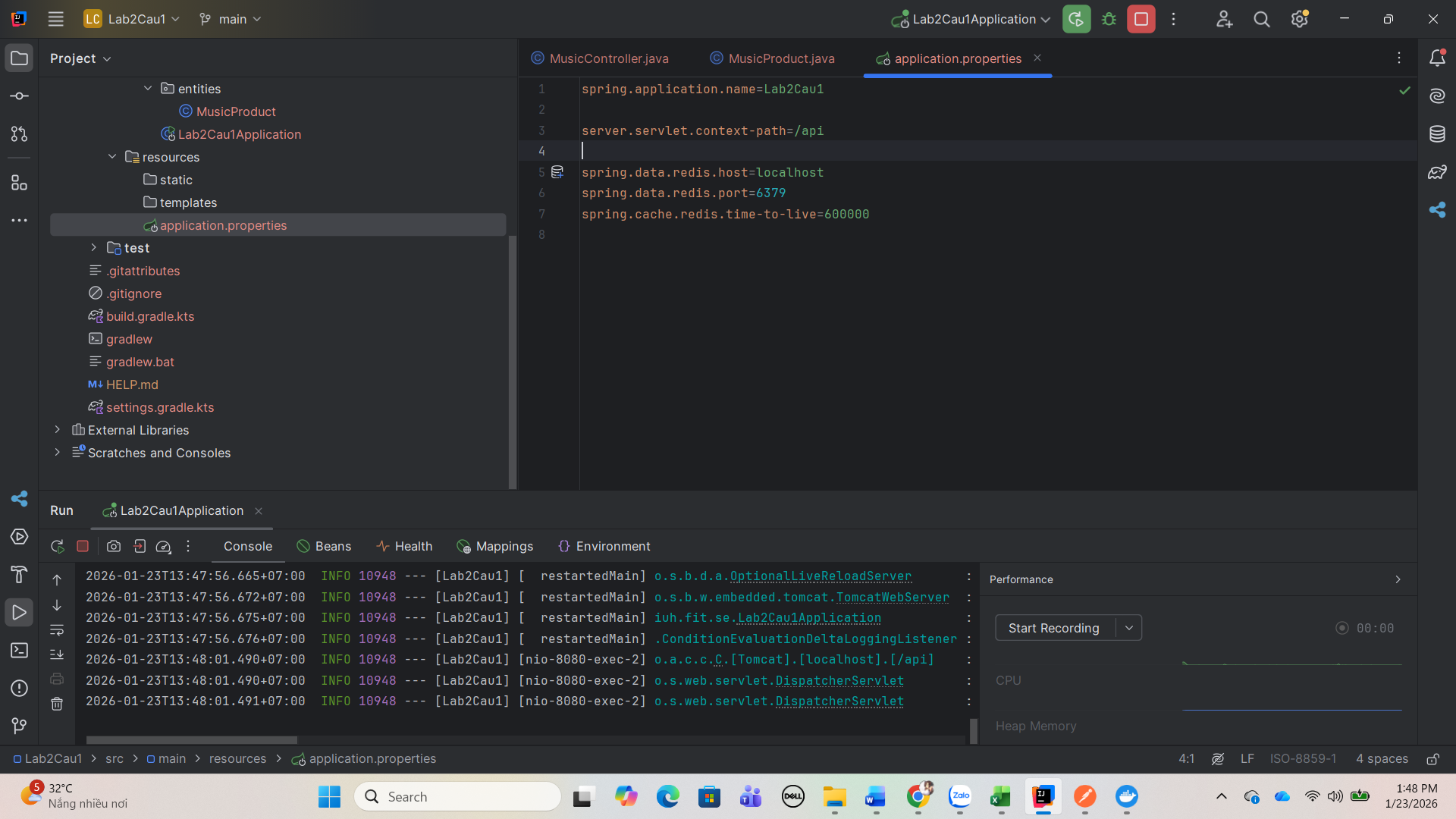
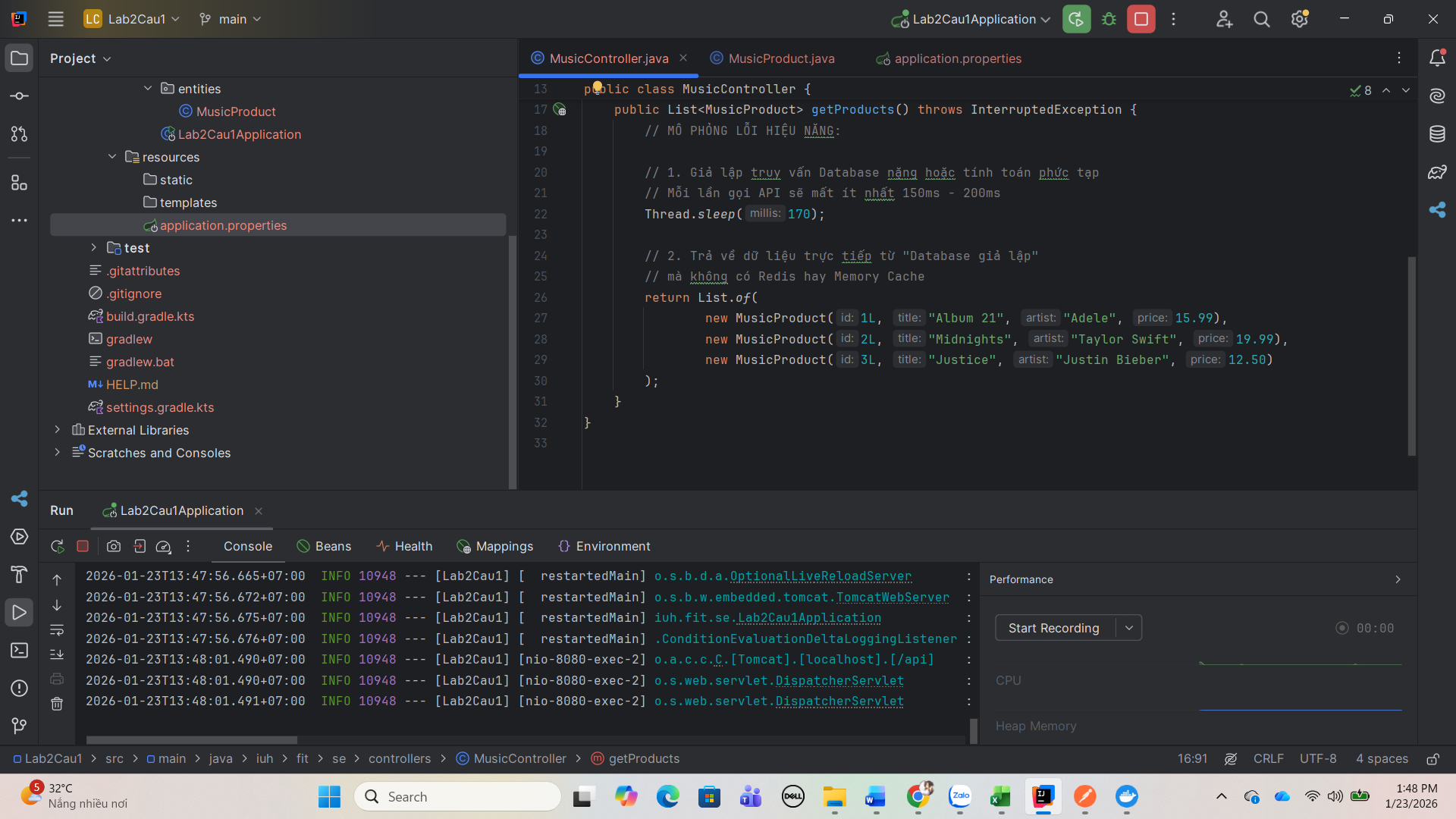
* **Trước khi được tối ưu:**

