1. Giới thiệu về Microservices và vấn đề đặt ra

Trong các bài giảng trước, chúng ta đã thảo luận chi tiết về kiến trúc Microservices, sự khác biệt của nó so với mô hình Monolithic và SOA. Sau khi hiểu rõ những lợi ích mà Microservices mang lại, câu hỏi quan trọng tiếp theo là làm thế nào để xây dựng một hệ thống Microservices trong thực tế.

2. Những thách thức khi phát triển Microservices

Việc phát triển Microservices yêu cầu lập trình viên sử dụng các ngôn ngữ backend như Java. Tuy nhiên, quá trình này không đơn giản vì phải đối mặt với nhiều thách thức. Trong khóa học này, chúng ta sẽ đi qua từng thách thức khi xây dựng Microservices, đồng thời đề xuất các giải pháp và tiêu chuẩn tốt nhất để khắc phục những khó khăn này.

3. Vấn đề triển khai Microservices theo phương pháp truyền thống

Thách thức đầu tiên chính là phương pháp tổ chức và triển khai Microservices. Trong các ứng dụng web truyền thống theo mô hình Monolithic, toàn bộ mã nguồn được phát triển bằng các lớp và phương thức Java, sau đó đóng gói thành các tệp WAR hoặc EAR để triển khai lên các máy chủ web như Tomcat. Quá trình này tốn nhiều thời gian vì phải phát triển, đóng gói và triển khai ứng dụng một cách thủ công. Khi các tổ chức lớn xây dựng hàng trăm hoặc hàng nghìn Microservices, cách tiếp cận truyền thống này trở nên không khả thi.

4. Giải pháp: Spring Boot Framework

Giải pháp để khắc phục vấn đề trên chính là sử dụng **Spring Boot Framework**. Công nghệ này giúp đơn giản hóa việc phát triển, đóng gói và triển khai Microservices một cách nhanh chóng, hiệu quả. Trong khóa học này, chúng ta sẽ sử dụng Spring Boot để xây dựng Microservices và tìm hiểu cách framework này giải quyết các thách thức thường gặp.

5. Điều kiện tiên quyết trước khi học về Microservices với Spring Boot

Nếu bạn chưa nắm vững kiến thức cơ bản về Spring Framework như khái niệm về Bean, Autowiring, Spring AOP, Spring MVC, Spring Security hay Spring Data JPA, bạn nên tham gia một khóa học về Spring trước khi tiếp tục với khóa học này. Việc nắm vững nền tảng Spring Framework là rất quan trọng để có thể phát triển Microservices hiệu quả.

1. Giới Thiệu Về Spring Boot Trong Xây Dựng Microservices

Trong bài giảng trước, chúng ta đã thảo luận về những thách thức khi xây dựng microservices và lý do tại sao Spring Boot là framework tốt nhất cho việc xây dựng các microservices bằng Java.

Spring Boot là một framework được xây dựng trên nền Spring Framework, giúp đồng bộ việc phát triển và triển khai các ứng dụng web Java, bao gồm microservices một cách dễ dàng và nhanh chóng.

2. Các lợi ích chính của Spring Boot

2.1. Tăng năng suất cho developer

Spring Boot giúp tăng năng suất cho developer bằng cách cung cấp các công cụ tự động cấu hình, giảm bớt sự cần thiết phải cài đặt và tổ chức môi trường.

2.2. Khả năng để triển khai dễ dàng

Spring Boot cho phép xây dựng các tệp JAR tự chứa tất cả các thành phần cần thiết (đọc gọi là fat JAR hoặc uber JAR), loại bỏ nhu cầu sử dụng các môi trường web server bên ngoài như Tomcat hoặc JBoss.

2.3. Hỗ trợ Auto Configuration và Dependency Injection

Spring Boot cung cấp các cấu hình mặc định và có thể điều chỉnh dễ dàng bằng các tệp cấu hình. Dependency Injection giúp quản lý và sử dụng các dependency trong dự án dễ hơn.

2.4. Tích hợp các công cụ quản lý và giám sát

Spring Boot cung cấp các công cụ monitoring như Spring Boot Actuator, giúp theo dõi sức khỏe ứng dụng, logs, metric và các cấu hình bên ngoài.

2.5. Hỗ trợ tích hợp với Cloud

Spring Boot dễ dàng triển khai lên các nền tảng cloud như Kubernetes, AWS, GCP và Azure. Ngoài ra, nó còn dễ dàng container hoá bằng Docker.

3. Sự khác biệt giữa Spring Boot và mô hình truyền thống

Trong các hệ thống truyền thống, developer cần cài đặt và duy trì các web server như Tomcat riêng lẻ. Trong khi đó, Spring Boot bao gồm sẵn web server trong tệp JAR, giúp quá trình triển khai trở nên nhẹ nhàng hơn.

XÂY DỰNG DỊCH VỤ MICROservices Sử DỤNG REST API

1. Giới thiệu

Trong quá trình phát triển ứng dụng, việc xây dựng các dịch vụ microservices là một xu hướng quan trọng. Microservices giúp hệ thống linh hoạt hơn, dễ dàng mở rộng và bảo trì. Để triển khai microservices hiệu quả, REST API là một trong những giao thức phổ biến được sử dụng.

2. Microservices và REST API

Microservices là các dịch vụ nhỏ, độc lập, có thể giao tiếp với nhau thông qua API. Một trong những cách phổ biến để triển khai microservices là sử dụng REST API. REST API cho phép các ứng dụng bên ngoài hoặc giao diện người dùng tương tác với microservices một cách hiệu quả.

REST API hoạt động dựa trên giao thức HTTP và cho phép các hệ thống giao tiếp một cách nhẹ nhàng thông qua định dạng dữ liệu JSON. Điều này giúp giảm tải cho hệ thống so với các giao thức truyền thống như SOAP.

3. Giao tiếp đồng bộ (Synchronous Communication) và bất đồng bộ (Asynchronous Communication)

REST API hỗ trợ giao tiếp đồng bộ, trong đó ứng dụng gửi yêu cầu và chờ phản hồi trước khi tiếp tục xử lý tiếp theo. Mặc dù đây là cách tiếp cận phổ biến, nhưng không phải lúc nào cũng phù hợp. Trong một số trường hợp, giao tiếp bất đồng bộ sử dụng message queues và Kafka có thể giúp cải thiện hiệu suất hệ thống.

4. Các quy tắc tiêu chuẩn khi xây dựng REST API

4.1. Các phương thức HTTP cơ bản

Khi xây dựng REST API, cần tuân thủ các quy tắc sử dụng phương thức HTTP:

- **POST**: Được sử dụng để tạo mới dữ liệu.
- **GET**: Dùng để đọc dữ liệu.
- **PUT**: Cập nhật toàn bộ một bản ghi dữ liệu.
- **PATCH**: Cập nhật một phần dữ liệu.
- **DELETE**: Xóa dữ liệu khỏi hệ thống.