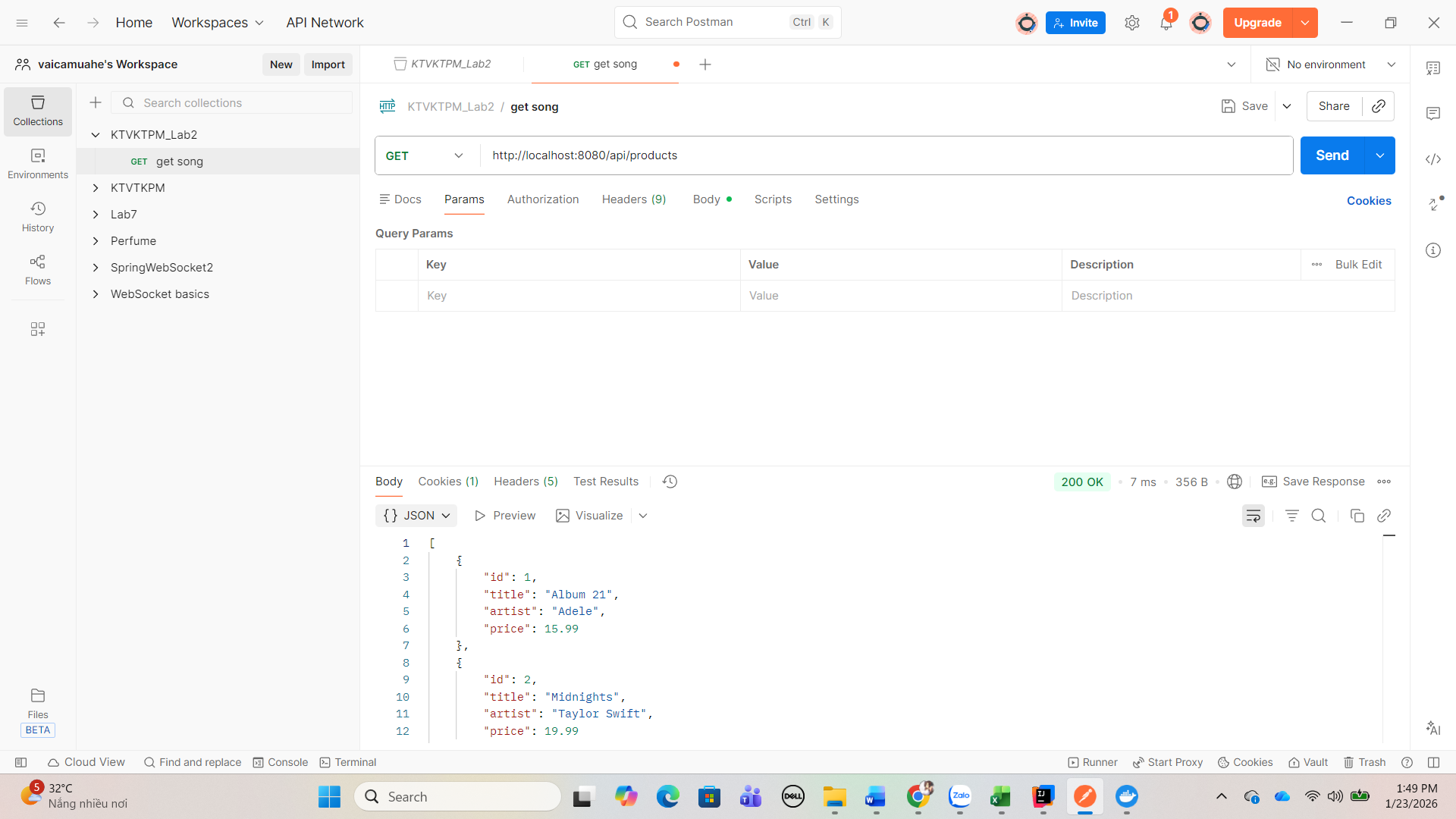
****

****

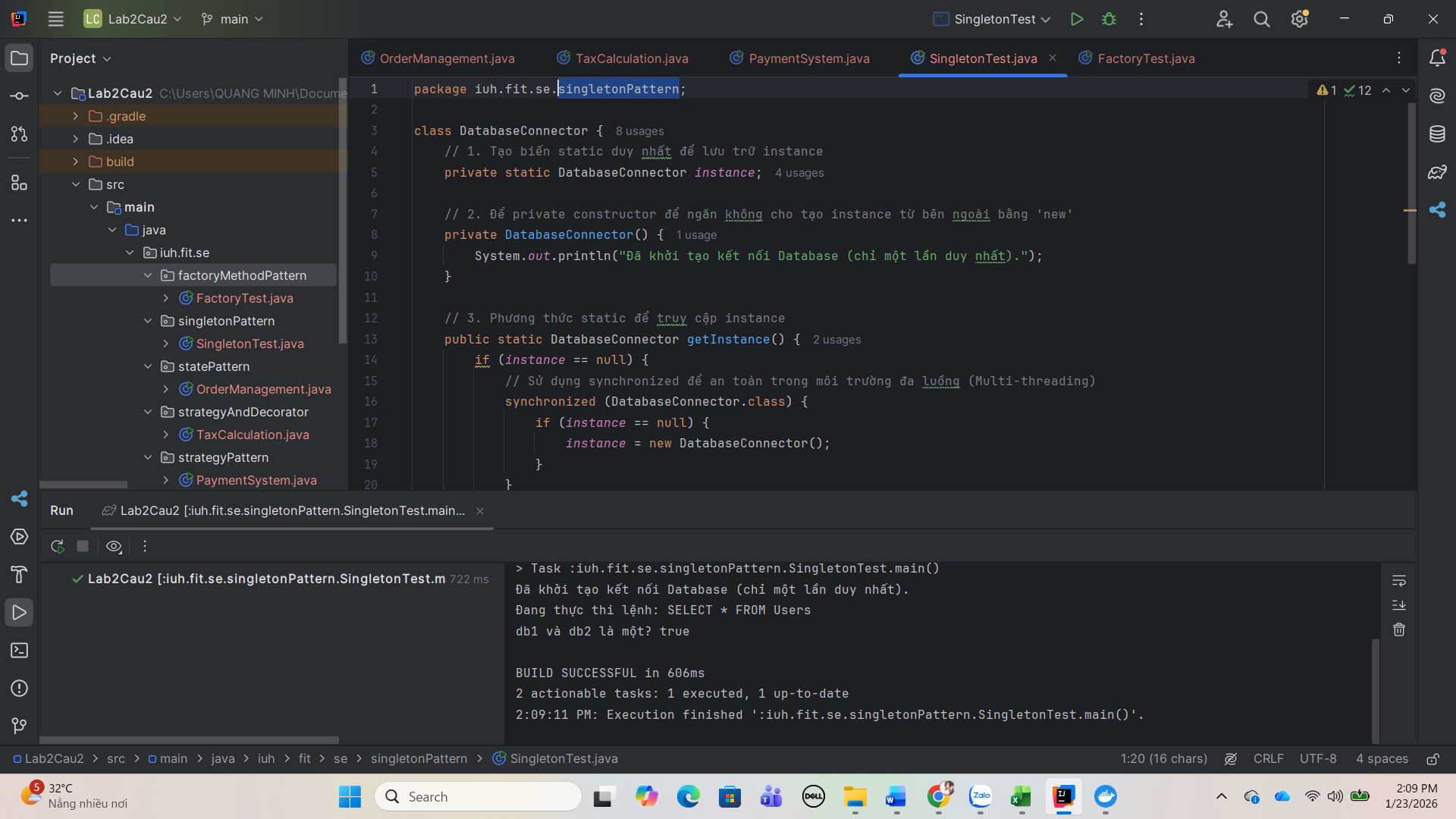
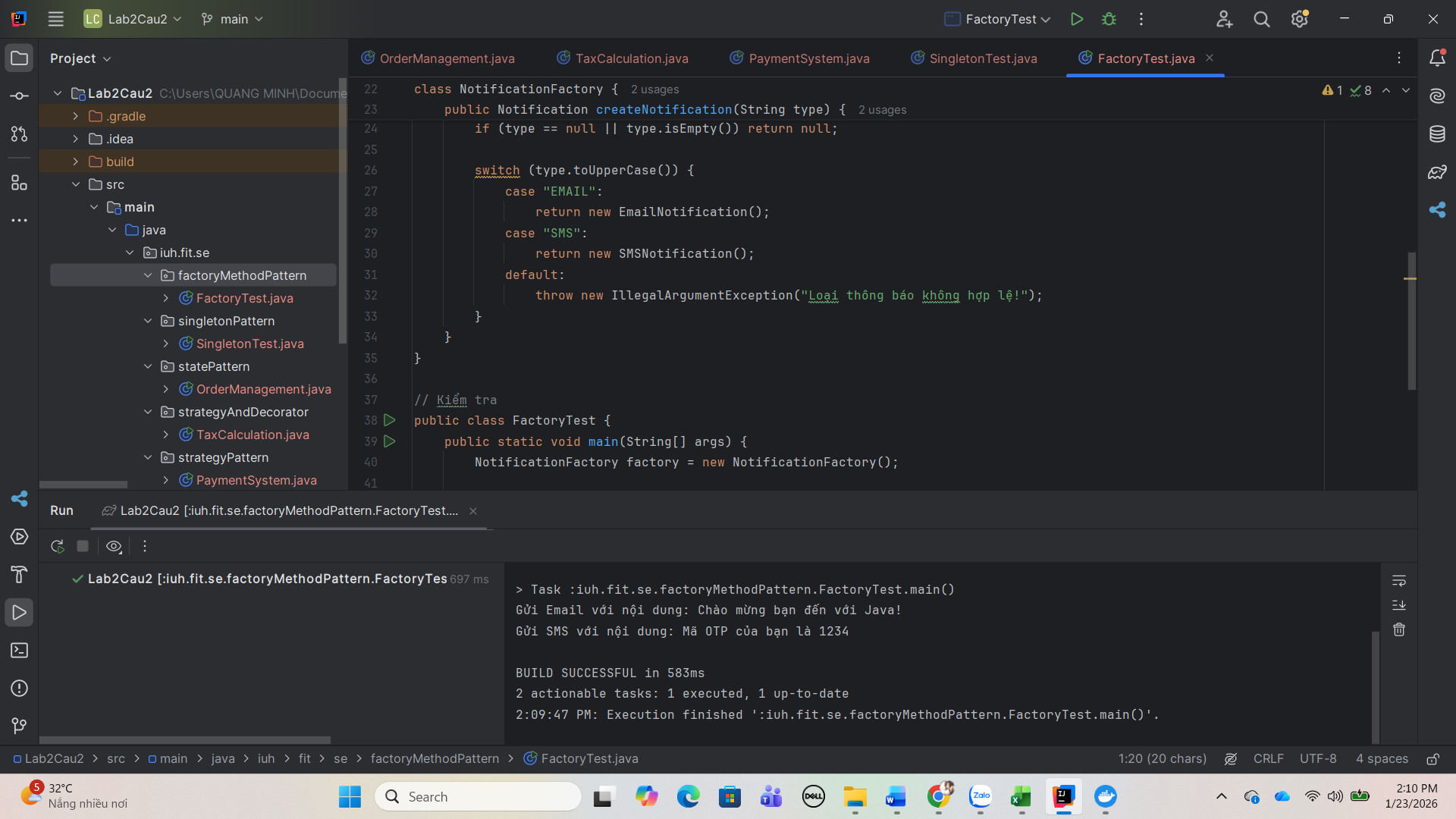