Việc tuân thủ đúng các phương thức giúp đảm bảo API hoạt động đúng với tiêu chuẩn và dễ dàng tích hợp với các hệ thống khác.

4.2. Kiểm tra đầu vào (Input Validation)

Các REST API có thể bị tấn công nếu không thực hiện kiểm tra dữ liệu đầu vào. Do đó, cần áp dụng các biện pháp kiểm tra như:

- Xác minh định dạng dữ liệu đầu vào.
- Giới hạn độ dài và kiểu dữ liệu.
- Sử dụng các thư viện kiểm tra dữ liệu đầu vào để tăng cường bảo mật.

4.3. Xử lý ngoại lệ (Exception Handling)

Việc xử lý ngoại lệ là một phần quan trọng của REST API để đảm bảo rằng hệ thống không bị lỗi không mong muốn. Các ngoại lệ có thể bao gồm:

- Lỗi thời gian chạy (Runtime Exception).
- Lỗi nghiệp vụ (Business Exception).

Cần cung cấp thông báo lỗi rõ ràng để giúp người dùng hoặc hệ thống tích hợp dễ dàng xác định và khắc phục sự cố.

4.4. Tài liệu API (API Documentation)

Khi phát triển microservices, số lượng API có thể rất lớn, lên đến hàng trăm hoặc hàng nghìn endpoint. Do đó, việc tài liệu hóa API là rất cần thiết. Các công cụ như OpenAPI Specification và Swagger giúp tạo tài liệu API tự động, giúp các nhóm phát triển dễ dàng hiểu và tích hợp hệ thống.

XÂY DỰNG MICROSERVICES VỚi SPRING BOOT

1. Giới Thiệu

Trong buổi học này, chúng ta sẽ tiến hành thực hành và triển khai microservices sử dụng Spring Boot. Việc tạo các dịch vụ REST hoặc web application với Spring Boot sẽ trở nên dễ dàng nhờ các tính năng hỗ trợ mạnh mẽ.

2. Cài Đặt Spring Boot

Bước đầu tiên, ta truy cập trang web start.spring.io để lựa chọn các thông số dự án:

- **Ngôn ngữ:** Java
- **Công cụ build:** Maven
- **Phiên bản Spring Boot:** 3.1.1 (hoặc phiên bản mới nhất)

3. Cấu Hình Dự Án

- **Group:** com.eazybytes
- **Artifact:** accounts
- **Mô tả:** Microservice cho Accounts

- **Package Name:** com.eazybytes.accounts
- **Packaging:** Jar
- **Java Version:** 17 (phiên bản LTS mới nhất)

4. Thêm Dependencies

Dự án sẽ cài đặt các dependencies quan trọng:

- **Spring Web:** Hỗ trợ tạo REST API
- **H2 Database:** Cơ sở dữ liệu nhẹ, dùng tạm thời trước khi chuyển sang MySQL
- **Spring Data JPA:** Hỗ trợ quản lý dữ liệu
- **Spring Boot Actuator:** Giám sát và quản lý microservices
- **Spring Boot DevTools:** Tự động tái tải khi có thay đổi trong code
- **Lombok:** Giảm thiểu boilerplate code
- **Validation:** Hỗ trợ kiểm tra dữ liệu nhập vào

5. Khởi Tạo Dự Án

Sau khi cấu hình, ta nhấn **Generate** để tải về file ZIP chứa dự án Maven. Giải nén và mở trong IDE (IntelliJ IDEA hoặc Eclipse) để bắt đầu làm việc.

TẠO VÀ CẤU HÌNH REST API CƠ BẢN TRONG SPRING BOOT

1. Giới thiệu

Trong bài giảng trước, chúng ta đã tạo một ứng dụng web cơ bản sử dụng Spring Boot. Trong bài này, chúng ta sẽ cập nhật ứng dụng bằng cách thêm một REST API mới, trả về một phản hồi “Hello World”.

2. Tạo REST API Cơ Bản

2.1. Tạo Package và Controller

- Tạo package mới: com.eazybytes.accounts.controller.
- Bên trong package, tạo lớp AccountsController.
- Lớp này sẽ chứa các REST API liên quan đến microservice tài khoản.

2.2. Thêm Annotation

- Thêm @RestController trên lớp AccountsController để Spring Boot nhận diện các REST API.
- Viết một phương thức sayHello() trả về chuỗi "Hello World".
- Để định tuyến API, sử dụng @GetMapping("/sayHello").

3. Xây Dựng và Chạy Ứng Dụng

- Biên dịch và xây dựng project.
- Bật Annotation Processing trong IDE (Đề hỗ trợ Lombok).
- Chạy ứng dụng ở chế độ debug.
- Kiểm tra console, ứng dụng khởi động trên port 8080.
- Xem log tự động cấu hình của Spring Boot.

4. Kiểm Tra API

- Mở trình duyệt, truy cập http://localhost:8080/sayHello.
- Nhận phản hồi "Hello World".

5. Lợi Ích Của Spring Boot

- Tự động cấu hình hầu hết các thành phần quan trọng.
- Hỗ trợ hot reload, giúm cập nhật nhanh khi thay đổi mã nguồn.
- Hỗ trợ nhiều productivity tools như DevTools.

6. Cấu Hình IntelliJ IDEA

- Tùy chỉnh giao diện theo theme mong muốn.
- Cài đặt plugin và các theme như "Dark Purple Theme".
- Kích hoạt Annotation Processing để sử dụng Lombok.

CẤU HÌNH DATABASE TRONG SPRING BOOT SỬ DỤNG YAML

1. Giới thiệu Trong bài giảng trước, chúng ta đã xây dựng một REST API đơn giản để phản hồi "Hello World". Trước khi phát triển các API khác hỗ trợ thao tác CRUD, cần đảm bảo rằng mã liên quan đến cơ sở dữ liệu đã hoàn tất và sẵn sàng sử dụng. Hiện tại, chúng ta đang sử dụng cơ sở dữ liệu nội bộ H2.

2. Cấu hình cơ sở dữ liệu trong Spring Boot Spring Boot cho phép cấu hình các thuộc tính liên quan đến cơ sở dữ liệu thông qua tệp cấu hình. Mặc định, các thuộc tính này được đặt trong application.properties. Tuy nhiên, thay vì sử dụng .properties, chúng ta sẽ chuyển sang định dạng YAML (application.yml). YAML là một định dạng phổ biến, dễ đọc và thường được sử dụng trong Docker, Kubernetes, AWS, GCP và Azure.

3. Lợi ích của YAML so với Properties

- YAML cho phép tổ chức dữ liệu có cấu trúc hơn bằng cách sử dụng thụt dòng thay vì lặp lại khóa.
- Dễ đọc và trực quan hơn so với định dạng key=value của .properties.
- Được sử dụng rộng rãi trong các nền tảng đám mây và công nghệ container hóa.