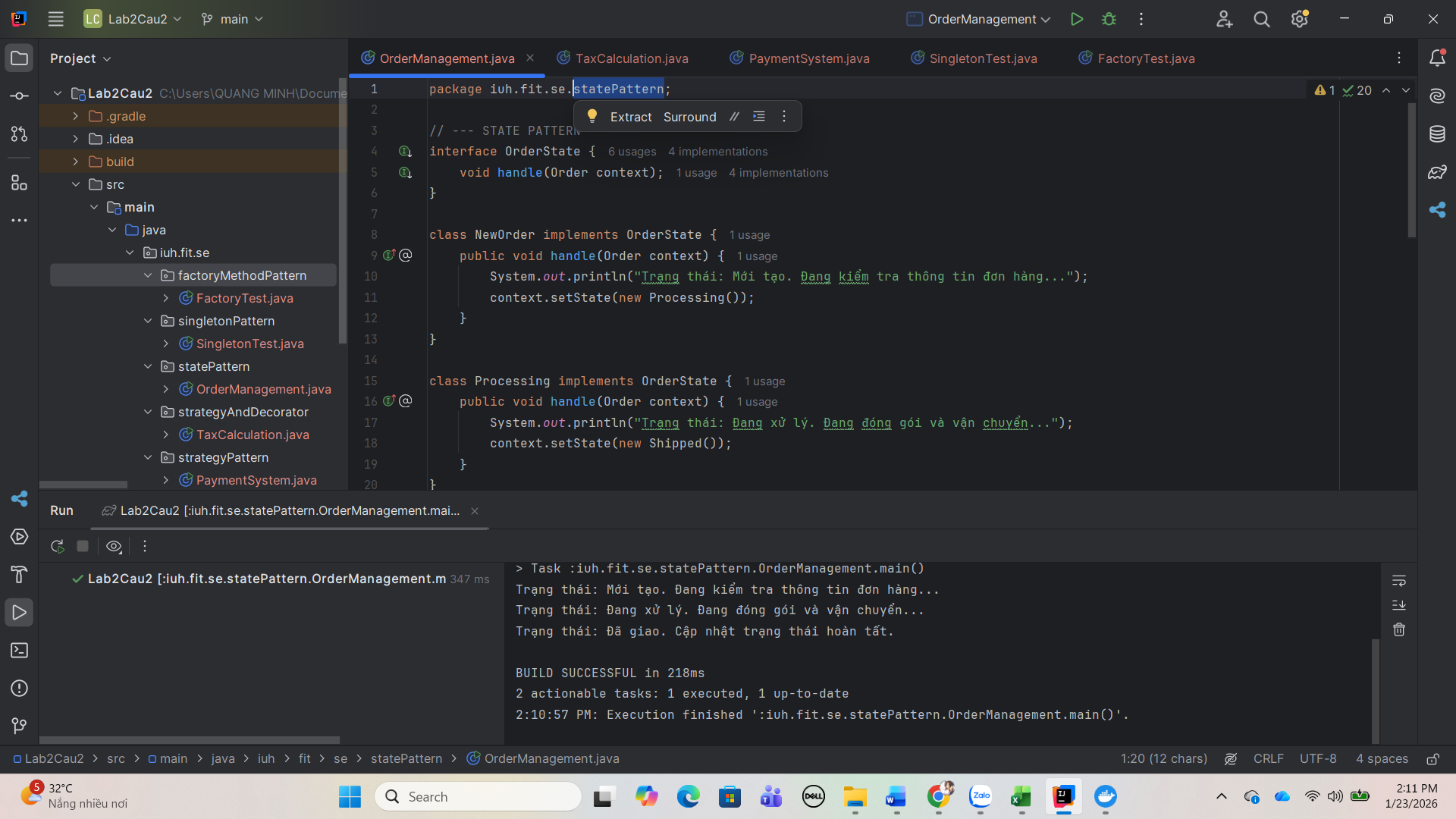
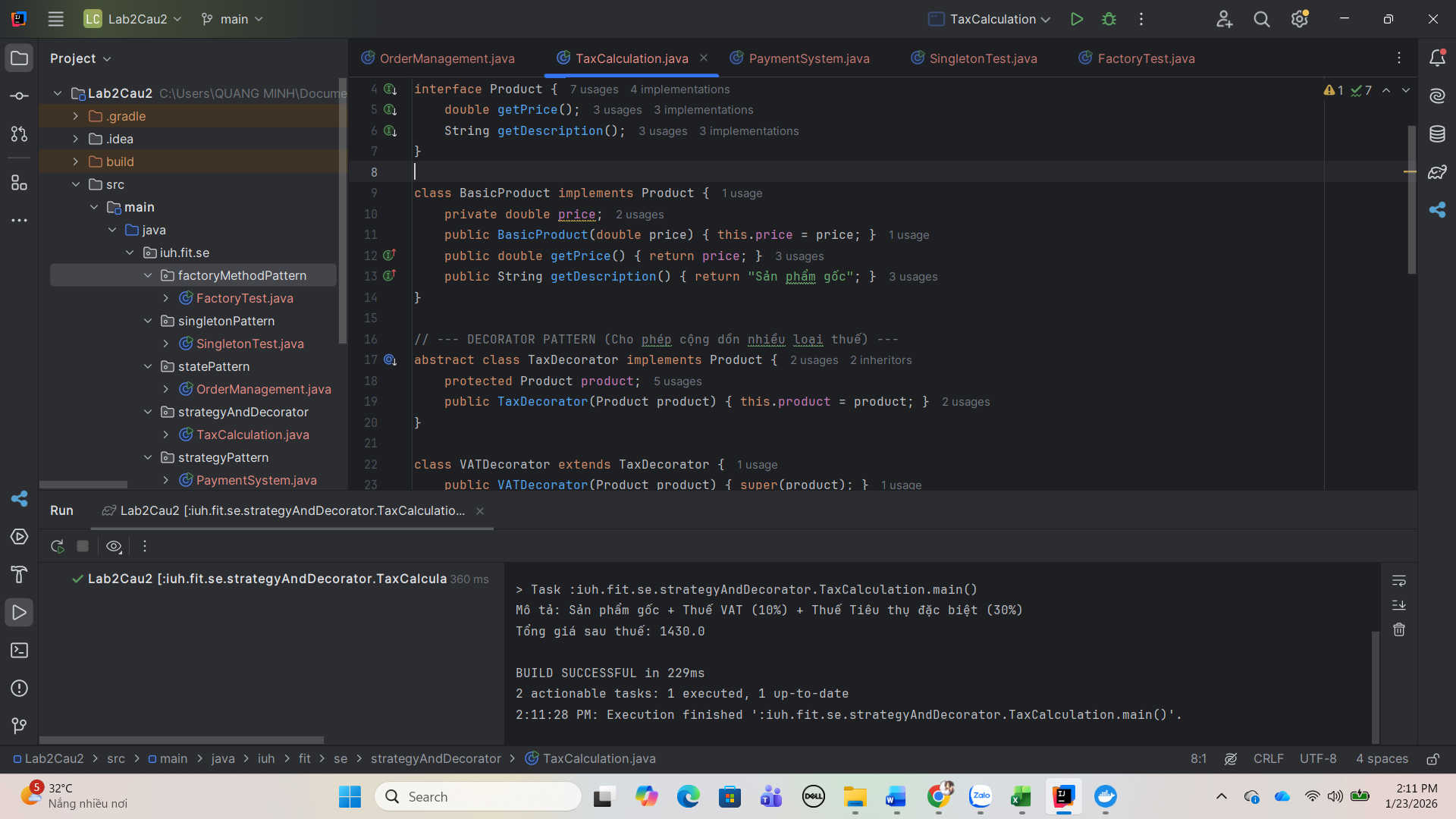
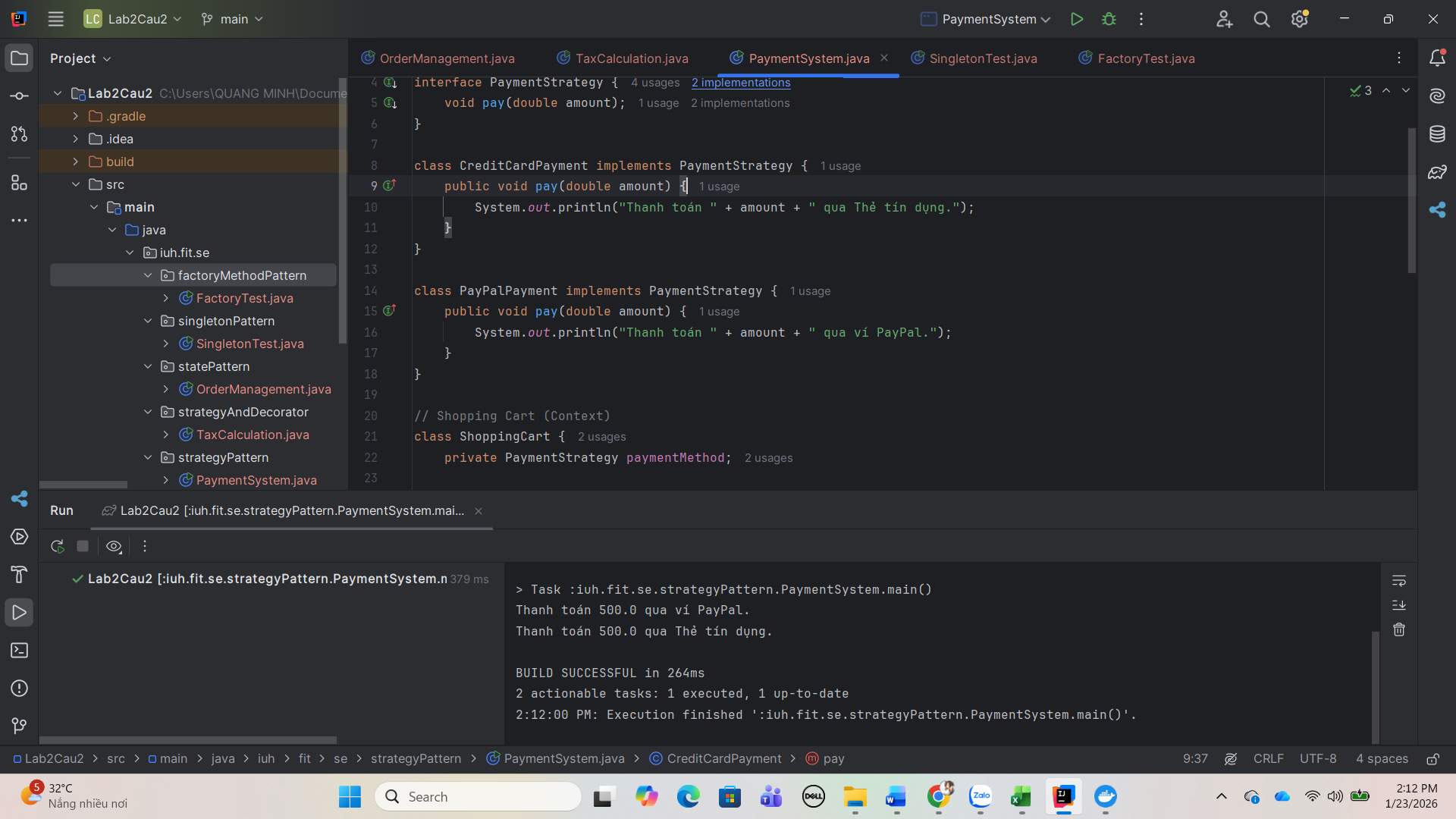
* **Sau khi được tối ưu:**