4. Cấu trúc YAML và cách viết cấu hình Dưới đây là cách chuyển đổi từ application.properties sang application.yml:

Tệp application.properties (cũ):

```
server.port=8080
spring.datasource.url=jdbc:h2:mem:testdb
```

```
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.h2.console.enabled=true
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Tệp application.yml (mới):

```
server:
  port: 8080

spring:
  datasource:
    url: jdbc:h2:mem:testdb
    driverClassName: org.h2.Driver
    username: sa
    password: ""
  h2:
    console:
      enabled: true
  jpa:
    database-platform: org.hibernate.dialect.H2Dialect
    hibernate:
      ddl-auto: update
    show-sql: true
```

5. Giải thích chi tiết từng thuộc tính

- server.port: Định nghĩa cổng chạy ứng dụng, mặc định là 8080.
- spring.datasource.url: Đường dẫn kết nối đến H2 database.

- spring.datasource.driverClassName: Xác định driver kết nối đến H2 database.
- spring.datasource.username & spring.datasource.password: Thông tin đăng nhập vào database.
- spring.h2.console.enabled: Cho phép truy cập H2 console từ trình duyệt.
- spring.jpa.database-platform: Chỉ định Hibernate sử dụng dialect nào để giao tiếp với H2.
- spring.jpa.hibernate.ddl-auto: Tự động tạo hoặc cập nhật bảng khi ứng dụng khởi động.
- spring.jpa.show-sql: Hiển thị các câu lệnh SQL trong quá trình thực thi.

Xây dựng cơ sở dữ liệu và tích hợp Spring Data JPA

1. Giới thiệu

Spring Data JPA là một phần mở rộng của Spring Framework, giúp đơn giản hóa việc truy cập và thao tác dữ liệu trong các ứng dụng Java thông qua JPA (Java Persistence API). Nó cung cấp một cách tiếp cận mạnh mẽ để làm việc với cơ sở dữ liệu mà không cần phải viết nhiều câu lệnh SQL thủ công.

2. Định nghĩa các Entity

2.1. Lớp Customer

Lớp Customer đại diện cho bảng customer trong cơ sở dữ liệu. Mỗi trường dữ liệu trong class này tương ứng với một cột trong bảng.

```
@Entity
@Table(name = "customer")
public class Customer {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;

    @Column(name = "name")
    private String name;

    @Column(name = "email")
    private String email;

    @Column(name = "mobile_number")
    private String mobileNumber;
}
```

2.2. Lớp Accounts

Tương tự, lớp Accounts đại diện cho bảng accounts. Một khách hàng có thể có nhiều tài khoản, vì vậy bảng này có khóa ngoại liên kết với bảng customer.

```
@Entity
@Table(name = "accounts")
public class Accounts {
    @Id
    private Long accountNumber;

    @Column(name = "customer_id")
    private Long customerId;

    @Column(name = "account_type")
    private String accountType;

    @Column(name = "branch_address")
```

```
    private String branchAddress;
}
```

3. Định nghĩa Repository

3.1. CustomerRepository

Đây là interface giúp truy xuất dữ liệu từ bảng customer.

```
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {
}
```

3.2. AccountsRepository

Tương tự, repository cho bảng accounts:

```
@Repository
public interface AccountsRepository extends JpaRepository<Accounts, Long> {
}
```

XÂY DỰNG CÁC LỚp DTO TRONG MICROSERVICE ACCOUNTS

1. Giới thiệu

Trong bài giảng này, chúng ta tạo các lớp Data Transfer Object (DTO) đại diện cho mỗi lớp entity trong microservice Accounts. DTO giúp tránh truyền dữ liệu trực tiếp từ các entity, bảo vệ tính toàn vẹ dữ liệu và giảm thiểu nguy cơ lộ đội tượng entity không cần thiết.

2. Xây dựng các lớp DTO

2.1. AccountsDto

Lớp AccountsDto đại diện cho entity Accounts và bao gồm các trường:

- accountNumber

- accountType
- branchAddress

Lý do loại bỏ: Trường customerId không cần truyền cho client do nó chỉ dùng nội bộ.

2.2. CustomerDto

Lớp CustomerDto đại diện cho entity Customer và bao gồm:

- name
- email
- mobileNumber

Lý do loại bỏ: customerId không cần truyền cho client.

2.3. ResponseDto

DTO này giúp gửi phản hồi từ server tới client khi thực hiện request:

- statusCode
- statusMsg

2.4. ErrorResponseDto

DTO này giúp xử lý khi gặp lỗi trong API:

- apiPath (API được gọi)
- errorCode (Mã lỗi HTTP)
- errorMsg (Nội dung lỗi)
- errorTime (Thời gian xảy ra lỗi)

3. Sử dụng Lombok trong DTOs

- Sử dụng @Data để tự động sinh getter, setter, equals(), hashCode(), toString().

- @AllArgsConstructor dùng cho ResponseDto và ErrorResponseDto nhằm sinh constructor với tất cả tham số.
- Tránh dùng @Data cho entity do ảnh hưởng đến Spring Data JPA.

4. Tính năng bổ sung

Khi tiến hành xa hơn, chúng ta sẽ xây dựng các DTO kết hợp Accounts và Customer, đồng thời tích hợp logic mapper để chuyển đổi giữa entity và DTO.

Xây dựng Business Logic cho Accounts Microservice

1. Giới thiệu

Trong bài giảng này, chúng ta tiến hành xây dựng một REST API trong Accounts Microservice, cho phép tạo mới tài khoản và thông tin khách hàng lưu trữ trong cơ sở dữ liệu H2. Bài giảng bao gồm nhiều tiêu chuẩn quan trọng như xử lý ngoại lệ, logic tầng dịch vụ, và mô hình DTO.

2. Tạo REST API Service

Chúng ta bắt đầu bằng việc tạo một API trong AccountsController. Trước khi thực hiện, API "sayHello" trước đó sẽ được xóa bỏ.

Bước 1: Thiết lập Request Mapping

- Sử dụng annotation @RequestMapping("/api") để đảm bảo tất cả API trong controller này có tiền tố "api".
- Trong các dự án thực tế, ta có thể thêm phiên bản API như /api/v1.

Bước 2: Xác định kiểu trả về

- Sử dụng @Produces(MediaType.APPLICATION_JSON_VALUE) để đảm bảo API sử dụng định dạng JSON.

Bước 3: Tạo API tạo tài khoản

- Tạo method createAccount() trong controller.
- Sử dụng @PostMapping("/create") để định nghĩa API POST nhận dữ liệu.
- Sử dụng @RequestBody CustomerDto customerDto để nhận dữ liệu từ client.

3. Trả về kết quả

- API trả về ResponseEntity<ResponseDto>
- Khi tài khoản được tạo, trả về HttpStatus.CREATED (201)
- Sử dụng AccountsConstants để lưu trữ các giá trị trả về

4. Xây dựng Tầng Service

Bước 1: Tạo Interface

- Interface IAccountsService định nghĩa các chức năng.
- Quy ước: "I" trong tên để định nghĩa interface.

Bước 2: Implement Service

- Tạo class AccountsServiceImpl implement IAccountsService.
- Viết logic tạo tài khoản và lưu trữ trong H2 Database.

Báo Cáo Triển Khai API Tạo Tài Khoản

Giới Thiệu

Báo cáo này tóm tắt quá trình triển khai API tạo tài khoản trong môi trường microservice, bao gồm xử lý exception, kiểm tra dữ liệu, và kết nối cơ sở dữ liệu H2.

1. Xử Lý Ngoại Lệ (Exception Handling)

Trong trường hợp một exception do lỗi logic doanh nghiệp xảy ra trong service, nó sẽ không truyền tới controller layer mà sẽ được xử lý bởi GlobalExceptionHandler. Kết quả

exception sẽ được trả về người dùng dưới dạng ErrorResponseDto. Trường hợp runtime exception do lỗi khác, các xử lý sẽ được bổ sung trong những bài học tiếp theo.

2. Quá Trình Triển Khai API

2.1. Xây Dựng API

- API tạo tài khoản được triển khai tại endpoint: POST localhost:8080/api/create
- Dữ liệu được gửi dưới dạng JSON bao gồm: name, email, mobileNumber.
- Dữ liệu nhập vào phải khớp với các trường trong CustomerData.

2.2. Kiểm Tra API Bằng Postman

- Sử dụng Postman để gửi request tạo tài khoản.
- Trường hợp request hợp lệ, API trả về mã trạng thái 201 Created.
- Trường hợp trùng số điện thoại, API trả về exception với mã lỗi 400 Bad Request.

2.3. Kiểm Tra Cơ Sở Dữ Liệu

- Kết nối với H2 Console để kiểm tra dữ liệu.
- Kiểm tra bảng customer và accounts để xác nhận dữ liệu được tạo thành công.

3. Cải Thiện API

3.1. Xử Lý Null Value

- Trước khi lưu dữ liệu, API cần gán giá trị cho createdAt và createdBy.
- Sử dụng LocalDateTime.now() để tạo thời gian hiện tại.
- Tạm thời gán createdBy là "anonymous".

3.2. Xây Dựng Thêm API CRUD

- Hiện tại, hệ thống chỉ có API tạo tài khoản.
- Cần phát triển thêm API **đọc, cập nhật và xóa** tài khoản.

- Tiếp theo, người dùng sẽ tự xây dựng logic cho các microservice Cards và Loans dựa trên tài liệu GitHub

XÂY DỰNG REST API CHO TRUY XUẤT THÔNG TIN KHÁCH HÀNG VÀ TÀI KHOẢN NGÂN HÀNG

1. Giới thiệu

Trong bài giảng này, chúng ta sẽ tạo một REST API để truy xuất thông tin khách hàng và tài khoản ngân hàng từ cơ sở dữ liệu. API này sẽ nhận số điện thoại của khách hàng làm tham số đầu vào và trả về toàn bộ thông tin khách hàng cũng như tài khoản ngân hàng tương ứng.

2. Tạo phương thức trong AccountsController

Trên **AccountsController**, chúng ta sẽ tạo một phương thức mới:

- Kiểu truy cập: public
- Kiểu trả về: CustomerDTO
- Tên phương thức: fetchAccountDetails()
- Tham số đầu vào: @RequestParam String mobileNumber
- Gán annotation @GetMapping("/fetch")

3. Tạo phương thức trong IAccountService và AccountServiceImpl

Trong **IAccountService**, chúng ta cần khai báo một phương thức abstract:

- CustomerDTO fetchAccount(String mobileNumber);

Trong **AccountServiceImpl**, chúng ta thực thi phương thức này với logic:

1. Truy vấn **customerRepository** bằng findByMobileNumber()
2. Nếu không tìm thấy khách hàng, ném **ResourceNotFoundException**
3. Nếu tìm thấy, truy vấn **accountRepository** bằng findByCustomerId()

4. Trả về **CustomerDTO** chứa thông tin khách hàng và tài khoản

4. Xử lý Ngoại lệ (Exception Handling)

Trong trường hợp không tìm thấy khách hàng, chúng ta sẽ tạo một exception mới **ResourceNotFoundException**:

- Chứa ba trường: resourceName, fieldName, fieldValue
- Sử dụng String.format() để tạo thông báo chi tiết
- Để quản lý exception, tạo phương thức handleResourceNotFoundException() trong **GlobalExceptionHandler**

5. Tạo Repository Methods

Trong **CustomerRepository**:

```
Optional<Customer> findByMobileNumber(String mobileNumber);
```

Trong **AccountRepository**:

```
List<Account> findByCustomerId(Long customerId);
```

Triển Khai API Cập Nhật Tài Khoản Trong Microservice

1. Giới Thiệu

Hiện tại, microservice quản lý tài khoản ("Accounts Microservice") đã cung cấp API REST hỗ trợ việc tạo và truy xuất thông tin tài khoản. Bước tiếp theo trong quá trình phát triển là xây dựng API REST hỗ trợ cập nhật thông tin tài khoản trong cơ sở dữ liệu.

2. Chức Năng Cập Nhật Tài Khoản

- API mới cho phép client gửi yêu cầu cập nhật thông tin tài khoản.
- Khi gọi API GET theo mobileNumber, hệ thống sẽ trả về toàn bộ thông tin tài khoản và khách hàng.

- Dữ liệu nhận được từ GET API có thể được sửa đổi và gửi lại qua PATCH API.
- Các trường được phép cập nhật: tên, email, số điện thoại, loại tài khoản, địa chỉ chi nhánh.
- Trường không thể thay đổi: account number (mã số tài khoản).

3. Triển Khai API Cập Nhật Tài Khoản

3.1. Interface IAccountService

- Tạo phương thức trụ abstract updateAccount(CustomerDto customerDto): boolean.

3.2. Service Implementation

- Override updateAccount trong AccountService.
- Truy xuất AccountsDto từ CustomerDto.
- Dò tìm tài khoản theo accountNumber trong database bằng findById() của Spring Data JPA.
- Kiểm tra tồn tại, nếu không, ném ResourceNotFoundException.
- Cập nhật thông tin và lưu vào database bằng save().
- Lưu cập nhật trong CustomerRepository theo customerId.

3.3. Controller AccountsController

- Tạo API endpoint /api/update nhận dữ liệu từ RequestBody.
- Gọi service updateAccount(CustomerDto).
- Trả về response:
    - HTTP 200 OK nếu cập nhật thành công.
    - HTTP 500 Internal Server Error nếu có lỗi.

4. Kiểm Thử API Cập Nhật

1. Tạo tài khoản mới.
2. Truy vấn thông tin bằng API GET.
3. Gửi yêu cầu cập nhật bằng API PUT với dữ liệu đã thay đổi.
4. Kiểm tra lại bằng API GET với số điện thoại mới, xác nhận thông tin đã được cập nhật.

XÂY DỰNG API XÓA Dữ LIỆU KHÁCH HÀNG TRONG ACCOUNT MICROSERVICE

1. Giới thiệu

Trong bài viết này, chúng ta sẽ tạo một API REST mới trong Account Microservice nhằm xóa thông tin khách hàng và tài khoản liên quan. API này sẽ nhận số điện thoại làm tham số đầu vào, dò tìm khách hàng tương ứng, sau đó xóa các bản ghi liên quan từ cả hai bảng Accounts và Customers.

2. Cách tiếp cận

2.1. Xác định khách hàng cần xóa

- Nhận tham số số điện thoại từ client.
- Dò tìm khách hàng trong CustomerRepository bằng findByMobileNumber().
- Nếu không tìm thấy, ném ResourceNotFoundException.

2.2. Xóa dữ liệu từ các bảng

- Lấy customerId từ Customer entity.
- Xóa bản ghi trong Customer và Accounts bằng các hàm deleteById() và deleteByCustomerId().
- Phương thức deleteById() trong CustomerRepository là một hàm do framework cung cấp.

- Phương thức deleteByCustomerId() trong AccountsRepository cần được định nghĩa bổ sung.

2.3. Sử dụng Spring Data JPA và giao dịch (Transaction Management)

- Sử dụng các annotation @Transactional và @Modifying để đảm bảo dữ liệu được xóa trong giao dịch an toàn.
- Nếu có lỗi xảy ra, giao dịch sẽ bị rollback, tránh trường hợp dữ liệu bị xóa dở chừ.

3. Cấu trúc API

3.1. Interface AccountService

- Thêm một hàm deleteAccount(String mobileNumber) trả về boolean.

3.2. Triển khai AccountServiceImpl

- Tìm Customer theo số điện thoại.
- Xóa khách hàng và tài khoản liên quan.
- Trả về true nếu xóa thành công.

3.3. API Controller

- Tạo hàm deleteAccountDetails(@RequestParam String mobileNumber)
- Sử dụng annotation @DeleteMapping("/api/delete").
- Trả về HTTP 200 OK nếu xóa thành công, nếu thất bại trả về HTTP 500 Internal Server Error.

4. Kiểm thử và kết quả

1. Tạo một tài khoản với API POST /api/createAccount.
2. Kiểm tra dữ liệu với API GET /api/getAccount.
3. Xóa dữ liệu bằng DELETE /api/delete?mobileNumber=1234567890.

4. Gọi API GET lại và kiểm tra lỗi 404 Not Found.
5. Gọi API DELETE lần nữa để đảm bảo xử lý khi dữ liệu đã bị xóa.

VẬN HÀNH VÀ Xử LÝ NGOẠI LỆ TRONG REST API

1. Giới thiệu

Trong hệ thống Microservice, chúng tôi đã xây dựng bốn REST API hỗ trợ các thao tác CRUD. Trong bài viết này, chúng tôi sẽ thực hiện việc xử lý các ngoại lệ runtime trong REST API để đảm bảo hệ thống hoạt động ổn định và cung cấp phản hồi chính xác đến client.

2. Xử lý ngoại lệ trong GlobalExceptionHandler

Hiện tại, trong **GlobalExceptionHandler**, chúng tôi chỉ đang xử lý hai ngoại lệ **ResourceNotFoundException** và **CustomerAlreadyExistsException**. Tuy nhiên, trong trường hợp xảy ra ngoại lệ runtime, cần phải có cơ chế xử lý chung để gửi phản hồi hợp lý đến client.

3. Cải tiến GlobalExceptionHandler

Chúng tôi đã bổ sung xử lý ngoại lệ chung trong **GlobalExceptionHandler** như sau:

- Tạo một method **handleGlobalException()**.
- Sử dụng **Exception.class** để xử lý tất cả các ngoại lệ (checked & unchecked exceptions).
- Trả về đối tượng **ErrorResponseDto**, chứa thông tin về đường dẫn API, mã lỗi **500 Internal Server Error**, thời gian xảy ra lỗi.

4. Kiểm tra runtime exception

Chúng tôi đã thực hiện kiểm tra bằng cách:

- Xóa **AllArgsConstructor** trong **AccountsController**, gây lỗi **NullPointerException** do **IAccountService** không được autowire.

- Thực thi API tạo tài khoản, dẫn đến runtime exception.
- Phản hồi nhận được chứa API path, mã lỗi **500 Internal Server Error** và chi tiết ngoại lệ.

Xử Lý Ngoại Lệ & Kiểm Tra Dữ Liệu Nhập Trong AccountsMicroservices

1. Xử Lý Ngoại Lệ Trong AccountsMicroservices

Trong AccountsMicroservices, chúng tôi đã tích hợp các cơ chế xử lý ngoại lệ runtime và ngoại lệ tùy chỉnh để đảm bảo tính ổn định và bảo mật của hệ thống.

2. Tiêu Chuẩn Kiểm Tra Dữ Liệu Nhập

Khi xây dựng microservices và REST APIs, việc kiểm tra dữ liệu nhập từ các ứng dụng khách hàng là rất quan trọng. Dưới đây là những trường hợp cần được xử lý:

- Số điện thoại khách hàng có độ dài khác với 10 chữ số.
- Email không đúng định dạng.
- Tên khách hàng ngắn hơn 3 ký tự.
- Khi gọi API GET với số điện thoại không hợp lệ, gây lãng phí truy vấn cơ sở dữ liệu.
- Các giá trị cập nhật (số tài khoản, loại tài khoản, địa chỉ chi nhánh) không hợp lệ.

Vì vậy, chúng ta cần đảm bảo việc kiểm tra dữ liệu nhập trước khi xử lý API.

3. Tích Hợp Kiểm Tra Dữ Liệu Trong Spring Boot

3.1. Thêm Dependency

Trước tiên, cần đảm bảo rằng dự án đã bao gồm dependency "spring-boot-starter-validation" trong pom.xml.

3.2. Cài Đặt Validation Trong DTOs

Tất cả dữ liệu nhập sẽ được biểu diễn bằng các Data Transfer Object (DTO). Chúng ta cài đặt validation như sau:

- **@NotEmpty**: Bắt buộc trường dữ liệu không được để trống.
- **@Size(min=5, max=30)**: Giới hạn độ dài của trường.
- **@Email**: Đảm bảo định dạng email hợp lệ.
- **@Pattern**: Dùng regex để xác thực dữ liệu (VD: số điện thoại 10 chữ số).

3.3. Cài Đặt Validation Trong Controller

Trong **AccountsController**, cần thêm:

- **@Validated**: Để bật validation trên toàn bộ controller.
- **@Valid**: Đảm bảo dữ liệu nhập vào từ client được kiểm tra trước khi xử lý.

Cải thiện Tính Năng Kiểm Tra Dữ Liệu và Xử Lý Ngoại Lệ Trong AccountsMicroservices

1. Giới thiệu

Trong các bảng của AccountsMicroservices, chúng tôi duy trì bốn cột siêu dữ liệu (metadata columns) gồm:

- CREATED_AT
- CREATED_BY
- UPDATED_AT
- UPDATED_BY

Những cột này giúp theo dõi ngày giờ tạo, người tạo, ngày giờ cập nhật và người cập nhật bản ghi. Hiện tại, chúng tôi đang cập nhật thông tin CREATED_AT và

CREATED_BY một cách thủ công. Tuy nhiên, có một lỗi chưa khắc phục được đó là UPDATED_AT và UPDATED_BY không được cập nhật khi thao tác update diễn ra.

2. Tự động Hóa Quá Trình Cập Nhật Metadata

Với Spring Data JPA, chúng ta có thể tự động hóa quá trình cập nhật metadata, giãm bớt việc làm thủ công. Do Spring Data JPA đã quản lý các truy vấn SQL (đọc, ghi, cập nhật, xóa), nên ta có thể gánh trách nhiệm cập nhật metadata cho framework này.

3. Các Bước Triển Khai Auditing

a. Thêm Annotation Trong Entity

Trong BaseEntity, chúng ta định nghĩa bốn cột metadata và sử dụng các annotation sau:

- @CreatedDate - cho cột createdAt
- @CreatedBy - cho cột createdBy
- @LastModifiedDate - cho cột updatedAt
- @LastModifiedBy - cho cột updatedBy

b. Tạo AuditAwareImpl

Do Spring Data JPA không thể xác định người tạo/cập nhật một bản ghi, chúng ta tạo class AuditAwareImpl trong package audit, implement interface AuditorAware<String>. Ban đầu, ta trả về giá trị mặc định ACCOUNTS_MS. Về sau, khi tích hợp Spring Security, ta sẽ thay thế giá trị này bằng user đăng nhập hiện tại.

c. Kích Hoạt Auditing Trong Spring Boot

1. Thêm @EntityListeners(AuditingEntityListener.class) trong BaseEntity.
2. Trong Application class, thêm @EnableJpaAuditing(auditorAwareRef = "auditAwareImpl") để bật tính năng auditing.

4. Kiểm Tra Kết Quả

1. **Tạo một bản ghi mới** và xác minh createdBy được cập nhật.
2. **Cập nhật bản ghi** và xác minh updatedBy và updatedAt đã thay đổi.
3. **Kiểm tra exception handling**: Thửi gian trước và sau khi thay đổi phải không được null hoặc bị lỗi logic.

Tài Liệu Hóa REST APIs Trong AccountsMicroservice

1. Giới Thiệu

Trong quá trình phát triển AccountsMicroservice, việc tài liệu hóa các REST APIs là bước quan trọng nhằm chuẩn hóa và tối ưu hóa khả năng tích hợp với các ứng dụng khác. Việc tài liệu hóa giúp những bên liên quan như nhóm giao diện (UI), nhóm phát triển ứng dụng di động và bộ phận kiểm thử hiểu rõ về cách sử dụng API mà không cần những cuộc họp trao đổi tốn nhiều thời gian.

2. Lợi ích Của Việc Tài Liệu Hóa REST APIs

- **Tối ưu hóa quy trình tích hợp**: Giúp các nhóm phát triển khác nhắc lại cách sử dụng API mà không cần trao đổi trực tiếp.
- **Giảm thiểu sai sót**: Đảm bảo mọi người truy cập đều hiểu API một cách đồng nhất.
- **Nâng cao khả năng tối ưu và mở rộng**: Giúp đồng bộ tài liệu khi API thay đổi.

3. Chuẩn Hóa Tài Liệu Hóa Bằng OpenAPI Specification

OpenAPI Specification là tiêu chuẩn mở giúp định nghĩa cách tài liệu hóa các HTTP API. Việc sử dụng OpenAPI Specification mang lại nhiều lợi ích:

- **Khám phá nhanh chóng**: Cung cấp thông tin API cho các bên sử dụng.
- **Tự động hóa**: Hỗ trợ tạo mã khách hàng và mã máy chủ.

- **Tích hợp Swagger UI**: Tạo giao diện truy vấn API tự động.

4. Sử Dụng SpringDoc OpenAPI

SpringDoc OpenAPI là thư viện giúp tái sử dụng OpenAPI Specification dễ dàng hơn. Các bước thực hiện bao gồm:

1. **Thêm phụ thuộc Maven** vào pom.xml:

```
<dependency>
   <groupId>org.springdoc</groupId>
   <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
   <version>3.x.x</version>
</dependency>
```

2. Xác nhận phiên bản Spring Boot: Phiên bản 3.x.x hỗ trợ mới nhất, còn Spring Boot 2.x hoặc 1.x cần phiên bản OpenAPI cũ hơn.

3. Cập nhật Maven và tải xuống các dependency.

4. Chạy lại ứng dụng và truy cập localhost:8080/swagger-ui/index.html để xem tài liệu.

5. Hiển Thị Swagger UI

Sau khi hoàn tất các bước trên, Swagger UI sẽ tự động hiển thị các API trong AccountsMicroservice. Tại đây, ta có thể:

- Kiểm tra các API endpoints.
- Xem các tham số truyền và định dạng dữ liệu.
- Kiểm tra request và response theo các tiêu chuẩn.

Cải Tiến Tài Liệu REST API Bằng SpringDoc OpenAPI

Giới Thiệu

Trong quá trình phát triển và hoàn thiện hệ thống API, chúng tôi nhận thấy rằng việc cung cấp tài liệu chi tiết và dễ hiểu là rất quan trọng để hỗ trợ người dùng hiểu và tận dụng được API. Do đó, chúng tôi quyết định sử dụng **SpringDoc OpenAPI** để tối ưu hóa tài liệu của REST API trong **Accounts Microservice**.

Mục Tiêu Cải Tiến

Trước khi thực hiện cải tiến, phần tài liệu API chưa cung cấp đủ thông tin quan trọng về REST API, bao gồm:

- Mô tả tóm tắt và chi tiết về API
- Liên hệ hỗ trợ
- Thông tin giấy phép
- Tài liệu tham khảo bên ngoài

Chúng tôi đã quyết định sử dụng các annotation trong **SpringDoc OpenAPI** để bổ sung các thông tin này và cải thiện tài liệu API một cách hệ thống.

Triển Khai Cải Tiến

1. Thêm Thông Tin Tài Liệu API

Chúng tôi đã chỉnh sửa tập tin **AccountsApplication.java** và sử dụng annotation @OpenAPIDefinition để cung cấp thông tin về API. Các thông tin đã được bổ sung:

- **Tiêu đề**: Accounts Microservice REST API Documentation
- **Mô tả**: Mô tả chi tiết về các API
- **Phiên bản**: v1
- **Liên hệ**:

    - Tên: EasyBank Support Team
    - Email: support@easybank.com
    - Website: eazybytes.com
- **Giấy phép**: Apache 2.0
- **Tài liệu bên ngoài**: URL tham khảo thêm về API

2. Tối ưu Giao Diện Swagger UI

Sau khi hoàn tất chỉnh sửa, chúng tôi đã tiến hành build và cập nhật Swagger UI. Kết quả nhận được:

- Phần đầu trang Swagger UI hiển thị đầy đủ các thông tin API.
- Người dùng có thể nhanh chóng nhắc tới contact khi cần hỗ trợ.
- Thông tin licensing rõ ràng, hỗ trợ việc quản lý quyền truy cập API.
- Liên kết tài liệu tham khảo giúp người dùng hiểu rõ hơn về hệ thống.

Tài Liệu Hóa REST API

Tài Liệu Hóa REST API

Giới Thiệu

Trong bài giảng này, chúng ta sẽ tăng cường việc tài liệu hóa REST API bằng cách cung cấp mô tả chi tiết về các API trong Swagger UI. Hiện tại, AccountsController chỉ có tên kỹ thuật và không cung cấp thông tin về các API.

Thực Hiện

1. Cải thiện mô tả cho AccountsController

- Truy cập lớp AccountsController.
- Sử dụng annotation @Tag từ Swagger OpenAPI để cung cấp tên và tóm tắt.
- Mã:

```
@Tag(name = "CRUD REST APIs for Accounts in EasyBank",
    description = "CRUD REST APIs in EasyBank to create, update, fetch, and delete account details.")
```

- Xây dựng lại dự án và kiểm tra Swagger UI. Các API hiện đã hiển thị với mô tả rõ ràng.

2. Cải thiện tài liệu các API cụ thể

- Đối với mỗi API, sử dụng annotation @Operation để cập nhật tóm tắt và mô tả chi tiết.
- Mã:

```
@Operation(summary = "Create Account REST API",
        description = "REST API to create a new Customer and Account in EasyBank.")
```

- Bổ sung @ApiResponse để hiển thị mã trả về:

```
@ApiResponse(responseCode = "201", description = "HttpStatus.CREATED")
```

- Xây dựng lại và kiểm tra Swagger UI. API hiển thị mã trả về chính xác.

3. Cải thiện API Response

- Đối với các API trả về nhiều trạng thái, sử dụng @ApiResponses:

```
@ApiResponses({
    @ApiResponse(responseCode = "200", description = "Success"),
    @ApiResponse(responseCode = "500", description = "Internal Server Error")
})
```

- Kiểm tra Swagger UI, các API hiển thị nhiều trạng thái khác nhau.

4. Cải thiện Schema Objects

- Cập nhật example data trong Swagger để người dùng hiểu rõ hơn.
- Hiện tại, Swagger UI hiển thị giá trị "String" thay vì tên rõ ràng.
- Trong lớp DTO, sử dụng annotation @Schema:

```
@Schema(description = "Name of the Account Holder", example = "John Doe")
private String accountHolderName;
```

- Xây dựng và kiểm tra Swagger UI, example data hiển thị rõ ràng.

API Bằng OpenAPI Schema

1. Giới Thiệu

Trong bài giảng này, chúng ta sẽ tối ưu và cải thiện tài liệu liên quan đến các đối tượng schema trong API. Việc sử dụng các annotation (@Schema) sẽ giúp tài liệu API trực quan hơn và dễ hiểu hơn.

2. Cải Thiện DTO Classes

Tương tự như khi chúng ta thêm annotation vào controller class và các method trong Java, chúng ta cũng cần làm điều tương tự cho các Data Transfer Object (DTO) classes.

2.1 CustomerDto

- Sửa tên kỹ thuật của schema thành tên thân thiện hơn, sử dụng annotation @Schema(name = "Customer").

- Thêm description cho schema: "Schema này chứa thông tin khách hàng và tài khoản".
- Xác định thông tin cụ thể cho các trường dữ liệu:
    - name: "Tên khách hàng", ví dụ "Eazy Byte".
    - email: "Địa chỉ email khách hàng", ví dụ "customer@example.com".
    - mobileNumber: "Số di động khách hàng", ví dụ "1234567890".
    - accounts: "Chi tiết tài khoản khách hàng".

2.2 AccountsDto

- Sử dụng annotation @Schema(name = "Accounts").
- Cung cấp thông tin chi tiết:
    - accountNumber: "Số tài khoản Easy Bank", regex bắt buộc.
    - accountType: "Loại tài khoản", ví dụ "Savings".
    - branchAddress: "Địa chỉ chi nhánh", ví dụ "123 New York".

2.3 ResponseDto

- @Schema(name = "Response").
- statusCode: "Mã trạng thái trong response", ví dụ "200".
- statusMessage: "Mô tả trạng thái", ví dụ "Request processed successfully".

2.4 ErrorResponseDto

- @Schema(name = "ErrorResponse").
- errorCode: "Mã lỗi xảy ra".
- errorMessage: "Mô tả chi tiết về lỗi".

3. Cải Thiện Tài Liệu API Trên Swagger UI

- Sau khi thêm các annotation, Swagger UI sẽ hiển thị:
    - Tên schema rõ ràng hơn.
    - Các trường dữ liệu được diễn giải cụ thể.
    - Example data hữu ích cho người dùng.

4. Khắc Phục Vấn Đề Mất Schema ErrorResponseDto

- OpenAPI không tự động quét GlobalExceptionLogic.
- Cần chỉ định ErrorResponseDto trong @ApiResponse:

```
@ApiResponse(
   responseCode = "500",
   description = "Internal Server Error",
   content = @Content(schema = @Schema(implementation =
ErrorResponseDto.class))
)
```

Xử Lý Lỗi API và OpenAPI Schema

1. Giới Thiệu

Trong bài giảng này, chúng ta đã tiến hành tối ưu hóa cách xử lý lỗi trong API bằng cách thay đổi mã trạng thái HTTP khi xảy ra lỗi. Trước đây, mã lỗi 500 (Internal Server Error) được sử dụng cho cả lỗi RuntimeException, lỗi cập nhật và lỗi xóa, điều này gây nhầm lẫn cho client. Do đó, ta đã thực hiện cập nhật để sử dụng mã lỗi phù hợp hơn.

2. Thay Đổi Trong GlobalExceptionHandler

GlobalExceptionHandler vẫn giữ nguyên việc xử lý lỗi RuntimeException với mã lỗi 500. Tuy nhiên, trong controller class, chúng ta đã thực hiện điều chỉnh cho các lỗi cập nhật và xóa:

- Khi thao tác cập nhật thất bại, sử dụng mã trạng thái 417 Expectation Failed.
- Khi thao tác xóa thất bại, sử dụng mã trạng thái 417 Expectation Failed.

Chúng ta cũng đã cập nhật ResponseDto để phản ánh các mã lỗi mới này, đồng thời thêm thông báo lỗi cụ thể:

- updateOperationFailed: "Cập nhật thất bại. Vui lòng thử lại hoặc liên hệ đội ngũ phát triển."
- deleteOperationFailed: "Xóa thất bại. Vui lòng thử lại hoặc liên hệ đội ngũ phát triển."

Với những thay đổi này, chúng ta không còn cần đến các hằng số lỗi 500 cũ, và chúng đã được đưa vào phần comment để tham khảo khi cần thiết.

3. Cải Thiện API Response

Sau khi thực hiện các thay đổi, tài liệu API hiện tại sẽ có các phản hồi như sau:

- **API cập nhật**:
    - 204 (No Content) khi cập nhật thành công.
    - 417 (Expectation Failed) khi cập nhật thất bại.
    - 500 (Internal Server Error) khi xảy ra RuntimeException.
- **API xóa**:
    - 204 (No Content) khi xóa thành công.
    - 417 (Expectation Failed) khi xóa thất bại.
    - 500 (Internal Server Error) khi xảy ra RuntimeException.

Bên cạnh đó, chúng ta cũng đã đảm bảo rằng ErrorResponseDto được định nghĩa rõ ràng trong tài liệu API bằng cách sử dụng annotation @Schema trong OpenAPI.

4. Khắc Phục Hiển Thị Trạng Thái Trong Swagger UI

Sau khi thực hiện thay đổi, chúng ta phát hiện rằng Swagger UI hiển thị sai mã trạng thái cho 204 và 417, do chúng được đặt mặc định là 200 trong ResponseDto. Để khắc phục:

- Đã loại bỏ giá trị mặc định example = "200" trong ResponseDto.
- Điều chỉnh tài liệu API để phản ánh đúng mã trạng thái.

Sau khi build lại và kiểm tra trên Swagger UI, các API hiển thị đúng trạng thái tương ứng với từng tình huống lỗi và thành công.

Accounts Microservice và các Annotation trong Spring Boot

1. Giới thiệu

Trong quá trình phát triển hệ thống, chúng tôi đã hoàn thành việc xây dựng **Accounts Microservice**. Microservice này bao gồm **4 loại REST API** khác nhau được triển khai trong cùng một tiến trình.

2. Sử dụng Annotation trong Spring Boot

Trong quá trình phát triển, chúng tôi đã tận dụng nhiều annotation của **Spring Boot Framework** để xây dựng REST API. Dưới đây là danh sách các annotation quan trọng và cách sử dụng của chúng:

2.1 @RestController

- **Chức năng**: Được sử dụng để đánh dấu một lớp Java như một REST Controller, giúp Spring Boot biết rằng các phương thức trong lớp này cần được expose thành REST API.
- **Cách sử dụng**: Khi xây dựng REST API bằng Java và sử dụng các annotation như @GetMapping, @PostMapping, cần đảm bảo khai báo @RestController trên đầu lớp.

- **So sánh với @Controller**: Nếu chỉ sử dụng @Controller, chúng ta cần khai báo thêm @ResponseBody trên từng phương thức để đảm bảo dữ liệu trả về ở định dạng JSON thay vì HTML.

2.2 @ResponseBody

- **Chức năng**: Chỉ định rằng dữ liệu trả về của phương thức sẽ được tự động chuyển đổi thành JSON.
- **Khi nào sử dụng**: Khi kết hợp với @Controller thay vì @RestController, cần thêm @ResponseBody để xác định dữ liệu phản hồi dưới dạng JSON.

2.3 ResponseEntity

- **Chức năng**: Một lớp trong Spring giúp chúng ta có thể gửi phản hồi HTTP với **status code, header và body**.
- **Cách sử dụng**: Khi cần gửi dữ liệu phản hồi có cấu trúc rõ ràng, bao gồm HTTP status code, header và nội dung body.

2.4 @ControllerAdvice và @ExceptionHandler

- **@ControllerAdvice**: Dùng để xử lý ngoại lệ toàn cục trong ứng dụng.
- **@ExceptionHandler**: Được sử dụng để xử lý các loại ngoại lệ cụ thể xảy ra trong ứng dụng.
- **So sánh với @RestControllerAdvice**: @RestControllerAdvice là sự kết hợp của @ControllerAdvice và @ResponseBody, đảm bảo rằng tất cả phản hồi lỗi đều được trả về dưới dạng JSON.

2.5 RequestEntity

- **Chức năng**: Tương tự ResponseEntity, nhưng dùng để xử lý **cả request body và request header** khi nhận yêu cầu từ client.

- **Khi nào sử dụng**: Khi cần xử lý đồng thời cả **request body** và **request header** từ client.

2.6 @RequestHeader và @RequestBody

- **@RequestHeader**: Được sử dụng để lấy thông tin từ HTTP header trong request.
- **@RequestBody**: Được sử dụng để nhận dữ liệu JSON từ request body và ánh xạ vào một đối tượng Java.

2.7 @RequestParam và @PathVariable

- **@RequestParam**: Được sử dụng để lấy tham số từ query string của URL.
- **@PathVariable**: Được sử dụng để lấy tham số từ đường dẫn của URL.

3. Ứng dụng thực tế

Những annotation trên rất quan trọng khi làm việc với Spring Boot để xây dựng REST API. Trong các buổi phỏng vấn, có thể sẽ có câu hỏi về cách xây dựng REST API trong Spring Boot, vì vậy việc nắm vững và hiểu cách sử dụng các annotation này sẽ giúp ứng viên có câu trả lời ấn tượng hơn.

Quy trình triển khai REST API:

1. Tạo một lớp Java và đánh dấu bằng @RestController.
2. Xây dựng các phương thức API với các annotation như @GetMapping, @PostMapping.
3. Sử dụng RequestEntity, RequestHeader, RequestBody khi cần thiết.
4. Xử lý ngoại lệ toàn cục với @ControllerAdvice và @ExceptionHandler.
5. Đảm bảo phản hồi HTTP có cấu trúc tốt bằng cách sử dụng ResponseEntity.
6. Sử dụng @RequestParam và @PathVariable để xử lý tham số từ URL.