****

****

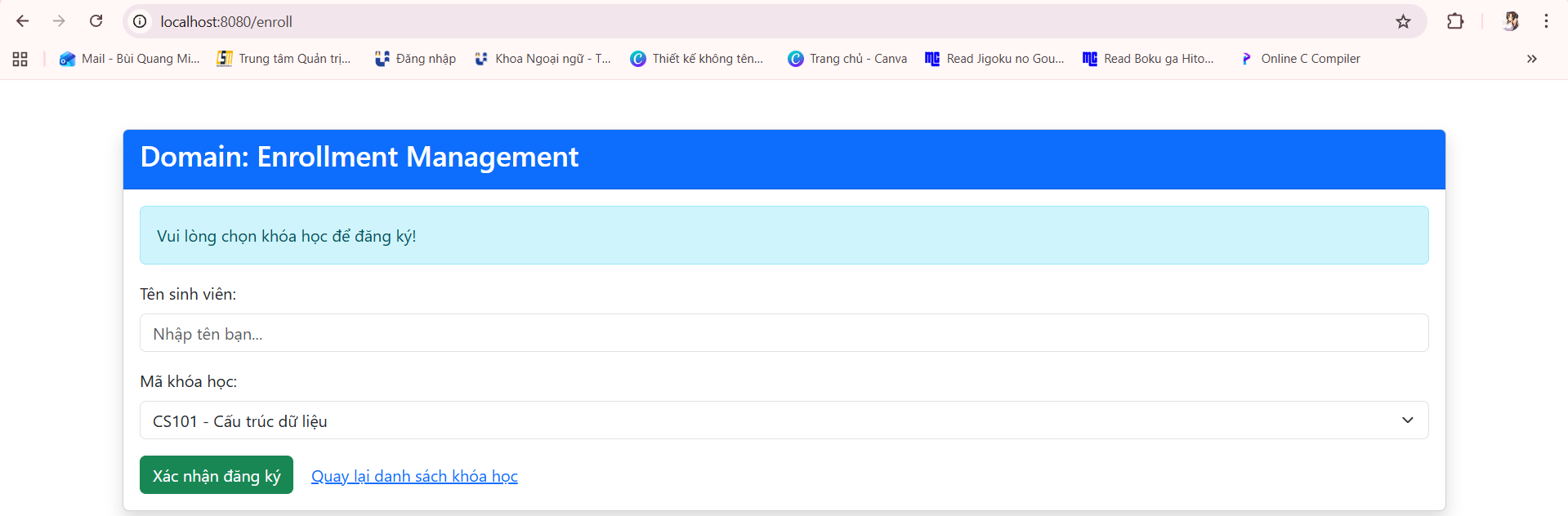
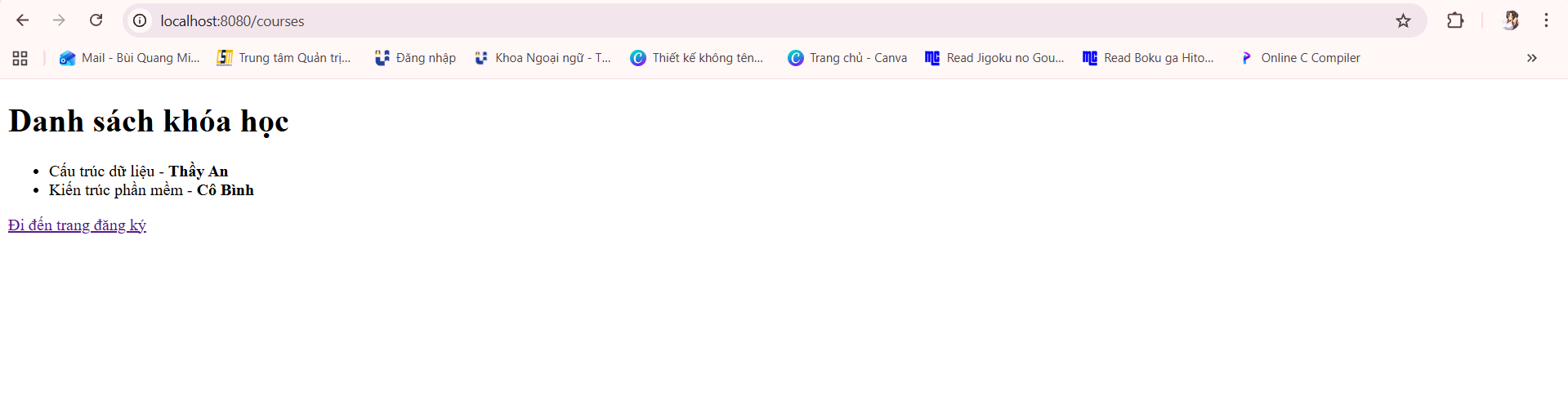
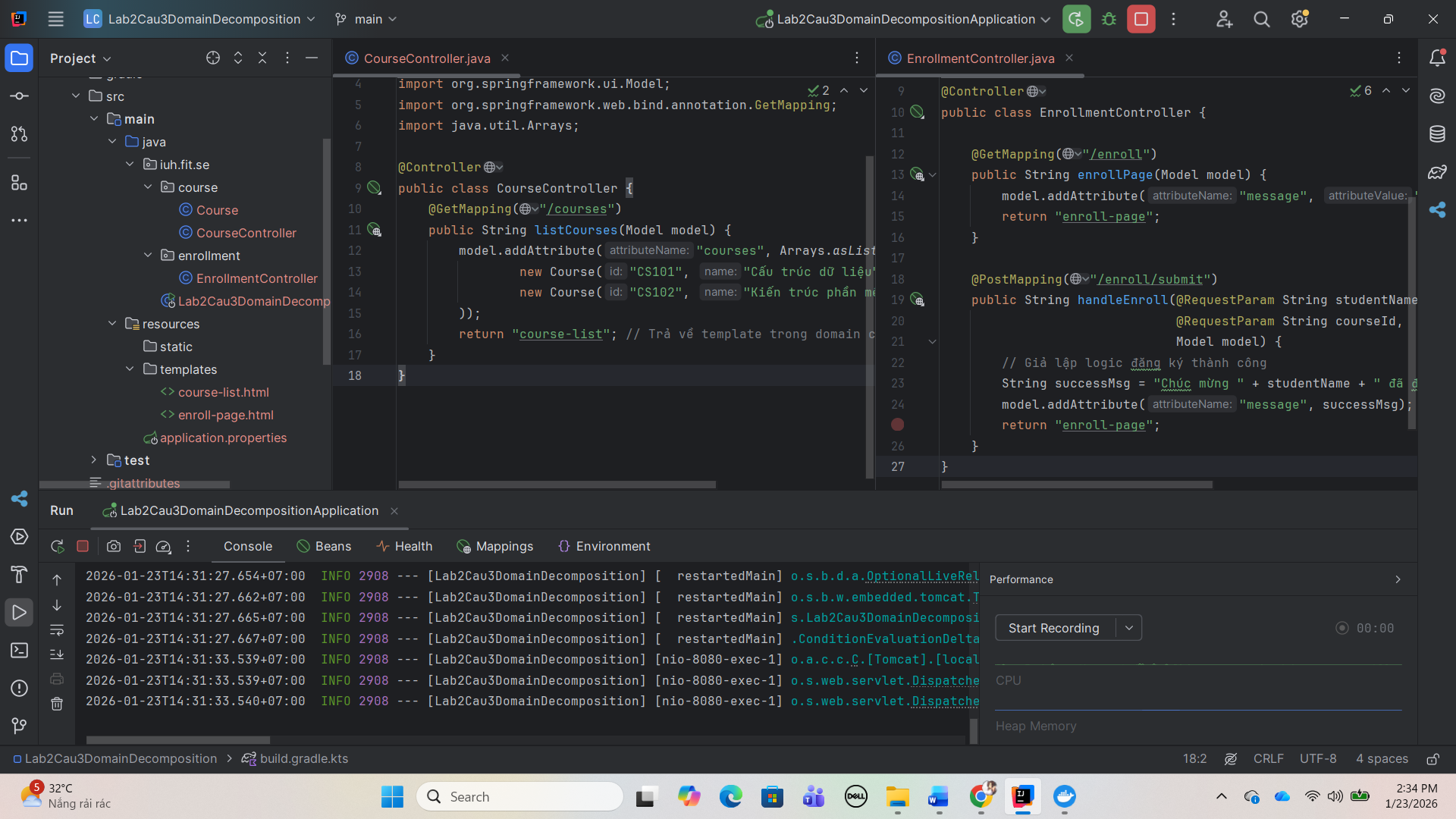
****

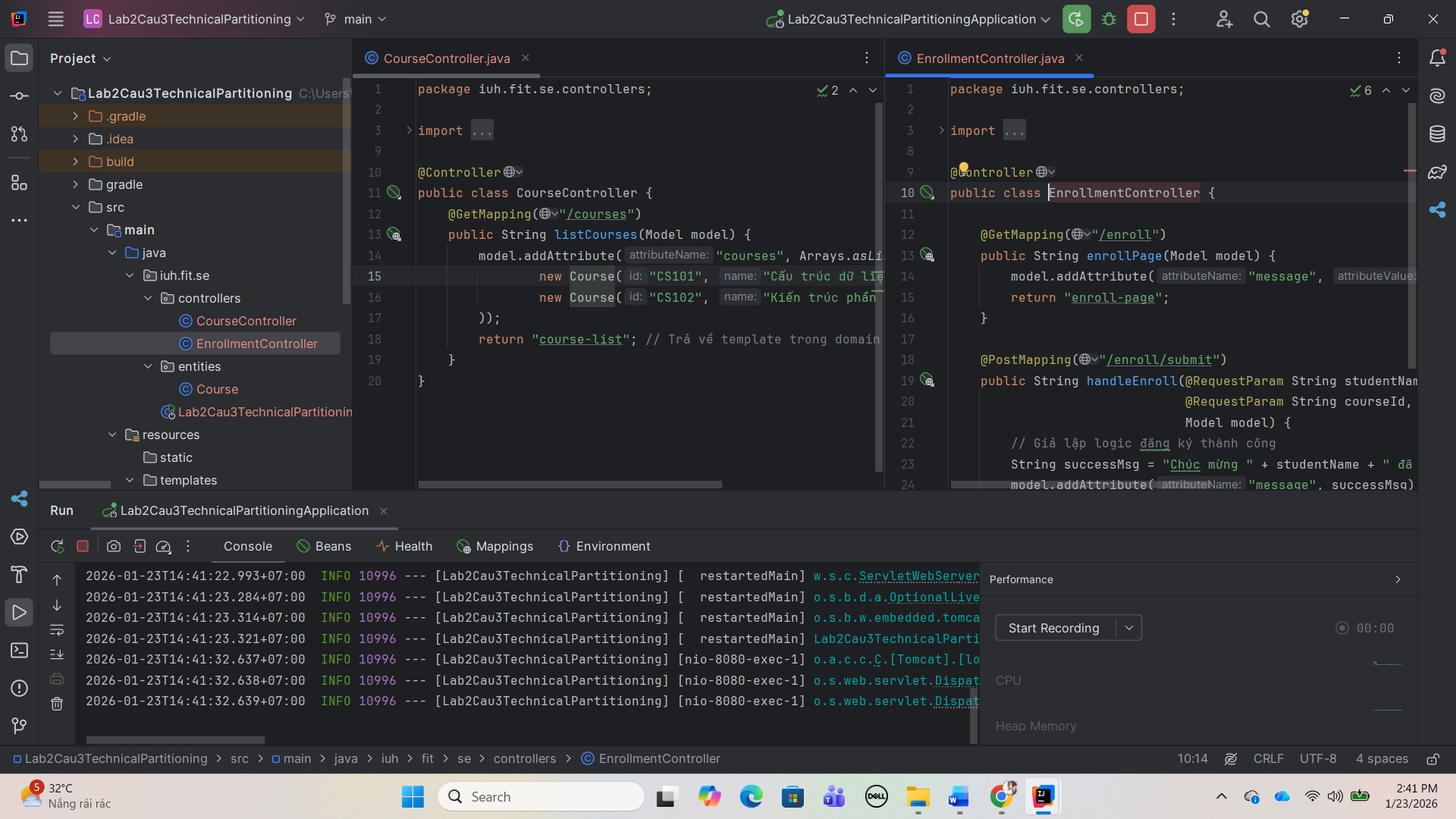
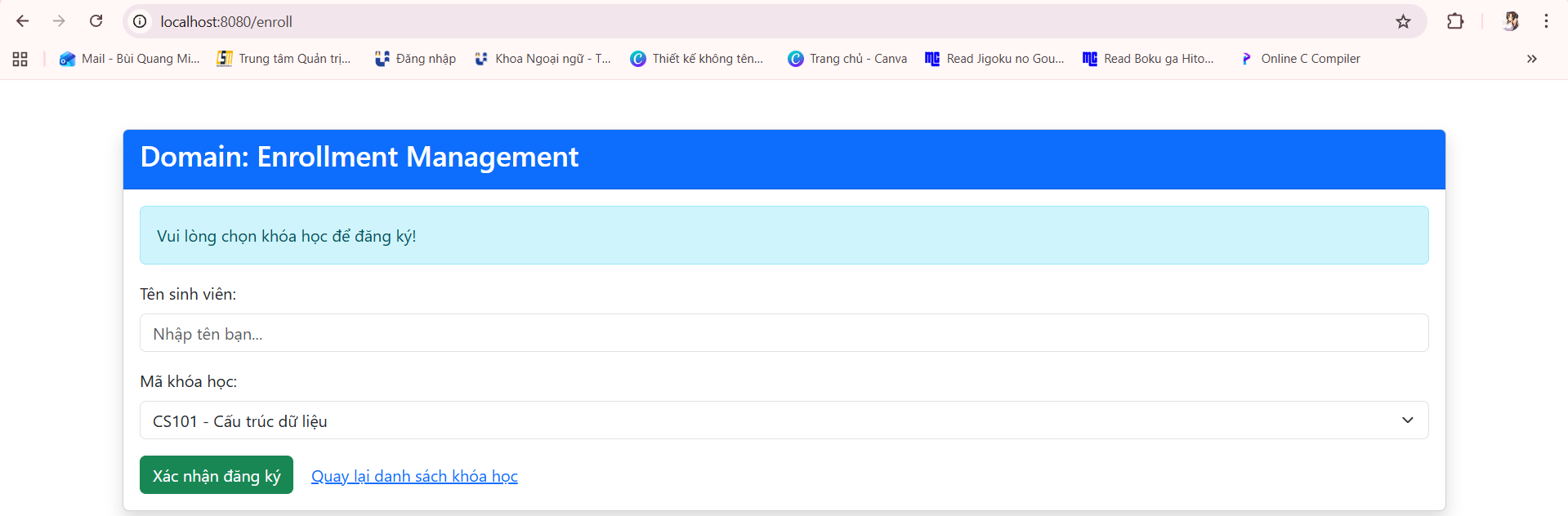
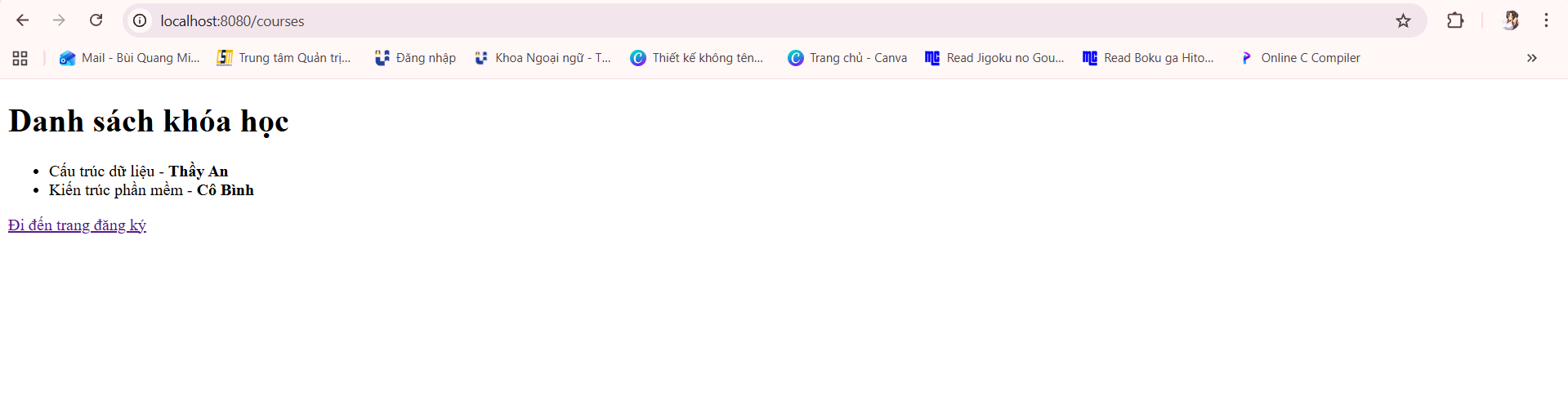
**Bài 2:**

* **Singleton Pattern**
* **Factory Method Pattern**
* **State Pattern**
* **Strategy And Decorator**
* **Strategy Pattern**

**Bài 3:**

* **Domain Decomposition:**

****

* **Technical Partitioning: **

**3. Giải thích tại sao hai cách cho kết quả khác nhau**

Dù cả hai cách phân chia đều giải quyết cùng một yêu cầu chức năng (xem khóa học và đăng ký), nhưng chúng tạo ra "hình hài" hệ thống khác nhau về mặt cấu trúc và quản lý:

* **Về cấu trúc mã nguồn:**
  + **Domain Decomposition:** Gom tất cả code liên quan đến một nghiệp vụ (ví dụ: Course) vào một chỗ. Điều này tạo ra sự **đóng gói (encapsulation)** cao. Nếu bạn xóa module Enrollment, module Course vẫn có thể tồn tại và chạy độc lập.
  + **Technical Partitioning:** Chia theo lớp (Layer). Code của một tính năng bị rải rác: Controller ở một nơi, Service ở một nơi, Repository ở một nơi khác. Để hiểu một tính năng, bạn phải "nhảy" qua nhiều thư mục khác nhau.
* **Về khả năng bảo trì và thay đổi (Modifiability):**
  + **Domain Decomposition:** Khi nghiệp vụ "Đăng ký" thay đổi quy trình, bạn chỉ cần tác động vào đúng module enrollment. Việc này giảm thiểu rủi ro làm hỏng các phần khác của hệ thống.
  + **Technical Partitioning:** Việc thay đổi thường đòi hỏi sửa đổi đồng thời ở nhiều lớp kỹ thuật, dễ dẫn đến sự phụ thuộc lẫn nhau (coupling) rất chặt chẽ.
* **Về khả năng mở rộng (Scalability):**
  + **Domain Decomposition:** Là tiền đề cho **Microservices**. Bạn có thể dễ dàng tách module Enrollment ra một server riêng nếu lượng người đăng ký quá lớn.
  + **Technical Partitioning:** Thường hướng tới kiến trúc **Monolithic** (khối thống nhất), khó mở rộng riêng lẻ từng thành phần nghiệp vụ.

**4. Đưa ra quyết định chọn kiến trúc mô-đun nào**

Đối với hệ thống "Course Management" (hoặc "E-commerce mini" như đề bài gốc), quyết định kiến trúc của mình là:

**Lựa chọn: Domain Decomposition**

**Lý do đưa ra quyết định (Justification):**

1. **Tính sẵn sàng cho tương lai:** Hệ thống quản lý giáo dục hoặc thương mại thường xuyên thay đổi nghiệp vụ. Việc chia theo Domain giúp chúng ta phản ứng nhanh với các thay đổi này mà không gây ảnh hưởng dây chuyền (Low Coupling).
2. **Hỗ trợ làm việc nhóm:** Trong thực tế, mỗi nhóm phát triển có thể đảm nhận một Domain riêng biệt, giúp giảm thiểu xung đột code (conflict) khi làm việc chung trên một project lớn.
3. **Tối ưu hóa hiệu năng theo nhu cầu:** Như bài toán Performance trước đó, chúng ta có thể chỉ áp dụng **Redis Caching** cho Domain Course (vì dữ liệu ít thay đổi) và áp dụng **Message Queue** cho Domain Enrollment (vì cần xử lý giao dịch) một cách độc lập.

**Đánh đổi (Trade-off):** Chúng ta chấp nhận một chút **trùng lặp code** (ví dụ: cả hai domain đều phải định nghĩa lại đối tượng người dùng hoặc cấu hình chung) để đổi lấy sự **linh hoạt** và **khả năng mở rộng** cực cao.